A Partition Theorem for the Ordinal ω^ω

Lawrence C. Paulson

March 24, 2023

Abstract

The theory of partition relations concerns generalisations of Ramsey's theorem. For any ordinal α, write $\alpha \to (\alpha, m)^2$ if for each function f from unordered pairs of elements of α into $\{0, 1\}$, either there is a subset $X \subseteq \alpha$ order-isomorphic to α such that $f\{x, y\} = 0$ for all $\{x, y\} \subseteq X$, or there is an m element set $Y \subseteq \alpha$ such that $f\{x, y\} = 1$ for all $\{x, y\} \subseteq Y$. (In both cases, with $\{x, y\}$ we require $x \neq y$.) In particular, the infinite Ramsey theorem can be written in this notation as $\omega \to (\omega, \omega)^2$, or if we restrict m to the positive integers as above, then $\omega \to (\omega, m)^2$ for all m [3].

This entry formalises Larson's proof of $\omega^\omega \to (\omega^\omega, m)^2$ along with a similar proof of a result due to Specker: $\omega^2 \to (\omega^2, m)^2$. Also proved is a necessary result by Erdős and Milner [1, 2]: $\omega^{1+\alpha\cdot n} \to (\omega^{1+\alpha}, 2^n)^2$.

These examples demonstrate the use of Isabelle/HOL to formalise advanced results that combine ZF set theory with basic concepts like lists and natural numbers.

Contents

1 Library additions

theory *Library-Additions*
imports *ZFC-in-HOL.Ordinal-Exp HOL−Library.Ramsey Nash-Williams.Nash-Williams*

begin

lemma *finite-enumerate-Diff-singleton*:
fixes S :: *'a::wellorder set*
assumes *finite* S **and** i: $i < card\ S$ *enumerate* $S\ i < x$
shows *enumerate* $(S - \{x\})\ i$ = *enumerate* $S\ i$
using i
proof (*induction* i)
case *0*
have (*LEAST i. i* ∈ $S \wedge i \neq x$) = (*LEAST i. i* ∈ S)
proof (*rule Least-equality*)
have ∃ *t*. $t \in S \wedge t \neq x$
using *0* ‹*finite S*› *finite-enumerate-in-set* **by** *blast*
then show (*LEAST i. i* ∈ S) ∈ $S \wedge$ (*LEAST i. i* ∈ S) ≠ x
by (*metis 0.prems*(*2*) *LeastI enumerate-0 not-less-Least*)
qed (*simp add*: *Least-le*)
then show *?case*
by (*auto simp*: *enumerate-0*)

next
case (*Suc i*)
then have *x*: *enumerate S i* < *x*
by (*meson enumerate-step finite-enumerate-step less-trans*)
have *cardSx*: *Suc i* < *card* (*S* − {*x*}) **and** *i* < *card S*
using *Suc* ‹*finite S*› *card-Diff-singleton-if*[*of S*] *finite-enumerate-Ex* **by** *fastforce*+
have (*LEAST s*. *s* ∈ *S* ∧ *s*≠*x* ∧ *enumerate* (*S* − {*x*}) *i* < *s*) = (*LEAST s*. *s* ∈ *S* ∧ *enumerate S i* < *s*)
(**is** - = *?r*)
proof (*intro Least-equality conjI*)
show *?r* ∈ *S*
by (*metis* (*lifting*) *LeastI Suc.prems*(*1*) *assms*(*1*) *finite-enumerate-in-set finite-enumerate-step*)
show *?r* ≠ *x*
using *Suc.prems not-less-Least* [*of* - λ*t*. *t* ∈ *S* ∧ *enumerate S i* < *t*]
‹*finite S*› *finite-enumerate-in-set finite-enumerate-step* **by** *blast*
show *enumerate* (*S* − {*x*}) *i* < *?r*
by (*metis* (*full-types*) *Suc.IH Suc.prems*(*1*) ‹*i* < *card S*› *enumerate-Suc″ enumerate-step finite-enumerate-Suc″ finite-enumerate-step x*)
show ⋀*y*. *y* ∈ *S* ∧ *y* ≠ *x* ∧ *enumerate* (*S* − {*x*}) *i* < *y* ⟹ *?r* ≤ *y*
by (*simp add*: *Least-le Suc.IH* ‹*i* < *card S*› *x*)
qed
then show *?case*
using *Suc assms* **by** (*simp add*: *finite-enumerate-Suc″ cardSx*)
qed

lemma *hd-lex*: ⟦*hd ms* < *hd ns*; *length ms* = *length ns*; *ns* ≠ []⟧ ⟹ (*ms*, *ns*) ∈ *lex less-than*
by (*metis hd-Cons-tl length-0-conv less-than-iff lexord-cons-cons lexord-lex*)

lemma *sorted-hd-le*:
assumes *sorted xs x* ∈ *list.set xs*
shows *hd xs* ≤ *x*
using *assms* **by** (*induction xs*) (*auto simp*: *less-imp-le*)

lemma *sorted-le-last*:
assumes *sorted xs x* ∈ *list.set xs*
shows *x* ≤ *last xs*
using *assms* **by** (*induction xs*) (*auto simp*: *less-imp-le*)

lemma *hd-list-of*:
assumes *finite A A* ≠ {}
shows *hd* (*sorted-list-of-set A*) = *Min A*
proof (*rule antisym*)
have *Min A* ∈ *A*
by (*simp add*: *assms*)
then show *hd* (*sorted-list-of-set A*) ≤ *Min A*
by (*simp add*: *sorted-hd-le* ‹*finite A*›)

next
show *Min A* ≤ *hd* (*sorted-list-of-set A*)
by (*metis Min-le assms hd-in-set set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff*)
qed

lemma *sorted-hd-le-last*:
assumes *sorted xs xs* ≠ []
shows *hd xs* ≤ *last xs*
using *assms* **by** (*simp add*: *sorted-hd-le*)

lemma *sorted-list-of-set-set-of* [*simp*]: *strict-sorted l* ⟹ *sorted-list-of-set* (*list.set l*) = *l*
by (*simp add*: *strict-sorted-equal*)

lemma *range-strict-mono-ext*:
fixes *f*::*nat* ⇒ *'a*::*linorder*
assumes *eq*: *range f* = *range g*
and *sm*: *strict-mono f strict-mono g*
shows *f* = *g*
proof
fix *n*
show *f n* = *g n*
proof (*induction n rule*: *less-induct*)
case (*less n*)
obtain *x y* **where** *xy*: *f n* = *g y f x* = *g n*
by (*metis eq imageE rangeI*)
then have *n* = *y*
by (*metis* (*no-types*) *less.IH neq-iff sm strict-mono-less xy*)
then show *?case* **using** *xy* **by** *auto*
qed
qed

1.1 Other material

definition *strict-mono-sets* :: [*'a*::*order set*, *'a*::*order* ⇒ *'b*::*order set*] ⇒ *bool* **where**
strict-mono-sets A f ≡ ∀ *x*∈*A*. ∀ *y*∈*A*. *x* < *y* ⟶ *less-sets* (*f x*) (*f y*)

lemma *strict-mono-setsD*:
assumes *strict-mono-sets A f x* < *y x* ∈ *A y* ∈ *A*
shows *less-sets* (*f x*) (*f y*)
using *assms* **by** (*auto simp*: *strict-mono-sets-def*)

lemma *strict-mono-sets-imp-disjoint*:
fixes *A* :: *'a*::*linorder set*
assumes *strict-mono-sets A f*
shows *pairwise* (λ*x y*. *disjnt* (*f x*) (*f y*)) *A*
using *assms* **unfolding** *strict-mono-sets-def pairwise-def*
by (*meson antisym-conv3 disjnt-sym less-sets-imp-disjnt*)

lemma *strict-mono-sets-subset*:
assumes *strict-mono-sets B f A* $\subseteq$ *B*
shows *strict-mono-sets A f*
using *assms* **by** (*auto simp*: *strict-mono-sets-def*)

lemma *strict-mono-less-sets-Min*:
assumes *strict-mono-sets I f finite I I* $\neq$ {}
shows *less-sets* (*f* (*Min I*)) ($\bigcup$ (*f* ' (*I* − {*Min I*}))))
using *assms* **by** (*simp add*: *strict-mono-sets-def less-sets-UN2 dual-order.strict-iff-order*)

lemma *pair-less-iff1* [*simp*]: ((*x*,*y*), (*x*,*z*)) $\in$ *pair-less* $\longleftrightarrow$ *y*<*z*
by (*simp add*: *pair-less-def*)

lemma *infinite-finite-Inter*:
assumes *finite* $\mathcal{A}$ $\mathcal{A}\neq\{\}$ $\bigwedge A.\ A \in \mathcal{A} \Longrightarrow$ *infinite A*
and $\bigwedge A\ B.\ [\![A \in \mathcal{A};\ B \in \mathcal{A}]\!] \Longrightarrow A \cap B \in \mathcal{A}$
shows *infinite* ($\bigcap \mathcal{A}$)
by (*simp add*: *assms finite-Inf-in*)

lemma *atLeast-less-sets*: $[\![$*less-sets A* {*x*}; *B* $\subseteq$ {*x*..}$]\!] \Longrightarrow$ *less-sets A B*
by (*force simp*: *less-sets-def subset-iff*)

1.2 The list-of function

lemma *sorted-list-of-set-insert-remove-cons*:
assumes *finite A less-sets* {*a*} *A*
shows *sorted-list-of-set* (*insert a A*) = *a* # *sorted-list-of-set A*
proof −
have *strict-sorted* (*a* # *sorted-list-of-set A*)
using *assms less-setsD* **by** *auto*
moreover have *list.set* (*a* # *sorted-list-of-set A*) = *insert a A*
using *assms* **by** *force*
moreover have *length* (*a* # *sorted-list-of-set A*) = *card* (*insert a A*)
using *assms card-insert-if less-setsD* **by** *fastforce*
ultimately show *?thesis*
by (*metis* ‹*finite A*› *finite-insert sorted-list-of-set-unique*)
qed

lemma *sorted-list-of-set-Un*:
assumes *AB*: *less-sets A B* **and** *fin*: *finite A finite B*
shows *sorted-list-of-set* (*A* ∪ *B*) = *sorted-list-of-set A* @ *sorted-list-of-set B*
proof −
have *strict-sorted* (*sorted-list-of-set A* @ *sorted-list-of-set B*)
using *AB* **unfolding** *less-sets-def*
by (*metis fin set-sorted-list-of-set sorted-wrt-append strict-sorted-list-of-set*)
moreover have *card A* + *card B* = *card* (*A* ∪ *B*)
using *less-sets-imp-disjnt* [*OF AB*]
by (*simp add*: *assms card-Un-disjoint disjnt-def*)
ultimately show *?thesis*

by (*simp add*: *assms strict-sorted-equal*)
qed

lemma *sorted-list-of-set-UN-lessThan*:
fixes *k*::*nat*
assumes *sm*: *strict-mono-sets* {..<*k*} *A* **and** ⋀*i*. *i* < *k* ⟹ *finite* (*A i*)
shows *sorted-list-of-set* (⋃*i*<*k*. *A i*) = *concat* (*map* (*sorted-list-of-set* ∘ *A*) (*sorted-list-of-set* {..<*k*}))
using *assms*
proof (*induction k*)
case *0*
then show *?case*
by *auto*
next
case (*Suc k*)
have *ls*: *less-sets* (⋃ (*A* ' {..<*k*})) (*A k*)
using *sm Suc.prems*(*1*) *strict-mono-setsD* **by** (*force simp*: *less-sets-UN1*)
have *sorted-list-of-set* (⋃ (*A* ' {..<*Suc k*})) = *sorted-list-of-set* (⋃ (*A* ' {..<*k*}) ∪ *A k*)
by (*simp add*: *Un-commute lessThan-Suc*)
also have ... = *sorted-list-of-set* (⋃ (*A* ' {..<*k*})) @ *sorted-list-of-set* (*A k*)
by (*rule sorted-list-of-set-Un*) (*auto simp*: *Suc.prems ls*)
also have ... = *concat* (*map* (*sorted-list-of-set* ∘ *A*) (*sorted-list-of-set* {..<*k*})) @ *sorted-list-of-set* (*A k*)
using *Suc strict-mono-sets-def* **by** *fastforce*
also have ... = *concat* (*map* (*sorted-list-of-set* ∘ *A*) (*sorted-list-of-set* {..<*Suc k*}))
using *strict-mono-sets-def* **by** *fastforce*
finally show *?case* **.**
qed

lemma *sorted-list-of-set-UN-atMost*:
fixes *k*::*nat*
assumes *strict-mono-sets* {..*k*} *A* **and** ⋀*i*. *i* ≤ *k* ⟹ *finite* (*A i*)
shows *sorted-list-of-set* (⋃*i*≤*k*. *A i*) = *concat* (*map* (*sorted-list-of-set* ∘ *A*) (*sorted-list-of-set* {..*k*}))
by (*metis assms lessThan-Suc-atMost less-Suc-eq-le sorted-list-of-set-UN-lessThan*)

1.3 Monotonic enumeration of a countably infinite set

abbreviation *enum* ≡ *enumerate*

Could be generalised to infinite countable sets of any type

lemma *nat-infinite-iff*:
fixes *N* :: *nat set*
shows *infinite N* ⟷ (∃*f*::*nat*⇒*nat*. *N* = *range f* ∧ *strict-mono f*)
proof *safe*
assume *infinite N*
then show ∃*f*. *N* = *range* (*f*::*nat* ⇒ *nat*) ∧ *strict-mono f*
by (*metis bij-betw-imp-surj-on bij-enumerate enumerate-mono strict-mono-def*)

next
fix $f :: nat \Rightarrow nat$
assume *strict-mono* f **and** $N = range\ f$ **and** *finite* $(range\ f)$
then show *False*
using *range-inj-infinite strict-mono-imp-inj-on* **by** *blast*
qed

lemma *enum-works*:
fixes $N :: nat\ set$
assumes *infinite* N
shows $N = range\ (enum\ N) \wedge strict\text{-}mono\ (enum\ N)$
by (*metis assms bij-betw-imp-surj-on bij-enumerate enumerate-mono strict-monoI*)

lemma *range-enum*: $range\ (enum\ N) = N$ **and** *strict-mono-enum*: *strict-mono* $(enum\ N)$
if *infinite* N **for** $N :: nat\ set$
using *enum-works* [*OF that*] **by** *auto*

lemma *enum-0-eq-Inf*:
fixes $N :: nat\ set$
assumes *infinite* N
shows $enum\ N\ 0 = Inf\ N$
proof −
have $enum\ N\ 0 \in N$
using *assms range-enum* **by** *auto*
moreover have $\bigwedge x.\ x \in N \Longrightarrow enum\ N\ 0 \leq x$
by (*metis* (*mono-tags*, *opaque-lifting*) *assms imageE le0 less-mono-imp-le-mono range-enum strict-monoD strict-mono-enum*)
ultimately show *?thesis*
by (*metis cInf-eq-minimum*)
qed

lemma *enum-works-finite*:
fixes $N :: nat\ set$
assumes *finite* N
shows $N = enum\ N\ `\ \{..<card\ N\} \wedge strict\text{-}mono\text{-}on\ \{..<card\ N\}\ (enum\ N)$
using *assms*
by (*metis bij-betw-imp-surj-on finite-bij-enumerate finite-enumerate-mono lessThan-iff strict-mono-onI*)

lemma *enum-obtain-index-finite*:
fixes $N :: nat\ set$
assumes $x \in N$ *finite* N
obtains i **where** $i < card\ N\ x = enum\ N\ i$
by (*metis* ‹$x \in N$› ‹*finite* N› *enum-works-finite imageE lessThan-iff*)

lemma *enum-0-eq-Inf-finite*:
fixes $N :: nat\ set$
assumes *finite* $N\ N \neq \{\}$

shows *enum N 0 = Inf N*
proof −
have *enum N 0 ∈ N*
by (*metis Nat.neq0-conv assms empty-is-image enum-works-finite image-eqI lessThan-empty-iff lessThan-iff*)
moreover have *enum N 0 ≤ x* **if** *x ∈ N* **for** *x*
proof −
obtain *i* **where** *i < card N x = enum N i*
by (*metis ‹x ∈ N› ‹finite N› enum-obtain-index-finite*)
with *assms* **show** *?thesis*
by (*metis Nat.neq0-conv finite-enumerate-mono less-or-eq-imp-le*)
qed
ultimately show *?thesis*
by (*metis cInf-eq-minimum*)
qed

lemma *greaterThan-less-enum*:
fixes *N* :: *nat set*
assumes *N ⊆ {x<..} infinite N*
shows *x < enum N i*
using *assms range-enum* **by** *fastforce*

lemma *atLeast-le-enum*:
fixes *N* :: *nat set*
assumes *N ⊆ {x..} infinite N*
shows *x ≤ enum N i*
using *assms range-enum* **by** *fastforce*

lemma *less-sets-empty1* [*simp*]: *less-sets {} A* **and** *less-sets-empty2* [*simp*]: *less-sets A {}*
by (*simp-all add*: *less-sets-def*)

lemma *less-sets-singleton1* [*simp*]: *less-sets {a} A ⟷ (∀ x∈A. a < x)*
and *less-sets-singleton2* [*simp*]: *less-sets A {a} ⟷ (∀ x∈A. x < a)*
by (*simp-all add*: *less-sets-def*)

lemma *less-sets-atMost* [*simp*]: *less-sets {..a} A ⟷ (∀ x∈A. a < x)*
and *less-sets-alLeast* [*simp*]: *less-sets A {a..} ⟷ (∀ x∈A. x < a)*
by (*auto simp*: *less-sets-def*)

lemma *less-sets-imp-strict-mono-sets*:
assumes *⋀i. less-sets (A i) (A (Suc i)) ⋀i. i>0 ⟹ A i ≠ {}*
shows *strict-mono-sets UNIV A*
proof (*clarsimp simp*: *strict-mono-sets-def*)
fix *i j*::*nat*
assume *i < j*
then show *less-sets (A i) (A j)*
proof (*induction j−i arbitrary*: *i j*)
case (*Suc x*)

then show *?case*
by (*metis Suc-diff-Suc Suc-inject Suc-mono assms less-Suc-eq less-sets-trans zero-less-Suc*)
qed *auto*
qed

lemma *less-sets-Suc-Max*:
assumes *finite A*
shows *less-sets A* {*Suc* (*Max A*)..}
proof (*cases A* = {})
case *False*
then show *?thesis*
by (*simp add*: *assms less-Suc-eq-le*)
qed *auto*

lemma *infinite-nat-greaterThan*:
fixes *m*::*nat*
assumes *infinite N*
shows *infinite* (*N* ∩ {*m*<..})
proof −
have *N* ⊆ −{*m*<..} ∪ (*N* ∩ {*m*<..})
by *blast*
moreover have *finite* (−{*m*<..})
by *simp*
ultimately show *?thesis*
using *assms finite-subset* **by** *blast*
qed

end

2 Ordinal Partitions

Material from Jean A. Larson, A short proof of a partition theorem for the ordinal ω^ω. *Annals of Mathematical Logic*, 6:129–145, 1973. Also from "Partition Relations" by A. Hajnal and J. A. Larson, in *Handbook of Set Theory*, edited by Matthew Foreman and Akihiro Kanamori (Springer, 2010).

theory *Partitions*
imports *Library-Additions ZFC-in-HOL.ZFC-Typeclasses ZFC-in-HOL.Cantor-NF*

begin

abbreviation *tp* :: *V set* ⇒ *V*
where *tp A* ≡ *ordertype A VWF*

2.1 Ordinal Partitions: Definitions

definition *partn-lst* :: [('*a* × '*a*) *set*, '*a set*, *V list*, *nat*] ⇒ *bool*
where *partn-lst r B* α *n* ≡ ∀ *f* ∈ [*B*]^n → {..<*length* α}.

$\exists\, i < \mathit{length}\ \alpha.\ \exists\, H.\ H \subseteq B \wedge \mathit{ordertype}\ H\ r = (\alpha!i) \wedge f\ `\ (\mathit{nsets}\ H\ n) \subseteq \{i\}$

abbreviation *partn-lst-VWF* :: $V \Rightarrow V\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
  where $\mathit{partn\text{-}lst\text{-}VWF}\ \beta \equiv \mathit{partn\text{-}lst}\ \mathit{VWF}\ (\mathit{elts}\ \beta)$

lemma *partn-lst-E*:
  assumes $\mathit{partn\text{-}lst}\ r\ B\ \alpha\ n\ f \in \mathit{nsets}\ B\ n \to \{..<l\}\ l = \mathit{length}\ \alpha$
  obtains $i\ H$ **where** $i < l\ H \subseteq B$
    $\mathit{ordertype}\ H\ r = \alpha!i\ f\ `\ (\mathit{nsets}\ H\ n) \subseteq \{i\}$
  using *assms* **by** (*auto simp*: *partn-lst-def*)

lemma *partn-lst-VWF-nontriv*:
  assumes $\mathit{partn\text{-}lst\text{-}VWF}\ \beta\ \alpha\ n\ l = \mathit{length}\ \alpha\ \mathit{Ord}\ \beta\ l > 0$
  obtains i **where** $i < l\ \alpha!i \leq \beta$
proof −
  have $\{..<l\} \neq \{\}$
    by (*simp add*: ‹$l > 0$› *lessThan-empty-iff*)
  then obtain f **where** $f \in \mathit{nsets}\ (\mathit{elts}\ \beta)\ n \to \{..<l\}$
    by (*meson Pi-eq-empty equals0I*)
  then obtain $i\ H$ **where** $i < l\ H \subseteq \mathit{elts}\ \beta$ **and** *eq*: $\mathit{tp}\ H = \alpha!i$
    using *assms* **by** (*metis partn-lst-E*)
  then have $\alpha!i \leq \beta$
    by (*metis* ‹$H \subseteq \mathit{elts}\ \beta$› ‹$\mathit{Ord}\ \beta$› *eq ordertype-le-Ord*)
  then show *thesis*
    using ‹$i < l$› *that* **by** *auto*
qed

lemma *partn-lst-triv0*:
  assumes $\alpha!i = 0\ i < \mathit{length}\ \alpha\ n \neq 0$
  shows $\mathit{partn\text{-}lst}\ r\ B\ \alpha\ n$
 by (*metis partn-lst-def assms bot-least image-empty nsets-empty-iff ordertype-empty*)

lemma *partn-lst-triv1*:
  assumes $\alpha!i \leq 1\ i < \mathit{length}\ \alpha\ n > 1\ B \neq \{\}\ \mathit{wf}\ r$
  shows $\mathit{partn\text{-}lst}\ r\ B\ \alpha\ n$
  unfolding *partn-lst-def*
proof *clarsimp*
  obtain γ **where** $\gamma \in B\ \alpha \neq []$
    using *assms mem-0-Ord* **by** *fastforce*
  have *01*: $\alpha!i = 0 \vee \alpha!i = 1$
    using *assms* **by** (*fastforce simp*: *one-V-def*)
  fix f
  assume f: $f \in [B]^n \to \{..<\mathit{length}\ \alpha\}$
  with *assms* **have** $\mathit{ordertype}\ \{\gamma\}\ r = 1 \wedge f\ `\ [\{\gamma\}]^n \subseteq \{i\}$
      $\mathit{ordertype}\ \{\}\ r = 0 \wedge f\ `\ [\{\}]^n \subseteq \{i\}$
    by (*auto simp*: *one-V-def ordertype-insert nsets-eq-empty*)
  with *assms 01* **show** $\exists\, i{<}\mathit{length}\ \alpha.\ \exists\, H{\subseteq}B.\ \mathit{ordertype}\ H\ r = \alpha\ !\ i \wedge f\ `\ [H]^n \subseteq \{i\}$

using ‹$\gamma \in B$› **by** *auto*
qed

lemma *partn-lst-two-swap*:
assumes *partn-lst r B* [*x*,*y*] *n* **shows** *partn-lst r B* [*y*,*x*] *n*
proof −
{ fix *f* :: *'a set* ⇒ *nat*
assume *f*: $f \in [B]^n \to \{..<2\}$
then have *f'*: $(\lambda i.\ 1 - i) \circ f \in [B]^n \to \{..<2\}$
by (*auto simp*: *Pi-def*)
obtain *i H* **where** $i<2$ $H \subseteq B$ *ordertype H r* = ([*x*,*y*]!*i*) ((λ*i*. 1 − *i*) ∘ *f*) ‘ (*nsets H n*) ⊆ {*i*}
by (*auto intro*: *partn-lst-E* [*OF assms f'*])
moreover have *f x* = *Suc 0* **if** *Suc 0* ≤ *f x* $x \in [H]^n$ **for** *x*
using *f that* ‹$H \subseteq B$› *nsets-mono* **by** (*fastforce simp*: *Pi-iff*)
ultimately have *ordertype H r* = [*y*,*x*] ! (1−*i*) ∧ *f* ‘ $[H]^n \subseteq \{1-i\}$
by (*force simp*: *eval-nat-numeral less-Suc-eq*)
then have ∃ *i H*. $i<2 \wedge H \subseteq B \wedge$ *ordertype H r* = [*y*,*x*] ! *i* ∧ *f* ‘ $[H]^n \subseteq \{i\}$
by (*metis Suc-1* ‹$H \subseteq B$› *diff-less-Suc*) **}**
then show *?thesis*
by (*auto simp*: *partn-lst-def eval-nat-numeral*)
qed

lemma *partn-lst-greater-resource*:
assumes *M*: *partn-lst r B α n* **and** $B \subseteq C$
shows *partn-lst r C α n*
proof (*clarsimp simp*: *partn-lst-def*)
fix *f*
assume $f \in [C]^n \to \{..<$*length α*$\}$
then have $f \in [B]^n \to \{..<$*length α*$\}$
by (*metis* ‹$B \subseteq C$› *part-fn-def part-fn-subset*)
then obtain *i H* **where** *i* < *length α*
and $H \subseteq B$ *ordertype H r* = (*α*!*i*)
and *f* ‘ *nsets H n* ⊆ {*i*}
using *M partn-lst-def* **by** *metis*
then show ∃ *i*<*length α*. ∃ *H*⊆*C*. *ordertype H r* = *α* ! *i* ∧ *f* ‘ $[H]^n \subseteq \{i\}$
using ‹$B \subseteq C$› **by** *blast*
qed

lemma *partn-lst-less*:
assumes *M*: *partn-lst r B α n* **and** *eq*: *length α'* = *length α* **and** *List.set α'* ⊆ *ON*
and *le*: ⋀*i*. *i* < *length α* ⟹ *α'*!*i* ≤ *α*!*i*
and *r*: *wf r trans r total-on B r* **and** *small B*
shows *partn-lst r B α' n*
proof (*clarsimp simp*: *partn-lst-def*)
fix *f*
assume $f \in [B]^n \to \{..<$*length α'*$\}$

```
  then obtain i H where i < length α
                    and H ⊆ B small H and H: ordertype H r = (α!i)
                    and fi: f ‘ nsets H n ⊆ {i}
    using assms by (auto simp: partn-lst-def smaller-than-small)
  then have bij: bij-betw (ordermap H r) H (elts (α!i))
    using ordermap-bij [of r H] r
    by (smt ‹small B› in-mono smaller-than-small total-on-def)
  define H' where H' = inv-into H (ordermap H r) ‘ (elts (α'!i))
  have H' ⊆ H
    using bij ‹i < length α› bij-betw-imp-surj-on le
    by (force simp: H'-def image-subset-iff intro: inv-into-into)
  moreover have ot: ordertype H' r = (α'!i)
  proof (subst ordertype-eq-iff)
    show Ord (α' ! i)
      using assms by (simp add: ‹i < length α› subset-eq)
    show small H'
      by (simp add: H'-def)
    show ∃f. bij-betw f H' (elts (α' ! i)) ∧ (∀x∈H'. ∀y∈H'. (f x < f y) = ((x, y)
∈ r))
    proof (intro exI conjI ballI)
      show bij-betw (ordermap H r) H' (elts (α' ! i))
        using ‹H' ⊆ H›
        by (metis H'-def ‹i < length α› bij bij-betw-inv-into-RIGHT bij-betw-subset
le less-eq-V-def)
      show (ordermap H r x < ordermap H r y) = ((x, y) ∈ r)
        if x ∈ H' y ∈ H' for x y
      proof (intro iffI ordermap-mono-less)
        assume ordermap H r x < ordermap H r y
        then show (x, y) ∈ r
          by (metis ‹H ⊆ B› ‹small H› ‹H' ⊆ H› leD ordermap-mono-le r subsetD
that total-on-def)
      qed (use assms that ‹H' ⊆ H› ‹small H› in auto)
    qed
    show total-on H' r
      using r by (meson ‹H ⊆ B› ‹H' ⊆ H› subsetD total-on-def)
  qed (use r in auto)
  ultimately show ∃i<length α'. ∃H⊆B. ordertype H r = α' ! i ∧ f ‘ [H]^n ⊆ {i}
    using ‹H ⊆ B› ‹i < length α› fi assms
    by (metis image-mono nsets-mono subset-trans)
qed
```

Holds because no n-sets exist!

```
lemma partn-lst-VWF-degenerate:
  assumes k < n
  shows partn-lst-VWF ω (ord-of-nat k # αs) n
proof (clarsimp simp: partn-lst-def)
  fix f :: V set ⇒ nat
  have [elts (ord-of-nat k)]^n = {}
    by (simp add: nsets-eq-empty assms finite-Ord-omega)
```

then have f ‘ $[elts\ (ord\text{-}of\text{-}nat\ k)]^n \subseteq \{0\}$
by *auto*
then show $\exists\, i < Suc\ (length\ \alpha s).\ \exists\, H{\subseteq}elts\ \omega.\ tp\ H = (ord\text{-}of\text{-}nat\ k\ \#\ \alpha s)\ !\ i \wedge f$ ‘ $[H]^n \subseteq \{i\}$
using *assms ordertype-eq-Ord* [*of ord-of-nat k*] *elts-ord-of-nat less-Suc-eq-0-disj*
by *fastforce*
qed

lemma *partn-lst-VWF-ω-2*:
assumes *Ord* α
shows *partn-lst-VWF* $(\omega \uparrow (1{+}\alpha))$ $[2,\ \omega \uparrow (1{+}\alpha)]$ *2* (**is** *partn-lst-VWF ?β - -*)
proof (*clarsimp simp*: *partn-lst-def*)
fix *f*
assume *f*: $f \in [elts\ ?\beta]^2 \to \{..{<}Suc\ (Suc\ 0)\}$
show $\exists\, i{<}Suc\ (Suc\ 0).\ \exists\, H{\subseteq}elts\ ?\beta.\ tp\ H = [2,\ ?\beta]\ !\ i \wedge f$ ‘ $[H]^2 \subseteq \{i\}$
proof (*cases* $\exists\, x \in elts\ ?\beta.\ \exists\, y \in elts\ ?\beta.\ x \neq y \wedge f\{x,y\} = 0$)
case *True*
then obtain *x y* **where** $x \in elts\ ?\beta\ y \in elts\ ?\beta\ x \neq y\ f\ \{x,\ y\} = 0$
by *auto*
then have $\{x,y\} \subseteq elts\ ?\beta\ tp\ \{x,y\} = 2\ f$ ‘ $[\{x,\ y\}]^2 \subseteq \{0\}$
by *auto* (*simp add*: *eval-nat-numeral ordertype-VWF-finite-nat*)
with ‹$x \neq y$› **show** *?thesis*
by (*metis nth-Cons-0 zero-less-Suc*)
next
case *False*
with *f* **have** $\forall\, x{\in}elts\ ?\beta.\ \forall\, y{\in}elts\ ?\beta.\ x \neq y \longrightarrow f\ \{x,\ y\} = 1$
unfolding *Pi-iff* **using** *lessThan-Suc* **by** *force*
then have *tp* $(elts\ ?\beta) = ?\beta\ f$ ‘ $[elts\ ?\beta]^2 \subseteq \{Suc\ 0\}$
by (*auto simp*: *assms nsets-2-eq*)
then show *?thesis*
by (*metis lessI nth-Cons-0 nth-Cons-Suc subsetI*)
qed
qed

2.2 Relating partition properties on VWF to the general case

Two very similar proofs here!

lemma *partn-lst-imp-partn-lst-VWF-eq*:
assumes *part*: *partn-lst r U α n* **and** *β*: *ordertype U r* $= \beta$ **and** *small U*
and *r*: *wf r trans r total-on U r*
shows *partn-lst-VWF β α n*
unfolding *partn-lst-def*
proof *clarsimp*
fix *f*
assume *f*: $f \in [elts\ \beta]^n \to \{..{<}length\ \alpha\}$
define *cv* **where** $cv \equiv \lambda X.\ ordermap\ U\ r$ ‘ X
have *bij*: *bij-betw* $(ordermap\ U\ r)\ U\ (elts\ \beta)$
using *ordermap-bij* [*of r U*] *assms* **by** *blast*
then have *bij-cv*: *bij-betw cv* $([U]^n)\ ([elts\ \beta]^n)$

using *bij-betw-nsets cv-def* **by** *blast*
then have *func*: $f \circ cv \in [U]^n \to \{..<length\ \alpha\}$ **and** *inj-on* $(ordermap\ U\ r)\ U$
using *bij bij-betw-def bij-betw-apply f* **by** *fastforce+*
then have *cv-part*: $[\![\forall x{\in}[X]^n.\ f\ (cv\ x) = i;\ X \subseteq U;\ a \in [cv\ X]^n]\!] \Longrightarrow f\ a = i$
for *a X i n*
by (*force simp*: *cv-def nsets-def subset-image-iff inj-on-subset finite-image-iff card-image*)
have *ot-eq* [*simp*]: $tp\ (cv\ X) = ordertype\ X\ r$ **if** $X \subseteq U$ **for** X
unfolding *cv-def*
by (*meson* ‹*small U*› *ordertype-image-ordermap r that total-on-subset*)
obtain $X\ i$ **where** $X \subseteq U$ **and** X: $ordertype\ X\ r = \alpha!i\ (f \circ cv)\ `\ [X]^n \subseteq \{i\}$
and $i < length\ \alpha$
using *part func* **by** (*auto simp*: *partn-lst-def*)
show $\exists i < length\ \alpha.\ \exists H{\subseteq}elts\ \beta.\ tp\ H = \alpha!i \wedge f\ `\ [H]^n \subseteq \{i\}$
proof (*intro exI conjI*)
show $i < length\ \alpha$
by (*simp add*: ‹$i < length\ \alpha$›)
show $cv\ X \subseteq elts\ \beta$
using ‹$X \subseteq U$› *bij bij-betw-imp-surj-on cv-def* **by** *blast*
show $tp\ (cv\ X) = \alpha\ !\ i$
by (*simp add*: *X*(*1*) ‹$X \subseteq U$›)
show $f\ `\ [cv\ X]^n \subseteq \{i\}$
using *X* ‹$X \subseteq U$› *cv-part* **unfolding** *image-subset-iff cv-def*
by (*metis comp-apply insertCI singletonD*)
qed
qed

lemma *partn-lst-imp-partn-lst-VWF*:
assumes *part*: *partn-lst r U α n* **and** *β*: $ordertype\ U\ r \le \beta$ *small U*
and *r*: *wf r trans r total-on U r*
shows *partn-lst-VWF β α n*
by (*metis assms less-eq-V-def partn-lst-imp-partn-lst-VWF-eq partn-lst-greater-resource*)

lemma *partn-lst-VWF-imp-partn-lst-eq*:
assumes *part*: *partn-lst-VWF β α n* **and** *β*: $ordertype\ U\ r = \beta$ *small U*
and *r*: *wf r trans r total-on U r*
shows *partn-lst r U α n*
unfolding *partn-lst-def*
proof *clarsimp*
fix *f*
assume *f*: $f \in [U]^n \to \{..<length\ \alpha\}$
define *cv* **where** $cv \equiv \lambda X.\ inv\text{-}into\ U\ (ordermap\ U\ r)\ `\ X$
have *bij*: *bij-betw* $(ordermap\ U\ r)\ U\ (elts\ \beta)$
using *ordermap-bij* [*of r U*] *assms* **by** *blast*
then have *bij-cv*: *bij-betw cv* $([elts\ \beta]^n)\ ([U]^n)$
using *bij-betw-nsets bij-betw-inv-into* **unfolding** *cv-def* **by** *blast*
then have *func*: $f \circ cv \in [elts\ \beta]^n \to \{..<length\ \alpha\}$
using *bij-betw-apply f* **by** *fastforce*
have *inj-on* $(ordermap\ U\ r)\ U$

```
    using bij bij-betw-def by blast
  then have cv-part: ⟦∀ x∈[X]^n. f (cv x) = i; X ⊆ elts β; a ∈ [cv X]^n⟧ ⟹ f a =
i for a X i n
    by (metis bij bij-betw-imp-surj-on cv-def inj-on-inv-into nset-image-obtains)
  have ot-eq [simp]: ordertype (cv X) r = tp X if X ⊆ elts β for X
    unfolding cv-def
  proof (rule ordertype-inc-eq)
    show small X
      using down that by auto
    show (inv-into U (ordermap U r) x, inv-into U (ordermap U r) y) ∈ r
      if x ∈ X y ∈ X and (x,y) ∈ VWF for x y
    proof −
      have xy: x ∈ ordermap U r ‘ U y ∈ ordermap U r ‘ U
        using ‹X ⊆ elts β› ‹x ∈ X› ‹y ∈ X› bij bij-betw-imp-surj-on by blast+
      then have eq: ordermap U r (inv-into U (ordermap U r) x) = x ordermap U
r (inv-into U (ordermap U r) y) = y
        by (meson f-inv-into-f)+
      then have y ∉ elts x
          by (metis (no-types) VWF-non-refl mem-imp-VWF that(3) trans-VWF
trans-def)
      with r show ?thesis
      by (metis (no-types) VWF-non-refl xy eq assms(3) inv-into-into ordermap-mono
that(3) total-on-def)
    qed
  qed (use r in auto)
  obtain X i where X ⊆ elts β and X: tp X = α!i (f ∘ cv) ‘ [X]^n ⊆ {i}
    and i < length α
    using part func by (auto simp: partn-lst-def)
  show ∃ i < length α. ∃ H⊆U. ordertype H r = α!i ∧ f ‘ [H]^n ⊆ {i}
  proof (intro exI conjI)
    show i < length α
      by (simp add: ‹i < length α›)
    show cv X ⊆ U
      using ‹X ⊆ elts β› bij bij-betw-imp-surj-on bij-betw-inv-into cv-def by blast
    show ordertype (cv X) r = α ! i
      by (simp add: X(1) ‹X ⊆ elts β›)
    show f ‘ [cv X]^n ⊆ {i}
      using X ‹X ⊆ elts β› cv-part unfolding image-subset-iff cv-def
      by (metis comp-apply insertCI singletonD)
  qed
qed

corollary partn-lst-VWF-imp-partn-lst:
  assumes partn-lst-VWF β α n and β: ordertype U r ≥ β small U
          wf r trans r total-on U r
  shows partn-lst r U α n
  by (metis assms less-eq-V-def partn-lst-VWF-imp-partn-lst-eq partn-lst-greater-resource)
```

2.3 Simple consequences of the definitions

A restatement of the infinite Ramsey theorem using partition notation

lemma *Ramsey-partn*: *partn-lst-VWF* ω [ω,ω] *2*
proof (*clarsimp simp*: *partn-lst-def*)
  fix *f*
  assume $f \in [elts\ \omega]^2 \to \{..<Suc\ (Suc\ 0)\}$
  then have $*$: $\forall x{\in}elts\ \omega.\ \forall y{\in}elts\ \omega.\ x \neq y \longrightarrow f\ \{x, y\} < 2$
    by (*auto simp*: *nsets-def eval-nat-numeral*)
  obtain *H i* **where** *H*: $H \subseteq elts\ \omega$ **and** *infinite H*
    and *t*: $i < Suc\ (Suc\ 0)$
    and *teq*: $\forall x{\in}H.\ \forall y{\in}H.\ x \neq y \longrightarrow f\ \{x, y\} = i$
    using *Ramsey2* [*OF infinite-ω* $*$] **by** (*auto simp*: *eval-nat-numeral*)
  then have *tp H* = [ω, ω] ! *i*
    using *less-2-cases eval-nat-numeral ordertype-infinite-ω* **by** *force*
  moreover have $f\ `\ \{N.\ N \subseteq H \wedge finite\ N \wedge card\ N = 2\} \subseteq \{i\}$
    by (*force simp*: *teq card-2-iff*)
  ultimately have $f\ `\ [H]^2 \subseteq \{i\}$
    by (*metis* (*no-types*) *nsets-def numeral-2-eq-2*)
  then show $\exists i{<}Suc\ (Suc\ 0).\ \exists H{\subseteq}elts\ \omega.$ *tp H* = [ω,ω] ! $i \wedge f\ `\ [H]^2 \subseteq \{i\}$
    using *H* ‹*tp H* = [ω, ω] ! *i*› *t* **by** *blast*
qed

This is the counterexample sketched in Hajnal and Larson, section 9.1.

proposition *omega-basic-counterexample*:
  assumes *Ord* α
  shows ¬ *partn-lst-VWF* α [*succ* (*vcard* α), ω] *2*
proof −
  obtain π **where** *funπ*: $\pi \in elts\ \alpha \to elts\ (vcard\ \alpha)$ **and** *injπ*: *inj-on* π (*elts* α)
    using *inj-into-vcard* **by** *auto*
  have *Ordπ*: *Ord* (π *x*) **if** $x \in elts\ \alpha$ **for** *x*
    using *Ord-in-Ord funπ that* **by** *fastforce*
  define *f* **where** $f\ A \equiv @i{::}nat.\ \exists x\ y.\ A = \{x,y\} \wedge x < y \wedge (\pi\ x < \pi\ y \wedge i{=}0 \vee \pi\ y < \pi\ x \wedge i{=}1)$ **for** *A*
  have *f-Pi*: $f \in [elts\ \alpha]^2 \to \{..<Suc\ (Suc\ 0)\}$
  proof
    fix *A*
    assume $A \in [elts\ \alpha]^2$
    then obtain *x y* **where** *xy*: $x \in elts\ \alpha\ y \in elts\ \alpha\ x < y$ **and** *A*: $A = \{x,y\}$
      apply (*clarsimp simp*: *nsets-2-eq*)
      by (*metis Ord-in-Ord Ord-linear-lt assms insert-commute*)
    consider $\pi\ x < \pi\ y \mid \pi\ y < \pi\ x$
      by (*metis Ordπ Ord-linear-lt injπ inj-onD less-imp-not-eq2 xy*)
    then show $f\ A \in \{..<Suc\ (Suc\ 0)\}$
      by (*metis A One-nat-def lessI lessThan-iff zero-less-Suc* ‹$x < y$› *A exE-some* [*OF - f-def*])
  qed
  have *fiff*: $\pi\ x < \pi\ y \wedge i{=}0 \vee \pi\ y < \pi\ x \wedge i{=}1$
    if *f*: $f\ \{x,y\} = i$ **and** *xy*: $x \in elts\ \alpha\ y \in elts\ \alpha\ x{<}y$ **for** *x y i*

proof −
consider $\pi\ x < \pi\ y \mid \pi\ y < \pi\ x$
using *xy* **by** (*metis Ordπ Ord-linear-lt injπ inj-onD less-V-def*)
then show *?thesis*
proof *cases*
case *1*
then have $f\{x,y\} = 0$
using ‹$x<y$› **by** (*force simp*: *f-def Set.doubleton-eq-iff*)
then show *?thesis*
using *1 f* **by** *auto*
next
case *2*
then have $f\{x,y\} = 1$
using ‹$x<y$› **by** (*force simp*: *f-def Set.doubleton-eq-iff*)
then show *?thesis*
using *2 f* **by** *auto*
qed
qed
have *False*
if *eq*: $tp\ H = succ\ (vcard\ \alpha)$ **and** *H*: $H \subseteq elts\ \alpha$
and *0*: $\bigwedge A.\ A \in [H]^2 \Longrightarrow f\ A = 0$ **for** *H*
proof −
have [*simp*]: *small H*
using *H down* **by** *auto*
have *OH*: *Ord x* **if** $x \in H$ **for** *x*
using *H Ord-in-Ord* ‹*Ord α*› *that* **by** *blast*
have π: $\pi\ x < \pi\ y$ **if** $x{\in}H\ y{\in}H\ x{<}y$ **for** *x y*
using *0* [*of* $\{x,y\}$] *that H fiff* **by** (*fastforce simp*: *nsets-2-eq*)
have *sub-vcard*: $\pi\ `\ H \subseteq elts\ (vcard\ \alpha)$
using *H funπ* **by** *auto*
have $tp\ H = tp\ (\pi\ `\ H)$
proof (*rule ordertype-VWF-inc-eq* [*symmetric*])
show $\pi\ `\ H \subseteq ON$
using *H Ordπ* **by** *blast*
qed (*auto simp*: *π OH subsetI*)
also have $\ldots \leq vcard\ \alpha$
by (*simp add*: *H sub-vcard assms ordertype-le-Ord*)
finally show *False*
by (*simp add*: *eq succ-le-iff*)
qed
moreover have *False*
if *eq*: $tp\ H = \omega$ **and** *H*: $H \subseteq elts\ \alpha$
and *1*: $\bigwedge A.\ A \in [H]^2 \Longrightarrow f\ A = Suc\ 0$ **for** *H*
proof −
have [*simp*]: *small H*
using *H down* **by** *auto*
define η **where** $\eta \equiv inv\text{-}into\ H\ (ordermap\ H\ VWF) \circ ord\text{-}of\text{-}nat$
have *bij*: *bij-betw* (*ordermap H VWF*) *H* (*elts ω*)
by (*metis ordermap-bij* ‹*small H*› *eq total-on-VWF wf-VWF*)

then have *bij-betw* (*inv-into H* (*ordermap H VWF*)) (*elts* ω) *H*
by (*simp add*: *bij-betw-inv-into*)
then have η: *bij-betw* η *UNIV H*
unfolding η-*def*
by (*metis* ω-*def bij-betw-comp-iff2 bij-betw-def elts-of-set inf inj-ord-of-nat order-refl*)
moreover have *Ord*η: *Ord* (η *k*) **for** *k*
by (*meson H Ord-in-Ord UNIV-I* η *assms bij-betw-apply subsetD*)
moreover obtain *k* **where** *k*: (π (η(*Suc k*)), π (η *k*)) $\notin$ *VWF*
by (*meson wf-VWF wf-iff-no-infinite-down-chain*)
moreover have π: π *y* < π *x* **if** *x*∈*H y*∈*H x*<*y* **for** *x y*
using *1* [*of* {*x*,*y*}] *that H fiff* **by** (*fastforce simp*: *nsets-2-eq*)
ultimately have η (*Suc k*) ≤ η *k*
by (*metis H Ord*π *Ord-linear2 VWF-iff-Ord-less bij-betw-def rangeI subset-iff*)
then have (η (*Suc k*), η *k*) ∈ *VWF* ∨ η (*Suc k*) = η *k*
using *Ord*η *Ord-mem-iff-lt* **by** *auto*
then have *ordermap H VWF* (η (*Suc k*)) ≤ *ordermap H VWF* (η *k*)
by (*metis* η ‹*small H*› *bij-betw-imp-surj-on ordermap-mono-le rangeI trans-VWF wf-VWF*)
moreover have *ordermap H VWF* (η (*Suc k*)) = *succ* (*ord-of-nat k*)
unfolding η-*def* **using** *bij bij-betw-inv-into-right* **by** *force*
moreover have *ordermap H VWF* (η *k*) = *ord-of-nat k*
using η-*def bij bij-betw-inv-into-right* **by** *force*
ultimately show *False*
by (*simp add*: *less-eq-V-def*)
qed
ultimately show *?thesis*
apply (*simp add*: *partn-lst-def image-subset-iff*)
by (*metis f-Pi less-2-cases nth-Cons-0 nth-Cons-Suc numeral-2-eq-2*)
qed

2.4 Specker's theorem

definition *form-split* :: [*nat*,*nat*,*nat*,*nat*,*nat*] ⇒ *bool* **where**
form-split a b c d i ≡ *a* ≤ *c* ∧ (*i*=*0* ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d* ∨
i=*1* ∧ *a*<*c* ∧ *c*<*b* ∧ *b*<*d* ∨
i=*2* ∧ *a*<*c* ∧ *c*<*d* ∧ *d*<*b* ∨
i=*3* ∧ *a*=*c* ∧ *b*≠*d*)

definition *form* :: [(*nat*∗*nat*)*set*, *nat*] ⇒ *bool* **where**
form u i ≡ ∃ *a b c d*. *u* = {(*a*,*b*),(*c*,*d*)} ∧ *form-split a b c d i*

definition *scheme* :: [(*nat*∗*nat*)*set*] ⇒ *nat set* **where**
scheme u ≡ *fst* ‘ *u* ∪ *snd* ‘ *u*

definition *UU* :: (*nat*∗*nat*) *set*
where *UU* ≡ {(*a*,*b*). *a* < *b*}

lemma *ordertype-UNIV-*ω*2*: *ordertype UNIV pair-less* = ω↑*2*

using *ordertype-Times* [*of concl*: *UNIV UNIV less-than less-than*]
by (*simp add*: *total-less-than pair-less-def ordertype-nat-ω numeral-2-eq-2*)

lemma *ordertype-UU-ge-ω2*: *ordertype UNIV pair-less* ≤ *ordertype UU pair-less*
proof (*rule ordertype-inc-le*)
define π **where** π ≡ λ(*m*,*n*). (*m*, *Suc* (*m*+*n*))
show (π (*x*::*nat* × *nat*), π *y*) ∈ *pair-less* **if** (*x*, *y*) ∈ *pair-less* **for** *x y*
using *that* **by** (*auto simp*: π*-def pair-less-def split*: *prod.split*)
show *range* π ⊆ *UU*
by (*auto simp*: π*-def UU-def*)
qed *auto*

lemma *ordertype-UU-ω2*: *ordertype UU pair-less* = *ω*↑*2*
by (*metis eq-iff ordertype-UNIV-ω2 ordertype-UU-ge-ω2 ordertype-mono small top-greatest trans-pair-less wf-pair-less*)

Lemma 2.3 of Jean A. Larson, A short proof of a partition theorem for the ordinal ω^ω. *Annals of Mathematical Logic*, 6:129–145, 1973.

lemma *lemma-2-3*:
fixes *f* :: (*nat* × *nat*) *set* ⇒ *nat*
assumes *f* ∈ [*UU*]^2 → {..<*Suc* (*Suc 0*)}
obtains *N js* **where** *infinite N* **and** ⋀*k u*. ⟦*k* < *4*; *u* ∈ [*UU*]^2; *form u k*; *scheme u* ⊆ *N*⟧ ⟹ *f u* = *js*!*k*
proof −
have *f-less2*: *f* {*p*,*q*} < *Suc* (*Suc 0*) **if** *p* ≠ *q p* ∈ *UU q* ∈ *UU* **for** *p q*
proof −
have {*p*,*q*} ∈ [*UU*]^2
using *that* **by** (*simp add*: *nsets-def*)
then show *?thesis*
using *assms* **by** (*simp add*: *Pi-iff*)
qed
define *f0* **where** *f0* ≡ (λ*A*::*nat set*. *THE x*. ∃ *a b c d*. *A* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d* ∧ *x* = *f* {(*a*,*b*),(*c*,*d*)})
have *f0*: *f0* {*a*,*b*,*c*,*d*} = *f* {(*a*,*b*),(*c*,*d*)} **if** *a*<*b b*<*c c*<*d* **for** *a b c d*
unfolding *f0-def*
apply (*rule theI2* [**where** *a* = *f* {(*a*,*b*), (*c*,*d*)}])
using *that* **by** (*force simp*: *insert-eq-iff split*: *if-split-asm*)+
have *f0 X* < *Suc* (*Suc 0*)
if *finite X* **and** *card X* = *4* **for** *X*
proof −
have *X* ∈ [*X*]^4
using *that* **by** (*auto simp*: *nsets-def*)
then obtain *a b c d* **where** *X* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d*
by (*auto simp*: *ordered-nsets-4-eq*)
then show *?thesis*
using *f0 f-less2* **by** (*auto simp*: *UU-def*)
qed
then have ∃ *N t*. *infinite N* ∧ *t* < *Suc* (*Suc 0*)
∧ (∀ *X*. *X* ⊆ *N* ∧ *finite X* ∧ *card X* = *4* ⟶ *f0 X* = *t*)

using *Ramsey* [*of UNIV 4 f0 2*] **by** (*simp add*: *eval-nat-numeral*)
then obtain *N0 j0* **where** *infinite N0* **and** *j0*: *j0* < *Suc* (*Suc 0*) **and** *N0*: $\bigwedge A$. $A \in [N0]^4 \Longrightarrow$ *f0 A* = *j0*
by (*auto simp*: *nsets-def*)

define *f1* **where** *f1* ≡ (λ*A*::*nat set*. *THE x*. ∃ *a b c d*. *A* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d* ∧ *x* = *f* {(*a*,*c*),(*b*,*d*)})
have *f1*: *f1* {*a*,*b*,*c*,*d*} = *f* {(*a*,*c*),(*b*,*d*)} **if** *a*<*b* *b*<*c* *c*<*d* **for** *a b c d*
unfolding *f1-def*
apply (*rule theI2* [**where** *a* = *f* {(*a*,*c*), (*b*,*d*)}])
using *that* **by** (*force simp*: *insert-eq-iff split*: *if-split-asm*)+
have *f1 X* < *Suc* (*Suc 0*)
if *finite X* **and** *card X* = *4* **for** *X*
proof −
have *X* ∈ [*X*]^*4*
using *that* **by** (*auto simp*: *nsets-def*)
then obtain *a b c d* **where** *X* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d*
by (*auto simp*: *ordered-nsets-4-eq*)
then show *?thesis*
using *f1 f-less2* **by** (*auto simp*: *UU-def*)
qed
then have ∃ *N t*. *N* ⊆ *N0* ∧ *infinite N* ∧ *t* < *Suc* (*Suc 0*)
∧ (∀ *X*. *X* ⊆ *N* ∧ *finite X* ∧ *card X* = *4* ⟶ *f1 X* = *t*)
using ‹*infinite N0*› *Ramsey* [*of N0 4 f1 2*] **by** (*simp add*: *eval-nat-numeral*)
then obtain *N1 j1* **where** *N1* ⊆ *N0 infinite N1* **and** *j1*: *j1* < *Suc* (*Suc 0*) **and** *N1*: $\bigwedge A$. *A* ∈ [*N1*]^*4* ⟹ *f1 A* = *j1*
by (*auto simp*: *nsets-def*)

define *f2* **where** *f2* ≡ (λ*A*::*nat set*. *THE x*. ∃ *a b c d*. *A* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d* ∧ *x* = *f* {(*a*,*d*),(*b*,*c*)})
have *f2*: *f2* {*a*,*b*,*c*,*d*} = *f* {(*a*,*d*),(*b*,*c*)} **if** *a*<*b* *b*<*c* *c*<*d* **for** *a b c d*
unfolding *f2-def*
apply (*rule theI2* [**where** *a* = *f* {(*a*,*d*), (*b*,*c*)}])
using *that* **by** (*force simp*: *insert-eq-iff split*: *if-split-asm*)+
have *f2 X* < *Suc* (*Suc 0*)
if *finite X* **and** *card X* = *4* **for** *X*
proof −
have *X* ∈ [*X*]^*4*
using *that* **by** (*auto simp*: *nsets-def*)
then obtain *a b c d* **where** *X* = {*a*,*b*,*c*,*d*} ∧ *a*<*b* ∧ *b*<*c* ∧ *c*<*d*
by (*auto simp*: *ordered-nsets-4-eq*)
then show *?thesis*
using *f2 f-less2* **by** (*auto simp*: *UU-def*)
qed
then have ∃ *N t*. *N* ⊆ *N1* ∧ *infinite N* ∧ *t* < *Suc* (*Suc 0*)
∧ (∀ *X*. *X* ⊆ *N* ∧ *finite X* ∧ *card X* = *4* ⟶ *f2 X* = *t*)
using ‹*infinite N1*› *Ramsey* [*of N1 4 f2 2*] **by** (*simp add*: *eval-nat-numeral*)
then obtain *N2 j2* **where** *N2* ⊆ *N1 infinite N2* **and** *j2*: *j2* < *Suc* (*Suc 0*) **and** *N2*: $\bigwedge A$. *A* ∈ [*N2*]^*4* ⟹ *f2 A* = *j2*

```
    by (auto simp: nsets-def)

  define f3 where f3 ≡ (λA::nat set. THE x. ∃ a b c. A = {a,b,c} ∧ a<b ∧ b<c
∧ x = f {(a,b),(a,c)})
  have f3: f3 {a,b,c} = f {(a,b),(a,c)} if a<b b<c for a b c
    unfolding f3-def
    apply (rule theI2 [where a = f {(a,b), (a,c)}])
    using that by (force simp: insert-eq-iff split: if-split-asm)+
  have f3': f3 {a,b,c} = f {(a,b),(a,c)} if a<c c<b for a b c
    using f3 [of a c b] that
    by (simp add: insert-commute)
  have f3 X < Suc (Suc 0)
    if finite X and card X = 3 for X
  proof −
    have X ∈ [X]^3
      using that by (auto simp: nsets-def)
    then obtain a b c where X = {a,b,c} ∧ a<b ∧ b<c
      by (auto simp: ordered-nsets-3-eq)
    then show ?thesis
      using f3 f-less2 by (auto simp: UU-def)
  qed
  then have ∃ N t. N ⊆ N2 ∧ infinite N ∧ t < Suc (Suc 0)
          ∧ (∀ X. X ⊆ N ∧ finite X ∧ card X = 3 ⟶ f3 X = t)
    using ‹infinite N2› Ramsey [of N2 3 f3 2] by (simp add: eval-nat-numeral)
  then obtain N3 j3 where N3 ⊆ N2 infinite N3 and j3: j3 < Suc (Suc 0) and
N3: ⋀A. A ∈ [N3]^3 ⟹ f3 A = j3
    by (auto simp: nsets-def)

  show thesis
  proof
    fix k u
    assume k < 4
      and u: form u k scheme u ⊆ N3
      and UU: u ∈ [UU]^2
    then consider (0) k=0 | (1) k=1 | (2) k=2 | (3) k=3
      by linarith
    then show f u = [j0,j1,j2,j3] ! k
    proof cases
      case 0
      have N3 ⊆ N0
        using ‹N1 ⊆ N0› ‹N2 ⊆ N1› ‹N3 ⊆ N2› by auto
      then show ?thesis
        using u 0
        apply (auto simp: form-def form-split-def scheme-def simp flip: f0)
        apply (force simp: nsets-def intro: N0)
        done
    next
      case 1
      have N3 ⊆ N1
```

```
    using ‹N2 ⊆ N1› ‹N3 ⊆ N2› by auto
   then show ?thesis
    using u 1
    apply (auto simp: form-def form-split-def scheme-def simp flip: f1)
    apply (force simp: nsets-def intro: N1)
    done
  next
   case 2
   then show ?thesis
    using u ‹N3 ⊆ N2›
    apply (auto simp: form-def form-split-def scheme-def nsets-def simp flip: f2)
    apply (force simp: nsets-def intro: N2)
    done
  next
   case 3
   { fix a b d
    assume {(a, b), (a, d)} ∈ [UU]^2
     and ∗: a ∈ N3 b ∈ N3 d ∈ N3 b ≠ d
    then have a<b a<d
     by (auto simp: UU-def nsets-def)
    then have f {(a, b), (a, d)} = j3
     using ∗
     apply (auto simp: neq-iff simp flip: f3 f3')
      apply (force simp: nsets-def intro: N3)+
     done
   }
   then show ?thesis
    using u UU 3
    by (auto simp: form-def form-split-def scheme-def)
  qed
 qed (rule ‹infinite N3›)
qed
```

Lemma 2.4 of Jean A. Larson, ibid.

```
lemma lemma-2-4:
 assumes infinite N k < 4
 obtains M where M ∈ [UU]^m ⋀u. u ∈ [M]^2 ⟹ form u k ⋀u. u ∈ [M]^2 ⟹
scheme u ⊆ N
proof −
 obtain f:: nat ⇒ nat where bij-betw f UNIV N strict-mono f
   using assms by (meson bij-enumerate enumerate-mono strict-monoI)
 then have iff[simp]: f x = f y ⟷ x=y f x < f y ⟷ x<y for x y
   by (simp-all add: strict-mono-eq strict-mono-less)
 have [simp]: f x ∈ N for x
   using bij-betw-apply [OF ‹bij-betw f UNIV N›] by blast
 define M0 where M0 = (λi. (f(2∗i), f(Suc(2∗i)))) ‘ {..<m}
 define M1 where M1 = (λi. (f i, f(m+i))) ‘ {..<m}
 define M2 where M2 = (λi. (f i, f(2∗m−i))) ‘ {..<m}
 define M3 where M3 = (λi. (f 0, f (Suc i))) ‘ {..<m}
```

consider (*0*) *k=0* | (*1*) *k=1* | (*2*) *k=2* | (*3*) *k=3*
using *assms* **by** *linarith*
then show *thesis*
proof *cases*
case *0*
show *?thesis*
proof
have *inj-on* (λ*i*. (*f* (*2* ∗ *i*), *f* (*Suc* (*2* ∗ *i*)))) {..<*m*}
by (*auto simp*: *inj-on-def*)
then show *M0* ∈ [*UU*]^m
by (*simp add*: *M0-def nsets-def card-image UU-def image-subset-iff*)
next
fix *u*
assume *u*: (*u*::(*nat* × *nat*) *set*) ∈ [*M0*]^2
then obtain *x y* **where** *u* = {*x*,*y*} *x* ≠ *y x* ∈ *M0 y* ∈ *M0*
by (*auto simp*: *nsets-2-eq*)
then obtain *i j* **where** *i*<*j j*<*m* **and** *ueq*: *u* = {(*f*(*2*∗*i*), *f*(*Suc*(*2*∗*i*))), (*f*(*2*∗*j*), *f*(*Suc*(*2*∗*j*)))}
apply (*clarsimp simp*: *M0-def*)
apply (*metis* (*full-types*) *insert-commute less-linear*)+
done
moreover have *f* (*2* ∗ *i*) ≤ *f* (*2* ∗ *j*)
by (*simp add*: ‹*i*<*j*› *less-imp-le-nat*)
ultimately show *form u k*
apply (*simp add*: *0 form-def form-split-def nsets-def*)
apply (*rule-tac x*=*f* (*2* ∗ *i*) **in** *exI*)
apply (*rule-tac x*=*f* (*Suc* (*2* ∗ *i*)) **in** *exI*)
apply (*rule-tac x*=*f* (*2* ∗ *j*) **in** *exI*)
apply (*rule-tac x*=*f* (*Suc* (*2* ∗ *j*)) **in** *exI*)
apply *auto*
done
show *scheme u* ⊆ *N*
using *ueq* **by** (*auto simp*: *scheme-def*)
qed
next
case *1*
show *?thesis*
proof
have *inj-on* (λ*i*. (*f i*, *f*(*m*+*i*))) {..<*m*}
by (*auto simp*: *inj-on-def*)
then show *M1* ∈ [*UU*]^m
by (*simp add*: *M1-def nsets-def card-image UU-def image-subset-iff*)
next
fix *u*
assume *u*: (*u*::(*nat* × *nat*) *set*) ∈ [*M1*]^2
then obtain *x y* **where** *u* = {*x*,*y*} *x* ≠ *y x* ∈ *M1 y* ∈ *M1*
by (*auto simp*: *nsets-2-eq*)
then obtain *i j* **where** *i*<*j j*<*m* **and** *ueq*: *u* = {(*f i*, *f*(*m*+*i*)), (*f j*, *f*(*m*+*j*))}
apply (*auto simp*: *M1-def*)

```
      apply (metis (full-types) insert-commute less-linear)+
      done
    then show form u k
      apply (simp add: 1 form-def form-split-def nsets-def)
      by (metis iff(2) nat-add-left-cancel-less nat-less-le trans-less-add1)
    show scheme u ⊆ N
      using ueq by (auto simp: scheme-def)
  qed
next
  case 2
  show ?thesis
  proof
    have inj-on (λi. (f i, f(2∗m−i))) {..<m}
      by (auto simp: inj-on-def)
    then show M2 ∈ [UU]^m
      by (auto simp: M2-def nsets-def card-image UU-def image-subset-iff)
  next
    fix u
    assume u: (u::(nat × nat) set) ∈ [M2]^2
    then obtain x y where u = {x,y} x ≠ y x ∈ M2 y ∈ M2
      by (auto simp: nsets-2-eq)
      then obtain i j where i<j j<m and ueq: u = {(f i, f(2∗m−i)), (f j,
f(2∗m−j))}
      apply (auto simp: M2-def)
      apply (metis (full-types) insert-commute less-linear)+
      done
    then show form u k
      apply (simp add: 2 form-def form-split-def nsets-def)
      apply (rule-tac x=f i in exI)
      apply (rule-tac x=f (2 ∗ m − i) in exI)
      apply (rule-tac x=f j in exI)
      apply (rule-tac x=f (2 ∗ m − j) in exI)
      apply (auto simp: less-imp-le-nat)
      done
    show scheme u ⊆ N
      using ueq by (auto simp: scheme-def)
  qed
next
  case 3
  show ?thesis
  proof
    have inj-on (λi. (f 0, f (Suc i))) {..<m}
      by (auto simp: inj-on-def)
    then show M3 ∈ [UU]^m
      by (auto simp: M3-def nsets-def card-image UU-def image-subset-iff)
  next
    fix u
    assume u: (u::(nat × nat) set) ∈ [M3]^2
    then obtain x y where u = {x,y} x ≠ y x ∈ M3 y ∈ M3
```

```
      by (auto simp: nsets-2-eq)
    then obtain i j where i<j j<m and ueq: u = {(f 0, f(Suc i)), (f 0, f(Suc
j))}
      apply (auto simp: M3-def)
      apply (metis (full-types) insert-commute less-linear)+
      done
    then show form u k
      by (fastforce simp: 3 form-def form-split-def nsets-def)
    show scheme u ⊆ N
      using ueq by (auto simp: scheme-def)
  qed
 qed
qed
```

Lemma 2.5 of Jean A. Larson, ibid.

```
lemma lemma-2-5:
 assumes infinite N
 obtains X where X ⊆ UU ordertype X pair-less = ω↑2
         ⋀u. u ∈ [X]^2 ⟹ (∃ k<4. form u k) ∧ scheme u ⊆ N
proof −
 obtain C
   where dis: pairwise (λi j. disjnt (C i) (C j)) UNIV
     and N: (⋃ i. C i) ⊆ N and infC: ⋀i::nat. infinite (C i)
   using assms infinite-infinite-partition by blast
 then have ∃φ::nat ⇒ nat. inj φ ∧ range φ = C i ∧ strict-mono φ for i
   by (metis nat-infinite-iff strict-mono-on-imp-inj-on)
 then obtain φ:: [nat,nat] ⇒ nat
    where φ: ⋀i. inj (φ i) ∧ range (φ i) = C i ∧ strict-mono (φ i)
   by metis
 then have π-in-C [simp]: φ i j ∈ C i' ⟷ i'=i for i i' j
   using dis by (fastforce simp: pairwise-def disjnt-def)
 have less-iff [simp]: φ i j' < φ i j ⟷ j' < j for i j' j
   by (simp add: φ strict-mono-less)
 let ?a = φ 0
 define X where X ≡ {(?a i, b) | i b. ?a i < b ∧ b ∈ C (Suc i)}
 show thesis
 proof
   show X ⊆ UU
     by (auto simp: X-def UU-def)
   show ordertype X pair-less = ω↑2
   proof (rule antisym)
     have ordertype X pair-less ≤ ordertype UU pair-less
       by (simp add: ‹X ⊆ UU› ordertype-mono)
     then show ordertype X pair-less ≤ ω↑2
       using ordertype-UU-ω2 by auto
     define π where π ≡ λ(i,j::nat). (?a i, φ (Suc i) (?a j))
     have ⋀i j. i < j ⟹ φ 0 i < φ (Suc i) (φ 0 j)
       by (meson φ le-less-trans less-iff strict-mono-imp-increasing)
     then have subX: π ‘ UU ⊆ X
```

```
      by (auto simp: UU-def π-def X-def)
    then have ordertype (π ‘ UU) pair-less ≤ ordertype X pair-less
      by (simp add: ordertype-mono)
    moreover have ordertype (π ‘ UU) pair-less = ordertype UU pair-less
    proof (rule ordertype-inc-eq)
      show (π x, π y) ∈ pair-less
        if x ∈ UU y ∈ UU and (x, y) ∈ pair-less for x y
        using that by (auto simp: UU-def π-def pair-less-def)
    qed auto
    ultimately show ω↑2 ≤ ordertype X pair-less
      using ordertype-UU-ω2 by simp
  qed
 next
  fix U
  assume U ∈ [X]^2
  then obtain a b c d where Ueq: U = {(a,b),(c,d)} and ne: (a,b) ≠ (c,d) and
inX: (a,b) ∈ X (c,d) ∈ X and a ≤ c
   apply (auto simp: nsets-def subset-iff eval-nat-numeral card-Suc-eq Set.doubleton-eq-iff)
    apply (metis  nat-le-linear)+
    done
  show (∃k<4. form U k) ∧ scheme U ⊆ N
  proof
    show scheme U ⊆ N
      using inX N φ by (fastforce simp: scheme-def Ueq X-def)
  next
    consider a < c | a = c ∧ b ≠ d
      using ‹a ≤ c› ne nat-less-le by blast
    then show ∃k<4. form U k
    proof cases
      case 1
      have ∗: a < b  b ≠ c c < d
        using inX by (auto simp: X-def)
      moreover have ⟦a < c; c < b; ¬ d < b⟧ ⟹ b < d
        using inX apply (clarsimp simp: X-def not-less)
        by (metis φ π-in-C imageE nat.inject nat-less-le)
       ultimately consider (k0) a<b ∧ b<c ∧ c<d | (k1) a<c ∧ c<b ∧ b<d |
(k2) a<c ∧ c<d ∧ d<b
        using 1 less-linear by blast
      then show ?thesis
      proof cases
        case k0
        then have form U 0
          unfolding form-def form-split-def using Ueq ‹a ≤ c› by blast
        then show ?thesis by force
      next
        case k1
        then have form U 1
          unfolding form-def form-split-def using Ueq ‹a ≤ c› by blast
        then show ?thesis by force
```

```
      next
        case k2
        then have form U 2
          unfolding form-def form-split-def using Ueq ‹a ≤ c› by blast
        then show ?thesis by force
      qed
    next
      case 2
      then have form-split a b c d 3
        by (auto simp: form-split-def)
      then show ?thesis
        using Ueq form-def leI by force
    qed
  qed
qed
```

Theorem 2.1 of Jean A. Larson, ibid.

```
lemma Specker-aux:
  assumes α ∈ elts ω
  shows partn-lst pair-less UU [ω↑2,α] 2
  unfolding partn-lst-def
proof clarsimp
  fix f
  assume f: f ∈ [UU]^2 → {..<Suc (Suc 0)}
  let ?P0 = ∃X ⊆ UU. ordertype X pair-less = ω↑2 ∧ f ‘ [X]^2 ⊆ {0}
  let ?P1 = ∃M ⊆ UU. ordertype M pair-less = α ∧ f ‘ [M]^2 ⊆ {1}
  have †: ?P0 ∨ ?P1
  proof (rule disjCI)
    assume ¬ ?P1
    then have not1: ⋀M. ⟦M ⊆ UU; ordertype M pair-less = α⟧ ⟹ ∃x∈[M]^2. f
x ≠ Suc 0
      by auto
    obtain m where m: α = ord-of-nat m
      using assms elts-ω by auto
    then have f-eq-0: M ∈ [UU]^m ⟹ ∃x∈[M]^2. f x = 0 for M
      using not1 [of M] finite-ordertype-eq-card [of M pair-less m] f
      apply (clarsimp simp: nsets-def eval-nat-numeral Pi-def)
      by (meson less-Suc0 not-less-less-Suc-eq subset-trans)
    obtain N js where infinite N and N: ⋀k u. ⟦k < 4; u ∈ [UU]^2; form u k;
scheme u ⊆ N⟧ ⟹ f u = js!k
      using f lemma-2-3 by blast
    obtain M0 where M0: M0 ∈ [UU]^m ⋀u. u ∈ [M0]^2 ⟹ form u 0 ⋀u. u ∈
[M0]^2 ⟹ scheme u ⊆ N
      by (rule lemma-2-4 [OF ‹infinite N›]) auto
    obtain M1 where M1: M1 ∈ [UU]^m ⋀u. u ∈ [M1]^2 ⟹ form u 1 ⋀u. u ∈
[M1]^2 ⟹ scheme u ⊆ N
      by (rule lemma-2-4 [OF ‹infinite N›]) auto
    obtain M2 where M2: M2 ∈ [UU]^m ⋀u. u ∈ [M2]^2 ⟹ form u 2 ⋀u. u ∈
```

$[M2]^2 \Longrightarrow$ *scheme* $u \subseteq N$
by (*rule lemma-2-4* [*OF* ‹*infinite N*›]) *auto*
obtain *M3* **where** *M3*: $M3 \in [UU]^m$ $\bigwedge u.\ u \in [M3]^2 \Longrightarrow$ *form u 3* $\bigwedge u.\ u \in [M3]^2 \Longrightarrow$ *scheme* $u \subseteq N$
by (*rule lemma-2-4* [*OF* ‹*infinite N*›]) *auto*
have *js*!*0* = *0*
using *N* [*of 0*] *M0 f-eq-0* [*of M0*] **by** (*force simp*: *nsets-def eval-nat-numeral*)
moreover have *js*!*1* = *0*
using *N* [*of 1*] *M1 f-eq-0* [*of M1*] **by** (*force simp*: *nsets-def eval-nat-numeral*)
moreover have *js*!*2* = *0*
using *N* [*of 2*] *M2 f-eq-0* [*of M2*] **by** (*force simp*: *nsets-def eval-nat-numeral*)
moreover have *js*!*3* = *0*
using *N* [*of 3*] *M3 f-eq-0* [*of M3*] **by** (*force simp*: *nsets-def eval-nat-numeral*)
ultimately have *js0*: *js*!*k* = *0* **if** $k < 4$ **for** *k*
using *that* **by** (*auto simp*: *eval-nat-numeral less-Suc-eq*)
obtain *X* **where** $X \subseteq UU$ **and** *otX*: *ordertype X pair-less* = $\omega\uparrow 2$
and *X*: $\bigwedge u.\ u \in [X]^2 \Longrightarrow (\exists k{<}4.\ form\ u\ k) \wedge scheme\ u \subseteq N$
using ‹*infinite N*› *lemma-2-5* **by** *auto*
moreover have *f* ‘ $[X]^2 \subseteq \{0\}$
proof (*clarsimp simp*: *image-subset-iff*)
fix *u*
assume *u*: $u \in [X]^2$
then have *u-UU2*: $u \in [UU]^2$
using ‹$X \subseteq UU$› *nsets-mono* **by** *blast*
show *f u* = *0*
using *X u N* [*OF - u-UU2*] *js0* **by** *auto*
qed
ultimately show $\exists X \subseteq UU.$ *ordertype X pair-less* = $\omega\uparrow 2 \wedge f$ ‘ $[X]^2 \subseteq \{0\}$
by *blast*
qed
then show $\exists i{<}Suc\ (Suc\ 0).\ \exists H{\subseteq}UU.$ *ordertype H pair-less* = $[\omega\uparrow 2, \alpha]$! $i \wedge f$ ‘ $[H]^2 \subseteq \{i\}$
by (*metis One-nat-def lessI nth-Cons-0 nth-Cons-Suc zero-less-Suc*)
qed

theorem *Specker*: $\alpha \in elts\ \omega \Longrightarrow$ *partn-lst-VWF* ($\omega\uparrow 2$) [$\omega\uparrow 2$,α] *2*
using *partn-lst-imp-partn-lst-VWF-eq* [*OF Specker-aux*] *ordertype-UU-ω2 wf-pair-less*
by *blast*

end
theory *Erdos-Milner*
imports *Partitions*

begin

2.5 Erds-Milner theorem

P. Erds and E. C. Milner, A Theorem in the Partition Calculus. Canadian Math. Bull. 15:4 (1972), 501-505. Corrigendum, Canadian Math. Bull.

17:2 (1974), 305.

The paper defines strong types as satisfying the criteria below. It remarks that ordinals satisfy those criteria. Here is a (too complicated) proof.

proposition *strong-ordertype-eq*:
assumes *D*: $D \subseteq elts\ \beta$ **and** $Ord\ \beta$
obtains *L* **where** $\bigcup(List.set\ L) = D$ $\bigwedge X.\ X \in List.set\ L \Longrightarrow indecomposable\ (tp\ X)$
and $\bigwedge M.\ [\![M \subseteq D;\ \bigwedge X.\ X \in List.set\ L \Longrightarrow tp\ (M \cap X) \geq tp\ X]\!] \Longrightarrow tp\ M = tp\ D$
proof −
define φ **where** $\varphi \equiv inv\text{-}into\ D\ (ordermap\ D\ VWF)$
then have *bij-φ*: $bij\text{-}betw\ \varphi\ (elts\ (tp\ D))\ D$
using *D bij-betw-inv-into down ordermap-bij* **by** *blast*
have *φ-cancel-left*: $\bigwedge d.\ d \in D \Longrightarrow \varphi\ (ordermap\ D\ VWF\ d) = d$
by (*metis D φ-def bij-betw-inv-into-left down ordermap-bij total-on-VWF wf-VWF*)
have *φ-cancel-right*: $\bigwedge \gamma.\ \gamma \in elts\ (tp\ D) \Longrightarrow ordermap\ D\ VWF\ (\varphi\ \gamma) = \gamma$
by (*metis φ-def f-inv-into-f ordermap-surj subsetD*)
have *small D D* $\subseteq ON$
using *assms down elts-subset-ON* [*of* β] **by** *auto*
then have *φ-less-iff*: $\bigwedge \gamma\ \delta.\ [\![\gamma \in elts\ (tp\ D);\ \delta \in elts\ (tp\ D)]\!] \Longrightarrow \varphi\ \gamma < \varphi\ \delta \longleftrightarrow \gamma < \delta$
by (*metis φ-def inv-ordermap-VWF-mono-iff inv-ordermap-mono-eq less-V-def*)
obtain αs **where** $List.set\ \alpha s \subseteq ON$ **and** *ω-dec* αs **and** *tpD-eq*: $tp\ D = \omega\text{-}sum\ \alpha s$
using *Ord-ordertype ω-nf-exists* **by** *blast* — Cantor normal form of the ordertype
have *ord* [*simp*]: $Ord\ (\alpha s\ !\ k)$ $Ord\ (\omega\text{-}sum\ (take\ k\ \alpha s))$ **if** $k < length\ \alpha s$ **for** *k*
using *that* ‹*list.set* $\alpha s \subseteq ON$›
by (*auto simp*: *dual-order.trans set-take-subset elim*: *ON-imp-Ord*)
define *E* **where** $E \equiv \lambda k.\ lift\ (\omega\text{-}sum\ (take\ k\ \alpha s))\ (\omega{\uparrow}(\alpha s!k))$
define *L* **where** $L \equiv map\ (\lambda k.\ \varphi\ `\ (elts\ (E\ k)))\ [0..<length\ \alpha s]$
have *lengthL* [*simp*]: $length\ L = length\ \alpha s$
by (*simp add*: *L-def*)
have *in-elts-E-less*: $elts\ (E\ k') \ll elts\ (E\ k)$ **if** $k'<k$ $k < length\ \alpha s$ **for** $k\ k'$
— The ordinals have been partitioned into disjoint intervals
proof −
have *ordω*: $Ord\ (\omega \uparrow \alpha s\ !\ k')$
using *that* **by** *auto*
from *that id-take-nth-drop* [*of* k' *take k* αs]
obtain *l* **where** $take\ k\ \alpha s = take\ k'\ \alpha s\ @\ (\alpha s!k')\ \#\ l$
by (*simp add*: *min-def*)
then show *?thesis*
using *that* **unfolding** *E-def lift-def less-sets-def*
by (*auto dest*!: *OrdmemD* [*OF ordω*] *intro*: *less-le-trans*)
qed
have *elts-E*: $elts\ (E\ k) \subseteq elts\ (\omega\text{-}sum\ \alpha s)$
if $k < length\ \alpha s$ **for** *k*
proof −

```
    have ω ↑ (αs!k) ≤ ω-sum (drop k αs)
      by (metis that Cons-nth-drop-Suc ω-sum-Cons add-le-cancel-left0)
    then have (+) (ω-sum (take k αs)) ‘ elts (ω ↑ (αs!k)) ⊆ elts (ω-sum (take k
αs) + ω-sum (drop k αs))
      by blast
    also have ... = elts (ω-sum αs)
      using ω-sum-take-drop by auto
    finally show ?thesis
      by (simp add: lift-def E-def)
  qed
  have ω-sum-in-tpD: ω-sum (take k αs) + γ ∈ elts (tp D)
    if γ ∈ elts (ω ↑ αs!k) k < length αs for γ k
    using elts-E lift-def that tpD-eq by (auto simp: E-def)
  have Ord-φ: Ord (φ (ω-sum (take k αs) + γ))
    if γ ∈ elts (ω ↑ αs!k) k < length αs for γ k
    using ω-sum-in-tpD ‹D ⊆ ON› bij-φ bij-betw-imp-surj-on that by fastforce
  define π where π ≡ λk. ((λy. odiff y (ω-sum (take k αs))) ∘ ordermap D VWF)
    — mapping the segments of D into some power of ω
  have bij-π: bij-betw (π k) (φ ‘ elts (E k)) (elts (ω ↑ (αs!k)))
    if k < length αs for k
   using that by (auto simp: bij-betw-def π-def E-def inj-on-def lift-def image-comp
ω-sum-in-tpD φ-cancel-right that)
  have π-iff: π k x < π k y ⟷ (x,y) ∈ VWF
    if x ∈ φ ‘ elts (E k) y ∈ φ ‘ elts (E k) k < length αs for k x y
    using that
   by (auto simp: π-def E-def lift-def ω-sum-in-tpD φ-cancel-right Ord-φ φ-less-iff)
  have tp-E-eq [simp]: tp (φ ‘ elts (E k)) = ω↑(αs!k)
    if k: k < length αs for k
    by (metis Ord-ω Ord-oexp π-iff bij-π ord(1) ordertype-VWF-eq-iff replacement
small-elts that)
  have tp-L-eq [simp]: tp (L!k) = ω↑(αs!k) if k < length αs for k
    by (simp add: L-def that)
  have UL-eq-D: ⋃ (list.set L) = D
  proof (rule antisym)
    show ⋃ (list.set L) ⊆ D
      by (force simp: L-def tpD-eq bij-betw-apply [OF bij-φ] dest: elts-E)
    show D ⊆ ⋃ (list.set L)
    proof
      fix δ
      assume δ ∈ D
      then have ordermap D VWF δ ∈ elts (ω-sum αs)
        by (metis ‹small D› ordermap-in-ordertype tpD-eq)
      then show δ ∈ ⋃ (list.set L)
        using ‹δ ∈ D› φ-cancel-left in-elts-ω-sum
        by (fastforce simp: L-def E-def image-iff lift-def)
    qed
  qed
  show thesis
  proof
```

show *indecomposable* (*tp X*) **if** $X \in$ *list.set L* **for** X
using *that* **by** (*auto simp*: *L-def indecomposable-ω-power*)
next
fix M
assume $M \subseteq D$ **and** $*$: $\bigwedge X.\ X \in$ *list.set L* $\Longrightarrow$ *tp X* $\leq$ *tp* ($M \cap X$)
show *tp M* = *tp D*
proof (*rule antisym*)
show *tp M* $\leq$ *tp D*
by (*simp add*: ‹$M \subseteq D$› ‹*small D*› *ordertype-VWF-mono*)
define σ **where** $\sigma \equiv \lambda X.$ *inv-into* ($M \cap X$) (*ordermap* ($M \cap X$) *VWF*)
— The bijection from an ω-power into the appropriate segment of M
have *bij-σ*: *bij-betw* (σ *X*) (*elts* (*tp* ($M \cap X$))) ($M \cap X$) **for** *X*
unfolding *σ-def*
by (*meson* ‹$M \subseteq D$› ‹*small D*› *bij-betw-inv-into inf-le1 ordermap-bij smaller-than-small total-on-VWF wf-VWF*)
have *Ord-σ*: *Ord* (σ *X* α) **if** $\alpha \in$ *elts* (*tp* ($M \cap X$)) **for** α *X*
using ‹$D \subseteq ON$› ‹$M \subseteq D$› *bij-betw-apply* [*OF bij-σ*] *that* **by** *blast*
have *σ-cancel-right*: $\bigwedge\gamma$ *X*. $\gamma \in$ *elts* (*tp* ($M \cap X$)) $\Longrightarrow$ *ordermap* ($M \cap X$) *VWF* (σ *X* γ) = γ
by (*metis σ-def f-inv-into-f ordermap-surj subsetD*)
have *smM*: *small* ($M \cap X$) **for** *X*
by (*meson* ‹$M \subseteq D$› ‹*small D*› *inf-le1 subset-iff-less-eq-V*)
have $\exists\, k <$ *length* $\alpha s.\ \gamma \in$ *elts* (*E k*) **if** γ: $\gamma \in$ *elts* (*tp D*) **for** γ
proof −
obtain *X* **where** $X \in$ *set L* **and** *X*: $\varphi\ \gamma \in X$
by (*metis UL-eq-D γ Union-iff φ-def in-mono inv-into-into ordermap-surj*)
then have $\exists\, k <$ *length* $\alpha s.\ \gamma \in$ *elts* (*E k*) $\wedge$ *X* = φ ‘ *elts* (*E k*)
apply (*clarsimp simp*: *L-def*)
by (*metis γ φ-cancel-right elts-E in-mono tpD-eq*)
then show *?thesis*
by *blast*
qed
then obtain *K* **where** *K*: $\bigwedge\gamma.\ \gamma \in$ *elts* (*tp D*) $\Longrightarrow$ *K* $\gamma <$ *length* $\alpha s \wedge \gamma \in$ *elts* (*E* (*K* γ))
by *metis* — The index from *tp D* to the appropriate segment number
define ψ **where** $\psi \equiv \lambda d.\ \sigma$ (*L* ! *K* (*ordermap D VWF d*)) (π (*K* (*ordermap D VWF d*)) *d*)
show *tp D* $\leq$ *tp M*
proof (*rule ordertype-inc-le*)
show *small D small M*
using ‹$M \subseteq D$› ‹*small D*› *subset-iff-less-eq-V* **by** *auto*
next
fix d' *d*
assume *xy*: $d' \in D$ $d \in D$ **and** (d',*d*) $\in$ *VWF*
then have $d' < d$
using *ON-imp-Ord* ‹$D \subseteq ON$› **by** *auto*
let *?γ′* = *ordermap D VWF d′*
let *?γ* = *ordermap D VWF d*

have *len'*: $K\ ?\gamma' < \mathit{length}\ \alpha s$ **and** *elts'*: $?\gamma' \in \mathit{elts}\ (E\ (K\ ?\gamma'))$
and *len*: $K\ ?\gamma < \mathit{length}\ \alpha s$ **and** *elts*: $?\gamma \in \mathit{elts}\ (E\ (K\ ?\gamma))$
using K ‹$d' \in D$› ‹$d \in D$› **by** (*auto simp*: ‹*small D*› *ordermap-in-ordertype*)
have *Ord-σL*: $\mathit{Ord}\ (\sigma\ (L!k)\ (\pi\ k\ d))$ **if** $d \in \varphi\ `\ \mathit{elts}\ (E\ k)\ k < \mathit{length}\ \alpha s$ **for** $k\ d$
by (*metis* (*mono-tags*) ∗ *Ord-σ bij-π bij-betw-apply lengthL nth-mem that tp-L-eq vsubsetD*)
have $?\gamma' < ?\gamma$
by (*simp add*: ‹$(d', d) \in \mathit{VWF}$› ‹*small D*› *ordermap-mono-less xy*)
then have $K\ ?\gamma' \leq K\ ?\gamma$
using *elts' elts in-elts-E-less* **by** (*meson leI len' less-asym less-sets-def*)
then consider (*less*) $K\ ?\gamma' < K\ ?\gamma$ | (*equal*) $K\ ?\gamma' = K\ ?\gamma$
by *linarith*
then have $\sigma\ (L\ !\ K\ ?\gamma')\ (\pi\ (K\ ?\gamma')\ d') < \sigma\ (L\ !\ K\ ?\gamma)\ (\pi\ (K\ ?\gamma)\ d)$
proof *cases*
case *less*
obtain †: $\sigma\ (L\ !\ K\ ?\gamma')\ (\pi\ (K\ ?\gamma')\ d') \in M \cap L\ !\ K\ ?\gamma'\ \sigma\ (L\ !\ K\ ?\gamma)\ (\pi\ (K\ ?\gamma)\ d) \in M \cap L\ !\ K\ ?\gamma$
using *elts' elts len' len* ∗ [*THEN vsubsetD*]
by (*metis lengthL φ-cancel-left bij-π bij-σ bij-betw-imp-surj-on imageI nth-mem tp-L-eq xy*)
then have *ordermap D VWF* $(\sigma\ (L\ !\ K\ ?\gamma')\ (\pi\ (K\ ?\gamma')\ d')) \in \mathit{elts}\ (E\ (K\ ?\gamma'))$ *ordermap D VWF* $(\sigma\ (L\ !\ K\ ?\gamma)\ (\pi\ (K\ ?\gamma)\ d)) \in \mathit{elts}\ (E\ (K\ ?\gamma))$
using *len' len elts-E tpD-eq*
by (*fastforce simp*: *L-def φ-cancel-right*)+
then have *ordermap D VWF* $(\sigma\ (L\ !\ K\ ?\gamma')\ (\pi\ (K\ ?\gamma')\ d')) <$ *ordermap D VWF* $(\sigma\ (L\ !\ K\ ?\gamma)\ (\pi\ (K\ ?\gamma)\ d))$
using *in-elts-E-less len less* **by** (*meson less-sets-def*)
moreover have $\sigma\ (L\ !\ K\ ?\gamma')\ (\pi\ (K\ ?\gamma')\ d') \in D\ \sigma\ (L\ !\ K\ ?\gamma)\ (\pi\ (K\ ?\gamma)\ d) \in D$
using ‹$M \subseteq D$› † **by** *auto*
ultimately show *?thesis*
by (*metis* ‹*small D*› *φ-cancel-left φ-less-iff ordermap-in-ordertype*)
next
case *equal*
have *σ-less*: $\bigwedge X\ \gamma\ \delta.\ [\![\gamma < \delta;\ \gamma \in \mathit{elts}\ (tp\ (M \cap X));\ \delta \in \mathit{elts}\ (tp\ (M \cap X))]\!] \Longrightarrow \sigma\ X\ \gamma < \sigma\ X\ \delta$
by (*metis* ‹$D \subseteq ON$› ‹$M \subseteq D$› *σ-def dual-order.trans inv-ordermap-VWF-strict-mono-iff le-infI1 smM*)
have $\pi\ (K\ ?\gamma)\ d' < \pi\ (K\ ?\gamma)\ d$
by (*metis equal* ‹$(d', d) \in \mathit{VWF}$› *φ-cancel-left π-iff elts elts' imageI len xy*)
then show *?thesis*
unfolding *equal*
by (*metis* ∗ [*THEN vsubsetD*] *lengthL φ-cancel-left σ-less bij-π bij-betw-imp-surj-on elts elts' image-eqI len local.equal nth-mem tp-L-eq xy*)
qed

```
      moreover have Ord (σ (L ! K ?γ') (π (K ?γ') d')) Ord (σ (L ! K ?γ) (π
(K ?γ) d))
        using Ord-σL φ-cancel-left elts len elts' len' xy by fastforce+
      ultimately show (ψ d', ψ d) ∈ VWF
        by (simp add: ψ-def)
    next
      show ψ ' D ⊆ M
      proof (clarsimp simp: ψ-def)
        fix d
        assume d ∈ D
        let ?γ = ordermap D VWF d
        have len: K ?γ < length αs and elts: ?γ ∈ elts (E (K ?γ))
          using K ‹d ∈ D› by (auto simp: ‹small D› ordermap-in-ordertype)
        have π (K ?γ) d ∈ elts (tp (L! (K ?γ)))
          using bij-π [OF len] ‹d ∈ D›
          by (metis φ-cancel-left bij-betw-apply elts imageI len tp-L-eq)
        then show σ (L ! K (ordermap D VWF d)) (π (K (ordermap D VWF
d)) d) ∈ M
          by (metis * lengthL Int-iff bij-σ bij-betw-apply len nth-mem vsubsetD)
      qed
    qed auto
  qed
 qed (simp add: UL-eq-D)
qed
```

The "remark" of Erds and E. C. Milner, Canad. Math. Bull. Vol. 17 (2), 1974

```
proposition indecomposable-imp-Ex-less-sets:
  assumes indec: indecomposable α and α ≥ ω
    and A: tp A = α small A A ⊆ ON
    and x ∈ A and A1: tp A1 = α A1 ⊆ A
  obtains A2 where tp A2 = α A2 ⊆ A1 {x} ≪ A2
proof −
  have Ord α
    using indec indecomposable-imp-Ord by blast
  have Limit α
    by (meson ω-gt1 assms indec indecomposable-imp-Limit less-le-trans)
  define φ where φ ≡ inv-into A (ordermap A VWF)
  then have bij-φ: bij-betw φ (elts α) A
    using A bij-betw-inv-into down ordermap-bij by blast
  have bij-om: bij-betw (ordermap A VWF) A (elts α)
    using A down ordermap-bij by blast
  define γ where γ ≡ ordermap A VWF x
  have γ: γ ∈ elts α
    unfolding γ-def using A ‹x ∈ A› down by auto
  then have Ord γ
    using Ord-in-Ord ‹Ord α› by blast
  define B where B ≡ φ ' (elts (succ γ))
  have B: {y ∈ A. ordermap A VWF y ≤ ordermap A VWF x} ⊆ B
```

```
    apply (clarsimp simp add: B-def γ-def image-iff φ-def)
   by (metis Ord-linear Ord-ordermap OrdmemD bij-betw-inv-into-left bij-om leD)
  show thesis
  proof
   have small A1
    by (meson ‹small A› A1 smaller-than-small)
   then have tp (A1 − B) ≤ tp A1
    by (simp add: ordertype-VWF-mono)
   moreover have tp (A1 − B) ≥ α
   proof −
    have ¬ (α ≤ tp B)
     unfolding B-def
    proof (subst ordertype-VWF-inc-eq)
     show elts (succ γ) ⊆ ON
      by (auto simp: γ-def ordertype-VWF-inc-eq intro: Ord-in-Ord)
     have sub: elts (succ γ) ⊆ elts α
      using Ord-trans γ ‹Ord α› by auto
     then show φ ‘ elts (succ γ) ⊆ ON
      using ‹A ⊆ ON› bij-φ bij-betw-imp-surj-on by blast
     have succ γ ∈ elts α
      using γ Limit-def ‹Limit α› by blast
     with A sub show φ u < φ v
      if u ∈ elts (succ γ) and v ∈ elts (succ γ) and u < v for u v
      by (metis φ-def inv-ordermap-VWF-strict-mono-iff subsetD that)
     show ¬ α ≤ tp (elts (succ γ))
       by (metis Limit-def Ord-succ γ ‹Limit α› ‹Ord γ› mem-not-refl order-
type-eq-Ord vsubsetD)
    qed auto
    then show ?thesis
     using indecomposable-ordertype-ge [OF indec, of A1 B] ‹small A1› A1 by
(auto simp: B-def)
   qed
   ultimately show tp (A1 − B) = α
    using A1 by blast
   have {x} ≪ (A − B)
    using assms B
    apply (clarsimp simp: less-sets-def φ-def subset-iff)
   by (metis Ord-not-le VWF-iff-Ord-less less-V-def order-refl ordermap-mono-less
trans-VWF wf-VWF)
   with ‹A1 ⊆ A› show {x} ≪ (A1 − B) by auto
  qed auto
qed
```

the main theorem, from which they derive the headline result

```
theorem Erdos-Milner-aux:
  assumes part: partn-lst-VWF α [k, γ] 2
   and indec: indecomposable α and k > 1 Ord γ and β: β ∈ elts ω1
  shows partn-lst-VWF (α∗β) [ord-of-nat (2∗k), min γ (ω∗β)] 2
proof (cases α≤1 ∨ β=0)
```

```
  case True
  have Ord α
    using indec indecomposable-def by blast
  show ?thesis
  proof (cases β=0)
    case True then show ?thesis
      by (simp add: partn-lst-triv0 [where i=1])
  next
    case False
    then consider (0) α=0 | (1) α=1
     by (metis Ord-0 OrdmemD True ‹Ord α› mem-0-Ord one-V-def order.antisym
succ-le-iff)
    then show ?thesis
    proof cases
      case 0
      with part show ?thesis
        by (force simp add: partn-lst-def nsets-empty-iff less-Suc-eq)
    next
      case 1
      then obtain Ord β
        by (meson ON-imp-Ord Ord-ω1 True β elts-subset-ON)
      then obtain i where i < Suc (Suc 0) [ord-of-nat k, γ] ! i ≤ α
        using partn-lst-VWF-nontriv [OF part] 1 by auto
      then have γ ≤ 1
        using ‹α=1› ‹k > 1› by (fastforce simp: less-Suc-eq)
      then have min γ (ω*β) ≤ 1
       by (metis Ord-1 Ord-ω Ord-linear-le Ord-mult ‹Ord β› min-def order-trans)
      then show ?thesis
       using False by (auto simp: True ‹Ord β› ‹β≠0› ‹α=1› intro!: partn-lst-triv1
[where i=1])
    qed
  qed
next
  case False
  then have α ≥ ω
   by (meson Ord-1 Ord-not-less indec indecomposable-def indecomposable-imp-Limit
omega-le-Limit)
  have 0 ∈ elts β β ≠ 0
    using False Ord-ω1 Ord-in-Ord β mem-0-Ord by blast+
  show ?thesis
    unfolding partn-lst-def
  proof clarsimp
    fix f
    assume f ∈ [elts (α*β)]^2 → {..<Suc (Suc 0)}
    then have f: f ∈ [elts (α*β)]^2 → {..<2::nat}
      by (simp add: eval-nat-numeral)
    obtain ord [iff]: Ord α Ord β Ord (α*β)
     using Ord-ω1 Ord-in-Ord β indec indecomposable-imp-Ord Ord-mult by blast
    have *: False
```

if *i* [*rule-format*]: $\forall H.\ tp\ H = ord\text{-}of\text{-}nat\ (2{*}k) \longrightarrow H \subseteq elts\ (\alpha{*}\beta) \longrightarrow \neg\ f\ `\ [H]^2 \subseteq \{0\}$
and *ii* [*rule-format*]: $\forall H.\ tp\ H = \gamma \longrightarrow H \subseteq elts\ (\alpha{*}\beta) \longrightarrow \neg\ f\ `\ [H]^2 \subseteq \{1\}$
and *iii* [*rule-format*]: $\forall H.\ tp\ H = (\omega{*}\beta) \longrightarrow H \subseteq elts\ (\alpha{*}\beta) \longrightarrow \neg\ f\ `\ [H]^2 \subseteq \{1\}$
proof −
have *Ak0*: $\exists X \in [A]^k.\ f\ `\ [X]^2 \subseteq \{0\}$ — remark (8) about $A \subseteq S$
if *A-αβ*: $A \subseteq elts\ (\alpha{*}\beta)$ **and** *ot*: $tp\ A \geq \alpha$ **for** *A*
proof −
let *?g* = *inv-into A* (*ordermap A VWF*)
have *small A*
using *down that* **by** *auto*
then have *inj-g*: *inj-on ?g* (*elts α*)
by (*meson inj-on-inv-into less-eq-V-def ordermap-surj ot subset-trans*)
have *om-A-less*: $\bigwedge x\ y.\ [\![x \in A;\ y \in A;\ x < y]\!] \Longrightarrow ordermap\ A\ VWF\ x < ordermap\ A\ VWF\ y$
using *ord*
by (*meson A-αβ Ord-in-Ord VWF-iff-Ord-less* ‹*small A*› *in-mono ordermap-mono-less trans-VWF wf-VWF*)
have *α-sub*: *elts α* ⊆ *ordermap A VWF ` A*
by (*metis* ‹*small A*› *elts-of-set less-eq-V-def ordertype-def ot replacement*)
have *g*: *?g* ∈ *elts α* → *elts* (*α*∗*β*)
by (*meson A-αβ Pi-I' α-sub inv-into-into subset-eq*)
then have *fg*: $f \circ (\lambda X.\ ?g\ `\ X) \in [elts\ \alpha]^2 \to \{..<2\}$
by (*rule nsets-compose-image-funcset* [*OF f - inj-g*])
obtain *i H* **where** $i < 2\ H \subseteq elts\ \alpha$
and *ot-eq*: *tp H* = [*k*,*γ*]!*i* (*f* ∘ (*λX. ?g ` X*)) ` (*nsets H 2*) ⊆ {*i*}
using *ii partn-lst-E* [*OF part fg*] **by** (*auto simp*: *eval-nat-numeral*)
then consider (*0*) *i=0* | (*1*) *i=1*
by *linarith*
then show *?thesis*
proof *cases*
case *0*
then have *f* ` [*inv-into A* (*ordermap A VWF*) ` *H*]^2 ⊆ {*0*}
using *ot-eq* ‹*H* ⊆ *elts α*› *α-sub* **by** (*auto simp*: *nsets-def* [*of - k*] *inj-on-inv-into elim*!: *nset-image-obtains*)
moreover have *finite H* ∧ *card H* = *k*
using *0 ot-eq* ‹*H* ⊆ *elts α*› *down* **by** (*simp add*: *finite-ordertype-eq-card*)
then have *inv-into A* (*ordermap A VWF*) ` *H* ∈ [*A*]^k
using ‹*H* ⊆ *elts α*› *α-sub* **by** (*auto simp*: *nsets-def* [*of - k*] *card-image inj-on-inv-into inv-into-into*)
ultimately show *?thesis*
by *blast*
next
case *1*
have *gH*: *?g ` H* ⊆ *elts* (*α*∗*β*)
by (*metis A-αβ α-sub* ‹*H* ⊆ *elts α*› *image-subsetI inv-into-into subset-eq*)
have *g-less*: *?g x < ?g y* **if** *x < y x* ∈ *elts α y* ∈ *elts α* **for** *x y*

using *Pi-mem* [*OF g*] *ord that*
by (*meson A-αβ Ord-in-Ord Ord-not-le* ‹*small A*› *dual-order.trans elts-subset-ON inv-ordermap-VWF-mono-le ot vsubsetD*)
have [*simp*]: *tp* (*?g* ‘ *H*) = *tp H*
by (*meson* ‹*H* ⊆ *elts* α› *ord down dual-order.trans elts-subset-ON gH g-less ordertype-VWF-inc-eq subsetD*)
show *?thesis*
using *ii* [*of ?g* ‘ *H*] *subset-inj-on* [*OF inj-g* ‹*H* ⊆ *elts* α›] *ot-eq 1*
by (*auto simp*: *gH elim*!: *nset-image-obtains*)
qed
qed
define *K* **where** *K* ≡ λ*i x*. {*y* ∈ *elts* (α∗β). *x* ≠ *y* ∧ *f*{*x*,*y*} = *i*}
have *small-K*: *small* (*K i x*) **for** *i x*
by (*simp add*: *K-def*)
define *KI* **where** *KI* ≡ λ*i X*. (⋂ *x*∈*X*. *K i x*)
have *KI-disj-self*: *X* ∩ *KI i X* = {} **for** *i X*
by (*auto simp*: *KI-def K-def*)
define *M* **where** *M* ≡ λ*D* 𝔄 *x*. {ν::*V*. ν ∈ *D* ∧ *tp* (*K 1 x* ∩ 𝔄 ν) ≥ α}
have *M-sub-D*: *M D* 𝔄 *x* ⊆ *D* **for** *D* 𝔄 *x*
by (*auto simp*: *M-def*)
have *small-M* [*simp*]: *small* (*M D* 𝔄 *x*) **if** *small D* **for** *D* 𝔄 *x*
by (*simp add*: *M-def that*)
have *9*: *tp* {*x* ∈ *A*. *tp* (*M D* 𝔄 *x*) ≥ *tp D*} ≥ α (**is** *ordertype ?AD* - ≥ α)
if *inD*: *indecomposable* (*tp D*) **and** *D*: *D* ⊆ *elts* β **and** *A*: *A* ⊆ *elts* (α∗β) **and** *tpA*: *tp A* = α
and 𝔄: 𝔄 ∈ *D* → {*X*. *X* ⊆ *elts* (α∗β) ∧ *tp X* = α} **for** *D A* 𝔄
— remark (9), assuming an indecomposable order type
proof (*rule ccontr*)
define *A′* **where** *A′* ≡ {*x* ∈ *A*. ¬ *tp* (*M D* 𝔄 *x*) ≥ *tp D*}
have *small* [*iff*]: *small A small D*
using *A D down* **by** (*auto simp*: *M-def*)
have *small-𝔄*: *small* (𝔄 δ) **if** δ ∈ *D* **for** δ
using *that* 𝔄 **by** (*auto simp*: *Pi-iff subset-iff-less-eq-V*)
assume *not-α-le*: ¬ α ≤ *tp* {*x* ∈ *A*. *tp* (*M D* 𝔄 *x*) ≥ *tp D*}
moreover
obtain *small A small A′ A′* ⊆ *A* **and** *A′-sub*: *A′* ⊆ *elts* (α∗β)
using *A′-def A down* **by** *auto*
moreover have *A′* = *A* − *?AD*
by (*force simp*: *A′-def*)
ultimately have *A′-ge*: *tp A′* ≥ α
by (*metis* (*no-types*, *lifting*) *dual-order.refl indec indecomposable-ordertype-eq mem-Collect-eq subsetI tpA*)
obtain *X* **where** *X* ⊆ *A′ finite X card X* = *k* **and** *fX0*: *f* ‘ [*X*]^2 ⊆ {*0*}
using *Ak0* [*OF A′-sub A′-ge*] **by** (*auto simp*: *nsets-def* [*of* - *k*])
then have ‡: ¬ *tp* (*M D* 𝔄 *x*) ≥ *tp D* **if** *x* ∈ *X* **for** *x*
using *that* **by** (*auto simp*: *A′-def*)
obtain *x* **where** *x* ∈ *X*
using ‹*card X* = *k*› ‹*k*>*1*› **by** *fastforce*
have ¬ *D* ⊆ (⋃ *x*∈*X*. *M D* 𝔄 *x*)

proof
assume *not*: $D \subseteq (\bigcup x{\in}X.\ M\ D\ \mathfrak{A}\ x)$
have $\exists\, X{\in}M\ D\ \mathfrak{A}\ `\ X.\ tp\ D \leq tp\ X$
proof (*rule indecomposable-ordertype-finite-ge* [*OF inD*])
show $M\ D\ \mathfrak{A}\ `\ X \neq \{\}$
using *A′-def A′-ge not not-α-le* **by** *auto*
show *small* $(\bigcup\ (M\ D\ \mathfrak{A}\ `\ X))$
using ‹*finite X*› **by** (*simp add*: *finite-imp-small*)
qed (*use* ‹*finite X*› *not* **in** *auto*)
then show *False*
by (*simp add*: ‡)
qed
then obtain ν **where** $\nu \in D$ **and** ν: $\nu \notin (\bigcup\ x{\in}X.\ M\ D\ \mathfrak{A}\ x)$
by *blast*
define $\mathcal{A}$ **where** $\mathcal{A} \equiv \{KI\ 0\ X \cap \mathfrak{A}\ \nu, \bigcup x{\in}X.\ K\ 1\ x \cap \mathfrak{A}\ \nu,\ X \cap \mathfrak{A}\ \nu\}$
have $\alpha\beta$: $X \subseteq elts\ (\alpha{*}\beta)\ \mathfrak{A}\ \nu \subseteq elts\ (\alpha{*}\beta)$
using *A′-sub* ‹$X \subseteq A'$› $\mathfrak{A}$ ‹$\nu \in D$› **by** *auto*
then have $KI\ 0\ X \cup (\bigcup x{\in}X.\ K\ 1\ x) \cup X = elts\ (\alpha{*}\beta)$
using ‹$x \in X$› *f* **by** (*force simp*: *K-def KI-def Pi-iff less-2-cases-iff*)
with $\alpha\beta$ **have** $\mathfrak{A}\nu$-$\mathcal{A}$: *finite* $\mathcal{A}\ \mathfrak{A}\ \nu \subseteq \bigcup \mathcal{A}$
by (*auto simp*: $\mathcal{A}$-*def*)
then have $\neg\ tp\ (K\ 1\ x \cap \mathfrak{A}\ \nu) \geq \alpha$ **if** $x \in X$ **for** x
using *that* ‹$\nu \in D$› ν ‹$k > 1$› ‹*card X* = *k*› **by** (*fastforce simp*: *M-def*)
moreover have *sm-K1*: *small* $(\bigcup x{\in}X.\ K\ 1\ x \cap \mathfrak{A}\ \nu)$
by (*meson Finite-V Int-lower2* ‹$\nu \in D$› ‹*finite X*› *small-*$\mathfrak{A}$ *small-UN smaller-than-small*)
ultimately have *not1*: $\neg\ tp\ (\bigcup x{\in}X.\ K\ 1\ x \cap \mathfrak{A}\ \nu) \geq \alpha$
using ‹*finite X*› ‹$x \in X$› *indecomposable-ordertype-finite-ge* [*OF indec*, *of* $(\lambda x.\ K\ 1\ x \cap \mathfrak{A}\ \nu)\ `\ X$] **by** *blast*
moreover have $\neg\ tp\ (X \cap \mathfrak{A}\ \nu) \geq \alpha$
using ‹*finite X*› ‹$\alpha \geq \omega$›
by (*meson finite-Int mem-not-refl ordertype-VWF-ω vsubsetD*)
moreover have $\alpha \leq tp\ (\mathfrak{A}\ \nu)$
using $\mathfrak{A}$ ‹$\nu \in D$› *small-*$\mathfrak{A}$ **by** *fastforce+*
moreover have *small* $(\bigcup\ \mathcal{A})$
using ‹$\nu \in D$› *small-*$\mathfrak{A}$ **by** (*fastforce simp*: $\mathcal{A}$-*def intro*: *smaller-than-small sm-K1*)
ultimately have *K0*$\mathfrak{A}$-*ge*: $tp\ (KI\ 0\ X \cap \mathfrak{A}\ \nu) \geq \alpha$
using *indecomposable-ordertype-finite-ge* [*OF indec* $\mathfrak{A}\nu$-$\mathcal{A}$] **by** (*auto simp*: $\mathcal{A}$-*def*)
have $\mathfrak{A}\nu$: $\mathfrak{A}\ \nu \subseteq elts\ (\alpha{*}\beta)\ tp\ (\mathfrak{A}\ \nu) = \alpha$
using ‹$\nu \in D$› $\mathfrak{A}$ **by** *blast+*
then obtain *Y* **where** *Ysub*: $Y \subseteq KI\ 0\ X \cap \mathfrak{A}\ \nu$ **and** *finite Y card Y* = *k* **and** *fY0*: $f\ `\ [Y]^2 \subseteq \{0\}$
using *Ak0* [*OF - K0*$\mathfrak{A}$-*ge*] **by** (*auto simp*: *nsets-def* [*of - k*])
have †: $X \cap Y = \{\}$
using *Ysub KI-disj-self* **by** *blast*
then have *card* $(X \cup Y) = 2{*}k$
by (*simp add*: ‹*card X* = *k*› ‹*card Y* = *k*› ‹*finite X*› ‹*finite Y*›

card-Un-disjoint)
    moreover have $X \cup Y \subseteq elts\ (\alpha * \beta)$
      using *A'-sub* ‹$X \subseteq A'$› $\mathfrak{A}\nu(1)$ ‹$Y \subseteq KI\ 0\ X \cap \mathfrak{A}\ \nu$› **by** *auto*
    moreover have $f\ `\ [X \cup Y]^2 \subseteq \{0\}$
     using *fX0 fY0 Ysub* **by** (*auto simp*: † *nsets-disjoint-2 image-Un image-UN KI-def K-def*)
    ultimately show *False*
      using *i* ‹*finite X*› ‹*finite Y*› *ordertype-VWF-finite-nat* **by** *auto*
  qed
  have *IX*: $tp\ \{x \in A.\ tp\ (M\ D\ \mathfrak{A}\ x) \geq tp\ D\} \geq \alpha$
    if *D*: $D \subseteq elts\ \beta$ **and** *A*: $A \subseteq elts\ (\alpha * \beta)$ **and** *tpA*: $tp\ A = \alpha$
      and $\mathfrak{A}$: $\mathfrak{A} \in D \to \{X.\ X \subseteq elts\ (\alpha * \beta) \wedge tp\ X = \alpha\}$ **for** $D\ A\ \mathfrak{A}$
      — remark (9) for any order type
  proof −
    obtain L **where** *UL*: $\bigcup(List.set\ L) \subseteq D$
      and *indL*: $\bigwedge X.\ X \in List.set\ L \Longrightarrow indecomposable\ (tp\ X)$
       and *eqL*: $\bigwedge M.\ [\![M \subseteq D; \bigwedge X.\ X \in List.set\ L \Longrightarrow tp\ (M \cap X) \geq tp\ X]\!] \Longrightarrow tp\ M = tp\ D$
      using *ord* **by** (*metis strong-ordertype-eq D order-refl*)
    obtain A'' **where** A'': $A'' \subseteq A\ tp\ A'' \geq \alpha$
      and $\bigwedge x\ X.\ [\![x \in A''; X \in List.set\ L]\!] \Longrightarrow tp\ (M\ D\ \mathfrak{A}\ x \cap X) \geq tp\ X$
      using *UL indL*
    proof (*induction L arbitrary*: *thesis*)
      case (*Cons X L*)
      then obtain A'' **where** A'': $A'' \subseteq A\ tp\ A'' \geq \alpha$ **and** $X \subseteq D$
       and *ge-X*: $\bigwedge x\ X.\ [\![x \in A''; X \in List.set\ L]\!] \Longrightarrow tp\ (M\ D\ \mathfrak{A}\ x \cap X) \geq tp\ X$ **by** *auto*
      then have *tp-A''*: $tp\ A'' = \alpha$
        by (*metis A antisym down ordertype-VWF-mono tpA*)
      have *ge-α*: $tp\ \{x \in A''.\ tp\ (M\ X\ \mathfrak{A}\ x) \geq tp\ X\} \geq \alpha$
        by (*rule 9*) (*use A A'' tp-A'' Cons.prems* ‹$D \subseteq elts\ \beta$› ‹$X \subseteq D$› $\mathfrak{A}$ **in** *auto*)
      let *?A* = $\{x \in A''.\ tp\ (M\ D\ \mathfrak{A}\ x \cap X) \geq tp\ X\}$
      have *M-eq*: $M\ D\ \mathfrak{A}\ x \cap X = M\ X\ \mathfrak{A}\ x$ **if** $x \in A''$ **for** x
        using *that* ‹$X \subseteq D$› **by** (*auto simp*: *M-def*)
      show *thesis*
      proof (*rule Cons.prems*)
        show $\alpha \leq tp\ ?A$
          using *ge-α* **by** (*simp add*: *M-eq cong*: *conj-cong*)
       show $tp\ Y \leq tp\ (M\ D\ \mathfrak{A}\ x \cap Y)$ **if** $x \in ?A\ Y \in list.set\ (X\ \#\ L)$ **for** $x\ Y$
          using *that ge-X* **by** *force*
        qed (*use A'' in auto*)
    qed (*use tpA in auto*)
    then have *tp-M-ge*: $tp\ (M\ D\ \mathfrak{A}\ x) \geq tp\ D$ **if** $x \in A''$ **for** x
      using *eqL that* **by** (*auto simp*: *M-def*)
    have $\alpha \leq tp\ A''$
      by (*simp add*: *A''*)
    also have $\ldots \leq tp\ \{x \in A''.\ tp\ (M\ D\ \mathfrak{A}\ x) \geq tp\ D\}$
      by (*metis* (*mono-tags*, *lifting*) *tp-M-ge eq-iff mem-Collect-eq subsetI*)

```
    also have ... ≤ tp {x ∈ A. tp D ≤ tp (M D 𝔄 x)}
      by (rule ordertype-mono) (use A'' A down in auto)
    finally show ?thesis .
  qed
  have IX': tp {x ∈ A'. tp (K 1 x ∩ A) ≥ α} ≥ α
    if A: A ⊆ elts (α*β) tp A = α and A': A' ⊆ elts (α*β) tp A' = α for A A'
  proof −
    have ‡: α ≤ tp (K 1 t ∩ A) if 1 ≤ tp {ν. ν = 0 ∧ α ≤ tp (K 1 t ∩ A)} for
t
      using that
      by (metis Collect-empty-eq less-eq-V-0-iff ordertype-empty zero-neq-one)
    have tp {x ∈ A'. 1 ≤ tp {ν. ν = 0 ∧ α ≤ tp (K 1 x ∩ A)}}
        ≤ tp {x ∈ A'. α ≤ tp (K 1 x ∩ A)}
      by (rule ordertype-mono) (use ‡ A' in ‹auto simp: down›)
    then show ?thesis
      using IX [of {0} A' λx. A] that ‹0 ∈ elts β› by (force simp: M-def)
  qed

  have 10: ∃x0 ∈ A. ∃g ∈ elts β → elts β. strict-mono-on (elts β) g ∧ (∀ν ∈
F. g ν = ν)
                                  ∧ (∀ν ∈ elts β. tp (K 1 x0 ∩ 𝔄 (g ν)) ≥ α)
    if F: finite F F ⊆ elts β
      and A: A ⊆ elts (α*β) tp A = α
      and 𝔄: 𝔄 ∈ elts β → {X. X ⊆ elts (α*β) ∧ tp X = α}
    for F A 𝔄
  proof −
    define p where p ≡ card F
    have β ∉ F
      using that by auto
    then obtain ι :: nat ⇒ V where bijι: bij-betw ι {..p} (insert β F) and
monι: strict-mono-on {..p} ι
    using ZFC-Cardinals.ex-bij-betw-strict-mono-card [of insert β F] elts-subset-ON
‹Ord β› F
      by (simp add: p-def lessThan-Suc-atMost) blast
    have less-ι-I: ι k < ι l if k < l l ≤ p for k l
      using monι that by (auto simp: strict-mono-on-def)
    then have less-ι-D: k < l if ι k < ι l k ≤ p for k l
      by (metis less-asym linorder-neqE-nat that)
    have Ord-ι: Ord (ι k) if k ≤ p for k
    by (metis (no-types, lifting) ON-imp-Ord atMost-iff insert-subset mem-Collect-eq
order-trans ‹F ⊆ elts β› bijι bij-betwE elts-subset-ON ‹Ord β› that)
    have le-ι0 [simp]: ⋀j. j ≤ p ⟹ ι 0 ≤ ι j
      by (metis eq-refl leI le-0-eq less-ι-I less-imp-le)
    have le-ι: ι i ≤ ι (j − Suc 0) if i < j j ≤ p for i j
    proof (cases i)
      case 0 then show ?thesis
        using le-ι0 that by auto
    next
      case (Suc i') then show ?thesis
```

by (*metis* (*no-types*, *opaque-lifting*) *Suc-pred le-less less-Suc-eq less-Suc-eq-0-disj less-ι-I not-less-eq that*)
qed

have [*simp*]: $\iota\ p = \beta$
proof −
obtain k **where** k: $\iota\ k = \beta\ k \leq p$
by (*meson atMost-iff bijι bij-betw-iff-bijections insertI1*)
then have $k = p \vee k < p$
by *linarith*
then show *?thesis*
using *bijι ord k that*(*2*)
by (*metis OrdmemD atMost-iff bij-betw-iff-bijections insert-iff leD less-ι-D order-refl subsetD*)
qed

have *F-imp-Ex*: $\exists k < p.\ \xi = \iota\ k$ **if** $\xi \in F$ **for** ξ
proof −
obtain k **where** k: $k \leq p\ \xi = \iota\ k$
by (*metis* ‹$\xi \in F$› *atMost-iff bijι bij-betw-def imageE insert-iff*)
then show *?thesis*
using ‹$\beta \notin F$› ‹$\iota\ p = \beta$› *le-imp-less-or-eq that* **by** *blast*
qed
have *F-imp-ge*: $\xi \geq \iota\ 0$ **if** $\xi \in F$ **for** ξ
using *F-imp-Ex* [*OF that*] **by** (*metis dual-order.order-iff-strict le0 less-ι-I*)
define D **where** $D \equiv \lambda k.$ (*if k=0 then* $\{..<\iota\ 0\}$ *else* $\{\iota\ (k-1)<..<\iota\ k\}$) $\cap$ *elts* β
have $D\beta$: $D\ k \subseteq$ *elts* β **for** k
by (*auto simp*: *D-def*)
then have *small-D* [*simp*]: *small* $(D\ k)$ **for** k
by (*meson down*)
have *M-Int-D*: $M\ (\textit{elts}\ \beta)\ \mathfrak{A}\ x \cap D\ k = M\ (D\ k)\ \mathfrak{A}\ x$ **if** $k \leq p$ **for** $x\ k$
using $D\beta$ **by** (*auto simp*: *M-def*)
have *ι-le-if-D*: $\iota\ k \leq \mu$ **if** $\mu \in D\ (\textit{Suc}\ k)$ **for** $\mu\ k$
using *that* **by** (*simp add*: *D-def order.order-iff-strict*)
have *mono-D*: $D\ i \ll D\ j$ **if** $i < j\ j \leq p$ **for** $i\ j$
proof (*cases j*)
case (*Suc j′*)
with *that* **show** *?thesis*
apply (*simp add*: *less-sets-def D-def Ball-def*)
by (*metis One-nat-def diff-Suc-1 le-ι less-le-trans less-trans*)
qed (*use that* **in** *auto*)
then have *disjnt-DD*: *disjnt* $(D\ i)\ (D\ j)$ **if** $i \neq j\ i \leq p\ j \leq p$ **for** $i\ j$
by (*meson disjnt-sym less-linear less-sets-imp-disjnt that*)
have *UN-D-eq*: $(\bigcup l \leq k.\ D\ l) = \{..<\iota\ k\} \cap (\textit{elts}\ \beta - F)$ **if** $k \leq p$ **for** k
using *that*
proof (*induction k*)
case *0*
then show *?case*

by (*auto simp*: *D-def F-imp-ge leD*)
next
case (*Suc k*)
have $D\ (Suc\ k) \cup \{..<\iota\ k\} \cap (elts\ \beta - F) = \{..<\iota\ (Suc\ k)\} \cap (elts\ \beta - F)$
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* ⊆ *?rhs*
using *Suc.prems*
by (*auto simp*: *D-def if-split-mem2 intro*: *less-ι-I less-trans dest*!: *less-ι-D F-imp-Ex*)
have $\bigwedge x.\ [\![x < \iota\ (Suc\ k);\ x \in elts\ \beta;\ x \notin F;\ \iota\ k \leq x]\!] \Longrightarrow \iota\ k < x$
using *Suc.prems* ‹$F \subseteq elts\ \beta$› *bijι le-imp-less-or-eq*
by (*fastforce simp*: *bij-betw-iff-bijections*)
then show *?rhs* ⊆ *?lhs*
using *Suc.prems* **by** (*auto simp*: *D-def Ord-not-less Ord-in-Ord* [*OF* ‹*Ord β*›] *Ord-ι if-split-mem2*)
qed
then
show *?case*
using *Suc* **by** (*simp add*: *atMost-Suc*)
qed
have *β-decomp*: $elts\ \beta = F \cup (\bigcup k \leq p.\ D\ k)$
using ‹$F \subseteq elts\ \beta$› *OrdmemD* [*OF* ‹*Ord β*›] **by** (*auto simp*: *UN-D-eq*)
define *βidx* **where** $\beta idx \equiv \lambda\nu.\ @k.\ \nu \in D\ k \wedge k \leq p$
have *βidx*: $\nu \in D\ (\beta idx\ \nu) \wedge \beta idx\ \nu \leq p$ **if** $\nu \in elts\ \beta - F$ **for** ν
using *that* **by** (*force simp*: *βidx-def β-decomp intro*: *someI-ex del*: *conjI*)
have *any-imp-βidx*: $k = \beta idx\ \nu$ **if** $\nu \in D\ k\ k \leq p$ **for** $k\ \nu$
proof (*rule ccontr*)
assume *non*: $k \neq \beta idx\ \nu$
have $\nu \notin F$
using *that UN-D-eq* **by** *auto*
then show *False*
using *disjnt-DD* [*OF non*] **by** (*metis Dβ Diff-iff βidx disjnt-iff subsetD that*)
qed
have $\exists A'.\ A' \subseteq A \wedge tp\ A' = \alpha \wedge (\forall x \in A'.\ F \subseteq M\ (elts\ \beta)\ \mathfrak{A}\ x)$
using *F*
proof *induction*
case (*insert ν F*)
then obtain A' **where** $A' \subseteq A$ **and** *A′*: $A' \subseteq elts\ (\alpha{*}\beta)\ tp\ A' = \alpha$ **and** *FN*: $\bigwedge x.\ x \in A' \Longrightarrow F \subseteq M\ (elts\ \beta)\ \mathfrak{A}\ x$
using *A*(*1*) **by** *auto*
define A'' **where** $A'' \equiv \{x \in A'.\ \alpha \leq tp\ (K\ 1\ x \cap \mathfrak{A}\ \nu)\}$
have $\nu \in elts\ \beta\ F \subseteq elts\ \beta$
using *insert* **by** *auto*
note *ordertype-eq-Ord* [*OF* ‹*Ord β*›, *simp*]
show *?case*
proof (*intro exI conjI*)
show $A'' \subseteq A$

```
      using ‹A' ⊆ A› by (auto simp: A''-def)
    show tp A'' = α
    proof (rule antisym)
      show tp A'' ≤ α
        using ‹A'' ⊆ A› down ordertype-VWF-mono A by blast
      have 𝔄 ν ⊆ elts (α*β) tp (𝔄 ν) = α
        using 𝔄 ‹ν ∈ elts β› by auto
      then show α ≤ tp A''
        using IX' [OF - - A'] by (simp add: A''-def)
    qed
    show ∀x∈A''. insert ν F ⊆ M (elts β) 𝔄 x
      using A''-def FN M-def ‹ν ∈ elts β› by blast
  qed
qed (use A in auto)
then obtain A' where A': A' ⊆ A tp A' = α and FN: ⋀x. x ∈ A' ⟹ F
⊆ M (elts β) 𝔄 x
  by metis
have False
  if *: ⋀x0 g. ⟦x0 ∈ A; g ∈ elts β → elts β; strict-mono-on (elts β) g⟧
        ⟹ (∃ν∈F. g ν ≠ ν) ∨ (∃ν∈elts β. tp (K 1 x0 ∩ 𝔄 (g ν)) < α)
proof −
  { fix x      — construction of the monotone map g mentioned above
    assume x ∈ A'
    with A' have x ∈ A by blast
    have ∃k. k ≤ p ∧ tp (M (D k) 𝔄 x) < tp (D k) (is ?P)
    proof (rule ccontr)
      assume ¬ ?P
      then have le: tp (D k) ≤ tp (M (D k) 𝔄 x) if k ≤ p for k
        by (meson Ord-linear2 Ord-ordertype that)
       have ∃f∈D k → M (D k) 𝔄 x. inj-on f (D k) ∧ (strict-mono-on (D
k) f)
        if k ≤ p for k
        using le [OF that] that VWF-iff-Ord-less
        apply (clarsimp simp: ordertype-le-ordertype strict-mono-on-def)
       by (metis (full-types) Dβ M-sub-D Ord-in-Ord PiE VWF-iff-Ord-less
ord(2) subsetD)
      then obtain h where fun-h: ⋀k. k ≤ p ⟹ h k ∈ D k → M (D k) 𝔄 x
        and inj-h: ⋀k. k ≤ p ⟹ inj-on (h k) (D k)
        and mono-h: ⋀k x y. k ≤ p ⟹ strict-mono-on (D k) (h k)
        by metis
      then have fun-hD: ⋀k. k ≤ p ⟹ h k ∈ D k → D k
        by (auto simp: M-def)
      have h-increasing: ν ≤ h k ν
        if k ≤ p ν ∈ D k for k ν
            by (meson Dβ Ord-mono-imp-increasing ord dual-order.trans
elts-subset-ON fun-hD mono-h that)
      define g where g ≡ λν. if ν ∈ F then ν else h (βidx ν) ν
      have [simp]: g ν = ν if ν ∈ F for ν
        using that by (auto simp: g-def)
```

have *fun-g*: $g \in elts\ \beta \to elts\ \beta$
proof (*rule Pi-I*)
fix x **assume** $x \in elts\ \beta$
then have $x \in D\ (\beta idx\ x)\ \beta idx\ x \leq p$ **if** $x \notin F$
using *that* **by** (*auto simp*: βidx)
then show $g\ x \in elts\ \beta$
by (*metis fun-h Dβ M-sub-D ‹x ∈ elts β› PiE g-def subsetD*)
qed
have *h-in-D*: $h\ (\beta idx\ \nu)\ \nu \in D\ (\beta idx\ \nu)$ **if** $\nu \notin F\ \nu \in elts\ \beta$ **for** ν
using βidx *fun-hD that* **by** *fastforce*
have *1*: $\iota\ k < h\ (\beta idx\ \nu)\ \nu$
if $k < p$ **and** ν: $\nu \notin F\ \nu \in elts\ \beta$ **and** $\iota\ k < \nu$ **for** $k\ \nu$
by (*meson that h-in-D* [*OF* ν] βidx *DiffI h-increasing order-less-le-trans*)
moreover have *2*: $h\ (\beta idx\ \mu)\ \mu < \iota\ k$
if μ: $\mu \notin F\ \mu \in elts\ \beta$ **and** $k < p\ \mu < \iota\ k$ **for** $\mu\ k$
proof −
have $\beta idx\ \mu \leq k$
proof (*rule ccontr*)
assume ¬ $\beta idx\ \mu \leq k$
then have $k < \beta idx\ \mu$
by *linarith*
then show *False*
using ι*-le-if-D* βidx *that* **by** (*metis Diff-iff Suc-pred le0 leD le-ι le-less-trans*)
qed
then show *?thesis*
using *that h-in-D* [*OF* μ]
by (*smt* (*verit*, *best*) *Int-lower1 UN-D-eq UN-I atMost-iff lessThan-iff less-imp-le subset-eq*)
qed
moreover have $h\ (\beta idx\ \mu)\ \mu < h\ (\beta idx\ \nu)\ \nu$
if μ: $\mu \notin F\ \mu \in elts\ \beta$ **and** ν: $\nu \notin F\ \nu \in elts\ \beta$ **and** $\mu < \nu$ **for** $\mu\ \nu$
proof −
have *le*: $\beta idx\ \mu \leq \beta idx\ \nu$ **if** $\iota\ (\beta idx\ \mu - Suc\ 0) < h\ (\beta idx\ \mu)\ \mu\ h\ (\beta idx\ \nu)\ \nu < \iota\ (\beta idx\ \nu)$
by (*metis 2 DiffI* $\beta idx\ \mu\ \nu$ *‹μ < ν› order.strict-trans h-increasing leI le-ι order-less-asym order-less-le-trans that*)
have $h\ 0\ \mu < h\ 0\ \nu$ **if** $\beta idx\ \mu = 0\ \beta idx\ \nu = 0$
using *that mono-h* **unfolding** *strict-mono-on-def*
by (*metis Diff-iff* $\beta idx\ \mu\ \nu$ *‹μ < ν›*)
moreover have $h\ 0\ \mu < h\ (\beta idx\ \nu)\ \nu$
if $0 < \beta idx\ \nu\ h\ 0\ \mu < \iota\ 0$ **and** $\iota\ (\beta idx\ \nu - Suc\ 0) < h\ (\beta idx\ \nu)\ \nu$
by (*meson DiffI* $\beta idx\ \nu$ *le-ι le-less-trans less-le-not-le that*)
moreover have $\beta idx\ \nu \neq 0$
if $0 < \beta idx\ \mu\ h\ 0\ \nu < \iota\ 0\ \iota\ (\beta idx\ \mu - Suc\ 0) < h\ (\beta idx\ \mu)\ \mu$
using *le le-0-eq that* **by** *fastforce*
moreover have $h\ (\beta idx\ \mu)\ \mu < h\ (\beta idx\ \nu)\ \nu$
if $\iota\ (\beta idx\ \mu - Suc\ 0) < h\ (\beta idx\ \mu)\ \mu\ h\ (\beta idx\ \nu)\ \nu < \iota\ (\beta idx\ \nu)$
$h\ (\beta idx\ \mu)\ \mu < \iota\ (\beta idx\ \mu)\ \iota\ (\beta idx\ \nu - Suc\ 0) < h\ (\beta idx\ \nu)\ \nu$

```
            using mono-h unfolding strict-mono-on-def
            by (metis le Diff-iff βidx μ ν ‹μ < ν› le-ι le-less le-less-trans that)
          ultimately show ?thesis
              using h-in-D [OF μ] h-in-D [OF ν] by (simp add: D-def split:
if-split-asm)
        qed
        ultimately have sm-g: strict-mono-on (elts β) g
          by (auto simp: g-def strict-mono-on-def dest!: F-imp-Ex)
        have False if ν ∈ elts β and ν: tp (K 1 x ∩ 𝔄 (g ν)) < α for ν
        proof −
          have F ⊆ M (elts β) 𝔄 x
            by (meson FN ‹x ∈ A'›)
          then have False if tp (K (Suc 0) x ∩ 𝔄 ν) < α ν ∈ F
            using that by (auto simp: M-def)
          moreover have False if tp (K (Suc 0) x ∩ 𝔄 (g ν)) < α ν ∈ D k k
≤ p ν ∉ F for k
          proof −
            have h (βidx ν) ν ∈ M (D (βidx ν)) 𝔄 x
              using fun-h βidx ‹ν ∈ elts β› ‹ν ∉ F› by auto
            then show False
              using that by (simp add: M-def g-def leD)
          qed
          ultimately show False
            using ‹ν ∈ elts β› ν by (force simp: β-decomp)
        qed
        then show False
          using ∗ [OF ‹x ∈ A› fun-g sm-g] by auto
      qed
      then have ∃ l. l ≤ p ∧ tp (M (elts β) 𝔄 x ∩ D l) < tp (D l)
        using M-Int-D by auto
    }
    then obtain l where lp: ⋀x. x ∈ A' ⟹ l x ≤ p
      and lless: ⋀x. x ∈ A' ⟹ tp (M (elts β) 𝔄 x ∩ D (l x)) < tp (D (l x))
      by metis
    obtain A'' L where A'' ⊆ A' and A'': A'' ⊆ elts (α∗β) tp A'' = α and
lL: ⋀x. x ∈ A'' ⟹ l x = L
    proof −
      have eq: A' = (⋃ i≤p. {x ∈ A'. l x = i})
        using lp by auto
      have ∃ X∈(λi. {x ∈ A'. l x = i}) ` {..p}. α ≤ tp X
      proof (rule indecomposable-ordertype-finite-ge [OF indec])
        show small (⋃ i≤p. {x ∈ A'. l x = i})
          by (metis A'(1) A(1) eq down smaller-than-small)
      qed (use A' eq in auto)
      then show thesis
      proof
        fix A''
        assume A'': A'' ∈ (λi. {x ∈ A'. l x = i}) ` {..p} and α ≤ tp A''
        then obtain L where L: ⋀x. x ∈ A'' ⟹ l x = L
```

```
        by auto
      have A'' ⊆ A'
        using A'' by force
      then have tp A'' ≤ tp A'
        by (meson A' A down order-trans ordertype-VWF-mono)
      with ‹α ≤ tp A''› have tp A'' = α
        using A'(2) by auto
      moreover have A'' ⊆ elts (α*β)
        using A' A ‹A'' ⊆ A'› by auto
      ultimately show thesis
        using L that [OF ‹A'' ⊆ A'›] by blast
    qed
  qed
  have 𝔄D: 𝔄 ∈ D L → {X. X ⊆ elts (α*β) ∧ tp X = α}
    using 𝔄 Dβ by blast
  have α: α ≤ tp {x ∈ A''. tp (D L) ≤ tp (M (D L) 𝔄 x)}
    using IX [OF Dβ A'' 𝔄D] by simp
  have M (elts β) 𝔄 x ∩ D L = M (D L) 𝔄 x for x
    using Dβ by (auto simp: M-def)
  then have tp (M (D L) 𝔄 x) < tp (D L) if x ∈ A'' for x
    using lless that ‹A'' ⊆ A'› lL by force
  then have [simp]: {x ∈ A''. tp (D L) ≤ tp (M (D L) 𝔄 x)} = {}
    using leD by blast
  show False
    using α ‹α ≥ ω› by simp
qed
then show ?thesis
  by (meson Ord-linear2 Ord-ordertype ‹Ord α›)
qed
let ?U = UNIV :: nat set
define μ where μ ≡ fst ∘ from-nat-into (elts β × ?U)
define q where q ≡ to-nat-on (elts β × ?U)
have co-βU: countable (elts β × ?U)
  by (simp add: β less-ω1-imp-countable)
moreover have elts β × ?U ≠ {}
  using ‹0 ∈ elts β› by blast
ultimately have range (from-nat-into (elts β × ?U)) = (elts β × ?U)
  by (metis range-from-nat-into)
then have μ-in-β [simp]: μ i ∈ elts β for i
  by (metis SigmaE μ-def comp-apply fst-conv range-eqI)

then have Ord-μ [simp]: Ord (μ i) for i
  using Ord-in-Ord by blast

have inf-βU: infinite (elts β × ?U)
  using ‹0 ∈ elts β› finite-cartesian-productD2 by auto

have 11 [simp]: μ (q (ν,n)) = ν if ν ∈ elts β for ν n
  by (simp add: μ-def q-def that co-βU)
```

have *range-μ* [*simp*]: *range μ = elts β*
by (*auto simp*: *image-iff*) (*metis 11*)
have [*simp*]: *KI i {} = UNIV KI i* (*insert a X*) = *K i a* ∩ *KI i X* **for** *i a X*
by (*auto simp*: *KI-def*)
define Φ **where** Φ ≡ λ*n*::*nat*. λ𝔄 *x*. (∀ν ∈ *elts β*. 𝔄 ν ⊆ *elts* (α∗β) ∧ *tp* (𝔄 ν) = α)
∧ *x* ‘ {..<*n*} ⊆ *elts* (α∗β)
∧ (⋃ν ∈ *elts β*. 𝔄 ν) ⊆ *KI 1* (*x* ‘ {..<*n*})
∧ *strict-mono-sets* (*elts β*) 𝔄
define Ψ **where** Ψ ≡ λ*n*::*nat*. λ*g* 𝔄 𝔄′ *xn*. *g* ∈ *elts β* → *elts β* ∧ *strict-mono-on* (*elts β*) *g*
∧ (∀ *i*≤*n*. *g* (μ *i*) = μ *i*)
∧ (∀ν ∈ *elts β*. 𝔄′ ν ⊆ *K 1 xn* ∩ 𝔄 (*g* ν))
∧ {*xn*} ≪ (𝔄′ (μ *n*)) ∧ *xn* ∈ 𝔄 (μ *n*)
let *?𝔄0* = λν. *plus* (α ∗ ν) ‘ *elts* α
have *base*: Φ *0 ?𝔄0 x* **for** *x*
by (*auto simp*: Φ-*def add-mult-less add-mult-less-add-mult ordertype-image-plus strict-mono-sets-def less-sets-def*)
have *step*: *Ex* (λ(*g*,𝔄′,*xn*). Ψ *n g* 𝔄 𝔄′ *xn* ∧ Φ (*Suc n*) 𝔄′ (*x*(*n*:=*xn*))) **if** Φ *n* 𝔄 *x* **for** *n* 𝔄 *x*
proof −
have 𝔄: ⋀ν. ν ∈ *elts β* ⟹ 𝔄 ν ⊆ *elts* (α∗β) ∧ *tp* (𝔄 ν) = α
and *x*: *x* ‘ {..<*n*} ⊆ *elts* (α∗β)
and *sub*: ⋃ (𝔄 ‘ *elts β*) ⊆ *KI* (*Suc 0*) (*x* ‘ {..<*n*})
and *sm*: *strict-mono-sets* (*elts β*) 𝔄
and μβ: μ ‘ {..*n*} ⊆ *elts β* **and** 𝔄*sub*: 𝔄 (μ *n*) ⊆ *elts* (α∗β)
and 𝔄*fun*: 𝔄 ∈ *elts β* → {*X*. *X* ⊆ *elts* (α∗β) ∧ *tp X* = α}
using *that* **by** (*auto simp*: Φ-*def*)
then obtain *xn g* **where** *xn*: *xn* ∈ 𝔄 (μ *n*) **and** *g*: *g* ∈ *elts β* → *elts β*
and *sm-g*: *strict-mono-on* (*elts β*) *g* **and** *g-μ*: ∀ν ∈ μ‘{..*n*}. *g* ν = ν
and *g-α*: ∀ν ∈ *elts β*. α ≤ *tp* (*K 1 xn* ∩ 𝔄 (*g* ν))
using *10* [*OF* - μβ 𝔄*sub* - 𝔄*fun*] **by** (*auto simp*: 𝔄)
have *tp1*: *tp* (*K 1 xn* ∩ 𝔄 (*g* ν)) = α **if** ν ∈ *elts β* **for** ν
by (*metis antisym Int-lower2 PiE* 𝔄 *down g g-α ordertype-VWF-mono that*)
have *tp2*: *tp* (𝔄 (μ *n*)) = α
by (*auto simp*: 𝔄)
obtain *small* (𝔄 (μ *n*)) 𝔄 (μ *n*) ⊆ *ON*
by (*meson* 𝔄*sub ord down elts-subset-ON subset-trans*)
then obtain *A2* **where** *A2*: *tp A2* = α *A2* ⊆ *K 1 xn* ∩ 𝔄 (μ *n*) {*xn*} ≪ *A2*
using *indecomposable-imp-Ex-less-sets* [*OF indec* ‹α ≥ ω› *tp2*]
by (*metis μ-in-β atMost-iff image-eqI inf-le2 le-refl xn tp1 g-μ*)
then have *A2-sub*: *A2* ⊆ 𝔄 (μ *n*) **by** *simp*
let *?𝔄* = λν. *if* ν = μ *n then A2 else K 1 xn* ∩ 𝔄 (*g* ν)
have [*simp*]: ({..<*Suc n*} ∩ {*x*. *x* ≠ *n*}) = ({..<*n*})
by *auto*
have *K* (*Suc 0*) *xn* ∩ (⋃*x*∈*elts β* ∩ {ν. ν ≠ μ *n*}. 𝔄 (*g x*)) ⊆ *KI* (*Suc 0*) (*x* ‘ {..<*n*})

using *sub g* **by** (*auto simp*: *KI-def*)
moreover have *A2* ⊆ *KI* (*Suc 0*) (*x* ‘ {..<*n*}) *A2* ⊆ *elts* (α∗β) *xn* ∈ *elts* (α∗β)
using 𝔄*sub sub A2 xn* **by** *fastforce*+
moreover have *strict-mono-sets* (*elts* β) *?*𝔄
using *sm sm-g g g-μ A2-sub*
unfolding *strict-mono-sets-def strict-mono-on-def less-sets-def Pi-iff subset-iff Ball-def Bex-def image-iff*
by (*simp* (*no-asm-use*) *add*: *if-split-mem2*) (*smt order-refl*)
ultimately have Φ (*Suc n*) *?*𝔄 (*x*(*n* := *xn*))
using *tp1 x A2* **by** (*auto simp*: Φ*-def K-def*)
with *A2* **show** *?thesis*
by (*rule-tac x*=(*g*,*?*𝔄,*xn*) **in** *exI*) (*simp add*: Ψ*-def g sm-g g-μ xn*)
qed
define *G* **where** *G* ≡ λ*n* 𝔄 *x*. @(*g*,𝔄′,*x*′). ∃*xn*. Ψ *n g* 𝔄 𝔄′ *xn* ∧ *x*′ = (*x*(*n*:=*xn*)) ∧ Φ (*Suc n*) 𝔄′ *x*′
have *G*Φ: (λ(*g*,𝔄′,*x*′). Φ (*Suc n*) 𝔄′ *x*′) (*G n* 𝔄 *x*)
and *G*Ψ: (λ(*g*,𝔄′,*x*′). Ψ *n g* 𝔄 𝔄′ (*x*′ *n*)) (*G n* 𝔄 *x*) **if** Φ *n* 𝔄 *x* **for** *n* 𝔄 *x*
using *step* [*OF that*] **by** (*force simp*: *G-def dest*: *some-eq-imp*)+
define *H* **where** *H* ≡ *rec-nat* (*id*,*?*𝔄*0*,*undefined*) (λ*n* (*g0*,𝔄,*x0*). *G n* 𝔄 *x0*)
have (λ(*g*,𝔄,*x*). Φ *n* 𝔄 *x*) (*H n*) **for** *n*
unfolding *H-def* **by** (*induction n*) (*use G*Φ *base* **in** *fastforce*)+
then have *H-imp*-Φ: Φ *n* 𝔄 *x* **if** *H n* = (*g*,𝔄,*x*) **for** *g* 𝔄 *x n*
by (*metis case-prodD that*)
then have *H-imp*-Ψ: (λ(*g*,𝔄′,*x*′). *let* (*g0*,𝔄,*x*) = *H n in* Ψ *n g* 𝔄 𝔄′ (*x*′ *n*)) (*H* (*Suc n*)) **for** *n*
using *G*Ψ **by** (*fastforce simp*: *H-def split*: *prod.split*)
define *g* **where** *g* ≡ λ*n*. (λ(*g*,𝔄,*x*). *g*) (*H* (*Suc n*))
have *g*: *g n* ∈ *elts* β → *elts* β **and** *sm-g*: *strict-mono-on* (*elts* β) (*g n*)
and *13*: ⋀*i*. *i*≤*n* ⟹ *g n* (μ *i*) = μ *i* **for** *n*
using *H-imp*-Ψ [*of n*] **by** (*auto simp*: *g-def* Ψ*-def*)
define 𝔄 **where** 𝔄 ≡ λ*n*. (λ(*g*,𝔄,*x*). 𝔄) (*H n*)
define *x* **where** *x* ≡ λ*n*. (λ(*g*,𝔄,*x*). *x n*) (*H* (*Suc n*))
have *14*: 𝔄 (*Suc n*) ν ⊆ *K 1* (*x n*) ∩ 𝔄 *n* (*g n* ν) **if** ν ∈ *elts* β **for** ν *n*
using *H-imp*-Ψ [*of n*] *that* **by** (*force simp*: Ψ*-def* 𝔄*-def x-def g-def*)
then have *x14*: 𝔄 (*Suc n*) ν ⊆ 𝔄 *n* (*g n* ν) **if** ν ∈ *elts* β **for** ν *n*
using *that* **by** *blast*
have *15*: *x n* ∈ 𝔄 *n* (μ *n*) **and** *16*: {*x n*} ≪ (𝔄 (*Suc n*) (μ *n*)) **for** *n*
using *H-imp*-Ψ [*of n*] **by** (*force simp*: Ψ*-def* 𝔄*-def x-def*)+
have 𝔄-αβ: 𝔄 *n* ν ⊆ *elts* (α∗β) **if** ν ∈ *elts* β **for** ν *n*
using *H-imp*-Φ [*of n*] *that* **by** (*auto simp*: Φ*-def* 𝔄*-def split*: *prod.split*)
have *12*: *strict-mono-sets* (*elts* β) (𝔄 *n*) **for** *n*
using *H-imp*-Φ [*of n*] *that* **by** (*auto simp*: Φ*-def* 𝔄*-def split*: *prod.split*)
let *?Z* = *range x*
have *S-dec*: ⋃ (𝔄 (*m*+*k*) ‘ *elts* β) ⊆ ⋃ (𝔄 *m* ‘ *elts* β) **for** *k m*
by (*induction k*) (*use 14 g* **in** ‹*fastforce*+›)
have *x n* ∈ *K 1* (*x m*) **if** *m*<*n* **for** *m n*
proof −
have *x n* ∈ (⋃ν ∈ *elts* β. 𝔄 *n* ν)

```
      by (meson 15 UN-I μ-in-β)
    also have ... ⊆ (⋃ν ∈ elts β. 𝔄 (Suc m) ν)
      using S-dec [of Suc m] less-iff-Suc-add that by auto
    also have ... ⊆ K 1 (x m)
      using 14 by auto
    finally show ?thesis .
  qed
  then have f{x m, x n} = 1 if m<n for m n
    using that by (auto simp: K-def)
  then have Z-K1: f ‘ [?Z]^2 ⊆ {1}
    by (clarsimp simp: nsets-2-eq) (metis insert-commute less-linear)
  moreover have Z-sub: ?Z ⊆ elts (α*β)
    using 15 𝔄-αβ μ-in-β by blast
  moreover have tp ?Z ≥ ω * β
  proof −
    define 𝔤 where 𝔤 ≡ λi j x. wfrec (measure (λk. j−k)) (λ𝔤 k. if k<j then g
k (𝔤 (Suc k)) else x) i
    have 𝔤: 𝔤 i j x = (if i<j then g i (𝔤 (Suc i) j x) else x) for i j x
      by (simp add: 𝔤-def wfrec cut-apply)
    have 17: 𝔤 k j (μ i) = μ i if i ≤ k for i j k
      using wf-measure [of λk. j−k] that
      by (induction k rule: wf-induct-rule) (simp add: 13 𝔤 le-imp-less-Suc)
    have 𝔤-in-β: 𝔤 i j ν ∈ elts β if ν ∈ elts β for i j ν
      using wf-measure [of λk. j−k] that
    proof (induction i rule: wf-induct-rule)
      case (less i)
      with g show ?case by (force simp: 𝔤 [of i])
    qed
    then have 𝔤-fun: 𝔤 i j ∈ elts β → elts β for i j
      by simp
    have sm-𝔤: strict-mono-on (elts β) (𝔤 i j) for i j
      using wf-measure [of λk. j−k]
    proof (induction i rule: wf-induct-rule)
      case (less i)
      with sm-g show ?case
        by (auto simp: 𝔤 [of i] strict-mono-on-def 𝔤-in-β)
    qed
    have ∗: 𝔄 j (μ j) ⊆ 𝔄 i (𝔤 i j (μ j)) if i < j for i j
      using wf-measure [of λk. j−k] that
    proof (induction i rule: wf-induct-rule)
      case (less i)
      then have j − Suc i < j − i
        by (metis (no-types) Suc-diff-Suc lessI)
      with less 𝔤-in-β show ?case
        by (simp add: 𝔤 [of i]) (metis 17 Suc-lessI μ-in-β order-refl order-trans
x14)
    qed
    have le-iff: 𝔤 i j (μ j) ≤ μ i ⟷ μ j ≤ μ i for i j
      using sm-𝔤 unfolding strict-mono-on-def
```

```
    by (metis 17 Ord-in-Ord Ord-linear2 μ-in-β leD le-refl less-V-def ‹Ord β›)
  then have less-iff: 𝔤 i j (μ j) < μ i ⟷ μ j < μ i for i j
  by (metis (no-types, lifting) 17 μ-in-β less-V-def order-refl sm-𝔤 strict-mono-on-def)
  have eq-iff: 𝔤 i j (μ j) = μ i ⟷ μ j = μ i for i j
    by (metis eq-refl le-iff less-iff less-le)
  have μ-if-ne: μ m < μ n if mn: 𝔄 m (μ m) ≪ 𝔄 n (μ n) m ≠ n for m n
  proof −
    have xmn: x m < x n
      using 15 less-setsD that(1) by blast
    have Ord𝔤: Ord (𝔤 n m (μ m))
      using Ord-in-Ord 𝔤-in-β μ-in-β ord(2) by presburger
    have ¬ 𝔄 m (μ m) ≪ 𝔄 n (μ n) if μ n = μ m
      using ∗ 15 eq-iff that unfolding less-sets-def
      by (metis in-mono less-irrefl not-less-iff-gr-or-eq)
    moreover
    have 𝔄 n (μ n) ⊆ 𝔄 m (𝔤 m n (μ n)) ∨ 𝔄 m (μ m) ⊆ 𝔄 n (𝔤 n m (μ m))
      using ∗ mn
      by (meson antisym-conv3)
    then have False if μ n < μ m
      using strict-mono-setsD [OF 12] 15 xmn 𝔤-in-β μ-in-β that
      by (smt (verit, best) Ord𝔤 Ord-μ Ord-linear2 leD le-iff less-asym less-iff
less-setsD subset-iff)
    ultimately show μ m < μ n
      by (meson that(1) Ord-μ Ord-linear-lt)
  qed
  have 18: 𝔄 m (μ m) ≪ 𝔄 n (μ n) ⟷ μ m < μ n for m n
  proof (cases n m rule: linorder-cases)
    case less
    show ?thesis
    proof (intro iffI)
      assume μ m < μ n
      then have 𝔄 n (𝔤 n m (μ m)) ≪ 𝔄 n (μ n)
        by (metis 12 𝔤-in-β μ-in-β eq-iff le-iff less-V-def strict-mono-sets-def)
      then show 𝔄 m (μ m) ≪ 𝔄 n (μ n)
        by (meson ∗ less less-sets-weaken1)
    qed (use μ-if-ne less in blast)
  next
    case equal
    with 15 show ?thesis by auto
  next
    case greater
    show ?thesis
    proof (intro iffI)
      assume μ m < μ n
      then have 𝔄 m (μ m) ≪ (𝔄 m (𝔤 m n (μ n)))
        by (meson 12 Ord-in-Ord Ord-linear2 𝔤-in-β μ-in-β le-iff leD ord(2)
strict-mono-sets-def)
      then show 𝔄 m (μ m) ≪ 𝔄 n (μ n)
        by (meson ∗ greater less-sets-weaken2)
```

qed (*use* μ*-if-ne greater* **in** *blast*)
qed
have $\mathfrak{A}$*-increasing-*μ: $\mathfrak{A}\ n\ (\mu\ n) \subseteq \mathfrak{A}\ m\ (\mu\ m)$ **if** $m \leq n\ \mu\ m = \mu\ n$ **for** $m\ n$
by (*metis* $*$ *17 dual-order.order-iff-strict that*)
moreover have *INF*: *infinite* $\{n.\ n \geq m \wedge \mu\ m = \mu\ n\}$ **for** m
proof −
have *infinite* (*range* $(\lambda n.\ q\ (\mu\ m,\ n)))$
unfolding *q-def*
using *to-nat-on-infinite* [*OF co-*β*U inf-*β*U*] *finite-image-iff*
by (*simp add*: *finite-image-iff inj-on-def*)
moreover have (*range* $(\lambda n.\ q\ (\mu\ m,\ n))) \subseteq \{n.\ \mu\ m = \mu\ n\}$
using *11* [*of* $\mu\ m$] **by** *auto*
ultimately have *infinite* $\{n.\ \mu\ m = \mu\ n\}$
using *finite-subset* **by** *auto*
then have *infinite* $(\{n.\ \mu\ m = \mu\ n\} - \{..<m\})$
by *simp*
then show *?thesis*
by (*auto simp*: *finite-nat-set-iff-bounded Bex-def not-less*)
qed
let *?eqv* $= \lambda m.\ \{n.\ m \leq n \wedge \mu\ m = \mu\ n\}$
have *sm-x*: *strict-mono-on* (*?eqv m*) *x* **for** *m*
proof (*clarsimp simp*: *strict-mono-on-def*)
fix *n p*
assume $m \leq n\ \mu\ p = \mu\ n\ \mu\ m = \mu\ n\ n < p$
with *16* [*of n*] **show** $x\ n < x\ p$
by (*metis* $*$ *15 17 Suc-lessI insert-absorb insert-subset le-SucI less-sets-singleton1*)
qed
then have *inj-x*: *inj-on x* (*?eqv m*) **for** *m*
using *strict-mono-on-imp-inj-on* **by** *blast*
define *ZA* **where** $ZA \equiv \lambda m.\ ?Z \cap \mathfrak{A}\ m\ (\mu\ m)$
have *small-ZA* [*simp*]: *small* (*ZA m*) **for** *m*
by (*metis ZA-def inf-le1 small-image-nat smaller-than-small*)
have *19*: *tp* (*ZA m*) $\geq \omega$ **for** *m*
proof −
have x ‘ $\{n.\ m \leq n \wedge \mu\ m = \mu\ n\} \subseteq ZA\ m$
unfolding *ZA-def* **using** *15* $\mathfrak{A}$*-increasing-*μ **by** *blast*
then have *infinite* (*ZA m*)
using *INF* [*of m*] *finite-image-iff* [*OF inj-x*] **by** (*meson finite-subset*)
then show *?thesis*
by (*simp add*: *ordertype-infinite-ge-*ω)
qed
have $\exists f \in elts\ \omega \to ZA\ m.$ *strict-mono-on* (*elts* ω) f **for** m
proof −
obtain Z **where** $Z \subseteq ZA\ m\ tp\ Z = \omega$
by (*meson 19 Ord-*ω *le-ordertype-obtains-subset small-ZA*)
moreover have $ZA\ m \subseteq ON$
using *Ord-in-Ord* $\mathfrak{A}$*-*$\alpha\beta$ μ*-in-*β **unfolding** *ZA-def* **by** *blast*
ultimately show *?thesis*
by (*metis strict-mono-on-ordertype Pi-mono small-ZA smaller-than-small*

subset-iff)
qed
then obtain φ where φ: $\bigwedge m.\ \varphi\ m \in elts\ \omega \to ZA\ m$
and *sm-φ*: $\bigwedge m.$ *strict-mono-on* (*elts* ω) ($\varphi\ m$)
by *metis*
have *Ex*($\lambda(m,\nu).\ \nu \in elts\ \beta \wedge \gamma = \omega * \nu +$ *ord-of-nat* m) if $\gamma \in elts\ (\omega * \beta)$ for γ
using *that* by (*auto simp*: *mult* [*of* $\omega\ \beta$] *lift-def elts-ω*)
then obtain *split* where *split*: $\bigwedge\gamma.\ \gamma \in elts\ (\omega * \beta) \Longrightarrow$
($\lambda(m,\nu).\ \nu \in elts\ \beta \wedge \gamma = \omega * \nu +$ *ord-of-nat* m)(*split* γ)
by *meson*
have *split-eq* [*simp*]: *split* ($\omega * \nu +$ *ord-of-nat* m) = (m,ν) if $\nu \in elts\ \beta$ for $\nu\ m$
proof −
have [*simp*]: $\omega * \nu +$ *ord-of-nat* $m = \omega * \xi +$ *ord-of-nat* $n \longleftrightarrow \xi = \nu \wedge n = m$ if $\xi \in elts\ \beta$ for $\xi\ n$
by (*metis Ord-ω that Ord-mem-iff-less-TC mult-cancellation-lemma ord-of-nat-ω ord-of-nat-inject*)
show *?thesis*
using *split* [*of* $\omega{*}\nu + m$] *that* by (*auto simp*: *mult* [*of* $\omega\ \beta$] *lift-def cong*: *conj-cong*)
qed
define π where $\pi \equiv \lambda\gamma.$ ($\lambda(m,\nu).\ \varphi$ ($q(\nu,0)$) m)(*split* γ)
have *π-Pi*: $\pi \in elts\ (\omega * \beta) \to (\bigcup m.\ ZA\ m)$
using φ by (*fastforce simp*: *π-def mult* [*of* $\omega\ \beta$] *lift-def elts-ω*)
moreover have $(\bigcup m.\ ZA\ m) \subseteq ON$
unfolding *ZA-def* using $\mathfrak{A}$-$\alpha\beta$ *μ-in-β elts-subset-ON* by *blast*
ultimately have *Ord-π-Pi*: $\pi \in elts\ (\omega * \beta) \to ON$
by *fastforce*
show *tp ?Z* $\geq \omega * \beta$
proof −
have †: $(\bigcup m.\ ZA\ m) =$ *?Z*
using *15* by (*force simp*: *ZA-def*)
moreover
have *tp* (*elts* ($\omega * \beta$)) $\leq$ *tp* $(\bigcup m.\ ZA\ m)$
proof (*rule ordertype-inc-le*)
show π ‘ *elts* ($\omega * \beta$) $\subseteq$ $(\bigcup m.\ ZA\ m)$
using *π-Pi* by *blast*
next
fix *u v*
assume *x*: $u \in elts\ (\omega * \beta)$ and *y*: $v \in elts\ (\omega * \beta)$ and $(u,v) \in VWF$
then have $u{<}v$
by (*meson Ord-ω Ord-in-Ord Ord-mult VWF-iff-Ord-less ord*(*2*))
moreover
obtain $m\ \nu\ n\ \xi$ where *ueq*: $u = \omega * \nu +$ *ord-of-nat* m and ν: $\nu \in elts\ \beta$
and *veq*: $v = \omega * \xi +$ *ord-of-nat* n and ξ: $\xi \in elts\ \beta$
using *x y* by (*auto simp*: *mult* [*of* $\omega\ \beta$] *lift-def elts-ω*)
ultimately have $\nu \leq \xi$
by (*meson Ord-ω Ord-in-Ord Ord-linear2* ‹*Ord β*› *add-mult-less-add-mult*

less-asym ord-of-nat-ω)
consider (*eq*) $\nu = \xi$ | (*lt*) $\nu < \xi$
using ‹$\nu \leq \xi$› *le-neq-trans* **by** *blast*
then have $\pi\ u < \pi\ v$
proof *cases*
case *eq*
then have $m < n$
using *ueq veq* ‹$u<v$› **by** *simp*
then have $\varphi\ (q\ (\xi,\ 0))\ m < \varphi\ (q\ (\xi,\ 0))\ n$
using *sm-φ strict-mono-onD* **by** *blast*
then show *?thesis*
using *eq ueq veq* ν ‹$m < n$› **by** (*simp add*: π-*def*)
next
case *lt*
have $\varphi\ (q(\nu,0))\ m \in \mathfrak{A}\ (q(\nu,0))\ (\mu(q(\nu,0)))\ \varphi\ (q\ (\xi,0))\ n \in \mathfrak{A}\ (q(\xi,0))\ (\mu(q(\xi,0)))$
using φ **unfolding** *ZA-def* **by** *blast+*
then show *?thesis*
using *lt ueq veq* ν ξ *18* [*of* $q(\nu,0)$ $q(\xi,0)$]
by (*simp add*: π-*def less-sets-def*)
qed
then show $(\pi\ u,\ \pi\ v) \in VWF$
using π-*Pi* **by** (*metis Ord-π-Pi PiE VWF-iff-Ord-less x y mem-Collect-eq*)
qed (*use* † **in** *auto*)
ultimately show *?thesis* **by** *simp*
qed
qed
then obtain Z **where** $Z \subseteq$ *?Z tp* $Z = \omega * \beta$
by (*meson Ord-ω Ord-mult ord Z-sub down le-ordertype-obtains-subset*)
ultimately show *False*
using *iii* [*of Z*] **by** (*meson dual-order.trans image-mono nsets-mono*)
qed
have *False*
if *0*: $\forall H.\ tp\ H = ord\text{-}of\text{-}nat\ (2{*}k) \longrightarrow H \subseteq elts\ (\alpha{*}\beta) \longrightarrow \neg\ f\ `\ [H]^2 \subseteq \{0\}$
and *1*: $\forall H.\ tp\ H = min\ \gamma\ (\omega * \beta) \longrightarrow H \subseteq elts\ (\alpha{*}\beta) \longrightarrow \neg\ f\ `\ [H]^2 \subseteq \{1\}$
proof (*cases* $\omega{*}\beta \leq \gamma$)
case *True*
then have †: $\exists H' \subseteq H.\ tp\ H' = \omega * \beta$ **if** $tp\ H = \gamma$ *small H* **for** *H*
by (*metis Ord-ω Ord-ω1 Ord-in-Ord Ord-mult β le-ordertype-obtains-subset that*)
have [*simp*]: $min\ \gamma\ (\omega{*}\beta) = \omega{*}\beta$
by (*simp add*: *min-absorb2 that True*)
then show *?thesis*
using $*$ [*OF 0*] *1 True*
by *simp* (*meson* † *down image-mono nsets-mono subset-trans*)
next
case *False*
then have †: $\exists H' \subseteq H.\ tp\ H' = \gamma$ **if** $tp\ H = \omega * \beta$ *small H* **for** *H*

by (*metis Ord-linear-le Ord-ordertype ‹Ord γ› le-ordertype-obtains-subset that*)
then have $\gamma \leq \omega{*}\beta$
by (*meson Ord-ω Ord-ω1 Ord-in-Ord Ord-linear-le Ord-mult β ‹Ord γ› False*)
then have [*simp*]: *min γ* ($\omega{*}\beta$) = *γ*
by (*simp add*: *min-absorb1*)
then show *?thesis*
using ∗ [*OF 0*] *1 False*
by *simp* (*meson † down image-mono nsets-mono subset-trans*)
qed
then show ∃*i*<*Suc* (*Suc 0*). ∃*H*⊆*elts* (*α*∗*β*). *tp H* = [*ord-of-nat* (*2*∗*k*), *min γ* (*ω*∗*β*)] ! *i* ∧ *f* ‘ [*H*]^2 ⊆ {*i*}
by *force*
qed
qed

theorem *Erdos-Milner*:
assumes *ν*: *ν* ∈ *elts ω1*
shows *partn-lst-VWF* (*ω*↑(*1* + *ν* ∗ *n*)) [*ord-of-nat* (*2*^*n*), *ω*↑(*1*+*ν*)] *2*
proof (*induction n*)
case *0*
then show *?case*
using *partn-lst-VWF-degenerate* [*of 1 2*] **by** *simp*
next
case (*Suc n*)
have *Ord ν*
using *Ord-ω1 Ord-in-Ord assms* **by** *blast*
have *1*+*ν* ≤ *ν*+*1*
by (*simp add*: *‹Ord ν› one-V-def plus-Ord-le*)
then have [*simp*]: *min* (*ω* ↑ (*1* + *ν*)) (*ω* ∗ *ω* ↑ *ν*) = *ω* ↑ (*1*+*ν*)
by (*simp add*: *‹Ord ν› oexp-add min-def*)
have *ind*: *indecomposable* (*ω* ↑ (*1* + *ν* ∗ *ord-of-nat n*))
by (*simp add*: *‹Ord ν› indecomposable-ω-power*)
show *?case*
proof (*cases n* = *0*)
case *True*
then show *?thesis*
using *partn-lst-VWF-ω-2 ‹Ord ν› one-V-def* **by** *auto*
next
case *False*
then have *Suc 0* < *2* ^ *n*
using *less-2-cases not-less-eq* **by** *fastforce*
then have *partn-lst-VWF* (*ω* ↑ (*1* + *ν* ∗ *n*) ∗ *ω* ↑ *ν*) [*ord-of-nat* (*2* ∗ *2* ^ *n*), *ω* ↑ (*1* + *ν*)] *2*
using *Erdos-Milner-aux* [*OF Suc ind*, **where** *β* = *ω*↑*ν*] *‹Ord ν› ν*
by (*auto simp*: *countable-oexp*)
then show *?thesis*

```
    using ‹Ord ν› by (simp add: mult-succ mult.assoc oexp-add)
  qed
qed


corollary remark-3: partn-lst-VWF (ω↑(Suc(4∗k))) [4, ω↑(Suc(2∗k))] 2
  using Erdos-Milner [of 2∗k 2]
  apply (simp flip: ord-of-nat-mult ord-of-nat.simps)
  by (simp add: one-V-def)
```

Theorem 3.2 of Jean A. Larson, ibid.

```
corollary Theorem-3-2:
  fixes k n::nat
  shows partn-lst-VWF (ω↑(n∗k)) [ω↑n, ord-of-nat k] 2
proof (cases n=0 ∨ k=0)
  case True
  then show ?thesis
    by (auto intro: partn-lst-triv0 [where i=1] partn-lst-triv1 [where i=0])
next
  case False
  then have n > 0 k > 0
    by auto
  from ‹k > 0› less-exp [of ‹k − 1›] have ‹k ≤ 2 ^ (k − 1)›
    by (cases k) (simp-all add: less-eq-Suc-le)
   have PV: partn-lst-VWF (ω ↑ (1 + ord-of-nat (n−1) ∗ ord-of-nat (k−1)))
[ord-of-nat (2 ^ (k−1)), ω ↑ (1 + ord-of-nat (n−1))] 2
    using Erdos-Milner [of ord-of-nat (n−1) k−1] Ord-ω1 Ord-mem-iff-lt less-imp-le
by blast
  have k+n ≤ Suc (k ∗ n)
    using False not0-implies-Suc by fastforce
  then have 1 + (n − 1) ∗ (k − 1) ≤ n∗k
    using False by (auto simp: algebra-simps)
  then have (1 + ord-of-nat (n − 1) ∗ ord-of-nat (k − 1)) ≤ ord-of-nat(n∗k)
   by (metis (mono-tags, lifting) One-nat-def one-V-def ord-of-nat.simps ord-of-nat-add
ord-of-nat-mono-iff ord-of-nat-mult)
  then have x: ω ↑ (1 + ord-of-nat (n − 1) ∗ ord-of-nat (k − 1)) ≤ ω↑(n∗k)
    by (simp add: oexp-mono-le)
  then have partn-lst-VWF (ω↑(n∗k)) [ω ↑ (1 + ord-of-nat (n−1)), ord-of-nat (2
^ (k−1))] 2
   by (metis PV partn-lst-two-swap Partitions.partn-lst-greater-resource less-eq-V-def)
  then have partn-lst-VWF (ω↑(n∗k)) [ω ↑ n, ord-of-nat (2 ^ (k−1))] 2
    using ord-of-minus-1 [OF ‹n > 0›] by (simp add: one-V-def)
  then show ?thesis
    using ‹k ≤ 2 ^ (k − 1)›
    by (auto elim!: partn-lst-less simp add: less-Suc-eq)
qed

end
```

3 An ordinal partition theorem by Jean A. Larson

Jean A. Larson, A short proof of a partition theorem for the ordinal ω^ω. *Annals of Mathematical Logic*, 6:129–145, 1973.

theory *Omega-Omega*
imports *HOL−Library.Product-Lexorder Erdos-Milner*

begin

abbreviation *list-of* ≡ *sorted-list-of-set*

3.1 Cantor normal form for ordinals below $\omega \uparrow \omega$

Unlike *Cantor-sum*, there is no list of ordinal exponents, which are instead taken as consecutive. We obtain an order-isomorphism between $\omega \uparrow \omega$ and increasing lists of natural numbers (ordered lexicographically).

fun *omega-sum-aux* **where**
Nil: *omega-sum-aux 0 - = 0*
| *Suc*: *omega-sum-aux* (*Suc n*) [] = *0*
| *Cons*: *omega-sum-aux* (*Suc n*) (*m*#*ms*) = (ω↑*n*) ∗ (*ord-of-nat m*) + *omega-sum-aux n ms*

abbreviation *omega-sum* **where** *omega-sum ms* ≡ *omega-sum-aux* (*length ms*) *ms*

A normal expansion has no leading zeroes

inductive *normal*:: *nat list* ⇒ *bool* **where**
normal-Nil[*iff*]: *normal* []
| *normal-Cons*: *m* > *0* ⟹ *normal* (*m*#*ms*)

inductive-simps *normal-Cons-iff* [*simp*]: *normal* (*m*#*ms*)

lemma *omega-sum-0-iff* [*simp*]: *normal ns* ⟹ *omega-sum ns* = *0* ⟷ *ns* = []
by (*induction ns rule*: *normal.induct*) *auto*

lemma *Ord-omega-sum-aux* [*simp*]: *Ord* (*omega-sum-aux k ms*)
by (*induction rule*: *omega-sum-aux.induct*) *auto*

lemma *Ord-omega-sum*: *Ord* (*omega-sum ms*)
by *simp*

lemma *omega-sum-less-ωω* [*intro*]: *omega-sum ms* < ω↑ω
proof (*induction ms*)
case (*Cons m ms*)
have ω ↑ (*length ms*) ∗ *ord-of-nat m* ∈ *elts* (ω ↑ *Suc* (*length ms*))
using *Ord-mem-iff-lt* **by** *auto*
then have ω↑(*length ms*) ∗ *ord-of-nat m* ∈ *elts* (ω↑ω)
using *Ord-ord-of-nat oexp-mono-le omega-nonzero ord-of-nat-le-omega* **by** *blast*

with *Cons* **show** *?case*
by (*auto simp*: *mult-succ OrdmemD oexp-less indecomposableD indecomposable-ω-power*)
qed (*auto simp*: *zero-less-Limit*)

lemma *omega-sum-aux-less*: *omega-sum-aux k ms* $< \omega \uparrow k$
proof (*induction rule*: *omega-sum-aux.induct*)
case (*3 n m ms*)
have $\omega\uparrow n * ord\text{-}of\text{-}nat\ m + \omega\uparrow n < \omega\uparrow n * \omega$
by (*metis Ord-ord-of-nat ω-power-succ-gtr mult-succ oexp-succ ord-of-nat.simps*(*2*))
with *3* **show** *?case*
using *dual-order.strict-trans* **by** *force*
qed *auto*

lemma *omega-sum-less*: *omega-sum ms* $< \omega \uparrow$ (*length ms*)
by (*rule omega-sum-aux-less*)

lemma *omega-sum-ge*: $m \neq 0 \Longrightarrow \omega \uparrow$ (*length ms*) $\leq$ *omega-sum* (*m#ms*)
apply *clarsimp*
by (*metis Ord-ord-of-nat add-le-cancel-left0 le-mult Nat.neq0-conv ord-of-eq-0-iff vsubsetD*)

lemma *omega-sum-length-less*:
assumes *normal ns length ms* $<$ *length ns*
shows *omega-sum ms* $<$ *omega-sum ns*
using *assms*
proof (*induction rule*: *normal.induct*)
case (*normal-Cons n ns'*)
have $\omega \uparrow$ *length ms* $\leq \omega \uparrow$ *length ns'*
using *normal-Cons oexp-mono-le* **by** *auto*
then show *?case*
by (*metis gr-implies-not-zero less-le-trans normal-Cons.hyps omega-sum-aux-less omega-sum-ge*)
qed *auto*

lemma *omega-sum-length-leD*:
assumes *omega-sum ms* $\leq$ *omega-sum ns normal ms*
shows *length ms* $\leq$ *length ns*
by (*meson assms leD leI omega-sum-length-less*)

lemma *omega-sum-less-eqlen-iff-cases* [*simp*]:
assumes *length ms* $=$ *length ns*
shows *omega-sum* (*m#ms*) $<$ *omega-sum* (*n#ns*) $\longleftrightarrow m<n \vee m=n \wedge$ *omega-sum ms* $<$ *omega-sum ns*
using *omega-sum-less* [*of ms*] *omega-sum-less* [*of ns*]
by (*metis Kirby.add-less-cancel-left Omega-Omega.Cons Ord-ω Ord-oexp Ord-omega-sum Ord-ord-of-nat assms length-Cons linorder-neqE-nat mult-nat-less-add-less order-less-asym*)

lemma *omega-sum-less-iff-cases*:

assumes $m > 0$ $n > 0$
shows *omega-sum* $(m\#ms) <$ *omega-sum* $(n\#ns)$
$\longleftrightarrow$ *length ms* $<$ *length ns*
$\vee$ *length ms* $=$ *length ns* $\wedge$ $m<n$
$\vee$ *length ms* $=$ *length ns* $\wedge$ $m=n \wedge$ *omega-sum ms* $<$ *omega-sum ns*
by (*smt* (*verit*) *assms length-Cons less-eq-Suc-le less-le-not-le nat-less-le normal-Cons not-le*
omega-sum-length-less omega-sum-less-eqlen-iff-cases)

lemma *omega-sum-less-iff*:
((*length ms*, *omega-sum ms*), (*length ns*, *omega-sum ns*)) $\in$ *less-than* $<*lex*>$ *VWF*
$\longleftrightarrow$ (*ms*,*ns*) $\in$ *lenlex less-than*
proof (*induction ms arbitrary*: *ns*)
case (*Cons m ms*)
then show *?case*
proof (*induction ns*)
case (*Cons n ns'*)
show *?case*
using *Cons.prems Cons-lenlex-iff omega-sum-less-eqlen-iff-cases* **by** *fastforce*
qed *auto*
qed *auto*

lemma *eq-omega-sum-less-iff*:
assumes *length ms* $=$ *length ns*
shows (*omega-sum ms*, *omega-sum ns*) $\in$ *VWF* $\longleftrightarrow$ (*ms*,*ns*) $\in$ *lenlex less-than*
by (*metis assms in-lex-prod less-not-refl less-than-iff omega-sum-less-iff*)

lemma *eq-omega-sum-eq-iff*:
assumes *length ms* $=$ *length ns*
shows *omega-sum ms* $=$ *omega-sum ns* $\longleftrightarrow$ *ms*$=$*ns*
proof
assume *omega-sum ms* $=$ *omega-sum ns*
then have (*omega-sum ms*, *omega-sum ns*) $\notin$ *VWF* (*omega-sum ns*, *omega-sum ms*) $\notin$ *VWF*
by *auto*
then obtain (*ms*,*ns*) $\notin$ *lenlex less-than* (*ns*,*ms*) $\notin$ *lenlex less-than*
using *assms eq-omega-sum-less-iff* **by** *metis*
moreover have *total* (*lenlex less-than*)
by (*simp add*: *total-lenlex total-less-than*)
ultimately show *ms*$=$*ns*
by (*meson UNIV-I total-on-def*)
qed *auto*

lemma *inj-omega-sum*: *inj-on omega-sum* $\{l.$ *length* $l = n\}$
unfolding *inj-on-def* **using** *eq-omega-sum-eq-iff* **by** *fastforce*

lemma *Ex-omega-sum*: $\gamma \in$ *elts* $(\omega\uparrow n) \Longrightarrow \exists$ *ns*. $\gamma =$ *omega-sum ns* $\wedge$ *length ns* $=$ *n*

proof (*induction n arbitrary*: γ)
  case *0*
  then show *?case*
    by (*rule-tac x*=[] **in** *exI*) *auto*
next
  case (*Suc n*)
  then obtain *k*::*nat* **where** *k*: $\gamma \in$ *elts* ($\omega \uparrow n * k$)
      and *kmin*: $\bigwedge k'.\ k'<k \Longrightarrow \gamma \notin$ *elts* ($\omega \uparrow n * k'$)
    by (*metis Ord-ord-of-nat elts-mult-ωE oexp-succ ord-of-nat.simps*(*2*))
  show *?case*
  proof (*cases k*)
    case (*Suc k'*)
    then obtain δ **where** δ: $\gamma = (\omega \uparrow n * k') + \delta$
      by (*metis lessI mult-succ ord-of-nat.simps*(*2*) *k kmin mem-plus-V-E*)
    then have δ*in*: $\delta \in$ *elts* ($\omega \uparrow n$)
      using *Suc k mult-succ* **by** *auto*
    then obtain *ns* **where** *ns*: δ = *omega-sum ns* **and** *len*: *length ns* = *n*
      using *Suc.IH* **by** *auto*
    moreover have *omega-sum ns* < $\omega\uparrow n$
      using *OrdmemD ns* δ*in* **by** *auto*
    ultimately show *?thesis*
      by (*rule-tac x*=*k'*#*ns* **in** *exI*) (*simp add*: δ)
  qed (*use k* **in** *auto*)
qed

lemma *omega-sum-drop* [*simp*]: *omega-sum* (*dropWhile* ($\lambda n.$ *n*=*0*) *ns*) = *omega-sum ns*
  by (*induction ns*) *auto*

lemma *normal-drop* [*simp*]: *normal* (*dropWhile* ($\lambda n.$ *n*=*0*) *ns*)
  by (*induction ns*) *auto*

lemma *omega-sum-ωω*:
  assumes $\gamma \in$ *elts* ($\omega\uparrow\omega$)
  obtains *ns* **where** γ = *omega-sum ns normal ns*
proof −
  obtain *ms* **where** γ = *omega-sum ms*
    using *assms Ex-omega-sum* **by** (*auto simp*: *oexp-Limit elts-ω*)
  then show *thesis*
    by (*metis normal-drop omega-sum-drop that*)
qed

definition *Cantor-ωω* :: *V* ⇒ *nat list*
  where *Cantor-ωω* ≡ $\lambda x.$ *SOME ns. x = omega-sum ns* ∧ *normal ns*

lemma
  assumes $\gamma \in$ *elts* ($\omega\uparrow\omega$)
  shows *Cantor-ωω*: *omega-sum* (*Cantor-ωω* γ) = γ
    and *normal-Cantor-ωω*: *normal* (*Cantor-ωω* γ)

by (*metis* (*mono-tags*, *lifting*) *Cantor-ωω-def assms omega-sum-ωω someI*)+

3.2 Larson's set $W(n)$

definition *WW* :: *nat list set*
where *WW* ≡ {*l*. *strict-sorted l*}

fun *into-WW* :: *nat* ⇒ *nat list* ⇒ *nat list* **where**
into-WW k [] = []
| *into-WW k* (*n*#*ns*) = (*k*+*n*) # *into-WW* (*Suc* (*k*+*n*)) *ns*

fun *from-WW* :: *nat* ⇒ *nat list* ⇒ *nat list* **where**
from-WW k [] = []
| *from-WW k* (*n*#*ns*) = (*n* − *k*) # *from-WW* (*Suc n*) *ns*

lemma *from-into-WW* [*simp*]: *from-WW k* (*into-WW k ns*) = *ns*
by (*induction ns arbitrary*: *k*) *auto*

lemma *inj-into-WW*: *inj* (*into-WW k*)
by (*metis from-into-WW injI*)

lemma *into-from-WW-aux*:
⟦*strict-sorted ns*; ∀ *n*∈*list.set ns*. *k* ≤ *n*⟧ ⟹ *into-WW k* (*from-WW k ns*) = *ns*
by (*induction ns arbitrary*: *k*) (*auto simp*: *Suc-leI*)

lemma *into-from-WW* [*simp*]: *strict-sorted ns* ⟹ *into-WW 0* (*from-WW 0 ns*) = *ns*
by (*simp add*: *into-from-WW-aux*)

lemma *into-WW-imp-ge*: *y* ∈ *List.set* (*into-WW x ns*) ⟹ *x* ≤ *y*
by (*induction ns arbitrary*: *x*) *fastforce*+

lemma *strict-sorted-into-WW*: *strict-sorted* (*into-WW x ns*)
by (*induction ns arbitrary*: *x*) (*auto simp*: *dest*: *into-WW-imp-ge*)

lemma *length-into-WW*: *length* (*into-WW x ns*) = *length ns*
by (*induction ns arbitrary*: *x*) *auto*

lemma *WW-eq-range-into*: *WW* = *range* (*into-WW 0*)
proof −
have ⋀*ns*. *strict-sorted ns* ⟹ *ns* ∈ *range* (*into-WW 0*)
by (*metis into-from-WW rangeI*)
then show *?thesis* **by** (*auto simp*: *WW-def strict-sorted-into-WW*)
qed

lemma *into-WW-lenlex-iff*: (*into-WW k ms*, *into-WW k ns*) ∈ *lenlex less-than* ⟷ (*ms*, *ns*) ∈ *lenlex less-than*
proof (*induction ms arbitrary*: *ns k*)
case *Nil*

then show *?case*
by *simp* (*metis length-0-conv length-into-WW*)
next
case (*Cons m ms*)
then show *?case*
by (*induction ns*) (*auto simp*: *Cons-lenlex-iff length-into-WW*)
qed

lemma *wf-llt* [*simp*]: *wf* (*lenlex less-than*) **and** *trans-llt* [*simp*]: *trans* (*lenlex less-than*)
by *blast+*

lemma *total-llt* [*simp*]: *total-on A* (*lenlex less-than*)
by (*meson UNIV-I total-lenlex total-less-than total-on-def*)

lemma *omega-sum-1-less*:
assumes (*ms*,*ns*) ∈ *lenlex less-than* **shows** *omega-sum* (*1*#*ms*) < *omega-sum* (*1*#*ns*)
proof −
have *omega-sum* (*1*#*ms*) < *omega-sum* (*1*#*ns*) **if** *length ms* < *length ns*
using *omega-sum-less-iff-cases that zero-less-one* **by** *blast*
then show *?thesis*
using *assms* **by** (*auto simp*: *mult-succ simp flip*: *omega-sum-less-iff*)
qed

lemma *ordertype-WW-1*: *ordertype WW* (*lenlex less-than*) ≤ *ordertype UNIV* (*lenlex less-than*)
by (*rule ordertype-mono*) *auto*

lemma *ordertype-WW-2*: *ordertype UNIV* (*lenlex less-than*) ≤ ω↑ω
proof (*rule ordertype-inc-le-Ord*)
show *range* (λ*ms*. *omega-sum* (*1*#*ms*)) ⊆ *elts* (ω↑ω)
by (*meson Ord-ω Ord-mem-iff-lt Ord-oexp Ord-omega-sum image-subset-iff omega-sum-less-ωω*)
qed (*use omega-sum-1-less* **in** *auto*)

lemma *ordertype-WW-3*: ω↑ω ≤ *ordertype WW* (*lenlex less-than*)
proof −
define π **where** π ≡ *into-WW 0* ∘ *Cantor-ωω*
have *ωω*: ω↑ω = *tp* (*elts* (ω↑ω))
by *simp*
also have ... ≤ *ordertype WW* (*lenlex less-than*)
proof (*rule ordertype-inc-le*)
fix α β
assume α: α ∈ *elts* (ω↑ω) **and** β: β ∈ *elts* (ω↑ω) **and** (α, β) ∈ *VWF*
then obtain ∗: *Ord* α *Ord* β α<β
by (*metis Ord-in-Ord Ord-ordertype VWF-iff-Ord-less ωω*)
then have *length* (*Cantor-ωω* α) ≤ *length* (*Cantor-ωω* β)
using α β **by** (*simp add*: *Cantor-ωω normal-Cantor-ωω omega-sum-length-leD*)
with α β ∗ **have** (*Cantor-ωω* α, *Cantor-ωω* β) ∈ *lenlex less-than*

by (*auto simp*: *Cantor-ωω simp flip*: *omega-sum-less-iff*)
then show (π α, π β) $\in$ *lenlex less-than*
by (*simp add*: π-*def into-WW-lenlex-iff*)
qed (*auto simp*: π-*def WW-def strict-sorted-into-WW*)
finally show $\omega\uparrow\omega \leq$ *ordertype WW* (*lenlex less-than*) **.**
qed

lemma *ordertype-WW*: *ordertype WW* (*lenlex less-than*) = $\omega\uparrow\omega$
and *ordertype-UNIV-ωω*: *ordertype UNIV* (*lenlex less-than*) = $\omega\uparrow\omega$
using *ordertype-WW-1 ordertype-WW-2 ordertype-WW-3* **by** *auto*

lemma *ordertype-ωω*:
fixes F :: *nat* $\Rightarrow$ *nat list set*
assumes $\bigwedge j$::*nat*. *ordertype* (F *j*) (*lenlex less-than*) = $\omega\uparrow j$
shows *ordertype* ($\bigcup j$. F *j*) (*lenlex less-than*) = $\omega\uparrow\omega$
proof (*rule antisym*)
show *ordertype* ($\bigcup$ (*range F*)) (*lenlex less-than*) $\leq \omega \uparrow \omega$
by (*metis ordertype-UNIV-ωω ordertype-mono small top-greatest trans-llt wf-llt*)
have $\bigwedge n$. $\omega \uparrow$ *ord-of-nat n* $\leq$ *ordertype* ($\bigcup$ (*range F*)) (*lenlex less-than*)
by (*metis TC-small Union-upper assms ordertype-mono rangeI trans-llt wf-llt*)
then show $\omega \uparrow \omega \leq$ *ordertype* ($\bigcup$ (*range F*)) (*lenlex less-than*)
by (*auto simp*: *oexp-ω-Limit ZFC-in-HOL.SUP-le-iff elts-ω*)
qed

definition *WW-seg* :: *nat* $\Rightarrow$ *nat list set*
where *WW-seg n* $\equiv$ {*l* $\in$ *WW*. *length l* = *n*}

lemma *WW-seg-subset-WW*: *WW-seg n* $\subseteq$ *WW*
by (*auto simp*: *WW-seg-def*)

lemma *WW-eq-UN-WW-seg*: *WW* = ($\bigcup$ *n*. *WW-seg n*)
by (*auto simp*: *WW-seg-def*)

lemma *ordertype-list-seg*: *ordertype* {*l*. *length l* = *n*} (*lenlex less-than*) = $\omega\uparrow n$
proof −
have *bij-betw omega-sum* {*l*. *length l* = *n*} (*elts* ($\omega\uparrow n$))
unfolding *WW-seg-def bij-betw-def*
by (*auto simp*: *inj-omega-sum Ord-mem-iff-lt omega-sum-less dest*: *Ex-omega-sum*)
then show *?thesis*
by (*force simp*: *ordertype-eq-iff simp flip*: *eq-omega-sum-less-iff*)
qed

lemma *ordertype-WW-seg*: *ordertype* (*WW-seg n*) (*lenlex less-than*) = $\omega\uparrow n$
(**is** *ordertype ?W ?R* = $\omega\uparrow n$)
proof −
have *ordertype* {*l*. *length l* = *n*} *?R* = *ordertype ?W ?R*

proof (*subst ordertype-eq-ordertype*)
show $\exists f.$ *bij-betw* f $\{l.$ *length* $l = n\}$ *?W* $\wedge$ $(\forall x{\in}\{l.$ *length* $l = n\}.$ $\forall y{\in}\{l.$ *length* $l = n\}.$ $((f\ x, f\ y) \in$ *lenlex less-than*$) = ((x, y) \in$ *lenlex less-than*$))$
proof (*intro exI conjI*)
have *inj-on* (*into-WW 0*) $\{l.$ *length* $l = n\}$
by (*metis from-into-WW inj-onI*)
then show *bij-betw* (*into-WW 0*) $\{l.$ *length* $l = n\}$ *?W*
by (*auto simp*: *bij-betw-def WW-seg-def WW-eq-range-into length-into-WW*)
qed (*simp add*: *into-WW-lenlex-iff*)
qed *auto*
then show *?thesis*
using *ordertype-list-seg* **by** *auto*
qed

3.3 Definitions required for the lemmas

3.3.1 Larson's "<"-relation on ordered lists

instantiation *list* :: (*ord*)*ord*
begin

definition $xs < ys \equiv xs \neq [] \wedge ys \neq [] \longrightarrow$ *last xs* $<$ *hd ys* **for** *xs ys* :: *'a list*
definition $xs \leq ys \equiv xs < ys \vee xs = ys$ **for** *xs ys* :: *'a list*

instance
by *standard*

end

lemma *less-Nil* [*simp*]: $xs < [] \ [] < xs$
by (*auto simp*: *less-list-def*)

lemma *less-sets-imp-list-less*:
assumes *list.set xs* $\ll$ *list.set ys*
shows $xs < ys$
by (*metis assms last-in-set less-list-def less-sets-def list.set-sel*(*1*))

lemma *less-sets-imp-sorted-list-of-set*:
assumes $A \ll B$ *finite A finite B*
shows *list-of A* $<$ *list-of B*
by (*simp add*: *assms less-sets-imp-list-less*)

lemma *sorted-list-of-set-imp-less-sets*:
assumes $xs < ys$ *sorted xs sorted ys*
shows *list.set xs* $\ll$ *list.set ys*
using *assms sorted-hd-le sorted-le-last*
by (*force simp*: *less-list-def less-sets-def intro*: *order.trans*)

lemma *less-list-iff-less-sets*:
assumes *sorted xs sorted ys*

shows *xs* < *ys* ⟷ *list.set xs* ≪ *list.set ys*
using *assms sorted-hd-le sorted-le-last*
by (*force simp*: *less-list-def less-sets-def intro*: *order.trans*)

lemma *strict-sorted-append-iff*:
strict-sorted (*xs* @ *ys*) ⟷ *xs* < *ys* ∧ *strict-sorted xs* ∧ *strict-sorted ys*
by (*metis less-list-iff-less-sets less-setsD sorted-wrt-append strict-sorted-imp-less-sets strict-sorted-imp-sorted*)

lemma *singleton-less-list-iff*: *sorted xs* ⟹ [*n*] < *xs* ⟷ {..*n*} ∩ *list.set xs* = {}
apply (*simp add*: *less-list-def disjoint-iff*)
by (*metis empty-iff less-le-trans list.set*(*1*) *list.set-sel*(*1*) *not-le sorted-hd-le*)

lemma *less-hd-imp-less*: *xs* < [*hd ys*] ⟹ *xs* < *ys*
by (*simp add*: *less-list-def*)

lemma *strict-sorted-concat-I*:
assumes ⋀*x*. *x* ∈ *list.set xs* ⟹ *strict-sorted x*
⋀*n*. *Suc n* < *length xs* ⟹ *xs*!*n* < *xs*!*Suc n*
xs ∈ *lists* (− {[]})
shows *strict-sorted* (*concat xs*)
using *assms*
proof (*induction xs*)
case (*Cons x xs*)
then have *x* < *concat xs*
apply (*simp add*: *less-list-def*)
by (*metis Compl-iff hd-concat insertI1 length-greater-0-conv length-pos-if-in-set list.sel*(*1*) *lists.cases nth-Cons-0*)
with *Cons* **show** *?case*
by (*force simp*: *strict-sorted-append-iff*)
qed *auto*

3.4 Nash Williams for lists

3.4.1 Thin sets of lists

inductive *initial-segment* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *initial-segment xs* (*xs*@*ys*)

definition *thin* :: *'a list set* ⇒ *bool*
where *thin A* ≡ ¬ (∃ *x y*. *x* ∈ *A* ∧ *y* ∈ *A* ∧ *x* ≠ *y* ∧ *initial-segment x y*)

lemma *initial-segment-ne*:
assumes *initial-segment xs ys xs* ≠ []
shows *ys* ≠ [] ∧ *hd ys* = *hd xs*
using *assms* **by** (*auto elim*!: *initial-segment.cases*)

lemma *take-initial-segment*:
assumes *initial-segment xs ys k* ≤ *length xs*
shows *take k xs* = *take k ys*

by (*metis append-eq-conv-conj assms initial-segment.cases min-def take-take*)

lemma *initial-segment-length-eq*:
assumes *initial-segment xs ys length xs = length ys*
shows *xs = ys*
using *assms initial-segment.cases* **by** *fastforce*

lemma *initial-segment-Nil* [*simp*]: *initial-segment* [] *ys*
by (*simp add*: *initial-segment.simps*)

lemma *initial-segment-Cons* [*simp*]: *initial-segment* (*x*#*xs*) (*y*#*ys*) ⟷ *x*=*y* ∧ *initial-segment xs ys*
by (*metis append-Cons initial-segment.simps list.inject*)

lemma *init-segment-iff-initial-segment*:
assumes *strict-sorted xs strict-sorted ys*
shows *init-segment* (*list.set xs*) (*list.set ys*) ⟷ *initial-segment xs ys* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *S′* **where** *S′*: *list.set ys = list.set xs* ∪ *S′ list.set xs* ≪ *S′*
by (*auto simp*: *init-segment-def*)
then have *finite S′*
by (*metis List.finite-set finite-Un*)
have *ys = xs* @ *list-of S′*
using *S′* ‹*strict-sorted xs*›
proof (*induction xs*)
case *Nil*
with ‹*strict-sorted ys*› **show** *?case*
by *auto*
next
case (*Cons a xs*)
with ‹*finite S′*› **have** *ys = a # xs* @ *list-of S′*
by (*metis List.finite-set append-Cons assms*(*2*) *sorted-list-of-set-Un sorted-list-of-set-set-of*)
then show *?case*
by (*auto simp*: *Cons*)
qed
then show *?rhs*
using *initial-segment.intros* **by** *blast*
next
assume *?rhs*
then show *?lhs*
proof *cases*
case (*1 ys*)
with *assms* **show** *?thesis*
by (*simp add*: *init-segment-Un strict-sorted-imp-less-sets*)
qed
qed

theorem *Nash-Williams-WW*:
fixes $h :: nat\ list \Rightarrow nat$
assumes *infinite M* **and** *h*: $h\ `\ \{l \in A.\ List.set\ l \subseteq M\} \subseteq \{..<2\}$ **and** *thin A A* $\subseteq WW$
obtains *i N* **where** $i < 2$ *infinite N* $N \subseteq M$ $h\ `\ \{l \in A.\ List.set\ l \subseteq N\} \subseteq \{i\}$
proof −
define *AM* **where** $AM \equiv \{l \in A.\ List.set\ l \subseteq M\}$
have *thin-set* (*list.set* ` *A*)
using ‹*thin A*› ‹$A \subseteq WW$› **unfolding** *thin-def thin-set-def WW-def*
by (*auto simp*: *subset-iff init-segment-iff-initial-segment*)
then have *thin-set* (*list.set* ` *AM*)
by (*simp add*: *AM-def image-subset-iff thin-set-def*)
then have *Ramsey* (*list.set* ` *AM*) *2*
using *Nash-Williams-2* **by** *metis*
moreover have $(h \circ list\text{-}of) \in list.set\ `\ AM \to \{..<2\}$
unfolding *AM-def*
proof *clarsimp*
fix *l*
assume $l \in A$ *list.set* $l \subseteq M$
then have *strict-sorted l*
using *WW-def* ‹$A \subseteq WW$› **by** *blast*
then show $h\ (list\text{-}of\ (list.set\ l)) < 2$
using *h* ‹$l \in A$› ‹*list.set l* $\subseteq M$› **by** *auto*
qed
ultimately obtain *N i* **where** *N*: $N \subseteq M$ *infinite N* $i<2$
and $list.set\ `\ AM \cap Pow\ N \subseteq (h \circ list\text{-}of) -`\ \{i\}$
unfolding *Ramsey-eq* **by** (*metis* ‹*infinite M*›)
then have *N-disjoint*: $(h \circ list\text{-}of) -`\ \{1-i\} \cap (list.set\ `\ AM) \cap Pow\ N = \{\}$
unfolding *subset-vimage-iff less-2-cases-iff* **by** *force*
have $h\ `\ \{l \in A.\ list.set\ l \subseteq N\} \subseteq \{i\}$
proof *clarify*
fix *l*
assume $l \in A$ **and** *list.set* $l \subseteq N$
then have $h\ l < 2$
using *h* ‹$N \subseteq M$› **by** *force*
with ‹$i<2$› **have** $h\ l \neq Suc\ 0 - i \Longrightarrow h\ l = i$
by (*auto simp*: *eval-nat-numeral less-Suc-eq*)
moreover have *strict-sorted l*
using ‹$A \subseteq WW$› ‹$l \in A$› **unfolding** *WW-def* **by** *blast*
moreover have $h\ (list\text{-}of\ (list.set\ l)) = 1 - i \longrightarrow \neg\ (list.set\ l \subseteq N)$
using *N-disjoint* ‹$N \subseteq M$› ‹$l \in A$› **by** (*auto simp*: *AM-def*)
ultimately show $h\ l = i$
using *N* ‹$N \subseteq M$› ‹$l \in A$› ‹*list.set l* $\subseteq N$›
by (*auto simp*: *vimage-def set-eq-iff AM-def WW-def subset-iff*)
qed
then show *thesis*
using *that* ‹$i<2$› *N* **by** *auto*
qed

3.5 Specialised functions on lists

lemma *mem-lists-non-Nil*: $xss \in lists\ (-\ \{[]\}) \longleftrightarrow (\forall x \in list.set\ xss.\ x \neq [])$
by *auto*

fun *acc-lengths* :: $nat \Rightarrow 'a\ list\ list \Rightarrow nat\ list$
where $acc\text{-}lengths\ acc\ [] = []$
| $acc\text{-}lengths\ acc\ (l\#ls) = (acc + length\ l)\ \#\ acc\text{-}lengths\ (acc + length\ l)\ ls$

lemma *length-acc-lengths* [*simp*]: $length\ (acc\text{-}lengths\ acc\ ls) = length\ ls$
by (*induction ls arbitrary*: *acc*) *auto*

lemma *acc-lengths-eq-Nil-iff* [*simp*]: $acc\text{-}lengths\ acc\ ls = [] \longleftrightarrow ls = []$
by (*metis length-0-conv length-acc-lengths*)

lemma *set-acc-lengths*:
assumes $ls \in lists\ (-\ \{[]\})$ **shows** $list.set\ (acc\text{-}lengths\ acc\ ls) \subseteq \{acc<..\}$
using *assms* **by** (*induction ls rule*: *acc-lengths.induct*) *fastforce+*

Useful because *acc-lengths.simps* will sometimes be deleted from the simpset.

lemma *hd-acc-lengths* [*simp*]: $hd\ (acc\text{-}lengths\ acc\ (l\#ls)) = acc + length\ l$
by *simp*

lemma *last-acc-lengths* [*simp*]:
$ls \neq [] \Longrightarrow last\ (acc\text{-}lengths\ acc\ ls) = acc + sum\text{-}list\ (map\ length\ ls)$
by (*induction acc ls rule*: *acc-lengths.induct*) *auto*

lemma *nth-acc-lengths* [*simp*]:
$[\![ls \neq []; k < length\ ls]\!] \Longrightarrow acc\text{-}lengths\ acc\ ls\ !\ k = acc + sum\text{-}list\ (map\ length\ (take\ (Suc\ k)\ ls))$
by (*induction acc ls arbitrary*: *k rule*: *acc-lengths.induct*) (*fastforce simp*: *less-Suc-eq nth-Cons'*)+

lemma *acc-lengths-plus*: $acc\text{-}lengths\ (m+n)\ as = map\ ((+)m)\ (acc\text{-}lengths\ n\ as)$
by (*induction n as arbitrary*: *m rule*: *acc-lengths.induct*) (*auto simp*: *add.assoc*)

lemma *acc-lengths-shift*: $NO\text{-}MATCH\ 0\ acc \Longrightarrow acc\text{-}lengths\ acc\ as = map\ ((+)acc)\ (acc\text{-}lengths\ 0\ as)$
by (*metis acc-lengths-plus add.comm-neutral*)

lemma *length-concat-acc-lengths*:
$ls \neq [] \Longrightarrow k + length\ (concat\ ls) \in list.set\ (acc\text{-}lengths\ k\ ls)$
by (*metis acc-lengths-eq-Nil-iff last-acc-lengths last-in-set length-concat*)

lemma *strict-sorted-acc-lengths*:
assumes $ls \in lists\ (-\ \{[]\})$ **shows** $strict\text{-}sorted\ (acc\text{-}lengths\ acc\ ls)$
using *assms*
proof (*induction ls rule*: *acc-lengths.induct*)
case (*2 acc l ls*)

```
  then have strict-sorted (acc-lengths (acc + length l) ls)
    by auto
  then show ?case
    using set-acc-lengths 2.prems by auto
qed auto

lemma acc-lengths-append:
  acc-lengths acc (xs @ ys)
   = acc-lengths acc xs @ acc-lengths (acc + sum-list (map length xs)) ys
by (induction acc xs rule: acc-lengths.induct) (auto simp: add.assoc)


lemma length-concat-ge:
  assumes as ∈ lists (− {[]})
  shows length (concat as) ≥ length as
  using assms
proof (induction as)
  case (Cons a as)
  then have length a ≥ Suc 0 ⋀l. l ∈ list.set as ⟹ length l ≥ Suc 0
    by (auto simp: Suc-leI)
  then show ?case
    using Cons.IH by force
qed auto


fun interact :: 'a list list ⇒ 'a list list ⇒ 'a list
  where
  interact [] ys = concat ys
| interact xs [] = concat xs
| interact (x#xs) (y#ys) = x @ y @ interact xs ys

lemma (in monoid-add) length-interact:
  length (interact xs ys) = sum-list (map length xs) + sum-list (map length ys)
  by (induction rule: interact.induct) (auto simp: length-concat)

lemma length-interact-ge:
  assumes xs ∈ lists (− {[]}) ys ∈ lists (− {[]})
  shows length (interact xs ys) ≥ length xs + length ys
  by (metis add-mono assms length-concat length-concat-ge length-interact)

lemma set-interact [simp]:
  shows list.set (interact xs ys) = list.set (concat xs) ∪ list.set (concat ys)
by (induction rule: interact.induct) auto

lemma interact-eq-Nil-iff [simp]:
  assumes xs ∈ lists (− {[]}) ys ∈ lists (− {[]})
  shows interact xs ys = [] ⟷ xs=[] ∧ ys=[]
  using length-interact-ge [OF assms]  by fastforce
```

lemma *interact-sing* [*simp*]: *interact* [*x*] *ys* = *x* @ *concat ys*
by (*metis* (*no-types*) *concat.simps*(*2*) *interact.simps neq-Nil-conv*)

lemma *hd-interact*: ⟦*xs* ≠ []; *hd xs* ≠ []⟧ ⟹ *hd* (*interact xs ys*) = *hd* (*hd xs*)
by (*smt* (*verit*, *best*) *hd-append2 hd-concat interact.elims list.sel*(*1*))

lemma *acc-lengths-concat-injective*:
assumes *concat as′* = *concat as acc-lengths n as′* = *acc-lengths n as*
shows *as′* = *as*
using *assms*
proof (*induction as arbitrary*: *n as′*)
case *Nil*
then show *?case*
by (*metis acc-lengths-eq-Nil-iff*)
next
case (*Cons a as*)
then obtain *a′ bs* **where** *as′* = *a′*#*bs*
by (*metis Suc-length-conv length-acc-lengths*)
with *Cons* **show** *?case*
by *simp*
qed

lemma *acc-lengths-interact-injective*:
assumes *interact as′ bs′* = *interact as bs acc-lengths a as′* = *acc-lengths a as*
acc-lengths b bs′ = *acc-lengths b bs*
shows *as′* = *as* ∧ *bs′* = *bs*
using *assms*
proof (*induction as bs arbitrary*: *a b as′ bs′ rule*: *interact.induct*)
case (*1 cs*) **then show** *?case*
by (*metis acc-lengths-concat-injective acc-lengths-eq-Nil-iff interact.simps*(*1*))
next
case (*2 c cs*)
then show *?case*
by (*metis acc-lengths-concat-injective acc-lengths-eq-Nil-iff interact.simps*(*2*)
list.exhaust)
next
case (*3 x xs y ys*)
then obtain *a′ us b′ vs* **where** *as′* = *a′*#*us bs′* = *b′*#*vs*
by (*metis length-Suc-conv length-acc-lengths*)
with *3* **show** *?case*
by *auto*
qed

lemma *strict-sorted-interact-I*:
assumes *length ys* ≤ *length xs length xs* ≤ *Suc* (*length ys*)
⋀*x*. *x* ∈ *list.set xs* ⟹ *strict-sorted x*
⋀*y*. *y* ∈ *list.set ys* ⟹ *strict-sorted y*
⋀*n*. *n* < *length ys* ⟹ *xs*!*n* < *ys*!*n*

$\bigwedge n.\ Suc\ n < length\ xs \Longrightarrow ys!n < xs!Suc\ n$
assumes *xs* ∈ *lists* (− {[]}) *ys* ∈ *lists* (− {[]})
shows *strict-sorted* (*interact xs ys*)
using *assms*
proof (*induction rule*: *interact.induct*)
case (*3 x xs y ys*)
then have $x < y$
by *force*
moreover have *strict-sorted* (*interact xs ys*)
using *3* **by** *simp* (*metis Suc-less-eq nth-Cons-Suc*)
moreover have $y <$ *interact xs ys*
using *3* **apply** (*simp add*: *less-list-def*)
by (*metis hd-interact le-zero-eq length-greater-0-conv list.sel*(*1*) *list.set-sel*(*1*) *list.size*(*3*) *lists.simps mem-lists-non-Nil nth-Cons-0*)
ultimately show *?case*
using *3* **by** (*simp add*: *strict-sorted-append-iff less-list-def*)
qed *auto*

3.6 Forms and interactions

3.6.1 Forms

inductive *Form-Body* :: [*nat*, *nat*, *nat list*, *nat list*, *nat list*] ⇒ *bool*
where *Form-Body ka kb xs ys zs*
if *length xs* < *length ys xs* = *concat* (*a*#*as*) *ys* = *concat* (*b*#*bs*)
a#*as* ∈ *lists* (− {[]}) *b*#*bs* ∈ *lists* (− {[]})
length (*a*#*as*) = *ka length* (*b*#*bs*) = *kb*
c = *acc-lengths 0* (*a*#*as*)
d = *acc-lengths 0* (*b*#*bs*)
zs = *concat* [*c*, *a*, *d*, *b*] @ *interact as bs*
strict-sorted zs

inductive *Form* :: [*nat*, *nat list set*] ⇒ *bool*
where *Form 0* {*xs*,*ys*} **if** *length xs* = *length ys xs* ≠ *ys*
| *Form* (*2*∗*k*−*1*) {*xs*,*ys*} **if** *Form-Body k k xs ys zs k* > *0*
| *Form* (*2*∗*k*) {*xs*,*ys*} **if** *Form-Body* (*Suc k*) *k xs ys zs k* > *0*

inductive-cases *Form-0-cases-raw*: *Form 0 u*

lemma *Form-elim-upair*:
assumes *Form l U*
obtains *xs ys* **where** *xs* ≠ *ys U* = {*xs*,*ys*} *length xs* ≤ *length ys*
using *assms*
by (*smt* (*verit*, *best*) *Form.simps Form-Body.cases less-or-eq-imp-le nat-neq-iff*)

lemma assumes *Form-Body ka kb xs ys zs*
shows *Form-Body-WW*: *zs* ∈ *WW*
and *Form-Body-nonempty*: *length zs* > *0*

and *Form-Body-length*: *length xs* < *length ys*
using *Form-Body.cases* [*OF assms*] **by** (*fastforce simp*: *WW-def*)+

lemma *form-cases*:
fixes *l*::*nat*
obtains (*zero*) *l* = *0* | (*nz*) *ka kb* **where** *l* = *ka*+*kb* − *1* *0* < *kb* *kb* ≤ *ka* *ka* ≤ *Suc kb*
proof −
have *l* = *0* ∨ (∃ *ka kb*. *l* = *ka*+*kb* − *1* ∧ *0* < *kb* ∧ *kb* ≤ *ka* ∧ *ka* ≤ *Suc kb*)
by *presburger*
then show *thesis*
using *nz zero* **by** *blast*
qed

3.6.2 Interactions

lemma *interact*:
assumes *Form l U l*>*0*
obtains *ka kb xs ys zs* **where** *l* = *ka*+*kb* − *1* *U* = {*xs*,*ys*} *Form-Body ka kb xs ys zs* *0* < *kb* *kb* ≤ *ka* *ka* ≤ *Suc kb*
using *assms*
unfolding *Form.simps*
by (*smt* (*verit*, *best*) *add-Suc diff-Suc-1 lessI mult-2 nat-less-le order-refl*)

definition *inter-scheme* :: *nat* ⇒ *nat list set* ⇒ *nat list*
where *inter-scheme l U* ≡
SOME zs. ∃ *k xs ys*. *U* = {*xs*,*ys*} ∧
(*l* = *2*∗*k*−*1* ∧ *Form-Body k k xs ys zs* ∨ *l* = *2*∗*k* ∧ *Form-Body* (*Suc k*) *k xs ys zs*)

lemma *inter-scheme*:
assumes *Form l U l*>*0*
obtains *ka kb xs ys* **where** *l* = *ka*+*kb* − *1* *U* = {*xs*,*ys*} *Form-Body ka kb xs ys* (*inter-scheme l U*) *0* < *kb* *kb* ≤ *ka* *ka* ≤ *Suc kb*
using *interact* [*OF* ‹*Form l U*›]
proof *cases*
case (*2 ka kb xs ys zs*)
then have §: ⋀*ka kb zs*. ¬ *Form-Body ka kb ys xs zs*
using *Form-Body-length less-asym′* **by** *blast*
have *Form-Body ka kb xs ys* (*inter-scheme l U*)
proof (*cases ka* = *kb*)
case *True*
with *2* **have** *l*: ∀ *k*. *l* ≠ *k* ∗ *2*
by *presburger*
have [*simp*]: ⋀*k*. *kb* + *kb* − *Suc 0* = *k* ∗ *2* − *Suc 0* ⟷ *k*=*kb*
by *auto*
show *?thesis*

unfolding *inter-scheme-def* **using** *2 l True*
by (*auto simp*: § ‹*l* > *0*› *Set.doubleton-eq-iff conj-disj-distribR ex-disj-distrib algebra-simps some-eq-ex*)
next
case *False*
with *2* **have** *l*: ∀ *k*. *l* ≠ *k* ∗ *2* − *Suc 0* **and** [*simp*]: *ka* = *Suc kb*
by *presburger+*
have [*simp*]: ⋀*k*. *kb* + *kb* = *k* ∗ *2* ⟷ *k*=*kb*
by *auto*
show *?thesis*
unfolding *inter-scheme-def* **using** *2 l False*
by (*auto simp*: § ‹*l* > *0*› *Set.doubleton-eq-iff conj-disj-distribR ex-disj-distrib algebra-simps some-eq-ex*)
qed
then show *?thesis*
by (*simp add*: *2 that*)
qed (*use* ‹*l* > *0*› **in** *auto*)

lemma *inter-scheme-strict-sorted*:
assumes *Form l U l*>*0*
shows *strict-sorted* (*inter-scheme l U*)
using *Form-Body.simps assms inter-scheme* **by** *fastforce*

lemma *inter-scheme-simple*:
assumes *Form l U l*>*0*
shows *inter-scheme l U* ∈ *WW* ∧ *length* (*inter-scheme l U*) > *0*
using *inter-scheme* [*OF assms*] **by** (*meson Form-Body-WW Form-Body-nonempty*)

3.6.3 Injectivity of interactions

proposition *inter-scheme-injective*:
assumes *Form l U Form l U′ l* > *0* **and** *eq*: *inter-scheme l U′* = *inter-scheme l U*
shows *U′* = *U*
proof −
obtain *ka kb xs ys*
where *l*: *l* = *ka*+*kb* − *1* **and** *U*: *U* = {*xs*,*ys*}
and *FB*: *Form-Body ka kb xs ys* (*inter-scheme l U*)
and *kb*: *0* < *kb kb* ≤ *ka ka* ≤ *Suc kb*
using *assms inter-scheme* **by** *blast*
then obtain *a as b bs c d*
where *xs*: *xs* = *concat* (*a*#*as*) **and** *ys*: *ys* = *concat* (*b*#*bs*)
and *len*: *length* (*a*#*as*) = *ka length* (*b*#*bs*) = *kb*
and *c*: *c* = *acc-lengths 0* (*a*#*as*)
and *d*: *d* = *acc-lengths 0* (*b*#*bs*)
and *Ueq*: *inter-scheme l U* = *concat* [*c*, *a*, *d*, *b*] @ *interact as bs*
by (*auto simp*: *Form-Body.simps*)
obtain *ka′ kb′ xs′ ys′*
where *l′*: *l* = *ka′*+*kb′* − *1* **and** *U′*: *U′* = {*xs′*,*ys′*}

```
    and FB': Form-Body ka' kb' xs' ys' (inter-scheme l U')
    and kb': 0 < kb' kb' ≤ ka' ka' ≤ Suc kb'
   using assms inter-scheme by blast
 then obtain a' as' b' bs' c' d'
   where xs': xs' = concat (a'#as') and ys': ys' = concat (b'#bs')
    and len': length (a'#as') = ka' length (b'#bs') = kb'
    and c': c' = acc-lengths 0 (a'#as')
    and d': d' = acc-lengths 0 (b'#bs')
    and Ueq': inter-scheme l U' = concat [c', a', d', b'] @ interact as' bs'
   using Form-Body.simps by auto
 have [simp]: ka' = ka ∧ kb' = kb
   using ‹l > 0› l l' kb kb' le-SucE le-antisym mult-2 by linarith
 have [simp]: length c = length c' length d = length d'
   using c c' d d' len' len by auto
 have c-off: c' = c a' @ d' @ b' @ interact as' bs' = a @ d @ b @ interact as bs
   using eq by (auto simp: Ueq Ueq')
 then have len-a: length a' = length a
   by (metis acc-lengths.simps(2) add.left-neutral c c' nth-Cons-0)
 with c-off have §: a' = a d' = d b' @ interact as' bs' = b @ interact as bs
   by auto
 then have length (interact as' bs') = length (interact as bs)
  by (metis acc-lengths.simps(2) add-left-cancel append-eq-append-conv d d' list.inject)
 with § have b' = b interact as' bs' = interact as bs
   by auto
 moreover have acc-lengths 0 as' = acc-lengths 0 as
   using ‹a' = a› ‹c' = c› by (simp add: c' c acc-lengths-shift)
 moreover have acc-lengths 0 bs' = acc-lengths 0 bs
   using ‹b' = b› ‹d' = d› by (simp add: d' d acc-lengths-shift)
 ultimately have as' = as ∧ bs' = bs
   using acc-lengths-interact-injective by blast
 then show ?thesis
   by (simp add: ‹a' = a› U U' ‹b' = b› xs xs' ys ys')
qed


lemma strict-sorted-interact-imp-concat:
   strict-sorted (interact as bs) ⟹ strict-sorted (concat as) ∧ strict-sorted (concat
bs)
proof (induction as bs rule: interact.induct)
 case (3 x xs y ys)
 have x < concat xs
   using 3.prems
  by (smt (verit, del-insts) Un-iff hd-in-set interact.simps(3) last-in-set less-list-def
set-append set-interact sorted-wrt-append)
 moreover have y < concat ys
   using 3 sorted-wrt-append strict-sorted-append-iff by fastforce
 ultimately show ?case
   using 3 by (auto simp add: strict-sorted-append-iff)
qed auto
```

lemma *strict-sorted-interact-hd*:
⟦*strict-sorted* (*interact cs ds*); *cs* ≠ []; *ds* ≠ []; *hd cs* ≠ []; *hd ds* ≠ []⟧
⟹ *hd* (*hd cs*) < *hd* (*hd ds*)
by (*metis append-is-Nil-conv hd-append2 hd-in-set interact.simps*(*3*) *list.exhaust-sel sorted-wrt-append*)

the lengths of the two lists can differ by one

proposition *interaction-scheme-unique-aux*:
assumes *concat as* = *concat as'* **and** *ys'*: *concat bs* = *concat bs'*
and *as* ∈ *lists* (− {[]}) *bs* ∈ *lists* (− {[]})
and *strict-sorted* (*interact as bs*)
and *length bs* ≤ *length as length as* ≤ *Suc* (*length bs*)
and *as'* ∈ *lists* (− {[]}) *bs'* ∈ *lists* (− {[]})
and *strict-sorted* (*interact as' bs'*)
and *length bs'* ≤ *length as' length as'* ≤ *Suc* (*length bs'*)
and *length as* = *length as' length bs* = *length bs'*
shows *as* = *as'* ∧ *bs* = *bs'*
using *assms*
proof (*induction length as arbitrary*: *as bs as' bs'*)
case *0* **then show** *?case*
by *auto*
next
case *SUC*: (*Suc k*)
show *?case*
proof (*cases k*)
case *0*
with *SUC* **obtain** *a a'* **where** *aa'*: *as* = [*a*] *as'* = [*a'*]
by (*metis Suc-length-conv length-0-conv*)
show *?thesis*
proof
show *as* = *as'*
using *aa'* ‹*concat as* = *concat as'*› **by** *force*
with *SUC 0* **show** *bs* = *bs'*
by (*metis Suc-leI append-Nil2 concat.simps impossible-Cons le-antisym length-greater-0-conv list.exhaust*)
qed
next
case (*Suc k'*)
then obtain *a cs b ds* **where** *eq*: *as* = *a*#*cs bs* = *b*#*ds*
using *SUC*
by (*metis le0 list.exhaust list.size*(*3*) *not-less-eq-eq*)
have *length as'* ≠ *0*
using *SUC* **by** *force*
then obtain *a' cs' b' ds'* **where** *eq'*: *as'* = *a'*#*cs' bs'* = *b'*#*ds'*
by (*metis* ‹*length bs* = *length bs'*› *eq*(*2*) *length-0-conv list.exhaust*)
obtain *k*: *k* = *length cs k* ≤ *Suc* (*length ds*)
using *eq SUC* **by** *auto*

case (*Suc k'*)
obtain [*simp*]: $b \neq []$ $b' \neq []$ $a \neq []$ $a' \neq []$
using *SUC* **by** (*simp add*: *eq eq'*)
then have *hd b' = hd b*
using *SUC* **by** (*metis concat.simps(2) eq'(2) eq(2) hd-append2*)
have *ss-ab*: *strict-sorted* (*concat as*) *strict-sorted* (*concat bs*)
using *strict-sorted-interact-imp-concat SUC.prems(5)* **by** *blast+*
have *sw-ab*: *strict-sorted* (*a @ b @ interact cs ds*)
by (*metis SUC.prems(5) eq interact.simps(3)*)
then obtain *a < b strict-sorted a strict-sorted b*
by (*metis append-assoc strict-sorted-append-iff*)
have *b-cs*: *strict-sorted* (*concat* (*b # cs*))
by (*metis append.simps(1) concat.simps(2) interact.simps(3) strict-sorted-interact-imp-concat sw-ab*)
then have *b < concat cs*
using *strict-sorted-append-iff* **by** *auto*
have *strict-sorted* (*a @ concat cs*)
using *eq(1) ss-ab(1)* **by** *force*
have *list.set a = list.set* (*concat as*) ∩ {*..< hd b*}
proof −
have *x ∈ list.set a*
if *x < hd b* **and** *l ∈ list.set cs* **and** *x ∈ list.set l* **for** *x l*
using *b-cs sorted-hd-le strict-sorted-imp-sorted that* **by** *fastforce*
then show *?thesis*
using ‹*b ≠ []*› *sw-ab* **by** (*force simp*: *strict-sorted-append-iff sorted-wrt-append eq*)
qed
moreover
have *ss-ab'*: *strict-sorted* (*concat as'*) *strict-sorted* (*concat bs'*)
using *strict-sorted-interact-imp-concat SUC.prems(10)* **by** *blast+*
have *sw-ab'*: *strict-sorted* (*a' @ b' @ interact cs' ds'*)
by (*metis SUC.prems(10) eq' interact.simps(3)*)
then obtain *a' < b' strict-sorted a' strict-sorted b'*
by (*metis append-assoc strict-sorted-append-iff*)
have *b-cs'*: *strict-sorted* (*concat* (*b' # cs'*))
by (*metis* (*no-types*) *SUC.prems(10) append-Nil eq' interact.simps(3) strict-sorted-append-iff strict-sorted-interact-imp-concat*)
then have *b' < concat cs'*
by (*simp add*: *strict-sorted-append-iff*)
then have *hd b' ∉ list.set* (*concat cs'*)
by (*metis Un-iff* ‹*b' ≠ []*› *list.set-sel(1) not-less-iff-gr-or-eq set-interact sorted-wrt-append sw-ab'*)
have *strict-sorted* (*a' @ concat cs'*)
using *eq'(1) ss-ab'(1)* **by** *force*
then have *b-cs'*: *strict-sorted* (*b' @ concat cs'*)
using ‹*b' < concat cs'*› *eq'(2) ss-ab'(2) strict-sorted-append-iff* **by** *auto*
have *list.set a' = list.set* (*concat as'*) ∩ {*..< hd b'*}
proof −
have *x ∈ list.set a'*

```
      if x < hd b' and l ∈ list.set cs' and x ∈ list.set l for x l
      using b-cs' sorted-hd-le strict-sorted-imp-sorted that by fastforce
    then show ?thesis
    using ‹b' ≠ []› sw-ab' by (force simp: strict-sorted-append-iff sorted-wrt-append
eq')
  qed
  ultimately have a=a'
    by (simp add: SUC.prems(1) ‹hd b' = hd b› ‹strict-sorted a'› ‹strict-sorted
a› strict-sorted-equal)
  moreover
  have ccat-cs-cs': concat cs = concat cs'
    using SUC.prems(1) ‹a = a'› eq'(1) eq(1) by fastforce
  have b=b'
  proof (cases ds = [] ∨ ds' = [])
    case True
    then show ?thesis
      using SUC eq'(2) eq(2) by fastforce
  next
    case False
    then have ds ≠ [] ds' ≠ [] sorted (concat ds) sorted (concat ds')
    using eq(2) ss-ab(2) eq'(2) ss-ab'(2) strict-sorted-append-iff strict-sorted-imp-sorted
by auto
    have strict-sorted b strict-sorted b'
      using b-cs b-cs' sorted-wrt-append by auto
    moreover
    have cs ≠ []
      using k local.Suc by auto
    then obtain hd cs ≠ [] hd ds ≠ []
      using SUC.prems(3) SUC.prems(4) eq list.set-sel(1)
      by (simp add: ‹ds ≠ []› mem-lists-non-Nil)
    then have concat cs ≠ []
      using ‹cs ≠ []› hd-in-set by auto
    have hd (concat cs) < hd (concat ds)
      using strict-sorted-interact-hd
    by (metis ‹cs ≠ []› ‹ds ≠ []› ‹hd cs ≠ []› ‹hd ds ≠ []› hd-concat sorted-wrt-append
sw-ab)

    have list.set b = list.set (concat bs) ∩ {..< hd (concat cs)}
    proof −
      have 1: x ∈ list.set b
        if x < hd (concat cs) and l ∈ list.set ds and x ∈ list.set l for x l
        using ‹hd (concat cs) < hd (concat ds)› ‹sorted (concat ds)› sorted-hd-le
that by fastforce
      have 2: l < hd (concat cs) if l ∈ list.set b for l
      by (metis ‹concat cs ≠ []› b-cs concat.simps(2) list.set-sel(1) sorted-wrt-append
that)
      show ?thesis
        using 1 2 by (auto simp: strict-sorted-append-iff sorted-wrt-append eq)
    qed
```

```
    moreover
    have cs' ≠ []
      using k Suc ‹concat cs ≠ []› ccat-cs-cs' by auto
    then obtain hd cs' ≠ [] hd ds' ≠ []
      using SUC.prems(8,9) ‹ds' ≠ []› eq'(1) eq'(2) list.set-sel(1) by auto
    then have concat cs' ≠ []
      using ‹cs' ≠ []› hd-in-set by auto
    have hd (concat cs') < hd (concat ds')
      using strict-sorted-interact-hd
        by (metis ‹cs' ≠ []› ‹ds' ≠ []› ‹hd cs' ≠ []› ‹hd ds' ≠ []› hd-concat
sorted-wrt-append sw-ab')
    have list.set b' = list.set (concat bs') ∩ {..< hd (concat cs')}
    proof −
      have 1: x ∈ list.set b'
        if x < hd (concat cs') and l ∈ list.set ds' and x ∈ list.set l for x l
       using ‹hd (concat cs') < hd (concat ds')› ‹sorted (concat ds')› sorted-hd-le
that by fastforce
      have 2: l < hd (concat cs') if l ∈ list.set b' for l
        by (metis ‹concat cs' ≠ []› b-cs' list.set-sel(1) sorted-wrt-append that)
      show ?thesis
        using 1 2 by (auto simp: strict-sorted-append-iff sorted-wrt-append eq')
    qed
    ultimately show b = b'
      by (simp add: SUC.prems(2) ccat-cs-cs' strict-sorted-equal)
  qed
  moreover
  have cs = cs' ∧ ds = ds'
  proof (rule SUC.hyps)
    show k = length cs
      using eq SUC.hyps(2) by auto[1]
    show concat ds = concat ds'
      using SUC.prems(2) ‹b = b'› eq'(2) eq(2) by auto
    show strict-sorted (interact cs ds)
      using eq SUC.prems(5) strict-sorted-append-iff by auto
    show length ds ≤ length cs length cs ≤ Suc (length ds)
      using eq SUC k by auto
    show strict-sorted (interact cs' ds')
      using eq' SUC.prems(10) strict-sorted-append-iff by auto
    show length cs = length cs'
      using SUC eq'(1) k(1) by force
  qed (use ccat-cs-cs' eq eq' SUC.prems in auto)
  ultimately show ?thesis
    by (simp add: ‹a = a'› ‹b = b'› eq eq')
 qed
qed


proposition Form-Body-unique:
 assumes Form-Body ka kb xs ys zs Form-Body ka kb xs ys zs' and kb ≤ ka ka ≤
```

Suc kb
shows *zs′ = zs*
proof −
obtain *a as b bs c d*
where *xs*: *xs = concat (a#as)* **and** *ys*: *ys = concat (b#bs)*
and *ne*: *a#as ∈ lists (− {[]}) b#bs ∈ lists (− {[]})*
and *len*: *length (a#as) = ka length (b#bs) = kb*
and *c*: *c = acc-lengths 0 (a#as)*
and *d*: *d = acc-lengths 0 (b#bs)*
and *Ueq*: *zs = concat [c, a, d, b] @ interact as bs*
and *ss-zs*: *strict-sorted zs*
using *Form-Body.cases [OF assms(1)]* **by** *(metis (no-types))*
obtain *a′ as′ b′ bs′ c′ d′*
where *xs′*: *xs = concat (a′#as′)* **and** *ys′*: *ys = concat (b′#bs′)*
and *ne′*: *a′#as′ ∈ lists (− {[]}) b′#bs′ ∈ lists (− {[]})*
and *len′*: *length (a′#as′) = ka length (b′#bs′) = kb*
and *c′*: *c′ = acc-lengths 0 (a′#as′)*
and *d′*: *d′ = acc-lengths 0 (b′#bs′)*
and *Ueq′*: *zs′ = concat [c′, a′, d′, b′] @ interact as′ bs′*
and *ss-zs′*: *strict-sorted zs′*
using *Form-Body.cases [OF assms(2)]* **by** *(metis (no-types))*
have *[simp]*: *length c = length c′ length d = length d′*
using *c c′ d d′ len′ len* **by** *auto*
note *acc-lengths.simps [simp del]*
have *a < b*
using *ss-zs* **by** *(auto simp: Ueq strict-sorted-append-iff less-list-def c d)*
have *a′ < b′*
using *ss-zs′* **by** *(auto simp: Ueq′ strict-sorted-append-iff less-list-def c′ d′)*
have *a#as = a′#as′ ∧ b#bs = b′#bs′*
proof *(rule interaction-scheme-unique-aux)*
show *strict-sorted (interact (a # as) (b # bs))*
using *ss-zs ‹a < b›* **by** *(auto simp: Ueq strict-sorted-append-iff less-list-def d)*
show *strict-sorted (interact (a′ # as′) (b′ # bs′))*
using *ss-zs′ ‹a′ < b′›* **by** *(auto simp: Ueq′ strict-sorted-append-iff less-list-def d′)*
show *length (b # bs) ≤ length (a # as) length (b′ # bs′) ≤ length (a′ # as′)*
using *‹kb ≤ ka› len len′* **by** *auto*
show *length (a # as) ≤ Suc (length (b # bs))*
using *‹ka ≤ Suc kb› len* **by** *linarith*
then show *length (a′ # as′) ≤ Suc (length (b′ # bs′))*
using *len len′* **by** *fastforce*
qed *(use len len′ xs xs′ ys ys′ ne ne′* **in** *fastforce)+*
then show *?thesis*
using *Ueq Ueq′ c c′ d d′* **by** *blast*
qed

lemma *Form-Body-imp-inter-scheme*:
assumes *FB*: *Form-Body ka kb xs ys zs* **and** *0 < kb kb ≤ ka ka ≤ Suc kb*

```
  shows zs = inter-scheme ((ka+kb) − Suc 0) {xs,ys}
proof −
  have length xs < length ys
    by (meson Form-Body-length assms(1))
  have [simp]: a + a = b + b ⟷ a=b  a + a − Suc 0 = b + b − Suc 0 ⟷
a=b for a b::nat
    by auto
  show ?thesis
  proof (cases ka = kb)
    case True
    show ?thesis
      unfolding inter-scheme-def
      apply (rule some-equality [symmetric], metis One-nat-def True FB mult-2)
      using assms ‹length xs < length ys›
        by (auto simp: True mult-2 Set.doubleton-eq-iff Form-Body-unique dest:
Form-Body-length, presburger)
  next
    case False
    then have eq: ka = Suc kb
      using assms by linarith
    show ?thesis
      unfolding inter-scheme-def
      apply (rule some-equality [symmetric], use assms False mult-2 one-is-add eq
in fastforce)
      using assms ‹length xs < length ys›
    by (auto simp: eq mult-2 Set.doubleton-eq-iff Form-Body-unique dest: Form-Body-length,
presburger)
  qed
qed
```

3.7 For Lemma 3.8 AND PROBABLY 3.7

```
definition grab :: nat set ⇒ nat ⇒ nat set × nat set
  where grab N n ≡ (N ∩ enumerate N ‘ {..<n}, N ∩ {enumerate N n..})

lemma grab-0 [simp]: grab N 0 = ({}, N)
  by (fastforce simp: grab-def enumerate-0 Least-le)

lemma less-sets-grab:
  infinite N ⟹ fst (grab N n) ≪ snd (grab N n)
  by (auto simp: grab-def less-sets-def intro: enumerate-mono less-le-trans)

lemma finite-grab [iff]: finite (fst (grab N n))
  by (simp add: grab-def)

lemma card-grab [simp]:
  assumes infinite N shows card (fst (grab N n)) = n
proof −
  have N ∩ enumerate N ‘ {..<n} = enumerate N ‘ {..<n}
```

using *assms* **by** (*auto simp*: *enumerate-in-set*)
with *assms* **show** *?thesis*
by (*simp add*: *card-image grab-def strict-mono-enum strict-mono-imp-inj-on*)
qed

lemma *fst-grab-subset*: *fst* (*grab N n*) ⊆ *N*
using *grab-def range-enum* **by** *fastforce*

lemma *snd-grab-subset*: *snd* (*grab N n*) ⊆ *N*
by (*auto simp*: *grab-def*)

lemma *grab-Un-eq*:
assumes *infinite N* **shows** *fst* (*grab N n*) ∪ *snd* (*grab N n*) = *N*
proof
show *N* ⊆ *fst* (*grab N n*) ∪ *snd* (*grab N n*)
unfolding *grab-def*
using *assms enumerate-Ex le-less-linear strict-mono-enum strict-mono-less* **by** *fastforce*
qed (*simp add*: *grab-def*)

lemma *finite-grab-iff* [*simp*]: *finite* (*snd* (*grab N n*)) ⟷ *finite N*
by (*metis finite-grab grab-Un-eq infinite-Un infinite-super snd-grab-subset*)

lemma *grab-eqD*:
⟦*grab N n* = (*A*,*M*); *infinite N*⟧
⟹ *A* ≪ *M* ∧ *finite A* ∧ *card A* = *n* ∧ *infinite M* ∧ *A* ⊆ *N* ∧ *M* ⊆ *N*
using *card-grab grab-def less-sets-grab finite-grab-iff* **by** *auto*

lemma *less-sets-fst-grab*: *A* ≪ *N* ⟹ *A* ≪ *fst* (*grab N n*)
by (*simp add*: *fst-grab-subset less-sets-weaken2*)

Possibly redundant, given *grab*

definition *nxt* **where** *nxt* ≡ λ*N*. λ*n*::*nat*. *N* ∩ {*n*<..}

lemma *infinite-nxtN*: *infinite N* ⟹ *infinite* (*nxt N n*)
by (*simp add*: *infinite-nat-greaterThan nxt-def*)

lemma *nxt-subset*: *nxt N n* ⊆ *N*
unfolding *nxt-def* **by** *blast*

lemma *nxt-subset-greaterThan*: *m* ≤ *n* ⟹ *nxt N n* ⊆ {*m*<..}
by (*auto simp*: *nxt-def*)

lemma *nxt-subset-atLeast*: *m* ≤ *n* ⟹ *nxt N n* ⊆ {*m*..}
by (*auto simp*: *nxt-def*)

lemma *enum-nxt-ge*: *infinite N* ⟹ *a* ≤ *enum* (*nxt N a*) *n*
by (*simp add*: *atLeast-le-enum infinite-nxtN nxt-subset-atLeast*)

lemma *inj-enum-nxt*: *infinite N* $\Longrightarrow$ *inj-on* (*enum* (*nxt N a*)) *A*
by (*simp add*: *infinite-nxtN strict-mono-enum strict-mono-imp-inj-on*)

3.8 Larson's Lemma 3.11

Again from Jean A. Larson, A short proof of a partition theorem for the ordinal ω^ω. *Annals of Mathematical Logic*, 6:129–145, 1973.

lemma *lemma-3-11*:
assumes *l* > *0*
shows *thin* (*inter-scheme l* ' {*U*. *Form l U*})
using *form-cases* [*of l*]
proof *cases*
case *zero*
then show *?thesis*
using *assms* **by** *auto*
next
case (*nz ka kb*)
note *acc-lengths.simps* [*simp del*]
show *?thesis*
unfolding *thin-def*
proof *clarify*
fix *U U'*
assume *ne*: *inter-scheme l U* ≠ *inter-scheme l U'* **and** *init*: *initial-segment* (*inter-scheme l U*) (*inter-scheme l U'*)
assume *Form l U*
then obtain *kp kq xs ys* **where** *l* = *kp*+*kq* − *1 U* = {*xs*,*ys*}
and *U*: *Form-Body kp kq xs ys* (*inter-scheme l U*) **and** *0* < *kq kq* ≤ *kp kp* ≤ *Suc kq*
using *assms inter-scheme* **by** *blast*
then have *kp* = *ka* ∧ *kq* = *kb*
using *nz* **by** *linarith*
then obtain *a as b bs c d*
where *len*: *length* (*a*#*as*) = *ka length* (*b*#*bs*) = *kb*
and *c*: *c* = *acc-lengths 0* (*a*#*as*)
and *d*: *d* = *acc-lengths 0* (*b*#*bs*)
and *Ueq*: *inter-scheme l U* = *concat* [*c*, *a*, *d*, *b*] @ *interact as bs*
using *U* **by** (*auto simp*: *Form-Body.simps*)
assume *Form l U'*
then obtain *kp' kq' xs' ys'* **where** *l* = *kp'*+*kq'* − *1 U'* = {*xs'*,*ys'*}
and *U'*: *Form-Body kp' kq' xs' ys'* (*inter-scheme l U'*) **and** *0* < *kq' kq'* ≤ *kp' kp'* ≤ *Suc kq'*
using *assms inter-scheme* **by** *blast*
then have *kp'* = *ka* ∧ *kq'* = *kb*
using *nz* **by** *linarith*
then obtain *a' as' b' bs' c' d'*
where *len'*: *length* (*a'*#*as'*) = *ka length* (*b'*#*bs'*) = *kb*
and *c'*: *c'* = *acc-lengths 0* (*a'*#*as'*)
and *d'*: *d'* = *acc-lengths 0* (*b'*#*bs'*)
and *Ueq'*: *inter-scheme l U'* = *concat* [*c'*, *a'*, *d'*, *b'*] @ *interact as' bs'*

using U' **by** (*auto simp*: *Form-Body.simps*)
have [*simp*]: *length bs'* = *length bs length as'* = *length as*
using *len len'* **by** *auto*
have *inter-scheme l U* ≠ [] *inter-scheme l U'* ≠ []
using *Form-Body-nonempty U U'* **by** *auto*
define *u1* **where** *u1* ≡ *hd* (*inter-scheme l U*)
have *u1-eq'*: *u1* = *hd* (*inter-scheme l U'*)
using ‹*inter-scheme l U* ≠ []› *init u1-def initial-segment-ne* **by** *fastforce*
have *au1*: *u1* = *length a*
by (*simp add*: *u1-def Ueq c*)
have *au1'*: *u1* = *length a'*
by (*simp add*: *u1-eq' Ueq' c'*)
have *len-eqk*: *length c'* = *ka length d'* = *kb length c'* = *ka length d'* = *kb*
using *c d len c' d' len'* **by** *auto*
have *take*: *take* (*ka* + *u1* + *kb*) (*c* @ *a* @ *d* @ *l*) = *c* @ *a* @ *d*
take (*ka* + *u1* + *kb*) (*c'* @ *a'* @ *d'* @ *l*) = *c'* @ *a'* @ *d'* **for** *l*
using *c d c' d' len* **by** (*simp-all flip*: *au1 au1'*)
have *leU*: *ka* + *u1* + *kb* ≤ *length* (*inter-scheme l U*)
using *c d len* **by** (*simp add*: *au1 Ueq*)
then have *take* (*ka* + *u1* + *kb*) (*inter-scheme l U*) = *take* (*ka* + *u1* + *kb*) (*inter-scheme l U'*)
using *take-initial-segment init* **by** *blast*
then have §: *c* @ *a* @ *d* = *c'* @ *a'* @ *d'*
by (*metis Ueq Ueq' append.assoc concat.simps*(*2*) *take*)
have *length* (*inter-scheme l U*) = *ka* + (*c* @ *a* @ *d*)!(*ka*−*1*) + *kb* + *last d*
by (*simp add*: *Ueq c d length-interact nth-append flip*: *len*)
moreover have *length* (*inter-scheme l U'*) = *ka* + (*c'* @ *a'* @ *d'*)!(*ka*−*1*) + *kb* + *last d'*
by (*simp add*: *Ueq' c' d' length-interact nth-append flip*: *len'*)
moreover have *last d* = *last d'*
using § *c d d' len'*(*1*) *len-eqk*(*1*) **by** *auto*
ultimately have *length* (*inter-scheme l U*) = *length* (*inter-scheme l U'*)
by (*simp add*: §)
then show *False*
using *init initial-segment-length-eq ne* **by** *blast*
qed
qed

3.9 Larson's Lemma 3.6

proposition *lemma-3-6*:
fixes *g* :: *nat list set* ⇒ *nat*
assumes *g*: *g* ∈ [*WW*]^2 → {*0*,*1*}
obtains *N j* **where** *infinite N*
and ⋀*k u*. ⟦*k* > *0*; *u* ∈ [*WW*]^2; *Form k u*; [*enum N k*] < *inter-scheme k u*; *List.set* (*inter-scheme k u*) ⊆ *N*⟧ ⟹ *g u* = *j k*
proof −
define Φ **where** Φ ≡ λ*m*::*nat*. λ*M*. *infinite M* ∧ *m* < *Inf M*
define Ψ **where** Ψ ≡ λ*l m n*::*nat*. λ*M N j*. *n* > *m* ∧ *N* ⊆ *M* ∧ *n* ∈ *M*

```
                ∧ (∀ U. Form l U ∧ U ⊆ WW ∧ [n] < inter-scheme l U ∧ list.set
(inter-scheme l U) ⊆ N ⟶ g U = j)
  have ∗: ∃ n N j. Φ n N ∧ Ψ l m n M N j if l > 0 Φ m M for l m::nat and M
:: nat set
  proof −
    define FF where FF ≡ {U ∈ [WW]^2. Form l U}
    define h where h ≡ λzs. g (inv-into FF (inter-scheme l) zs)
    have thin (inter-scheme l ‘ FF)
      using ‹l > 0› lemma-3-11 by (simp add: thin-def FF-def)
    moreover
    have inter-scheme l ‘ FF ⊆ WW
      using inter-scheme-simple ‹0 < l› FF-def by blast
    moreover
    have h ‘ {xs ∈ inter-scheme l ‘ FF. List.set xs ⊆ M} ⊆ {..<2}
      using g inv-into-into[of concl: FF inter-scheme l]
      by (force simp: h-def FF-def Pi-iff)
    ultimately
    obtain j N where j < 2 infinite N N ⊆ M and hj: h ‘ {xs ∈ inter-scheme l ‘
FF. List.set xs ⊆ N} ⊆ {j}
       using ‹Φ m M› unfolding Φ-def by (blast intro: Nash-Williams-WW [of
M])
    let ?n = Inf N
    have ?n > m
    using ‹Φ m M› ‹infinite N› unfolding Φ-def Inf-nat-def infinite-nat-iff-unbounded
      by (metis LeastI-ex ‹N ⊆ M› le-less-trans not-less not-less-Least subsetD)
      have g U = j if Form l U U ⊆ WW [?n] < inter-scheme l U list.set
(inter-scheme l U) ⊆ N − {?n} for U
    proof −
      obtain xs ys where xys: xs ≠ ys U = {xs,ys}
        using Form-elim-upair ‹Form l U› by blast
      moreover have inj-on (inter-scheme l) FF
        using ‹0 < l› inj-on-def inter-scheme-injective FF-def by blast
      moreover
      have g (inv-into FF (inter-scheme l) (inter-scheme l U)) = j
      using hj that xys subset-Diff-insert by (fastforce simp: h-def FF-def image-iff)
      ultimately show ?thesis
        using that FF-def by auto
    qed
    moreover have ?n < Inf (N − {?n})
       by (metis Diff-iff Inf-nat-def Inf-nat-def1 ‹infinite N› finite.emptyI infi-
nite-remove linorder-neqE-nat not-less-Least singletonI)
    moreover have ?n ∈ M
      by (metis Inf-nat-def1 ‹N ⊆ M› ‹infinite N› finite.emptyI subsetD)
    ultimately have Φ ?n (N − {?n}) ∧ Ψ l m ?n M (N − {?n}) j
     using ‹Φ m M› ‹infinite N› ‹N ⊆ M› ‹?n > m› by (auto simp: Φ-def Ψ-def)
    then show ?thesis
      by blast
  qed
  have base: Φ 0 {0<..}
```

unfolding Φ-*def* **by** (*metis infinite-Ioi Inf-nat-def1 greaterThan-iff greaterThan-non-empty*)
have *step*: *Ex* (λ(*n*,*N*,*j*). Φ *n N* $\wedge$ Ψ *l m n M N j*) **if** Φ *m M l* > *0* **for** *m M l*
using $*$ [*of l m M*] *that* **by** (*auto simp*: Φ-*def*)
define *G* **where** *G* $\equiv$ λ*l m M*. @(*n*,*N*,*j*). Φ *n N* $\wedge$ Ψ (*Suc l*) *m n M N j*
have *G*Φ: (λ(*n*,*N*,*j*). Φ *n N*) (*G l m M*) **and** *G*Ψ: (λ(*n*,*N*,*j*). Ψ (*Suc l*) *m n M N j*) (*G l m M*)
if Φ *m M* **for** *l m M*
using *step* [*OF that*, *of Suc l*] **by** (*force simp*: *G-def dest*: *some-eq-imp*)+
have *G-increasing*: (λ(*n*,*N*,*j*). *n* > *m* $\wedge$ *N* $\subseteq$ *M* $\wedge$ *n* $\in$ *M*) (*G l m M*) **if** Φ *m M* **for** *l m M*
using *G*Ψ [*OF that*, *of l*] *that* **by** (*simp add*: Ψ-*def split*: *prod.split-asm*)
define *H* **where** *H* $\equiv$ *rec-nat* (*0*,{*0*<..},*0*) (λ*l* (*m*,*M*,*j*). *G l m M*)
have *H-simps*: *H 0* = (*0*,{*0*<..},*0*) $\bigwedge$*l*. *H* (*Suc l*) = (*case H l of* (*m*,*M*,*j*) $\Rightarrow$ *G l m M*)
by (*simp-all add*: *H-def*)
have *H*Φ: (λ(*n*,*N*,*j*). Φ *n N*) (*H l*) **for** *l*
by (*induction l*) (*use base G*Φ **in** ‹*auto simp*: *H-simps split*: *prod.split-asm*›)
define ν **where** ν $\equiv$ (λ*l*. *case H l of* (*n*,*M*,*j*) $\Rightarrow$ *n*)
have *H-inc*: ν *l* $\geq$ *l* **for** *l*
proof (*induction l*)
case (*Suc l*)
then show *?case*
using *H*Φ [*of l*] *G-increasing* [*of* ν *l*]
apply (*clarsimp simp*: *H-simps* ν-*def split*: *prod.split*)
by (*metis* (*no-types*, *lifting*) *case-prodD leD le-less-trans not-less-eq-eq*)
qed *auto*
let *?N* = *range* ν
define *j* **where** *j* $\equiv$ λ*l*. *case H l of* (*n*,*M*,*j*) $\Rightarrow$ *j*
have *H-increasing-Suc*: (*case H k of* (*n*, *N*, *j'*) $\Rightarrow$ *N*) $\supseteq$ (*case H* (*Suc k*) *of* (*n*, *N*, *j'*) $\Rightarrow$ *insert n N*) **for** *k*
using *H*Φ [*of k*]
by (*force simp*: *H-simps split*: *prod.split dest*: *G-increasing* [**where** *l*=*k*])
have *H-increasing-superset*: (*case H k of* (*n*, *N*, *j'*) $\Rightarrow$ *N*) $\supseteq$ (*case H* (*n*+*k*) *of* (*n*, *N*, *j'*) $\Rightarrow$ *N*) **for** *k n*
proof (*induction n*)
case (*Suc n*)
then show *?case*
using *H-increasing-Suc* [*of n*+*k*] **by** (*auto split*: *prod.split-asm*)
qed *auto*
then have *H-increasing-less*: (*case H k of* (*n*, *N*, *j'*) $\Rightarrow$ *N*) $\supseteq$ (*case H l of* (*n*, *N*, *j'*) $\Rightarrow$ *insert n N*)
if *k*<*l* **for** *k l*
by (*smt* (*verit*, *best*) *H-increasing-Suc add.commute less-natE order-trans that*)
have ν *k* < ν (*Suc k*) **for** *k*
using *H*Φ [*of k*] **unfolding** ν-*def*
by (*auto simp*: *H-simps split*: *prod.split dest*: *G-increasing* [**where** *l*=*k*])
then have *strict-mono-*ν: *strict-mono* ν
by (*simp add*: *strict-mono-Suc-iff*)
then have *enum-N*: *enum ?N* = ν

```
    by (metis enum-works nat-infinite-iff range-strict-mono-ext)
  have **: ?N ∩ {n<..} ⊆ N' if H: H k = (n, N', j) for n N' k j
  proof clarify
    fix l
    assume n < ν l
    then have False if l ≤ k
      using that strict-monoD [OF strict-mono-ν, of l k ] H by (force simp: ν-def)
    then have k < l using not-less by blast
    then obtain M j where Mj: H l = (ν l,M,j)
      unfolding ν-def
      by (metis (mono-tags, lifting) case-prod-conv old.prod.exhaust)
    then show ν l ∈ N'
      using that H-increasing-less [OF ‹k<l›] Mj by auto
  qed
  show thesis
  proof
    show infinite (?N::nat set)
      using H-inc infinite-nat-iff-unbounded-le by auto
  next
    fix l U
    assume 0 < l and U: U ∈ [WW]^2
      and interU: [enum ?N l] < inter-scheme l U Form l U
      and sub: list.set (inter-scheme l U) ⊆ ?N
    obtain k where k: l = Suc k
      using ‹0 < l› gr0-conv-Suc by blast
    have g U = v if H k = (m, M, j0) and G k m M = (n, N', v)
      for m M j0 n N' v
    proof −
      have n: ν (Suc k) = n
        using that by (simp add: ν-def H-simps)
      have {..enum (range ν) l} ∩ list.set (inter-scheme l U) = {}
     using inter-scheme-strict-sorted ‹0 < l› interU singleton-less-list-iff strict-sorted-iff
by blast
      then have list.set (inter-scheme (Suc k) U) ⊆ N'
        using that sub ** [of Suc k n N' v] Suc-le-eq not-less-eq-eq
        by (fastforce simp:  k n enum-N H-simps)
      then show ?thesis
        using that interU U GΨ [of m M k] HΦ [of k]
        by (auto simp: Ψ-def k enum-N H-simps n nsets-def)
    qed
    with U show g U = j l
      by (auto simp: k j-def H-simps split: prod.split)
  qed
qed
```

3.10 Larson's Lemma 3.7

3.10.1 Preliminaries

Analogous to *ordered-nsets-2-eq*, but without type classes

lemma *total-order-nsets-2-eq*:
assumes *tot*: *total-on A r* **and** *irr*: *irrefl r*
shows *nsets A 2* = {{*x*,*y*} | *x y*. *x* ∈ *A* ∧ *y* ∈ *A* ∧ (*x*,*y*) ∈ *r*}
(**is** - = *?rhs*)
proof
show *nsets A 2* ⊆ *?rhs*
using *tot*
unfolding *numeral-nat total-on-def nsets-def*
by (*fastforce simp*: *card-Suc-eq Set.doubleton-eq-iff not-less*)
show *?rhs* ⊆ *nsets A 2*
using *irr* **unfolding** *numeral-nat* **by** (*force simp*: *nsets-def card-Suc-eq irrefl-def*)
qed

lemma *lenlex-nsets-2-eq*: *nsets A 2* = {{*x*,*y*} | *x y*. *x* ∈ *A* ∧ *y* ∈ *A* ∧ (*x*,*y*) ∈ *lenlex less-than*}
using *total-order-nsets-2-eq* **by** (*simp add*: *total-order-nsets-2-eq irrefl-def*)

lemma *sum-sorted-list-of-set-map*: *finite I* ⟹ *sum-list* (*map f* (*list-of I*)) = *sum f I*
proof (*induction card I arbitrary*: *I*)
case (*Suc n I*)
then have [*simp*]: *I* ≠ {}
by *auto*
have *sum-list* (*map f* (*list-of* (*I* − {*Min I*}))) = *sum f* (*I* − {*Min I*})
using *Suc* **by** *auto*
then show *?case*
using *Suc.prems sum.remove* [*of I Min I f*]
by (*simp add*: *sorted-list-of-set-nonempty Suc*)
qed *auto*

lemma *sorted-list-of-set-UN-eq-concat*:
assumes *I*: *strict-mono-sets I f finite I* **and** *fin*: ⋀*i*. *finite* (*f i*)
shows *list-of* (⋃ *i* ∈ *I*. *f i*) = *concat* (*map* (*list-of* ∘ *f*) (*list-of I*))
using *I*
proof (*induction card I arbitrary*: *I*)
case (*Suc n I*)
then have *I* ≠ {} **and** *Iexp*: *I* = *insert* (*Min I*) (*I* − {*Min I*})
using *Min-in Suc.hyps*(*2*) *Suc.prems*(*2*) **by** *fastforce*+
have *IH*: *list-of* (⋃ (*f* ‘ (*I* − {*Min I*}))) = *concat* (*map* (*list-of* ∘ *f*) (*list-of* (*I* − {*Min I*})))
using *Suc* **unfolding** *strict-mono-sets-def*
by (*metis DiffE Iexp card-Diff-singleton diff-Suc-1 finite-Diff insertI1*)
have *list-of* (⋃ (*f* ‘ *I*)) = *list-of* (⋃ (*f* ‘ (*insert* (*Min I*) (*I* − {*Min I*}))))
using *Iexp* **by** *auto*
also have ... = *list-of* (*f* (*Min I*) ∪ ⋃ (*f* ‘ (*I* − {*Min I*})))
by (*metis Union-image-insert*)
also have ... = *list-of* (*f* (*Min I*)) @ *list-of* (⋃ (*f* ‘ (*I* − {*Min I*})))

proof (*rule sorted-list-of-set-Un*)
show *f* (*Min I*) ≪ ⋃ (*f* ‘ (*I* − {*Min I*}))
using *Suc.prems* ‹*I* ≠ {}› *strict-mono-less-sets-Min* **by** *blast*
show *finite* (⋃ (*f* ‘ (*I* − {*Min I*})))
by (*simp add*: ‹*finite I*› *fin*)
qed (*use fin* **in** *auto*)
also have ... = *list-of* (*f* (*Min I*)) @ *concat* (*map* (*list-of* ∘ *f*) (*list-of* (*I* − {*Min I*})))
using *IH* **by** *metis*
also have ... = *concat* (*map* (*list-of* ∘ *f*) (*list-of I*))
by (*simp add*: *Suc.prems*(*2*) ‹*I* ≠ {}› *sorted-list-of-set-nonempty*)
finally show *?case* **.**
qed *auto*

3.10.2 Lemma 3.7 of Jean A. Larson, ibid.

proposition *lemma-3-7*:
assumes *infinite N l* > *0*
obtains *M* **where** *M* ∈ [*WW*]^m
⋀*U*. *U* ∈ [*M*]^2 ⟹ *Form l U* ∧ *List.set* (*inter-scheme l U*) ⊆ *N*
proof (*cases m* < *2*)
case *True*
obtain *w* **where** *w*: *w* ∈ *WW*
using *WW-def strict-sorted-into-WW* **by** *auto*
define *M* **where** *M* ≡ *if m=0 then* {} *else* {*w*}
have *M*: *M* ∈ [*WW*]^m
using *True* **by** (*auto simp*: *M-def nsets-def w*)
have [*simp*]: [*M*]^2 = {}
using *True* **by** (*auto simp*: *M-def nsets-def w dest*: *subset-singletonD*)
show *?thesis*
using *M that* **by** *fastforce*
next
case *False*
then have *m* ≥ *2*
by *auto*
have *nonz*: (*enum N* ∘ *Suc*) *i* > *0* **for** *i*
using *assms*(*1*) *le-enumerate less-le-trans* **by** *fastforce*
note *infinite-nxt-N* = *infinite-nxtN* [*OF* ‹*infinite N*›, *iff*]
note ‹*infinite N*› [*iff*]
have [*simp*]: {*n*<..<*Suc n*} = {} {..<*1*::*nat*} = {*0*} **for** *n*
by *auto*
note *One-nat-def* [*simp del*]

define *DF-Suc* **where** *DF-Suc* ≡ λ*k D*. *enum* (*nxt N* (*enum* (*nxt N* (*Max D*)) (*Inf D* − *1*))) ‘ {..<*Suc k*}
define *DF* **where** *DF* ≡ λ*k n*. (*DF-Suc k* ^^ *n*) ((*enum N* ∘ *Suc*) ‘ {..<*Suc k*})
have *DF-simps*: *DF k 0* = (*enum N* ∘ *Suc*) ‘ {..<*Suc k*} *DF k* (*Suc i*) = *DF-Suc k* (*DF k i*) **for** *i k*
by (*auto simp*: *DF-def*)

```
  have card-DF: card (DF k i) = Suc k for i k
  proof (induction i)
    case 0
    have inj-on (enum N ∘ Suc) {..<Suc k}
      by (simp add: assms(1) strict-mono-def strict-mono-imp-inj-on)
    with 0 show ?case
      using card-image DF-simps by fastforce
  next
    case (Suc i)
    then show ?case
      by (simp add: ‹infinite N› DF-simps DF-Suc-def card-image infinite-nxtN
strict-mono-enum strict-mono-imp-inj-on)
  qed
  have DF-ne: DF k i ≠ {} for i k
    by (metis card-DF card-lessThan lessThan-empty-iff nat.simps(3))

  have finite-DF: finite (DF k i) for i k
    by (induction i) (auto simp: DF-simps DF-Suc-def)
  have DF-Suc: DF k i ≪ DF k (Suc i) for i k
    unfolding less-sets-def
    by (force simp: finite-DF DF-simps DF-Suc-def
      intro!: greaterThan-less-enum nxt-subset-greaterThan atLeast-le-enum nxt-subset-atLeast
infinite-nxtN [OF ‹infinite N›])
  have DF-DF: DF k i ≪ DF k j if i<j for i j k
   by (meson DF-Suc DF-ne UNIV-I less-sets-imp-strict-mono-sets strict-mono-setsD
that)
  then have sm-DF: strict-mono-sets UNIV (DF k) for k
    by (simp add: strict-mono-sets-def)

  have DF-gt0: 0 < Inf (DF k i) for i k
  proof (cases i)
    case 0
    then show ?thesis
      by (metis DF-ne DF-simps(1) Inf-nat-def1 imageE nonz)
  next
    case (Suc n)
    then show ?thesis
      by (metis DF-Suc DF-ne Inf-nat-def1 gr0I gr-implies-not0 less-sets-def)
  qed
  have DF-N: DF k i ⊆ N for i k
  proof (induction i)
    case 0
    then show ?case
      using ‹infinite N› range-enum by (auto simp: DF-simps)
  next
    case (Suc i)
    then show ?case
      unfolding DF-simps DF-Suc-def image-subset-iff
```

by (*metis IntE ‹infinite N› enumerate-in-set infinite-nxtN nxt-def*)
qed

have *sm-enum-DF*: *strict-mono-on* {..*k*} (*enum* (*DF k i*)) **for** *k i*
by (*metis card-DF enum-works-finite finite-DF lessThan-Suc-atMost*)

define *AF* **where** *AF* ≡ λ*k i*. *enum* (*nxt N* (*Max* (*DF k i*))) ‘ {..<*Inf* (*DF k i*)}
have *AF-ne*: *AF k i* ≠ {} **for** *i k*
by (*auto simp*: *AF-def lessThan-empty-iff DF-gt0*)
have *finite-AF* [*simp*]: *finite* (*AF k i*) **for** *i k*
by (*simp add*: *AF-def*)
have *card-AF*: *card* (*AF k i*) = ∏ (*DF k i*) **for** *k i*
by (*simp add*: *AF-def card-image inj-enum-nxt*)

have *DF-AF*: *DF k i* ≪ *AF k i* **for** *i k*
unfolding *less-sets-def AF-def*
by (*simp add*: *finite-DF greaterThan-less-enum nxt-subset-greaterThan*)

have *E*: ⟦*x* ≤ *y*; *infinite M*⟧ ⟹ *enum M x* < *enum* (*nxt N* (*enum M y*)) *z* **for** *x y z M*
by (*metis infinite-nxt-N dual-order.eq-iff enumerate-mono greaterThan-less-enum nat-less-le nxt-subset-greaterThan*)

have *AF-DF-Suc*: *AF k i* ≪ *DF k* (*Suc i*) **for** *i k*
by (*auto simp*: *DF-simps DF-Suc-def less-sets-def AF-def E*)

have *AF-DF*: *AF k p* ≪ *DF k q* **if** *p*<*q* **for** *k p q*
by (*metis AF-DF-Suc DF-ne Suc-lessI UNIV-I less-sets-trans sm-DF strict-mono-sets-def that*)

have *AF-Suc*: *AF k i* ≪ *AF k* (*Suc i*) **for** *i k*
using *AF-DF-Suc DF-AF DF-ne less-sets-trans* **by** *blast*
then have *sm-AF*: *strict-mono-sets UNIV* (*AF k*) **for** *k*
by (*simp add*: *AF-ne less-sets-imp-strict-mono-sets*)

define *del* **where** *del* ≡ λ*k i j*. *enum* (*DF k i*) *j* − *enum* (*DF k i*) (*j* − *1*)

define *QF* **where** *QF k* ≡ *wfrec pair-less* (λ*f* (*j*,*i*).
if j=*0 then AF k i*
else let r = (*if i*=*0 then f* (*j*−*1*,*m*−*1*) *else f* (*j*,*i*−*1*)) *in*
enum (*nxt N* (*Suc* (*Max r*))) ‘ {..< *del k* (*if j*=*k then m* − *Suc i else i*) *j*})
for *k*
note *cut-apply* [*simp*]

have *finite-QF* [*simp*]: *finite* (*QF k p*) **for** *p k*
using *wf-pair-less*
proof (*induction p rule*: *wf-induct-rule*)

```
    case (less p)
    then show ?case
      by (simp add: def-wfrec [OF QF-def, of k p] split: prod.split)
  qed

  have del-gt-0: ⟦j < Suc k; 0 < j⟧ ⟹ 0 < del k i j for i j k
    by (simp add: card-DF del-def finite-DF)

  have QF-ne [simp]: QF k (j,i) ≠ {} if j: j < Suc k for j i k
    using wf-pair-less j
  proof (induction (j,i) rule: wf-induct-rule)
    case less
    then show ?case
      by (auto simp: def-wfrec [OF QF-def, of k (j,i)] AF-ne lessThan-empty-iff
del-gt-0)
  qed

  have QF-0 [simp]: QF k (0,i) = AF k i for i k
    by (simp add: def-wfrec [OF QF-def])

  have QF-Suc: QF k (Suc j,0) = enum (nxt N (Suc (Max (QF k (j, m − 1)))))
‘
                    {..< del k (if Suc j = k then m − 1 else 0) (Suc j)} for j k
    apply (simp add: def-wfrec [OF QF-def, of k (Suc j,0)] One-nat-def)
    apply (simp add: pair-less-def cut-def)
    done

  have QF-Suc-Suc: QF k (Suc j, Suc i)
                = enum (nxt N (Suc (Max (QF k (Suc j, i))))) ‘ {..< del k (if Suc
j = k then m − Suc(Suc i) else Suc i) (Suc j)}
    for i j k
    by (simp add: def-wfrec [OF QF-def, of k (Suc j,Suc i)])

  have less-QF1: QF k (j, m − 1) ≪ QF k (Suc j,0) for j k
    by (auto simp: def-wfrec [OF QF-def, of k (Suc j,0)] pair-lessI1 enum-nxt-ge
        intro!: less-sets-weaken2 [OF less-sets-Suc-Max])

  have less-QF2: QF k (j,i) ≪ QF k (j, Suc i) for j i k
    by (auto simp: def-wfrec [OF QF-def, of k (j, Suc i)] pair-lessI2 enum-nxt-ge
          intro: less-sets-weaken2 [OF less-sets-Suc-Max] strict-mono-setsD [OF
sm-AF])

  have less-QF-same: QF k (j,i′) ≪ QF k (j,i)
    if i′ < i j ≤ k for i′ i j k
  proof (rule strict-mono-setsD [OF less-sets-imp-strict-mono-sets])
    show QF k (j, i) ≪ QF k (j, Suc i) for i
      by (simp add: less-QF2)
    show QF k (j, i) ≠ {} if 0 < i for i
      using that by (simp add: ‹j ≤ k› le-imp-less-Suc)
```

qed (*use that* **in** *auto*)

have *less-QF-step*: *QF k* (*j*−*1*, *i′*) ≪ *QF k* (*j*,*i*)
if *0* < *j j* ≤ *k i′* < *m* **for** *j i′ i k*
proof −
have *less-QF1′*: *QF k* (*j* − *1*, *m*−*1*) ≪ *QF k* (*j*,*0*) **if** *j* > *0* **for** *j*
by (*metis less-QF1 that Suc-pred One-nat-def*)
have *QF k* (*j*−*1*, *i′*) ≪ *QF k* (*j*,*0*)
proof (*cases i′* = *m* − *1*)
case *True*
then show *?thesis*
using *less-QF1′* ‹*0* < *j*› **by** *blast*
next
case *False*
show *?thesis*
using *False that less-sets-trans* [*OF less-QF-same less-QF1′ QF-ne*] **by** *auto*
qed
then show *?thesis*
by (*metis QF-ne less-QF-same less-Suc-eq-le less-sets-trans* ‹*j* ≤ *k*› *zero-less-iff-neq-zero*)
qed

have *less-QF*: *QF k* (*j′*,*i′*) ≪ *QF k* (*j*,*i*)
if *j*: *j′* < *j j* ≤ *k* **and** *i*: *i′* < *m i* < *m* **for** *j′ j i′ i k*
using *j*
proof (*induction j*−*j′ arbitrary*: *j*)
case (*Suc d*)
show *?case*
proof (*cases j′* < *j* − *1*)
case *True*
then have *QF k* (*j′*, *i′*) ≪ *QF k* (*j* − *1*, *i*)
using *Suc.hyps Suc.prems*(*2*) **by** *force*
then show *?thesis*
by (*rule less-sets-trans* [*OF* - *less-QF-step QF-ne*]) (*use Suc i* **in** *auto*)
next
case *False*
then have *j′* = *j* − *1*
using ‹*j′* < *j*› **by** *linarith*
then show *?thesis*
using *Suc.hyps* ‹*j* ≤ *k*› *less-QF-step i* **by** *auto*
qed
qed *auto*

have *sm-QF*: *strict-mono-sets* ({..*k*} × {..<*m*}) (*QF k*) **for** *k*
unfolding *strict-mono-sets-def*
proof (*intro strip*)
fix *p q*
assume *p*: *p* ∈ {..*k*} × {..<*m*} **and** *q*: *q* ∈ {..*k*} × {..<*m*} **and** *p* < *q*
then obtain *j′ i′ j i* **where** §: *p* = (*j′*,*i′*) *q* = (*j*,*i*) *i′* < *m i* < *m j′* ≤ *k j* ≤ *k*
using *surj-pair* [*of p*] *surj-pair* [*of q*] **by** *blast*

```
    with ‹p < q› have j' < j ∨ j' = j ∧ i' < i
      by auto
    then show QF k p ≪ QF k q
      using § less-QF less-QF-same by presburger
  qed
  then have sm-QF1: strict-mono-sets {..<ka} (λj. QF k (j,i))
    if i<m Suc k ≥ ka ka ≥ k for ka k i
  proof −
    have {..<ka} ⊆ {..k}
      by (metis lessThan-Suc-atMost lessThan-subset-iff ‹Suc k ≥ ka›)
    then show ?thesis
      by (simp add: less-QF strict-mono-sets-def subset-iff that)
  qed

  have disjoint-QF: i'=i ∧ j'=j if ¬ disjnt (QF k (j', i')) (QF k (j,i)) j' ≤ k j ≤
k i' < m i < m for i' i j' j k
    using that strict-mono-sets-imp-disjoint [OF sm-QF]
    by (force simp: pairwise-def)

  have card-QF: card (QF k (j,i)) = (if j=0 then ∏ (DF k i) else del k (if j = k
then m − Suc i else i) j)
    for i k j
  proof (cases j)
    case 0
    then show ?thesis
      by (simp add: AF-def card-image inj-enum-nxt)
  next
    case (Suc j')
    show ?thesis
        by (cases i; simp add: Suc One-nat-def QF-Suc QF-Suc-Suc card-image
inj-enum-nxt)
  qed
  have AF-non-Nil: list-of (AF k i) ≠ [] for k i
    by (simp add: AF-ne)
  have QF-non-Nil: list-of (QF k (j,i)) ≠ [] if j < Suc k for i j k
    by (simp add: that)

  have AF-subset-N: AF k i ⊆ N for i k
    unfolding AF-def image-subset-iff
    using nxt-subset enumerate-in-set infinite-nxtN ‹infinite N› by blast

  have QF-subset-N: QF k (j,i) ⊆ N for i j k
  proof (induction j)
    case (Suc j)
    show ?case
    by (cases i) (use nxt-subset enumerate-in-set in ‹(force simp: QF-Suc QF-Suc-Suc)+›)
  qed (use AF-subset-N in auto)

  obtain ka k where k>0 and kka: k ≤ ka ka ≤ Suc k l = ((ka+k) − 1)
```

by (*metis One-nat-def assms(2) diff-add-inverse form-cases le0 le-refl*)
then have *ka* > *0*
using *dual-order.strict-trans1* **by** *blast*
have *ka-k-or-Suc*: *ka* = *k* ∨ *ka* = *Suc k*
using *kka* **by** *linarith*
have *lessThan-k*: {..<*k*} = *insert 0* {*0*<..<*k*} **if** *k*>*0* **for** *k*::*nat*
using *that* **by** *auto*
then have *sorted-list-of-set-k*: *list-of* {..<*k*} = *0* # *list-of* {*0*<..<*k*} **if** *k*>*0* **for** *k*::*nat*
using *sorted-list-of-set-insert-remove-cons* [*of concl*: *0* {*0*<..<*k*}] *that* **by** *simp*

define *RF* **where** *RF* ≡ λ*j i*. *if j* = *k then QF k* (*j*, *m* − *Suc i*) *else QF k* (*j*,*i*)
have *RF-subset-N*: *RF j i* ⊆ *N* **if** *i*<*m* **for** *i j*
using *that QF-subset-N* **by** (*simp add*: *RF-def*)
have *finite-RF* [*simp*]: *finite* (*RF k p*) **for** *p k*
by (*simp add*: *RF-def*)
have *RF-0*: *RF 0 i* = *AF k i* **for** *i*
using *RF-def* ‹*0* < *k*› **by** *auto*
have *disjoint-RF*: *i'*=*i* ∧ *j'*=*j* **if** ¬ *disjnt* (*RF j' i'*) (*RF j i*) *j'* ≤ *k j* ≤ *k i'* < *m i* < *m* **for** *i' i j' j*
using *disjoint-QF that*
by (*auto simp*: *RF-def split*: *if-split-asm dest*: *disjoint-QF*)

have *sum-card-RF* [*simp*]: (∑*j*≤*n*. *card* (*RF j i*)) = *enum* (*DF k i*) *n* **if** *n* ≤ *k i* < *m* **for** *i n*
using *that*
proof (*induction n*)
case *0*
then show *?case*
using *DF-ne* [*of k i*] *finite-DF* [*of k i*] ‹*k*>*0*›
by (*simp add*: *RF-def AF-def card-image inj-enum-nxt enum-0-eq-Inf-finite*)
next
case (*Suc n*)
then have *enum* (*DF k i*) *0* ≤ *enum* (*DF k i*) *n* ∧ *enum* (*DF k i*) *n* ≤ *enum* (*DF k i*) (*Suc n*)
using *sm-enum-DF* [*of k i*]
by (*metis card-DF finite-DF finite-enumerate-mono-iff le-imp-less-Suc less-nat-zero-code linorder-not-le nless-le*)
with *Suc* **show** *?case*
by (*auto simp*: *RF-def card-QF del-def*)
qed
have *DF-in-N*: *enum* (*DF k i*) *j* ∈ *N* **if** *j* ≤ *k* **for** *i j*
by (*metis DF-N card-DF finite-DF finite-enumerate-in-set le-imp-less-Suc subsetD that*)
have *Inf-DF-N*: ∏(*DF k p*) ∈ *N* **for** *k p*
using *DF-N DF-ne Inf-nat-def1* **by** *blast*
have *RF-in-N*: (∑*j*≤*n*. *card* (*RF j i*)) ∈ *N* **if** *n* ≤ *k i* < *m* **for** *i n*
by (*auto simp*: *DF-in-N that*)

```
  have ka − 1 ≤ k
    using kka(2) by linarith
  then have sum-card-RF′ [simp]:
    (∑j<ka. card (RF j i)) = enum (DF k i) (ka − 1) if i < m for i
    using sum-card-RF [of ka − 1 i]
    by (metis Suc-diff-1 ‹0 < ka› lessThan-Suc-atMost that)

  have enum-DF-le-iff [simp]:
    enum (DF k i) j ≤ enum (DF k i′) j ⟷ i ≤ i′ (is ?lhs = -)
    if j ≤ k for i′ i j k
  proof
    show i ≤ i′ if ?lhs
    proof −
      have enum (DF k i) j ∈ DF k i
        by (simp add: card-DF finite-enumerate-in-set finite-DF le-imp-less-Suc ‹j
≤ k›)
      moreover have enum (DF k i′) j ∈ DF k i′
      by (simp add: ‹j ≤ k› card-DF finite-enumerate-in-set finite-DF le-imp-less-Suc
that)
      ultimately have enum (DF k i′) j < enum (DF k i) j if i′ < i
       using sm-DF [of k] by (meson UNIV-I less-sets-def strict-mono-setsD that)
      then show ?thesis
       using not-less that by blast
    qed
    show ?lhs if i ≤ i′
    by (metis DF-DF ‹j ≤ k› card-DF finite-DF finite-enumerate-in-set le-eq-less-or-eq

              less-Suc-eq-le less-sets-def that)
  qed
  then have enum-DF-eq-iff[simp]:
    enum (DF k i) j = enum (DF k i′) j ⟷ i = i′ if j ≤ k for i′ i j k
    by (metis le-antisym order-refl that)
  have enum-DF-less-iff [simp]:
    enum (DF k i) j < enum (DF k i′) j ⟷ i < i′ if j ≤ k for i′ i j k
    by (meson enum-DF-le-iff not-less that)

  have card-AF-sum: card (AF k i) + (∑j∈{0<..<ka}. card (RF j i)) = enum
(DF k i) (ka−1)
    if i < m for i
    using that ‹k > 0› ‹k ≤ ka› ‹ka ≤ Suc k›
    by (simp add: lessThan-k RF-0 flip: sum-card-RF′)

  have sorted-list-of-set-iff [simp]: list-of {0<..<k} = [] ⟷ k = 1 if k>0 for
k::nat
    using atLeastSucLessThan-greaterThanLessThan that by fastforce
  show thesis — proof of main result
  proof
    have inj: inj-on (λi. list-of (⋃j<ka. RF j i)) {..<m}
    proof (clarsimp simp: inj-on-def)
```

```
    fix x y
    assume x < m y < m list-of (⋃j<ka. RF j x) = list-of (⋃j<ka. RF j y)
    then have eq: (⋃j<ka. RF j x) = (⋃j<ka. RF j y)
      by (simp add: sorted-list-of-set-inject)
    show x = y
    proof −
      obtain n where n: n ∈ RF 0 x
       using AF-ne QF-0 ‹0 < k› Inf-nat-def1 ‹k ≤ ka› by (force simp: RF-def)
      with eq ‹ka > 0› obtain j' where j' < ka n ∈ RF j' y
        by blast
      then show ?thesis
        using disjoint-QF [of k 0 x j'] n ‹x < m› ‹y < m› ‹ka ≤ Suc k› ‹0 < k›
        by (force simp: RF-def disjnt-iff simp del: QF-0 split: if-split-asm)
    qed
  qed

  define M where M ≡ (λi. list-of (⋃j<ka. RF j i)) ‘ {..<m}
  have finite M
    unfolding M-def by blast
  moreover have card M = m
    by (simp add: M-def ‹k ≤ ka› card-image inj)
  moreover have M ⊆ WW
    by (force simp: M-def WW-def)
  ultimately show M ∈ [WW]^m
    by (simp add: nsets-def)

  have sm-RF: strict-mono-sets {..<ka} (λj. RF j i) if i<m for i
    using sm-QF1 that kka
    by (simp add: less-QF RF-def strict-mono-sets-def)

  have RF-non-Nil: list-of (RF j i) ≠ [] if j < Suc k for i j
    using that by (simp add: RF-def)

  have less-RF-same: RF j i' ≪ RF j i
    if i' < i j < k for i' i j
    using that by (simp add: less-QF-same RF-def)

  have less-RF-same-k: RF k i' ≪ RF k i — reversed version for k
    if i < i' i' < m for i' i
    using that by (simp add: less-QF-same RF-def)

  show Form l U ∧ list.set (inter-scheme l U) ⊆ N if U ∈ [M]^2 for U
  proof −
   from that obtain x y where U = {x,y} x ∈ M y ∈ M and xy: (x,y) ∈ lenlex
less-than
      by (auto simp: lenlex-nsets-2-eq)
    let ?R = λp. list-of ∘ (λj. RF j p)
    obtain p q where x: x = list-of (⋃j<ka. RF j p)
      and y: y = list-of (⋃j<ka. RF j q) and p < m q < m
```

using ‹$x \in M$› ‹$y \in M$› **by** (*auto simp*: *M-def*)
then have *pq*: $p<q$ *length* $x <$ *length* y
using *xy* ‹$k \leq ka$› ‹$ka \leq Suc\ k$› *lexl-not-refl* [*OF irreft-less-than*]
by (*auto simp*: *lenlex-def sm-RF sorted-list-of-set-UN-lessThan length-concat sum-sorted-list-of-set-map*)
have *xc*: $x = concat\ (map\ (?R\ p)\ (list\text{-}of\ \{..<ka\}))$
by (*simp add*: *x sorted-list-of-set-UN-eq-concat* ‹$k \leq ka$› ‹$ka \leq Suc\ k$› ‹$p < m$› *sm-RF*)
have *yc*: $y = concat\ (map\ (?R\ q)\ (list\text{-}of\ \{..<ka\}))$
by (*simp add*: *y sorted-list-of-set-UN-eq-concat* ‹$k \leq ka$› ‹$ka \leq Suc\ k$› ‹$q < m$› *sm-RF*)
have *enum-DF-AF*: $enum\ (DF\ k\ p)\ (ka - 1) < hd\ (list\text{-}of\ (AF\ k\ p))$ **for** p
proof (*rule less-setsD* [*OF DF-AF*])
show $enum\ (DF\ k\ p)\ (ka - 1) \in DF\ k\ p$
using ‹$ka \leq Suc\ k$› *card-DF finite-DF* **by** (*auto simp*: *finite-enumerate-in-set*)
show $hd\ (list\text{-}of\ (AF\ k\ p)) \in AF\ k\ p$
using *AF-non-Nil finite-AF hd-in-set set-sorted-list-of-set* **by** *blast*
qed

have *less-RF-RF*: $RF\ n\ p \ll RF\ n\ q$ **if** $n < k$ **for** n
using *that* ‹$p<q$› **by** (*simp add*: *less-RF-same*)
have *less-RF-Suc*: $RF\ n\ q \ll RF\ (Suc\ n)\ q$ **if** $n < k$ **for** n
using ‹$q < m$› *that* **by** (*auto simp*: *RF-def less-QF*)
have *less-RF-k*: $RF\ k\ q \ll RF\ k\ p$
using ‹$q < m$› *less-RF-same-k* ‹$p<q$› **by** *blast*
have *less-RF-k-ka*: $RF\ (k-1)\ p \ll RF\ (ka - 1)\ q$
using *ka-k-or-Suc less-RF-RF*
by (*metis One-nat-def RF-def* ‹$0 < k$› ‹$ka - 1 \leq k$› ‹$p < m$› *diff-Suc-1 diff-Suc-less less-QF-step*)
have *Inf-DF-eq-enum*: $\prod\ (DF\ k\ i) = enum\ (DF\ k\ i)\ 0$ **for** $k\ i$
by (*simp add*: *Inf-nat-def enumerate-0*)

have *Inf-DF-less*: $\prod\ (DF\ k\ i') < \prod\ (DF\ k\ i)$ **if** $i'<i$ **for** $i'\ i\ k$
by (*metis DF-ne enum-0-eq-Inf enum-0-eq-Inf-finite enum-DF-less-iff le0 that*)
have *AF-Inf-DF-less*: $\bigwedge x.\ x \in AF\ k\ i \Longrightarrow \prod\ (DF\ k\ i') < x$ **if** $i' \leq i$ **for** $i'\ i\ k$
using *less-setsD* [*OF DF-AF*] *DF-ne that*
by (*metis Inf-DF-less Inf-nat-def1 dual-order.order-iff-strict dual-order.strict-trans*)

show *?thesis* — The general case requires $1 < k$, necessitating a painful special case
proof (*cases k=1*)
case *True*
with *kka* **consider** *ka=1* | *ka=2* **by** *linarith*
then show *?thesis*
proof *cases*
case *1*
define *zs* **where** $zs = card\ (AF\ 1\ p) \#\ list\text{-}of\ (AF\ 1\ p)$
$@\ card\ (AF\ 1\ q) \#\ list\text{-}of\ (AF\ 1\ q)$

have *zs*: *Form-Body ka k x y zs*
proof (*intro that exI conjI Form-Body.intros*)
show *x = concat* ([*list-of* (*AF k p*)]) *y = concat* ([*list-of* (*AF k q*)])
by (*simp-all add*: *x y 1 lessThan-Suc RF-0*)
have *AF k p* ≪ *insert* (∏ (*DF k q*)) (*AF k q*)
by (*metis AF-DF DF-ne Inf-nat-def1 RF-0* ‹*0 < k*› *insert-iff less-RF-RF less-sets-def pq*(*1*))
then have *strict-sorted* (*list-of* (*AF k p*) @ ∏ (*DF k q*) # *list-of* (*AF k q*))
by (*auto simp*: *strict-sorted-append-iff intro*: *less-sets-imp-list-less AF-Inf-DF-less*)
moreover have ⋀*x*. *x* ∈ *AF k q* ⟹ ∏ (*DF k p*) < *x*
by (*meson AF-Inf-DF-less less-imp-le-nat* ‹*p < q*›)
moreover have ⋀*x*. *x* ∈ *AF 1 p* ⟹ ∏ (*DF 1 p*) < *x*
by (*meson DF-AF DF-ne Inf-nat-def1 less-setsD*)
ultimately show *strict-sorted zs*
using ‹*p < q*› *True Inf-DF-less DF-AF DF-ne*
by (*auto simp*: *zs-def less-sets-def card-AF AF-Inf-DF-less*)
qed (*auto simp*: ‹*k=1*› ‹*ka=1*› *zs-def AF-ne* ‹*length x < length y*›)
have *zs-N*: *list.set zs* ⊆ *N*
using *AF-subset-N* **by** (*auto simp*: *zs-def card-AF Inf-DF-N* ‹*k=1*›)
show *?thesis*
proof
have *l = 1*
using *kka* ‹*k=1*› ‹*ka=1*› **by** *auto*
have *Form* (*2∗1−1*) {*x*,*y*}
using *1 Form.intros*(*2*) *True zs* **by** *fastforce*
then show *Form l U*
by (*simp add*: ‹*U = {x,y}*› ‹*l = 1*› *One-nat-def*)
show *list.set* (*inter-scheme l U*) ⊆ *N*
using *kka zs zs-N* ‹*k=1*› *Form-Body-imp-inter-scheme* **by** (*fastforce simp*: ‹*U = {x,y}*›)
qed
next
case *2* — Still in our painful special case
note *True* [*simp*] **note** *2* [*simp*]
have [*simp*]: {*0*<..<*2*} = {*1*::*nat*}
by *auto*

have *enum-DF1-eq*: *enum* (*DF 1 i*) *1 = card* (*AF 1 i*) + *card* (*RF 1 i*)
if *i < m* **for** *i*
using *card-AF-sum that* **by** (*simp add*: *One-nat-def*)
have *card-RF*: *card* (*RF 1 i*) = *enum* (*DF 1 i*) *1* − *enum* (*DF 1 i*) *0* **if** *i < m* **for** *i*
using *that* **by** (*auto simp*: *RF-def card-QF del-def*)
have *list-of-AF-RF*: *list-of* (*AF 1 q* ∪ *RF 1 q*) = *list-of* (*AF 1 q*) @ *list-of* (*RF 1 q*)
by (*metis One-nat-def RF-0 True* ‹*0 < k*› *finite-RF less-RF-Suc sorted-list-of-set-Un*)

define *zs* **where** *zs* = *card* (*AF 1 p*) # (*card* (*AF 1 p*) + *card* (*RF 1 p*)) # *list-of* (*AF 1 p*)
@ (*card* (*AF 1 q*) + *card* (*RF 1 q*)) # *list-of* (*AF 1 q*) @ *list-of* (*RF 1 q*) @ *list-of* (*RF 1 p*)
have *zs*: *Form-Body ka k x y zs*
proof (*intro that exI conjI Form-Body.intros*)
have *x* = *list-of* (*RF 0 p* ∪ *RF 1 p*)
by (*simp add*: *x eval-nat-numeral lessThan-Suc RF-0 Un-commute One-nat-def*)
also have ... = *list-of* (*RF 0 p*) @ *list-of* (*RF 1 p*)
using *RF-def True* ‹*p* < *m*› *less-QF-step*
by (*metis QF-0 RF-0 diff-self-eq-0 finite-RF le-refl sorted-list-of-set-Un zero-less-one*)
finally show *x* = *concat* ([*list-of* (*AF 1 p*),*list-of* (*RF 1 p*)])
by (*simp add*: *RF-0*)

have ∗: *i* ∈ *AF 1 q* ∨ *i* ∈ *RF 1 q* ∨ *i* ∈ *RF 1 p* ⟹ *enum* (*DF 1 q*) *1* < *i* **for** *i*
using *True card-DF finite-enumerate-in-set*[*OF finite-DF*]
by (*metis AF-ne DF-AF One-nat-def RF-0 RF-non-Nil finite-RF lessI less-RF-Suc less-RF-k less-setsD*
less-sets-trans sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff)

show *y* = *concat* [*list-of* (*RF 1 q* ∪ *AF 1 q*)]
by (*simp add*: *y eval-nat-numeral lessThan-Suc RF-0 One-nat-def*)
show *zs*: *zs* = *concat* [[*card* (*AF 1 p*), *card* (*AF 1 p*) + *card* (*RF 1 p*)], *list-of* (*AF 1 p*),
[*card* (*AF 1 q*) + *card* (*RF 1 q*)], *list-of* (*RF 1 q* ∪ *AF 1 q*)] @ *interact* [*list-of* (*RF 1 p*)] []
using *list-of-AF-RF* **by** (*simp add*: *zs-def Un-commute*)

show *strict-sorted zs*
proof (*simp add*: ‹*p*<*m*› ‹*q*<*m*› ‹*p*<*q*› *zs-def strict-sorted-append-iff*, *intro conjI strip*)
show *0* < *card* (*RF 1 p*)
using ‹*p*<*m*› **by** (*simp add*: *card-RF card-DF finite-DF*)
show *G1*: *card* (*AF 1 p*) < *card* (*AF 1 q*) + *card* (*RF 1 q*)
by (*simp add*: *Inf-DF-less card-AF* ‹*p*<*q*› *trans-less-add1*)
show *card* (*AF 1 p*) < *x*
if *x* ∈ *AF 1 p* ∪ (*AF 1 q* ∪ (*RF 1 q* ∪ *RF 1 p*)) **for** *x*
using *that* ‹*q* < *m*› ∗
by (*metis* (*no-types*) *order-refl AF-Inf-DF-less Un-iff G1 card-AF order.strict-trans enum-DF1-eq that*)
show *G2*: *card* (*AF 1 p*) + *card* (*RF 1 p*) < *card* (*AF 1 q*) + *card* (*RF 1 q*)
using ‹*p* < *q*› ‹*p* < *m*› ‹*q* < *m*› **by** (*metis enum-DF1-eq enum-DF-less-iff le-refl*)
show *card* (*AF 1 q*) + *card* (*RF 1 q*) < *x*

```
            if x ∈ AF 1 q ∪ (RF 1 q ∪ RF 1 p) for x
            using that ‹q < m› * enum-DF1-eq by force
          then show card (AF 1 p) + card (RF 1 p) < x
            if x ∈ AF 1 p ∪ (AF 1 q ∪ (RF 1 q ∪ RF 1 p)) for x
            using that ‹p < m› finite-enumerate-in-set[OF finite-DF]
        by (metis DF-AF G2 Un-iff card-DF dual-order.strict-trans enum-DF1-eq
lessI less-setsD)
          have list-of (AF 1 p) < list-of {enum (DF 1 q) 1}
          proof (rule less-sets-imp-sorted-list-of-set)
            show AF 1 p ≪ {enum (DF 1 q) 1}
        by (metis AF-DF card-DF empty-subsetI finite-DF finite-enumerate-in-set
insert-subset
                  less-Suc-eq less-sets-weaken2 ‹p<q›)
          qed auto
            then show list-of (AF 1 p) < (card (AF 1 q) + card (RF 1 q)) #
list-of (AF 1 q) @ list-of (RF 1 q) @ list-of (RF 1 p)
            using ‹q < m› by (simp add: less-list-def enum-DF1-eq)
          have list-of (AF 1 q) < list-of (RF 1 q)
              by (metis One-nat-def RF-0 True ‹0 < k› finite-RF less-RF-Suc
less-sets-imp-sorted-list-of-set)
          then show list-of (AF 1 q) < list-of (RF 1 q) @ list-of (RF 1 p)
            using RF-non-Nil by (auto simp: less-list-def)
          show list-of (RF 1 q) < list-of (RF 1 p)
          using True finite-RF less-RF-k less-sets-imp-sorted-list-of-set by metis
        qed
        show [list-of (AF 1 p), list-of (RF 1 p)] ∈ lists (- {[]})
          using RF-non-Nil ‹0 < k› by (auto simp: zs-def AF-ne)
        show [card (AF 1 q) + card (RF 1 q)] = acc-lengths 0 [list-of (RF 1 q
∪ AF 1 q)]
          using list-of-AF-RF by (auto simp: zs-def AF-ne sup-commute)
      qed (auto simp: zs-def AF-ne ‹length x < length y›)
      have zs-N: list.set zs ⊆ N
        using ‹p < m› ‹q < m› DF-in-N  enum-DF1-eq [symmetric]
        by (auto simp: zs-def card-AF AF-subset-N RF-subset-N Inf-DF-N)
      show ?thesis
      proof
        have Form (2*1) {x,y}
          by (metis 2 Form.simps Suc-1 True zero-less-one zs)
        with kka show Form l U
          by (simp add: ‹U = {x,y}›)
        show list.set (inter-scheme l U) ⊆ N
          using kka zs zs-N ‹k=1› Form-Body-imp-inter-scheme by (fastforce
simp: ‹U = {x, y}›)
      qed
    qed
  next
    case False
    then have k ≥ 2 ka ≥ 2
      using kka ‹k>0› by auto
```

then have *k-minus-1* [*simp*]: *Suc* (*k* − *Suc* (*Suc 0*)) = *k* − *Suc 0*
by *auto*
have [*simp*]: *Suc* (*k* − *2*) = *k*−*1*
using ‹*k* ≥ *2*› **by** *linarith*
define *PP* **where** *PP* ≡ *map* (*?R p*) (*list-of* {*0*<..<*ka*})
define *QQ* **where** *QQ* ≡ *map* (*?R q*) (*list-of* {*0*<..<*k*−*1*}) @ ([*list-of* (*RF* (*k*−*1*) *q* ∪ *RF* (*ka*−*1*) *q*)])
let *?INT* = *interact PP QQ*
— No separate sets A and B as in the text, but instead we treat both cases as once
have [*simp*]: *length PP* = *ka* − *1*
by (*simp add*: *PP-def*)
have [*simp*]: *length QQ* = *k*−*1*
using ‹*k* ≥ *2*› **by** (*simp add*: *QQ-def*)

have *PP-n*: *PP* ! *n* = *list-of* (*RF* (*Suc n*) *p*)
if *n* < *ka*−*1* **for** *n*
using *that kka* **by** (*auto simp*: *PP-def nth-sorted-list-of-set-greaterThanLessThan*)

have *QQ-n*: *QQ* ! *n* = (*if n* < *k*−*2 then list-of* (*RF* (*Suc n*) *q*)
else list-of (*RF* (*k*−*1*) *q* ∪ *RF* (*ka* − *1*) *q*))
if *n* < *k*−*1* **for** *n*
using *that kka* **by** (*auto simp*: *QQ-def nth-append nth-sorted-list-of-set-greaterThanLessThan*)

have *QQ-n-same*: *QQ* ! *n* = *list-of* (*RF* (*Suc n*) *q*)
if *n* < *k*−*1 k*=*ka* **for** *n*
using *that kka Suc-diff-Suc*
by (*fastforce simp*: *One-nat-def QQ-def nth-append nth-sorted-list-of-set-greaterThanLessThan*)

have *split-nat-interval*: {*0*<..<*n*} = *insert* (*n*−*1*) {*0*<..<*n*−*1*} **if** *n* ≥ *2* **for** *n*::*nat*
using *that* **by** *auto*
have *split-list-interval*: *list-of*{*0*<..<*n*} = *list-of*{*0*<..<*n*−*1*} @ [*n*−*1*] **if** *n* ≥ *2* **for** *n*::*nat*
proof (*intro sorted-list-of-set-unique* [*THEN iffD1*] *conjI*)
have *list-of* {*0*<..<*n* − *1*} < [*n* − *1*]
by (*auto intro*: *less-sets-imp-list-less*)
then show *strict-sorted* (*list-of* {*0*<..<*n* − *1*} @ [*n* − *1*])
by (*auto simp*: *strict-sorted-append-iff*)
qed (*use* ‹*n* ≥ *2*› **in** *auto*)

have *list-of-RF-Un*: *list-of* (*RF* (*k*−*1*) *q* ∪ *RF k q*) = *list-of* (*RF* (*k*−*1*) *q*) @ *list-of* (*RF k q*)
by (*metis Suc-diff-1* ‹*0* < *k*› *finite-RF lessI less-RF-Suc sorted-list-of-set-Un*)

have *card-AF-sum-QQ*: *card* (*AF k q*) + *sum-list* (*map length QQ*) = (∑ *j*<*ka*. *card* (*RF j q*))
proof (*cases ka* = *Suc k*)
case *True*

```
      have RF (k−1) q ∩ RF k q = {}
        using less-RF-Suc [of k−1] ‹k > 0› by (auto simp: less-sets-def)
       then have card (RF (k−1) q ∪ RF k q) = card (RF (k−1) q) + card
(RF k q)
        by (simp add: card-Un-disjoint)
      then show ?thesis
        using ‹k≥2› ‹q < m›
        apply (simp add: QQ-def True flip: RF-0)
       apply (simp add: lessThan-k split-nat-interval sum-sorted-list-of-set-map)
        done
    next
      case False
      with kka have ka=k by linarith
    with ‹k≥2› show ?thesis by (simp add: QQ-def lessThan-k split-nat-interval
sum-sorted-list-of-set-map flip: RF-0)
    qed

     define LENS where LENS ≡ λi. acc-lengths 0 (list-of (AF k i) # map
(?R i) (list-of {0<..<ka}))
    have LENS-subset-N: list.set (LENS i) ⊆ N if i < m for i
    proof −
      have eq: (list-of (AF k i) # map (?R i) (list-of {0<..<ka})) = map (?R
i) (list-of {..<ka})
        using RF-0 ‹0 < ka› sorted-list-of-set-k by auto
      let ?f = rec-nat [card (AF k i)] (λn r. r @ [(∑j≤Suc n. card (RF j i))])
      have f: acc-lengths 0 (map (?R i) (list-of {..v})) = ?f v for v
     by (induction v) (auto simp: RF-0 acc-lengths-append sum-sorted-list-of-set-map)
      have 3: list.set (?f v) ⊆ N if v ≤ k for v
        using that
      proof (induction v)
        case 0
        have card (AF k i) ∈ N
          by (metis DF-N DF-ne Inf-nat-def1 card-AF subsetD)
        with 0 show ?case by simp
      next
        case (Suc v)
        then have enum (DF k i) (Suc v) ∈ N
            by (metis DF-N card-DF finite-enumerate-in-set finite-DF in-mono
le-imp-less-Suc)
        with Suc ‹i < m› show ?case
          by (simp del: sum.atMost-Suc)
      qed
      show ?thesis
        unfolding LENS-def
       by (metis 3 Suc-pred′ ‹0 < ka› ‹ka − 1 ≤ k› eq f lessThan-Suc-atMost)
    qed
     define LENS-QQ where LENS-QQ ≡ acc-lengths 0 (list-of (AF k q) #
QQ)
    have LENS-QQ-subset: list.set LENS-QQ ⊆ list.set (LENS q)
```

proof (*cases ka = Suc k*)
case *True*
with ‹*k ≥ 2*› **show** *?thesis*
unfolding *QQ-def LENS-QQ-def LENS-def*
by (*auto simp*: *list-of-RF-Un split-list-interval acc-lengths-append*)
next
case *False*
then have *ka=k*
using *kka* **by** *linarith*
with ‹*k ≥ 2*› **show** *?thesis*
by (*simp add*: *QQ-def LENS-QQ-def LENS-def split-list-interval*)
qed
have *ss-INT*: *strict-sorted ?INT*
proof (*rule strict-sorted-interact-I*)
fix *n*
assume *n < length QQ*
then have *n*: *n < k−1*
by *simp*
have *n = k − 2* **if** ¬ *n < k − 2*
using *n that* **by** *linarith*
moreover have *list-of* (*RF* (*Suc* (*k − 2*)) *p*) *< list-of* (*RF* (*k−1*) *q* ∪ *RF* (*ka − 1*) *q*)
by (*auto simp*: *less-sets-imp-sorted-list-of-set less-sets-Un2 less-RF-RF less-RF-k-ka* ‹*0 < k*›)
ultimately show *PP ! n < QQ ! n*
using ‹*k ≤ ka*› *n* **by** (*auto simp*: *PP-n QQ-n less-sets-imp-sorted-list-of-set less-RF-RF*)
next
fix *n*
have *V*: ⟦*Suc n < ka − 1*⟧ ⟹ *list-of* (*RF* (*Suc n*) *q*) *< list-of* (*RF* (*Suc* (*Suc n*)) *p*) **for** *n*
by (*smt RF-def Suc-leI* ‹*ka − 1 ≤ k*› ‹*q < m*› *diff-Suc-1 finite-RF less-QF-step less-le-trans less-sets-imp-sorted-list-of-set nat-neq-iff zero-less-Suc*)
have *RF* (*k − 1*) *q ≪ RF k p*
by (*metis One-nat-def RF-non-Nil Suc-pred* ‹*0 < k*› *finite-RF lessI less-RF-Suc less-RF-k less-sets-trans sorted-list-of-set-eq-Nil-iff*)
with *kka* **have** *RF* (*k−1*) *q* ∪ *RF* (*ka − 1*) *q ≪ RF k p*
by (*metis less-RF-k One-nat-def less-sets-Un1 antisym-conv2 diff-Suc-1 le-less-Suc-eq*)
then have *VI*: *list-of* (*RF* (*k−1*) *q* ∪ *RF* (*ka − 1*) *q*) *< list-of* (*RF k p*)
by (*rule less-sets-imp-sorted-list-of-set*) *auto*
assume *Suc n < length PP*
with ‹*ka ≤ Suc k*› *VI*
show *QQ ! n < PP ! Suc n*
apply (*clarsimp simp*: *PP-n QQ-n V*)
by (*metis One-nat-def Suc-1 Suc-lessI add.right-neutral add-Suc-right diff-Suc-Suc ka-k-or-Suc less-diff-conv*)
next
show *PP ∈ lists* (− {[]})

```
        using RF-non-Nil kka
        by (clarsimp simp: PP-def) (metis RF-non-Nil less-le-trans)
      show QQ ∈ lists (− {[]})
        using RF-non-Nil kka
      by (clarsimp simp: QQ-def) (metis RF-non-Nil Suc-pred ‹0 < k› less-SucI
One-nat-def)
    qed (use kka PP-def QQ-def in auto)
    then have ss-QQ: strict-sorted (concat QQ)
      using strict-sorted-interact-imp-concat by blast

    obtain zs where zs: Form-Body ka k x y zs and zs-N: list.set zs ⊆ N
    proof (intro that exI conjI Form-Body.intros [OF ‹length x < length y›])
      show x = concat (list-of (AF k p) # PP)
        using ‹ka > 0› by (simp add: PP-def RF-0 xc sorted-list-of-set-k)
      let ?YR = (map (list-of ∘ (λj. RF j q)) (list-of {0<..<ka}))
      have concat ?YR = concat QQ
      proof (rule strict-sorted-equal [OF ss-QQ])
        show strict-sorted (concat ?YR)
        proof (rule strict-sorted-concat-I, simp-all)
          fix n
          assume 0: Suc n < ka − Suc 0
          then have Suc n < k
            by (metis One-nat-def ‹ka − 1 ≤ k› less-le-trans)
         then show list-of (RF (list-of {0<..<ka} ! n) q) < list-of (RF (list-of
{0<..<ka} ! Suc n) q)
            by (simp add: Suc-lessD 0 less-RF-Suc less-sets-imp-sorted-list-of-set
nth-sorted-list-of-set-greaterThanLessThan)
        next
          show ?YR ∈ lists (− {[]})
            using RF-non-Nil ‹ka ≤ Suc k› by (auto simp: mem-lists-non-Nil)
        qed auto
        show list.set (concat ?YR) = list.set (concat QQ)
          using ka-k-or-Suc
        proof
          assume ka = k
              then show list.set (concat (map (list-of ∘ (λj. RF j q)) (list-of
{0<..<ka}))) = list.set (concat QQ)
            using ‹k≥2› by simp (simp add: split-nat-interval QQ-def)
        next
          assume ka = Suc k
              then show list.set (concat (map (list-of ∘ (λj. RF j q)) (list-of
{0<..<ka}))) = list.set (concat QQ)
            using ‹k≥2› by simp (auto simp: QQ-def split-nat-interval)
        qed
      qed
      then show y = concat (list-of (AF k q) # QQ)
        using ‹ka > 0› by (simp add: RF-0 yc sorted-list-of-set-k)
     show list-of (AF k p) # PP ∈ lists (− {[]}) list-of (AF k q) # QQ ∈ lists
(− {[]})
```

```
      using RF-non-Nil kka by (auto simp: AF-ne PP-def QQ-def eq-commute
[of []])
    show list.set ((LENS p @ list-of (AF k p) @ LENS-QQ @ list-of (AF k
q) @ ?INT)) ⊆ N
      using AF-subset-N RF-subset-N LENS-subset-N ‹p < m› ‹q < m›
LENS-QQ-subset
      by (auto simp: subset-iff PP-def QQ-def)
    show length (list-of (AF k p) # PP) = ka length (list-of (AF k q) # QQ)
= k
      using ‹0 < ka› ‹0 < k› by auto
     show LENS p = acc-lengths 0 (list-of (AF k p) # PP)
      by (auto simp: LENS-def PP-def)
    show strict-sorted (LENS p @ list-of (AF k p) @ LENS-QQ @ list-of (AF
k q) @ ?INT)
      unfolding strict-sorted-append-iff
     proof (intro conjI ss-INT)
       show LENS p < list-of (AF k p) @ LENS-QQ @ list-of (AF k q) @
?INT
        using AF-non-Nil [of k p] ‹k ≤ ka› ‹ka ≤ Suc k› ‹p < m› card-AF-sum
enum-DF-AF
          by (simp add: enum-DF-AF less-list-def card-AF-sum LENS-def
sum-sorted-list-of-set-map
              del: acc-lengths.simps)
      show strict-sorted (LENS p)
        unfolding LENS-def
        by (rule strict-sorted-acc-lengths) (use RF-non-Nil AF-non-Nil kka in
‹auto simp: in-lists-conv-set›)
      show strict-sorted LENS-QQ
        unfolding LENS-QQ-def QQ-def
        by (rule strict-sorted-acc-lengths) (use RF-non-Nil AF-non-Nil kka in
‹auto simp: in-lists-conv-set›)
      have last-AF-DF: last (list-of (AF k p)) < ∏ (DF k q)
        using AF-DF [OF ‹p < q›, of k] AF-non-Nil [of k p] DF-ne [of k q]
     by (metis Inf-nat-def1 finite-AF last-in-set less-sets-def set-sorted-list-of-set)
      then show list-of (AF k p) < LENS-QQ @ list-of (AF k q) @ ?INT
        by (simp add: less-list-def card-AF LENS-QQ-def)
      show LENS-QQ < list-of (AF k q) @ ?INT
          using AF-non-Nil [of k q] ‹q < m› card-AF-sum enum-DF-AF
card-AF-sum-QQ
       by (auto simp: less-list-def AF-ne hd-append card-AF-sum LENS-QQ-def)
      show list-of (AF k q) < ?INT
      proof −
        have AF k q ≪ RF 1 p
          using ‹0 < k› ‹p < m› ‹q < m› by (simp add: RF-def less-QF flip:
QF-0)
        then have last (list-of (AF k q)) < hd (list-of (RF 1 p))
        proof (rule less-setsD)
          show last (list-of (AF k q)) ∈ AF k q
           using AF-non-Nil finite-AF last-in-set set-sorted-list-of-set by blast
```

```
            show hd (list-of (RF 1 p)) ∈ RF 1 p
                by (metis One-nat-def RF-non-Nil ‹0 < k› finite-RF hd-in-set
not-less-eq set-sorted-list-of-set)
          qed
          with ‹k > 0› ‹ka ≥ 2› RF-non-Nil show ?thesis
         by (simp add: One-nat-def hd-interact less-list-def sorted-list-of-set-greaterThanLessThan
PP-def QQ-def)
          qed
        qed auto
      qed (auto simp: LENS-QQ-def)
      show ?thesis
      proof (cases ka = k)
        case True
        then have l = 2*k-1
          by (simp add: kka(3) mult-2)
        then show ?thesis
         by (metis One-nat-def Form.intros(2) Form-Body-imp-inter-scheme True
‹0 < k› ‹U = {x, y}› kka zs zs-N)
      next
        case False
        then have l = 2*k
          using kka by linarith
        then show ?thesis
         by (metis One-nat-def False Form.intros(3) Form-Body-imp-inter-scheme
‹0 < k› ‹U = {x, y}› antisym kka le-SucE zs zs-N)
      qed
    qed
  qed
 qed
qed
```

3.11 Larson's Lemma 3.8

3.11.1 Primitives needed for the inductive construction of b

```
definition IJ where IJ ≡ λk. Sigma {..k} (λj::nat. {..<j})

lemma IJ-iff: u ∈ IJ k ⟷ (∃j i. u = (j,i) ∧ i<j ∧ j≤k)
  by (auto simp: IJ-def)

lemma finite-IJ: finite (IJ k)
  by (auto simp: IJ-def)

fun prev where
  prev 0 0 = None
| prev (Suc 0) 0 = None
| prev (Suc j) 0 = Some (j, j - Suc 0)
| prev j (Suc i) = Some (j,i)

lemma prev-eq-None-iff: prev j i = None ⟷ j ≤ Suc 0 ∧ i = 0
```

by (*auto simp*: *le-Suc-eq elim*: *prev.elims*)

lemma *prev-pair-less*:
prev j i = *Some ji'* ⟹ (*ji'*, (*j*,*i*)) ∈ *pair-less*
by (*auto simp*: *pair-lessI1 elim*: *prev.elims*)

lemma *prev-Some-less*: ⟦*prev j i* = *Some* (*j'*,*i'*); *i* ≤ *j*⟧ ⟹ *i'* < *j'*
by (*auto elim*: *prev.elims*)

lemma *prev-maximal*:
⟦*prev j i* = *Some* (*j'*,*i'*); (*ji''*, (*j*,*i*)) ∈ *pair-less*; *ji''* ∈ *IJ k*⟧
⟹ (*ji''*, (*j'*,*i'*)) ∈ *pair-less* ∨ *ji''* = (*j'*,*i'*)
by (*force simp*: *IJ-def pair-less-def elim*: *prev.elims*)

lemma *pair-less-prev*:
assumes (*u*, (*j*,*i*)) ∈ *pair-less u* ∈ *IJ k*
shows *prev j i* = *Some u* ∨ (∃ *x*. (*u*, *x*) ∈ *pair-less* ∧ *prev j i* = *Some x*)
using *assms*
proof (*cases prev j i*)
case *None*
then show *?thesis*
using *assms* **by** (*force simp*: *prev-eq-None-iff pair-less-def IJ-def split*: *prod.split*)
next
case (*Some a*)
then show *?thesis*
by (*metis assms prev-maximal prod.exhaust-sel*)
qed

3.11.2 Special primitives for the ordertype proof

definition *USigma* :: *'a set set* ⇒ (*'a set* ⇒ *'a set*) ⇒ *'a set set*
where *USigma* 𝒜 *B* ≡ ⋃ *X*∈𝒜. ⋃ *y* ∈ *B X*. {*insert y X*}

definition *usplit*
where *usplit f A* ≡ *f* (*A* − {*Max A*}) (*Max A*)

lemma *USigma-empty* [*simp*]: *USigma* {} *B* = {}
by (*auto simp*: *USigma-def*)

lemma *USigma-iff*:
assumes ⋀*I j*. *I* ∈ ℐ ⟹ *I* ≪ *J I* ∧ *finite I*
shows *x* ∈ *USigma* ℐ *J* ⟷ *usplit* (λ*I j*. *I* ∈ ℐ ∧ *j* ∈ *J I* ∧ *x* = *insert j I*) *x*
proof −
have [*simp*]: ⋀*I j*. ⟦*I* ∈ ℐ; *j* ∈ *J I*⟧ ⟹ *Max* (*insert j I*) = *j*
by (*meson Max-insert2 assms less-imp-le less-sets-def*)
show *?thesis*
proof −
have §: *j* ∉ *I* **if** *I* ∈ ℐ *j* ∈ *J I* **for** *I j*
using *that* **by** (*metis assms less-irrefl less-sets-def*)

have $\exists I{\in}\mathcal{I}.\ \exists j{\in}J\ I.\ x = \mathit{insert}\ j\ I$
if $x - \{\mathit{Max}\ x\} \in \mathcal{I}$ **and** $\mathit{Max}\ x \in J\ (x - \{\mathit{Max}\ x\})\ x \neq \{\}$
using *that* **by** (*metis Max-in assms infinite-remove insert-Diff*)
then show *?thesis*
by (*auto simp*: *USigma-def usplit-def* §)
qed
qed

proposition *ordertype-append-image-IJ*:
assumes *lenB* [*simp*]: $\bigwedge i\ j.\ i \in \mathcal{I} \Longrightarrow j \in J\ i \Longrightarrow \mathit{length}\ (B\ j) = c$
and *AB*: $\bigwedge i\ j.\ i \in \mathcal{I} \Longrightarrow j \in J\ i \Longrightarrow A\ i < B\ j$
and *IJ*: $\bigwedge i.\ i \in \mathcal{I} \Longrightarrow i \ll J\ i \wedge \mathit{finite}\ i$
and β: $\bigwedge i.\ i \in \mathcal{I} \Longrightarrow \mathit{ordertype}\ (B\ `\ J\ i)\ (\mathit{lenlex\ less\text{-}than}) = \beta$
and *A*: *inj-on A* $\mathcal{I}$
shows *ordertype* (*usplit* ($\lambda i\ j.\ A\ i\ @\ B\ j$) ` *USigma* $\mathcal{I}$ *J*) (*lenlex less-than*)
$= \beta * \mathit{ordertype}\ (A\ `\ \mathcal{I})\ (\mathit{lenlex\ less\text{-}than})$
(**is** *ordertype ?AB ?R* = - * *?*α)
proof (*cases* $\mathcal{I} = \{\}$)
case *False*
have *Ord* β
using β *False wf-Ord-ordertype* **by** *fastforce*
show *?thesis*
proof (*subst ordertype-eq-iff*)
define *split* **where** *split* ≡ λ*l*::*nat list*. (*take* (*length l* − *c*) *l*, (*drop* (*length l* − *c*) *l*))
have *oB*: *ordermap* (*B* ` *J i*) *?R* (*B j*) ⊏ β **if** ‹$i \in \mathcal{I}$› ‹$j \in J\ i$› **for** *i j*
using β *less-TC-iff that* **by** *fastforce*
then show *Ord* (β * *?*α)
by (*intro* ‹*Ord* β› *wf-Ord-ordertype Ord-mult*; *simp*)
define *f* **where** *f* ≡ λ*u*. *let* (*x*,*y*) = *split u in let i* = *inv-into* $\mathcal{I}$ *A x in*
β * *ordermap* (*A*`$\mathcal{I}$) *?R x* + *ordermap* (*B*`*J i*) *?R y*
have *inv-into-IA* [*simp*]: *inv-into* $\mathcal{I}$ *A* (*A i*) = *i* **if** $i \in \mathcal{I}$ **for** *i*
by (*simp add*: *A that*)
show ∃*f*. *bij-betw f ?AB* (*elts* (β * *?*α)) ∧ (∀ *x*∈*?AB*. ∀ *y*∈*?AB*. (*f x* < *f y*) = ((*x*, *y*) ∈ *?R*))
unfolding *bij-betw-def*
proof (*intro exI conjI strip*)
show *inj-on f ?AB*
proof (*clarsimp simp*: *f-def inj-on-def split-def USigma-iff IJ usplit-def*)
fix *x y*
assume §: β * *ordermap* (*A* ` $\mathcal{I}$) *?R* (*A* (*x* − {*Max x*})) + *ordermap* (*B* ` *J* (*x* − {*Max x*})) *?R* (*B* (*Max x*))
= β * *ordermap* (*A* ` $\mathcal{I}$) *?R* (*A* (*y* − {*Max y*})) + *ordermap* (*B* ` *J* (*y* − {*Max y*})) *?R* (*B* (*Max y*))
and *x*: *x* − {*Max x*} ∈ $\mathcal{I}$
and *y*: *y* − {*Max y*} ∈ $\mathcal{I}$
and *mx*: *Max x* ∈ *J* (*x* − {*Max x*})
and *x* = *insert* (*Max x*) *x*

and *my*: *Max y* ∈ *J* (*y* − {*Max y*})
have *ordermap* (*A*‘*I*) *?R* (*A* (*x* − {*Max x*})) = *ordermap* (*A*‘*I*) *?R* (*A* (*y* − {*Max y*}))
and *B-eq*: *ordermap* (*B* ‘ *J* (*x* − {*Max x*})) *?R* (*B* (*Max x*)) = *ordermap* (*B* ‘ *J* (*y* − {*Max y*})) *?R* (*B* (*Max y*))
using *mult-cancellation-lemma* [*OF* §] *oB mx my x y* **by** *blast+*
then have *A* (*x* − {*Max x*}) = *A* (*y* − {*Max y*})
using *x y* **by** *auto*
then have *x* − {*Max x*} = *y* − {*Max y*}
by (*metis x y inv-into-IA*)
then show *A* (*x* − {*Max x*}) = *A* (*y* − {*Max y*}) ∧ *B* (*Max x*) = *B* (*Max y*)
using *B-eq mx my* **by** *auto*
qed
show *f* ‘ *?AB* = *elts* (*β* ∗ *?α*)
proof
show *f* ‘ *?AB* ⊆ *elts* (*β* ∗ *?α*)
using ‹*Ord β*›
apply (*clarsimp simp add*: *f-def split-def USigma-iff IJ usplit-def*)
by (*metis TC-small β add-mult-less image-eqI ordermap-in-ordertype trans-llt wf-Ord-ordertype wf-llt*)
show *elts* (*β* ∗ *?α*) ⊆ *f* ‘ *?AB*
proof (*clarsimp simp*: *f-def split-def image-iff USigma-iff IJ usplit-def Bex-def elim*!: *elts-multE split*: *prod.split*)
fix *γ δ*
assume *δ*: *δ* ∈ *elts β* **and** *γ*: *γ* ∈ *elts ?α*
have *γ* ∈ *ordermap* (*A* ‘ *I*) (*lenlex less-than*) ‘ *A* ‘ *I*
by (*meson γ ordermap-surj subset-iff*)
then obtain *i* **where** *i* ∈ *I* **and** *yv*: *γ* = *ordermap* (*A*‘*I*) *?R* (*A i*)
by *blast*
have *δ* ∈ *ordermap* (*B* ‘ *J i*) (*lenlex less-than*) ‘ *B* ‘ *J i*
by (*metis* (*no-types*) *β δ* ‹*i* ∈ *I*› *in-mono ordermap-surj*)
then obtain *j* **where** *j* ∈ *J i* **and** *xu*: *δ* = *ordermap* (*B*‘*J i*) *?R* (*B j*)
by *blast*
then have *mji*: *Max* (*insert j i*) = *j*
by (*meson IJ Max-insert2* ‹*i* ∈ *I*› *less-imp-le less-sets-def*)
have [*simp*]: *i* − {*j*} = *i*
using *IJ* ‹*i* ∈ *I*› ‹*j* ∈ *J i*› *less-setsD* **by** *fastforce*
show ∃ *l*. (∃ *K*. *K* − {*Max K*} ∈ *I* ∧ *Max K* ∈ *J* (*K* − {*Max K*}) ∧ *K* = *insert* (*Max K*) *K* ∧
l = *A* (*K* − {*Max K*}) @ *B* (*Max K*)) ∧ *β* ∗ *γ* + *δ* =
β ∗
ordermap (*A* ‘ *I*) *?R* (*take* (*length l* − *c*) *l*) +
ordermap (*B* ‘ *J* (*inv-into I A* (*take* (*length l* − *c*) *l*)))
?R (*drop* (*length l* − *c*) *l*)
proof (*intro conjI exI*)
let *?ji* = *insert j i*
show *A i* @ *B j* = *A* (*?ji* − {*Max ?ji*}) @ *B* (*Max ?ji*)
by (*auto simp*: *mji*)

```
      qed (use ‹i ∈ 𝓘› ‹j ∈ J i› mji xu yv in auto)
    qed
  qed
next
  fix p q
  assume p ∈ ?AB and q ∈ ?AB
  then obtain x y where peq: p = A (x − {Max x}) @ B (Max x)
                    and qeq: q = A (y − {Max y}) @ B (Max y)
                    and x: x − {Max x} ∈ 𝓘
                    and y: y − {Max y} ∈ 𝓘
                    and mx: Max x ∈ J (x − {Max x})
                    and my: Max y ∈ J (y − {Max y})
    by (auto simp: USigma-iff IJ usplit-def)
  let ?mx = x − {Max x}
  let ?my = y − {Max y}
  show (f p < f q) ⟷ ((p, q) ∈ ?R)
  proof
    assume f p < f q
    then
   consider ordermap (A‘𝓘) ?R (A (x − {Max x})) < ordermap (A‘𝓘) ?R (A
(y − {Max y}))
      | ordermap (A‘𝓘) ?R (A (x − {Max x})) = ordermap (A‘𝓘) ?R (A (y −
{Max y}))
       ordermap (B‘J (x − {Max x})) ?R (B (Max x)) < ordermap (B‘J (y −
{Max y})) ?R (B (Max y))
      using x y mx my
      by (auto dest: mult-cancellation-less simp: f-def split-def peq qeq oB)
    then have (A ?mx @ B (Max x), A ?my @ B (Max y)) ∈ ?R
    proof cases
      case 1
      then have (A ?mx, A ?my) ∈ ?R
       using x y by (force simp: Ord-mem-iff-lt intro: converse-ordermap-mono)
      then show ?thesis
        using x y mx my lenB lenlex-append1 by blast
    next
      case 2
      then have A ?mx = A ?my
        using ‹?my ∈ 𝓘› ‹?mx ∈ 𝓘› by auto
      then have eq: ?mx = ?my
        by (metis ‹?my ∈ 𝓘› ‹?mx ∈ 𝓘› inv-into-IA)
      then have (B (Max x), B (Max y)) ∈ ?R
     using mx my 2 by (force simp: Ord-mem-iff-lt intro: converse-ordermap-mono)
      with 2 show ?thesis
        by (simp add: eq irefl-less-than)
    qed
    then show (p,q) ∈ ?R
      by (simp add: peq qeq f-def split-def sorted-list-of-set-Un AB)
  next
    assume pqR: (p,q) ∈ ?R
```

```
      then have §: (A ?mx @ B (Max x), A ?my @ B (Max y)) ∈ ?R
        using peq qeq by blast
      then consider (A ?mx, A ?my) ∈ ?R | A ?mx = A ?my ∧ (B (Max x), B
(Max y)) ∈ ?R
      proof (cases (A ?mx, A ?my) ∈ ?R)
        case False
        have False if (A ?my, A ?mx) ∈ ?R
          by (metis ‹?my ∈ ℐ› ‹?mx ∈ ℐ› § ‹(Max y) ∈ J ?my› ‹(Max x) ∈ J
?mx› lenB lenlex-append1 omega-sum-1-less order.asym that)
        then have A ?mx = A ?my
          by (meson False UNIV-I total-llt total-on-def)
        then show ?thesis
          using § irrefl-less-than that by auto
      qed (use that in blast)
      then have β * ordermap (A‘ℐ) ?R (A ?mx) + ordermap (B‘J ?mx) ?R (B
(Max x))
              < β * ordermap (A‘ℐ) ?R (A ?my) + ordermap (B‘J ?my) ?R (B
(Max y))
      proof cases
        case 1
        show ?thesis
        proof (rule add-mult-less-add-mult)
          show ordermap (A‘ℐ) (lenlex less-than) (A ?mx) < ordermap (A‘ℐ)
(lenlex less-than) (A ?my)
            by (simp add: 1 ‹?my ∈ ℐ› ‹?mx ∈ ℐ› ordermap-mono-less)
          show Ord (ordertype (A‘ℐ) ?R)
            using wf-Ord-ordertype by blast
          show ordermap (B ‘ J ?mx) ?R (B (Max x)) ∈ elts β
            using Ord-less-TC-mem ‹Ord β› ‹?mx ∈ ℐ› ‹(Max x) ∈ J ?mx› oB
by blast
          show ordermap (B ‘ J ?my) ?R (B (Max y)) ∈ elts β
            using Ord-less-TC-mem ‹Ord β› ‹?my ∈ ℐ› ‹(Max y) ∈ J ?my› oB
by blast
        qed (use ‹?my ∈ ℐ› ‹?mx ∈ ℐ› ‹Ord β› in auto)
      next
        case 2
        with ‹?mx ∈ ℐ› show ?thesis
          using ‹(Max y) ∈ J ?my› ‹(Max x) ∈ J ?mx› ordermap-mono-less
          by (metis (no-types, opaque-lifting) Kirby.add-less-cancel-left TC-small
image-iff inv-into-IA trans-llt wf-llt y)
      qed
      then show f p < f q
        using ‹?my ∈ ℐ› ‹?mx ∈ ℐ› ‹(Max y) ∈ J ?my› ‹(Max x) ∈ J ?mx›
        by (auto simp: peq qeq f-def split-def AB)
    qed
  qed
 qed auto
qed auto
```

3.11.3 The final part of 3.8, where two sequences are merged

inductive *merge* :: [*nat list list*,*nat list list*,*nat list list*,*nat list list*] ⇒ *bool*
  where *NullNull*: *merge* [] [] [] []
    | *Null*: *as* ≠ [] ⟹ *merge as* [] [*concat as*] []
    | *App*: ⟦*as1* ≠ []; *bs1* ≠ [];
          concat as1 < *concat bs1*; *concat bs1* < *concat as2*; *merge as2 bs2 as bs*⟧
        ⟹ *merge* (*as1*@*as2*) (*bs1*@*bs2*) (*concat as1* # *as*) (*concat bs1* # *bs*)

inductive-simps *Null1* [*simp*]: *merge* [] *bs us vs*
inductive-simps *Null2* [*simp*]: *merge as* [] *us vs*

lemma *merge-single*:
  ⟦*concat as* < *concat bs*; *concat as* ≠ []; *concat bs* ≠ []⟧ ⟹ *merge as bs* [*concat as*] [*concat bs*]
  using *merge.App* [*of as bs* [] []]
  by (*fastforce simp*: *less-list-def*)

lemma *merge-length1-nonempty*:
  assumes *merge as bs us vs as* ∈ *lists* (− {[]})
  shows *us* ∈ *lists* (− {[]})
  using *assms* **by** *induction* (*auto simp*: *mem-lists-non-Nil*)

lemma *merge-length2-nonempty*:
  assumes *merge as bs us vs bs* ∈ *lists* (− {[]})
  shows *vs* ∈ *lists* (− {[]})
  using *assms* **by** *induction* (*auto simp*: *mem-lists-non-Nil*)

lemma *merge-length1-gt-0*:
  assumes *merge as bs us vs as* ≠ []
  shows *length us* > *0*
  using *assms* **by** *induction auto*

lemma *merge-length-le*:
  assumes *merge as bs us vs*
  shows *length vs* ≤ *length us*
  using *assms* **by** *induction auto*

lemma *merge-length-le-Suc*:
  assumes *merge as bs us vs*
  shows *length us* ≤ *Suc* (*length vs*)
  using *assms* **by** *induction auto*

lemma *merge-length-less2*:
  assumes *merge as bs us vs*
  shows *length vs* ≤ *length as*
  using *assms*
proof *induction*
case (*App as1 bs1 as2 bs2 as bs*)

then show *?case*
using *length-greater-0-conv* [*of as1*] **by** (*simp*, *presburger*)
qed *auto*

lemma *merge-preserves*:
assumes *merge as bs us vs*
shows *concat as* = *concat us* ∧ *concat bs* = *concat vs*
using *assms* **by** *induction auto*

lemma *merge-interact*:
assumes *merge as bs us vs strict-sorted* (*concat as*) *strict-sorted* (*concat bs*)
bs ∈ *lists* (− {[]})
shows *strict-sorted* (*interact us vs*)
using *assms*
proof *induction*
case (*App as1 bs1 as2 bs2 as bs*)
then have *bs*: *concat bs1* < *concat bs concat bs1* < *concat as*
and *nonmt*: *concat bs1* ≠ []
using *merge-preserves strict-sorted-append-iff* **by** *fastforce*+
then have *concat bs1* < *interact as bs*
unfolding *less-list-def* **using** *App bs*
by (*metis* (*no-types*, *lifting*) *Un-iff concat-append hd-in-set last-in-set merge-preserves set-interact sorted-wrt-append strict-sorted-append-iff*)
with *App* **show** *?case*
by (*metis append-in-lists-conv concat-append hd-append2 interact.simps*(*3*) *less-list-def strict-sorted-append-iff nonmt*)
qed *auto*

lemma *acc-lengths-merge1*:
assumes *merge as bs us vs*
shows *list.set* (*acc-lengths k us*) ⊆ *list.set* (*acc-lengths k as*)
using *assms*
proof (*induction arbitrary*: *k*)
case (*App as1 bs1 as2 bs2 as bs*)
then show *?case*
apply (*simp add*: *acc-lengths-append strict-sorted-append-iff length-concat-acc-lengths*)
by (*simp add*: *le-supI2 length-concat*)
qed (*auto simp*: *length-concat-acc-lengths*)

lemma *acc-lengths-merge2*:
assumes *merge as bs us vs*
shows *list.set* (*acc-lengths k vs*) ⊆ *list.set* (*acc-lengths k bs*)
using *assms*
proof (*induction arbitrary*: *k*)
case (*App as1 bs1 as2 bs2 as bs*)
then show *?case*
apply (*simp add*: *acc-lengths-append strict-sorted-append-iff length-concat-acc-lengths*)
by (*simp add*: *le-supI2 length-concat*)
qed (*auto simp*: *length-concat-acc-lengths*)

lemma *length-hd-le-concat*:
assumes $as \neq []$ **shows** *length* (*hd as*) ≤ *length* (*concat as*)
by (*metis* (*no-types*) *add.commute assms concat.simps*(*2*) *le-add2 length-append list.exhaust-sel*)

lemma *length-hd-merge2*:
assumes *merge as bs us vs*
shows *length* (*hd bs*) ≤ *length* (*hd vs*)
using *assms* **by** *induction* (*auto simp*: *length-hd-le-concat*)

lemma *merge-less-sets-hd*:
assumes *merge as bs us vs strict-sorted* (*concat as*) *strict-sorted* (*concat bs*) *bs* ∈ *lists* (− {[]})
shows *list.set* (*hd us*) ≪ *list.set* (*concat vs*)
using *assms*
proof *induction*
case (*App as1 bs1 as2 bs2 as bs*)
then have §: *list.set* (*concat bs1*) ≪ *list.set* (*concat bs2*)
by (*force simp*: *dest*: *strict-sorted-imp-less-sets*)
have ∗: *list.set* (*concat as1*) ≪ *list.set* (*concat bs1*)
using *App* **by** (*metis concat-append strict-sorted-append-iff strict-sorted-imp-less-sets*)
then have *list.set* (*concat as1*) ≪ *list.set* (*concat bs*)
using *App* § *less-sets-trans merge-preserves*
by (*metis List.set-empty append-in-lists-conv le-zero-eq length-0-conv length-concat-ge*)
with ∗ *App.hyps* **show** *?case*
by (*fastforce simp*: *less-sets-UN1 less-sets-UN2 less-sets-Un2*)
qed *auto*

lemma *set-takeWhile*:
assumes *strict-sorted* (*concat as*) *as* ∈ *lists* (− {[]})
shows *list.set* (*takeWhile* (λ*x*. *x* < *y*) *as*) = {*x* ∈ *list.set as*. *x* < *y*}
using *assms*
proof (*induction as*)
case (*Cons a as*)
have *a* < *y*
if *a*: *a* < *concat as strict-sorted a strict-sorted* (*concat as*) *x* < *y x* ≠ [] *x* ∈ *list.set as*
for *x*
proof −
have §: *last x* ∈ *list.set* (*concat as*)
using *set-concat that* **by** *fastforce*
have *last a* < *hd* (*concat as*)
using *Cons.prems that* **by** (*auto simp*: *less-list-def*)
also have ... ≤ *hd y* **if** *y* ≠ []
using *that a*
by (*meson* § *order.strict-trans less-list-def not-le sorted-hd-le strict-sorted-imp-sorted*)
finally show *?thesis*

```
    by (simp add: less-list-def)
  qed
  then show ?case
    using Cons by (auto simp: strict-sorted-append-iff)
qed auto

proposition merge-exists:
  assumes strict-sorted (concat as) strict-sorted (concat bs)
          as ∈ lists (− {[]}) bs ∈ lists (− {[]})
          hd as < hd bs as ≠ [] bs ≠ []
  and disj: ⋀a b. ⟦a ∈ list.set as; b ∈ list.set bs⟧ ⟹ a<b ∨ b<a
shows ∃ us vs. merge as bs us vs
  using assms
proof (induction length as + length bs arbitrary: as bs rule: less-induct)
  case (less as bs)
  obtain as1 as2 bs1 bs2
    where A: as1 ≠ [] bs1 ≠ [] concat as1 < concat bs1 concat bs1 < concat as2
      and B: as = as1@as2 bs = bs1@bs2 and C: bs2 = [] ∨ (as2 ≠ [] ∧ hd as2
< hd bs2)
  proof
    define as1 where as1 ≡ takeWhile (λx. x < hd bs) as
    define as2 where as2 ≡ dropWhile (λx. x < hd bs) as
    define bs1 where bs1 ≡ if as2=[] then bs else takeWhile (λx. x < hd as2) bs
    define bs2 where bs2 ≡ if as2=[] then [] else dropWhile (λx. x < hd as2) bs

    have as1: as1 = takeWhile (λx. last x < hd (hd bs)) as
      using less.prems by (auto simp: as1-def less-list-def cong: takeWhile-cong)
    have as2: as2 = dropWhile (λx. last x < hd (hd bs)) as
      using less.prems by (auto simp: as2-def less-list-def cong: dropWhile-cong)

    have hd-as2: as2 ≠ [] ⟹ ¬ hd as2 < hd bs
      using as2-def hd-dropWhile by metis
    have hd-bs2: bs2 ≠ [] ⟹ ¬ hd bs2 < hd as2
      using bs2-def hd-dropWhile by metis
    show as1 ≠ []
      by (simp add: as1-def less.prems takeWhile-eq-Nil-iff)
    show bs1 ≠ []
     by (metis as2 bs1-def hd-as2 hd-in-set less.prems(7) less.prems(8) set-dropWhileD
takeWhile-eq-Nil-iff)
    show bs2 = [] ∨ (as2 ≠ [] ∧ hd as2 < hd bs2)
      by (metis as2-def bs2-def hd-bs2 less.prems(8) list.set-sel(1) set-dropWhileD)
    have AB: list.set A ≪ list.set B
      if A ∈ list.set as1 B ∈ list.set bs for A B
    proof −
      have A ∈ list.set as
        using that by (metis as1 set-takeWhileD)
      then have sorted A
          by (metis concat.simps(2) concat-append less.prems(1) sorted-append
split-list-last strict-sorted-imp-sorted)
```

moreover have *sorted* (*hd bs*)
by (*metis concat.simps*(*2*) *hd-Cons-tl less.prems*(*2*) *less.prems*(*7*) *strict-sorted-append-iff strict-sorted-imp-sorted*)
ultimately show *?thesis*
using *that less.prems*
apply (*clarsimp simp add*: *as1-def set-takeWhile less-list-iff-less-sets less-sets-def*)
by (*metis* (*full-types*) *UN-I hd-concat less-le-trans list.set-sel*(*1*) *set-concat sorted-hd-le strict-sorted-imp-sorted*)
qed
show *as* = *as1*@*as2*
by (*simp add*: *as1-def as2-def*)
show *bs* = *bs1*@*bs2*
by (*simp add*: *bs1-def bs2-def*)
have *list.set* (*concat as1*) ≪ *list.set* (*concat bs1*)
using *AB set-takeWhileD* **by** (*fastforce simp*: *as1-def bs1-def less-sets-UN1 less-sets-UN2*)
then show *concat as1* < *concat bs1*
by (*rule less-sets-imp-list-less*)
have *list.set* (*concat bs1*) ≪ *list.set* (*concat as2*) **if** *as2* ≠ []
proof (*clarsimp simp add*: *bs1-def less-sets-UN1 less-sets-UN2 set-takeWhile less.prems*)
fix *A B*
assume *A* ∈ *list.set as2 B* ∈ *list.set bs B* < *hd as2*
with *that* **show** *list.set B* ≪ *list.set A*
using *hd-as2 less.prems*(*1*,*2*)
apply (*clarsimp simp add*: *less-sets-def less-list-def*)
apply (*auto simp*: *as2-def*)
apply (*simp flip*: *as2-def*)
by (*smt* (*verit*, *ccfv-SIG*) *UN-I* ‹*as* = *as1* @ *as2*› *concat.simps*(*2*) *concat-append hd-concat in-set-conv-decomp-first le-less-trans less-le-trans set-concat sorted-append sorted-hd-le sorted-le-last strict-sorted-imp-sorted that*)
qed
then show *concat bs1* < *concat as2*
by (*simp add*: *bs1-def less-sets-imp-list-less*)
qed
obtain *cs ds* **where** *merge as2 bs2 cs ds*
proof (*cases as2* = [] ∨ *bs2* = [])
case *True*
then show *thesis*
using *that C NullNull Null* **by** *metis*
next
have †: *length as2* + *length bs2* < *length as* + *length bs*
by (*simp add*: *A B*)
case *False*
moreover have *strict-sorted* (*concat as2*) *strict-sorted* (*concat bs2*)
as2 ∈ *lists* (− {[]}) *bs2* ∈ *lists* (− {[]})
⋀*a b*. ⟦*a* ∈ *list.set as2*; *b* ∈ *list.set bs2*⟧ ⟹ *a* < *b* ∨ *b* < *a*
using *B less.prems strict-sorted-append-iff* **by** *auto*
ultimately show *?thesis*

using *C less.hyps* [*OF* †] *False that* **by** *force*
qed
then obtain *cs* **where** *merge* (*as1* @ *as2*) (*bs1* @ *bs2*) (*concat as1* # *cs*) (*concat bs1* # *ds*)
using *A merge.App* **by** *blast*
then show *?case*
using *B* **by** *blast*
qed

3.11.4 Actual proof of Larson's Lemma 3.8

proposition *lemma-3-8*:
assumes *infinite N*
obtains *X* **where** $X \subseteq$ *WW ordertype X* (*lenlex less-than*) $= \omega\uparrow\omega$
$\bigwedge u.\ u \in [X]^2 \Longrightarrow$
$\exists\, l.$ *Form l u* $\wedge$ ($l > 0 \longrightarrow$ [*enum N l*] $<$ *inter-scheme l u* $\wedge$ *List.set* (*inter-scheme l u*) $\subseteq$ *N*)
proof −
let *?LL* = *lenlex less-than*
define *bf* **where** *bf* $\equiv \lambda M\ q.$ *wfrec pair-less* (λf (*j*,*i*).
let R = (*case prev j i of None* ⇒ *M* | *Some u* ⇒ *snd* (*f u*))
in grab R (*q j i*))

have *bf-rec*: *bf M q* (*j*,*i*) =
(*let R* = (*case prev j i of None* ⇒ *M* | *Some u* ⇒ *snd* (*bf M q u*))
in grab R (*q j i*)) **for** *M q j i*
by (*subst* (*1*) *bf-def*) (*simp add*: *Let-def wfrec bf-def cut-apply prev-pair-less cong*: *conj-cong split*: *option.split*)

have *infinite* (*snd* (*bf M q u*)) = *infinite M* $\wedge$ *fst* (*bf M q u*) $\subseteq M \wedge$ *snd* (*bf M q u*) $\subseteq M$ **for** *M q u*
using *wf-pair-less*
proof (*induction u rule*: *wf-induct-rule*)
case (*less u*)
then show *?case*
proof (*cases u*)
case (*Pair j i*)
with *less.IH prev-pair-less* **show** *?thesis*
apply (*simp add*: *bf-rec* [*of M q j i*] *split*: *option.split*)
using *fst-grab-subset snd-grab-subset* **by** *blast*
qed
qed
then have *infinite-bf* [*simp*]: *infinite* (*snd* (*bf M q u*)) = *infinite M*
and *bf-subset*: *fst* (*bf M q u*) $\subseteq M \wedge$ *snd* (*bf M q u*) $\subseteq M$ **for** *M q u*
by *auto*

have *bf-less-sets*: *fst* (*bf M q ij*) $\ll$ *snd* (*bf M q ij*) **if** *infinite M* **for** *M q ij*
using *wf-pair-less*

```
  proof (induction ij rule: wf-induct-rule)
    case (less u)
    then show ?case
      by (metis bf-rec finite-grab-iff infinite-bf less-sets-grab prod.exhaust-sel that)
  qed

  have card-fst-bf: finite (fst (bf M q (j,i))) ∧ card (fst (bf M q (j,i))) = q j i if
infinite M for M q j i
    by (simp add: that bf-rec [of M q j i] split: option.split)

  have bf-cong: bf M q u = bf M q' u
    if snd u ≤ fst u and eq: ⋀y x. ⟦x≤y; y≤fst u⟧ ⟹ q' y x = q y x for M q q' u
    using wf-pair-less that
  proof (induction u rule: wf-induct-rule)
    case (less u)
    show ?case
    proof (cases u)
      case (Pair j i)
      with less.prems show ?thesis
      proof (clarsimp simp add: bf-rec [of M - j i] split: option.split)
        fix j' i'
        assume *: prev j i = Some (j',i')
        then have **: ((j', i'), u) ∈ pair-less
          by (simp add: Pair prev-pair-less)
        moreover have i' < j'
          using Pair less.prems by (simp add: prev-Some-less [OF *])
        moreover have ⋀x y. ⟦x ≤ y; y ≤ j'⟧ ⟹ q' y x = q y x
          using ** less.prems by (auto simp: pair-less-def Pair)
        ultimately show grab (snd (bf M q (j',i'))) (q j i) = grab (snd (bf M q'
(j',i'))) (q j i)
          using less.IH by auto
      qed
    qed
  qed

  define ediff where ediff ≡ λD:: nat ⇒ nat set. λj i. enum (D j) (Suc i) − enum
(D j) i
  define F where F ≡ λl (dl,a0::nat set,b0::nat × nat ⇒ nat set,M).
        let (d,Md) = grab (nxt M (enum N (Suc (2 * Suc l)))) (Suc l) in
        let (a,Ma) = grab Md (Min d) in
        let Gb = bf Ma (ediff (dl(l := d))) in
        let dl' = dl(l := d) in
        (dl', a, fst ∘ Gb, snd (Gb(l, l−1)))
  define DF where DF ≡ rec-nat (λi∈{..<0}. {}, {}, λp. {}, N) F
  have DF-simps: DF 0 = (λi∈{..<0}. {}, {}, λp. {}, N)
              DF (Suc l) = F l (DF l) for l
    by (auto simp: DF-def)
  note cut-apply [simp]
```

have *inf* [*rule-format*]: $\forall$ *dl al bl L. DF l* = (*dl*,*al*,*bl*,*L*) $\longrightarrow$ *infinite L* **for** *l*
by (*induction l*) (*auto simp*: *DF-simps F-def Let-def grab-eqD infinite-nxtN assms split*: *prod.split*)

define Ψ **where**
$\Psi \equiv \lambda(dl,\ a,\ b,\ M).\ \lambda l{::}nat.$
dl l $\ll$ *a* $\wedge$ *card a* > *0* $\wedge$
($\forall j{\leq}l.$ *card* (*dl j*) = *Suc j*) $\wedge$ *a* $\ll \bigcup$(*range b*) $\wedge$ *range b* $\subseteq$ *Collect finite* $\wedge$
a $\subseteq$ *N* $\wedge \bigcup$(*range b*) $\subseteq$ *N* $\wedge$ *infinite M* $\wedge$ *b*(*l*,*l*−*1*) $\ll$ *M* $\wedge$ *M* $\subseteq$ *N*
have Ψ-*DF*: Ψ (*DF* (*Suc l*)) *l* **for** *l*
proof (*induction l*)
case *0*
show *?case*
using *assms*
apply (*clarsimp simp add*: *bf-rec F-def DF-simps* Ψ-*def split*: *prod.split*)
apply (*drule grab-eqD*, *blast dest*: *grab-eqD infinite-nxtN*)+
apply (*auto simp*: *less-sets-UN2 less-sets-grab card-fst-bf elim*!: *less-sets-weaken2*)
apply (*metis card-1-singleton-iff Min-singleton greaterThan-iff insertI1 le0 nxt-subset-greaterThan subsetD*)
using *nxt-subset snd-grab-subset bf-subset* **by** *blast*+
next
case (*Suc l*)
then show *?case*
using *assms*
unfolding *Let-def DF-simps*(*2*)[*of Suc l*] *F-def* Ψ-*def*
apply (*clarsimp simp add*: *bf-rec DF-simps split*: *prod.split*)
apply (*drule grab-eqD*, *metis grab-eqD infinite-nxtN*)+
apply (*safe*, *simp-all add*: *less-sets-UN2 less-sets-grab card-fst-bf card-Suc-eq-finite*)
apply (*meson less-sets-weaken2*)
apply (*metis Min-in gr0I greaterThan-iff insert-not-empty le-inf-iff less-asym nxt-def subsetD*)
apply (*meson bf-subset less-sets-weaken2*)
apply (*meson nxt-subset subset-eq*)
apply (*meson bf-subset nxt-subset subset-eq*)
using *bf-rec infinite-bf* **apply** *force*
using *bf-less-sets bf-rec* **apply** *force*
by (*metis bf-rec bf-subset nxt-subset subsetD*)
qed

define *d* **where** *d* $\equiv \lambda k.$ *let* (*dk*,*ak*,*bk*,*M*) = *DF*(*Suc k*) *in dk k*
define *a* **where** *a* $\equiv \lambda k.$ *let* (*dk*,*ak*,*bk*,*M*) = *DF*(*Suc k*) *in ak*
define *b* **where** *b* $\equiv \lambda k.$ *let* (*dk*,*ak*,*bk*,*M*) = *DF*(*Suc k*) *in bk*
define *M* **where** *M* $\equiv \lambda k.$ *let* (*dk*,*ak*,*bk*,*M*) = *DF k in M*

have *infinite-M* [*simp*]: *infinite* (*M k*) **for** *k*
by (*auto simp*: *M-def inf split*: *prod.split*)

have *M-Suc-subset*: *M* (*Suc k*) $\subseteq$ *M k* **for** *k*

apply (*clarsimp simp add*: *Let-def M-def F-def DF-simps split*: *prod.split*)
by (*metis bf-subset in-mono nxt-subset snd-conv snd-grab-subset*)

have *Inf-M-Suc-ge*: *Inf* (*M k*) ≤ *Inf* (*M* (*Suc k*)) **for** *k*
by (*simp add*: *M-Suc-subset cInf-superset-mono infinite-imp-nonempty*)

have *Inf-M-telescoping*: {*Inf* (*M k*)..} ⊆ {*Inf* (*M k'*)..} **if** *k'*: *k'*≤*k* **for** *k k'*
using *that Inf-nat-def1 infinite-M* **unfolding** *Inf-nat-def atLeast-subset-iff*
by (*metis M-Suc-subset finite.emptyI le-less-linear lift-Suc-antimono-le not-less-Least subsetD*)

have *d-eq*: *d k* = *fst* (*grab* (*nxt* (*M k*) (*enum N* (*Suc* (2 ∗ *Suc k*)))) (*Suc k*)) **for** *k*
by (*simp add*: *d-def M-def Let-def DF-simps F-def split*: *prod.split*)
then have *finite-d* [*simp*]: *finite* (*d k*) **for** *k*
by *simp*
then have *d-ne* [*simp*]: *d k* ≠ {} **for** *k*
by (*metis card.empty card-grab d-eq infinite-M infinite-nxtN nat.distinct*(*1*))
have *a-eq*: ∃ *M*. *a k* = *fst* (*grab M* (*Min* (*d k*))) ∧ *infinite M* **for** *k*
apply (*simp add*: *a-def d-def M-def Let-def DF-simps F-def split*: *prod.split*)
by (*metis fst-conv grab-eqD infinite-nxtN local.inf*)
then have *card-a*: *card* (*a k*) = *Inf* (*d k*) **for** *k*
by (*metis cInf-eq-Min card-grab d-ne finite-d*)

have *d-eq-dl*: *d k* = *dl k* **if** (*dl*,*a*,*b*,*P*) = *DF l k* < *l* **for** *k l dl a b P*
using *that*
by (*induction l arbitrary*: *dl a b P*) (*simp-all add*: *d-def DF-simps F-def Let-def split*: *prod.split-asm prod.split*)

have *card-d* [*simp*]: *card* (*d k*) = *Suc k* **for** *k*
by (*auto simp*: *d-eq infinite-nxtN*)

have *d-ne* [*simp*]: *d j* ≠ {} **and** *a-ne* [*simp*]: *a j* ≠ {}
and *finite-d* [*simp*]: *finite* (*d j*) **and** *finite-a* [*simp*]: *finite* (*a j*) **for** *j*
using Ψ*-DF* [*of j*] **by** (*auto simp*: Ψ*-def a-def d-def card-gt-0-iff split*: *prod.split-asm*)

have *da*: *d k* ≪ *a k* **for** *k*
using Ψ*-DF* [*of k*] **by** (*simp add*: Ψ*-def a-def d-def split*: *prod.split-asm*)

have *ab-same*: *a k* ≪ ⋃(*range*(*b k*)) **for** *k*
using Ψ*-DF* [*of k*] **by** (*simp add*: Ψ*-def a-def b-def M-def split*: *prod.split-asm*)

have *snd-bf-subset*: *snd* (*bf M r* (*j*,*i*)) ⊆ *snd* (*bf M r* (*j'*,*i'*))
if *ji*: ((*j'*,*i'*), (*j*,*i*)) ∈ *pair-less* (*j'*,*i'*) ∈ *IJ k*
for *M r k j i j' i'*
using *wf-pair-less ji*
proof (*induction rule*: *wf-induct-rule* [**where** *a*= (*j*,*i*)])
case (*less u*)
show *?case*

```
    proof (cases u)
      case (Pair j i)
     then consider prev j i = Some (j', i') | x where ((j', i'), x) ∈ pair-less prev
j i = Some x
        using less.prems pair-less-prev by blast
      then show ?thesis
      proof cases
        case 2 with less.IH show ?thesis
          unfolding bf-rec Pair
          by (metis in-mono option.simps(5) prev-pair-less snd-grab-subset subsetI
that(2))
      qed (simp add: Pair bf-rec snd-grab-subset)
    qed
  qed

  have less-bf: fst (bf M r (j',i')) ≪ fst (bf M r (j,i))
    if ji: ((j',i'), (j,i)) ∈ pair-less (j',i') ∈ IJ k and infinite M
    for M r k j i j' i'
  proof −
    consider prev j i = Some (j', i') | j'' i'' where ((j', i'), (j'',i'')) ∈ pair-less
prev j i = Some (j'',i'')
      by (metis pair-less-prev ji prod.exhaust-sel)
    then show ?thesis
    proof cases
      case 1
      then show ?thesis
        using bf-less-sets bf-rec less-sets-fst-grab ‹infinite M› by force
    next
      case 2
      then have fst (bf M r (j',i')) ≪ snd (bf M r (j'',i''))
        by (meson bf-less-sets snd-bf-subset less-sets-weaken2 that)
      with 2 show ?thesis
        using bf-rec bf-subset less-sets-fst-grab ‹infinite M› by auto
    qed
  qed

  have aM: a k ≪ M (Suc k) for k
   apply (clarsimp simp add: a-def M-def DF-simps F-def Let-def split: prod.split)
    by (meson bf-subset grab-eqD infinite-nxtN less-sets-weaken2 local.inf)
  then have a k ≪ a (Suc k) for k
   by (metis IntE card-d card.empty d-eq da fst-grab-subset less-sets-trans less-sets-weaken2
nat.distinct(1) nxt-def subsetI)
  then have aa: a j ≪ a k if j<k for k j
   by (meson UNIV-I a-ne less-sets-imp-strict-mono-sets strict-mono-sets-def that)
  then have ab: a k' ≪ b k (j,i) if k'≤k for k k' j i
    by (metis a-ne ab-same le-less less-sets-UN2 less-sets-trans rangeI that)
  have db: d j ≪ b k (j,i) if j≤k for k j i
    by (meson a-ne ab da less-sets-trans that)
```

have *bMkk*: *b k* (*k*,*k*−*1*) ≪ *M* (*Suc k*) **for** *k*
using Ψ-*DF* [*of k*]
by (*simp add*: Ψ-*def b-def d-def M-def split*: *prod.split-asm*)

have *b*: ∃ *P* ⊆ *M k*. *infinite P* ∧ (∀ *j i*. *i*≤*j* ⟶ *j*≤*k* ⟶ *b k* (*j*,*i*) = *fst* (*bf P* (*ediff d*) (*j*,*i*))) **for** *k*
proof (*clarsimp simp*: *b-def DF-simps F-def Let-def split*: *prod.split*)
fix *a a′ d′ dl bb P M′ M″*
assume *gr*: *grab M″* (*Min d′*) = (*a′*, *M′*) *grab* (*nxt P* (*enum N* (*Suc* (*Suc* (*Suc* (*2* ∗ *k*)))))) (*Suc k*) = (*d′*, *M″*)
and *DF*: *DF k* = (*dl*, *a*, *bb*, *P*)
have *deq*: *d j* = (*if j* = *k then d′ else dl j*) **if** *j*≤*k* **for** *j*
proof (*cases j* < *k*)
case *True*
then show *?thesis* **by** (*metis DF d-eq-dl less-not-refl*)
next
case *False*
then show *?thesis*
using *that DF gr* **by** (*auto simp*: *d-def DF-simps F-def Let-def split*: *prod.split*)
qed
have *M′* ⊆ *P*
by (*metis gr in-mono nxt-subset snd-conv snd-grab-subset subsetI*)
also have *P* ⊆ *M k*
using *DF* **by** (*simp add*: *M-def*)
finally have *M′* ⊆ *M k* **.**
moreover have *infinite M′*
using *DF* **by** (*metis* (*mono-tags*) *finite-grab-iff gr infinite-nxtN local.inf snd-conv*)
moreover
have *ediff* (*dl*(*k* := *d′*)) *j i* = *ediff d j i* **if** *j*≤*k* **for** *j i*
by (*simp add*: *deq that ediff-def*)
then have *bf M′* (*ediff* (*dl*(*k* := *d′*))) (*j*,*i*)
= *bf M′* (*ediff d*) (*j*,*i*) **if** *i* ≤ *j j*≤*k* **for** *j i*
using *bf-cong that* **by** *fastforce*
ultimately show ∃ *P*⊆*M k*. *infinite P* ∧
(∀ *j i*. *i* ≤ *j* ⟶ *j* ≤ *k*
⟶ *fst* (*bf M′* (*ediff* (*dl*(*k* := *d′*))) (*j*,*i*))
= *fst* (*bf P* (*ediff d*) (*j*,*i*)))
by *auto*
qed

have *card-b*: *card* (*b k* (*j*,*i*)) = *enum* (*d j*) (*Suc i*) − *enum* (*d j*) *i* **if** *j*≤*k* **for** *k j i*
— there's a short proof of this from the previous result but it would need *i* ≤ *j*
proof (*clarsimp simp*: *b-def DF-simps F-def Let-def split*: *prod.split*)
fix *dl*
and *a a′ d′*:: *nat set*
and *bb M M′ M″*
assume *gr*: *grab M″* (*Min d′*) = (*a′*, *M′*) *grab* (*nxt M* (*enum N* (*Suc* (*Suc* (*Suc*

```
(2 * k))))))) (Suc k) = (d',M'')
      and DF: DF k = (dl, a, bb, M)
    have d j = (if j = k then d' else dl j)
    proof (cases j < k)
      case True
      then show ?thesis by (metis DF d-eq-dl less-not-refl)
    next
      case False
      then show ?thesis
      using that DF gr by (auto simp: d-def DF-simps F-def Let-def split: prod.split)
    qed
    then show card (fst (bf M' (ediff (dl(k := d')) (j,i))))
          = enum (d j) (Suc i) - enum (d j) i
      using DF gr card-fst-bf grab-eqD infinite-nxtN local.inf ediff-def by auto
  qed

  have card-b-pos: card (b k (j,i)) > 0 if i < j j≤k for k j i
    by (simp add: card-b that finite-enumerate-step)
  have b-ne [simp]: b k (j,i) ≠ {} if i < j j≤k for k j i
    using card-b-pos [OF that] less-imp-neq by fastforce+

  have card-b-finite [simp]: finite (b k u) for k u
    using Ψ-DF [of k] by (fastforce simp: Ψ-def b-def)

  have bM: b k (j,i) ≪ M (Suc k) if i<j j≤k for i j k
  proof -
    obtain M' where M' ⊆ M k infinite M'
      and bk: ⋀j i. i≤j ⟹ j≤k ⟹ b k (j,i) = fst (bf M' (ediff d) (j,i))
      using b by (metis (no-types, lifting))
    show ?thesis
    proof (cases j=k ∧ i = k-1)
      case False
      show ?thesis
      proof (rule less-sets-trans [OF - bMkk])
        show b k (j,i) ≪ b k (k, k-1)
          using that ‹infinite M'› False
            by (force simp: bk pair-less-def IJ-def intro: less-bf)
        show b k (k, k-1) ≠ {}
          using b-ne that by auto
      qed
    qed (use bMkk in auto)
  qed

  have b-InfM: ⋃ (range (b k)) ⊆ {∏ (M k)..} for k
   proof (clarsimp simp add: Ψ-def b-def M-def DF-simps F-def Let-def split:
prod.split)
    fix r dl :: nat ⇒ nat set
      and a b and d' a' M'' M' P and x j' i' :: nat
    assume gr: grab M'' (Min d') = (a', M')
```

```
        grab (nxt P (enum N (Suc (Suc (Suc (2 * k)))))) (Suc k) = (d', M'')
    and DF: DF k = (dl, a, b, P)
    and x: x ∈ fst (bf M' (ediff (dl(k := d'))) (j', i'))
  have infinite P
    using DF local.inf by blast
  then have M' ⊆ P
    by (meson gr grab-eqD infinite-nxtN nxt-subset order.trans)
  with bf-subset show ⋂ P ≤ x
    using Inf-nat-def x le-less-linear not-less-Least by fastforce
qed

have b-Inf-M-Suc: b k (j,i) ≪ {Inf(M (Suc k))} if i<j j≤k for k j i
  using bMkk [of k] that
 by (metis Inf-nat-def1 bM finite.emptyI infinite-M less-setsD less-sets-singleton2)

have bb-same: b k (j',i') ≪ b k (j,i)
  if ((j',i'), (j,i)) ∈ pair-less (j',i') ∈ IJ k for k j i j' i'
  using that
  unfolding b-def DF-simps F-def Let-def
  by (auto simp: less-bf grab-eqD infinite-nxtN local.inf split: prod.split)

have bb: b k' (j',i') ≪ b k (j,i)
  if j: i' < j' j'≤k' and k: k'<k for i i' j j' k' k
proof (rule atLeast-less-sets)
  show b k' (j', i') ≪ {Inf(M (Suc k'))}
    using Suc-lessD b-Inf-M-Suc nat-less-le j by blast
  show b k (j,i) ⊆ {Inf(M (Suc k'))..}
    by (meson Inf-M-telescoping Suc-leI UnionI b-InfM rangeI subset-eq k)
qed

have M-subset-N: M k ⊆ N for k
proof (cases k)
  case (Suc k')
  with Ψ-DF [of k'] show ?thesis
    by (auto simp: M-def Let-def Ψ-def split: prod.split)
qed (auto simp: M-def DF-simps)
have a-subset-N: a k ⊆ N for k
  using Ψ-DF [of k] by (simp add: a-def Ψ-def split: prod.split prod.split-asm)
have d-subset-N: d k ⊆ N for k
  using M-subset-N [of k] d-eq fst-grab-subset nxt-subset by blast
have b-subset-N: b k (j,i) ⊆ N for k j i
  using Ψ-DF [of k] by (force simp: b-def Ψ-def)

define K:: [nat,nat] ⇒ nat set set
  where K ≡ λj0 j. nsets {j0<..} j
have K-finite: finite K and K-card: card K = j if K ∈ K j0 j for K j0 j
  using that by (auto simp add: K-def nsets-def)
have K-enum: j0 < enum K i if K ∈ K j0 j i < card K for K j0 j i
  using that by (auto simp: K-def nsets-def finite-enumerate-in-set subset-eq)
```

have $\mathcal{K}$-*0* [*simp*]: $\mathcal{K}\ k\ 0 = \{\{\}\}$ **for** k
by (*auto simp*: $\mathcal{K}$-*def*)

have $\mathcal{K}$-*Suc*: $\mathcal{K}\ j0\ (Suc\ j) = USigma\ (\mathcal{K}\ j0\ j)\ (\lambda K.\ \{Max\ (insert\ j0\ K)<..\})$ (**is** *?lhs = ?rhs*)
for *j j0*
proof
show $\mathcal{K}\ j0\ (Suc\ j) \subseteq USigma\ (\mathcal{K}\ j0\ j)\ (\lambda K.\ \{Max\ (insert\ j0\ K)<..\})$
unfolding $\mathcal{K}$-*def nsets-def USigma-def*
proof *clarsimp*
fix *K*
assume *K*: $K \subseteq \{j0<..\}$ *finite K card K = Suc j*
then have $Max\ K \in K$
by (*metis Max-in card-0-eq nat.distinct*(*1*))
then obtain *i* **where** $Max\ (insert\ j0\ (K - \{Max\ K\})) < i\ K = insert\ i\ (K - \{Max\ K\})$
using *K*
by (*simp add*: *subset-iff*) (*metis DiffE Max.coboundedI insertCI insert-Diff le-neq-implies-less*)
then show $\exists L{\subseteq}\{j0<..\}.\ finite\ L \wedge card\ L = j \wedge (\exists i{\in}\{Max\ (insert\ j0\ L)<..\}.\ K = insert\ i\ L)$
using *K*
by (*metis* ‹*Max K ∈ K*› *card-Diff-singleton-if diff-Suc-1 finite-Diff greaterThan-iff insert-subset*)
qed
show *?rhs* $\subseteq \mathcal{K}\ j0\ (Suc\ j)$
by (*force simp*: $\mathcal{K}$-*def nsets-def USigma-def*)
qed

define *BB* **where** $BB \equiv \lambda j0\ j\ K.\ list\text{-}of\ (a\ j0 \cup (\bigcup i{<}j.\ b\ (enum\ K\ i)\ (j0{,}i)))$
define *XX* **where** $XX \equiv \lambda j.\ BB\ j\ j\ `\ \mathcal{K}\ j\ j$

have *less-list-of*: $BB\ j\ i\ K < list\text{-}of\ (b\ l\ (j{,}i))$
if *K*: $K \in \mathcal{K}\ j\ i\ \forall j{\in}K.\ j < l$ **and** $i \le j\ j \le l$ **for** *j i K l*
unfolding *BB-def*
proof (*rule less-sets-imp-sorted-list-of-set*)
have $\bigwedge i.\ i < card\ K \Longrightarrow b\ (enum\ K\ i)\ (j{,}i) \ll b\ l\ (j,\ card\ K)$
using *that* **by** (*metis* $\mathcal{K}$-*card* $\mathcal{K}$-*enum* $\mathcal{K}$-*finite bb finite-enumerate-in-set nat-less-le less-le-trans*)
then show $a\ j \cup (\bigcup i{<}i.\ b\ (enum\ K\ i)\ (j{,}i)) \ll b\ l\ (j{,}i)$
using *that* **unfolding** $\mathcal{K}$-*def nsets-def*
by (*auto simp*: *less-sets-Un1 less-sets-UN1 ab finite-enumerate-in-set subset-eq*)
qed *auto*
have *BB-Suc*: $BB\ j0\ (Suc\ j)\ K = usplit\ (\lambda L\ k.\ BB\ j0\ j\ L\ @\ list\text{-}of\ (b\ k\ (j0,\ j)))\ K$
if *j*: $j \le j0$ **and** *K*: $K \in \mathcal{K}\ j0\ (Suc\ j)$ **for** *j0 j K*
— towards the ordertype proof
proof −
have *Kj*: $K \subseteq \{j0<..\}$ **and** [*simp*]: *finite K* **and** *cardK*: *card K = Suc j*

```
      using K by (auto simp: 𝒦-def nsets-def)
    have KMK: K − {Max K} ∈ 𝒦 j0 j
      using that by (simp add: 𝒦-Suc USigma-iff 𝒦-finite less-sets-def usplit-def)
    have j0 < Max K
     by (metis Kj Max-in cardK card-gt-0-iff greaterThan-iff subsetD zero-less-Suc)
    have MaxK: Max K = enum K j
    proof (rule Max-eqI)
      fix k
      assume k ∈ K
      with K cardK show k ≤ enum K j
        by (metis ‹finite K› finite-enumerate-Ex finite-enumerate-mono-iff leI lessI
not-less-eq)
    qed (auto simp: cardK finite-enumerate-in-set)
    have ene: i<j ⟹ enum (K − {enum K j}) i = enum K i for i
      using finite-enumerate-Diff-singleton [OF ‹finite K›] by (simp add: cardK)
    have BB j0 (Suc j) K = list-of ((a j0 ∪ (⋃x<j. b (enum K x) (j0, x))) ∪ b
(enum K j) (j0, j))
      by (simp add: BB-def lessThan-Suc Un-ac)
    also have ... = list-of ((a j0 ∪ (⋃i<j. b (enum K i) (j0, i)))) @ list-of (b
(enum K j) (j0, j))
    proof (rule sorted-list-of-set-Un)
      have b (enum K i) (j0, i) ≪ b (enum K j) (j0, j) if i<j for i
        using K 𝒦-enum bb cardK j le-eq-less-or-eq that by auto
      moreover have a j0 ≪ b (enum K j) (j0, j)
        using MaxK ‹j0 < Max K› ab by auto
     ultimately show a j0 ∪ (⋃x<j. b (enum K x) (j0, x)) ≪ b (enum K j) (j0,
j)
        by (simp add: less-sets-Un1 less-sets-UN1)
    qed (auto simp: finite-UnI)
    also have ... = BB j0 j (K − {Max K}) @ list-of (b (Max K) (j0, j))
      by (simp add: BB-def MaxK ene)
    also have ... = usplit (λL k. BB j0 j L @ list-of (b k (j0, j))) K
      by (simp add: usplit-def)
    finally show ?thesis .
  qed

  have enum-d-0: enum (d j) 0 = Inf (d j) for j
    using enum-0-eq-Inf-finite by auto

  have Inf-b-less: ⨅(b k' (j',i')) < ⨅(b k (j,i))
    if j: i' < j' i < j j'≤k' j≤k and k: k'<k for i i' j j' k' k
    using bb [of i' j' k' k j i] that b-ne [of i' j' k'] b-ne [of i j k]
    by (simp add: less-sets-def Inf-nat-def1)

  have b-ge-k: ⨅ (b k (k, k−1)) ≥ k−1 for k
  proof (induction k)
    case (Suc k)
    show ?case
    proof (cases k=0)
```

```
    case False
    then have ∏ (b k (k, k − 1)) < ∏ (b (Suc k) (Suc k, k))
      using Inf-b-less by auto
    with Suc show ?thesis
      by simp
  qed auto
qed auto

have b-ge: ∏ (b k (j,i)) ≥ k−1 if k ≥ j j > i for k j i
  by (metis Inf-b-less Suc-leI b-ge-k diff-Suc-1 lessI not-less that diff-le-mono)
have hd-b: hd (list-of (b k (j,i))) = ∏ (b k (j,i))
  if i < j j ≤ k for k j i
  using that by (simp add: hd-list-of cInf-eq-Min)

have b-disjoint-less: b (enum K i) (j0, i) ∩ b (enum K i') (j0, i') = {}
  if K: K ⊆ {j0<..} finite K card K ≥ j0 i' < j i < i' j ≤ j0 for i i' j j0 K
proof (intro bb less-sets-imp-disjnt [unfolded disjnt-def])
  show i < j0
    using that by linarith
  then show j0 ≤ enum K i
   by (meson K finite-enumerate-in-set greaterThan-iff less-imp-le-nat less-le-trans
subsetD)
  show enum K i < enum K i'
    using K ‹j ≤ j0› that by auto
qed

have b-disjoint: b (enum K i) (j0, i) ∩ b (enum K i') (j0, i') = {}
  if K: K ⊆ {j0<..} finite K card K ≥ j0 i < j i' < j i ≠ i' j ≤ j0 for i i' j j0 K
  using that b-disjoint-less inf-commute neq-iff by metis

have otω: ordertype ((λk. list-of (b k (j,i))) ‘ {Max (insert j K)<..}) ?LL = ω
          (is ?lhs = -)
  if K: K ∈ 𝒦 j i j > i for j i K
proof −
  have Sucj: Suc (Max (insert j K)) ≥ j
    using 𝒦-finite that(1) le-Suc-eq by auto
  let ?N = {Inf(b k (j,i))| k. Max (insert j K) < k}
  have infN: infinite ?N
  proof (clarsimp simp add: infinite-nat-iff-unbounded-le)
    fix m
    show ∃ n≥m. ∃ k. n = ∏ (b k (j,i)) ∧ Max (insert j K) < k
      using b-ge ‹j > i› Sucj
    by (metis (no-types, lifting) diff-Suc-1 le-SucI le-trans less-Suc-eq-le nat-le-linear)
  qed
  have [simp]: Max (insert j K) < k ⟷ j < k ∧ (∀ a∈K. a < k) for k
    using that by (auto simp: 𝒦-finite)
  have ?lhs = ordertype ?N less-than
  proof (intro ordertype-eqI strip)
    have list-of (b k (j,i)) = list-of (b k' (j,i))
```

if $j \leq k$ $j \leq k'$ *hd* (*list-of* (*b k* (*j*,*i*))) = *hd* (*list-of* (*b k'* (*j*,*i*)))
for *k k'*
by (*metis Inf-b-less* ‹*i* < *j*› *hd-b nat-less-le not-le that*)
moreover have ∃ *k' j' i'*. *hd* (*list-of* (*b k* (*j*,*i*))) = ∏ (*b k'* (*j'*, *i'*)) ∧ *i'* < *j'* ∧ *j'* ≤ *k'*
if *j* ≤ *k* **for** *k*
using *that* ‹*i* < *j*› *hd-b less-imp-le-nat* **by** *blast*
moreover have ∃ *k'*. *hd* (*list-of* (*b k* (*j*,*i*))) = ∏ (*b k'* (*j*,*i*)) ∧ *j* < *k'* ∧ (∀ *a*∈*K*. *a* < *k'*)
if *j* < *k* ∀ *a*∈*K*. *a* < *k* **for** *k*
using *that K hd-b less-imp-le-nat* **by** *blast*
moreover have ∏ (*b k* (*j*,*i*)) ∈ *hd* ‘ (λ*k*. *list-of* (*b k* (*j*,*i*))) ‘ {*Max* (*insert j K*)<..}
if *j* < *k* ∀ *a*∈*K*. *a* < *k* **for** *k*
using *that K* **by** (*auto simp*: *hd-b image-iff*)
ultimately
show *bij-betw hd* ((λ*k*. *list-of* (*b k* (*j*,*i*))) ‘ {*Max* (*insert j K*)<..}) {∏ (*b k* (*j*,*i*)) |*k*. *Max* (*insert j K*) < *k*}
by (*auto simp*: *bij-betw-def inj-on-def*)
next
fix *ms ns*
assume *ms* ∈ (λ*k*. *list-of* (*b k* (*j*,*i*))) ‘ {*Max* (*insert j K*)<..}
and *ns* ∈ (λ*k*. *list-of* (*b k* (*j*,*i*))) ‘ {*Max* (*insert j K*)<..}
with *that* **obtain** *k k'* **where**
ms: *ms* = *list-of* (*b k* (*j*,*i*)) **and** *ns*: *ns* = *list-of* (*b k'* (*j*,*i*))
and *j* < *k* *j* < *k'* **and** *lt-k*: ∀ *a*∈*K*. *a* < *k* **and** *lt-k'*: ∀ *a*∈*K*. *a* < *k'*
by (*auto simp*: *K-finite*)
then have *len-eq* [*simp*]: *length ns* = *length ms*
by (*simp add*: *card-b*)
have *nz*: *length ns* ≠ *0*
using *b-ne* ‹*i* < *j*› ‹*j* < *k'*› *ns* **by** *auto*
show (*hd ms*, *hd ns*) ∈ *less-than* ⟷ (*ms*, *ns*) ∈ *?LL*
proof
assume (*hd ms*, *hd ns*) ∈ *less-than*
then show (*ms*, *ns*) ∈ *?LL*
using *that nz*
by (*fastforce simp*: *lenlex-def K-finite card-b intro*: *hd-lex*)
next
assume §: (*ms*, *ns*) ∈ *?LL*
then have (*list-of* (*b k'* (*j*,*i*)), *list-of* (*b k* (*j*,*i*))) ∉ *?LL*
using *less-asym ms ns omega-sum-1-less* **by** *blast*
then show (*hd ms*, *hd ns*) ∈ *less-than*
using ‹*j* < *k*› ‹*j* < *k'*› *Inf-b-less* [*of i j i j*] *ms ns*
by (*metis Cons-lenlex-iff* § *len-eq b-ne card-b-finite diff-Suc-1 hd-Cons-tl hd-b length-Cons less-or-eq-imp-le less-than-iff linorder-neqE-nat sorted-list-of-set-eq-Nil-iff that*(*2*))
qed
qed *auto*
also have … = ω

```
    using infN ordertype-nat-ω by blast
  finally show ?thesis .
qed

have otωj: ordertype (BB j0 j ‘ 𝒦 j0 j) ?LL = ω↑j if j ≤ j0 for j j0
  using that
proof (induction j) — a difficult proof, but no hints in Larson's text
  case 0
  then show ?case
    by (auto simp: XX-def)
next
  case (Suc j)
  then have ih: ordertype (BB j0 j ‘ 𝒦 j0 j) ?LL = ω ↑ j
    by simp
  have j ≤ j0
    by (simp add: Suc.prems Suc-leD)
  have inj-BB: inj-on (BB j0 j) ([{j0<..}]^j)
 proof (clarsimp simp: inj-on-def BB-def nsets-def sorted-list-of-set-Un less-sets-UN2)
    fix X Y
    assume X: X ⊆ {j0<..} and Y: Y ⊆ {j0<..}
      and finite X finite Y
      and jeq: j = card X
      and card Y = card X
      and eq: list-of (a j0 ∪ (⋃i<card X. b (enum X i) (j0,i)))
            = list-of (a j0 ∪ (⋃i<card X. b (enum Y i) (j0,i)))
    have enumX: ⋀n. ⟦n < card X⟧ ⟹ j0 ≤ enum X n
      using X ‹finite X› finite-enumerate-in-set less-imp-le-nat by blast
    have enumY: ⋀n. ⟦n < card X⟧ ⟹ j0 ≤ enum Y n
      using subsetD [OF Y]
    by (metis ‹card Y = card X› ‹finite Y› finite-enumerate-in-set greaterThan-iff
less-imp-le-nat)
    have smX: strict-mono-sets {..<card X} (λi. b (enum X i) (j0, i))
      and smY: strict-mono-sets {..<card X} (λi. b (enum Y i) (j0, i))
     using Suc.prems ‹card Y = card X› ‹finite X› ‹finite Y› bb enumX enumY
jeq
      by (auto simp: strict-mono-sets-def)

    have len-eq: length ms = length ns
      if (ms, ns) ∈ list.set (zip (map (list-of ∘ (λi. b (enum X i) (j0,i))) (list-of
{..<n}))
                                  (map (list-of ∘ (λi. b (enum Y i) (j0,i))) (list-of
{..<n})))
        n ≤ card X
      for ms ns n
      using that
    by (induction n rule: nat.induct) (auto simp: card-b enumX enumY)
    have concat (map (list-of ∘ (λi. b (enum X i) (j0, i))) (list-of {..<card X}))
      = concat (map (list-of ∘ (λi. b (enum Y i) (j0, i))) (list-of {..<card X}))
      using eq
```

by (*simp add*: *sorted-list-of-set-Un less-sets-UN2 sorted-list-of-set-UN-lessThan ab enumX enumY smX smY*)
then have *map-eq*: *map* (*list-of* ∘ (λ*i*. *b* (*enum X i*) (*j0*, *i*))) (*list-of* {..<*card X*})
= *map* (*list-of* ∘ (λ*i*. *b* (*enum Y i*) (*j0*, *i*))) (*list-of* {..<*card X*})
by (*rule concat-injective*) (*auto simp*: *len-eq split*: *prod.split*)
have *enum X i* = *enum Y i* **if** *i* < *card X* **for** *i*
proof −
have *Inf* (*b* (*enum X i*) (*j0*,*i*)) = *Inf* (*b* (*enum Y i*) (*j0*,*i*))
using *iffD1* [*OF map-eq-conv*, *OF map-eq*] *Suc.prems that*
by (*metis* (*mono-tags*, *lifting*) *card-b-finite comp-apply finite-lessThan lessThan-iff set-sorted-list-of-set*)
moreover have *Inf* (*b* (*enum X i*) (*j0*,*i*)) ∈ (*b* (*enum X i*) (*j0*,*i*))
Inf (*b* (*enum Y i*) (*j0*,*i*)) ∈ (*b* (*enum Y i*) (*j0*,*i*)) *i* < *j0*
using *Inf-nat-def1 Suc.prems b-ne enumX enumY jeq that* **by** *auto*
ultimately show *?thesis*
by (*metis Inf-b-less enumX enumY leI nat-less-le that*)
qed
then show *X* = *Y*
by (*simp add*: ‹*card Y* = *card X*› ‹*finite X*› ‹*finite Y*› *finite-enum-ext*)
qed
have *BB-Suc′*: *BB j0* (*Suc j*) *X* = *usplit* (λ*L k*. *BB j0 j L* @ *list-of* (*b k* (*j0*, *j*))) *X*
if *X* ∈ *USigma* (*K j0 j*) (λ*K*. {*Max* (*insert j0 K*)<..}) **for** *X*
using *that*
by (*simp add*: *USigma-iff K-finite less-sets-def usplit-def K-Suc BB-Suc* ‹*j* ≤ *j0*›)
have *ordertype* (*BB j0* (*Suc j*) ‘ *K j0* (*Suc j*)) *?LL*
= *ordertype*
(*usplit* (λ*L k*. *BB j0 j L* @ *list-of* (*b k* (*j0*, *j*))) ‘ *USigma* (*K j0 j*) (λ*K*. {*Max* (*insert j0 K*)<..})) *?LL*
by (*simp add*: *BB-Suc′ K-Suc*)
also have ... = ω ∗ *ordertype* (*BB j0 j* ‘ *K j0 j*) *?LL*
proof (*intro ordertype-append-image-IJ*)
fix *L k*
assume *L* ∈ *K j0 j* **and** *k* ∈ {*Max* (*insert j0 L*)<..}
then have *j0* < *k* **and** *L*: ⋀*a*. *a* ∈ *L* ⟹ *a* < *k*
by (*simp-all add*: *K-finite*)
then show *BB j0 j L* < *list-of* (*b k* (*j0*, *j*))
by (*simp add*: ‹*L* ∈ *K j0 j*› ‹*j* ≤ *j0*› *K-finite less-list-of*)
next
show *inj-on* (*BB j0 j*) (*K j0 j*)
by (*simp add*: *K-def inj-BB*)
next
fix *L*
assume *L*: *L* ∈ *K j0 j*
then show *L* ≪ {*Max* (*insert j0 L*)<..} ∧ *finite L*
by (*simp add*: *K-finite less-sets-def*)
show *ordertype* ((λ*i*. *list-of* (*b i* (*j0*, *j*))) ‘ {*Max* (*insert j0 L*)<..}) *?LL* = ω

using *L Suc.prems Suc-le-lessD otω* **by** *blast*
qed (*auto simp*: *K-finite card-b*)
also have ... = $\omega \uparrow$ *ord-of-nat* (*Suc j*)
by (*simp add*: *oexp-mult-commute ih*)
finally show *?case* **.**
qed

define *seqs* **where** *seqs* ≡ λ*j0 j K*. *list-of* (*a j0*) # (*map* (*list-of* ∘ (λ*i*. *b* (*enum K i*) (*j0*,*i*))) (*list-of* {..<*j*}))

have *length-seqs* [*simp*]: *length* (*seqs j0 j K*) = *Suc j* **for** *j0 j K*
by (*simp add*: *seqs-def*)

have *BB-eq-concat-seqs*: *BB j0 j K* = *concat* (*seqs j0 j K*)
and *seqs-ne*: *seqs j0 j K* ∈ *lists* (− {[]})
if *K*: *K* ∈ $\mathcal{K}$ *j0 j* **and** *j* ≤ *j0* **for** *K j j0*
proof −
have *j0*: ⋀*i*. *i* < *card K* ⟹ *j0* ≤ *enum K i* **and** *le-j0*: *card K* ≤ *j0*
using *finite-enumerate-in-set that* **unfolding** *K-def nsets-def* **by** *fastforce*+
show *BB j0 j K* = *concat* (*seqs j0 j K*)
using *that* **unfolding** *BB-def K-def nsets-def seqs-def*
by (*fastforce simp*: *j0 ab bb less-sets-UN2 sorted-list-of-set-Un*
strict-mono-sets-def sorted-list-of-set-UN-lessThan)
have *b* (*enum K i*) (*j0*, *i*) ≠ {} **if** *i* < *card K* **for** *i*
using *j0 le-j0 less-le-trans that* **by** *simp*
moreover have *card K* = *j*
using *K K-card* **by** *blast*
ultimately show *seqs j0 j K* ∈ *lists* (− {[]})
by (*clarsimp simp*: *seqs-def*) (*metis card-b-finite sorted-list-of-set-eq-Nil-iff*)
qed

have *BB-decomp*: ∃ *cs*. *BB j0 j K* = *concat cs* ∧ *cs* ∈ *lists* (− {[]})
if *K*: *K* ∈ $\mathcal{K}$ *j0 j* **and** *j* ≤ *j0* **for** *K j j0*
using *BB-eq-concat-seqs seqs-ne K that*(*2*) **by** *blast*

have *a-subset-M*: *a k* ⊆ *M k* **for** *k*
apply (*clarsimp simp*: *a-def M-def DF-simps F-def Let-def split*: *prod.split-asm*)
by (*metis* (*no-types*) *fst-conv fst-grab-subset nxt-subset snd-conv snd-grab-subset subsetD*)
have *ba-Suc*: *b k* (*j*,*i*) ≪ *a* (*Suc k*) **if** *i* < *j j* ≤ *k* **for** *i j k*
by (*meson a-subset-M bM less-sets-weaken2 nat-less-le that*)
have *ba*: *b k* (*j*,*i*) ≪ *a r* **if** *i* < *j j* ≤ *k k* < *r* **for** *i j k r*
by (*metis Suc-lessI a-ne aa ba-Suc less-sets-trans that*)

have *disjnt-ba*: *disjnt* (*b k* (*j*,*i*)) (*a r*) **if** *i* < *j j* ≤ *k* **for** *i j k r*
by (*meson ab ba disjnt-sym less-sets-imp-disjnt not-le that*)

have *bb-disjnt*: *disjnt* (*b k* (*j*,*i*)) (*b l* (*r*,*q*))
if *q* < *r i* < *j j* ≤ *k r* ≤ *l j* < *r* **for** *i j q r k l*

proof (*cases k=l*)
case *True*
with *that* **show** *?thesis*
by (*force simp*: *pair-less-def IJ-def intro*: *bb-same less-sets-imp-disjnt*)
next
case *False*
with *that* **show** *?thesis*
by (*metis bb less-sets-imp-disjnt disjnt-sym nat-neq-iff*)
qed

have *sum-card-b*: $(\sum i{<}j.\ card\ (b\ (enum\ K\ i)\ (j0,\ i))) = enum\ (d\ j0)\ j - enum\ (d\ j0)\ 0$
if *K*: $K \subseteq \{j0{<}..\}$ *finite K card K* $\geq$ *j0* **and** $j \leq j0$ **for** *j0 j K*
using ‹$j \leq j0$›
proof (*induction j*)
case *0*
then show *?case*
by *auto*
next
case (*Suc j*)
then have $j <$ *card K*
using *that*(*3*) **by** *linarith*
have *dis*: *disjnt* (*b* (*enum K j*) (*j0*, *j*)) $(\bigcup i{<}j.\ b\ (enum\ K\ i)\ (j0,\ i))$
unfolding *disjoint-UN-iff*
by (*meson Suc.prems b-disjoint-less disjnt-def disjnt-sym lessThan-iff less-Suc-eq that*)
have *j0-less*: *j0* $<$ *enum K j*
using *K* ‹$j <$ *card K*› **by** (*force simp*: *finite-enumerate-in-set*)
have $(\sum i{<}Suc\ j.\ card\ (b\ (enum\ K\ i)\ (j0,\ i)))$
$= card\ (b\ (enum\ K\ j)\ (j0,\ j)) + (\sum i{<}j.\ card\ (b\ (enum\ K\ i)\ (j0,\ i)))$
by (*simp add*: *lessThan-Suc card-Un-disjnt* [*OF - - dis*])
also have $\ldots = card\ (b\ (enum\ K\ j)\ (j0,\ j)) + enum\ (d\ j0)\ j - enum\ (d\ j0)\ 0$
using ‹*Suc j* $\leq$ *j0*› **by** (*simp add*: *Suc.IH split*: *nat-diff-split*)
also have $\ldots = enum\ (d\ j0)\ (Suc\ j) - enum\ (d\ j0)\ 0$
using *j0-less Suc.prems card-b less-or-eq-imp-le* **by** *force*
finally show *?case* **.**
qed

have *card-UN-b*: *card* $(\bigcup i{<}j.\ b\ (enum\ K\ i)\ (j0,\ i)) = enum\ (d\ j0)\ j - enum\ (d\ j0)\ 0$
if *K*: $K \subseteq \{j0{<}..\}$ *finite K card K* $\geq$ *j0* **and** $j \leq j0$ **for** *j0 j K*
using *that* **by** (*simp add*: *card-UN-disjoint sum-card-b b-disjoint*)

have *len-BB*: *length* (*BB j j K*) = *enum* (*d j*) *j*
if *K*: $K \in \mathcal{K}\ j\ j$ **and** $j \leq j$ **for** *j K*
proof −
have *dis-ab*: $\bigwedge i.\ i < j \Longrightarrow$ *disjnt* (*a j*) (*b* (*enum K i*) (*j*,*i*))
using *K* $\mathcal{K}$*-card* $\mathcal{K}$*-enum ab less-sets-imp-disjnt nat-less-le* **by** *blast*
show *?thesis*

```
    using K unfolding BB-def 𝒦-def nsets-def
   by (simp add: card-UN-b card-Un-disjnt dis-ab card-a cInf-le-finite finite-enumerate-in-set
enum-0-eq-Inf-finite)
  qed

  have d k ≪ d (Suc k) for k
    by (metis aM a-ne d-eq da less-sets-fst-grab less-sets-trans less-sets-weaken2
nxt-subset)
  then have dd: d k' ≪ d k if k' < k for k' k
   by (meson UNIV-I d-ne less-sets-imp-strict-mono-sets strict-mono-sets-def that)

  show thesis
  proof
    show (⋃ (range XX)) ⊆ WW
      by (auto simp: XX-def BB-def WW-def)
    show ordertype (⋃ (range XX)) (?LL) = ω ↑ ω
      using otωj by (simp add: XX-def ordertype-ωω)
  next
    fix U
    assume U: U ∈ [⋃ (range XX)]^2
    then obtain x y where Ueq: U = {x,y} and len-xy: length x ≤ length y
      by (auto simp: lenlex-nsets-2-eq lenlex-length)

    show ∃ l. Form l U ∧ (0 < l ⟶ [enum N l] < inter-scheme l U ∧ list.set
(inter-scheme l U) ⊆ N)
    proof (cases length x = length y)
      case True
      then show ?thesis
        using Form.intros(1) U Ueq by fastforce
    next
      case False
      then have xy: length x < length y
        using len-xy by auto
      obtain j r K L where K: K ∈ 𝒦 j j and xeq: x = BB j j K and ne: BB j j
K ≠ BB r r L
        and L: L ∈ 𝒦 r r and yeq: y = BB r r L
        using U by (auto simp: Ueq XX-def)
      then have length x = enum (d j) j length y = enum (d r) r
        by (auto simp: len-BB)
      then have j < r
        using xy dd
        by (metis card-d finite-enumerate-in-set finite-d lessI less-asym less-setsD
linorder-neqE-nat)
      then have aj-ar: a j ≪ a r
        using aa by auto
      have Ksub: K ⊆ {j<..} and finite K card K ≥ j
        using K by (auto simp: 𝒦-def nsets-def)
      have Lsub: L ⊆ {r<..} and finite L card L ≥ r
        using L by (auto simp: 𝒦-def nsets-def)
```

have *enumK*: *enum K i* > *j* **if** *i* < *j* **for** *i*
using *K* $\mathcal{K}$*-card* $\mathcal{K}$*-enum that* **by** *blast*
have *enumL*: *enum L i* > *r* **if** *i* < *r* **for** *i*
using *L* $\mathcal{K}$*-card* $\mathcal{K}$*-enum that* **by** *blast*
have *list.set* (*acc-lengths w* (*seqs j0 j K*)) ⊆ (+) *w* ‘ *d j0*
if *K*: *K* ⊆ {*j0*<..} *finite K card K* ≥ *j0* **and** *j* ≤ *j0* **for** *j0 j K w*
using ‹*j* ≤ *j0*›
proof (*induction j arbitrary*: *w*)
case *0*
then show *?case*
by (*simp add*: *seqs-def Inf-nat-def1 card-a*)
next
case (*Suc j*)
let *?db* = ∏ (*d j0*) + ((∑ *i*<*j*. *card* (*b* (*enum K i*) (*j0*,*i*))) + *card* (*b* (*enum K j*) (*j0*,*j*)))
have *j0* < *enum K j*
by (*meson Suc.prems Suc-le-lessD finite-enumerate-in-set greaterThan-iff le-trans subsetD K*)
then have *enum* (*d j0*) *j* ≥ ∏ (*d j0*)
using *Suc.prems card-d* **by** (*simp add*: *cInf-le-finite finite-enumerate-in-set*)
then have *?db* = *enum* (*d j0*) (*Suc j*)
using *Suc.prems that*
by (*simp add*: *cInf-le-finite finite-enumerate-in-set sum-card-b card-b enum-d-0* ‹*j0* < *enum K j*› *less-or-eq-imp-le*)
then have *?db* ∈ *d j0*
using *Suc.prems finite-enumerate-in-set* **by** (*auto simp*: *finite-enumerate-in-set*)
moreover have *list.set* (*acc-lengths w* (*seqs j0 j K*)) ⊆ (+) *w* ‘ *d j0*
by (*simp add*: *Suc Suc-leD*)
then have *list.set* (*acc-lengths* (*w* + ∏ (*d j0*))
(*map* (*list-of* ∘ (λ*i*. *b* (*enum K i*) (*j0*,*i*))) (*list-of* {..<*j*})))
⊆ (+) *w* ‘ *d j0*
by (*simp add*: *seqs-def card-a subset-insertI*)
ultimately show *?case*
by (*simp add*: *seqs-def acc-lengths-append image-iff Inf-nat-def1*
sum-sorted-list-of-set-map card-a)
qed
then have *acc-lengths-subset-d*: *list.set* (*acc-lengths 0* (*seqs j0 j K*)) ⊆ *d j0*
if *K*: *K* ⊆ {*j0*<..} *finite K card K* ≥ *j0* **and** *j* ≤ *j0* **for** *j0 j K*
by (*metis image-add-0 that*)

have *strict-sorted x strict-sorted y*
by (*auto simp*: *xeq yeq BB-def*)
have *disjnt-xy*: *disjnt* (*list.set x*) (*list.set y*)
proof −
have *disjnt* (*a j*) (*a r*)
using ‹*j* < *r*› *aa less-sets-imp-disjnt* **by** *blast*
moreover have *disjnt* (*b* (*enum K i*) (*j*,*i*)) (*a r*) **if** *i* < *j* **for** *i*
by (*simp add*: *disjnt-ba enumK less-imp-le-nat that*)
moreover have *disjnt* (*a j*) (*b* (*enum L q*) (*r*,*q*)) **if** *q* < *r* **for** *q*

```
      by (meson disjnt-ba disjnt-sym enumL less-imp-le-nat that)
    moreover have disjnt (b (enum K i) (j,i)) (b (enum L q) (r,q)) if i < j q
< r for i q
      by (meson ‹j < r› bb-disjnt enumK enumL less-imp-le that)
    ultimately show ?thesis
      by (simp add: xeq yeq BB-def)
    qed
    have ∃ us vs. merge (seqs j j K) (seqs r r L) us vs
    proof (rule merge-exists)
      show strict-sorted (concat (seqs j j K))
        using BB-eq-concat-seqs K ‹strict-sorted x› xeq by auto
      show strict-sorted (concat (seqs r r L))
        using BB-eq-concat-seqs L ‹strict-sorted y› yeq by auto
      show seqs j j K ∈ lists (− {[]}) seqs r r L ∈ lists (− {[]})
        by (auto simp: K L seqs-ne)
      show hd (seqs j j K) < hd (seqs r r L)
        by (simp add: aj-ar less-sets-imp-list-less seqs-def)
      show seqs j j K ≠ [] seqs r r L ≠ []
        using seqs-def by blast+
      have less-bb: b (enum K i) (j,i) ≪ b (enum L p) (r, p)
        if ¬ b (enum L p) (r, p) ≪ b (enum K i) (j,i) and i < j p < r
        for i p
          by (metis IJ-iff ‹j < r› bb bb-same enumK enumL less-imp-le-nat
linorder-neqE-nat pair-lessI1 that)
      show u < v ∨ v < u
        if u ∈ list.set (seqs j j K) and v ∈ list.set (seqs r r L) for u v
        using that enumK enumL unfolding seqs-def
        apply (auto simp: seqs-def aj-ar intro!: less-bb less-sets-imp-list-less)
        apply (meson ab ba less-imp-le-nat not-le)+
        done
    qed
    then obtain uus vvs where merge: merge (seqs j j K) (seqs r r L) uus vvs
      by metis
    then have uus ≠ []
      using merge-length1-gt-0 by (auto simp: seqs-def)
    then obtain u1 us where us: u1#us = uus
      by (metis neq-Nil-conv)
    define ku where ku ≡ length (u1#us)
    define ps where ps ≡ acc-lengths 0 (u1#us)
    have us-ne: u1#us ∈ lists (− {[]})
      using merge-length1-nonempty seqs-ne us merge us K by auto
    have xu-eq: x = concat (u1#us)
      using BB-eq-concat-seqs K merge merge-preserves us xeq by auto
    then have strict-sorted u1
      using ‹strict-sorted x› strict-sorted-append-iff by auto
    have u-sub: list.set ps ⊆ list.set (acc-lengths 0 (seqs j j K))
      using acc-lengths-merge1 merge ps-def us by blast
    have vvs ≠ []
      using merge BB-eq-concat-seqs L merge-preserves xy yeq by auto
```

then obtain *v1 vs* **where** *vs*: *v1*#*vs* = *vvs*
by (*metis neq-Nil-conv*)
define *kv* **where** *kv* ≡ *length* (*v1*#*vs*)
define *qs* **where** *qs* ≡ *acc-lengths 0* (*v1*#*vs*)
have *vs-ne*: *v1*#*vs* ∈ *lists* (− {[]})
using *L merge merge-length2-nonempty seqs-ne vs* **by** *auto*
have *yv-eq*: *y* = *concat* (*v1*#*vs*)
using *BB-eq-concat-seqs L merge merge-preserves vs yeq* **by** *auto*
then have *strict-sorted v1*
using ‹*strict-sorted y*› *strict-sorted-append-iff* **by** *auto*
have *v-sub*: *list.set qs* ⊆ *list.set* (*acc-lengths 0* (*seqs r r L*))
using *acc-lengths-merge2 merge qs-def vs* **by** *blast*

have *ss-concat-jj*: *strict-sorted* (*concat* (*seqs j j K*))
using *BB-eq-concat-seqs K* ‹*strict-sorted x*› *xeq* **by** *auto*
then obtain *k*: *0* < *kv kv* ≤ *ku ku* ≤ *Suc kv kv* ≤ *Suc j*
using *us vs merge-length-le merge-length-le-Suc merge-length-less2 merge*
unfolding *ku-def kv-def* **by** *fastforce*

define *zs* **where** *zs* ≡ *concat* [*ps*,*u1*,*qs*,*v1*] @ *interact us vs*
have *ss*: *strict-sorted zs*
proof −
have *ssp*: *strict-sorted ps*
unfolding *ps-def* **by** (*meson strict-sorted-acc-lengths us-ne*)
have *ssq*: *strict-sorted qs*
unfolding *qs-def* **by** (*meson strict-sorted-acc-lengths vs-ne*)

have *d j* ≪ *list.set x*
using *da* [*of j*] *db* [*of j*] *K K-card K-enum nat-less-le*
by (*auto simp*: *xeq BB-def less-sets-Un2 less-sets-UN2*)
then have *ac-x*: *acc-lengths 0* (*seqs j j K*) < *x*
by (*meson Ksub* ‹*finite K*› ‹*j* ≤ *card K*› *acc-lengths-subset-d le-refl less-sets-imp-list-less less-sets-weaken1*)
then have *ps* < *x*
by (*meson Ksub* ‹*d j* ≪ *list.set x*› ‹*finite K*› ‹*j* ≤ *card K*› *acc-lengths-subset-d le-refl less-sets-imp-list-less less-sets-weaken1 u-sub*)
then have *ps* < *u1*
by (*metis Nil-is-append-conv concat.simps*(*2*) *hd-append2 less-list-def xu-eq*)

have *d r* ≪ *list.set y*
using *da* [*of r*] *db* [*of r*] *L K-card K-enum nat-less-le*
by (*auto simp*: *yeq BB-def less-sets-Un2 less-sets-UN2*)
then have *acc-lengths 0* (*seqs r r L*) < *y*
by (*meson Lsub* ‹*finite L*› ‹*r* ≤ *card L*› *acc-lengths-subset-d le-refl less-sets-imp-list-less less-sets-weaken1*)
then have *qs* < *y*
by (*metis L Lsub K-card* ‹*d r* ≪ *list.set y*› ‹*finite L*› *acc-lengths-subset-d less-sets-imp-list-less less-sets-weaken1 order-refl v-sub*)
then have *qs* < *v1*

by (*metis concat.simps*(*2*) *gr-implies-not0 hd-append2 less-list-def list.size*(*3*) *xy yv-eq*)

have *carda-v1*: *card* (*a r*) ≤ *length v1*
using *length-hd-merge2* [*OF merge*] **unfolding** *vs* [*symmetric*] **by** (*simp add*: *seqs-def*)
have *ab-enumK*: ⋀*i*. *i* < *j* ⟹ *a j* ≪ *b* (*enum K i*) (*j*,*i*)
by (*meson ab enumK le-trans less-imp-le-nat*)

have *ab-enumL*: ⋀*q*. *q* < *r* ⟹ *a j* ≪ *b* (*enum L q*) (*r*,*q*)
by (*meson* ‹*j* < *r*› *ab enumL le-trans less-imp-le-nat*)
then have *ay*: *a j* ≪ *list.set y*
by (*auto simp*: *yeq BB-def less-sets-Un2 less-sets-UN2 aj-ar*)

have *disjnt-hd-last-K-y*: *disjnt* {*hd l*..*last l*} (*list.set y*)
if *l*: *l* ∈ *list.set* (*seqs j j K*) **for** *l*
proof (*clarsimp simp add*: *yeq BB-def disjnt-iff Ball-def*, *intro conjI strip*)
fix *u*
assume *u*: *u* ≤ *last l* **and** *hd l* ≤ *u*
with *l* **consider** *u* ≤ *last* (*list-of* (*a j*)) *hd* (*list-of* (*a j*)) ≤ *u*
| *i* **where** *i*<*j u* ≤ *last* (*list-of* (*b* (*enum K i*) (*j*,*i*))) *hd* (*list-of* (*b* (*enum K i*) (*j*,*i*))) ≤ *u*
by (*force simp*: *seqs-def*)
note *l-cases* = *this*
then show *u* ∉ *a r*
proof *cases*
case *1*
then show *?thesis*
by (*metis a-ne aj-ar finite-a last-in-set leD less-setsD set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff*)
next
case *2*
then show *?thesis*
by (*metis enumK ab ba Inf-nat-def1 b-ne card-b-finite hd-b last-in-set less-asym less-setsD not-le set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff*)
qed
fix *q*
assume *q* < *r*
show *u* ∉ *b* (*enum L q*) (*r*,*q*)
using *l-cases*
proof *cases*
case *1*
then show *?thesis*
by (*metis* ‹*q* < *r*› *a-ne ab-enumL finite-a last-in-set leD less-setsD set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff*)
next
case *2*
show *?thesis*
proof (*cases enum K i* = *enum L q*)

```
            case True
            then show ?thesis
              using 2 bb-same [of concl: enum L q j i r q] ‹j < r› u
            by (metis IJ-iff b-ne card-b-finite enumK last-in-set leD less-imp-le-nat
less-setsD pair-lessI1 set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff)
          next
            case False
            with 2 bb enumK enumL show ?thesis
              unfolding less-sets-def
                by (metis ‹q < r› b-ne card-b-finite last-in-set leD less-imp-le-nat
list.set-sel(1) nat-neq-iff set-sorted-list-of-set sorted-list-of-set-eq-Nil-iff)
          qed
        qed
      qed

      have u1-y: list.set u1 ≪ list.set y
       using vs yv-eq L ‹strict-sorted y› merge merge-less-sets-hd merge-preserves
seqs-ne ss-concat-jj us by fastforce
      have u1-subset-seqs: list.set u1 ⊆ list.set (concat (seqs j j K))
        using merge-preserves [OF merge] us by auto

      have b k (j,i) ≪ d (Suc k) if j≤k i<j for k j i
        by (metis bM d-eq less-sets-fst-grab less-sets-weaken2 nxt-subset that)
      then have bd: b k (j,i) ≪ d k' if j≤k i<j k < k' for k k' j i
        by (metis Suc-lessI d-ne dd less-sets-trans that)

      have a k ≪ d (Suc k) for k
        by (metis aM d-eq less-sets-fst-grab less-sets-weaken2 nxt-subset)
      then have ad: a k ≪ d k' if k<k' for k k'
        by (metis Suc-lessI d-ne dd less-sets-trans that)

      have u1 < y
        by (simp add: u1-y less-sets-imp-list-less)
      have n < Inf (d r) if n: n ∈ list.set u1 for n
      proof −
        obtain l where l: l ∈ list.set (seqs j j K) and n: n ∈ list.set l
          using n u1-subset-seqs by auto
        then consider l = list-of (a j) | i where l = list-of (b (enum K i) (j,i))
i < j
          by (force simp: seqs-def)
        then show ?thesis
        proof cases
          case 1
          then show ?thesis
        by (metis Inf-nat-def1 ‹j < r› ad d-ne finite-a less-setsD n set-sorted-list-of-set)
        next
          case 2
          then have hd (list-of (b (enum K i) (j,i))) = Min (b (enum K i) (j,i))
            by (meson b-ne card-b-finite enumK hd-list-of less-imp-le-nat)
```

also have $\ldots \leq n$
using *2 n* **by** (*simp add*: *less-list-def disjnt-iff less-sets-def*)
also have *f8*: $n < hd\ y$
using *less-setsD that u1-y*
by (*metis gr-implies-not0 list.set-sel*(*1*) *list.size*(*3*) *xy*)
finally have $l < y$
using *2 disjnt-hd-last-K-y* [*OF l*]
by (*simp add*: *disjnt-iff*) (*metis leI less-imp-le-nat less-list-def list.set-sel*(*1*))
moreover have *last* (*list-of* (*b* (*enum K i*) (*j*,*i*))) < *hd* (*list-of* (*a r*))
using ‹$l < y$› *L n* **by** (*auto simp*: *2yeq BB-eq-concat-seqs seqs-def less-list-def*)
then have *enum K i* < *r*
by (*metis 2*(*1*) *a-ne ab card-b-finite empty-iff finite.emptyI finite-a last-in-set leI less-asym less-setsD list.set-sel*(*1*) *n set-sorted-list-of-set*)
moreover have $j \leq$ *enum K i*
by (*simp add*: *2*(*2*) *enumK less-imp-le-nat*)
ultimately show *?thesis*
using *2 n bd* [*of j enum K i i r*] *Inf-nat-def1 less-setsD* **by** *force*
qed
qed
then have *last u1* < *Inf* (*d r*)
using ‹*uus* ≠ []› *us-ne* **by** *auto*
also have $\ldots \leq$ *length v1*
using *card-a carda-v1* **by** *auto*
finally have *last u1* < *length v1* **.**
then have *u1* < *qs*
by (*simp add*: *qs-def less-list-def*)

have *strict-sorted* (*interact* (*u1*#*us*) (*v1*#*vs*))
using *L* ‹*strict-sorted x*› ‹*strict-sorted y*› *merge merge-interact merge-preserves seqs-ne us vs xu-eq yv-eq* **by** *auto*
then have *strict-sorted* (*interact us vs*) *v1* < *interact us vs*
by (*auto simp*: *strict-sorted-append-iff*)
moreover have *ps* < *u1* @ *qs* @ *v1* @ *interact us vs*
using ‹*ps* < *u1*› *us-ne* **unfolding** *less-list-def* **by** *auto*
moreover have *u1* < *qs* @ *v1* @ *interact us vs*
by (*metis* ‹*u1* < *qs*› ‹*vvs* ≠ []› *acc-lengths-eq-Nil-iff hd-append less-list-def qs-def vs*)
moreover have *qs* < *v1* @ *interact us vs*
using ‹*qs* < *v1*› *us-ne* ‹*last u1* < *length v1*› *vs-ne* **by** (*auto simp*: *less-list-def*)
ultimately show *?thesis*
by (*simp add*: *zs-def strict-sorted-append-iff ssp ssq* ‹*strict-sorted u1*› ‹*strict-sorted v1*›)
qed
have *ps-subset-d*: *list.set ps* ⊆ *d j*
using *K Ksub K-card* ‹*finite K*› *acc-lengths-subset-d u-sub* **by** *blast*
have *ps-less-u1*: *ps* < *u1*

```
    by (metis append.assoc concat.simps(2) ss strict-sorted-append-iff zs-def)
  have qs-subset-d: list.set qs ⊆ d r
    using L Lsub 𝒦-card ‹finite L› acc-lengths-subset-d v-sub by blast
  have qs-less-v1: qs < v1
    by (metis append.assoc concat.simps(2) ss strict-sorted-append-iff zs-def)
  have FB: Form-Body ku kv x y zs
    unfolding Form-Body.simps ku-def kv-def
    using ps-def qs-def ss us-ne vs-ne xu-eq xy yv-eq zs-def by blast
  then have zs = (inter-scheme ((ku+kv) − Suc 0) {x,y})
    by (simp add: Form-Body-imp-inter-scheme k)
 obtain l where l ≤ 2 ∗ (Suc j) and l: Form l U and zs-eq-interact: zs =
inter-scheme l {x,y}
  proof
    show ku+kv−1 ≤ 2 ∗ (Suc j)
      using k by auto
    show Form (ku+kv−1) U
    proof (cases ku=kv)
      case True
      then show ?thesis
        using FB Form.simps Ueq ‹0 < kv› by (auto simp: mult-2)
    next
      case False
      then have ku = Suc kv
        using k by auto
      then show ?thesis
        using FB Form.simps Ueq ‹0 < kv› by auto
    qed
    show zs = inter-scheme (ku + kv − 1) {x, y}
      using Form-Body-imp-inter-scheme by (simp add: FB k)
  qed
  then have enum N l ≤ enum N (Suc (2 ∗ Suc j))
    by (simp add: assms less-imp-le-nat)
  also have ... < Min (d j)
 by (smt (verit, best) Min-gr-iff d-eq d-ne finite-d fst-grab-subset greaterThan-iff
in-mono le-inf-iff nxt-def)
  finally have ls: {enum N l} ≪ d j
    by simp
  have l > 0
    by (metis l False Form-0-cases-raw Set.doubleton-eq-iff Ueq gr0I)
  show ?thesis
    unfolding Ueq
  proof (intro exI conjI impI)
     have zs-subset: list.set zs ⊆ list.set (acc-lengths 0 (seqs j j K)) ∪ list.set
(acc-lengths 0 (seqs r r L)) ∪ list.set x ∪ list.set y
      using u-sub v-sub by (auto simp: zs-def xu-eq yv-eq)
    also have ... ⊆ N
    proof (simp, intro conjI)
      show list.set (acc-lengths 0 (seqs j j K)) ⊆ N
        using d-subset-N Ksub ‹finite K› ‹j ≤ card K› acc-lengths-subset-d by
```

blast
        show *list.set* (*acc-lengths 0* (*seqs r r L*)) ⊆ *N*
          using *d-subset-N Lsub* ‹*finite L*› ‹*r* ≤ *card L*› *acc-lengths-subset-d* **by**
blast
        show *list.set x* ⊆ *N list.set y* ⊆ *N*
          by (*simp-all add*: *xeq yeq BB-def a-subset-N UN-least b-subset-N*)
      qed
      finally show *list.set* (*inter-scheme l* {*x*, *y*}) ⊆ *N*
        using *zs-eq-interact* **by** *blast*
      have [*enum N l*] < *ps*
        using *ps-subset-d ls*
       by (*metis empty-set less-sets-imp-list-less less-sets-weaken2 list.simps*(*15*))
      then show [*enum N l*] < *inter-scheme l* {*x*, *y*}
        by (*simp add*: *zs-def less-list-def ps-def flip*: *zs-eq-interact*)
    qed (*use Ueq l* **in** *blast*)
  qed
 qed
qed

3.12 The main partition theorem for $\omega \uparrow \omega$

definition *iso-ll* **where** *iso-ll A B* ≡ *iso* (*lenlex less-than* ∩ (*A*×*A*)) (*lenlex less-than*
∩ (*B*×*B*))

corollary *ordertype-eq-ordertype-iso-ll*:
 assumes *Field* (*Restr* (*lenlex less-than*) *A*) = *A Field* (*Restr* (*lenlex less-than*)
B) = *B*
 shows (*ordertype A* (*lenlex less-than*) = *ordertype B* (*lenlex less-than*))
       ⟷ (∃*f*. *iso-ll A B f*)
proof −
 have *total-on A* (*lenlex less-than*) ∧ *total-on B* (*lenlex less-than*)
   by (*meson UNIV-I total-lenlex total-on-def total-on-less-than*)
 then show *?thesis*
  by (*simp add*: *assms wf-lenlex lenlex-transI iso-ll-def ordertype-eq-ordertype-iso-Restr*)
qed

theorem *partition-ωω-aux*:
 assumes *α* ∈ *elts ω*
 shows *partn-lst* (*lenlex less-than*) *WW* [*ω*↑*ω*,*α*] *2* (**is** *partn-lst ?R WW* [*ω*↑*ω*,*α*]
2)
proof (*cases α* ≤ *1*)
 case *True*
 then show *?thesis*
  using *strict-sorted-into-WW* **unfolding** *WW-def* **by** (*auto intro*!: *partn-lst-triv1* [**where**
i=*1*])
next
 case *False*
 obtain *m* **where** *m*: *α* = *ord-of-nat m*
   using *assms elts-ω* **by** *auto*

then have $m>1$
using *False* **by** *auto*
show *?thesis*
unfolding *partn-lst-def*
proof *clarsimp*
fix f
assume f: $f \in [WW]^2 \to \{..<Suc\ (Suc\ 0)\}$
let $?P0 = \exists X \subseteq WW.\ ordertype\ X\ ?R = \omega{\uparrow}\omega \wedge f\ `\ [X]^2 \subseteq \{0\}$
let $?P1 = \exists M \subseteq WW.\ ordertype\ M\ ?R = \alpha \wedge f\ `\ [M]^2 \subseteq \{1\}$
have †: $?P0 \vee ?P1$
proof (*rule disjCI*)
assume *not1*: $\neg\ ?P1$
have $\exists W'.\ ordertype\ W'\ ?R = \omega{\uparrow}n \wedge f\ `\ [W']^2 \subseteq \{0\} \wedge W' \subseteq WW\text{-}seg\ (n{*}m)$ **for** n::*nat*
proof −
have *fnm*: $f \in [WW\text{-}seg\ (n{*}m)]^2 \to \{..<Suc\ (Suc\ 0)\}$
using *f WW-seg-subset-WW* [*of* $n{*}m$] **by** (*meson in-mono nsets-Pi-contra*)
have $*$: *partn-lst ?R* (*WW-seg* $(n{*}m)$) [$\omega{\uparrow}n$, *ord-of-nat m*] *2*
using *ordertype-WW-seg* [*of* $n{*}m$]
by (*simp add*: *partn-lst-VWF-imp-partn-lst* [*OF Theorem-3-2*])
show *?thesis*
using *partn-lst-E* [*OF* $*$ *fnm*, *simplified*]
by (*metis One-nat-def WW-seg-subset-WW less-2-cases m not1 nth-Cons-0 nth-Cons-Suc numeral-2-eq-2 subset-trans*)
qed
then obtain W':: *nat* ⇒ *nat list set*
where *otW'*: $\bigwedge n.\ ordertype\ (W'\ n)\ ?R = \omega{\uparrow}n$
and *f-W'*: $\bigwedge n.\ f\ `\ [W'\ n]^2 \subseteq \{0\}$
and *seg-W'*: $\bigwedge n.\ W'\ n \subseteq WW\text{-}seg\ (n{*}m)$
by *metis*
define WW' **where** $WW' \equiv (\bigcup n.\ W'\ n)$
have $WW' \subseteq WW$
using *seg-W' WW-seg-subset-WW* **by** (*force simp*: *WW'-def*)
with f **have** f': $f \in [WW']^2 \to \{..<Suc\ (Suc\ 0)\}$
using *nsets-mono* **by** *fastforce*
have *ot'*: $ordertype\ WW'\ ?R = \omega{\uparrow}\omega$
proof (*rule antisym*)
have $ordertype\ WW'\ ?R \leq ordertype\ WW\ ?R$
by (*simp add*: ‹$WW' \subseteq WW$› *lenlex-transI ordertype-mono wf-lenlex*)
with *ordertype-WW*
show $ordertype\ WW'\ ?R \leq \omega \uparrow \omega$
by *simp*
have $\omega \uparrow n \leq ordertype\ (\bigcup\ (range\ W'))\ ?R$ **for** n::*nat*
using *oexp-Limit ordertype-ωω otW'* **by** *auto*
then show $\omega \uparrow \omega \leq ordertype\ WW'\ ?R$
by (*auto simp*: *elts-ω oexp-Limit ZFC-in-HOL.SUP-le-iff WW'-def*)
qed
have *FR-WW*: $Field\ (Restr\ (lenlex\ less\text{-}than)\ WW) = WW$
by (*simp add*: *Limit-omega-oexp Limit-ordertype-imp-Field-Restr order-*

type-WW)
have *FR-WW'*: *Field* (*Restr* (*lenlex less-than*) *WW'*) = *WW'*
by (*simp add*: *Limit-omega-oexp Limit-ordertype-imp-Field-Restr ot'*)
have *FR-W*: *Field* (*Restr* (*lenlex less-than*) (*WW-seg n*)) = *WW-seg n* **if** *n>0* **for** *n*
by (*simp add*: *Limit-omega-oexp ordertype-WW-seg that Limit-ordertype-imp-Field-Restr*)
have *FR-W'*: *Field* (*Restr* (*lenlex less-than*) (*W' n*)) = *W' n* **if** *n>0* **for** *n*
by (*simp add*: *Limit-omega-oexp otW' that Limit-ordertype-imp-Field-Restr*)
have ∃ *h*. *iso-ll* (*WW-seg n*) (*W' n*) *h* **if** *n>0* **for** *n*
proof (*subst ordertype-eq-ordertype-iso-ll* [*symmetric*])
show *ordertype* (*WW-seg n*) (*lenlex less-than*) = *ordertype* (*W' n*) (*lenlex less-than*)
by (*simp add*: *ordertype-WW-seg otW'*)
qed (*auto simp*: *FR-W FR-W' that*)
then obtain *h-seg* **where** *h-seg*: ⋀*n*. *n* > *0* ⟹ *iso-ll* (*WW-seg n*) (*W' n*) (*h-seg n*)
by *metis*
define *h* **where** *h* ≡ λ*l*. *if l*=[] *then* [] *else h-seg* (*length l*) *l*

have *bij-h-seg*: ⋀*n*. *n* > *0* ⟹ *bij-betw* (*h-seg n*) (*WW-seg n*) (*W' n*)
using *h-seg* **by** (*simp add*: *iso-ll-def iso-iff2 FR-W FR-W'*)
have *len-h-seg*: *length* (*h-seg* (*length l*) *l*) = *length l* ∗ *m*
if *length l* > *0 l* ∈ *WW* **for** *l*
using *bij-betwE* [*OF bij-h-seg*] *seg-W' that* **by** (*simp add*: *WW-seg-def subset-iff*)
have *hlen*: *length* (*h x*) = *length* (*h y*) ⟷ *length x* = *length y* **if** *x* ∈ *WW y* ∈ *WW* **for** *x y*
using *that* ‹*1* < *m*› *h-def len-h-seg* **by** *force*

have *h*: *iso-ll WW WW' h*
unfolding *iso-ll-def iso-iff2 FR-WW FR-WW'*
proof (*intro conjI strip*)
have *W'-ne*: *W' n* ≠ {} **for** *n*
using *otW'* [*of n*] **by** *auto*
then have [] ∈ *WW'*
using *seg-W'* [*of 0*] **by** (*auto simp*: *WW'-def WW-seg-def*)
let *?g* = λ*l*. *if l*=[] *then l else inv-into* (*WW-seg* (*length l div m*)) (*h-seg* (*length l div m*)) *l*
have *h-seg-iff*: ⋀*n a b*. ⟦*a* ∈ *WW-seg n*; *b* ∈ *WW-seg n*; *n>0*⟧ ⟹
(*a*, *b*) ∈ *lenlex less-than* ⟷
(*h-seg n a*, *h-seg n b*) ∈ *lenlex less-than* ∧ *h-seg n a* ∈ *W' n* ∧ *h-seg n b* ∈ *W' n*
using *h-seg* **by** (*auto simp*: *iso-ll-def iso-iff2 FR-W FR-W'*)

show *bij-betw h WW WW'*
unfolding *bij-betw-iff-bijections*
proof (*intro exI conjI ballI*)
fix *l*
assume *l* ∈ *WW*

```
      then have l: l ∈ WW-seg (length l)
        by (simp add: WW-seg-def)
      have h l ∈ W' (length l)
      proof (cases l=[])
        case True
        with seg-W' [of 0] W'-ne show ?thesis
          by (auto simp: WW-seg-def h-def)
      next
        case False
        then show ?thesis
          using bij-betwE bij-h-seg h-def l by fastforce
      qed
      show h l ∈ WW'
        using WW'-def ‹h l ∈ W' (length l)› by blast
      show ?g (h l) = l
      proof (cases l=[])
        case False
        then have length l > 0
          by auto
        then have h-seg (length l) l ≠ []
          using ‹1 < m› ‹l ∈ WW› len-h-seg by fastforce
          moreover have bij-betw (h-seg (length l)) (WW-seg (length l)) (W'
(length l))
          using ‹0 < length l› bij-h-seg by presburger
        ultimately show ?thesis
          using ‹l ∈ WW› bij-betw-inv-into-left h-def l len-h-seg by fastforce
      qed (auto simp: h-def)
    next
      fix l
      assume l ∈ WW'
      then have l: l ∈ W' (length l div m)
        using WW-seg-def ‹1 < m› seg-W' by (fastforce simp: WW'-def)
      show ?g l ∈ WW
      proof (cases l=[])
        case False
        then have l ∉ W' 0
          using WW-seg-def seg-W' by fastforce
        with l have inv-into (WW-seg (length l div m)) (h-seg (length l div m))
l ∈ WW-seg (length l div m)
          by (metis Nat.neq0-conv bij-betwE bij-betw-inv-into bij-h-seg)
        then show ?thesis
          using False WW-seg-subset-WW by auto
      qed (auto simp: WW-def)

      show h (?g l) = l
      proof (cases l=[])
        case False
        then have 0 < length l div m
          using WW-seg-def l seg-W' by fastforce
```

```
        then have inv-into (WW-seg (length l div m)) (h-seg (length l div m)) l
∈ WW-seg (length l div m)
          by (metis bij-betw-imp-surj-on bij-h-seg inv-into-into l)
        then show ?thesis
          using bij-h-seg [of length l div m] WW-seg-def ‹0 < length l div m›
bij-betw-inv-into-right l
          by (fastforce simp: h-def)
      qed (auto simp: h-def)
    qed
    fix a b
    assume a ∈ WW b ∈ WW
    show (a, b) ∈ Restr (lenlex less-than) WW ⟷ (h a, h b) ∈ Restr (lenlex
less-than) WW'
      (is ?lhs = ?rhs)
    proof
      assume L: ?lhs
        then consider length a < length b | length a = length b (a, b) ∈ lex
less-than
        by (auto simp: lenlex-conv)
      then show ?rhs
      proof cases
        case 1
        then have length (h a) < length (h b)
          using ‹1 < m› ‹a ∈ WW› ‹b ∈ WW› h-def len-h-seg by auto
        then have (h a, h b) ∈ lenlex less-than
          by (auto simp: lenlex-conv)
        then show ?thesis
          using ‹a ∈ WW› ‹b ∈ WW› ‹bij-betw h WW WW'› bij-betwE by
fastforce
      next
        case 2
        then have ab: a ∈ WW-seg (length a) b ∈ WW-seg (length a)
          using ‹a ∈ WW› ‹b ∈ WW› by (auto simp: WW-seg-def)
        have length (h a) = length (h b)
          using 2 ‹a ∈ WW› ‹b ∈ WW› h-def len-h-seg by force
        moreover have (a, b) ∈ lenlex less-than
          using L by blast
        then have (h-seg (length a) a, h-seg (length a) b) ∈ lenlex less-than
          using 2 ab h-seg-iff by blast
        ultimately show ?thesis
          using 2 ‹a ∈ WW› ‹b ∈ WW› ‹bij-betw h WW WW'› bij-betwE h-def
by fastforce
      qed
    next
      assume R: ?rhs
      then have R': (h a, h b) ∈ lenlex less-than
        by blast
      then consider length a < length b
        | length a = length b (h a, h b) ∈ lex less-than
```

```
      using ‹a ∈ WW› ‹b ∈ WW› ‹m > 1›
      by (auto simp: lenlex-conv h-def len-h-seg split: if-split-asm)
    then show ?lhs
    proof cases
      case 1
      then show ?thesis
        using omega-sum-less-iff ‹a ∈ WW› ‹b ∈ WW› by auto
    next
      case 2
      then have ab: a ∈ WW-seg (length a) b ∈ WW-seg (length a)
        using ‹a ∈ WW› ‹b ∈ WW› by (auto simp: WW-seg-def)
      then have (a, b) ∈ lenlex less-than
        using bij-betwE [OF bij-h-seg] ‹a ∈ WW› ‹b ∈ WW› R' 2
        by (simp add: h-def h-seg-iff split: if-split-asm)
      then show ?thesis
        using ‹a ∈ WW› ‹b ∈ WW› by blast
    qed
  qed
qed

let ?fh = f ∘ image h
have bij-betw h WW WW'
 using h unfolding iso-ll-def iso-iff2 by (fastforce simp: FR-WW FR-WW')
moreover have {..<Suc (Suc 0)} = {0,1}
  by auto
ultimately have fh: ?fh ∈ [WW]^2 → {0,1}
unfolding Pi-iff using bij-betwE f' bij-betw-nsets by (metis PiE comp-apply)
have f{x,y} = 0 if x ∈ WW' y ∈ WW' length x = length y x ≠ y for x y
proof −
  obtain p q where x ∈ W' p and y ∈ W' q
    using WW'-def ‹x ∈ WW'› ‹y ∈ WW'› by blast
  then obtain n where {x,y} ∈ [W' n]^2
    using seg-W' ‹1 < m› ‹length x = length y› ‹x ≠ y›
    by (auto simp: WW'-def WW-seg-def subset-iff)
  then show f{x,y} = 0
    using f-W' by blast
qed
 then have fh-eq-0-eqlen: ?fh{x,y} = 0 if x ∈ WW y ∈ WW length x = length
y x≠y for x y
  using ‹bij-betw h WW WW'› that hlen by (simp add: bij-betw-iff-bijections)
metis
have m-f-0: ∃x∈[M]^2. f x = 0 if M ⊆ WW card M = m for M
proof −
  have finite M
    using False m that by auto
  with not1 [simplified, rule-format, of M] that
  have ∃x∈[M]^2. f x ≠ Suc 0
    by (simp add: image-subset-iff finite-ordertype-eq-card m)
  with that show ?thesis
```

by (*metis PiE f lessThan-iff less-2-cases nsets-mono numeral-2-eq-2 subset-iff*)
qed
have *m-fh-0*: $\exists x{\in}[M]^2$. *?fh x* = *0* **if** $M \subseteq WW$ *card M* = *m* **for** *M*
proof −
have h ‘ $M \subseteq WW$
using ‹$WW' \subseteq WW$› ‹*bij-betw h WW WW'*› *bij-betwE that*(*1*) **by** *fastforce*
moreover have *card* (h ‘ M) = *m*
by (*metis* ‹*bij-betw h WW WW'*› *bij-betw-def bij-betw-subset card-image that*)
ultimately have $\exists x \in [h$ ‘ $M]^2$. $f\, x$ = *0*
by (*metis m-f-0*)
then obtain *Y* **where** *Y*: *f* (h ‘ *Y*) = *0* *Y* $\subseteq$ *M* **and** *finite* (h ‘ *Y*) *card* (h ‘ *Y*) = *2*
by (*auto simp*: *nsets-def subset-image-iff*)
then have *card Y* = *2*
using ‹*bij-betw h WW WW'*› ‹$M \subseteq WW$›
by (*metis bij-betw-def card-image inj-on-subset*)
with *Y card.infinite*[*of Y*] **show** *?thesis*
by (*auto simp*: *nsets-def*)
qed

obtain *N j* **where** *infinite N*
and *N*: $\bigwedge k\ u.\ [\![k > 0;\ u \in [WW]^2;\ Form\ k\ u;\ [enum\ N\ k] < inter\text{-}scheme\ k\ u;\ List.set\ (inter\text{-}scheme\ k\ u) \subseteq N]\!] \Longrightarrow$ *?fh u* = *j k*
using *lemma-3-6* [*OF fh*] **by** *blast*

have *infN'*: *infinite* (*enum N* ‘ {*k*<..}) **for** *k*
by (*simp add*: ‹*infinite N*› *enum-works finite-image-iff infinite-Ioi strict-mono-imp-inj-on*)
have *j-0*: *j k* = *0* **if** *k*>*0* **for** *k*
proof −
obtain *M* **where** *M*: $M \in [WW]^m$
and *MF*: $\bigwedge u.\ u \in [M]^2 \Longrightarrow Form\ k\ u$
and *Mi*: $\bigwedge u.\ u \in [M]^2 \Longrightarrow List.set\ (inter\text{-}scheme\ k\ u) \subseteq enum\ N$ ‘ {*k*<..}
using *lemma-3-7* [*OF infN'* ‹*k* > *0*›] **by** *metis*
obtain *u* **where** *u*: $u \in [M]^2$ *?fh u* = *0*
using *m-fh-0* [*of M*] *M* [*unfolded nsets-def*] **by** *force*
moreover
have §: *Form k u List.set* (*inter-scheme k u*) $\subseteq$ *enum N* ‘ {*k*<..}
by (*simp-all add*: *MF Mi* ‹$u \in [M]^2$›)
then have *hd* (*inter-scheme k u*) $\in$ *enum N* ‘ {*k*<..}
using *hd-in-set inter-scheme-simple that* **by** *blast*
then have [*enum N k*] < *inter-scheme k u*
using *strict-mono-enum* [*OF* ‹*infinite N*›] **by** (*auto simp*: *less-list-def strict-mono-def*)
moreover have $u \in [WW]^2$
using *M u* **by** (*auto simp*: *nsets-def*)
moreover have *enum N* ‘ {*k*<..} $\subseteq$ *N*

```
      using ‹infinite N› range-enum by auto
    ultimately show ?thesis
      using N § that by auto
  qed
  obtain X where X ⊆ WW and otX: ordertype X (lenlex less-than) = ω↑ω
      and X: ⋀u. u ∈ [X]^2 ⟹
          ∃ l. Form l u ∧ (l > 0 ⟶ [enum N l] < inter-scheme l u ∧ List.set
(inter-scheme l u) ⊆ N)
    using lemma-3-8 [OF ‹infinite N›] ot' by blast
  have 0: ?fh ‘ [X]^2 ⊆ {0}
  proof clarsimp
    fix u
    assume u: u ∈ [X]^2
    obtain l where Form l u and l: l > 0 ⟶ [enum N l] < inter-scheme l u
∧ List.set (inter-scheme l u) ⊆ N
      using u X by blast
    have ?fh u = 0
    proof (cases l = 0)
      case True
      then show ?thesis
       by (metis Form-0-cases-raw ‹Form l u› ‹X ⊆ WW› doubleton-in-nsets-2
fh-eq-0-eqlen subset-iff u)
    next
      case False
      then obtain [enum N l] < inter-scheme l u List.set (inter-scheme l u) ⊆
N j l = 0
        using Nat.neq0-conv j-0 l by blast
      with False show ?thesis
       using ‹X ⊆ WW› N inter-scheme ‹Form l u› doubleton-in-nsets-2 u by
(auto simp: nsets-def)
    qed
    then show f (h ‘ u) = 0
      by auto
  qed
  show ?P0
  proof (intro exI conjI)
    show h ‘ X ⊆ WW
   using ‹WW' ⊆ WW› ‹X ⊆ WW› ‹bij-betw h WW WW'› bij-betw-imp-surj-on
by fastforce
    show ordertype (h ‘ X) (lenlex less-than) = ω ↑ ω
    proof (subst ordertype-inc-eq)
      show (h x, h y) ∈ lenlex less-than
        if x ∈ X y ∈ X (x, y) ∈ lenlex less-than for x y
        using that h ‹X ⊆ WW› by (auto simp: FR-WW FR-WW' iso-iff2
iso-ll-def)
    qed (use otX in auto)
    show f ‘ [h ‘ X]^2 ⊆ {0}
    proof (clarsimp simp: image-subset-iff nsets-def)
      fix Y
```

```
        assume Y: Y ⊆ h ‘ X finite Y card Y = 2
        then have inv-into WW h ‘ Y ⊆ X
          by (metis ‹X ⊆ WW› ‹bij-betw h WW WW'› bij-betw-inv-into-LEFT
image-mono)
        moreover have finite (inv-into WW h ‘ Y)
          using ‹finite Y› by blast
        moreover have card (inv-into WW h ‘ Y) = 2
        using Y by (metis ‹X ⊆ WW› card-image inj-on-inv-into subset-image-iff
subset-trans)
        ultimately have f (h ‘ inv-into WW h ‘ Y) = 0
          using 0 by (auto simp: image-subset-iff nsets-def)
        then show f Y = 0
          by (metis ‹X ⊆ WW› ‹Y ⊆ h ‘ X› image-inv-into-cancel image-mono
order-trans)
      qed
    qed
  qed
  then show ∃i<Suc (Suc 0). ∃H⊆WW. ordertype H ?R = [ω↑ω, α] ! i ∧ f ‘
[H]^2 ⊆ {i}
    by (metis One-nat-def lessI nth-Cons-0 nth-Cons-Suc zero-less-Suc)
 qed
qed
```

Theorem 3.1 of Jean A. Larson, ibid.

```
theorem partition-ωω: α ∈ elts ω ⟹ partn-lst-VWF (ω↑ω) [ω↑ω,α] 2
 using partn-lst-imp-partn-lst-VWF-eq [OF partition-ωω-aux] ordertype-WW by
auto

end
```

4 Acknowledgements

The author was supported by the ERC Advanced Grant ALEXANDRIA (Project 742178) funded by the European Research Council. Many thanks to Mirna Džamonja (who suggested the project) and Angeliki Koutsoukou-Argyraki for assistance at tricky moments.

References

[1] P. Erdös and E. C. Milner. A theorem in the partition calculus. *Canadian Mathematical Bulletin*, 15(4):501–505, Dec. 1972.

[2] P. Erdös and E. C. Milner. A theorem in the partition calculus corrigendum. *Canadian Mathematical Bulletin*, 17(2):305, June 1974.

[3] J. A. Larson. A short proof of a partition theorem for the ordinal ω^ω. *Annals of Mathematical Logic*, 6(2):129–145, Dec. 1973.