Expressiveness of Deep Learning

Alexander Bentkamp

October 11, 2017

Abstract

Deep learning has had a profound impact on computer science in recent years, with applications to search engines, image recognition and language processing, bioinformatics, and more. Recently, Cohen et al. [2] provided theoretical evidence for the superiority of deep learning over shallow learning. For my master's thesis [1], I formalized their mathematical proof using Isabelle/HOL. This formalization simplifies and generalizes the original proof, while working around the limitations of the Isabelle type system. To support the formalization, I developed reusable libraries of formalized mathematics, including results about the matrix rank, the Lebesgue measure, and multivariate polynomials, as well as a library for tensor analysis.

Contents

1 Tensor

theory *Tensor*
imports *Main*
begin

typedef *'a tensor* = {*t*::*nat list* × *'a list*. *length* (*snd t*) = *prod-list* (*fst t*)}
by (*simp add*: *Ex-list-of-length*)

definition *dims*::*'a tensor* ⇒ *nat list* **where**
dims A = *fst* (*Rep-tensor A*)

definition *vec*::*'a tensor* ⇒ *'a list* **where**
vec A = *snd* (*Rep-tensor A*)

definition *tensor-from-vec*::*nat list* ⇒ *'a list* ⇒ *'a tensor* **where**
tensor-from-vec d v = *Abs-tensor* (*d*,*v*)

lemma
assumes *length v* = *prod-list d*
shows *dims-tensor*[*simp*]: *dims* (*tensor-from-vec d v*) = *d*
and *vec-tensor*[*simp*]: *vec* (*tensor-from-vec d v*) = *v*
by (*simp add*: *Abs-tensor-inverse assms dims-def tensor-from-vec-def vec-def*)+

lemma *tensor-from-vec-simp*[*simp*]: *tensor-from-vec* (*dims A*) (*vec A*) = *A*
by (*simp add*: *Rep-tensor-inverse Tensor.vec-def dims-def tensor-from-vec-def*)

lemma *length-vec*: *length* (*vec A*) = *prod-list* (*dims A*)
by (*metis* (*mono-tags*, *lifting*) *Rep-tensor Tensor.vec-def dims-def mem-Collect-eq*)

lemma *tensor-eqI*[*intro*]:
assumes *dims A* = *dims B* **and** *vec A* = *vec B*
shows *A*=*B*
by (*metis assms tensor-from-vec-simp*)

abbreviation *order*::*'a tensor* ⇒ *nat* **where**
order t == *length* (*dims t*)

inductive *valid-index*::*nat list* ⇒ *nat list* ⇒ *bool* (**infix** ◁ *50*) **where**
Nil: [] ◁ [] |
Cons: *is* ◁ *ds* ⟹ *i*<*d* ⟹ *i*#*is* ◁ *d*#*ds*

inductive-cases *valid-indexE*[*elim*]: *is* ◁ *ds*
inductive-cases *valid-index-dimsE*[*elim*]: *is* ◁ *dims A*

lemma *valid-index-length*: *is* ◁ *ds* ⟹ *length is* = *length ds*
by (*induction rule*:*valid-index.induct*; *auto*)

lemma *valid-index-lt*: *is* ◁ *ds* ⟹ *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
proof (*induction arbitrary*:*m rule*:*valid-index.induct*)
case *Nil*
then show *?case* **by** *auto*
next
case *Cons*
then show *?case* **by** (*metis gr0-conv-Suc length-Cons linorder-neqE-nat not-less-eq nth-Cons' nth-Cons-Suc*)
qed

lemma *valid-indexI*:
assumes *length is* = *length ds* **and** ⋀*m*. *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*

shows *is* ◁ *ds*
using *assms* **proof** (*induction is arbitrary*:*ds*)
case *Nil*
then show *?case* **by** (*metis length-0-conv valid-index.simps*)
next
case (*Cons a is ds*)
then obtain *d ds'* **where** *ds* = *d* # *ds'* **by** (*metis length-Suc-conv*)
then have *is* ◁ *ds'* **using** *Cons* **by** (*metis length-Cons less-irrefl linorder-neqE-nat not-less-eq nth-Cons-Suc*)
then show *?case* **using** *Cons.prems*(*2*) ‹*ds* = *d* # *ds'*› *valid-index.Cons* **by** *fastforce*
qed

lemma *valid-index-append*:
assumes *is1-valid*:*is1* ◁ *ds1* **and** *is2-valid*:*is2* ◁ *ds2*
shows *is1* @ *is2* ◁ *ds1* @ *ds2*
apply (*rule valid-indexI*[*of is1* @ *is2 ds1* @ *ds2*])
unfolding *nth-append*
using *valid-index-lt*[*OF is2-valid*] *valid-index-lt*[*OF is1-valid*] *valid-index-length*[*OF is1-valid*] *valid-index-length*[*OF is2-valid*] *length-append*
by (*auto simp add*: ‹*length is1* = *length ds1*›)

lemma *valid-index-list-all2-iff*: *is* ◁ *ds* ⟷ *list-all2* (*op* <) *is ds*
by (*metis list-all2-conv-all-nth list-all2-nthD valid-indexI valid-index-length valid-index-lt*)

definition *fixed-length-sublist*::'*a list* ⇒ *nat* ⇒ *nat* ⇒ '*a list* **where**
fixed-length-sublist xs l i = (*take l* (*drop* (*l*∗*i*) *xs*))

fun *lookup-base*::*nat list* ⇒ '*a list* ⇒ *nat list* ⇒ '*a* **where**
lookup-base-Nil: *lookup-base* [] *v* [] = *hd v* |
lookup-base-Cons: *lookup-base* (*d* # *ds*) *v* (*i* # *is*) =
lookup-base ds (*fixed-length-sublist v* (*prod-list ds*) *i*) *is*

definition *lookup*::'*a tensor* ⇒ *nat list* ⇒ '*a* **where**
lookup A = *lookup-base* (*dims A*) (*vec A*)

fun *tensor-vec-from-lookup*::*nat list* ⇒ (*nat list* ⇒ '*a*) ⇒ '*a list* **where**
tensor-vec-from-lookup-Nil: *tensor-vec-from-lookup* [] *e* = [*e* []] |
tensor-vec-from-lookup-Cons: *tensor-vec-from-lookup* (*d* # *ds*) *e* = *concat* (*map* (λ*i*. *tensor-vec-from-lookup ds* (λ*is*. *e* (*i* # *is*))) [*0*..<*d*])

definition *tensor-from-lookup*::*nat list* ⇒ (*nat list* ⇒ '*a*) ⇒ '*a tensor* **where**
tensor-from-lookup ds e = *tensor-from-vec ds* (*tensor-vec-from-lookup ds e*)

lemma *concat-parts-leq*:
assumes *a* ∗ *d* ≤ *length v*
shows *concat* (*map* (*fixed-length-sublist v d*) [*0*..<*a*]) = *take* (*a*∗*d*) *v*
using *assms* **proof** (*induction a*)
case *0*

then show *?case* **by** *simp*
next
case (*Suc a*)
then have *concat* (*map* (*fixed-length-sublist v d*) [*0..<a*]) = *take* (*a* ∗ *d*) *v* **by** *auto*
then have *concat* (*map* (*fixed-length-sublist v d*) [*0..<Suc a*]) =
take (*a* ∗ *d*) *v* @ *fixed-length-sublist v d a* **using** *fixed-length-sublist-def* **by** *auto*
then show *?case* **using** *Suc* **by** (*metis add.commute mult.commute mult-Suc take-add fixed-length-sublist-def*)
qed

lemma *concat-parts-eq*:
assumes *a* ∗ *d* = *length v*
shows *concat* (*map* (*fixed-length-sublist v d*) [*0..<a*]) = *v*
by (*simp add*: *concat-parts-leq assms*)

lemma *tensor-lookup-base*:
assumes *length v* = *prod-list ds*
and ⋀*is*. *is* ◁ *ds* ⟹ *lookup-base ds v is* = *e is*
shows *tensor-vec-from-lookup ds e* = *v*
using *assms* **proof** (*induction ds arbitrary*:*v e*)
case *Nil*
then show *?case* **unfolding** *tensor-vec-from-lookup.simps*
by (*metis One-nat-def Tensor.lookup-base-Nil length-0-conv length-Suc-conv list.sel*(*1*) *prod-list.Nil valid-index.Nil*)
next
case (*Cons a ds*)
then have *a* ∗ *prod-list ds* = *length v* **by** *auto*
{
fix *i* **assume** *i*<*a*
then have *prod-list ds* ∗ (*i*+*1*) ≤ *length v* **using** ‹*a* ∗ *prod-list ds* = *length v*›
using *discrete mult.commute mult-le-mono1* **by** *metis*
have ⋀*is′*. *is′* ◁ *ds* ⟹ *e* (*i* # *is′*) = *lookup-base ds* (*fixed-length-sublist v* (*prod-list ds*) *i*) *is′*
using ‹*i*<*a*› **by** (*metis Cons.prems*(*2*) *Tensor.lookup-base-Cons valid-index.simps*)
then have *tensor-vec-from-lookup ds* (λ*is′*. *e* (*i* # *is′*)) = *fixed-length-sublist v* (*prod-list ds*) *i*
using *Cons* **using** ‹*prod-list ds* ∗ (*i* + *1*) ≤ *length v*› **by** (*simp add*: *Cons.IH fixed-length-sublist-def*)
}
then show *?case* **unfolding** *tensor-vec-from-lookup-Cons lookup-base-Cons*
using *concat-parts-eq*[*OF* ‹*a* ∗ *prod-list ds* = *length v*›]
atLeastLessThan-iff map-eq-conv set-upt Cons **by** (*metis* (*no-types*, *lifting*))
qed

lemma *tensor-lookup*:
assumes ⋀*is*. *is* ◁ *dims A* ⟹ *lookup A is* = *e is*
shows *tensor-from-lookup* (*dims A*) *e* = *A*

using *tensor-lookup-base lookup-def length-vec tensor-from-lookup-def* **by** (*metis assms tensor-from-vec-simp*)

lemma *concat-equal-length*:
assumes $\bigwedge xs.\ xs \in set\ xss \Longrightarrow length\ xs = l$
shows *length* (*concat xss*) = *length xss*∗*l*
using *assms* **by** (*induction xss*;*auto*)

lemma *concat-equal-length-map*:
assumes $\bigwedge i.\ i < a \Longrightarrow length\ (f\ i) = d$
shows *length* (*concat* (*map* (λ*i*. *f i*) [*0*..<*a*])) = *a*∗*d*
using *assms* **by** (*induction a*;*auto*)

lemma *concat-parts*:
assumes $\bigwedge xs.\ xs \in set\ xss \Longrightarrow length\ xs = d$ **and** $i < length\ xss$
shows *fixed-length-sublist* (*concat xss*) *d i* = *xss* ! *i*
using *assms* **proof** (*induction xss arbitrary*:*i*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons xs xss*)
then have *length* (*concat xss*) = *length xss* ∗ *d* **by** (*simp add*: *Cons.prems*(*1*) *concat-equal-length*)
show *?case*
proof (*cases i*)
case *0*
then have *fixed-length-sublist* (*concat* (*xs* # *xss*)) *d i* = *xs*
unfolding *fixed-length-sublist-def* **by** (*simp add*: *Cons.prems*(*1*))
then show *?thesis* **using** *0* **by** *auto*
next
case (*Suc i′*)
then have *fixed-length-sublist* (*concat xss*) *d i′* = *xss* ! *i′* **using** *Cons* **by** *auto*
then show *?thesis* **unfolding** *fixed-length-sublist-def* **using** *Suc Cons.prems*(*1*) **by** *auto*
qed
qed

lemma *concat-parts′*:
assumes $\bigwedge i.\ i < a \Longrightarrow length\ (f\ i) = d$
and $i < a$
shows *fixed-length-sublist* (*concat* (*map* (λ*i*. *f i*) [*0*..<*a*])) *d i* = *f i*
using *assms* **proof** (*induction a*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc a*)
then have $(\bigwedge i.\ i < a \Longrightarrow length\ (f\ i) = d)$ **by** *auto*
then have *length* (*concat* (*map f* [*0*..<*a*])) = *a*∗*d* **using** *concat-equal-length-map* **by** *auto*

```
  show ?case
  proof (cases i=a)
    assume i=a
    then have fixed-length-sublist (concat (map f [0..<Suc a])) d i = f a
        by (simp add: Suc.prems(1) ‹length (concat (map f [0..<a])) = a * d›
fixed-length-sublist-def)
    then show ?case using ‹i=a› by auto
  next
    assume i≠a
    then have fixed-length-sublist (concat (map f [0..<a])) d i = f i
      concat (map f [0..<Suc a]) = concat (map f [0..<a]) @ f a using Suc by
auto
    show ?case unfolding ‹concat (map f [0..<Suc a]) = concat (map f [0..<a])
@ f a›
      unfolding fixed-length-sublist-def drop-append
     using ‹length (concat (map f [0..<a])) = a * d› ‹fixed-length-sublist (concat
(map f [0..<a])) d i = f i›
    using append-assoc append-eq-conv-conj append-take-drop-id assms(1) assms(2)
fixed-length-sublist-def
      by metis
  qed
qed

lemma length-tensor-vec-from-lookup:
length (tensor-vec-from-lookup ds e) = prod-list ds
by (induction ds arbitrary:e; auto simp add: concat-equal-length-map)

lemma lookup-tensor-vec:
assumes is◁ds
shows lookup-base ds (tensor-vec-from-lookup ds e) is = e is
using assms proof (induction arbitrary:e rule:valid-index.induct)
  case Nil
  then show ?case by simp
next
  case (Cons is ds i d e)
  then show ?case unfolding tensor-vec-from-lookup-Cons lookup-base-Cons
   by (simp add: length-tensor-vec-from-lookup concat-parts'[of d λi. tensor-vec-from-lookup
ds (λis. e (i # is)) prod-list ds i] ‹i < d›)
qed

lemma lookup-tensor-from-lookup:
assumes is◁ds
shows lookup (tensor-from-lookup ds e) is = e is
  unfolding lookup-def tensor-from-lookup-def
  by (simp add: lookup-tensor-vec assms length-tensor-vec-from-lookup)

lemma dims-tensor-from-lookup: dims (tensor-from-lookup ds e) = ds
  unfolding tensor-from-lookup-def
  by (simp add: length-tensor-vec-from-lookup)
```

lemma *tensor-lookup-cong*:
assumes *tensor-from-lookup ds* e_1 = *tensor-from-lookup ds* e_2
and *is*◁*ds*
shows e_1 *is* = e_2 *is* **using** *assms lookup-tensor-from-lookup* **by** *metis*

lemma *tensor-from-lookup-eqI*:
assumes ⋀*is*. *is*◁*ds* ⟹ e_1 *is* = e_2 *is*
shows *tensor-from-lookup ds* e_1 = *tensor-from-lookup ds* e_2
by (*metis assms lookup-tensor-vec length-tensor-vec-from-lookup tensor-lookup-base tensor-from-lookup-def*)

lemma *tensor-lookup-eqI*:
assumes *dims A* = *dims B* **and** ⋀*is*. *is*◁(*dims A*) ⟹ *lookup A is* = *lookup B is*
shows *A* = *B* **by** (*metis assms*(*1*) *assms*(*2*) *tensor-lookup*)

end

2 Subtensors

theory *Tensor-Subtensor*
imports *Tensor*
begin

definition *subtensor*::*'a tensor* ⇒ *nat* ⇒ *'a tensor* **where**
subtensor A i = *tensor-from-vec* (*tl* (*dims A*)) (*fixed-length-sublist* (*vec A*) (*prod-list* (*tl* (*dims A*))) *i*)

definition *subtensor-combine*::*nat list* ⇒ *'a tensor list* ⇒ *'a tensor* **where**
subtensor-combine ds As = *tensor-from-vec* (*length As* # *ds*) (*concat* (*map vec As*))

lemma *length-fixed-length-sublist*[*simp*]:
assumes (*Suc i*)∗*l* ≤ *length xs*
shows *length* (*fixed-length-sublist xs l i*) = *l*
unfolding *fixed-length-sublist-def*
by (*metis assms diff-add-inverse2 length-drop length-take min.absorb2 mult.commute mult-Suc take-drop*)

lemma *vec-subtensor*[*simp*]:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *vec* (*subtensor A i*) = *fixed-length-sublist* (*vec A*) (*prod-list* (*tl* (*dims A*))) *i*
by (*metis* (*no-types*, *lifting*) *Suc-leI assms*(*1*) *assms*(*2*) *hd-Cons-tl length-fixed-length-sublist length-vec prod-list.Cons mult-le-mono1 subtensor-def vec-tensor*)

lemma *dims-subtensor*[*simp*]:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *dims* (*subtensor A i*) = *tl* (*dims A*)
using *Suc-leI assms*(*1*) *assms*(*2*) *dims-tensor length-fixed-length-sublist length-vec*

list.collapse prod-list.Cons mult-le-mono1 subtensor-def
by *metis*

lemma *subtensor-combine-subtensor*[*simp*]:
assumes *dims A* ≠ []
shows *subtensor-combine* (*tl* (*dims A*)) (*map* (*subtensor A*) [*0*..<*hd* (*dims A*)]) = *A*
proof −
have *length-vec-A*: *hd* (*dims A*) ∗ *prod-list* (*tl* (*dims A*)) = *length* (*Tensor.vec A*)
by (*metis assms length-vec list.collapse prod-list.Cons*)
let *?subtensor-vec* = *fixed-length-sublist* (*vec A*) (*prod-list* (*tl* (*dims A*)))
{
fix *i* **assume** *i* < *hd* (*dims A*)
then have (*Suc i*)∗(*prod-list* (*tl* (*dims A*))) ≤ *length* (*vec A*)
by (*metis Suc-leI length-vec-A mult-le-mono1*)
then have (*vec* ∘ (λ*i*. *tensor-from-vec* (*tl* (*dims A*)) (*?subtensor-vec i*))) *i* = *?subtensor-vec i*
by *simp*
}
then have *1*:*map* (*Tensor.vec* ∘ (λ*i*. *tensor-from-vec* (*tl* (*dims A*)) (*?subtensor-vec i*))) [*0*..<*hd* (*dims A*)]
= *map ?subtensor-vec* [*0*..<*hd* (*dims A*)] **by** *auto*
then have *subtensor-combine* (*tl* (*dims A*)) (*map* (λ*i*. *subtensor A i*) [*0*..<*hd* (*dims A*)]) = *A*
unfolding *subtensor-combine-def subtensor-def* **using** *concat-parts-eq*[*OF length-vec-A*]
by (*auto simp add*: *1 assms*)
then show *?thesis* **by** *auto*
qed

lemma
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *subtensor-combine-dims*[*simp*]: *dims* (*subtensor-combine ds As*) = *length As* # *ds* (**is** *?D*)
and *subtensor-combine-vec*[*simp*]: *vec* (*subtensor-combine ds As*) = *concat* (*map vec As*) (**is** *?V*)
proof −
have ⋀*v*. *v*∈*set* (*map Tensor.vec As*) ⟹ *length v* = *prod-list ds* **using** *assms length-vec* **by** *fastforce*
then have *length As* ∗ *prod-list ds* = *length* (*concat* (*map Tensor.vec As*)) **using** *concat-equal-length*
by (*metis length-map*)
then show *?D ?V* **unfolding** *subtensor-combine-def* **by** *simp*+
qed

lemma *subtensor-subtensor-combine*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds* **and** *i* < *length As*
shows *subtensor* (*subtensor-combine ds As*) *i* = *As* ! *i*
proof −

have *fixed-length-sublist* (*concat* (*map vec As*)) (*prod-list ds*) *i* = *vec* (*As* ! *i*)
using *concat-parts*[*of map vec As prod-list ds i*] *assms imageE length-map length-vec*
nth-map set-map in-set-conv-nth **by** *fastforce*
then show *?thesis*
unfolding *subtensor-def* **using** *subtensor-combine-dims subtensor-combine-vec*
by (*metis assms list.sel*(*3*) *nth-mem tensor-from-vec-simp*)
qed

lemma *subtensor-induct*[*case-names order-0 order-step*]:
assumes *order-0*: ⋀*A*. *dims A* = [] ⟹ *P A*
and *order-step*: ⋀*A*. *dims A* ≠ [] ⟹ (⋀*i*. *i* < *hd* (*dims A*) ⟹ *P* (*subtensor A i*)) ⟹ *P A*
shows *P B*
using *assms* **proof** (*induction dims B arbitrary*:*B*)
case *Nil*
then show *?case* **by** *auto*
next
case *Cons*
then show *?case* **by** (*metis dims-subtensor list.sel*(*3*))
qed

lemma *subtensor-combine-induct*[*case-names order-0 order-step*]:
assumes *order-0*:⋀*A*. *dims A* = [] ⟹ *P A*
and *order-step*:⋀*As ds*. (⋀*A*. *A*∈*set As* ⟹ *P A*) ⟹ (⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*) ⟹ *P* (*subtensor-combine ds As*)
shows *P A*
proof (*induction A rule*:*subtensor-induct*)
case (*order-0 A*)
then show *?case* **by** (*simp add*: *assms*(*1*))
next
case (*order-step A*)
have *P* (*subtensor-combine* (*tl* (*dims A*)) (*map* (*subtensor A*) [*0*..<*hd* (*dims A*)]))
apply (*rule assms*(*2*))
using *atLeastLessThan-iff dims-subtensor imageE set-map set-upt order-step* **by** *auto*
then show *?case* **using** *subtensor-combine-subtensor*[*OF order-step.hyps*] **by** *metis*
qed

lemma *lookup-subtensor1*[*simp*]:
assumes *i* # *is* ◁ *dims A*
shows *lookup* (*subtensor A i*) *is* = *lookup A* (*i* # *is*)
using *assms*
proof (*induction A rule*: *subtensor-combine-induct*)
case *order-0*
then show *?case* **by** *auto*
next

case (*order-step As ds*)
have *0*:*subtensor* (*subtensor-combine ds As*) *i* = *As* ! *i*
by (*metis list.discI list.sel*(*1*) *order-step.hyps order-step.prems subtensor-combine-dims subtensor-subtensor-combine valid-index-dimsE*)
have *1*:*dims* (*subtensor-combine ds As*) = *length As* # *ds*
using *order-step subtensor-combine-def subtensor-combine-dims* **by** *force*
show *?case* **unfolding** *0 lookup-def 1* **unfolding** *lookup-base-Cons* **using** *order-step.prems*
using *Tensor.lookup-base-Cons dims-subtensor lookup-def list.discI list.sel*(*1*) *list.sel*(*3*) *valid-index-dimsE vec-subtensor* **by** (*metis 0 1*)
qed

lemma *lookup-subtensor*:
assumes *is* ◁ *dims A*
shows *lookup A is* = *hd* (*vec* (*fold* (λ*i A*. *subtensor A i*) *is A*))
using *assms* **proof** (*induction is arbitrary*: *A*)
case *Nil*
then show *?case* **by** (*metis Tensor.lookup-base-Nil lookup-def fold-simps*(*1*) *length-0-conv valid-index-length*)
next
case (*Cons a is A*)
then show *?case*
using *dims-subtensor lookup-subtensor1 fold-simps*(*2*) *list.discI list.sel*(*1*) *list.sel*(*3*) *valid-indexE* **by** (*metis* (*no-types*, *lifting*))
qed

lemma *subtensor-eqI*:
assumes *dims A* ≠ []
and *dims-eq*:*dims A* = *dims B*
and ⋀*i*. *i* < *hd* (*dims A*) ⟹ *subtensor A i* = *subtensor B i*
shows *A*=*B*
proof −
{
fix *is* **assume** *is* ◁ *dims A*
then obtain *i is*′ **where** *is-Cons*:*is* = *i* # *is*′ **using** *assms*(*1*) **by** *blast*
then have *lookup A is* = *lookup B is*
using *lookup-subtensor1 assms* **by** (*metis* ‹*is* ◁ *dims A*› *is-Cons list.sel*(*1*) *valid-index-dimsE*)
}
then show *?thesis* **using** *tensor-lookup-eqI*[*OF dims-eq*] **by** *auto*
qed

end

3 Tensor Addition

theory *Tensor-Plus*
imports *Tensor HOL.Option Tensor-Subtensor*
begin

definition *vec-plus a b = map* (λ(*x*,*y*). *plus x y*) (*zip a b*)

definition *plus-base*::′*a*::*semigroup-add tensor* ⇒ ′*a tensor* ⇒ ′*a tensor*
where *plus-base A B* = (*tensor-from-vec* (*dims A*) (*vec-plus* (*vec A*) (*vec B*)))

instantiation *tensor*:: (*semigroup-add*) *plus*
begin
definition *plus-def*: *A* + *B* = (*if* (*dims A* = *dims B*)
then plus-base A B
else undefined)
instance ..
end

lemma *plus-dim1*[*simp*]: *dims A* = *dims B* ⟹ *dims* (*A* + *B*) = *dims A* **unfolding** *plus-def plus-base-def*
using *dims-tensor length-vec length-map map-fst-zip vec-plus-def* **by** (*metis* (*full-types*))
lemma *plus-dim2*[*simp*]: *dims A* = *dims B* ⟹ *dims* (*A* + *B*) = *dims B* **using** *plus-dim1* **by** *metis*
lemma *plus-base*: *dims A* = *dims B* ⟹ *A* + *B* = *plus-base A B* **unfolding** *plus-def* **by** *metis*

lemma *fixed-length-sublist-plus*:
assumes *length xs1* = *c* ∗ *l length xs2* = *c* ∗ *l i* < *c*
shows *fixed-length-sublist* (*vec-plus xs1 xs2*) *l i*
= *vec-plus* (*fixed-length-sublist xs1 l i*) (*fixed-length-sublist xs2 l i*)
unfolding *vec-plus-def fixed-length-sublist-def* **using** *drop-map drop-zip take-map take-zip* **by** *metis*

lemma *vec-plus*[*simp*]:
assumes *dims A* = *dims B*
shows *vec* (*A*+*B*) = *vec-plus* (*vec A*) (*vec B*)
unfolding *plus-def plus-base-def vec-plus-def* **using** *assms*
by (*auto*; *metis* (*no-types*, *lifting*) *length-map length-tensor-vec-from-lookup map-fst-zip tensor-lookup tensor-from-lookup-def vec-tensor*)

lemma *subtensor-plus*:
fixes *A*::′*a*::*semigroup-add tensor* **and** *B*::′*a*::*semigroup-add tensor*
assumes *i* < *hd* (*dims A*)
and *dims A* = *dims B*
and *dims A* ≠ []
shows *subtensor* (*A* + *B*) *i* = *subtensor A i* + *subtensor B i*
proof −
have *length* (*vec A*) = *hd* (*dims A*) ∗ *prod-list* (*tl* (*dims A*))
length (*Tensor.vec B*) = *hd* (*dims A*) ∗ *prod-list* (*tl* (*dims A*))
using *length-vec prod-list.Cons assms* **by** (*metis* (*no-types*) *list.exhaust-sel*)+
then show *?thesis*
using *Tensor-Plus.vec-plus assms fixed-length-sublist-plus vec-subtensor tensor-eqI*

dims-subtensor plus-dim1 **by** *fastforce*
qed

lemma *lookup-plus*[*simp*]:
assumes *dims A* = *dims B*
and *is* ◁ *dims A*
shows *lookup* (*A* + *B*) *is* = *lookup A is* + *lookup B is*
using *assms* **proof** (*induction A*+*B arbitrary*:*A B is rule*: *subtensor-induct*)
case (*order-0 A B is*)
then have *is* = [] **by** *auto*
have *1*:[] ◁ *dims A* **using** *order-0* ‹*is* = []› **by** *auto*
have *2*:[] ◁ *dims B* **using** *order-0* ‹*is* = []› **by** *auto*
have *3*:[] ◁ *dims* (*A* + *B*) **using** *order-0* ‹*is* = []› **by** *auto*
have *length* (*vec A*) = *1 length* (*vec B*) = *1*
by (*metis length-vec prod-list.Nil order-0.hyps order-0.prems*(*1*) *plus-dim1*)+
then show *?case* **unfolding** *lookup-subtensor*[*OF 1*] *lookup-subtensor*[*OF 2*] *lookup-subtensor*[*OF 3*] ‹*is* = []›
fold-simps(*1*) *vec-plus*[*OF order-0.prems*(*1*)] **unfolding** *vec-plus-def* **using** *order-0.prems length-map*
list.map-sel(*1*) *list.size*(*3*) *map-fst-zip map-snd-zip order-0.hyps*
zero-neq-one case-prod-unfold length-vec **by** *metis*
next
case (*order-step A B is*)
then obtain *i is*′ **where** *is* = *i* # *is*′ **by** *auto*
have *1*:*is* ◁ *dims A* **using** *order-step* **by** *auto*
have *2*:*is* ◁ *dims B* **using** *order-step* **by** *auto*
have *3*:*is* ◁ *dims* (*A* + *B*) **using** *order-step* **by** *auto*
have *lookup* (*subtensor A i* + *subtensor B i*) *is*′ = *lookup* (*subtensor A i*) *is*′ + *lookup* (*subtensor B i*) *is*′
apply (*rule order-step.hyps*(*2*)[*of i*])
using ‹*is* = *i* # *is*′› *3 hd-conv-nth length-greater-0-conv nth-Cons-0 order-step.hyps*(*1*) *valid-index-lt*
apply *auto*[*1*]
apply (*metis 2* ‹*is* = *i* # *is*′› *list.inject list.sel*(*1*) *list.simps*(*3*) *order-step.prems*(*1*) *subtensor-plus valid-index.cases*)
using *1* ‹*is* = *i* # *is*′› *order-step.prems*(*1*) *plus-dim1* **apply** *auto*[*1*]
using *1* ‹*is* = *i* # *is*′› *plus-dim1* **by** *auto*
then show *?case* **using** *lookup-subtensor*[*OF 1*] *lookup-subtensor*[*OF 2*] *lookup-subtensor*[*OF 3*]
using *order-step* ‹*is* = *i* # *is*′› *plus-dim1 lookup-subtensor1 list.sel*(*1*) *subtensor-plus valid-index-dimsE* **by** *metis*
qed

lemma *plus-assoc*:
assumes *dimsA*:*dims A* = *ds* **and** *dimsB*:*dims B* = *ds* **and** *dimsC*:*dims C* = *ds*
shows (*A* + *B*) + *C* = *A* + (*B* + *C*)
by (*rule tensor-lookup-eqI*; *simp add*: *dimsA dimsB dimsC add.assoc*)+

lemma *tensor-comm*[*simp*]:

fixes $A::'a::ab\text{-}semigroup\text{-}add\ tensor$
shows $A + B = B + A$
proof $(cases\ dims\ A = dims\ B)$
case *True*
then show *?thesis* **unfolding** *plus-def plus-base-def*
using *add.commute lookup-plus*[*OF True*] *plus-dim1*[*OF True*] *tensor-lookup-eqI*[*OF True*] *vec-plus*[*OF True*]
by (*metis lookup-plus plus-dim1 tensor-lookup-eqI vec-plus*)
next
case *False*
then show *?thesis* **unfolding** *plus-def plus-base-def* **by** *simp*
qed

definition $vec0\ n = replicate\ n\ 0$

definition $tensor0::nat\ list \Rightarrow 'a::zero\ tensor$ **where**
$tensor0\ d = tensor\text{-}from\text{-}vec\ d\ (vec0\ (prod\text{-}list\ d))$

lemma *dims-tensor0*[*simp*]: $dims\ (tensor0\ d) = d$
and *vec-tensor0*[*simp*]: $vec\ (tensor0\ d) = vec0\ (prod\text{-}list\ d)$
unfolding *tensor0-def vec0-def* **by** *simp-all*

lemma *lookup-is-in-vec*: $is \lhd (dims\ A) \Longrightarrow lookup\ A\ is \in set\ (vec\ A)$
proof (*induction arbitrary*:*is rule*:*subtensor-induct*)
case *order-0*
then show *?case* **unfolding** *lookup-def* **using** *lookup-base-Nil*
by (*metis length-0-conv length-vec list.set-sel*(*1*) *prod-list.Nil valid-index-length zero-neq-one*)
next
case (*order-step A is*)
then obtain *i is'* **where** $is = i\ \#\ is'$ **using** *valid-index-dimsE* **by** *blast*
then have *1*:$i < hd\ (dims\ A)$ **using** *dims-def order-step.prems* **by** *auto*
have *2*:$is' \lhd dims\ (subtensor\ A\ i)$ **using** $\langle is = i\ \#\ is'\rangle$ *dims-subtensor order-step.prems*
by *auto*
have $lookup\ A\ is \in set\ (Tensor.vec\ (subtensor\ A\ i))$
using *order-step.IH* [*OF 1 2*] *lookup-subtensor1* $\langle is = i\ \#\ is'\rangle$ *order-step.prems*
by *auto*
then show *?case* **using** *vec-subtensor fixed-length-sublist-def* **by** (*metis 1 in-set-dropD in-set-takeD order-step.hyps*)
qed

lemma *lookup-tensor0*:
assumes $is \lhd ds$
shows $lookup\ (tensor0\ ds)\ is = 0$
proof −
have $lookup\ (tensor0\ ds)\ is \in set\ (vec\ (tensor0\ ds))$ **using** *lookup-is-in-vec assms*
by (*metis dims-tensor0*)
moreover have $set\ (vec\ (tensor0\ ds)) \subseteq \{0\}$ **unfolding** *vec-tensor0 vec0-def*
by (*metis in-set-replicate singleton-iff subsetI*)

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

lemma
fixes *A*::*'a*::*monoid-add tensor*
shows *tensor-add-0-right*[*simp*]: *A* + *tensor0* (*dims A*) = *A*
unfolding *plus-def plus-base-def dims-tensor0*
apply (*simp-all*)
apply (*rule tensor-lookup-eqI*)
apply (*metis* (*no-types*, *lifting*) *dims-tensor dims-tensor0 length-vec plus-dim2 vec-plus vec-tensor0*)
by (*metis add.right-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0*)

lemma
fixes *A*::*'a*::*monoid-add tensor*
shows *tensor-add-0-left*[*simp*]: *tensor0* (*dims A*) + *A* = *A*
unfolding *plus-def plus-base-def dims-tensor0*
apply (*simp-all*)
apply (*rule tensor-lookup-eqI*)
apply (*metis* (*no-types*, *lifting*) *dims-tensor dims-tensor0 length-vec plus-dim2 vec-plus vec-tensor0*)
by (*metis add.left-neutral dims-tensor0 lookup-plus lookup-tensor0 plus-dim2 tensor-from-vec-simp vec-plus vec-tensor0*)

definition *listsum*::*nat list* ⇒ *'a*::*monoid-add tensor list* ⇒ *'a tensor* **where**
listsum ds As = *foldr* (*op* +) *As* (*tensor0 ds*)

definition *listsum'*::*'a*::*monoid-add tensor list* ⇒ *'a tensor* **where**
listsum' As = *listsum* (*dims* (*hd As*)) *As*

lemma *listsum-Nil*: *listsum ds* [] = *tensor0 ds* **by** (*simp add*: *Tensor-Plus.listsum-def*)

lemma *listsum-one*: *listsum* (*dims A*) [*A*] = *A* **unfolding** *listsum-def* **by** *simp*

lemma *listsum-Cons*: *listsum ds* (*A* # *As*) = *A* + *listsum ds As*
unfolding *listsum-def* **by** *auto*

lemma *listsum-dims*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *dims* (*listsum ds As*) = *ds*
using *assms* **proof** (*induction As*)
case *Nil*
then show *?case* **by** (*metis dims-tensor0 listsum-Nil*)
next
case (*Cons A As*)
then show *?case* **using** *listsum-Cons*
by (*metis list.set-intros*(*1*) *list.set-intros*(*2*) *plus-dim2*)

qed

lemma *subtensor0*:
assumes $ds \neq []$ **and** $i<hd\ ds$
shows *subtensor* (*tensor0 ds*) i = *tensor0* (*tl ds*)
proof (*rule tensor-lookup-eqI*)
show *1*:*dims* (*subtensor* (*tensor0 ds*) i) = *dims* (*tensor0* (*tl ds*)) **by** (*simp add*: *assms*(*1*) *assms*(*2*))
fix *is* **assume** *is* $\lhd$ *dims* (*subtensor* (*tensor0 ds*) i)
then have $i \# is \lhd$ *dims* (*tensor0 ds*) **using** *assms*(*1*) *assms*(*2*) *valid-index.Cons* **by** *fastforce*
then show *lookup* (*subtensor* (*tensor0 ds*) i) *is* = *lookup* (*tensor0* (*tl ds*)) *is*
using *lookup-subtensor1* *1* ‹*is* $\lhd$ *dims* (*subtensor* (*tensor0 ds*) i)› *dims-tensor0* *lookup-tensor0*
by *metis*
qed

lemma *subtensor-listsum*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
and $ds \neq []$ **and** $i<hd\ ds$
shows *subtensor* (*listsum ds As*) i = *listsum* (*tl ds*) (*map* ($\lambda A.$ *subtensor A i*) *As*)
using *assms* **proof** (*induction As*)
case *Nil*
then show *?case* **using** *lookup-tensor0 assms*(*2*) *assms*(*3*) *subtensor0* **by** (*auto simp add*: *listsum-Nil*)
next
case (*Cons A As*)
then show *?case* **by** (*simp add*: *listsum-Cons*; *metis subtensor-plus listsum-dims*)
qed

lemma *listsum0*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow A$ = *tensor0 ds*
shows *listsum ds As* = *tensor0 ds*
using *assms* **proof** (*induction As*)
case *Nil*
show *?case* **by** (*simp add*: *listsum-Nil*)
next
case *Cons*
then show *?case* **using** *listsum-Cons*
by (*metis dims-tensor0 list.set-intros*(*1*) *set-subset-Cons subsetCE tensor-add-0-right*)
qed

lemma *listsum-all-0-but-one*:
assumes $\bigwedge i.\ i \neq j \Longrightarrow i<length\ As \Longrightarrow As!i$ = *tensor0 ds*
and *dims* ($As!j$) = *ds*
and $j <$ *length As*
shows *listsum ds As* = $As!j$
using *assms* **proof** (*induction As arbitrary*:*j*)

case *Nil*
then show *?case* **by** *auto*
next
case (*Cons A As j*)
then show *?case*
proof (*cases j*)
case *0*
then have $\bigwedge i.\ i < length\ As \Longrightarrow As\ !\ i = tensor0\ ds$ **using** *Cons* **using** *Suc-less-eq length-Cons list.sel*(*3*) *nat.simps*(*3*) *nth-tl* **by** *fastforce*
then have *listsum ds As* = *tensor0 ds* **using** *listsum0* **by** (*metis in-set-conv-nth*)
then show *?thesis* **by** (*metis 0 Cons.prems*(*2*) *listsum-Cons nth-Cons-0 tensor-add-0-right*)
next
case (*Suc j'*)
then have *listsum ds As* = *As*!*j'* **by** (*metis* (*no-types*, *lifting*) *Cons.IH Cons.prems*(*1*) *Cons.prems*(*2*) *Cons.prems*(*3*) *Suc-less-eq length-Cons less-Suc-eq list.sel*(*3*) *not-less-eq nth-tl*)
then show *?thesis* **by** (*metis Cons.prems*(*1*) *Cons.prems*(*2*) *Suc length-greater-0-conv list.simps*(*3*) *listsum-Cons nat.simps*(*3*) *nth-Cons-0 nth-Cons-Suc tensor-add-0-left*)
qed
qed

lemma *lookup-listsum*:
assumes *is* ◁ *ds*
and $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows *lookup* (*listsum ds As*) *is* = ($\sum A \leftarrow As.\ lookup\ A\ is$)
using *assms* **proof** (*induction As*)
case *Nil*
then show *?case* **by** (*simp add*: *assms*(*1*) *listsum-Nil lookup-tensor0*)
next
case (*Cons A As*)
then show *?case* **by** (*simp add*: *listsum-Cons list.set-intros listsum-dims*)
qed

end

4 Tensor Scalar Multiplication

theory *Tensor-Scalar-Mult*
imports *Tensor-Plus Tensor-Subtensor*
begin

definition *vec-smult*::'*a*::*ring* ⇒ '*a list* ⇒ '*a list* **where**
vec-smult α β = *map* (*op* ∗ α) β

lemma *vec-smult0*: *vec-smult 0 as* = *vec0* (*length as*)
by (*induction as*; *auto simp add*:*vec0-def vec-smult-def*)

lemma *vec-smult-distr-right*:

shows *vec-smult* $(\alpha + \beta)$ *as* = *vec-plus* (*vec-smult* α *as*) (*vec-smult* β *as*)
unfolding *vec-smult-def vec-plus-def*
by (*induction as*; *simp add*: *distrib-right*)

lemma *vec-smult-Cons*:
shows *vec-smult* α (*a* # *as*) = (α ∗ *a*) # *vec-smult* α *as* **by** (*simp add*: *vec-smult-def*)

lemma *vec-plus-Cons*:
shows *vec-plus* (*a* # *as*) (*b* # *bs*) = (*a*+*b*) # *vec-plus as bs* **by** (*simp add*: *vec-plus-def*)

lemma *vec-smult-distr-left*:
assumes *length as* = *length bs*
shows *vec-smult* α (*vec-plus as bs*) = *vec-plus* (*vec-smult* α *as*) (*vec-smult* α *bs*)
using *assms* **proof** (*induction as arbitrary*:*bs*)
case *Nil*
then show *?case* **unfolding** *vec-smult-def vec-plus-def* **by** *simp*
next
case (*Cons a as'*)
then obtain *b bs'* **where** *bs* = *b* # *bs'* **by** (*metis Suc-length-conv*)
then have *0*:*vec-smult* α (*vec-plus* (*a* # *as'*) *bs*) = (α∗(*a*+*b*)) # *vec-smult* α (*vec-plus as' bs'*)
unfolding *vec-smult-def vec-plus-def* **using** *Cons.IH*[*of bs'*] **by** *simp*
have *length bs'* = *length as'* **using** *Cons.prems* ‹*bs* = *b* # *bs'*› **by** *auto*
then show *?case* **unfolding** *0* **unfolding** ‹*bs* = *b* # *bs'*› *vec-smult-Cons vec-plus-Cons*
by (*simp add*: *Cons.IH distrib-left*)
qed

lemma *length-vec-smult*: *length* (*vec-smult* α *v*) = *length v* **unfolding** *vec-smult-def* **by** *simp*

definition *smult*::'*a*::*ring* ⇒ '*a tensor* ⇒ '*a tensor* (**infixl** · *70*) **where**
smult α *A* = (*tensor-from-vec* (*dims A*) (*vec-smult* α (*vec A*)))

lemma *tensor-smult0*: **fixes** *A*::'*a*::*ring tensor*
shows *0* · *A* = *tensor0* (*dims A*)
unfolding *smult-def tensor0-def vec-smult-def* **using** *vec-smult0 length-vec*
by (*metis* (*no-types*) *vec-smult-def*)

lemma *dims-smult*[*simp*]:*dims* (α · *A*) = *dims A*
and *vec-smult*[*simp*]: *vec* (α · *A*) = *map* (*op* ∗ α) (*vec A*)
unfolding *smult-def vec-smult-def* **by** (*simp add*: *length-vec*)+

lemma *tensor-smult-distr-right*: (α + β) · *A* = α · *A* + β · *A*
unfolding *plus-def plus-base-def*
by (*auto*; *metis smult-def vec-smult-def vec-smult-distr-right*)

lemma *tensor-smult-distr-left*: *dims A = dims B ⟹ α · (A + B) = α · A + α · B*
proof −
assume *a1*: *dims A = dims B*
then have *f2*: *length* (*vec-plus* (*vec A*) (*vec B*)) = *length* (*vec A*)
by (*simp add*: *length-vec vec-plus-def*)
have *f3*: *dims* (*tensor-from-vec* (*dims B*) (*vec-smult α* (*vec A*))) = *dims B*
using *a1* **by** (*simp add*: *length-vec vec-smult-def*)
have *f4*: *vec* (*α · A*) = *vec-smult α* (*vec A*)
by (*simp add*: *vec-smult-def*)
have *length* (*vec-smult α* (*vec B*)) = *length* (*vec B*)
by (*simp add*: *vec-smult-def*)
then show *?thesis*
unfolding *plus-def plus-base-def* **using** *f4 f3 f2 a1*
by (*simp add*: *length-vec smult-def vec-smult-distr-left*)
qed

lemma *smult-fixed-length-sublist*:
assumes *length xs = l ∗ c i<c*
shows *fixed-length-sublist* (*vec-smult α xs*) *l i* = *vec-smult α* (*fixed-length-sublist xs l i*)
unfolding *fixed-length-sublist-def vec-smult-def* **by** (*simp add*: *drop-map take-map*)

lemma *smult-subtensor*:
assumes *dims A ≠ [] i < hd* (*dims A*)
shows *α · subtensor A i = subtensor* (*α · A*) *i*
proof (*rule tensor-eqI*)
show *dims* (*α · subtensor A i*) = *dims* (*subtensor* (*α · A*) *i*)
using *dims-smult dims-subtensor assms*(*1*) *assms*(*2*) **by** *simp*
show *vec* (*α · subtensor A i*) = *vec* (*subtensor* (*α · A*) *i*)
unfolding *vec-smult*
unfolding *vec-subtensor*[*OF* ‹*dims A ≠ []*› ‹*i < hd* (*dims A*)›]
using *vec-subtensor*[*of α · A i*]
by (*simp add*: *assms*(*1*) *assms*(*2*) *drop-map fixed-length-sublist-def take-map*)
qed

lemma *lookup-smult*:
assumes *is ◁ dims A*
shows *lookup* (*α · A*) *is* = *α ∗ lookup A is*
using *assms* **proof** (*induction A arbitrary*:*is rule*:*subtensor-induct*)
case (*order-0 A is*)
then have *length* (*vec A*) = *1* **by** (*simp add*: *length-vec*)
then have *hd* (*vec-smult α* (*vec A*)) = *α ∗ hd* (*vec A*) **unfolding** *vec-smult-def*
by (*metis list.map-sel*(*1*) *list.size*(*3*) *zero-neq-one*)
moreover have *is = []* **using** *order-0* **by** *auto*
ultimately show *?case* **unfolding** *smult-def* **by** (*auto simp add*: ‹*length* (*Tensor.vec A*) = *1*› *lookup-def length-vec-smult order-0.hyps*)
next
case (*order-step A is*)

then obtain *i is'* **where** *is = i # is'* **by** *blast*
then have *lookup* (*α · subtensor A i*) *is' = α * lookup* (*subtensor A i*) *is'*
by (*metis* (*no-types*, *lifting*) *dims-subtensor list.sel*(*1*) *list.sel*(*3*) *order-step.IH order-step.hyps order-step.prems valid-index-dimsE*)
then show *?case* **using** *smult-subtensor* ‹*is = i # is'*› *dims-smult lookup-subtensor1 list.sel*(*1*) *order-step.hyps order-step.prems valid-index-dimsE*
by *metis*
qed

lemma *tensor-smult-assoc*:
fixes *A*::*'a*::*ring tensor*
shows *α · (β · A) = (α * β) · A*
by (*rule tensor-lookup-eqI*, *simp*, *metis lookup-smult dims-smult mult.assoc*)

end

5 Tensor Product

theory *Tensor-Product*
imports *Tensor-Scalar-Mult Tensor-Subtensor*
begin

instantiation *tensor*:: (*ring*) *semigroup-mult*
begin
definition *tensor-prod-def*:*A * B = tensor-from-vec* (*dims A @ dims B*) (*concat* (*map* (*λa. vec-smult a* (*vec B*)) (*vec A*)))

abbreviation *tensor-prod-otimes* :: *'a tensor ⇒ 'a tensor ⇒ 'a tensor* (**infixl** *⊗ 70*)
where *A ⊗ B ≡ A * B*

lemma *vec-tensor-prod*[*simp*]: *vec* (*A ⊗ B*) = *concat* (*map* (*λa. vec-smult a* (*vec B*)) (*vec A*)) (**is** *?V*)
and *dims-tensor-prod*[*simp*]: *dims* (*A ⊗ B*) = *dims A @ dims B* (**is** *?D*)
proof −
have *length* (*concat* (*map* (*λa. vec-smult a* (*vec B*)) (*vec A*))) = *prod-list* (*dims A @ dims B*)
proof −
have *⋀xs. xs ∈ set* (*map* (*λa. vec-smult a* (*vec B*)) (*vec A*)) *⟹ length xs = length* (*vec B*)
using *length-vec-smult* **by** *force*
then show *?thesis* **using** *concat-equal-length* **by** (*metis length-map length-vec prod-list.append*)
qed
then show *?V ?D* **by** (*simp add*: *tensor-prod-def*)+
qed

lemma *tensorprod-subtensor-base*:

shows *concat (map f (concat xss)) = concat (map (λxs. concat (map f xs)) xss)*
by (*induction xss*; *auto*)

lemma *subtensor-combine-tensor-prod*:
assumes ⋀*A. A∈set As ⟹ dims A = ds*
shows *subtensor-combine ds As ⊗ B = subtensor-combine (ds @ dims B) (map (λA. A ⊗ B) As)*
proof −
let *?f = λa. vec-smult a (Tensor.vec B)*
let *?xss = map Tensor.vec As*
have *1*:*prod-list (length As # ds) = length (concat ?xss)* **by** (*metis assms length-vec subtensor-combine-dims subtensor-combine-vec*)
have *2*:⋀*A. A∈set As ⟹ prod-list (dims A @ dims B) = length (concat (map ?f (Tensor.vec A)))*
by (*metis dims-tensor-prod length-vec vec-tensor-prod*)
have *3*: *length As # ds @ dims B = (length (map (λA. tensor-from-vec (dims A @ dims B)*
(concat (map (λa. vec-smult a (vec B)) (vec A)))) As) # ds @ dims B) **by** *simp*
have *4*:(*concat (map (λxs. concat (map (λa. vec-smult a (vec B)) xs)) (map vec As))*)
= (concat (map vec (map (λA. tensor-from-vec (dims A @ dims B) (concat (map (λa. vec-smult a (vec B)) (vec A)))) As)))
unfolding *map-map*[*unfolded comp-def*] **using** *vec-tensor* **by** (*metis* (*no-types*, *lifting*) *2 map-eq-conv*)
have *subtensor-combine ds As ⊗ B = tensor-from-vec (length As # ds @ dims B) (concat (map ?f (concat (?xss))))*
unfolding *subtensor-combine-def tensor-prod-def* **using** *1* **by** *auto*
also have *... = tensor-from-vec (length As # ds @ dims B) (concat (map (λxs. concat (map ?f xs)) ?xss))*
using *tensorprod-subtensor-base*[*of ?f ?xss*] **by** *auto*
also have *... =subtensor-combine (ds @ dims B) (map (λA. A ⊗ B) As)*
unfolding *subtensor-combine-def tensor-prod-def* **using** *3 4* **by** *metis*
finally show *?thesis* **by** *metis*
qed

lemma *subtensor-tensor-prod*:
assumes *dims A ≠ []* **and** *i < hd (dims A)*
shows *subtensor (A ⊗ B) i = subtensor A i ⊗ B*
using *assms* **proof** (*induction A rule*:*subtensor-combine-induct*)
case *order-0*
then show *?case* **by** *auto*
next
case (*order-step As ds*)
have *1*:*i < length (map (λA. A ⊗ B) As)* **using** *order-step* **by** (*simp add*: *order-step.hyps order-step.prems*(*1*))
have *2*:(⋀*A. A ∈ set (map (λA. A ⊗ B) As) ⟹ dims A = ds @ dims B*) **using** *order-step* **by** *auto*

have *subtensor* (*subtensor-combine ds As* ⊗ *B*) *i* = *subtensor* (*subtensor-combine* (*ds* @ *dims B*) (*map* (λ*A*. *A* ⊗ *B*) *As*)) *i*
using *subtensor-combine-tensor-prod order-step* **by** *metis*
also have ... = *As* ! *i* ⊗ *B*
using *order-step subtensor-subtensor-combine*[*of* (*map* (λ*A*. *A* ⊗ *B*) *As*) *ds* @ *dims B i*] *1 2* **by** *auto*
also have ... = *subtensor* (*subtensor-combine ds As*) *i* ⊗ *B*
by (*metis 1 length-map order-step.hyps subtensor-subtensor-combine*)
finally show *?case* **by** *auto*
qed

lemma *lookup-tensor-prod*[*simp*]:
assumes *is1-valid*:*is1* ◁ *dims A* **and** *is2-valid*:*is2* ◁ *dims B*
shows *lookup* (*A* ⊗ *B*) (*is1* @ *is2*) = *lookup A is1* ∗ *lookup B is2*
using *assms* **proof** (*induction A arbitrary*:*is1 rule*:*subtensor-induct*)
case (*order-0 A is1*)
then obtain *a* **where** *vec A* = [*a*]
using *Suc-length-conv Tensor.tensor-vec-from-lookup-Nil length-0-conv length-tensor-vec-from-lookup length-vec* **by** *metis*
then have *A* ⊗ *B* = *a* · *B* **unfolding** *tensor-prod-def smult-def* **using** *order-0* **by** *simp*
moreover have *lookup A* [] = *a* **by** (*simp add*: ‹*Tensor.vec A* = [*a*]› *lookup-def order-0.hyps*)
ultimately have *lookup* (*A* ⊗ *B*) (*is2*) = *a* ∗ *lookup B is2* **by** (*simp add*: *lookup-smult is2-valid*)
then show *?case* **using** ‹*lookup A* [] = *a*› *null-rec*(*1*) *order-0.hyps order-0.prems*(*1*) **by** *auto*
next
case (*order-step A is1*)
then obtain *i is1* ′ **where** *i* # *is1* ′ = *is1* **by** *blast*
have *lookup* (*subtensor A i* ⊗ *B*) (*is1* ′ @ *is2*) = *lookup* (*subtensor A i*) *is1* ′ ∗ *lookup B is2* **using** *order-step*
by (*metis* ‹*i* # *is1* ′ = *is1*› *dims-subtensor list.sel*(*1*) *list.sel*(*3*) *valid-index-dimsE*)
then show *lookup* (*A* ⊗ *B*) (*is1* @ *is2*) = *lookup A is1* ∗ *lookup B is2*
using *lookup-subtensor1*[*of i is1* ′ *A*] *lookup-subtensor1*[*of i is1* ′ @ *is2 A*⊗*B*] *subtensor-tensor-prod*[*of A i B*]
Cons-eq-appendI ‹*i* # *is1* ′ = *is1*› *dims-tensor-prod is2-valid list.sel*(*1*) *order-step.hyps order-step.prems*(*1*) *valid-index-append valid-index-dimsE*
by *metis*
qed

lemma *valid-index-split*:
assumes *is* ◁ *ds1* @ *ds2*
obtains *is1 is2* **where** *is1* @ *is2* = *is is1* ◁ *ds1 is2* ◁ *ds2*
proof
assume *a*: ⋀*is1 is2*. *is1* @ *is2* = *is* ⟹ *is1* ◁ *ds1* ⟹ *is2* ◁ *ds2* ⟹ *thesis*
have *length-is*:*length is* = *length ds1* + *length ds2* **using** *valid-index-length* **using** *assms* **by** *auto*
show *take* (*length ds1*) *is* ◁ *ds1*

apply (*rule valid-indexI*)
using *valid-index-length* **using** *assms* **apply** *auto*[*1*]
by (*metis add-leD1 assms length-append not-less nth-append nth-take valid-index-lt*)
show *drop* (*length ds1*) *is* ◁ *ds2*
apply (*rule valid-indexI*)
using *valid-index-length* **using** *assms* **apply** *auto*[*1*]
using *nth-drop*[*of length ds1 - is*] *valid-index-lt*[*OF assms*(*1*)] *nth-append*[*of ds1 ds2*] *length-is*
by (*metis* (*no-types*) *add-strict-left-mono length-append less-imp-le nth-append-length-plus*)
show *take* (*length ds1*) *is* @ *drop* (*length ds1*) *is* = *is* **using** *length-is* **by** *auto*
qed

instance proof
fix *A B C*::′*a*::*ring tensor*
show (*A* ⊗ *B*) ⊗ *C* = *A* ⊗ (*B* ⊗ *C*)
proof (*rule tensor-lookup-eqI*, *simp*)
fix *is* **assume** *is* ◁ *dims* ((*A* ⊗ *B*) ⊗ *C*)
obtain *is1 is23* **where** *is1* ◁ *dims A is23* ◁ *dims* (*B* ⊗ *C*) *is1* @ *is23* = *is*
by (*metis* (*mono-tags*, *lifting*) ‹*is* ◁ *dims* ((*A* ⊗ *B*) ⊗ *C*)› *Tensor-Product.dims-tensor-prod append-assoc valid-index-split*)
obtain *is2 is3* **where** *is2* ◁ *dims B is3* ◁ *dims C is2* @ *is3* = *is23*
by (*metis* ‹*is23* ◁ *dims* (*local.tensor-prod-otimes B C*)› *dims-tensor-prod valid-index-split*)
def *is12* == *is1* @ *is2*
have *is12* ◁ *dims* (*A* ⊗ *B*) **by** (*simp add*: ‹*is1* ◁ *dims A*› ‹*is2* ◁ *dims B*› *is12-def valid-index-append*)
have *is12* @ *is3* = *is* **by** (*simp add*: ‹*is1* @ *is23* = *is*› ‹*is2* @ *is3* = *is23*› *is12-def*)
show *lookup* ((*A* ⊗ *B*) ⊗ *C*) *is* = *lookup* (*A* ⊗ (*B* ⊗ *C*)) *is*
unfolding *lookup-tensor-prod*[*OF* ‹*is1* ◁ *dims A*› ‹*is23* ◁ *dims* (*B* ⊗ *C*)›, *unfolded* ‹*is1* @ *is23* = *is*›]
lookup-tensor-prod[*OF* ‹*is12* ◁ *dims* (*A* ⊗ *B*)› ‹*is3* ◁ *dims C*›, *unfolded* ‹*is12* @ *is3* = *is*›]
using ‹*is1* ◁ *dims A*› ‹*is2* @ *is3* = *is23*› ‹*is2* ◁ *dims B*› ‹*is3* ◁ *dims C*› *is12-def mult.assoc* **by** *fastforce*
qed
qed

end

lemma *tensor-prod-distr-left*:
assumes *dims A* = *dims B*
shows (*A* + *B*) ⊗ *C* = (*A* ⊗ *C*) + (*B* ⊗ *C*)
proof −
have ⋀*is*. *is* ◁ *dims A* @ *dims C* ⟹ *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* (*A* ⊗ *C* + *B* ⊗ *C*) *is*
proof −
fix *is* **assume** *is* ◁ *dims A* @ *dims C*
obtain *is1 is2* **where** *is* = *is1* @ *is2 is1* ◁ *dims A is2* ◁ *dims C* **using**

valid-index-split **using** ‹*is* ◁ *dims A* @ *dims C*› **by** *blast*
then show *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* ((*A* ⊗ *C*) + (*B* ⊗ *C*)) *is*
using *lookup-plus*
‹*is1* ◁ *dims A*› ‹*is2* ◁ *dims C*› *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distribs*(*2*) *valid-index-append*
by *fastforce*
qed
moreover have *tensor-from-lookup* (*dims A* @ *dims C*) (*lookup* ((*A* + *B*) ⊗ *C*)) = (*A* + *B*) ⊗ *C*
tensor-from-lookup (*dims A* @ *dims C*) (*lookup* ((*A* ⊗ *C*) + (*B* ⊗ *C*))) = (*A* ⊗ *C*) + (*B* ⊗ *C*)
by (*metis* (*no-types*, *lifting*) *assms plus-dim1 dims-tensor-prod tensor-lookup*)+
ultimately show *?thesis* **using** *tensor-from-lookup-eqI*
by (*metis* ‹⋀*is*. *is* ◁ *dims A* @ *dims C* ⟹ *lookup* ((*A* + *B*) ⊗ *C*) *is* = *lookup* (*A* ⊗ *C* + *B* ⊗ *C*) *is*›)
qed

lemma *tensor-prod-distr-right*:
assumes *dims A* = *dims B*
shows *C* ⊗ (*A* + *B*) = (*C* ⊗ *A*) + (*C* ⊗ *B*)
proof −
have ⋀*is*. *is* ◁ *dims C* @ *dims A* ⟹ *lookup* (*C* ⊗ (*A* + *B*)) *is* = *lookup* (*C* ⊗ *A* + *C* ⊗ *B*) *is*
proof −
fix *is* **assume** *is* ◁ *dims C* @ *dims A*
obtain *is1 is2* **where** *is* = *is1* @ *is2 is1* ◁ *dims C is2* ◁ *dims A* **using** *valid-index-split* **using** ‹*is* ◁ *dims C* @ *dims A*› **by** *blast*
then show *lookup* (*C* ⊗ (*A* + *B*)) *is* = *lookup* ((*C* ⊗ *A*) + (*C* ⊗ *B*)) *is*
using *lookup-plus*
using ‹*is2* ◁ *dims A*› ‹*is1* ◁ *dims C*› *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distribs*(*1*) *valid-index-append*
by *fastforce*
qed
moreover have *tensor-from-lookup* (*dims C* @ *dims A*) (*lookup* (*C* ⊗ (*A* + *B*))) = *C* ⊗ (*A* + *B*)
tensor-from-lookup (*dims C* @ *dims A*) (*lookup* ((*C* ⊗ *A*) + (*C* ⊗ *B*))) = (*C* ⊗ *A*) + (*C* ⊗ *B*)
by (*metis* (*no-types*, *lifting*) *assms plus-dim1 dims-tensor-prod tensor-lookup*)+
ultimately show *?thesis* **using** *tensor-from-lookup-eqI*
by (*metis* ‹⋀*is*. *is* ◁ *dims C* @ *dims A* ⟹ *lookup* (*C* ⊗ (*A* + *B*)) *is* = *lookup* (*C* ⊗ *A* + *C* ⊗ *B*) *is*›)
qed

instantiation *tensor* :: (*ring-1*) *monoid-mult*
begin
definition *tensor-one-def*:*1* = *tensor-from-vec* [] [*1*]

lemma *tensor-one-from-lookup*: *1* = *tensor-from-lookup* [] (λ-. *1*)
unfolding *tensor-one-def* **by** (*rule tensor-eqI*; *simp-all add*: *tensor-from-lookup-def*

)

instance proof
fix $A::'a::$*ring-1 tensor*
show $A * 1 = A$ **unfolding** *tensor-one-from-lookup*
by (*rule tensor-lookup-eqI*;*metis lookup-tensor-prod*[*of - A* [] *tensor-from-lookup* [] (λ-. *1*)]
lookup-tensor-from-lookup valid-index.Nil append-Nil2 dims-tensor dims-tensor-prod
length-tensor-vec-from-lookup mult.right-neutral tensor-from-lookup-def)
next
fix $A::'a::$*ring-1 tensor*
show *1* $* A = A$ **unfolding** *tensor-one-from-lookup*
by (*rule tensor-lookup-eqI*; *metis lookup-tensor-prod*[*of* [] *tensor-from-lookup* [] (λ-. *1*) - *A*]
lookup-tensor-from-lookup valid-index.Nil List.append.append-Nil dims-tensor dims-tensor-prod
length-tensor-vec-from-lookup mult.left-neutral tensor-from-lookup-def)
qed
end

lemma *order-tensor-one*: *order 1 = 0* **unfolding** *tensor-one-def* **by** *simp*

lemma *smult-prod-extract1*:
fixes $a::'a::$*comm-ring-1*
shows $a \cdot (A \otimes B) = (a \cdot A) \otimes B$
proof (*rule tensor-lookup-eqI*)
show *dims* $(a \cdot (A \otimes B)) =$ *dims* $((a \cdot A) \otimes B)$ **by** *simp*
fix *is* **assume** *is* $\lhd$ *dims* $(a \cdot (A \otimes B))$
then have *is* $\lhd$ *dims* $(A \otimes B)$ **by** *auto*
then obtain *is1 is2* **where** *is1* $\lhd$ *dims A is2* $\lhd$ *dims B is = is1 @ is2* **by** (*metis dims-tensor-prod valid-index-split*)
then have *is1* $\lhd$ *dims* $(a \cdot A)$ **by** *auto*
show *lookup* $(a \cdot (A \otimes B))$ *is = lookup* $(a \cdot A \otimes B)$ *is*
using *lookup-tensor-prod*[*OF* ‹*is1* $\lhd$ *dims A*› ‹*is2* $\lhd$ *dims B*›] *lookup-tensor-prod*[*OF* ‹*is1* $\lhd$ *dims* $(a \cdot A)$› ‹*is2* $\lhd$ *dims B*›]
lookup-smult[*OF* ‹*is* $\lhd$ *dims* $(A \otimes B)$›] *lookup-smult*[*OF* ‹*is1* $\lhd$ *dims A*›] ‹*is = is1 @ is2*› **by** *simp*
qed

lemma *smult-prod-extract2*:
fixes $a::'a::$*comm-ring-1*
shows $a \cdot (A \otimes B) = A \otimes (a \cdot B)$
proof (*rule tensor-lookup-eqI*)
show *dims* $(a \cdot (A \otimes B)) =$ *dims* $(A \otimes (a \cdot B))$ **by** *simp*
fix *is* **assume** *is* $\lhd$ *dims* $(a \cdot (A \otimes B))$
then have *is* $\lhd$ *dims* $(A \otimes B)$ **by** *auto*
then obtain *is1 is2* **where** *is1* $\lhd$ *dims A is2* $\lhd$ *dims B is = is1 @ is2* **by** (*metis dims-tensor-prod valid-index-split*)
then have *is2* $\lhd$ *dims* $(a \cdot B)$ **by** *auto*

show *lookup* $(a \cdot (A \otimes B))$ *is* = *lookup* $(A \otimes (a \cdot B))$ *is*
using *lookup-tensor-prod*[*OF* ‹*is1* ◁ *dims A*› ‹*is2* ◁ *dims B*›] *lookup-tensor-prod*[*OF* ‹*is1* ◁ *dims A*› ‹*is2* ◁ *dims* $(a \cdot B)$›]
lookup-smult[*OF* ‹*is* ◁ *dims* $(A \otimes B)$›] *lookup-smult*[*OF* ‹*is2* ◁ *dims B*›] ‹*is* = *is1* @ *is2*› **by** *simp*
qed

lemma *order-0-multiple-of-one*:
assumes *order A* = *0*
obtains *a* **where** $A = a \cdot 1$
proof
assume $(\bigwedge a.\ A = a \cdot 1 \Longrightarrow thesis)$
have *length* (*vec A*) = *1* **using** *assms* **by** (*simp add*:*length-vec*)
then obtain *a* **where** *vec A* = [*a*] **by** (*metis One-nat-def Suc-length-conv length-0-conv*)
moreover have *vec* $(a \cdot 1)$ = [*a*] **unfolding** *smult-def tensor-one-def* **by** (*simp add*: *vec-smult-def*)
ultimately have $A = a \cdot 1$ **using** *tensor-eqI* **by** (*metis assms dims-smult length-0-conv order-tensor-one*)
then show $A = hd\ (vec\ A) \cdot 1$ **using** ‹*vec A* = [*a*]› **by** *auto*
qed

lemma *smult-1*:
fixes *A*::*'a*::*ring-1 tensor*
shows $A = 1 \cdot A$ **unfolding** *smult-def tensor-one-def*
apply (*rule tensor-eqI*)
apply (*simp add*: *length-vec length-vec-smult*)
by (*metis dims-tensor length-vec length-vec-smult lookup-smult mult.left-neutral smult-def tensor-lookup-eqI*)

lemma *tensor0-prod-right*[*simp*]: $A \otimes$ *tensor0 ds* = *tensor0* (*dims A* @ *ds*)
proof (*rule tensor-lookup-eqI*,*simp*)
fix *is* **assume** *is* ◁ *dims* $(A \otimes$ *tensor0 ds*$)$
then obtain *is1 is2* **where** *is1* ◁ *dims A is2* ◁ *dims* (*tensor0 ds*) *is* = *is1* @ *is2*
by (*metis dims-tensor0 dims-tensor-prod valid-index-split*)
then show *lookup* $(A \otimes$ *tensor0 ds*$)$ *is* = *lookup* (*tensor0* (*dims A* @ *ds*)) *is*
by (*metis* (*no-types*, *lifting*) ‹*is* ◁ *dims* $(A \otimes$ *tensor0 ds*$)$› *dims-tensor0 dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-right*)
qed

lemma *tensor0-prod-left*[*simp*]: *tensor0 ds* $\otimes$ *A* = *tensor0* (*ds* @ *dims A*)
proof (*rule tensor-lookup-eqI*,*simp*)
fix *is* **assume** *is* ◁ *dims* (*tensor0 ds* $\otimes$ *A*)
then obtain *is1 is2* **where** *is1* ◁ *dims* (*tensor0 ds*) *is2* ◁ *dims A is* = *is1* @ *is2*
by (*metis dims-tensor0 dims-tensor-prod valid-index-split*)

then show *lookup (tensor0 ds ⊗ A) is = lookup (tensor0 (ds @ dims A)) is*
by *(metis (no-types, lifting) ‹is ◁ dims (tensor0 ds ⊗ A)› dims-tensor0 dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-left)*
qed

lemma *subtensor-prod-with-vec*:
assumes *order A = 1 i < hd (dims A)*
shows *subtensor (A ⊗ B) i = lookup A [i] · B*
proof *(rule tensor-lookup-eqI)*
have *dims (A ⊗ B) ≠ []* **using** *assms(1)* **by** *auto*
have *hd (dims A) = hd (dims (A ⊗ B))*
by *(metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(1))*
show *dims (subtensor (A ⊗ B) i) = dims (lookup A [i] · B)*
unfolding *dims-smult dims-subtensor[OF ‹dims (A ⊗ B) ≠ []› ‹i < hd (dims A)›[unfolded ‹hd (dims A) = hd (dims (A ⊗ B))›]]*
by *(metis One-nat-def Suc-length-conv append.simps(2) append-self-conv2 assms(1) dims-tensor-prod length-0-conv list.sel(3))*
next
fix *is* **assume** *is ◁ dims (subtensor (A ⊗ B) i)*
have *dims (A ⊗ B) ≠ []* **using** *assms(1)* **by** *auto*
have *hd (dims A) = hd (dims (A ⊗ B))*
by *(metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(1))*
then have *is ◁ dims B*
using *‹is ◁ dims (subtensor (A ⊗ B) i)›[unfolded dims-subtensor[OF ‹dims (A ⊗ B) ≠ []› ‹i < hd (dims A)›[unfolded ‹hd (dims A) = hd (dims (A ⊗ B))›]]]*
by *(metis One-nat-def Suc-length-conv append-self-conv2 assms(1) dims-tensor-prod length-0-conv list.sel(3) list.simps(3) tl-append2)*
have *[i] ◁ dims A* **using** *assms* **by** *(metis One-nat-def Suc-length-conv length-0-conv list.sel(1) valid-index.Nil valid-index.simps)*
then have *i # is ◁ dims (A ⊗ B)* **using** *‹is ◁ dims (subtensor (A ⊗ B) i)› dims-subtensor valid-index.Cons* **by** *auto*
then show *lookup (subtensor (A ⊗ B) i) is = lookup (lookup A [i] · B) is*
unfolding *lookup-subtensor1[OF ‹i # is ◁ dims (A ⊗ B)›]*
using *lookup-tensor-prod[OF ‹[i] ◁ dims A› ‹is ◁ dims B›] lookup-smult ‹is ◁ dims B›* **using** *append-Cons* **by** *fastforce*
qed

end

6 Unit Vectors as Tensors

theory *Tensor-Unit-Vec*
imports *Tensor-Product*
begin

definition *unit-vec::nat ⇒ nat ⇒ 'a::ring-1 tensor*
where *unit-vec n i = tensor-from-lookup [n] (λx. if x=[i] then 1 else 0)*

lemma *dims-unit-vec*: *dims* (*unit-vec n i*) = [*n*] **unfolding** *unit-vec-def* **by** (*simp add*: *tensor-from-lookup-def*)

lemma *lookup-unit-vec*:
assumes *j*<*n*
shows *lookup* (*unit-vec n i*) [*j*] = (*if i*=*j then 1 else 0*)
proof −
  have [*j*] ◁ [*n*] **by** (*simp add*: *assms valid-index.Cons valid-index.Nil*)
  then have *lookup* (*unit-vec n i*) [*j*] = (λ*x*. *if x*=[*i*] *then 1 else 0*) [*j*]
    by (*simp add*: *lookup-tensor-from-lookup unit-vec-def*)
  then show *?thesis* **by** *auto*
qed

lemma *subtensor-prod-with-unit-vec*:
fixes *A*::'*a*::*ring-1 tensor*
assumes *j*<*n*
shows *subtensor* (*unit-vec n i* ⊗ *A*) *j* = (*if i*=*j then A else* (*tensor0* (*dims A*)))
proof −
  have *0*:*lookup* (*unit-vec n i*) [*j*] = (*if i*=*j then 1 else 0*) **unfolding** *unit-vec-def*
   by (*simp add*: *assms lookup-tensor-from-lookup valid-index.Cons valid-index.Nil*)
 have *1*:*order* (*unit-vec n i*) = *1* **unfolding** *unit-vec-def* **by** (*simp add*: *tensor-from-lookup-def*)
  have *2*:*j* < *hd* (*dims* (*tensor-from-lookup* [*n*] (λ*x*. *if x* = [*i*] *then 1 else 0*)))
   using *assms dims-tensor length-tensor-vec-from-lookup list.sel*(*1*) *tensor-from-lookup-def transfer-nat-int-list-functions*(*2*)
    by *metis*
 show *?thesis* **using** *unit-vec-def subtensor-prod-with-vec 1 2 0 smult-1 tensor-smult0*
    by (*metis* (*no-types*, *lifting*) *tensor-from-lookup-eqI*)
qed

lemma *subtensor-decomposition*:
assumes *dims A* ≠ []
shows *listsum* (*dims A*) (*map* (λ*i*. *unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*) [*0*..<*hd* (*dims A*)]) = *A* (**is** *?LS* = *A*)
proof −
  let *?f* = λ*i*. *unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*
  have *correct-dims*:⋀*B*. *B* ∈ *set* (*map ?f* [*0*..<*hd* (*dims A*)]) ⟹ *dims B* = *dims A*
  proof−
    fix *B*
    assume *B* ∈ *set* (*map ?f* [*0*..<*hd* (*dims A*)])
    then obtain *i* **where** *B*:*B* = *?f i* **and** *i*<*hd* (*dims A*) **by** *auto*
     then have *dims* (*subtensor A i*) = *tl* (*dims A*) **using** *dims-subtensor* **using** *assms* **by** *blast*
    then show *dims B* = *dims A* **unfolding** *B*
      by (*metis append-Cons assms dims-tensor-prod dims-unit-vec list.exhaust-sel self-append-conv2*)
  qed
  have ⋀*j*. *j* < *hd* (*dims A*) ⟹ *subtensor ?LS j* = *subtensor A j*

proof −
fix j
assume $j < hd\ (dims\ A)$
have *1*:*subtensor ?LS j = listsum* (*tl* (*dims A*)) (*map* (λ*A*. *subtensor A j*) (*map ?f* [*0*..<*hd* (*dims A*)]))
using *subtensor-listsum*[*of* (*map* (λ*i*. *?f i*) [*0*..<*hd* (*dims A*)]) *dims A j*, *OF correct-dims assms* ‹*j* < *hd* (*dims A*)›]
by *linarith*
also have ... = *listsum* (*tl* (*dims A*)) (*map* (λ*i*. *subtensor* (*?f i*) *j*) [*0*..<*hd* (*dims A*)])
proof −
have *map* (λ*A*. *subtensor A j*) (*map ?f* [*0*..<*hd* (*dims A*)]) = *map* (λ*i*. *subtensor* (*?f i*) *j*) [*0*..<*hd* (*dims A*)]
unfolding *map-map*[*of* (λ*A*. *subtensor A j*) *?f* [*0*..<*hd* (*dims A*)]] **by** *simp*
with *1* **show** *?thesis* **by** *metis*
qed
also have ... = *map* (λ*i*. *if i* = *j then subtensor A i else tensor0* (*dims* (*subtensor A i*))) [*0*..<*hd* (*dims A*)] ! *j*
unfolding *subtensor-prod-with-unit-vec*[*OF* ‹*j* < *hd* (*dims A*)›]
using *listsum-all-0-but-one*[*of j* (*map* (λ*i*. *if i* = *j then subtensor A i else tensor0* (*dims* (*subtensor A i*))) [*0*..<*hd* (*dims A*)]) *tl* (*dims A*)]
by (*simp add*: ‹*j* < *hd* (*dims A*)› *assms*)
also have ... = *subtensor A j* **by** (*simp add*: ‹*j* < *hd* (*dims A*)›)
finally show *subtensor ?LS j* = *subtensor A j* **by** *auto*
qed
moreover have *dims ?LS* = *dims A* **using** *correct-dims listsum-dims* **by** *blast*
ultimately show *?thesis* **using** *subtensor-eqI* **by** (*metis* (*no-types*, *lifting*) *assms*)
qed

end

7 Tensor CP-Rank

theory *Tensor-Rank*
imports *Tensor-Unit-Vec*
begin

inductive *cprank-max1*::*'a*::*ring-1 tensor* ⇒ *bool* **where**
order1: *order A* ≤ *1* ⟹ *cprank-max1 A* |
higher-order: *order A* = *1* ⟹ *cprank-max1 B* ⟹ *cprank-max1* (*A* ⊗ *B*)

lemma *cprank-max1-order0*: *cprank-max1 B* ⟹ *order A* = *0* ⟹ *cprank-max1* (*A* ⊗ *B*)
proof (*induction B rule*:*cprank-max1.induct*)
case *order1*
then show *?case* **by** (*simp add*: *cprank-max1.order1*)
next
case (*higher-order A' B*)

then have *order* $(A \otimes A') = 1$ **by** *simp*
then show *?case* **using** *higher-order cprank-max1.higher-order* **by** (*metis mult.assoc*)
qed

lemma *cprank-max1-order-le1*: *order* $A \leq 0 \Longrightarrow$ *cprank-max1* $B \Longrightarrow$ *cprank-max1* $(A \otimes B)$
by (*simp add*: *cprank-max1-order0*)

lemma *cprank-max1-prod*: *cprank-max1* $A \Longrightarrow$ *cprank-max1* $B \Longrightarrow$ *cprank-max1* $(A \otimes B)$
apply(*induction A rule*: *cprank-max1.induct*)
apply (*meson higher-order le-neq-trans less-one cprank-max1-order0*)
by (*simp add*: *higher-order mult.assoc*)

lemma *cprank-max1-prod-list*:
assumes $\bigwedge B.\ B \in set\ Bs \Longrightarrow$ *cprank-max1* B
shows *cprank-max1* (*prod-list Bs*)
using *assms* **by** (*induction Bs*, *metis dims-smult dims-tensor0 list.size*(*3*) *prod-list.Nil order1 order-0-multiple-of-one zero-le-one*, *simp add*: *cprank-max1-prod*)

lemma *cprank-max1-prod-listE*:
fixes A::$'a$::*comm-ring-1 tensor*
assumes *cprank-max1 A*
obtains *Bs a* **where** $\bigwedge B.\ B \in set\ Bs \Longrightarrow order\ B = 1$ $a \cdot$ *prod-list Bs* $= A$
using *assms* **proof** (*induction A arbitrary*:*thesis rule*:*cprank-max1.induct*)
case (*order1 A*)
then show *?case*
proof (*cases order A = 0*)
case *True*
then obtain *a* **where** $A = a \cdot$ *prod-list* [] **using** *order-0-multiple-of-one* **using** *prod-list.Nil* **by** *auto*
then show *?thesis* **using** *length-pos-if-in-set order1.prems* **by** *fastforce*
next
case *False*
then have *order A = 1* **using** *order1* **by** *linarith*
then have $A = 1 \cdot$ *prod-list* [*A*] **by** (*simp add*: *smult-1*)
then show *?thesis* **by** (*metis* ‹*order A = 1*› *length-greater-0-conv length-pos-if-in-set order1.prems set-ConsD*)
qed
next
case (*higher-order A B*)
then obtain *Bs b* **where** $(\bigwedge B'.\ B' \in set\ Bs \Longrightarrow order\ B' = 1)$ $b \cdot$ *prod-list Bs* $= B$ **by** *metis*
then have $(\bigwedge B.\ B \in set\ (A \# Bs) \Longrightarrow order\ B = 1)$ **using** *higher-order* **by** *auto*
have $A \otimes B = b \cdot (A \otimes$ *prod-list Bs*) **using** *smult-prod-extract2* ‹$b \cdot$ *prod-list Bs* $= B$› **by** *metis*
then show *?case* **by** (*metis* ‹$\bigwedge Ba.\ Ba \in set\ (A \# Bs) \Longrightarrow order\ Ba = 1$› *higher-order.prems prod-list.Cons*)

qed

inductive *cprank-max* :: *nat* ⇒ *'a*::*ring-1 tensor* ⇒ *bool* **where**
cprank-max0: *cprank-max 0* (*tensor0 ds*) |
cprank-max-Suc: *dims A* = *dims B* ⟹ *cprank-max1 A* ⟹ *cprank-max j B* ⟹
cprank-max (*Suc j*) (*A*+*B*)

lemma *cprank-max1*: *cprank-max1 A* ⟹ *cprank-max 1 A*
by (*metis One-nat-def dims-tensor0 cprank-max.simps cprank-max0 tensor-add-0-right*)

lemma *cprank-max-plus*: *cprank-max i A* ⟹ *cprank-max j B* ⟹ *dims A* = *dims*
B ⟹ *cprank-max* (*i*+*j*) (*A*+*B*)
apply (*induction A rule*:*cprank-max.induct*)
apply *auto*[*1*]
by (*metis add-Suc plus-assoc plus-dim1 cprank-max.intros*(*2*))

lemma *cprank-max-listsum*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
and ⋀*A*. *A*∈*set As* ⟹ *cprank-max n A*
shows *cprank-max* (*n*∗*length As*) (*listsum ds As*)
using *assms* **proof** (*induction As*)
case *Nil*
then show *?case* **using** *listsum-Nil cprank-max.simps* **by** *fastforce*
next
case (*Cons A As*)
then show *?case* **using** *cprank-max-plus*[*of n A n* ∗ *length As listsum ds As*]
by (*simp add*: *length-Cons list.set-intros*(*1*) *listsum-Cons listsum-dims set-subset-Cons*
subsetCE)
qed

lemma *cprank-maxE*:
assumes *cprank-max n A*
obtains *BS* **where** (⋀*B*. *B*∈*set BS* ⟹ *cprank-max1 B*) **and** (⋀*B*. *B*∈*set BS*
⟹ *dims A* = *dims B*) **and** *listsum* (*dims A*) *BS* = *A* **and** *length BS* = *n*
using *assms* **proof** (*induction arbitrary*:*thesis rule*:*cprank-max.induct*)
case (*cprank-max0 ds*)
have *Tensor-Plus.listsum* (*dims* (*tensor0 ds*)) [] = *tensor0 ds* **by** (*simp add*:
listsum-Nil)
then show *?case* **using** *cprank-max0.prems* **by** *fastforce*
next
case (*cprank-max-Suc A B j*)
then obtain *BS* **where** *BS-def*:(⋀*B*. *B* ∈ *set BS* ⟹ *cprank-max1 B*) (⋀*B*′.
B′ ∈ *set BS* ⟹ *dims B*′ = *dims B*)
listsum (*dims B*) *BS* = *B length BS* = *j* **by** *metis*
then have *listsum* (*dims* (*A* + *B*)) (*A* # *BS*) = *A* + *B*
by (*simp add*: *listsum-Cons cprank-max-Suc.hyps*(*1*))
then show *?case* **using** *BS-def length-Cons cprank-max-Suc.hyps*(*2*) *cprank-max-Suc.prems*
set-ConsD
by (*metis plus-dim1 cprank-max-Suc.hyps*(*1*))

qed

lemma *cprank-maxI*:
assumes $\bigwedge B.\ B{\in}set\ BS \Longrightarrow cprank\text{-}max1\ B$
and $\bigwedge B.\ B{\in}set\ BS \Longrightarrow dims\ B = ds$
shows *cprank-max* (*length BS*) (*listsum ds BS*)
using *assms* **proof** (*induction BS*)
case *Nil*
then show *?case* **by** (*simp add*: *listsum-Nil cprank-max0*)
next
case (*Cons B BS*)
then show *?case*
by (*simp add*: *length-Cons list.set-intros*(*1*) *list.set-intros*(*2*) *listsum-Cons listsum-dims cprank-max-Suc*)
qed

lemma *cprank-max-0E*: *cprank-max 0 A* $\Longrightarrow$ *A* = *tensor0* (*dims A*) **by** (*metis dims-tensor0 length-0-conv cprank-max0 cprank-maxE*)

lemma *listsum-prod-distr-right*:
assumes ($\bigwedge C.\ C{\in}set\ CS \Longrightarrow dims\ C = ds$)
shows $A \otimes listsum\ ds\ CS = listsum\ (dims\ A\ @\ ds)\ (map\ (\lambda C.\ A \otimes C)\ CS)$
using *assms* **proof** (*induction CS*)
case *Nil*
then show *?case* **by** (*simp add*:*listsum-Nil*)
next
case (*Cons C CS*)
then have *dims C* = *dims* (*listsum ds CS*) **by** (*simp add*: *list.set-intros*(*1*) *list.set-intros*(*2*) *listsum-dims*)
then show *?case* **unfolding** *listsum-Cons list.map*(*2*)
using *tensor-prod-distr-right Cons.IH Cons.prems list.set-intros*(*2*) **by** *fastforce*
qed

lemma *cprank-max-prod-order1*:
assumes *order A* = *1*
and *cprank-max n B*
shows *cprank-max n* (*A*⊗*B*)
proof −
obtain *CS* **where** ($\bigwedge C.\ C{\in}set\ CS \Longrightarrow cprank\text{-}max1\ C$)
and ($\bigwedge C.\ C{\in}set\ CS \Longrightarrow dims\ C = dims\ B$)
and *listsum* (*dims B*) *CS* = *B*
and *length CS* = *n*
using *assms*(*2*) *cprank-maxE* **by** *metis*
def *CS′* == *map* (λ*C*. *A*⊗*C*) *CS*
then have $\bigwedge C'.\ C'{\in}set\ CS' \Longrightarrow cprank\text{-}max1\ C'$
using *assms*(*1*) *higher-order* ‹$\bigwedge C.\ C \in set\ CS \Longrightarrow cprank\text{-}max1\ C$› *imageE set-map* **by** *auto*
have *listsum* (*dims A* @ *dims B*) *CS′* = *A*⊗*B* **using** *CS′-def* ‹*Tensor-Plus.listsum* (*dims B*) *CS* = *B*›

using ‹⋀*Ca*. *Ca* ∈ *set CS* ⟹ *dims Ca* = *dims B*› *listsum-prod-distr-right* **by** *fastforce*
then show *?thesis* **by** (*metis* (*mono-tags*, *lifting*) *CS'-def* ‹⋀*C'*. *C'* ∈ *set CS'* ⟹ *cprank-max1 C'*› ‹⋀*Ca*. *Ca* ∈ *set CS* ⟹ *dims Ca* = *dims B*› ‹*length CS* = *n*› *dims-tensor-prod imageE length-map cprank-maxI set-map*)
qed

lemma *cprank-max-upper-bound*:
shows *cprank-max* (*prod-list* (*dims A*)) *A*
proof (*induction A rule*:*subtensor-induct*)
case (*order-0 A*)
then have *cprank-max 1 A* **using** *order1 cprank-max1* **by** *force*
then show *?case* **using** *order-0* **by** *auto*
next
case (*order-step A*)
def *Bs* == *map* (λ*i*. *unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*) [*0*..<*hd* (*dims A*)]
have ⋀*B*. *B* ∈ *set Bs* ⟹ *dims A* = *dims B*
proof −
fix *B* **assume** *B* ∈ *set Bs*
obtain *i* **where** *i*<*hd* (*dims A*) *Bs*!*i*=*B* **using** *Bs-def* ‹*B* ∈ *set Bs*› **by** *auto*
then have *dims* (*unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*) = *dims A*
using *dims-unit-vec order-step.hyps*
by (*metis append-Cons dims-subtensor dims-tensor-prod list.exhaust-sel self-append-conv2*)
then show *dims A* = *dims B* **using** *Bs-def* ‹*Bs* ! *i* = *B*› ‹*i* < *hd* (*dims A*)›
by *auto*
qed
have ⋀*B*. *B* ∈ *set Bs* ⟹ *cprank-max* (*prod-list* (*tl* (*dims A*))) *B*
proof −
fix *B* **assume** *B* ∈ *set Bs*
obtain *i* **where** *i*<*hd* (*dims A*) *Bs*!*i*=*B* **using** *Bs-def* ‹*B* ∈ *set Bs*› **by** *auto*
then have *cprank-max* (*prod-list* (*tl* (*dims A*))) (*unit-vec* (*hd* (*dims A*)) *i* ⊗ *subtensor A i*)
by (*metis One-nat-def dims-subtensor dims-unit-vec length-Cons list.size*(*3*) *order-step.IH order-step.hyps cprank-max-prod-order1*)
then show *cprank-max* (*prod-list* (*tl* (*dims A*))) *B* **using** *Bs-def* ‹*Bs* ! *i* = *B*› ‹*i* < *hd* (*dims A*)› **by** *auto*
qed
then show *?case* **using** *subtensor-decomposition*[*OF order-step.hyps*] *cprank-max-listsum*
by (*metis* (*no-types*, *lifting*) *Bs-def* ‹⋀*Ba*. *Ba* ∈ *set Bs* ⟹ *dims A* = *dims Ba*› *diff-zero length-map length-upt list.exhaust-sel prod-list.Cons mult.commute order-step.hyps*)
qed

definition *cprank* :: *'a*::*ring-1 tensor* ⇒ *nat* **where**
cprank A = (*LEAST n*. *cprank-max n A*)

lemma *cprank-upper-bound*: *cprank A* ≤ *prod-list* (*dims A*)
unfolding *cprank-def* **using** *cprank-max-upper-bound Least-le* **by** *fastforce*

lemma *cprank-max-cprank*: *cprank-max* (*cprank A*) *A*
unfolding *cprank-def* **using** *cprank-max-upper-bound* **by** (*metis LeastI*)

end

8 Missing Lemmas of Vector_Space

theory *DL-Missing-Vector-Space*
imports *Jordan-Normal-Form.Missing-VectorSpace*
begin
find-theorems *vectorspace.basis*

lemma (**in** *vectorspace*) *dim1I*:
assumes *gen-set* {*v*}
assumes $v \neq \mathbf{0}_V$ *v* ∈ *carrier V*
shows *dim* = *1*
proof −
have *basis* {*v*} **by** (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *basis-def empty-iff empty-subsetI*
finite.emptyI finite-lin-indpt2 insert-iff insert-subset insert-union lin-dep-iff-in-span
span-empty)
then show *?thesis* **using** *dim-basis* **by** *force*
qed

lemma (**in** *vectorspace*) *dim0I*:
assumes *gen-set* {$\mathbf{0}_V$}
shows *dim* = *0*
proof −
have *basis* {} **unfolding** *basis-def* **using** *already-in-span assms finite-lin-indpt2*
span-zero **by** *auto*
then show *?thesis* **using** *dim-basis* **by** *force*
qed

lemma (**in** *vectorspace*) *dim-le1I*:
assumes *gen-set* {*v*}
assumes *v* ∈ *carrier V*
shows *dim* ≤ *1*
by (*metis One-nat-def assms*(*1*) *assms*(*2*) *bot.extremum card.empty card.insert*
empty-iff finite.intros(*1*)
finite.intros(*2*) *insert-subset vectorspace.gen-ge-dim vectorspace-axioms*)

end

9 Missing Lemmas of VS_Connect

theory *DL-Missing-VS-Connect*
imports *Jordan-Normal-Form.VS-Connect DL-Missing-Vector-Space*
begin

lemma (**in** *vec-space*) *fin-dim-span*:
assumes *finite A A* ⊆ *carrier V*
shows *vectorspace.fin-dim class-ring* (*vs* (*span A*))
proof −
have *vectorspace class-ring* (*span-vs A*)
using *assms span-is-subspace subspace-def subspace-is-vs* **by** *simp*
have *A* ⊆ *span A* **using** *assms in-own-span* **by** *simp*
have *submodule class-ring* (*span A*) *V* **using** *assms span-is-submodule* **by** *simp*
have *LinearCombinations.module.span class-ring* (*vs* (*span A*)) *A* = *carrier* (*vs* (*span A*))
using *span-li-not-depend*(*1*)[*OF* ‹*A* ⊆ *span A*› ‹*submodule class-ring* (*span A*) *V*›] **by** *auto*
then show *?thesis* **unfolding** *vectorspace.fin-dim-def*[*OF* ‹*vectorspace class-ring* (*span-vs A*)›]
using *List.finite-set* ‹*A* ⊆ *span A*› ‹*vectorspace class-ring* (*vs* (*span A*))›
vec-vs vectorspace.carrier-vs-is-self[*OF* ‹*vectorspace class-ring* (*span-vs A*)›]
using *assms*(*1*) **by** *auto*
qed

lemma (**in** *vec-space*) *fin-dim-span-cols*:
assumes *A* ∈ *carrier-mat n nc*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*cols A*))))
using *fin-dim-span cols-dim List.finite-set assms carrier-matD*(*1*) *module-vec-simps*(*3*)
by *force*

end

10 Missing Lemmas of List

theory *DL-Missing-List*
imports *Main*
begin

lemma *nth-map-zip*:
assumes *i* < *length xs*
assumes *i* < *length ys*
shows *map f* (*zip xs ys*) ! *i* = *f* (*xs* ! *i*, *ys* ! *i*)
using *nth-zip nth-map length-zip* **by** (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *nth-map-zip2*:
assumes *i* < *length* (*map f* (*zip xs ys*))
shows *map f* (*zip xs ys*) ! *i* = *f* (*xs* ! *i*, *ys* ! *i*)
using *nth-zip nth-map length-zip assms* **by** *simp*

fun *find-first* **where**

find-first a [] = *undefined* |
find-first a (*x* # *xs*) = (*if x* = *a then 0 else Suc* (*find-first a xs*))

lemma *find-first-le*:
assumes *a* ∈ *set xs*
shows *find-first a xs* < *length xs*
using *assms* **proof** (*induction xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs*)
then show *?case*
using *find-first.simps*(*2*) *nth-Cons-0 nth-Cons-Suc set-ConsD* **by** *auto*
qed

lemma *nth-find-first*:
assumes *a* ∈ *set xs*
shows *xs* ! (*find-first a xs*) = *a*
using *assms* **proof** (*induction xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs*)
then show *?case*
using *find-first.simps*(*2*) *nth-Cons-0 nth-Cons-Suc set-ConsD* **by** *auto*
qed

lemma *find-first-unique*:
assumes *distinct xs*
and *i* < *length xs*
shows *find-first* (*xs* ! *i*) *xs* = *i*
using *assms* **proof** (*induction xs arbitrary*:*i*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs i*)
then show *?case* **by** (*cases i*; *auto*)
qed

end

11 Matrix Rank

theory *DL-Rank*
imports *DL-Missing-VS-Connect DL-Missing-List*
Jordan-Normal-Form.Determinant
Jordan-Normal-Form.Missing-VectorSpace
Jordan-Normal-Form.Matrix
begin

lemma (**in** *vectorspace*) *full-dim-span*:
assumes $S \subseteq$ *carrier* V
and *finite* S
and *vectorspace.dim* K (*span-vs* S) = *card* S
shows *lin-indpt* S
proof −
have *vectorspace* K (*span-vs* S)
using *field.field-axioms vectorspace-def submodule-is-module*[*OF span-is-submodule*[*OF assms*(*1*)]] **by** *metis*
have $S \subseteq$ *carrier* (*span-vs* S) **by** (*simp add*: *assms*(*1*) *in-own-span*)
have *LinearCombinations.module.span* K (*vs* (*span* S)) S = *carrier* (*vs* (*span* S))
using *module.span-li-not-depend*[*OF - span-is-submodule*[*OF assms*(*1*)]]
by (*simp add*: *assms*(*1*) *in-own-span*)
have *vectorspace.basis* K (*vs* (*span* S)) S
using *vectorspace.dim-gen-is-basis*[*OF* ‹*vectorspace* K (*span-vs* S)› ‹*finite* S› ‹$S \subseteq$ *carrier* (*span-vs* S)›
‹*LinearCombinations.module.span* K (*vs* (*span* S)) S = *carrier* (*vs* (*span* S))›]
‹*vectorspace.dim* K (*span-vs* S) = *card* S›
by *simp*
then have *LinearCombinations.module.lin-indpt* K (*vs* (*span* S)) S
using *vectorspace.basis-def*[*OF* ‹*vectorspace* K (*span-vs* S)›] **by** *blast*
then show *?thesis* **using** *module.span-li-not-depend*[*OF - span-is-submodule*[*OF assms*(*1*)]]
by (*simp add*: *assms*(*1*) *in-own-span*)
qed

lemma (**in** *vectorspace*) *dim-span*:
assumes $S \subseteq$ *carrier* V
and *finite* S
and *maximal* U ($\lambda T.\ T \subseteq S \wedge$ *lin-indpt* T)
shows *vectorspace.dim* K (*span-vs* S) = *card* U
proof −
have *lin-indpt* U $U \subseteq S$ **by** (*metis assms*(*3*) *maximal-def*)+
then have $U \subseteq$ *span* S **using** *in-own-span*[*OF assms*(*1*)] **by** *blast*
then have *lin-indpt*: *LinearCombinations.module.lin-indpt* K (*span-vs* S) U
using *module.span-li-not-depend*(*2*)[*OF* ‹$U \subseteq$ *span* S›] ‹*lin-indpt* U› *assms*(*1*) *span-is-submodule* **by** *blast*
have *span* U = *span* S
proof (*rule ccontr*)
assume *span* $U \neq$ *span* S
have *span* $U \subseteq$ *span* S **using** *span-is-monotone* ‹$U \subseteq S$› **by** *metis*
then have ¬ $S \subseteq$ *span* U **by** (*meson* ‹$U \subseteq S$› ‹*span* $U \neq$ *span* S› *assms*(*1*) *span-is-submodule*
span-is-subset subset-antisym subset-trans)
then obtain s **where** $s \in S$ $s \notin$ *span* U **by** *blast*
then have *lin-indpt* ($U \cup \{s\}$) **using** *lindep-span*
by (*meson* ‹$U \subseteq S$› ‹*lin-indpt* U› *assms*(*1*) *lin-dep-iff-in-span set-rev-mp*

span-mem subset-trans)
  have $s \notin U$ **using** ‹$U \subseteq S$› ‹$s \notin$ *span* U› *assms*(*1*) *span-mem* **by** *auto*
  then have ($U \cup \{s\}$) $\subseteq S \wedge$ *lin-indpt* ($U \cup \{s\}$) **using** ‹$U \subseteq S$› ‹*lin-indpt* ($U \cup \{s\}$)› ‹$s \in S$› **by** *auto*
  then have $\neg$*maximal* U ($\lambda T.\ T \subseteq S \wedge$ *lin-indpt* T)
   unfolding *maximal-def* **using** *Un-subset-iff* ‹$s \notin U$› *insert-subset order-refl* **by** *auto*
  then show *False* **using** *assms* **by** *metis*
 qed
 then have *span*:*LinearCombinations.module.span* K (*vs* (*span* S)) U = *span* S
  using *module.span-li-not-depend*[*OF* ‹$U \subseteq$ *span* S›]
 by (*simp add*: *LinearCombinations.module.span-is-submodule assms*(*1*) *module-axioms*)
 have *vectorspace* K (*vs* (*span* S))
  using *field.field-axioms vectorspace-def submodule-is-module*[*OF span-is-submodule*[*OF assms*(*1*)]] **by** *metis*
 then have *vectorspace.basis* K (*vs* (*span* S)) U **using** *vectorspace.basis-def*[*OF* ‹*vectorspace* K (*vs* (*span* S))›]
  by (*simp add*: *span* ‹$U \subseteq$ *span* S› *lin-indpt*)
 then show *?thesis*
  using ‹$U \subseteq S$› ‹*vectorspace* K (*vs* (*span* S))› *assms*(*2*) *infinite-super vectorspace.dim-basis* **by** *blast*
qed

definition (**in** *vec-space*) *rank* ::'*a mat* ⇒ *nat*
where *rank* A = *vectorspace.dim class-ring* (*span-vs* (*set* (*cols* A)))

lemma (**in** *vec-space*) *rank-card-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes *maximal* S ($\lambda T.\ T \subseteq$ *set* (*cols* A) $\wedge$ *lin-indpt* T)
shows *rank* A = *card* S
proof −
 have *set* (*cols* A) $\subseteq$ *carrier-vec n* **using** *cols-dim assms*(*1*) **by** *blast*
 have *finite* (*set* (*cols* A)) **by** *blast*
 show *?thesis* **using** *dim-span*[*OF* ‹*set* (*cols* A) $\subseteq$ *carrier-vec n*› ‹*finite* (*set* (*cols* A))› *assms*(*2*)]
  unfolding *rank-def* **by** *blast*
qed

lemma *maximal-exists-superset*:
 assumes *finite* S
 assumes *maxc*: $\bigwedge A.\ P\ A \Longrightarrow A \subseteq S$ **and** $P\ B$
 shows $\exists A.$ *finite* $A \wedge$ *maximal* $A\ P \wedge B \subseteq A$
proof −
 have *finite* ($S - B$) **using** *assms*(*1*) *assms*(*3*) *infinite-super maxc* **by** *blast*
 then show *?thesis* **using** ‹$P\ B$›
 proof (*induction* $S - B$ *arbitrary*:B *rule*: *finite-psubset-induct*)
  case (*psubset* B)
  then show *?case*
  proof (*cases maximal* $B\ P$)

case *True*
then show *?thesis* **using** *order-refl psubset.hyps* **by** (*metis assms*(*1*) *maxc psubset.prems rev-finite-subset*)
next
case *False*
then obtain B' **where** $B \subset B'$ *P* B' **using** *maximal-def psubset.prems* **by** (*metis dual-order.order-iff-strict*)
then have $B' \subseteq S$ $B \subseteq S$ **using** *maxc* ‹*P B*› **by** *auto*
then have $S - B' \subset S - B$ **using** ‹$B \subset B'$› **by** *blast*
then show *?thesis* **using** *psubset*(*2*)[*OF* ‹$S - B' \subset S - B$› ‹*P* B'›] **using** ‹$B \subset B'$› **by** *fast*
qed
qed
qed

lemma (**in** *vec-space*) *rank-ge-card-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes $U \subseteq$ *set* (*cols A*)
assumes *lin-indpt U*
shows *rank A* $\geq$ *card U*
proof −
obtain *S* **where** *maximal S* ($\lambda T.$ $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) $U \subseteq S$ *finite S*
using *maximal-exists-superset*[*of set* (*cols A*) ($\lambda T.$ $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) *U*]
using *List.finite-set assms*(*2*) *assms*(*3*) *maximal-exists-superset* **by** *blast*
then show *?thesis*
unfolding *rank-card-indpt*[*OF* ‹$A \in$ *carrier-mat n nc*› ‹*maximal S* ($\lambda T.$ $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)›]
using *card-mono* **by** *blast*
qed

lemma (**in** *vec-space*) *lin-indpt-full-rank*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
assumes *lin-indpt* (*set* (*cols A*))
shows *rank A* = *nc*
proof −
have *maximal* (*set* (*cols A*)) ($\lambda T.$ $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)
by (*simp add*: *assms*(*3*) *maximal-def subset-antisym*)
then have *rank A* = *card* (*set* (*cols A*)) **using** *assms*(*1*) *vec-space.rank-card-indpt* **by** *blast*
then show *?thesis* **using** *assms*(*1*) *assms*(*2*) *distinct-card* **by** *fastforce*
qed

lemma (**in** *vec-space*) *rank-le-nc*:
assumes $A \in$ *carrier-mat n nc*
shows *rank A* $\leq$ *nc*
proof −
obtain *S* **where** *maximal S* ($\lambda T.$ $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)

using *maximal-exists*[*of* ($\lambda T.\ T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) *card* (*set* (*cols A*)) {}]
by (*meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2 rev-finite-subset*)
then have *card S* $\leq$ *card* (*set* (*cols A*)) **by** (*simp add*: *card-mono maximal-def*)
then have *card S* $\leq$ *nc*
using *assms*(*1*) *cols-length card-length carrier-matD*(*2*) **by** (*metis dual-order.trans*)
then show *?thesis*
using *rank-card-indpt*[*OF* ‹*A* $\in$ *carrier-mat n nc*› ‹*maximal S* ($\lambda T.\ T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)›]
by *simp*
qed

lemma (**in** *vec-space*) *full-rank-lin-indpt*:
assumes *A* $\in$ *carrier-mat n nc*
assumes *rank A* = *nc*
assumes *distinct* (*cols A*)
shows *lin-indpt* (*set* (*cols A*))
proof −
have *1*:*set* (*cols A*) $\subseteq$ *carrier-vec n* **using** *assms*(*1*) *cols-dim* **by** *blast*
have *2*:*finite* (*set* (*cols A*)) **by** *simp*
have *card* (*set* (*cols A*)) = *nc*
using *assms*(*1*) *assms*(*3*) *distinct-card* **by** *fastforce*
have *3*:*vectorspace.dim class-ring* (*span-vs* (*set* (*cols A*))) = *card* (*set* (*cols A*))
using ‹*rank A* = *nc*›[*unfolded rank-def*]
using *assms*(*1*) *assms*(*3*) *distinct-card* **by** *fastforce*
show *?thesis* **using** *full-dim-span*[*OF 1 2 3*] **.**
qed

lemma (**in** *vec-space*) *mat-mult-eq-lincomb*:
assumes *A* $\in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
shows *A* $*_v$ (*vec nc* ($\lambda i.$ *a* (*col A i*))) = *lincomb a* (*set* (*cols A*))
proof (*rule eq-vecI*)
have *finite* (*set* (*cols A*)) **using** *assms*(*1*) **by** *simp*
then show *dim-vec* (*A* $*_v$ (*vec nc* ($\lambda i.$ *a* (*col A i*)))) = *dim-vec* (*lincomb a* (*set* (*cols A*)))
using *assms cols-dim vec-space.lincomb-dim* **by** (*metis dim-mult-mat-vec carrier-matD*(*1*))
fix *i* **assume** *i* < *dim-vec* (*lincomb a* (*set* (*cols A*)))
then have *i* < *n* **using** ‹*dim-vec* (*A* $*_v$ (*vec nc* ($\lambda i.$ *a* (*col A i*)))) = *dim-vec* (*lincomb a* (*set* (*cols A*)))› *assms* **by** *auto*
have *set* (*cols A*) $\subseteq$ *carrier-vec n* **using** *cols-dim* ‹*A* $\in$ *carrier-mat n nc*› *carrier-matD*(*1*) **by** *blast*
have *bij-betw* (*nth* (*cols A*)) {..<*length* (*cols A*)} (*set* (*cols A*))
unfolding *bij-betw-def* **by** (*rule conjI*, *simp add*: *inj-on-nth* ‹*distinct* (*cols A*)›;
metis subset-antisym in-set-conv-nth lessThan-iff rev-image-eqI subsetI image-subsetI lessThan-iff nth-mem)
then have ($\sum x \in$ *set* (*cols A*). *a x* $*$ *x* $ *i*) =

$(\sum j{\in}\{..{<}\mathit{length}\ (\mathit{cols}\ A)\}.\ a\ (\mathit{cols}\ A\ !\ j) * (\mathit{cols}\ A\ !\ j)\ \$\ i)$
using *bij-betw-imp-surj-on bij-betw-imp-inj-on* **by** (*metis* (*no-types*, *lifting*) *sum.reindex-cong*)
also have ... = $(\sum j{\in}\{..{<}\mathit{length}\ (\mathit{cols}\ A)\}.\ a\ (\mathit{col}\ A\ j) * (\mathit{cols}\ A\ !\ j)\ \$\ i)$
using *assms*(*1*) *assms*(*2*) *find-first-unique*[*OF* ‹*distinct* (*cols A*)›] ‹*i* < *n*› **by** *auto*
also have ... = $(\sum j{\in}\{..{<}\mathit{length}\ (\mathit{cols}\ A)\}.\ (\mathit{cols}\ A\ !\ j)\ \$\ i * a\ (\mathit{col}\ A\ j))$ **by** (*metis mult-commute-abs*)
also have ... = $(\sum j{\in}\{..{<}\mathit{length}\ (\mathit{cols}\ A)\}.\ \mathit{row}\ A\ i\ \$\ j * a\ (\mathit{col}\ A\ j))$ **using** ‹*i* < *n*› *assms*(*1*) *assms*(*2*) **by** *auto*
finally show $(A *_v (\mathit{vec}\ \mathit{nc}\ (\lambda i.\ a\ (\mathit{col}\ A\ i)))) \$ i = \mathit{lincomb}\ a\ (\mathit{set}\ (\mathit{cols}\ A))\ \$\ i$
unfolding *lincomb-index*[*OF* ‹*i* < *n*› ‹*set* (*cols A*) ⊆ *carrier-vec n*›]
unfolding *mult-mat-vec-def scalar-prod-def*
using ‹*i* < *n*› *assms*(*1*) *atLeast0LessThan lessThan-def carrier-matD*(*1*) *index-vec sum.cong* **by** *auto*
qed

lemma (**in** *vec-space*) *lincomb-eq-mat-mult*:
assumes $A \in \mathit{carrier\text{-}mat}\ n\ \mathit{nc}$
assumes $v \in \mathit{carrier\text{-}vec}\ \mathit{nc}$
assumes *distinct* (*cols A*)
shows $\mathit{lincomb}\ (\lambda a.\ v\ \$\ \mathit{find\text{-}first}\ a\ (\mathit{cols}\ A))\ (\mathit{set}\ (\mathit{cols}\ A)) = (A *_v v)$
proof −
have $\bigwedge i.\ i < \mathit{nc} \Longrightarrow \mathit{find\text{-}first}\ (\mathit{col}\ A\ i)\ (\mathit{cols}\ A) = i$
using *assms*(*1*) *assms*(*3*) *find-first-unique* **by** *fastforce*
then have $\mathit{vec}\ \mathit{nc}\ (\lambda i.\ v\ \$\ \mathit{find\text{-}first}\ (\mathit{col}\ A\ i)\ (\mathit{cols}\ A)) = v$
using *assms*(*2*) **by** *auto*
then show *?thesis*
using *mat-mult-eq-lincomb*[**where** $a = (\lambda a.\ v\ \$\ \mathit{find\text{-}first}\ a\ (\mathit{cols}\ A))$, *OF assms*(*1*) *assms*(*3*)] **by** *auto*
qed

lemma (**in** *vec-space*) *lin-depI*:
assumes $A \in \mathit{carrier\text{-}mat}\ n\ \mathit{nc}$
assumes $v \in \mathit{carrier\text{-}vec}\ \mathit{nc}\ v \neq 0_v\ \mathit{nc}\ A *_v v = 0_v\ n$
assumes *distinct* (*cols A*)
shows *lin-dep* (*set* (*cols A*))
proof −
have *1*: *finite* (*set* (*cols A*)) **by** *simp*
have *2*: *set* (*cols A*) ⊆ *set* (*cols A*) **by** *auto*
have *3*: $(\lambda a.\ v\ \$\ \mathit{find\text{-}first}\ a\ (\mathit{cols}\ A)) \in \mathit{set}\ (\mathit{cols}\ A) \to \mathit{UNIV}$ **by** *simp*
obtain *i* **where** $v\ \$\ i \neq 0\ i < \mathit{nc}$
using ‹$v \neq 0_v$ *nc*›
by (*metis assms*(*2*) *dim-vec carrier-vecD vec-eq-iff zero-vec-def index-zero-vec*(*1*))
then have $i < \mathit{dim\text{-}col}\ A$ **using** *assms*(*1*) **by** *blast*
have *4*:*col A i* ∈ *set* (*cols A*)
using *cols-nth*[*OF* ‹*i* < *dim-col A*›] ‹*i* < *dim-col A*› *in-set-conv-nth* **by** *fastforce*
have *5*:$v\ \$\ \mathit{find\text{-}first}\ (\mathit{col}\ A\ i)\ (\mathit{cols}\ A) \neq 0$
using *find-first-unique*[*OF* ‹*distinct* (*cols A*)›] *cols-nth*[*OF* ‹*i* < *dim-col A*›] ‹*i*

```
< nc› ‹v $ i ≠ 0›
    assms(1) by auto
  have 6:lincomb (λa. v $ find-first a (cols A)) (set (cols A)) = 0_v n
    using assms(1) assms(2) assms(4) assms(5) lincomb-eq-mat-mult by auto
  show ?thesis using lin-dep-crit[OF 1 2 - 4 5 6] by metis
qed

lemma (in vec-space) lin-depE:
assumes A ∈ carrier-mat n nc
assumes lin-dep (set (cols A))
assumes distinct (cols A)
obtains v where v ∈ carrier-vec nc v ≠ 0_v nc A *_v v = 0_v n
proof −
  have finite (set (cols A)) by simp
  obtain a w where a ∈ set (cols A) → UNIV lincomb a (set (cols A)) = 0_v n w
∈ set (cols A) a w ≠ 0
    using finite-lin-dep[OF ‹finite (set (cols A))› ‹lin-dep (set (cols A))›]
    using assms(1) cols-dim carrier-matD(1) by blast
  def v == vec nc (λi. a (col A i))
  have 1:v ∈ carrier-vec nc by (simp add: v-def)
  have 2:v ≠ 0_v nc
  proof −
    obtain i where w = col A i i < length (cols A)
      by (metis ‹w ∈ set (cols A)› cols-length cols-nth in-set-conv-nth)
    have v $ i ≠ 0
      unfolding v-def
      using ‹a w ≠ 0›[unfolded ‹w = col A i›] index-vec[OF ‹i < length (cols A)›]
      assms(1) cols-length carrier-matD(2) by (metis (no-types) ‹A ∈ carrier-mat
n nc›
    ‹⋀f. vec (length (cols A)) f $ i = f i› ‹a (col A i) ≠ 0› cols-length carrier-matD(2))
    then show ?thesis using ‹i < length (cols A)› assms(1) by auto
  qed
  have 3:A *_v v = 0_v n unfolding v-def
   using ‹lincomb a (set (cols A)) = 0_v n› mat-mult-eq-lincomb[OF ‹A ∈ carrier-mat
n nc› ‹distinct (cols A)›] by auto
  show thesis using 1 2 3 by (simp add: that)
qed

lemma (in vec-space) non-distinct-low-rank:
assumes A ∈ carrier-mat n n
and ¬ distinct (cols A)
shows rank A < n
proof −
  obtain S where maximal S (λT. T ⊆ set (cols A) ∧ lin-indpt T)
    using maximal-exists[of (λT. T ⊆ set (cols A) ∧ lin-indpt T) card (set (cols
A)) {}]
     by (meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2
rev-finite-subset)
  then have card S ≤ card (set (cols A)) by (simp add: card-mono maximal-def)
```

then have *card S* $<$ *n*
using *assms*(*1*) *cols-length card-length* ‹¬ *distinct* (*cols A*)› *card-distinct carrier-matD*(*2*) *nat-less-le*
by (*metis dual-order.antisym dual-order.trans*)
then show *?thesis*
using *rank-card-indpt*[*OF* ‹*A* ∈ *carrier-mat n n*› ‹*maximal S* (λ*T*. *T* ⊆ *set* (*cols A*) ∧ *lin-indpt T*)›]
by *simp*
qed

The theorem "det non-zero ⟷ full rank" is practically proven in det_0_iff_vec_prod_zero_field, but without an actual definition of the rank.

lemma (**in** *vec-space*) *det-zero-low-rank*:
assumes *A* ∈ *carrier-mat n n*
and *det A* = *0*
shows *rank A* $<$ *n*
proof (*rule ccontr*)
assume ¬ *rank A* $<$ *n*
then have *rank A* = *n* **using** *rank-le-nc assms le-neq-implies-less* **by** *blast*
obtain *v* **where** *v* ∈ *carrier-vec n v* ≠ 0_v *n A* ∗$_v$ *v* = 0_v *n*
using *det-0-iff-vec-prod-zero-field*[*OF assms*(*1*)] *assms*(*2*) **by** *blast*
then show *False*
proof (*cases distinct* (*cols A*))
case *True*
then have *lin-indpt* (*set* (*cols A*)) **using** *full-rank-lin-indpt* **using** ‹*rank A* = *n*› *assms*(*1*) **by** *auto*
then show *False* **using** *lin-depI*[*OF assms*(*1*) ‹*v* ∈ *carrier-vec n*› ‹*v* ≠ 0_v *n*› ‹*A* ∗$_v$ *v* = 0_v *n*›] *True* **by** *blast*
next
case *False*
then show *False* **using** *non-distinct-low-rank* ‹*rank A* = *n*› ‹¬ *rank A* $<$ *n*› *assms*(*1*) **by** *blast*
qed
qed

lemma *det-identical-cols*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *ij*: *i* ≠ *j*
and *i*: *i* $<$ *n* **and** *j*: *j* $<$ *n*
and *r*: *col A i* = *col A j*
shows *det A* = *0*
using *det-identical-rows det-transpose*
by (*metis A i ij j carrier-matD*(*2*) *transpose-carrier-mat r row-transpose*)

lemma (**in** *vec-space*) *low-rank-det-zero*:
assumes *A* ∈ *carrier-mat n n*
and *det A* ≠ *0*
shows *rank A* = *n*
proof −

have *distinct* (*cols A*)
proof (*rule ccontr*)
assume ¬ *distinct* (*cols A*)
then obtain *i j* **where** *i≠j* (*cols A*) ! *i* = (*cols A*) ! *j i<length* (*cols A*) *j<length* (*cols A*)
using *distinct-conv-nth* **by** *blast*
then have *col A i* = *col A j i<n j<n* **using** *assms*(*1*) **by** *auto*
then have *det A* = *0* **using** *det-identical-cols* **using** ‹*i* ≠ *j*› *assms*(*1*) **by** *blast*
then show *False* **using** ‹*det A* ≠ *0*› **by** *auto*
qed
have ⋀*v*. *v* ∈ *carrier-vec n* ⟹ *v* ≠ 0_v *n* ⟹ *A* ∗$_v$ *v* ≠ 0_v *n*
using *det-0-iff-vec-prod-zero-field*[*OF assms*(*1*)] *assms*(*2*) **by** *auto*
then have *lin-indpt* (*set* (*cols A*)) **using** *lin-depE*[*OF assms*(*1*) - ‹*distinct* (*cols A*)›] **by** *auto*
then show *?thesis* **using** *lin-indpt-full-rank*[*OF assms*(*1*) ‹*distinct* (*cols A*)›] **by** *metis*
qed

lemma (**in** *vec-space*) *det-rank-iff*:
assumes *A* ∈ *carrier-mat n n*
shows *det A* ≠ *0* ⟷ *rank A* = *n*
using *assms det-zero-low-rank low-rank-det-zero* **by** *force*

12 Subadditivity of rank

Subadditivity is the property of rank, that rank (A + B) ¡= rank A + rank B.

lemma (**in** *module*) *lincomb-add*:
assumes *finite* (*b1* ∪ *b2*)
assumes *b1* ∪ *b2* ⊆ *carrier M*
assumes *x1* = *lincomb a1 b1 a1*∈ (*b1*→*carrier R*)
assumes *x2* = *lincomb a2 b2 a2*∈ (*b2*→*carrier R*)
assumes *x* = *x1* ⊕$_M$ *x2*
shows *lincomb* (λ*v*. (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) *v* ⊕ (λ*v*. *if v* ∈ *b2 then a2 v else* **0**) *v*) (*b1* ∪ *b2*) = *x*
proof −
have *finite* (*b1* ∪ (*b2*−*b1*)) *finite* (*b2* ∪ (*b1*−*b2*))
b1 ∪ (*b2* − *b1*) ⊆ *carrier M b2* ∪ (*b1*−*b2*) ⊆ *carrier M*
b1 ∩ (*b2* − *b1*) = {} *b2* ∩ (*b1* − *b2*) = {}
(λ*b*. $\mathbf{0}_R$) ∈ *b2* − *b1* → *carrier R* (λ*b*. $\mathbf{0}_R$) ∈ *b1* − *b2* → *carrier R*
using ‹*finite* (*b1* ∪ *b2*)› ‹*b1* ∪ *b2* ⊆ *carrier M*› ‹*a2*∈ (*b2*→*carrier R*)› **by** *auto*
have *lincomb* (λ*b*. $\mathbf{0}_R$) (*b2* − *b1*) = $\mathbf{0}_M$ *lincomb* (λ*b*. $\mathbf{0}_R$) (*b1* − *b2*) = $\mathbf{0}_M$
unfolding *lincomb-def* **using** *M.finsum-all0 assms*(*2*) *lmult-0 subset-iff*
by (*metis* (*no-types*, *lifting*) *Un-Diff-cancel2 inf-sup-aci*(*5*) *le-sup-iff*)+
then have *x1* = *lincomb* (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) (*b1* ∪ *b2*)
x2 = *lincomb* (λ*v*. *if v* ∈ *b2 then a2 v else* **0**) (*b1* ∪ *b2*)
using *lincomb-union2*[*OF* ‹*finite* (*b1* ∪ (*b2*−*b1*))› ‹*b1* ∪ (*b2* − *b1*) ⊆ *carrier M*› ‹*b1* ∩ (*b2* − *b1*) = {}› ‹*a1*∈ (*b1*→*carrier R*)› ‹(λ*b*. $\mathbf{0}_R$) ∈ *b2* − *b1* → *carrier*

R⟩]
lincomb-union2[*OF* ⟨*finite* (*b2* ∪ (*b1*−*b2*))⟩ ⟨*b2* ∪ (*b1*−*b2*) ⊆ *carrier M*⟩
⟨*b2* ∩ (*b1* − *b2*) = {}⟩ ⟨*a2*∈ (*b2*→*carrier R*)⟩ ⟨(λ*b*. $\mathbf{0}_R$) ∈ *b1* − *b2* → *carrier R*⟩]
using *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) **by** (*simp-all add*:*Un-commute*)
have (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) ∈ (*b1* ∪ *b2*) → *carrier R*
(λ*v*. *if v* ∈ *b2 then a2 v else* **0**) ∈ (*b1* ∪ *b2*) → *carrier R* **using** *assms*(*4*)
assms(*6*) **by** *auto*
show *lincomb* (λ*v*. (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) *v* ⊕ (λ*v*. *if v* ∈ *b2 then a2 v*
else **0**) *v*) (*b1* ∪ *b2*) = *x*
using *lincomb-sum*[*OF* ⟨*finite* (*b1* ∪ *b2*)⟩ ⟨*b1* ∪ *b2* ⊆ *carrier M*⟩
⟨(λ*v*. *if v* ∈ *b1 then a1 v else* **0**) ∈ (*b1* ∪ *b2*) → *carrier R*⟩ ⟨(λ*v*. *if v* ∈ *b2 then*
a2 v else **0**) ∈ (*b1* ∪ *b2*) → *carrier R*⟩]
⟨*x1* = *lincomb* (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) (*b1* ∪ *b2*)⟩ ⟨*x2* = *lincomb* (λ*v*.
if v ∈ *b2 then a2 v else* **0**) (*b1* ∪ *b2*)⟩ *assms*(*7*) **by** *blast*
qed

lemma (**in** *vectorspace*) *dim-subadditive*:
assumes *subspace K W1 V*
and *vectorspace.fin-dim K* (*vs W1*)
assumes *subspace K W2 V*
and *vectorspace.fin-dim K* (*vs W2*)
shows *vectorspace.dim K* (*vs* (*subspace-sum W1 W2*)) ≤ *vectorspace.dim K* (*vs*
W1) + *vectorspace.dim K* (*vs W2*)
proof −
have *vectorspace K* (*vs W1*) *vectorspace K* (*vs W2*) *submodule K W1 V sub-*
module K W2 V
by (*simp add*: ⟨*subspace K W1 V*⟩ ⟨*subspace K W2 V*⟩ *subspace-is-vs*)+
obtain *b1 b2* **where** *vectorspace.basis K* (*vs W1*) *b1 vectorspace.basis K* (*vs*
W2) *b2 finite b1 finite b2*
using *vectorspace.finite-basis-exists*[*OF* ⟨*vectorspace K* (*vs W1*)⟩ ⟨*vectorspace.fin-dim*
K (*vs W1*)⟩]
using *vectorspace.finite-basis-exists*[*OF* ⟨*vectorspace K* (*vs W2*)⟩ ⟨*vectorspace.fin-dim*
K (*vs W2*)⟩]
by *blast*
then have *LinearCombinations.module.gen-set K* (*vs W1*) *b1 LinearCombina-*
tions.module.gen-set K (*vs W2*) *b2*
using ⟨*vectorspace K* (*vs W1*)⟩ ⟨*vectorspace K* (*vs W2*)⟩ *vectorspace.basis-def*
by *blast*+
then have *span b1 = W1 span b2 = W2*
using *module.span-li-not-depend*(*1*) ⟨*submodule K W1 V*⟩ ⟨*submodule K W2*
V⟩
⟨*vectorspace K* (*vs W1*)⟩ ⟨*vectorspace.basis K* (*vs W1*) *b1*⟩ ⟨*vectorspace K* (*vs*
W2)⟩
⟨*vectorspace.basis K* (*vs W2*) *b2*⟩ *vectorspace.basis-def* **by** *force*+
have *W1* ⊆ *carrier V W2* ⊆ *carrier V* **using** ⟨*subspace K W1 V*⟩ ⟨*subspace K*
W2 V⟩ *subspace-def submodule-def* **by** *metis*+
have *b1* ⊆ *carrier V*
using ⟨*vectorspace.basis K* (*vs W1*) *b1*⟩ ⟨*vectorspace K* (*vs W1*)⟩ *vectorspace.basis-def*
⟨*W1* ⊆ *carrier V*⟩ **by** *fastforce*

have $b2 \subseteq$ *carrier* V
using ‹*vectorspace.basis* K (*vs* $W2$) $b2$› ‹*vectorspace* K (*vs* $W2$)› *vectorspace.basis-def* ‹$W2 \subseteq$ *carrier* V› **by** *fastforce*
have *finite* ($b1 \cup b2$) $b1 \cup b2 \subseteq$ *carrier* V
by (*simp-all add*: ‹*finite* $b1$› ‹*finite* $b2$› ‹$b2 \subseteq$ *carrier* V› ‹$b1 \subseteq$ *carrier* V›)
have *subspace-sum* $W1$ $W2 \subseteq$ *span* ($b1 \cup b2$)
proof (*rule subsetI*)
fix x **assume** $x \in$ *subspace-sum* $W1$ $W2$
obtain $x1$ $x2$ **where** $x1 \in W1$ $x2 \in W2$ $x = x1 \oplus_V x2$
using *imageE*[*OF* ‹$x \in$ *subspace-sum* $W1$ $W2$›[*unfolded submodule-sum-def*]]
by (*metis* (*no-types*, *lifting*) *BNF-Def.Collect-case-prodD split-def*)
obtain $a1$ **where** $x1 =$ *lincomb* $a1$ $b1$ $a1 \in (b1 \to$ *carrier* $K)$
using ‹*span* $b1 = W1$› *finite-span*[*OF* ‹*finite* $b1$› ‹$b1 \subseteq$ *carrier* V›] ‹$x1 \in W1$›
by *auto*
obtain $a2$ **where** $x2 =$ *lincomb* $a2$ $b2$ $a2 \in (b2 \to$ *carrier* $K)$
using ‹*span* $b2 = W2$› *finite-span*[*OF* ‹*finite* $b2$› ‹$b2 \subseteq$ *carrier* V›] ‹$x2 \in W2$›
by *auto*
obtain a **where** $x =$ *lincomb* a ($b1 \cup b2$) **using** *lincomb-add*[*OF* ‹*finite* ($b1 \cup b2$)› ‹$b1 \cup b2 \subseteq$ *carrier* V›
‹$x1 =$ *lincomb* $a1$ $b1$› ‹$a1 \in (b1 \to$ *carrier* $K)$› ‹$x2 =$ *lincomb* $a2$ $b2$› ‹$a2 \in (b2 \to$ *carrier* $K)$› ‹$x = x1 \oplus_V x2$›] **by** *blast*
then show $x \in$ *span* ($b1 \cup b2$) **using** *finite-span*[*OF* ‹*finite* ($b1 \cup b2$)› ‹($b1 \cup b2$) $\subseteq$ *carrier* V›]
using ‹$b1 \subseteq$ *carrier* V› ‹$b2 \subseteq$ *carrier* V› ‹*span* $b1 = W1$› ‹*span* $b2 = W2$› ‹$x \in$ *subspace-sum* $W1$ $W2$› *span-union-is-sum* **by** *auto*
qed
have $b1 \subseteq W1$ $b2 \subseteq W2$
using ‹*vectorspace* K (*vs* $W1$)› ‹*vectorspace* K (*vs* $W2$)› ‹*vectorspace.basis* K (*vs* $W1$) $b1$›
‹*vectorspace.basis* K (*vs* $W2$) $b2$› *vectorspace.basis-def local.carrier-vs-is-self* **by** *blast+*
then have $b1 \cup b2 \subseteq$ *subspace-sum* $W1$ $W2$ **using** ‹*submodule* K $W1$ V› ‹*submodule* K $W2$ V› *in-sum*
by (*metis assms*(1) *assms*(3) *dual-order.trans sup-least vectorspace.vsum-comm vectorspace-axioms*)
have *subspace-sum* $W1$ $W2 =$ *LinearCombinations.module.span* K (*vs* (*subspace-sum* $W1$ $W2$)) ($b1 \cup b2$)
proof (*rule subset-antisym*)
have *submodule* K (*subspace-sum* $W1$ $W2$) V **by** (*simp add*: ‹*submodule* K $W1$ V› ‹*submodule* K $W2$ V› *sum-is-submodule*)
show *subspace-sum* $W1$ $W2 \subseteq$ *LinearCombinations.module.span* K (*vs* (*subspace-sum* $W1$ $W2$)) ($b1 \cup b2$)
using *module.span-li-not-depend*(1)[*OF* ‹$b1 \cup b2 \subseteq$ *subspace-sum* $W1$ $W2$› ‹*submodule* K (*subspace-sum* $W1$ $W2$) V›]
by (*simp add*: ‹*subspace-sum* $W1$ $W2 \subseteq$ *span* ($b1 \cup b2$)›)
show *subspace-sum* $W1$ $W2 \supseteq$ *LinearCombinations.module.span* K (*vs* (*subspace-sum* $W1$ $W2$)) ($b1 \cup b2$)
using ‹$b1 \cup b2 \subseteq$ *subspace-sum* $W1$ $W2$› **by** (*metis* (*full-types*) *LinearCombinations.module.span-is-subset2*

LinearCombinations.module.submodule-is-module ‹*submodule K* (*subspace-sum W1 W2*) *V*› *local.carrier-vs-is-self submodule-def*)
qed
have *vectorspace K* (*vs* (*subspace-sum W1 W2*)) **using** *assms*(*1*) *assms*(*3*) *subspace-def sum-is-subspace vectorspace.subspace-is-vs* **by** *blast*
then have *vectorspace.dim K* (*vs* (*subspace-sum W1 W2*)) ≤ *card* (*b1* ∪ *b2*)
using *vectorspace.gen-ge-dim*[*OF* ‹*vectorspace K* (*vs* (*subspace-sum W1 W2*))› ‹*finite* (*b1* ∪ *b2*)›]
‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*›
‹*subspace-sum W1 W2* = *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)›
local.carrier-vs-is-self **by** *blast*
also have ... ≤ *card b1* + *card b2* **by** (*simp add*: *card-Un-le*)
also have ... = *vectorspace.dim K* (*vs W1*) + *vectorspace.dim K* (*vs W2*)
by (*metis* ‹*finite b1*› ‹*finite b2*› ‹*vectorspace K* (*vs W1*)› ‹*vectorspace K* (*vs W2*)›
‹*vectorspace.basis K* (*vs W1*) *b1*› ‹*vectorspace.basis K* (*vs W2*) *b2*› *vectorspace.dim-basis*)
finally show *?thesis* **by** *auto*
qed

lemma (**in** *module*) *nested-submodules*:
assumes *submodule R W M*
assumes *submodule R X M*
assumes *X* ⊆ *W*
shows *submodule R X* (*md W*)
unfolding *submodule-def*
using ‹*X* ⊆ *W*› *submodule-is-module*[*OF* ‹*submodule R W M*›] **using** ‹*submodule R X M*›[*unfolded submodule-def*] **by** *auto*

lemma (**in** *vectorspace*) *nested-subspaces*:
assumes *subspace K W V*
assumes *subspace K X V*
assumes *X* ⊆ *W*
shows *subspace K X* (*vs W*)
using *assms nested-submodules subspace-def subspace-is-vs* **by** *blast*

lemma (**in** *vectorspace*) *subspace-dim*:
assumes *subspace K X V fin-dim vectorspace.fin-dim K* (*vs X*)
shows *vectorspace.dim K* (*vs X*) ≤ *dim*
proof −
have *vectorspace K* (*vs X*) **using** *assms*(*1*) *subspace-is-vs* **by** *auto*
then obtain *b* **where** *vectorspace.basis K* (*vs X*) *b* **using** *vectorspace.finite-basis-exists*
using *assms*(*3*) **by** *blast*
then have *b* ⊆ *carrier V LinearCombinations.module.lin-indpt K* (*vs X*) *b*
using *vectorspace.basis-def*[*OF* ‹*vectorspace K* (*vs X*)›] ‹*subspace K X V*›[*unfolded subspace-def submodule-def*] **by** *auto*
then have *lin-indpt b*
by (*metis LinearCombinations.module.span-li-not-depend*(*2*) ‹*vectorspace K* (*vs X*)› ‹*vectorspace.basis K* (*vs X*) *b*›

assms(1) is-module local.carrier-vs-is-self submodule-def vectorspace.basis-def)
show *?thesis* **using** *li-le-dim(2)*[*OF* ‹*fin-dim*› ‹*b* ⊆ *carrier V*› ‹*lin-indpt b*›]
using ‹*b* ⊆ *carrier V*› ‹*lin-indpt b*› ‹*vectorspace K* (*vs X*)› ‹*vectorspace.basis K* (*vs X*) *b*› *assms(2)*
fin-dim-li-fin vectorspace.dim-basis **by** *fastforce*
qed

lemma (**in** *vectorspace*) *fin-dim-subspace-sum*:
assumes *subspace K W1 V*
assumes *subspace K W2 V*
assumes *vectorspace.fin-dim K* (*vs W1*) *vectorspace.fin-dim K* (*vs W2*)
shows *vectorspace.fin-dim K* (*vs* (*subspace-sum W1 W2*))
proof −
obtain *b1* **where** *finite b1 b1* ⊆ *W1 LinearCombinations.module.gen-set K* (*vs W1*) *b1*
using *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*
obtain *b2* **where** *finite b2 b2* ⊆ *W2 LinearCombinations.module.gen-set K* (*vs W2*) *b2*
using *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*
have *1*:*finite* (*b1* ∪ *b2*) **by** (*simp add*: ‹*finite b1*› ‹*finite b2*›)
have *2*:*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*
by (*metis* (*no-types*, *lifting*) ‹*b1* ⊆ *W1*› ‹*b2* ⊆ *W2*› *assms(1) assms(2)*
le-sup-iff subset-Un-eq vectorspace.in-sum-vs vectorspace.vsum-comm vectorspace-axioms)
have *3*:*LinearCombinations.module.gen-set K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)
proof (*rule subset-antisym*)
have *0*:*LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*) = *span* (*b1* ∪ *b2*)
using *span-li-not-depend(1)*[*OF* ‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*›] *sum-is-subspace*[*OF assms(1) assms(2)*] **by** *auto*
then show *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*) ⊆ *carrier* (*vs* (*subspace-sum W1 W2*))
using ‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*› *span-is-subset sum-is-subspace*[*OF assms(1) assms(2)*] **by** *auto*
show *carrier* (*vs* (*subspace-sum W1 W2*)) ⊆ *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)
unfolding *0*
proof
fix *x* **assume** *assumption*:*x* ∈ *carrier* (*vs* (*subspace-sum W1 W2*))
then have *x*∈*subspace-sum W1 W2* **by** *auto*
then obtain *x1 x2* **where** *x* = *x1* ⊕$_V$ *x2 x1*∈*W1 x2*∈*W2*
using *imageE*[*OF* ‹*x* ∈ *subspace-sum W1 W2*›[*unfolded submodule-sum-def*]]
by (*metis* (*no-types*, *lifting*) *BNF-Def.Collect-case-prodD split-def*)
have *x1*∈*span b1 x2*∈*span b2*
using ‹*LinearCombinations.module.span K* (*vs W1*) *b1* = *carrier* (*vs W1*)› ‹*b1* ⊆ *W1*› ‹*x1* ∈ *W1*›
‹*LinearCombinations.module.span K* (*vs W2*) *b2* = *carrier* (*vs W2*)› ‹*b2* ⊆ *W2*› ‹*x2* ∈ *W2*›
assms(1) assms(2) span-li-not-depend(1) **by** *auto*

then have *x1∈span (b1 ∪ b2) x2∈span (b1 ∪ b2)* **by** (*meson le-sup-iff set-mp span-is-monotone subsetI*)+
then show $x \in span\ (b1 \cup b2)$ **unfolding** ‹$x = x1 \oplus_V x2$›
by (*meson ‹b1 ∪ b2 ⊆ subspace-sum W1 W2› assms(1) assms(2) is-module submodule.subset subset-trans sum-is-submodule vectorspace.span-add1 vectorspace-axioms*)
qed
qed
show *?thesis* **using** *1 2 3 vectorspace.fin-dim-def*
by (*metis assms(1) assms(2) local.carrier-vs-is-self subspace-def sum-is-subspace vectorspace.subspace-is-vs*)
qed

lemma (**in** *vec-space*) *rank-subadditive*:
assumes $A \in carrier\text{-}mat\ n\ nc$
assumes $B \in carrier\text{-}mat\ n\ nc$
shows $rank\ (A + B) \leq rank\ A + rank\ B$
proof −
def $W1 == span\ (set\ (cols\ A))$
def $W2 == span\ (set\ (cols\ B))$
have $set\ (cols\ (A + B)) \subseteq subspace\text{-}sum\ W1\ W2$
proof
fix x **assume** $x \in set\ (cols\ (A + B))$
obtain i **where** $x = col\ (A + B)\ i\ i < length\ (cols\ (A + B))$
using ‹$x \in set\ (cols\ (A + B))$› *nth-find-first cols-nth find-first-le* **by** (*metis cols-length*)
then have $x = col\ A\ i + col\ B\ i$ **using** ‹$i < length\ (cols\ (A + B))$› *assms(1) assms(2)* **by** *auto*
have $col\ A\ i \in span\ (set\ (cols\ A))\ col\ B\ i \in span\ (set\ (cols\ B))$
using ‹$i < length\ (cols\ (A + B))$› *assms(1) assms(2) in-set-conv-nth*
by (*metis cols-dim cols-length cols-nth carrier-matD(1) carrier-matD(2) index-add-mat(3) span-mem*)+
then show $x \in subspace\text{-}sum\ W1\ W2$
unfolding *W1-def W2-def* ‹$x = col\ A\ i + col\ B\ i$› *submodule-sum-def* **by** *blast*
qed
have *subspace class-ring* (*subspace-sum W1 W2*) *V*
by (*metis W1-def W2-def assms(1) assms(2) cols-dim carrier-matD(1) span-is-submodule subspace-def sum-is-submodule vec-vs*)
then have $span\ (set\ (cols\ (A + B))) \subseteq subspace\text{-}sum\ W1\ W2$
by (*simp add*: ‹$set\ (cols\ (A + B)) \subseteq subspace\text{-}sum\ W1\ W2$› *span-is-subset*)
have *subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) *V* **by** (*metis assms(2) cols-dim add-carrier-mat carrier-matD(1) span-is-subspace*)
have *subspace*:*subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) (*vs* (*subspace-sum W1 W2*))
using *nested-subspaces*[*OF* ‹*subspace class-ring* (*subspace-sum W1 W2*) *V*› ‹*subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) *V*›
‹*span* (*set* (*cols* (*A* + *B*))) ⊆ *subspace-sum W1 W2*›] .
have *vectorspace.fin-dim class-ring* (*vs W1*) *vectorspace.fin-dim class-ring* (*vs*

W2)
    subspace class-ring W1 V subspace class-ring W2 V
  using *span-is-subspace W1-def W2-def assms*(*1*) *assms*(*2*) *cols-dim carrier-matD fin-dim-span-cols* **by** *auto*
 then have *fin-dim*: *vectorspace.fin-dim class-ring* (*vs* (*subspace-sum W1 W2*)) **using** *fin-dim-subspace-sum* **by** *auto*
 have *vectorspace.fin-dim class-ring* (*span-vs* (*set* (*cols* (*A* + *B*)))) **using** *assms*(*2*) *add-carrier-mat vec-space.fin-dim-span-cols* **by** *blast*
 then have *rank* (*A* + *B*) ≤ *vectorspace.dim class-ring* (*vs* (*subspace-sum W1 W2*)) **unfolding** *rank-def*
  using *vectorspace.subspace-dim*[*OF subspace-is-vs*[*OF* ‹*subspace class-ring* (*subspace-sum W1 W2*) *V*›] *subspace fin-dim*] **by** *auto*
 also have *vectorspace.dim class-ring* (*vs* (*subspace-sum W1 W2*)) ≤ *rank A* + *rank B* **unfolding** *rank-def*
   using *W1-def W2-def* ‹*subspace class-ring W1 V*› ‹*subspace class-ring W2 V*› ‹*vectorspace.fin-dim class-ring* (*vs W1*)›
  ‹*vectorspace.fin-dim class-ring* (*vs W2*)› *subspace-def vectorspace.dim-subadditive* **by** *blast*
 finally show *?thesis* **by** *auto*
qed

lemma (**in** *vec-space*) *span-zero*: *span* {*zero V*} = {*zero V*}
 by (*metis* (*no-types*, *lifting*) *empty-subsetI in-own-span span-is-submodule span-is-subset span-is-subset2 subset-antisym vectorspace.span-empty vectorspace-axioms*)

lemma (**in** *vec-space*) *dim-zero-vs*: *vectorspace.dim class-ring* (*span-vs* {}) = *0*
proof −
 have *vectorspace class-ring* (*span-vs* {}) **using** *field.field-axioms span-is-submodule submodule-is-module vectorspace-def* **by** *auto*
 have {} ⊆ *carrier-vec n* ∧ *lin-indpt* {}
  by (*metis* (*no-types*) *empty-subsetI fin-dim finite-basis-exists subset-li-is-li vec-vs vectorspace.basis-def*)
 then have *vectorspace.basis class-ring* (*span-vs* {}) {} **using** *vectorspace.basis-def*
  by (*simp add*: ‹*vectorspace class-ring* (*vs* (*span* {}))› *span-is-submodule span-li-not-depend*(*1*) *span-li-not-depend*(*2*) *vectorspace.basis-def*)
 then show *?thesis* **using** ‹*vectorspace class-ring* (*vs* (*span* {}))› *vectorspace.dim-basis* **by** *fastforce*
qed

lemma (**in** *vec-space*) *rank-0I*: *rank* (*0*$_m$ *n nc*) = *0*
proof −
 have *set* (*cols* (*0*$_m$ *n nc*)) ⊆ {*0*$_v$ *n*}
  by (*metis col-zero cols-length cols-nth in-set-conv-nth insertCI index-zero-mat*(*3*) *subsetI*)
 have *set* (*cols* (*0*$_m$ *n nc*::′*a mat*)) = {} ∨ *set* (*cols* (*0*$_m$ *n nc*)) = {*0*$_v$ *n*::′*a vec*}
  by (*meson* ‹*set* (*cols* (*0*$_m$ *n nc*)) ⊆ {*0*$_v$ *n*}› *subset-singletonD*)
 then have *span* (*set* (*cols* (*0*$_m$ *n nc*))) = {*0*$_v$ *n*}
  by (*metis* (*no-types*) *span-empty span-zero vectorspace.span-empty vectorspace-axioms*)
 then show *?thesis* **unfolding** *rank-def* ‹*span* (*set* (*cols* (*0*$_m$ *n nc*))) = {*0*$_v$ *n*}›

using *span-empty dim-zero-vs* **by** *simp*
qed

lemma (**in** *vec-space*) *rank-le-1-product-entries*:
fixes *f g*::*nat* ⇒ *'a*
assumes *A* ∈ *carrier-mat n nc*
assumes ⋀*r c*. *r*<*dim-row A* ⟹ *c*<*dim-col A* ⟹ *A* $$ (*r*,*c*) = *f r* ∗ *g c*
shows *rank A* ≤ *1*
proof −
have *set* (*cols A*) ⊆ *span* {*vec n f*}
proof
fix *v* **assume** *v* ∈ *set* (*cols A*)
then obtain *c* **where** *c* < *dim-col A v* = *col A c* **by** (*metis cols-length cols-nth in-set-conv-nth*)
have *g c* ·*v vec n f* = *v*
proof (*rule eq-vecI*)
show *dim-vec* (*g c* ·*v Matrix.vec n f*) = *dim-vec v* **using** ‹*v* = *col A c*› *assms*(*1*) **by** *auto*
fix *r* **assume** *r* < *dim-vec v*
then have *r* < *dim-vec* (*Matrix.vec n f*) **using** ‹*dim-vec* (*g c* ·*v Matrix.vec n f*) = *dim-vec v*› **by** *auto*
then have *r* < *n r* < *dim-row A***using** *index-smult-vec*(*2*) ‹*A* ∈ *carrier-mat n nc*› **by** *auto*
show (*g c* ·*v Matrix.vec n f*) $ *r* = *v* $ *r*
unfolding ‹*v* = *col A c*› *col-def index-smult-vec*(*1*)[*OF* ‹*r* < *dim-vec* (*Matrix.vec n f*)›]
index-vec[*OF* ‹*r* < *n*›] *index-vec*[*OF* ‹*r* < *dim-row A*›] **by** (*simp add*: ‹*c* < *dim-col A*› ‹*r* < *dim-row A*› *assms*(*2*))
qed
then show *v* ∈ *span* {*vec n f*} **using** *submodule.smult-closed*[*OF span-is-submodule*]
using *UNIV-I empty-subsetI insert-subset span-self dim-vec module-vec-simps*(*4*)
by *auto*
qed
have *vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*})) **using** *span-is-subspace*[*THEN subspace-is-vs*, *of* {*vec n f*}] **by** *auto*
have *submodule class-ring* (*span* {*Matrix.vec n f*}) *V* **by** (*simp add*: *span-is-submodule*)
have *subspace class-ring*(*span* (*set* (*cols A*))) (*vs* (*span* {*Matrix.vec n f*}))
using *vectorspace.span-is-subspace*[*OF* ‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*}))›, *of set* (*cols A*), *unfolded*
span-li-not-depend(*1*)[*OF* ‹*set* (*cols A*) ⊆ *span* {*vec n f*}› ‹*submodule class-ring* (*span* {*Matrix.vec n f*}) *V*›]]
‹*set* (*cols A*) ⊆ *span* {*vec n f*}› **by** *auto*
have *fin-dim*:*vectorspace.fin-dim class-ring* (*vs* (*span* {*Matrix.vec n f*}))
vectorspace.fin-dim class-ring (*vs* (*span* {*Matrix.vec n f*})(|*carrier* := *span* (*set* (*cols A*))|))
using *fin-dim-span fin-dim-span-cols* ‹*A* ∈ *carrier-mat n nc*› **by** *auto*
have *vectorspace.dim class-ring* (*vs* (*span* {*Matrix.vec n f*})) ≤ *1*
using *vectorspace.dim-le1I*[*OF* ‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n*

f}))›]
span-mem span-li-not-depend(1)[*OF* - ‹*submodule class-ring* (*span* {*Matrix.vec n f*}) *V*›] **by** *simp*
then show *?thesis* **unfolding** *rank-def* **using** *vectorspace.subspace-dim*[*OF* ‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*}))› ‹*subspace class-ring* (*span* (*set* (*cols A*))) (*vs* (*span* {*Matrix.vec n f*}))›
fin-dim(1) fin-dim(2)] **by** *simp*
qed

end

13 Missing Lemmas of Sublist

theory *DL-Missing-Sublist*
imports *Main*
begin

lemma *nths-only-one*:
assumes {*i*. *i* < *length xs* ∧ *i*∈*I*} = {*j*}
shows *nths xs I* = [*xs*!*j*]
proof −
have *set* (*nths xs I*) = {*xs*!*j*}
unfolding *set-nths* **using** *subset-antisym assms* **by** *fastforce*
moreover have *length* (*nths xs I*) = *1*
unfolding *length-nths assms* **by** *auto*
ultimately show *?thesis*
by (*metis One-nat-def length-0-conv length-Suc-conv the-elem-eq the-elem-set*)
qed

lemma *nths-replicate*:
nths (*replicate n x*) *A* = (*replicate* (*card* {*i*. *i* < *n* ∧ *i* ∈ *A*}) *x*)
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
proof (*cases n*∈*A*)
case *True*
then have *0*:(*if 0* ∈ {*j*. *j* + *length* (*replicate n x*) ∈ *A*} *then* [*x*] *else* []) = [*x*]
by *simp*
have {*i*. *i* < *Suc n* ∧ *i* ∈ *A*} = *insert n* {*i*. *i* < *n* ∧ *i* ∈ *A*} **using** *True* **by** *auto*
have *Suc* (*card* {*i*. *i* < *n* ∧ *i* ∈ *A*}) = *card* {*i*. *i* < *Suc n* ∧ *i* ∈ *A*}
unfolding ‹{*i*. *i* < *Suc n* ∧ *i* ∈ *A*} = *insert n* {*i*. *i* < *n* ∧ *i* ∈ *A*}›
using *finite-Collect-conjI*[*THEN card-insert-if*] *finite-Collect-less-nat less-irrefl-nat mem-Collect-eq* **by** *simp*
then show *?thesis* **unfolding** *replicate-Suc replicate-append-same*[*symmetric*] *nths-append Suc nths-singleton 0*

unfolding *replicate-append-same replicate-Suc*[*symmetric*] **by** *simp*
next
case *False*
then have *0*:(*if 0* ∈ {*j*. *j* + *length* (*replicate n x*) ∈ *A*} *then* [*x*] *else* []) = []
by *simp*
have {*i*. *i* < *Suc n* ∧ *i* ∈ *A*} = {*i*. *i* < *n* ∧ *i* ∈ *A*} **using** *False* **using** *le-less less-Suc-eq-le* **by** *auto*
then show *?thesis* **unfolding** *replicate-Suc replicate-append-same*[*symmetric*] *nths-append Suc nths-singleton 0*
by *simp*
qed
qed

lemma *length-nths-even*:
assumes *even* (*length xs*)
shows *length* (*nths xs* (*Collect even*)) = *length* (*nths xs* (*Collect odd*))
using *assms* **proof** (*induction length xs div 2 arbitrary*:*xs*)
case *0*
then have *length xs* = *0*
using *div-eq-0-iff length-0-conv length-greater-0-conv nat-dvd-not-less zero-not-eq-two*
by *auto*
then show *?case* **by** *simp*
next
case (*Suc l xs*)
then have *length-drop2*: *length* (*nths* (*drop 2 xs*) (*Collect even*)) = *length* (*nths* (*drop 2 xs*) {*a*. *odd a*}) **by** *simp*

have *length* (*take 2 xs*) = *2* **using** *Suc.hyps*(*2*) **by** *auto*
then have *plus-odd*: {*j*. *j* + *length* (*take 2 xs*) ∈ *Collect odd*} = *Collect odd* **and**
plus-even: {*j*. *j* + *length* (*take 2 xs*) ∈ *Collect even*} = *Collect even* **by**
simp-all
have *nths-take2*: *nths* (*take 2 xs*) (*Collect even*) = [*take 2 xs* ! *0*] *nths* (*take 2 xs*) (*Collect odd*) = [*take 2 xs* ! *1*]
using ‹*length* (*take 2 xs*) = *2*› *less-2-cases nths-only-one*[*of take 2 xs Collect even 0*]
nths-only-one[*of take 2 xs Collect odd 1*]
by *fastforce*+
then have *length* (*nths* (*take 2 xs* @ *drop 2 xs*) (*Collect even*))
= *length* (*nths* (*take 2 xs* @ *drop 2 xs*) {*a*. *odd a*})
unfolding *nths-append length-append plus-odd plus-even nths-take2 length-drop2*
by *auto*
then show *?case* **using** *append-take-drop-id*[*of 2 xs*] **by** *simp*
qed

lemma *nths-map*:
nths (*map f xs*) *A* = *map f* (*nths xs A*)
proof (*induction xs arbitrary*:*A*)
case *Nil*
then show *?case* **by** *simp*

next
case (*Cons x xs*)
then show *?case*
by (*simp add*: *nths-Cons*)
qed

14 Pick

fun *pick* :: *nat set* ⇒ *nat* ⇒ *nat* **where**
pick S 0 = (*LEAST a*. *a*∈*S*) |
pick S (*Suc n*) = (*LEAST a*. *a*∈*S* ∧ *a* > *pick S n*)

lemma *pick-in-set-inf*:
assumes *infinite S*
shows *pick S n* ∈ *S*
proof (*cases n*)
show *n* = *0* ⟹ *pick S n* ∈ *S*
unfolding *pick.simps* **using** ‹*infinite S*› *LeastI pick.simps*(*1*) **by** (*metis Collect-mem-eq not-finite-existsD*)
next
fix *n'* **assume** *n* = *Suc n'*
obtain *a* **where** *a*∈*S* ∧ *a* > *pick S n'* **using** *assms* **by** (*metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff*)
show *pick S n* ∈ *S* **unfolding** ‹*n* = *Suc n'*› *pick.simps*(*2*)
using *LeastI*[*of* λ*a*. *a* ∈ *S* ∧ *pick S n'* < *a a*, *OF* ‹*a*∈*S* ∧ *a* > *pick S n'*›] **by** *blast*
qed

lemma *pick-mono-inf*:
assumes *infinite S*
shows *m* < *n* ⟹ *pick S m* < *pick S n*
using *assms* **proof** (*induction n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
then obtain *a* **where** *a* ∈ *S* ∧ *pick S n* < *a* **by** (*metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff*)
then have *pick S n* < *pick S* (*Suc n*) **unfolding** *pick.simps*
using *LeastI*[*of* λ*a*. *a* ∈ *S* ∧ *pick S n* < *a a*, *OF* ‹*a*∈*S* ∧ *a* > *pick S n*›] **by** *simp*
then show *?case* **using** *Suc.IH Suc.prems*(*1*) *assms dual-order.strict-trans less-Suc-eq* **by** *auto*
qed

lemma *pick-eq-iff-inf*:
assumes *infinite S*
shows *x* = *y* ⟷ *pick S x* = *pick S y*
by (*metis assms nat-neq-iff pick-mono-inf*)

lemma *card-le-pick-inf*:
assumes *infinite S*
and *pick S n* $\geq$ *i*
shows *card* $\{a{\in}S.\ a < i\} \leq n$
using *assms* **proof** (*induction n arbitrary*:*i*)
case *0*
then show *?case* **unfolding** *pick.simps* **using** *not-less-Least*
by (*metis* (*no-types*, *lifting*) *Collect-empty-eq card-0-eq card-ge-0-finite dual-order.strict-trans1 leI le-0-eq*)
next
case (*Suc n*)
then show *?case*
proof −
have *card* $\{a \in S.\ a < pick\ S\ n\} \leq n$ **using** *Suc* **by** *blast*
have $\{a \in S.\ a < i\} \subseteq \{a \in S.\ a < pick\ S\ (Suc\ n)\}$ **using** *Suc.prems*(*2*) **by** *auto*
have $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = \{a \in S.\ a < pick\ S\ n\} \cup \{pick\ S\ n\}$
apply (*rule subset-antisym*; *rule subsetI*)
using *not-less-Least UnCI mem-Collect-eq nat-neq-iff singleton-conv pick-mono-inf*[*OF Suc.prems*(*1*), *of n Suc n*] *pick-in-set-inf*[*OF Suc.prems*(*1*), *of n*] **by** *fastforce*+
then have *card* $\{a \in S.\ a < i\} \leq$ *card* $\{a \in S.\ a < pick\ S\ n\}$ + *card* $\{pick\ S\ n\}$
using *card-Un-disjoint card-mono*[*OF* - ‹$\{a \in S.\ a < i\} \subseteq \{a \in S.\ a < pick\ S\ (Suc\ n)\}$›] **by** *simp*
then show *?thesis* **using** ‹*card* $\{a \in S.\ a < pick\ S\ n\} \leq n$› **by** *auto*
qed
qed

lemma *card-pick-inf*:
assumes *infinite S*
shows *card* $\{a{\in}S.\ a < pick\ S\ n\} = n$
using *assms* **proof** (*induction n*)
case *0*
then show *?case* **unfolding** *pick.simps* **using** *not-less-Least* **by** *auto*
next
case (*Suc n*)
then show *card* $\{a{\in}S.\ a < pick\ S\ (Suc\ n)\} = Suc\ n$
proof −
have $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = \{a \in S.\ a < pick\ S\ n\} \cup \{pick\ S\ n\}$
apply (*rule subset-antisym*; *rule subsetI*)
using *not-less-Least UnCI mem-Collect-eq nat-neq-iff singleton-conv pick-mono-inf*[*OF Suc.prems*, *of n Suc n*] *pick-in-set-inf*[*OF Suc.prems*, *of n*] **by** *fastforce*+
then have *card* $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = card\ \{a \in S.\ a < pick\ S\ n\}$ + *card* $\{pick\ S\ n\}$ **using** *card-Un-disjoint* **by** *auto*
then show *?thesis* **by** (*metis One-nat-def Suc-eq-plus1 Suc card-empty card-insert-if empty-iff finite.emptyI*)

```
  qed
qed

lemma
assumes n < card S
shows
  pick-in-set-le:pick S n ∈ S and
  card-pick-le: card {a∈S. a < pick S n} = n and
  pick-mono-le: m < n ⟹ pick S m < pick S n
using assms proof (induction n)
  assume 0 < card S
  then obtain x where x∈S by fastforce
  then show pick S 0 ∈ S unfolding pick.simps by (meson LeastI)
  then show card {a ∈ S. a < pick S 0} = 0 using not-less-Least by auto
  show m < 0 ⟹  pick S m < pick S 0 by auto
next
  fix n
  assume n < card S ⟹ pick S n ∈ S
    and n < card S ⟹ card {a ∈ S. a < pick S n} = n
    and Suc n < card S
    and m < n ⟹ n < card S ⟹ pick S m < pick S n
  then have card {a ∈ S. a < pick S n} = n pick S n ∈ S by linarith+
  have card {a ∈ S. a > pick S n} > 0
  proof −
    have S = {a ∈ S. a < pick S n} ∪ {a ∈ S. a ≥ pick S n} by fastforce
    then have card {a ∈ S. a ≥ pick S n} > 1
      using ‹Suc n < card S› ‹card {a ∈ S. a < pick S n} = n›
      card-Un-le[of {a ∈ S. a < pick S n} {a ∈ S. pick S n ≤ a}] by force
    then have 0:{a ∈ S. a ≥ pick S n} ⊆ {pick S n} ∪ {a ∈ S. a > pick S n}
by auto
    have 1:finite ({pick S n} ∪ {a ∈ S. pick S n < a})
      unfolding finite-Un using Collect-mem-eq assms card-infinite conjI by force
    have 1 < card {pick S n} + card {a ∈ S. pick S n < a}
      using card-mono[OF 1 0] card-Un-le[of {pick S n} {a ∈ S. a > pick S n}]
‹card {a ∈ S. a ≥ pick S n} > 1›
      by linarith
    then show ?thesis by simp
  qed
  then show pick S (Suc n) ∈ S unfolding pick.simps
     by (metis (no-types, lifting) Collect-empty-eq LeastI card-0-eq card-infinite
less-numeral-extra(3))
  have pick S (Suc n) > pick S n
    by (metis (no-types, lifting) pick.simps(2) ‹card {a ∈ S. a > pick S n} > 0›
Collect-empty-eq LeastI card-0-eq card-infinite less-numeral-extra(3))
  then show m < Suc n ⟹ pick S m < pick S (Suc n)
    using ‹m < n ⟹ n < card S ⟹ pick S m < pick S n›
    using ‹Suc n < card S› dual-order.strict-trans less-Suc-eq by auto
  then show card {a∈S. a < pick S (Suc n)} = Suc n
  proof −
```

```
    have {a ∈ S. a < pick S (Suc n)} = {a ∈ S. a < pick S n} ∪ {pick S n}
      apply (rule subset-antisym; rule subsetI)
       using pick.simps not-less-Least ‹pick S (Suc n) > pick S n› ‹pick S n ∈ S›
by fastforce+
    then have card {a ∈ S. a < pick S (Suc n)} = card {a ∈ S. a < pick S n}
+ card {pick S n}  using card-Un-disjoint by auto
    then show ?thesis by (metis One-nat-def Suc-eq-plus1 ‹card {a ∈ S. a < pick
S n} = n› card-empty card-insert-if empty-iff finite.emptyI)
  qed
qed

lemma card-le-pick-le:
assumes n < card S
and pick S n ≥ i
shows card {a∈S. a < i} ≤ n
using assms proof (induction n arbitrary:i)
  case 0
  then show ?case unfolding pick.simps using not-less-Least
   by (metis (no-types, lifting) Collect-empty-eq card-0-eq card-ge-0-finite dual-order.strict-trans1
leI le-0-eq)
next
  case (Suc n)
  have card {a ∈ S. a < pick S n} ≤ n using Suc by (simp add: less-eq-Suc-le
nat-less-le)
  have {a ∈ S. a < i} ⊆ {a ∈ S. a < pick S (Suc n)} using Suc.prems(2) by
auto
  have {a ∈ S. a < pick S (Suc n)} = {a ∈ S. a < pick S n} ∪ {pick S n}
    apply (rule subset-antisym; rule subsetI)
   using pick.simps not-less-Least  pick-mono-le[OF Suc.prems(1), of n, OF lessI]
pick-in-set-le[of n S] Suc by fastforce+
  then have card {a ∈ S. a < i} ≤ card {a ∈ S. a < pick S n} + card {pick S
n}
    using card-Un-disjoint  card-mono[OF - ‹{a ∈ S. a < i} ⊆ {a ∈ S. a < pick
S (Suc n)}›] by simp
  then show ?case using ‹card {a ∈ S. a < pick S n} ≤ n›  by auto
qed

lemma
assumes n < card S ∨ infinite S
shows
  pick-in-set:pick S n ∈ S and
  card-le-pick: i ≤ pick S n ==> card {a∈S. a < i} ≤ n and
  card-pick: card {a∈S. a < pick S n} = n and
  pick-mono: m < n ⟹ pick S m < pick S n
   using assms pick-in-set-inf pick-in-set-le card-pick-inf card-pick-le card-le-pick-le
card-le-pick-inf
    pick-mono-inf pick-mono-le by auto

lemma pick-card:
```

pick I (*card* {*a*∈*I*. *a* < *i*}) = (*LEAST a*. *a*∈*I* ∧ *a* ≥ *i*)
proof (*induction i*)
case *0*
then show *?case* **by** (*simp add*: *pick-in-set-le*)
next
case (*Suc i*)
then show *?case*
proof (*cases i*∈*I*)
case *True*
then have *1*:*pick I* (*card* {*a*∈*I*. *a* < *i*}) = *i* **by** (*metis* (*mono-tags*, *lifting*) *Least-equality Suc.IH order-refl*)
have {*a* ∈ *I*. *a* < *Suc i*} = {*a* ∈ *I*. *a* < *i*} ∪ {*i*} **using** *True* **by** *auto*
then have *2*:*card* {*a* ∈ *I*. *a* < *Suc i*} = *Suc* (*card* {*a* ∈ *I*. *a* < *i*}) **by** *auto*
then show *?thesis* **unfolding** *2 pick.simps 1* **using** *Suc-le-eq* **by** *auto*
next
case *False*
then have *1*:{*a* ∈ *I*. *a* < *Suc i*} = {*a* ∈ *I*. *a* < *i*} **using** *Collect-cong less-Suc-eq* **by** *auto*
have *2*:⋀*a*. (*a* ∈ *I* ∧ *Suc i* ≤ *a*) = (*a* ∈ *I* ∧ *i* ≤ *a*) **using** *False Suc-leD le-less-Suc-eq not-le* **by** *blast*
then show *?thesis* **unfolding** *1 2* **using** *Suc.IH* **by** *blast*
qed
qed

lemma *pick-card-in-set*: *i*∈*I* ⟹ *pick I* (*card* {*a*∈*I*. *a* < *i*}) = *i*
unfolding *pick-card* **using** *Least-equality order-refl* **by** (*metis* (*no-types*, *lifting*))

15 Sublist

lemma *nth-nths-card*:
assumes *j*<*length xs*
and *j*∈*J*
shows *nths xs J* ! *card* {*j0*. *j0* < *j* ∧ *j0* ∈ *J*} = *xs*!*j*
using *assms* **proof** (*induction xs rule*:*rev-induct*)
case *Nil*
then show *?case* **using** *gr-implies-not0 list.size*(*3*) **by** *auto*
next
case (*snoc x xs*)
then show *?case*
proof (*cases j* < *length xs*)
case *True*
have {*j0*. *j0* < *j* ∧ *j0* ∈ *J*} ⊂ {*i*. *i* < *length xs* ∧ *i* ∈ *J*}
using *True snoc.prems*(*2*) **by** *auto*
then have *card* {*j0*. *j0* < *j* ∧ *j0* ∈ *J*} < *length* (*nths xs J*) **unfolding** *length-nths*
using *psubset-card-mono*[*of* {*i*. *i* < *length xs* ∧ *i* ∈ *J*}] **by** *simp*
then show *?thesis* **unfolding** *nths-append nth-append* **by** (*simp add*: *True snoc.IH snoc.prems*(*2*))
next

```
    case False
    then have length xs = j
      using length-append-singleton less-antisym snoc.prems(1) by auto
    then show ?thesis unfolding nths-append nth-append length-nths ‹length xs =
j›
      by (simp add: snoc.prems(2))
  qed
qed

lemma pick-reduce-set:
assumes i<card {a. a<m ∧ a∈I}
shows pick I i = pick {a. a < m ∧ a ∈ I} i
using assms proof (induction i)
  let ?L = LEAST a. a ∈ {a. a < m ∧ a ∈ I}
  case 0
  then have {a. a < m ∧ a ∈ I} ≠ {} using card-empty less-numeral-extra(3)
by fastforce
  then have ?L ∈ I ?L < m by (metis (mono-tags, lifting) Collect-empty-eq LeastI
mem-Collect-eq)+
  have ⋀x. x ∈ {a. a < m ∧ a ∈ I} ⟹ ?L ≤ x by (simp add: Least-le)
  have ⋀x. x ∈ I ⟹ ?L ≤ x
    by (metis (mono-tags) ‹?L < m› ‹⋀x. x ∈ {a. a < m ∧ a ∈ I} ⟹ ?L ≤ x›
dual-order.strict-trans2 le-cases mem-Collect-eq)
  then show ?case unfolding pick.simps using Least-equality[of λx. x∈I, OF
‹?L ∈ I›] by blast
next
  case (Suc i)
  let ?L = LEAST x. x ∈ {a. a < m ∧ a ∈ I} ∧ pick I i < x
  have 0:pick {a. a < m ∧ a ∈ I} i = pick I i using Suc-lessD Suc by linarith
  then have ?L ∈ {a. a < m ∧ a ∈ I} pick I i < ?L
    using LeastI[of λa. a ∈ {a. a < m ∧ a ∈ I} ∧ pick I i < a] using Suc.prems
pick-in-set-le pick-mono-le by fastforce+
  then have ?L ∈ I by blast
  show ?case unfolding pick.simps 0 using Least-equality[of λa. a ∈ I ∧ pick I
i < a ?L]
    by (metis (no-types, lifting) Least-le ‹?L ∈ {a. a < m ∧ a ∈ I}› ‹pick I i <
?L› mem-Collect-eq not-le not-less-iff-gr-or-eq order.trans)
qed

lemma nth-nths:
assumes i<card {i. i<length xs ∧ i∈I}
shows nths xs I ! i = xs ! pick I i
proof −
  have {a ∈ {i. i < length xs ∧ i ∈ I}. a < pick {i. i < length xs ∧ i ∈ I} i}
        = {a.  a < pick {i. i < length xs ∧ i ∈ I} i ∧ a ∈ I}
    using assms pick-in-set by fastforce
  then have card {a. a < pick {i. i < length xs ∧ i ∈ I} i ∧ a ∈ I} = i
    using card-pick-le[OF assms] by simp
  then have nths xs I ! i = xs ! pick {i. i < length xs ∧ i ∈ I} i
```

using *nth-nths-card*[**where** $j = pick\ \{i.\ i < length\ xs \wedge i \in I\}\ i$, *of xs I*]
assms pick-in-set pick-in-set **by** *auto*
then show *?thesis* **using** *pick-reduce-set* **using** *assms* **by** *auto*
qed

lemma *pick-UNIV*: $pick\ UNIV\ j = j$
by (*induction j*, *simp*, *metis* (*no-types*, *lifting*) *LeastI pick.simps*(2) *Suc-mono UNIV-I less-Suc-eq not-less-Least*)

lemma *pick-le*:
assumes $n < card\ \{a.\ a < i \wedge a \in S\}$
shows $pick\ S\ n < i$
proof −
have 0:$\{a \in \{a.\ a < i \wedge a \in S\}.\ a < i\} = \{a.\ a < i \wedge a \in S\}$ **by** *blast*
show *?thesis* **apply** (*rule ccontr*)
using *card-le-pick-le*[*OF assms*, *unfolded pick-reduce-set*[*OF assms*, *symmetric*], *of i*, *unfolded 0*]
assms not-less not-le **by** *blast*
qed

lemma *prod-list-complementary-nthss*:
fixes $f :: 'a \Rightarrow 'b$::*comm-monoid-mult*
shows $prod\text{-}list\ (map\ f\ xs) = prod\text{-}list\ (map\ f\ (nths\ xs\ A)) * prod\text{-}list\ (map\ f\ (nths\ xs\ (-A)))$
proof (*induction xs rule*:*rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
show *?case* **unfolding** *map-append prod-list.append nths-append nths-singleton snoc*
by (*cases* (*length xs*)∈*A*; *simp*;*metis mult.assoc mult.commute*)
qed

lemma *nths-zip*: $nths\ (zip\ xs\ ys)\ I = zip\ (nths\ xs\ I)\ (nths\ ys\ I)$
proof (*rule nth-equalityI*)
show $length\ (nths\ (zip\ xs\ ys)\ I) = length\ (zip\ (nths\ xs\ I)\ (nths\ ys\ I))$
proof (*cases length xs* ≤ *length ys*)
case *True*
then have $\{i.\ i < length\ xs \wedge i \in I\} \subseteq \{i.\ i < length\ ys \wedge i \in I\}$ **by** (*simp add*: *Collect-mono less-le-trans*)
then have $card\ \{i.\ i < length\ xs \wedge i \in I\} \leq card\ \{i.\ i < length\ ys \wedge i \in I\}$
by (*metis* (*mono-tags*, *lifting*) *card-mono finite-nat-set-iff-bounded mem-Collect-eq*)
then show *?thesis* **unfolding** *length-nths length-zip* **using** *True* **using** *min-def* **by** *linarith*
next
case *False*
then have $\{i.\ i < length\ ys \wedge i \in I\} \subseteq \{i.\ i < length\ xs \wedge i \in I\}$ **by** (*simp add*: *Collect-mono less-le-trans*)

then have *card* $\{i.\ i < length\ ys \wedge i \in I\} \leq card\ \{i.\ i < length\ xs \wedge i \in I\}$
by (*metis* (*mono-tags*, *lifting*) *card-mono finite-nat-set-iff-bounded mem-Collect-eq*)
then show *?thesis* **unfolding** *length-nths length-zip* **using** *False* **using** *min-def*
by *linarith*
qed
show $\forall i{<}length\ (nths\ (zip\ xs\ ys)\ I).\ nths\ (zip\ xs\ ys)\ I\ !\ i = zip\ (nths\ xs\ I)\ (nths\ ys\ I)\ !\ i$
proof (*rule allI*; *rule impI*)
fix *i* **assume** $i < length\ (nths\ (zip\ xs\ ys)\ I)$
then have $i < length\ (nths\ xs\ I)$ $i < length\ (nths\ ys\ I)$
by (*simp-all add*: ‹*length* (*nths* (*zip xs ys*) *I*) = *length* (*zip* (*nths xs I*) (*nths ys I*))›)
show *nths* (*zip xs ys*) *I* ! *i* = *zip* (*nths xs I*) (*nths ys I*) ! *i*
unfolding *nth-nths*[*OF* ‹*i* < *length* (*nths* (*zip xs ys*) *I*)›[*unfolded length-nths*]]
unfolding *nth-zip*[*OF* ‹*i* < *length* (*nths xs I*)› ‹*i* < *length* (*nths ys I*)›]
unfolding *nth-zip*[*OF pick-le*[*OF* ‹*i* < *length* (*nths xs I*)›[*unfolded length-nths*]]
pick-le[*OF* ‹*i* < *length* (*nths ys I*)›[*unfolded length-nths*]]]
by (*metis* (*full-types*) ‹*i* < *length* (*nths xs I*)› ‹*i* < *length* (*nths ys I*)› *length-nths nth-nths*)
qed
qed

16 weave

definition *weave* :: *nat set* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
weave A xs ys = *map* (λ*i*. *if i*∈*A then xs*!(*card* {*a*∈*A*. *a*<*i*}) *else ys*!(*card* {*a*∈−*A*. *a*<*i*})) [*0*..<*length xs* + *length ys*]

lemma *length-weave*:
shows *length* (*weave A xs ys*) = *length xs* + *length ys*
unfolding *weave-def length-map* **by** *simp*

lemma *nth-weave*:
assumes *i* < *length* (*weave A xs ys*)
shows *weave A xs ys* ! *i* = (*if i*∈*A then xs*!(*card* {*a*∈*A*. *a*<*i*}) *else ys*!(*card* {*a*∈−*A*. *a*<*i*}))
proof −
have *i* < *length xs* + *length ys* **using** *length-weave* **using** *assms* **by** *metis*
then have *i* < *length* [*0*..<*length xs* + *length ys*] **by** *auto*
then have [*0*..<*length xs* + *length ys*] ! *i* = *i*
by (*metis* ‹*i* < *length xs* + *length ys*› *add.left-neutral nth-upt*)
then show *?thesis*
unfolding *weave-def nth-map*[*OF* ‹*i* < *length* [*0*..<*length xs* + *length ys*]›] **by** *presburger*
qed

lemma *weave-append1*:
assumes *length xs* + *length ys* ∈ *A*
assumes *length xs* = *card* {*a*∈*A*. *a* < *length xs* + *length ys*}

shows *weave A (xs @ [x]) ys = weave A xs ys @ [x]*
proof (*rule nth-equalityI*)
show *length (weave A (xs @ [x]) ys) = length (weave A xs ys @ [x])*
unfolding *weave-def length-map* **by** *simp*
show ∀ *i*<*length (weave A (xs @ [x]) ys). weave A (xs @ [x]) ys ! i = (weave A xs ys @ [x]) ! i*
proof (*rule allI*, *rule impI*)
fix *i* **assume** *i < length (weave A (xs @ [x]) ys)*
show *weave A (xs @ [x]) ys ! i = (weave A xs ys @ [x]) ! i*
proof (*cases i = length xs + length ys*)
case *True*
then have *(weave A xs ys @ [x]) ! i = x* **using** *length-weave* **by** (*metis nth-append-length*)
have *card {a ∈ A. a < i} = length xs* **using** *assms(2) True* **by** *auto*
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹*i < length (weave A (xs @ [x]) ys)*›]
‹*(weave A xs ys @ [x]) ! i = x*› **using** *True assms(1)* **by** *simp*
next
case *False*
have *i < length (weave A xs ys)* **using** ‹*i < length (weave A (xs @ [x]) ys)*›
‹*length (weave A (xs @ [x]) ys) = length (weave A xs ys @ [x])*› *length-append-singleton length-weave less-antisym False* **by** *fastforce*
then have *(weave A xs ys @ [x]) ! i = (weave A xs ys) ! i* **by** (*simp add*: *nth-append*)
{
assume *i∈A*
have *i*<*length xs + length ys* **by** (*metis* ‹*i < length (weave A xs ys)*› *length-weave*)
then have *{a ∈ A. a < i} ⊂ {a∈A. a < length xs + length ys}*
using *assms(1)* ‹*i*<*length xs + length ys*› ‹*i∈A*› **by** *auto*
then have *card {a ∈ A. a < i} < card {a∈A. a < length xs + length ys}*
using *psubset-card-mono*[*of* *{a∈A. a < length xs + length ys}* *{a ∈ A. a < i}*] **by** *simp*
then have *(xs @ [x]) ! card {a ∈ A. a < i} = xs ! card {a ∈ A. a < i}*
by (*metis* (*no-types*, *lifting*) *assms(2) nth-append*)
}
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹*i < length (weave A (xs @ [x]) ys)*›]
‹*(weave A xs ys @ [x]) ! i = (weave A xs ys) ! i*› *nth-weave*[*OF* ‹*i < length (weave A xs ys)*›]
by *simp*
qed
qed
qed

lemma *weave-append2*:
assumes *length xs + length ys ∉ A*
assumes *length ys = card {a∈−A. a < length xs + length ys}*
shows *weave A xs (ys @ [y]) = weave A xs ys @ [y]*

proof (*rule nth-equalityI*)
show *length* (*weave A xs* (*ys* @ [*y*])) = *length* (*weave A xs ys* @ [*y*])
unfolding *weave-def length-map* **by** *simp*
show ∀ *i*<*length* (*weave A xs* (*ys* @ [*y*])). *weave A xs* (*ys* @ [*y*]) ! *i* = (*weave A xs ys* @ [*y*]) ! *i*
proof (*rule allI*, *rule impI*)
fix *i* **assume** *i* < *length* (*weave A xs* (*ys* @ [*y*]))
show *weave A xs* (*ys* @ [*y*]) ! *i* = (*weave A xs ys* @ [*y*]) ! *i*
proof (*cases i* = *length xs* + *length ys*)
case *True*
then have (*weave A xs ys* @ [*y*]) ! *i* = *y* **using** *length-weave* **by** (*metis nth-append-length*)
have *card* {*a* ∈ −*A*. *a* < *i*} = *length ys* **using** *assms*(*2*) *True* **by** *auto*
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹*i* < *length* (*weave A xs* (*ys* @ [*y*]))›]
‹(*weave A xs ys* @ [*y*]) ! *i* = *y*› **using** *True assms*(*1*) **by** *simp*
next
case *False*
have *i* < *length* (*weave A xs ys*) **using** ‹*i* < *length* (*weave A xs* (*ys* @ [*y*]))›
‹*length* (*weave A xs* (*ys* @ [*y*])) = *length* (*weave A xs ys* @ [*y*])› *length-append-singleton*
length-weave less-antisym False **by** *fastforce*
then have (*weave A xs ys* @ [*y*]) ! *i* = (*weave A xs ys*) ! *i* **by** (*simp add*: *nth-append*)
{
assume *i*∉*A*
have *i*<*length xs* + *length ys* **by** (*metis* ‹*i* < *length* (*weave A xs ys*)› *length-weave*)
then have {*a* ∈ −*A*. *a* < *i*} ⊂ {*a*∈−*A*. *a* < *length xs* + *length ys*}
using *assms*(*1*) ‹*i*<*length xs* + *length ys*› ‹*i*∉*A*› **by** *auto*
then have *card* {*a* ∈ −*A*. *a* < *i*} < *card* {*a*∈−*A*. *a* < *length xs* + *length ys*}
using *psubset-card-mono*[*of* {*a*∈−*A*. *a* < *length xs* + *length ys*} {*a* ∈ −*A*. *a* < *i*}] **by** *simp*
then have (*ys* @ [*y*]) ! *card* {*a* ∈ −*A*. *a* < *i*} = *ys* ! *card* {*a* ∈ −*A*. *a* < *i*}
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *nth-append*)
}
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹*i* < *length* (*weave A xs* (*ys* @ [*y*]))›]
‹(*weave A xs ys* @ [*y*]) ! *i* = (*weave A xs ys*) ! *i*› *nth-weave*[*OF* ‹*i* < *length* (*weave A xs ys*)›]
by *simp*
qed
qed
qed

lemma *nths-nth*:
assumes *n*∈*A n*<*length xs*
shows *nths xs A* ! (*card* {*i*. *i*<*n* ∧ *i*∈*A*}) = *xs* ! *n*
using *assms* **proof** (*induction xs rule*:*rev-induct*)

```
  case Nil
  then show ?case by simp
next
  case (snoc x xs)
  then show ?case
  proof (cases n = length xs)
    case True
    then show ?thesis unfolding nths-append[of xs [x] A] nth-append
      using length-nths[of xs A] nths-singleton snoc.prems(1) by auto
  next
    case False
    then have n < length xs using snoc by auto
    then have 0:nths xs A ! card {i. i < n ∧ i ∈ A} = xs ! n using snoc by auto

    have {i. i < n ∧ i ∈ A} ⊂ {i. i < length xs ∧ i ∈ A} using ‹n < length xs›
snoc by force
   then have card {i. i < n ∧ i ∈ A} < length (nths xs A) unfolding length-nths
     by (simp add: psubset-card-mono)
    then show ?thesis unfolding nths-append[of xs [x] A] nth-append using 0
     by (simp add: ‹n < length xs›)
  qed
qed

lemma list-all2-nths:
assumes list-all2 P (nths xs A) (nths ys A)
and     list-all2 P (nths xs (−A)) (nths ys (−A))
shows list-all2 P xs ys
proof −
  have length xs = length ys
  proof (rule ccontr; cases length xs < length ys)
    case True
    then show False
    proof (cases length xs ∈ A)
      case False
      have {i. i < length xs ∧ i ∈ − A} ⊂ {i. i < length ys ∧ i ∈ − A}
        using False ‹length xs < length ys› by force
      then have length (nths ys (−A)) > length (nths xs (−A))
        unfolding length-nths by (simp add: psubset-card-mono)
       then show False using assms(2) list-all2-lengthD not-less-iff-gr-or-eq by
blast
    next
      case True
      have {i. i < length xs ∧ i ∈ A} ⊂ {i. i < length ys ∧ i ∈ A}
        using True ‹length xs < length ys› by force
      then have length (nths ys A) > length (nths xs A)
        unfolding length-nths by (simp add: psubset-card-mono)
       then show False using assms(1) list-all2-lengthD not-less-iff-gr-or-eq by
blast
    qed
```

```
  next
    assume length xs ≠ length ys
    case False
    then have length xs > length ys using ‹length xs ≠ length ys› by auto
    then show False
    proof (cases length ys ∈ A)
      case False
      have {i. i < length ys ∧ i ∈ −A} ⊂ {i. i < length xs ∧ i ∈ −A}
        using False ‹length xs > length ys› by force
      then have length (nths xs (−A)) > length (nths ys (−A))
        unfolding length-nths by (simp add: psubset-card-mono)
    then show False using assms(2) list-all2-lengthD dual-order.strict-implies-not-eq
by blast
    next
      case True
      have {i. i < length ys ∧ i ∈ A} ⊂ {i. i < length xs ∧ i ∈ A}
        using True ‹length xs > length ys› by force
      then have length (nths xs A) > length (nths ys A)
        unfolding length-nths by (simp add: psubset-card-mono)
    then show False using assms(1) list-all2-lengthD dual-order.strict-implies-not-eq
by blast
    qed
  qed

  have ⋀n. n < length xs ⟹ P (xs ! n) (ys ! n)
  proof −
    fix n assume n < length xs
    then have n < length ys using ‹length xs = length ys› by auto
    then show P (xs ! n) (ys ! n)
    proof (cases n∈A)
      case True
      have {i. i < n ∧ i ∈ A} ⊂ {i. i < length xs ∧ i ∈ A} using ‹n < length xs›
‹n∈A› by force
    then have card {i. i < n ∧ i ∈ A} < length (nths xs A) unfolding length-nths
        by (simp add: psubset-card-mono)
      show ?thesis using nths-nth[OF ‹n∈A› ‹n < length xs›] nths-nth[OF ‹n∈A›
‹n < length ys›]
        list-all2-nthD[OF assms(1), of card {i. i < n ∧ i ∈ A}] length-nths
        by (simp add: ‹card {i. i < n ∧ i ∈ A} < length (nths xs A)›)
    next
      case False then have n∈−A by auto
      have {i. i < n ∧ i ∈ −A} ⊂ {i. i < length xs ∧ i ∈ −A} using ‹n < length
xs› ‹n∈−A› by force
      then have card {i. i < n ∧ i ∈ −A} < length (nths xs (−A)) unfolding
length-nths
        by (simp add: psubset-card-mono)
    show ?thesis using nths-nth[OF ‹n∈−A› ‹n < length xs›] nths-nth[OF ‹n∈−A›
‹n < length ys›]
        list-all2-nthD[OF assms(2), of card {i. i < n ∧ i ∈ −A}] length-nths
```

```
    using ‹card {i. i < n ∧ i ∈ − A} < length (nths xs (− A))› by auto  next
   qed
  qed
  then show ?thesis using ‹length xs = length ys› list-all2-all-nthI by blast
qed

lemma nths-weave:
assumes length xs = card {a∈A. a < length xs + length ys}
assumes length ys = card {a∈(−A). a < length xs + length ys}
shows nths (weave A xs ys) A = xs ∧ nths (weave A xs ys) (−A) = ys
using assms proof (induction length xs + length ys arbitrary: xs ys)
  case 0
  then show ?case
   unfolding weave-def nths-map by simp
next
  case (Suc l)
  then show ?case
  proof (cases l∈A)
   case True
  then have l∈{a ∈ A. a < length xs + length ys} using Suc.hyps mem-Collect-eq
zero-less-Suc by auto
   then have length xs > 0 using Suc by fastforce
    then obtain xs' x where xs = xs' @ [x] by (metis append-butlast-last-id
length-greater-0-conv)
   then have l = length xs' + length ys using Suc.hyps by simp
   have length-xs':length xs' = card {a ∈ A. a < length xs' + length ys}
   proof −
    have {a ∈ A. a < length xs + length ys} = {a ∈ A. a < length xs' + length
ys} ∪ {l}
     using ‹xs = xs' @ [x]› ‹l∈{a ∈ A. a < length xs + length ys}› ‹l = length
xs' + length ys›
     by force
    then have card {a ∈ A. a < length xs + length ys} = card {a ∈ A. a <
length xs' + length ys} + 1
     using ‹l = length xs' + length ys› by fastforce
    then show ?thesis by (metis One-nat-def Suc.prems(1) ‹xs = xs' @ [x]›
add-right-imp-eq
     length-Cons length-append list.size(3))
   qed
   have length-ys:length ys = card {a ∈ − A. a < length xs' + length ys}
   proof −
    have l∉{a ∈ − A. a < length xs + length ys} using ‹l∈A› ‹l = length xs' +
length ys› by blast
    have {a ∈ −A. a < length xs + length ys} = {a ∈ −A. a < length xs' +
length ys}
     apply (rule subset-antisym)
     using ‹l = length xs' + length ys› ‹Suc l = length xs + length ys› ‹l∉{a ∈
− A. a < length xs + length ys}›
     apply (metis (no-types, lifting) Collect-mono less-Suc-eq mem-Collect-eq)
```

using *Collect-mono Suc.hyps(2)* ‹$l = length\ xs' + length\ ys$› **by** *auto*
then show *?thesis* **using** *Suc.prems(2)* **by** *auto*
qed
have $length\ xs' + length\ ys \in A$ **using** ‹$l \in A$› ‹$l = length\ xs' + length\ ys$› **by** *blast*

then have $nths\ (weave\ A\ xs\ ys)\ A = nths\ (weave\ A\ xs'\ ys\ @\ [x])\ A$ **unfolding** ‹$xs = xs'\ @\ [x]$› **using** *weave-append1*[*OF* ‹$length\ xs' + length\ ys \in A$› *length-xs'*] **by** *metis*
also have $... = nths\ (weave\ A\ xs'\ ys)\ A\ @\ nths\ [x]\ \{a.\ a + (length\ xs' + length\ ys) \in A\}$
using *nths-append length-weave* **by** *metis*
also have $... = nths\ (weave\ A\ xs'\ ys)\ A\ @\ [x]$
using *nths-singleton* ‹$length\ xs' + length\ ys \in A$› **by** *auto*
also have $... = xs$ **using** *Suc.hyps(1)*[*OF* ‹$l = length\ xs' + length\ ys$› *length-xs' length-ys*] ‹$xs = xs'\ @\ [x]$› **by** *presburger*
finally have $nths\ (weave\ A\ xs\ ys)\ A = xs$ **by** *metis*

have $nths\ (weave\ A\ xs\ ys)\ (-A) = nths\ (weave\ A\ xs'\ ys\ @\ [x])\ (-A)$ **unfolding** ‹$xs = xs'\ @\ [x]$› **using** *weave-append1*[*OF* ‹$length\ xs' + length\ ys \in A$› *length-xs'*] **by** *metis*
also have $... = nths\ (weave\ A\ xs'\ ys)\ (-A)\ @\ nths\ [x]\ \{a.\ a + (length\ xs' + length\ ys) \in (-A)\}$
using *nths-append length-weave* **by** *metis*
also have $... = nths\ (weave\ A\ xs'\ ys)\ (-A)$
using *nths-singleton* ‹$length\ xs' + length\ ys \in A$› **by** *auto*
also have $... = ys$
using *Suc.hyps(1)*[*OF* ‹$l = length\ xs' + length\ ys$› *length-xs' length-ys*] **by** *presburger*
finally show *?thesis* **using** ‹$nths\ (weave\ A\ xs\ ys)\ A = xs$› **by** *auto*
next
case *False*
then have $l \notin \{a \in A.\ a < length\ xs + length\ ys\}$ **using** *Suc.hyps mem-Collect-eq zero-less-Suc* **by** *auto*
then have $length\ ys > 0$ **using** *Suc* **by** *fastforce*
then obtain $ys'\ y$ **where** $ys = ys'\ @\ [y]$ **by** (*metis append-butlast-last-id length-greater-0-conv*)
then have $l = length\ xs + length\ ys'$ **using** *Suc.hyps* **by** *simp*
have *length-ys'*:$length\ ys' = card\ \{a \in -A.\ a < length\ xs + length\ ys'\}$
proof −
have $\{a \in -A.\ a < length\ xs + length\ ys\} = \{a \in -A.\ a < length\ xs + length\ ys'\} \cup \{l\}$
using ‹$ys = ys'\ @\ [y]$› ‹$l \notin \{a \in A.\ a < length\ xs + length\ ys\}$› ‹$l = length\ xs + length\ ys'$›
by *force*
then have $card\ \{a \in -A.\ a < length\ xs + length\ ys\} = card\ \{a \in -A.\ a < length\ xs + length\ ys'\} + 1$
using ‹$l = length\ xs + length\ ys'$› **by** *fastforce*

then show *?thesis* **by** (*metis One-nat-def Suc.prems*(*2*) ‹*ys* = *ys'* @ [*y*]› *add-right-imp-eq*
length-Cons length-append list.size(*3*))
qed
have *length-xs*:*length xs* = *card* {*a* ∈ *A*. *a* < *length xs* + *length ys'*}
proof −
have *l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*} **using** ‹*l*∉*A*› ‹*l* = *length xs* + *length ys'*› **by** *blast*
have {*a* ∈ *A*. *a* < *length xs* + *length ys*} = {*a* ∈ *A*. *a* < *length xs* + *length ys'*}
apply (*rule subset-antisym*)
using ‹*l* = *length xs* + *length ys'*› ‹*Suc l* = *length xs* + *length ys*› ‹*l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*}›
apply (*metis* (*no-types*, *lifting*) *Collect-mono less-Suc-eq mem-Collect-eq*)
using *Collect-mono Suc.hyps*(*2*) ‹*l* = *length xs* + *length ys'*› **by** *auto*
then show *?thesis* **using** *Suc.prems*(*1*) **by** *auto*
qed
have *length xs* + *length ys'* ∉ *A* **using** ‹*l*∉*A*› ‹*l* = *length xs* + *length ys'*› **by** *blast*

then have *nths* (*weave A xs ys*) *A* = *nths* (*weave A xs ys'* @ [*y*]) *A* **unfolding** ‹*ys* = *ys'* @ [*y*]› **using** *weave-append2*[*OF* ‹*length xs* + *length ys'* ∉ *A*› *length-ys'*] **by** *metis*
also have ... = *nths* (*weave A xs ys'*) *A* @ *nths* [*y*] {*a*. *a* + (*length xs* + *length ys'*) ∈ *A*}
using *nths-append length-weave* **by** *metis*
also have ... = *nths* (*weave A xs ys'*) *A*
using *nths-singleton* ‹*length xs* + *length ys'* ∉ *A*› **by** *auto*
also have ... = *xs*
using *Suc.hyps*(*1*)[*OF* ‹*l* = *length xs* + *length ys'*› *length-xs length-ys'*] **by** *auto*
finally have *nths* (*weave A xs ys*) *A* = *xs* **by** *auto*

have *nths* (*weave A xs ys*) (−*A*) = *nths* (*weave A xs ys'* @ [*y*]) (−*A*) **unfolding** ‹*ys* = *ys'* @ [*y*]› **using** *weave-append2*[*OF* ‹*length xs* + *length ys'* ∉ *A*› *length-ys'*] **by** *metis*
also have ... = *nths* (*weave A xs ys'*) (−*A*) @ *nths* [*y*] {*a*. *a* + (*length xs* + *length ys'*) ∈ (−*A*)}
using *nths-append length-weave* **by** *metis*
also have ... = *nths* (*weave A xs ys'*) (−*A*) @ [*y*]
using *nths-singleton* ‹*length xs* + *length ys'* ∉ *A*› **by** *auto*
also have ... = *ys*
using *Suc.hyps*(*1*)[*OF* ‹*l* = *length xs* + *length ys'*› *length-xs length-ys'*] ‹*ys* = *ys'* @ [*y*]› **by** *simp*
finally show *?thesis* **using** ‹*nths* (*weave A xs ys*) *A* = *xs*› **by** *auto*
qed
qed

lemma *set-weave*:

assumes *length xs = card {a∈A. a < length xs + length ys}*
assumes *length ys = card {a∈−A. a < length xs + length ys}*
shows *set (weave A xs ys) = set xs ∪ set ys*
proof
show *set (weave A xs ys) ⊆ set xs ∪ set ys*
proof
fix *x* **assume** *x∈set (weave A xs ys)*
then obtain *i* **where** *weave A xs ys ! i = x i<length (weave A xs ys)* **by** (*meson in-set-conv-nth*)
show *x ∈ set xs ∪ set ys*
proof (*cases i∈A*)
case *True*
then have *i ∈ {a∈A. a < length xs + length ys}* **unfolding** *length-weave*
by (*metis ‹i < length (weave A xs ys)› length-weave mem-Collect-eq*)
then have *{a ∈ A. a < i} ⊂ {a∈A. a < length xs + length ys}*
using *Collect-mono ‹i < length (weave A xs ys)›*[*unfolded length-weave*] *le-Suc-ex less-imp-le-nat trans-less-add1*
le-neq-trans less-irefl mem-Collect-eq **by** *auto*
then have *card {a ∈ A. a < i} < card {a∈A. a < length xs + length ys}* **by** (*simp add*: *psubset-card-mono*)
then show *x ∈ set xs ∪ set ys*
unfolding *nth-weave*[*OF ‹i<length (weave A xs ys)›, unfolded ‹weave A xs ys ! i = x›*] **using** *True*
using *UnI1 assms(1) nth-mem* **by** *auto*
next
case *False*
have *i∉A ⟹ i ∈ {a∈−A. a < length xs + length ys}* **unfolding** *length-weave*
by (*metis ComplI ‹i < length (weave A xs ys)› length-weave mem-Collect-eq*)
then have *{a ∈ −A. a < i} ⊂ {a∈−A. a < length xs + length ys}*
using *Collect-mono ‹i < length (weave A xs ys)›*[*unfolded length-weave*] *le-Suc-ex less-imp-le-nat trans-less-add1*
le-neq-trans less-irefl mem-Collect-eq **using** *False* **by** *auto*
then have *card {a ∈ −A. a < i} < card {a∈−A. a < length xs + length ys}* **by** (*simp add*: *psubset-card-mono*)
then show *x ∈ set xs ∪ set ys*
unfolding *nth-weave*[*OF ‹i<length (weave A xs ys)›, unfolded ‹weave A xs ys ! i = x›*] **using** *False*
using *UnI1 assms(2) nth-mem* **by** *auto*
qed
qed
show *set xs ∪ set ys ⊆ set (weave A xs ys)*
using *nths-weave*[*OF assms*] **by** (*metis Un-subset-iff set-nths-subset*)
qed

lemma *weave-complementary-nthss*[*simp*]:
weave A (nths xs A) (nths xs (−A)) = xs
proof (*induction xs rule*:*rev-induct*)
case *Nil*

then show *?case* **by** (*metis gen-length-def length-0-conv length-code length-weave nths-nil*)
next
case (*snoc x xs*)
have *length-xs*:*length xs* = *length* (*nths xs A*) + *length* (*nths xs* (−*A*)) **by** (*metis length-weave snoc.IH*)
show *?case*
proof (*cases* (*length xs*)∈*A*)
case *True*
have *0*:*length* (*nths xs A*) + *length* (*nths xs* (−*A*)) ∈ *A* **using** *length-xs True* **by** *metis*
have *1*:*length* (*nths xs A*) = *card* {*a* ∈ *A*. *a* < *length* (*nths xs A*) + *length* (*nths xs* (−*A*))}
using *length-nths*[*of xs A*] **by** (*metis* (*no-types*, *lifting*) *Collect-cong length-xs*)
have *2*:*nths* (*xs* @ [*x*]) *A* = *nths xs A* @ [*x*]
unfolding *nths-append*[*of xs* [*x*] *A*] **using** *nths-singleton True* **by** *auto*
have *3*:*nths* (*xs* @ [*x*]) (−*A*) = *nths xs* (−*A*)
unfolding *nths-append*[*of xs* [*x*] −*A*] **using** *True* **by** *auto*
show *?thesis* **unfolding** *2 3 weave-append1*[*OF 0 1*] *snoc.IH* **by** *metis*
next
case *False*
have *0*:*length* (*nths xs A*) + *length* (*nths xs* (−*A*)) ∉ *A* **using** *length-xs False* **by** *metis*
have *1*:*length* (*nths xs* (−*A*)) = *card* {*a* ∈ −*A*. *a* < *length* (*nths xs A*) + *length* (*nths xs* (−*A*))}
using *length-nths*[*of xs* −*A*] **by** (*metis* (*no-types*, *lifting*) *Collect-cong length-xs*)
have *2*:*nths* (*xs* @ [*x*]) *A* = *nths xs A*
unfolding *nths-append*[*of xs* [*x*] *A*] **using** *nths-singleton False* **by** *auto*
have *3*:*nths* (*xs* @ [*x*]) (−*A*) = *nths xs* (−*A*) @ [*x*]
unfolding *nths-append*[*of xs* [*x*] −*A*] **using** *False* **by** *auto*
show *?thesis* **unfolding** *2 3 weave-append2*[*OF 0 1*] *snoc.IH* **by** *metis*
qed
qed

lemma *length-nths′*:
length (*nths xs I*) = *card* {*i*∈*I*. *i* < *length xs*}
unfolding *length-nths* **by** *meson*

end

17 Tensor Matricization

theory *Tensor-Matricization*
imports *Tensor-Plus*
Jordan-Normal-Form.Matrix DL-Missing-Sublist
begin

```
fun digit-decode :: nat list ⇒ nat list ⇒ nat where
digit-decode [] [] = 0 |
digit-decode (d # ds) (i # is) = i + d * digit-decode ds is

fun digit-encode :: nat list ⇒ nat ⇒ nat list where
digit-encode [] a = [] |
digit-encode (d # ds) a = a mod d # digit-encode ds (a div d)

lemma digit-encode-decode[simp]:
assumes is ◁ ds
shows digit-encode ds (digit-decode ds is) = is
  using assms apply (induction rule:valid-index.induct)
  unfolding digit-decode.simps digit-encode.simps
  by simp-all

lemma digit-decode-encode[simp]:
shows digit-decode ds (digit-encode ds a) = a mod (prod-list ds)
by (induction ds arbitrary:a; simp add: Divides.mod-mult2-eq add.commute)

lemma digit-decode-encode-lt[simp]:
assumes a < prod-list ds
shows digit-decode ds (digit-encode ds a) = a
by (simp add: assms)

lemma digit-decode-lt:
assumes is ◁ ds
shows digit-decode ds is < prod-list ds
using assms proof (induction rule:valid-index.induct)
  case Nil
  then show ?case by simp
next
  case (Cons is ds i d)
  have (i + d * digit-decode ds is) div (d * prod-list ds) = 0
    using Cons.IH Cons.hyps(2) Divides.div-mult2-eq by force
  then show ?case unfolding digit-decode.simps prod-list.Cons
    by (metis (no-types) Cons.IH Cons.hyps(2) div-eq-0-iff mult-eq-0-iff not-less0)
qed

lemma digit-encode-valid-index:
assumes a < prod-list ds
shows digit-encode ds a ◁ ds
using assms proof (induction ds arbitrary:a)
  case Nil
  show ?case by (simp add: valid-index.Nil)
next
  case (Cons d ds a)
  then show ?case
    unfolding digit-encode.simps using Cons
   by (metis Divides.div-mult2-eq div-eq-0-iff gr-implies-not0 prod-list.Cons mod-div-trivial
```

mult-not-zero valid-index.Cons)
qed

lemma *length-digit-encode*:
shows *length* (*digit-encode ds a*) = *length ds*
by (*induction ds arbitrary*:*a*; *simp-all*)

lemma *digit-encode-0*:
prod-list ds dvd a ⟹ *digit-encode ds a* = *replicate* (*length ds*) *0*
proof (*induction ds arbitrary*:*a*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons d ds a*)
then have *prod-list ds dvd* (*a div d*) **unfolding** *prod-list.Cons*
by (*metis dvd-0-right dvd-div-iff-mult dvd-mult-left mult.commute split-div*)
then show *?case* **unfolding** *digit-encode.simps length-Cons replicate-Suc prod-list.Cons*
using *Cons*
using *dvd-imp-mod-0 dvd-mult-left prod-list.Cons* **by** *force*
qed

lemma *valid-index-weave*:
assumes *is1* ◁ (*nths ds A*)
and *is2* ◁ (*nths ds* (−*A*))
shows *weave A is1 is2* ◁ *ds*
and *nths* (*weave A is1 is2*) *A* = *is1*
and *nths* (*weave A is1 is2*) (−*A*) = *is2*
proof −
have *length-ds*: *length is1* + *length is2* = *length ds*
using *valid-index-length*[*OF assms*(*1*)] *valid-index-length*[*OF assms*(*2*)]
length-weave weave-complementary-nthss **by** *metis*
have *1*:*length is1* = *card* {*i* ∈ *A*. *i* < *length is1* + *length is2*} **unfolding** *length-ds*
using *length-nths*′ *assms*(*1*) *valid-index-length* **by** *auto*
have *2*:*length is2* = *card* {*i* ∈ −*A*. *i* < *length is1* + *length is2*} **unfolding** *length-ds*
using *length-nths*′[*of ds* −*A*] *assms*(*2*) *valid-index-length* **by** *auto*
show *nths* (*weave A is1 is2*) *A* = *is1 nths* (*weave A is1 is2*) (−*A*) = *is2* **using** *nths-weave*[*OF 1 2*] **by** *blast*+
then have *nths* (*weave A is1 is2*) *A* ◁ (*nths ds A*)
nths (*weave A is1 is2*) (−*A*) ◁ (*nths ds* (−*A*)) **using** *assms* **by** *auto*
then show *weave A is1 is2* ◁ *ds* **using** *list-all2-nths valid-index-list-all2-iff* **by** *blast*
qed

definition *matricize* :: *nat set* ⇒ ′*a tensor* ⇒ ′*a mat* **where**
matricize rmodes T = *mat*
(*prod-list* (*nths* (*Tensor.dims T*) *rmodes*))
(*prod-list* (*nths* (*Tensor.dims T*) (−*rmodes*)))

```
  (λ(r, c). Tensor.lookup T (weave rmodes
    (digit-encode (nths (Tensor.dims T) rmodes) r)
    (digit-encode (nths (Tensor.dims T) (−rmodes)) c)
  ))


definition dematricize::nat set ⇒ 'a mat ⇒ nat list ⇒ 'a tensor where
dematricize rmodes A ds  = tensor-from-lookup ds
  (λis. A $$ (digit-decode (nths ds rmodes) (nths is rmodes),
              digit-decode (nths ds (−rmodes)) (nths is (−rmodes)))
 )


lemma dims-matricize:
dim-row (matricize rmodes T) = prod-list (nths (Tensor.dims T) rmodes)
dim-col (matricize rmodes T) = prod-list (nths (Tensor.dims T) (−rmodes))
  unfolding matricize-def using dim-row-mat by simp-all

lemma dims-dematricize: Tensor.dims (dematricize rmodes A ds) = ds
  by (simp add: dematricize-def dims-tensor-from-lookup)

lemma valid-index-nths:
assumes is ◁ ds
shows nths is A ◁ nths ds A
using assms proof (induction arbitrary:A rule:valid-index.induct)
  case Nil
  then show ?case using nths-nil valid-index.simps by blast
next
  case (Cons is ds i d)
  then have  nths is {j. Suc j ∈ A} ◁ nths ds {j. Suc j ∈ A}
    by simp
  then show ?case unfolding nths-Cons
    by (cases 0∈A; simp-all add: Cons.hyps(2) valid-index.Cons)
qed

lemma dematricize-matricize:
shows dematricize rmodes (matricize rmodes T) (Tensor.dims T) = T
proof (rule tensor-lookup-eqI)
   show 1:Tensor.dims (dematricize rmodes (matricize rmodes T) (Tensor.dims
T)) = Tensor.dims T
    by (simp add: dematricize-def dims-tensor-from-lookup)
   fix is assume is ◁ Tensor.dims (dematricize rmodes (matricize rmodes T)
(Tensor.dims T))
  then have is ◁ Tensor.dims T using 1 by auto
  let ?rds = (nths (Tensor.dims T) rmodes)
  let ?cds = (nths (Tensor.dims T) (−rmodes))
  have decode-r: digit-decode ?rds (nths is rmodes) < prod-list ?rds
    by (simp add: ‹is ◁ Tensor.dims T› valid-index-nths digit-decode-lt)
  have decode-c: digit-decode ?cds (nths is (−rmodes)) < prod-list ?cds
```

by (*simp add*: ‹*is* ◁ *Tensor.dims T*› *valid-index-nths digit-decode-lt*)
have (*matricize rmodes T*) $$
(*digit-decode ?rds* (*nths is rmodes*),
digit-decode ?cds (*nths is* (− *rmodes*))) =
Tensor.lookup T is
unfolding *matricize-def*
by (*simp add*: *decode-r decode-c* ‹*is* ◁ *Tensor.dims T*› *valid-index-nths*)
then show *Tensor.lookup* (*dematricize rmodes* (*matricize rmodes T*) (*Tensor.dims T*)) *is* = *Tensor.lookup T is*
by (*simp add*: *dematricize-def dims-tensor-from-lookup lookup-tensor-from-lookup*[*OF* ‹*is* ◁ *Tensor.dims T*›])
qed

lemma *matricize-dematricize*:
assumes *dim-row A* = *prod-list* (*nths ds rmodes*)
and *dim-col A* = *prod-list* (*nths ds* (−*rmodes*))
shows *matricize rmodes* (*dematricize rmodes A ds*) = *A*
proof (*rule eq-matI*)
show *dim-row* (*matricize rmodes* (*dematricize rmodes A ds*)) = *dim-row A*
unfolding *assms*(*1*) *dematricize-def dims-tensor-from-lookup matricize-def dim-row-mat* **by** *metis*
show *dim-col* (*matricize rmodes* (*dematricize rmodes A ds*)) = *dim-col A*
unfolding *assms*(*2*) *dematricize-def dims-tensor-from-lookup matricize-def dim-col-mat* **by** *metis*
fix *r c* **assume** *r* < *dim-row A c* < *dim-col A*
have *valid1*:*digit-encode* (*nths ds rmodes*) *r* ◁ *nths ds rmodes* **and**
valid2:*digit-encode* (*nths ds* (− *rmodes*)) *c* ◁ *nths ds* (− *rmodes*)
using ‹*r* < *dim-row A*› *assms*(*1*) ‹*c* < *dim-col A*› *assms*(*2*) *digit-encode-valid-index*
by *auto*
have *0*:*Tensor.lookup* (*dematricize rmodes A ds*)
(*weave rmodes*
(*digit-encode* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) *rmodes*) *r*)
(*digit-encode* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) (− *rmodes*)) *c*)
) = *A* $$ (*r*, *c*)
unfolding *dematricize-def* **unfolding** *dims-tensor-from-lookup*
unfolding *lookup-tensor-from-lookup*[*OF valid-index-weave*(*1*)[*OF valid1 valid2*]]
using *digit-decode-encode-lt*[*OF* ‹*c* < *dim-col A*›[*unfolded assms*(*2*)]]
digit-decode-encode-lt[*OF* ‹*r* < *dim-row A*›[*unfolded assms*(*1*)]]
valid-index-weave(*2*)[*OF valid1 valid2*] *valid-index-weave*(*3*)[*OF valid1 valid2*]
by *presburger*
from ‹*r* < *dim-row A*› **have** *r-le*: *r* < *prod-list* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) *rmodes*)
by (*metis* ‹*dim-row* (*matricize rmodes* (*dematricize rmodes A ds*)) = *dim-row A*› *matricize-def dim-row-mat*(*1*))
from ‹*c* < *dim-col A* ›**have** *c-le*: *c* < *prod-list* (*nths* (*Tensor.dims* (*dematricize rmodes A ds*)) (− *rmodes*))
by (*metis* ‹*dim-col* (*matricize rmodes* (*dematricize rmodes A ds*)) = *dim-col A*› *matricize-def dim-col-mat*(*1*))
then show (*matricize rmodes* (*dematricize rmodes A ds*)) $$ (*r*, *c*) = *A* $$ (*r*,

c)
unfolding *matricize-def* **using** *r-le c-le 0* **by** *simp*
qed

lemma *matricize-add*:
assumes *dims A* = *dims B*
shows *matricize I A* + *matricize I B* = *matricize I* (*A*+*B*)
proof (*rule eq-matI*)
show *dim-row* (*matricize I A* + *matricize I B*) = *dim-row* (*matricize I* (*A* + *B*)) **by** (*simp add*: *assms dims-matricize*(*1*))
show *dim-col* (*matricize I A* + *matricize I B*) = *dim-col* (*matricize I* (*A* + *B*)) **by** (*simp add*: *assms dims-matricize*(*2*))
fix *i j* **assume** *ij-le1*:*i* < *dim-row* (*matricize I* (*A* + *B*)) *j* < *dim-col* (*matricize I* (*A* + *B*))
then have
ij-le2:*i* < *prod-list* (*nths* (*Tensor.dims A*) *I*) *j* < *prod-list* (*nths* (*Tensor.dims A*) (−*I*)) **and**
ij-le3:*i* < *prod-list* (*nths* (*Tensor.dims B*) *I*) *j* < *prod-list* (*nths* (*Tensor.dims B*) (−*I*)) **and**
ij-le4:*i* < *prod-list* (*nths* (*Tensor.dims* (*A* + *B*)) *I*) *j* < *prod-list* (*nths* (*Tensor.dims* (*A* + *B*)) (−*I*))
by (*simp-all add*: *assms dims-matricize*)
then have *ij-le5*:*i* < *dim-row* (*matricize I B*) *j* < *dim-col* (*matricize I B*)
by (*simp-all add*: *assms dims-matricize*)
show (*matricize I A* + *matricize I B*) $$ (*i*, *j*) = *matricize I* (*A* + *B*) $$ (*i*, *j*)
unfolding *index-add-mat*(*1*)[*OF ij-le5*] **unfolding** *matricize-def* **unfolding** *index-mat*[*OF ij-le2*] *index-mat*[*OF ij-le3*] *index-mat*[*OF ij-le4*]
using *assms digit-encode-valid-index ij-le2*(*1*) *ij-le2*(*2*) *valid-index-weave*(*1*)
by *auto*
qed

lemma *matricize-0*:
shows *matricize I* (*tensor0 ds*) = 0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*)))
proof (*rule eq-matI*)
show *dim-row* (*matricize I* (*tensor0 ds*)) = *dim-row* (0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*))))
unfolding *zero-mat-def dim-row-mat* **by** (*simp add*: *dims-matricize*(*1*))
show *dim-col* (*matricize I* (*tensor0 ds*)) = *dim-col* (0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*))))
unfolding *zero-mat-def dim-row-mat* **by** (*simp add*: *dims-matricize*(*2*))
fix *i j* **assume** *ij-le1*: *i* < *dim-row* (0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*))))
j < *dim-col* (0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*))))
then have *ij-le2*:*i* < *dim-row* (*matricize I* (*tensor0 ds*)) *j* < *dim-col* (*matricize I* (*tensor0 ds*))
unfolding *zero-mat-def dim-row-mat* **by** (*simp-all add*: *dims-matricize*)
show *matricize I* (*tensor0 ds*) $$ (*i*, *j*) = 0_m (*dim-row* (*matricize I* (*tensor0*

ds))) (*dim-col* (*matricize I* (*tensor0 ds*))) $$ (*i*, *j*)
unfolding *zero-mat-def index-mat*[*OF ij-le2*] **unfolding** *matricize-def index-mat*[*OF ij-le2*[*unfolded dims-matricize*]]
by (*simp*, *metis lookup-tensor0 digit-encode-valid-index dims-matricize*(*1*) *dims-matricize*(*2*) *dims-tensor0*
ij-le2(*1*) *ij-le2*(*2*) *valid-index-weave*(*1*))
qed

end

18 Submatrices

theory *DL-Submatrix*
imports *Jordan-Normal-Form.Matrix DL-Missing-Sublist*
begin

19 Submatrix

definition *submatrix* :: *'a mat* ⇒ *nat set* ⇒ *nat set* ⇒ *'a mat* **where**
submatrix A I J = *mat* (*card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}) (*card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}) (λ(*i*,*j*). *A* $$ (*pick I i*, *pick J j*))

lemma *dim-submatrix*: *dim-row* (*submatrix A I J*) = *card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
dim-col (*submatrix A I J*) = *card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
unfolding *submatrix-def* **by** *simp-all*

lemma *submatrix-index*:
assumes *i*<*card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
assumes *j*<*card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
shows *submatrix A I J* $$ (*i*,*j*) = *A* $$ (*pick I i*, *pick J j*)
unfolding *submatrix-def* **by** (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *set-le-in*:{*a*. *a* < *n* ∧ *a* ∈ *I*} = {*a* ∈ *I*. *a* < *n*} **by** *meson*

lemma *submatrix-index-card*:
assumes *i*<*dim-row A j*<*dim-col A i*∈*I j*∈*J*
shows *submatrix A I J* $$ (*card* {*a*∈*I*. *a* < *i*}, *card* {*a*∈*J*. *a* < *j*}) = *A* $$ (*i*, *j*)
proof −
have *i* = *pick I* (*card* {*a*∈*I*. *a* < *i*})
j = *pick J* (*card* {*a*∈*J*. *a* < *j*}) **using** *pick-card-in-set assms* **by** *auto*
have {*a*∈*I*. *a* < *i*} ⊂ {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*}
{*a*∈*J*. *a* < *j*} ⊂ {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}
unfolding *set-le-in* **using** ‹*i*<*dim-row A*› ‹*j*<*dim-col A*› *Collect-mono less-imp-le less-le-trans* ‹*i*∈*I*› ‹*j*∈*J*› **by** *auto*
then have *card* {*a*∈*I*. *a* < *i*} < *card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*}
card {*a*∈*J*. *a* < *j*} < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*} **by** (*simp-all add*: *psubset-card-mono*)

then show *?thesis*
using ‹i = *pick I* (*card* {$a \in I$. $a < i$})› ‹j = *pick J* (*card* {$a \in J$. $a < j$})› *submatrix-index* **by** *fastforce*
qed

lemma *submatrix-split*: *submatrix A I J* = *submatrix* (*submatrix A UNIV J*) *I UNIV*
proof (*rule eq-matI*)
show *dim-row* (*submatrix A I J*) = *dim-row* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
by (*simp add*: *dim-submatrix*(*1*))
show *dim-col* (*submatrix A I J*) = *dim-col* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
by (*simp add*: *dim-submatrix*(*2*))
fix *i j* **assume** *ij-le*:*i* < *dim-row* (*submatrix* (*submatrix A UNIV J*) *I UNIV*) *j* < *dim-col* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
then have *ij-le1*:*i*<*card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*} *j*<*card* {*i*. *i* < *dim-col A* ∧ *i* ∈ *J*}
by (*simp-all add*: *dim-submatrix*)
then have *ij-le2*:*i*<*card* {*i*. *i* < *dim-row* (*submatrix A UNIV J*) ∧ *i* ∈ *I*} *j*<*card* {*i*. *i* < *dim-col* (*submatrix A UNIV J*) ∧ *i* ∈ *UNIV*}
by (*simp-all add*: *dim-submatrix*)
then have *i-le3*:*pick I i*<*card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *UNIV*}
using *ij-le1*(*1*) *pick-le* **by** *auto*
have *j-le3*: *pick UNIV j*<*card* {*i*. *i* < *dim-col A* ∧ *i* ∈ *J*} **unfolding** *pick-UNIV* **by** (*simp add*: *ij-le1*(*2*))
then show *submatrix A I J* $$ (*i*, *j*) = *submatrix* (*submatrix A UNIV J*) *I UNIV* $$ (*i*, *j*)
unfolding *submatrix-index*[*OF ij-le1*] *submatrix-index*[*OF ij-le2*] *submatrix-index*[*OF i-le3 j-le3*]
unfolding *pick-UNIV* **by** *metis*
qed

end

20 CP-Rank and Matrix Rank

theory *DL-Rank-CP-Rank*
imports *Tensor-Rank DL-Rank Tensor-Matricization DL-Submatrix DL-Missing-Vector-Space*
begin

abbreviation *mrank A* == *vec-space.rank* (*dim-row A*) *A*

no-notation *normal-rel* (**infixl** ◁ *60*)

lemma *lookup-order1-prod*:
assumes ⋀*B*. *B*∈*set Bs* ⟹ *Tensor.order B* = *1*
assumes *is* ◁ *dims* (*prod-list Bs*)
shows *lookup* (*prod-list Bs*) *is* = *prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip is Bs*))

using *assms* **proof** (*induction Bs arbitrary*:*is*)
case *Nil*
then show *?case* **unfolding** *prod-list.Nil* **unfolding** *zip.simps tensor-one-def*
by (*metis* (*no-types*, *lifting*) *dims-tensor-from-lookup length-greater-0-conv length-map prod-list.Nil*
lookup-tensor-from-lookup tensor-one-def tensor-one-from-lookup)
next
case (*Cons B Bs is'*)
then obtain *i is* **where** *is'* = *i* # *is*
by (*metis append-is-Nil-conv dims-tensor-prod length-0-conv list.set-intros*(*1*) *prod-list.Cons valid-index.simps zero-neq-one*)
have *Tensor.order B* = *1* **using** *Cons* **by** *auto*
then have *valid1*:[*i*] ◁ *dims B*
using ‹*is'* ◁ *dims* (*prod-list* (*B* # *Bs*))›[*unfolded prod-list.Cons dims-tensor-prod* ‹*is'* = *i* # *is*›]
by (*metis One-nat-def Suc-length-conv hd-append2 length-0-conv list.sel*(*1*) *list.simps*(*3*) *valid-index.Nil valid-index.simps*)
have *valid2*:*is* ◁ *dims* (*prod-list Bs*)
using ‹*is'* ◁ *dims* (*prod-list* (*B* # *Bs*))›[*unfolded prod-list.Cons dims-tensor-prod* ‹*is'* = *i* # *is*›] ‹*Tensor.order B* = *1*›
by (*metis One-nat-def Suc-length-conv append-eq-Cons-conv length-0-conv list.sel*(*3*) *list.simps*(*3*) *self-append-conv2 valid-indexE*)
show *?case* **unfolding** ‹*is'* = *i* # *is*› *List.zip-Cons-Cons List.list.map*(*2*) *prod-list.Cons*
lookup-tensor-prod[*OF valid1 valid2*, *simplified*] **by** (*simp add*: *Cons.IH Cons.prems*(*1*) *valid2*)
qed

lemma *matricize-cprank-max1*:
fixes *A*::*'a*::*field tensor*
assumes *cprank-max1 A*
shows *mrank* (*matricize I A*) ≤ *1*
proof −
obtain *Bs a* **where** ⋀*B*. *B* ∈ *set Bs* ⟹ *Tensor.order B* = *1 a* · *prod-list Bs* = *A*
using *cprank-max1-prod-listE assms* **by** *metis*
def *row-factor* == λ*ris*. *a* ∗ (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip ris* (*nths Bs I*))))
def *col-factor* == λ*cis*. (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip cis* (*nths Bs* (−*I*)))))
have ⋀*is*. *is* ◁ *dims A* ⟹ *lookup A is* = *row-factor* (*nths is I*) ∗ *col-factor* (*nths is* (−*I*))
proof −
fix *is* **assume** *is* ◁ *dims A*
then have *lookup A is* = *a* ∗ (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*zip is Bs*)))
using *lookup-order1-prod*[*OF* ‹⋀*B*. *B* ∈ *set Bs* ⟹ *Tensor.order B* = *1*›] *lookup-smult*
using ‹*a* · *prod-list Bs* = *A*› *dims-smult* **by** *fastforce*
also have ... = *a* ∗ (*prod-list* (*map* (λ(*i*,*B*). *lookup B* [*i*]) (*nths* (*zip is Bs*) *I*)))

```
*
                    (prod-list (map (λ(i,B). lookup B [i]) (nths (zip is Bs) (−I))))
     using prod-list-complementary-nthss by auto
   also have ... = row-factor (nths is I) * col-factor (nths is (−I))
     using nths-zip row-factor-def col-factor-def by metis
   finally show lookup A is = row-factor (nths is I) * col-factor (nths is (−I)) .
 qed
 def row-factor' == λr. row-factor (digit-encode (nths (Tensor.dims A) I) r)
 def col-factor' == λc. col-factor (digit-encode (nths (Tensor.dims A) (−I)) c)
  have ⋀r c. r<dim-row (matricize I A) ⟹ c<dim-col (matricize I A) ⟹
matricize I A $$ (r,c) = row-factor' r * col-factor' c
 proof −
   fix r c assume r<dim-row (matricize I A) c<dim-col (matricize I A)
   then have matricize I A $$ (r,c) = Tensor.lookup A (weave I
     (digit-encode (nths (Tensor.dims A) I) r)
     (digit-encode (nths (Tensor.dims A) (−I)) c)
   ) unfolding dims-matricize unfolding matricize-def by simp
   also have ... = row-factor' r * col-factor' c
    using ‹⋀is. is ◁ dims A ⟹ lookup A is = row-factor (nths is I) * col-factor
(nths is (− I))›
     valid-index-weave[OF
    digit-encode-valid-index[OF ‹r < dim-row (matricize I A)›[unfolded dims-matricize]]
    digit-encode-valid-index[OF ‹c < dim-col (matricize I A)›[unfolded dims-matricize]]]
      valid-index-weave(2) valid-index-weave(3) row-factor'-def col-factor'-def by
metis
   finally show matricize I A $$ (r,c) = row-factor' r * col-factor' c .
 qed
  then show ?thesis using vec-space.rank-le-1-product-entries[of matricize I A]
by blast
qed

lemma matrix-rank-le-cprank-max:
fixes A :: ('a::field) tensor
assumes cprank-max r A
shows mrank (matricize I A) ≤ r
using assms
proof (induction rule:cprank-max.induct)
 fix ds :: nat list
 have matricize I (tensor0 ds) = 0_m (dim-row (matricize I (tensor0 ds))) (dim-col
(matricize I (tensor0 ds)))
   using matricize-0 by auto
 then show mrank (matricize I (tensor0 ds)) ≤ 0
   using eq-imp-le vec-space.rank-0I by metis
next
 fix A B::'a tensor and j::nat
 assume dims A = dims B
 assume cprank-max1 A
 assume mrank (matricize I B) ≤ j
 have mrank (matricize I A) ≤ 1 using ‹cprank-max1 A› matricize-cprank-max1
```

by *auto*
have *mrank* (*matricize I* (*A* + *B*)) ≤ *mrank* (*matricize I A*) + *mrank* (*matricize I B*)
using *matricize-add vec-space.rank-subadditive dims-matricize carrier-matI index-add-mat*(*2*) ‹*dims A* = *dims B*› **by** *metis*
then show *mrank* (*matricize I* (*A* + *B*)) ≤ *Suc j*
using ‹*mrank* (*matricize I A*) ≤ *1*› ‹*mrank* (*matricize I B*) ≤ *j*› **by** *linarith*
qed

lemma *matrix-rank-le-cp-rank*:
fixes *A* :: ('*a*::*field*) *tensor*
shows *mrank* (*matricize I A*) ≤ *cprank A*
using *matrix-rank-le-cprank-max* **using** *cprank-max-cprank* **by** *auto*

end

21 Missing Lemmas of Matrix

theory *DL-Missing-Matrix*
imports *Jordan-Normal-Form.Matrix*
begin

lemma *dim-vec-of-list*[*simp*] :*dim-vec* (*vec-of-list as*) = *length as* **by** *transfer auto*

lemma *list-vec*: *list-of-vec* (*vec-of-list xs*) = *xs*
by (*transfer*, *metis* (*mono-tags*, *lifting*) *atLeastLessThan-iff map-eq-conv map-nth mk-vec-def old.prod.case set-upt*)

lemma *vec-list*: *vec-of-list* (*list-of-vec v*) = *v*
apply *transfer* **unfolding** *mk-vec-def* **by** *auto*

lemma *index-vec-of-list*: *i*<*length xs* ⟹ (*vec-of-list xs*) $ *i* = *xs* ! *i*
by (*metis vec.abs-eq index-vec vec-of-list.abs-eq*)

lemma *nth-list-of-vec*: *i*<*dim-vec v* ⟹ (*list-of-vec v*) ! *i* = *v* $ *i*
by (*metis dim-vec-of-list index-vec-of-list vec-list*)

lemma *vec-of-list-index*: *vec-of-list xs* $ *j* = *xs* ! *j*
apply *transfer* **unfolding** *mk-vec-def* **unfolding** *undef-vec-def*
by (*simp*, *metis append-Nil2 nth-append*)

lemma *list-of-vec-index*: *list-of-vec v* ! *j* = *v* $ *j*
by (*metis vec-list vec-of-list-index*)

lemma *list-of-vec-map*: *list-of-vec xs* = *map* (*op* $ *xs*) [*0*..<*dim-vec xs*] **by** *transfer auto*

definition *component-mult v w* = *vec* (*min* (*dim-vec v*) (*dim-vec w*)) (λ*i*. *v* $ *i* ∗ *w* $ *i*)

definition *vec-set*::*'a vec* ⇒ *'a set* (*set*$_v$)
where *vec-set v* = *vec-index v* ‘ {..<*dim-vec v*}

lemma *index-component-mult*:
assumes *i* < *dim-vec v i* < *dim-vec w*
shows *component-mult v w* $ *i* = *v* $ *i* ∗ *w* $ *i*
unfolding *component-mult-def* **using** *assms index-vec* **by** *auto*

lemma *dim-component-mult*:
dim-vec (*component-mult v w*) = *min* (*dim-vec v*) (*dim-vec w*)
unfolding *component-mult-def* **using** *index-vec* **by** *auto*

lemma *vec-setE*:
assumes *a* ∈ *set*$_v$ *v*
obtains *i* **where** *v*$*i* = *a i*<*dim-vec v* **using** *assms* **unfolding** *vec-set-def* **by** *blast*

lemma *vec-setI*:
assumes *v*$*i* = *a i*<*dim-vec v*
shows *a* ∈ *set*$_v$ *v* **using** *assms* **unfolding** *vec-set-def* **using** *image-eqI lessThan-iff* **by** *blast*

lemma *set-list-of-vec*:
set (*list-of-vec v*) = *set*$_v$ *v* **unfolding** *vec-set-def* **by** *transfer auto*

lemma *length-list-of-vec*[*simp*] :*length* (*list-of-vec v*) = *dim-vec v* **by** *transfer auto*

end

22 Matrix to Vector Conversion

theory *DL-Flatten-Matrix*
imports *HOL.Real Jordan-Normal-Form.Matrix*
begin

definition *extract-matrix* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *nat* ⇒ *'a mat* **where**
extract-matrix a m n = *mat m n* (λ(*i*,*j*). *a* (*i*∗*n* + *j*))

definition *flatten-matrix* :: *'a mat* ⇒ (*nat* ⇒ *'a*) **where**
flatten-matrix A k = *A* $$ (*k div dim-col A*, *k mod dim-col A*)

lemma *two-digit-le*:
fixes *i j* :: *nat*
assumes *i* < *m j* < *n*
shows *i*∗*n* + *j* < *m*∗*n*
proof −
have (*i*∗*n* + *j*) *div n* = *i m*∗*n div n* = *m* **using** *assms* **by** *auto*
then have (*i* ∗ *n* + *j*) *div m div n* = *i div m*
by (*metis Divides.div-mult2-eq mult.commute*)

then show *?thesis*
by (*metis Divides.div-mult2-eq* ‹$i < m$› ‹$m * n$ *div* $n = m$› *div-eq-0-iff not-less0*)
qed

lemma *extract-matrix-cong*:
assumes $\bigwedge i.\ i < m * n \Longrightarrow a\ i = b\ i$
shows *extract-matrix a m n* = *extract-matrix b m n*
proof −
have $\bigwedge i\ j.\ i < m \Longrightarrow j < n \Longrightarrow a\ (i*n + j) = b\ (i*n + j)$ **using** *two-digit-le assms* **by** *blast*
then show *?thesis* **unfolding** *extract-matrix-def* **by** *auto*
qed

lemma *extract-matrix-flatten-matrix*:
extract-matrix (*flatten-matrix A*) (*dim-row A*) (*dim-col A*) = *A*
unfolding *extract-matrix-def flatten-matrix-def* **by** *auto*

lemma *flatten-matrix-extract-matrix*:
shows $\bigwedge k.\ k<m*n \Longrightarrow$ *flatten-matrix* (*extract-matrix a m n*) *k* = *a k*
unfolding *extract-matrix-def flatten-matrix-def*
by (*metis* (*no-types*, *lifting*) *Divides.div-mult2-eq case-prod-conv div-eq-0-iff dim-col-mat*(*1*) *index-mat*(*1*) *div-mult-mod-eq mod-less-divisor mult.commute mult-zero-right not-gr0 not-less0*)

lemma *index-extract-matrix*:
assumes $i<m$ $j<n$
shows *extract-matrix a m n* $$ (*i*,*j*) = *a* (*i*∗*n* + *j*)
unfolding *extract-matrix-def* **using** *assms* **by** *simp*

lemma *dim-extract-matrix*:
shows *dim-row* (*extract-matrix as m n*) = *m*
and *dim-col* (*extract-matrix as m n*) = *n*
unfolding *extract-matrix-def* **by** *simp-all*

end

23 Deep Learning Networks

theory *DL-Network*
imports *Tensor-Product HOL.Real DL-Missing-Matrix Jordan-Normal-Form.Matrix Tensor-Unit-Vec DL-Flatten-Matrix DL-Missing-List*
begin

This symbol is used for the Tensor product:

no-notation *Group.monoid.mult* (**infixl** ⊗$_1$ *70*)

datatype *'a convnet* = *Input nat* | *Conv 'a 'a convnet* | *Pool 'a convnet 'a convnet*

fun *input-sizes* :: *'a convnet* ⇒ *nat list* **where**

input-sizes (*Input M*) = [*M*] |
input-sizes (*Conv A m*) = *input-sizes m* |
input-sizes (*Pool m1 m2*) = *input-sizes m1* @ *input-sizes m2*

fun *count-weights* :: (*nat* × *nat*) *convnet* ⇒ *nat* **where**
count-weights (*Input M*) = *0* |
count-weights (*Conv* (*r0*, *r1*) *m*) = *r0* ∗ *r1* + *count-weights m* |
count-weights (*Pool m1 m2*) = *count-weights m1* + *count-weights m2*

fun *output-size* :: (*nat* × *nat*) *convnet* ⇒ *nat* **where**
output-size (*Input M*) = *M* |
output-size (*Conv* (*r0*,*r1*) *m*) = *r0* |
output-size (*Pool m1 m2*) = *output-size m1*

inductive *valid-net* :: (*nat*×*nat*) *convnet* ⇒ *bool* **where**
valid-net (*Input M*) |
output-size m = *r1* ⟹ *valid-net m* ⟹ *valid-net* (*Conv* (*r0*,*r1*) *m*) |
output-size m1 = *output-size m2* ⟹ *valid-net m1* ⟹ *valid-net m2* ⟹ *valid-net* (*Pool m1 m2*)

fun *insert-weights* :: (*nat* × *nat*) *convnet* ⇒ (*nat* ⇒ *real*) ⇒ *real mat convnet* **where**
insert-weights (*Input M*) *w* = *Input M* |
insert-weights (*Conv* (*r0*,*r1*) *m*) *w* = *Conv*
(*extract-matrix w r0 r1*)
(*insert-weights m* (λ*i*. *w* (*i*+*r0*∗*r1*))) |
insert-weights (*Pool m1 m2*) *w* = *Pool*
(*insert-weights m1 w*)
(*insert-weights m2* (λ*i*. *w* (*i*+(*count-weights m1*))))

fun *remove-weights* :: *real mat convnet* ⇒ (*nat* × *nat*) *convnet* **where**
remove-weights (*Input M*) = *Input M* |
remove-weights (*Conv A m*) = *Conv* (*dim-row A*, *dim-col A*) (*remove-weights m*)
|
remove-weights (*Pool m1 m2*) = *Pool* (*remove-weights m1*) (*remove-weights m2*)

abbreviation *output-size′* == (λ*m*. *output-size* (*remove-weights m*))
abbreviation *valid-net′* == (λ*m*. *valid-net* (*remove-weights m*))

fun *evaluate-net* :: *real mat convnet* ⇒ *real vec list* ⇒ *real vec* **where**
evaluate-net (*Input M*) *inputs* = *hd inputs* |
evaluate-net (*Conv A m*) *inputs* = *A* ∗$_v$ *evaluate-net m inputs* |
evaluate-net (*Pool m1 m2*) *inputs* = *component-mult*
(*evaluate-net m1* (*take* (*length* (*input-sizes m1*)) *inputs*))
(*evaluate-net m2* (*drop* (*length* (*input-sizes m1*)) *inputs*))

definition *mat-tensorlist-mult* :: *real mat* ⇒ *real tensor vec* ⇒ *nat list* ⇒ *real tensor vec*

where *mat-tensorlist-mult A Ts ds*
= *Matrix.vec* (*dim-row A*) (λ*j*. *tensor-from-lookup ds* (λ*is*. (*A* ∗$_v$ (*map-vec* (λ*T*. *Tensor.lookup T is*) *Ts*)) $*j*))

lemma *insert-weights-cong*:
assumes (⋀*i*. *i*<*count-weights m* ⟹ *w1 i* = *w2 i*)
shows *insert-weights m w1* = *insert-weights m w2*
using *assms* **proof** (*induction m arbitrary*: *w1 w2*)
case *Input*
then show *?case* **by** *simp*
next
case (*Conv r01 m*)
then obtain *r0 r1* **where** *r01* = (*r0*,*r1*) **by** (*meson surj-pair*)
have *2*:*insert-weights m* (λ*i*. *w1* (*i* + *r0* ∗ *r1*)) = *insert-weights m* (λ*i*. *w2* (*i* + *r0* ∗ *r1*)) **using** *Conv*
using ‹*r01* = (*r0*, *r1*)› *add.commute add-less-cancel-right count-weights.simps*(*2*)
by *fastforce*
then show *?case* **unfolding** ‹*r01* = (*r0*,*r1*)› *insert-weights.simps*
by (*metis Conv.prems* ‹*r01* = (*r0*, *r1*)› *count-weights.simps*(*2*) *extract-matrix-cong trans-less-add1*)
next
case (*Pool m1 m2*)
have *1*:*insert-weights m1 w1* = *insert-weights m1 w2*
using *Pool*(*1*)[*of w1 w2*] *Pool*(*3*)[*unfolded count-weights.simps*] **by** *simp*
have *2*:*insert-weights m2* (λ*i*. *w1* (*i* + *count-weights m1*)) = *insert-weights m2* (λ*i*. *w2* (*i* + *count-weights m1*))
using *Pool*(*2*)[*of* λ*i*. *w1* (*i* + *count-weights m1*) λ*i*. *w2* (*i* + *count-weights m1*)] *Pool*(*3*)[*unfolded count-weights.simps*] **by** *simp*
show *?case* **unfolding** *insert-weights.simps 1 2* **by** *metis*
qed

lemma *dims-mat-tensorlist-mult*:
assumes *T*∈*set*$_v$ (*mat-tensorlist-mult A Ts ds*)
shows *Tensor.dims T* = *ds*
proof −
obtain *j* **where** *T* = *tensor-from-lookup ds* (λ*is*. (*A* ∗$_v$ (*map-vec* (λ*T*. *Tensor.lookup T is*) *Ts*)) $*j*)
using *vec-setE*[*OF assms*, *unfolded mat-tensorlist-mult-def*] **by** (*metis dim-vec index-vec*)
then show *?thesis* **by** (*simp add*: *length-tensor-vec-from-lookup tensor-from-lookup-def*)
qed

fun *tensors-from-net* :: *real mat convnet* ⇒ *real tensor vec* **where**
tensors-from-net (*Input M*) = *Matrix.vec M* (λ*i*. *unit-vec M i*) |
tensors-from-net (*Conv A m*) = *mat-tensorlist-mult A* (*tensors-from-net m*) (*input-sizes m*) |
tensors-from-net (*Pool m1 m2*) = *component-mult* (*tensors-from-net m1*) (*tensors-from-net m2*)

lemma *output-size-correct-tensors*:
assumes *valid-net' m*
shows *output-size' m = dim-vec (tensors-from-net m)*
using *assms* **proof** (*induction m*)
case *Input*
then show *?case* **by** *simp*
next
case (*Conv A m*)
then show *?case*
unfolding *remove-weights.simps output-size.simps tensors-from-net.simps*
using *mat-tensorlist-mult-def* **by** *auto*
next
case (*Pool m1 m2*)
then show *?case* **by** (*metis convnet.distinct*(*3*) *convnet.distinct*(*5*) *convnet.inject*(*3*) *dim-component-mult*
min.idem output-size.simps(*3*) *remove-weights.simps*(*3*) *tensors-from-net.simps*(*3*) *valid-net.simps*)
qed

lemma *output-size-correct*:
assumes *valid-net' m*
and *map dim-vec inputs = input-sizes m*
shows *output-size' m = dim-vec (evaluate-net m inputs)*
using *assms* **proof** (*induction m arbitrary*:*inputs*)
case *Input*
then show *?case* **using** *length-Cons list.map-sel*(*1*) *list.sel*(*1*) *list.simps*(*8*) *list.size*(*3*) *nat.simps*(*3*) **by** *auto*
next
case (*Conv A m*)
then show *?case* **unfolding** *evaluate-net.simps remove-weights.simps output-size.simps dim-mult-mat-vec*
by *auto*
next
case (*Pool m1 m2*)
then have *valid-net' m1 valid-net' m2*
using *convnet.distinct*(*3*) *convnet.distinct*(*5*) *convnet.inject*(*3*) *remove-weights.simps*(*3*) *valid-net.cases* **by** *fastforce+*
moreover have *map dim-vec (take (length (input-sizes m1)) inputs) = input-sizes m1*
map dim-vec (drop (length (input-sizes m1)) inputs) = input-sizes m2
using *Pool.prems*(*2*) **by** (*metis append-eq-conv-conj drop-map input-sizes.simps*(*3*) *take-map*)*+*
ultimately have
output-size' m1 = dim-vec (evaluate-net m1 (take (length (input-sizes m1)) inputs))
output-size' m2 = dim-vec (evaluate-net m2 (drop (length (input-sizes m1)) inputs))
using *Pool.IH* **by** *blast+*

then show *?case* **unfolding** *evaluate-net.simps remove-weights.simps output-size.simps*
by (*metis Pool.prems*(*1*) ‹*valid-net' m1*› ‹*valid-net' m2*› *dim-component-mult output-size.simps*(*3*) *output-size-correct-tensors remove-weights.simps*(*3*) *tensors-from-net.simps*(*3*))
qed

lemma *input-sizes-remove-weights*: *input-sizes m* = *input-sizes* (*remove-weights m*)
by (*induction m*; *simp*)

lemma *dims-tensors-from-net*:
assumes *T* ∈ set_v (*tensors-from-net m*)
shows *Tensor.dims T* = *input-sizes m*
using *assms* **proof** (*induction m arbitrary*:*T*)
case (*Input M*)
then obtain *j* **where** *T* = *unit-vec M j*
using *vec-setE tensors-from-net.simps*(*1*) **by** (*metis dim-vec index-vec*)
then show *?case* **by** (*simp add*: *dims-unit-vec*)
next
case (*Conv A m*)
then show *?case* **unfolding** *remove-weights.simps input-sizes.simps*
using *dims-mat-tensorlist-mult* **by** (*simp add*: *input-sizes-remove-weights*)
next
case (*Pool m1 m2 T*)
then obtain *i* **where**
component-mult (*tensors-from-net m1*) (*tensors-from-net m2*) $ *i* = *T*
i < *dim-vec* (*tensors-from-net m1*) *i* < *dim-vec* (*tensors-from-net m2*)
using *tensors-from-net.simps vec-setE dim-component-mult* **by** (*metis min.strict-boundedE*)
then obtain *T1 T2* **where** *T* = *T1* ⊗ *T2 T1* ∈ set_v (*tensors-from-net m1*) *T2* ∈ set_v (*tensors-from-net m2*)
using *vec-setI* **by** (*metis index-component-mult*)
then show *?case* **unfolding** *remove-weights.simps input-sizes.simps* **by** (*simp add*: *Pool.IH*(*1*) *Pool.IH*(*2*))
qed

notation *Matrix.unit-vec* ($unit_v$)

definition *base-input* :: *real mat convnet* ⇒ *nat list* ⇒ *real vec list* **where**
base-input m is = (*map* (λ(*n*, *i*). $unit_v$ *n i*) (*zip* (*input-sizes m*) *is*))

lemma *base-input-length*:
assumes *is* ◁ *input-sizes m*
shows *input-sizes m* = *map dim-vec* (*base-input m is*)
proof (*rule nth-equalityI*)
have *length* (*input-sizes m*) = *length is* **using** *assms valid-index-length* **by** *auto*
then show *length* (*input-sizes m*) = *length* (*map dim-vec* (*base-input m is*))
unfolding *base-input-def* **by** *auto*
{
fix *i*
assume *i*<*length* (*input-sizes m*)

then have *map* $(\lambda(n,\ i).\ unit_v\ n\ i)$ (*zip* (*input-sizes m*) *is*) ! *i* = $unit_v$ (*input-sizes m* ! *i*) (*is* ! *i*)
using ‹*length* (*input-sizes m*) = *length is*› **by** *auto*
then have *input-sizes m* ! *i* = *map dim-vec* (*base-input m is*) ! *i* **unfolding** *base-input-def* **using** *index-unit-vec*(*3*)
using ‹*i* < *length* (*input-sizes m*)› ‹*length* (*input-sizes m*) = *length* (*map dim-vec* (*base-input m is*))›
base-input-def assms length-map nth-map valid-index-lt **by** (*simp add*: *input-sizes-remove-weights*)
}
then show ∀ *i*<*length* (*input-sizes m*). *input-sizes m* ! *i* = *map dim-vec* (*base-input m is*) ! *i* **by** *auto*
qed

lemma *nth-mat-tensorlist-mult*:
assumes ⋀*A*. *A*∈set_v *Ts* ⟹ *dims A* = *ds*
assumes *i* < *dim-row A*
assumes *dim-vec Ts* = *dim-col A*
shows *mat-tensorlist-mult A Ts ds* $ *i* = *listsum ds* (*map* (λ*j*. (*A* $$ (*i*,*j*)) · *Ts* $ *j*) [*0*..<*dim-vec Ts*])
(**is** - = *listsum ds ?Ts'*)
proof (*rule tensor-lookup-eqI*)
have *dims-Ts'*:⋀*T*. *T*∈*set ?Ts'* ⟹ *dims T* = *ds*
proof −
fix *T* **assume** *T*∈*set ?Ts'*
then obtain *k* **where** *T* = *?Ts'* ! *k* **and** *k* < *length ?Ts' k* < *dim-vec Ts* **using** *in-set-conv-nth* **by** *force*
show *dims T* = *ds* **unfolding** ‹*T* = *?Ts'* ! *k*› *nth-map*[*OF* ‹*k* < *length ?Ts'*›[*unfolded length-map*]]
using *assms*(*1*) ‹*k* < *dim-vec Ts*›
by (*simp add*: ‹*k* < *length* (*map* (λ*j*. *A* $$ (*i*, *j*) · *Ts* $ *j*) [*0*..<*dim-vec Ts*])› *vec-setI*)
qed
then show *dims-eq*:*dims* (*mat-tensorlist-mult A Ts ds* $ *i*) = *dims* (*Tensor-Plus.listsum ds* (*map* (λ*j*. *A* $$ (*i*, *j*) · *Ts* $ *j*) [*0*..<*dim-vec Ts*]))
using *dims-mat-tensorlist-mult assms mat-tensorlist-mult-def listsum-dims*
by (*metis* (*no-types*, *lifting*) *dim-vec vec-setI*)

fix *is* **assume** *is-valid*:*is* ◁ *dims* (*mat-tensorlist-mult A Ts ds* $ *i*)
then have *is* ◁ *ds* **using** *dims-eq dims-Ts' listsum-dims* **by** (*metis* (*no-types*, *lifting*))

have *summand-eq*: ⋀*j*. *j* ∈ {*0* ..< *dim-vec Ts*} ⟹ *row A i* $ *j* ∗ (*map-vec* (λ*T*. *Tensor.lookup T is*) *Ts*) $ *j* = *lookup* (*A* $$ (*i*, *j*) · *Ts* $ *j*) *is*
using *index-vec* ‹*i*<*dim-row A*› *row-def* ‹*dim-vec Ts* = *dim-col A*›
‹*is* ◁ *ds*› *assms*(*1*) *lookup-smult atLeastLessThan-iff index-map-vec*(*1*) *vec-setI*
by *metis*

have *lookup* (*mat-tensorlist-mult A Ts ds* $ *i*) *is* = (*A* ∗$_v$ (*map-vec* (λ*T*. *Ten-*

sor.lookup T is) *Ts*)) $ *i*
unfolding *mat-tensorlist-mult-def* **using** *lookup-tensor-from-lookup*[*OF* ‹*is* ◁ *ds*›] **using** ‹*i*<*dim-row A*› **by** *auto*
also have ... = *row A i* · *map-vec* (λ*T*. *Tensor.lookup T is*) *Ts*
using ‹*i*<*dim-row A*› **by** *simp*
also have ... = (∑ *j* ∈ {*0* ..< *dim-vec Ts*}. *row A i* $ *j* ∗ (*map-vec* (λ*T*. *Tensor.lookup T is*) *Ts*) $ *j*)
unfolding *scalar-prod-def nth-rows*[*OF* ‹*i*<*dim-row A*›] **by** *simp*
also have ... = (∑*j*∈{*0*..<*dim-vec Ts*}. *lookup* (*A* $$ (*i*, *j*) · *Ts* $ *j*) *is*) **using** *summand-eq* **by** *force*
also have ... = (∑ *A*←*?Ts'*. *lookup A is*) **unfolding** *map-map*
Groups-List.sum-set-upt-conv-sum-list-nat[*symmetric*] *atLeastLessThan-upt*[*symmetric*]
by *auto*
also have ... = *lookup* (*listsum ds ?Ts'*) *is* **using** *lookup-listsum*[*OF* ‹*is* ◁ *ds*›] *dims-Ts'* **by** *fastforce*
finally show *lookup* (*mat-tensorlist-mult A Ts ds* $ *i*) *is* = *lookup* (*listsum ds ?Ts'*) *is* **by** *metis*
qed

lemma *lookup-tensors-from-net*:
assumes *valid-net' m*
and *is* ◁ *input-sizes m*
and *j* < *output-size' m*
shows *Tensor.lookup* (*tensors-from-net m* $ *j*) *is* = *evaluate-net m* (*base-input m is*) $ *j*
using *assms* **proof** (*induction m arbitrary*:*j is*)
case (*Input M*)
then have *j* < *M* **using** *output-size.simps*(*1*) **using** *Input* **by** *auto*
then have *1*:*tensors-from-net* (*Input M*) $ *j* = *unit-vec M j* **by** *simp*
obtain *i* **where** *is* = [*i*] *i*<*M* **using** *Input Suc-length-conv input-sizes.simps*(*1*) *length-0-conv list.size*(*3*) *valid-index-length* **by** *auto*
then have *2*:*Tensor.lookup* (*tensors-from-net* (*Input M*) $ *j*) *is* = (*if i*=*j then 1 else 0*) **using** *lookup-unit-vec 1* **by** *metis*
have *evaluate-net* (*Input M*) (*map* (λ(*n*, *i*). *unit*$_v$ *n i*) (*zip* (*input-sizes* (*Input M*)) *is*)) = *unit*$_v$ *M i* **using** ‹*is* = [*i*]› **by** *auto*
then show *?case* **using** *2* ‹*j* < *M*› *base-input-def* **by** (*simp add*: ‹*i* < *M*›)
next
case (*Conv A m j is*)
have *is-valid*:*is* ◁ *input-sizes m* **using** *Conv.prems* **by** *simp*
have *valid-net*:*valid-net' m* **using** *Conv.prems*(*1*) **unfolding** *remove-weights.simps*
using *valid-net.simps convnet.distinct*(*1*) *convnet.distinct*(*5*) *convnet.inject*(*2*)
by *blast*
then have *length-em*: *dim-vec* (*evaluate-net m* (*base-input m is*)) = *output-size' m*
using *output-size-correct base-input-length is-valid* **by** *metis*

have *IH'*:*map-vec* (λ*T*. *Tensor.lookup T is*) (*tensors-from-net m*) =
evaluate-net m (*base-input m is*)
proof (*rule eq-vecI*)

```
    show equal-lengths: dim-vec (map-vec (λT. lookup T is) (tensors-from-net m))
      = dim-vec (evaluate-net m (base-input m is)) using length-em
      by (simp add: output-size-correct-tensors valid-net)
    show ⋀i. i < dim-vec (evaluate-net m (base-input m is)) ⟹
          map-vec (λT. lookup T is) (tensors-from-net m) $ i = evaluate-net m
(base-input m is) $ i
    proof −
      fix i
      assume i < dim-vec (evaluate-net m (base-input m is))
      then have i < output-size' m using equal-lengths length-em by auto
      then show map-vec (λT. lookup T is) (tensors-from-net m) $ i
        = evaluate-net m (base-input m is) $ i
          using Conv.IH is-valid equal-lengths valid-net base-input-def length-em
nth-map-upt
        length-map nth-map by auto
    qed
  qed

  have Tensor.lookup ((tensors-from-net (Conv A m)) $ j) is =
    (A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
  proof −
    have dim-vec (tensors-from-net (Conv A m)) = output-size' (Conv A m)
      using Conv by (simp add: mat-tensorlist-mult-def)
    then have j<dim-vec (tensors-from-net (Conv A m)) using Conv.prems by
auto
   then have (tensors-from-net (Conv A m)) $ j = tensor-from-lookup (input-sizes
m)
              (λis. (A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m)))
$ j)
      unfolding tensors-from-net.simps mat-tensorlist-mult-def by fastforce
    then show ?thesis
      using lookup-tensor-from-lookup[OF is-valid] by auto
  qed
  also have (A *v (map-vec (λT. Tensor.lookup T is) (tensors-from-net m))) $ j
    = (A *v (evaluate-net m (base-input m is))) $ j using IH' by auto
  also have ... = evaluate-net (Conv A m) (base-input (Conv A m) is) $ j
    unfolding base-input-def using evaluate-net.simps by auto
  finally show ?case by auto
next
  case (Pool m1 m2 j is)
```

We split "is" into two parts for each subnet:

```
  obtain is1 is2 where is12-def:is = is1 @ is2 is1 ◁ input-sizes m1 is2 ◁
input-sizes m2
    by (metis Pool.prems(2) input-sizes.simps(3) valid-index-split)
```

Apply the induction hypothesis to the subnets:

```
  have IH:Tensor.lookup (tensors-from-net m1 $ j) is1
      = evaluate-net m1 (map (λ(x, y). unitv x y) (zip (input-sizes m1) is1)) $ j
```

Tensor.lookup (*tensors-from-net m2* $ *j*) *is2*
= *evaluate-net m2* (*map* (λ(*x*, *y*). *unit*$_v$ *x y*) (*zip* (*input-sizes m2*) *is2*)) $ *j*
using *Pool convnet.distinct*(*3*) *convnet.distinct*(*5*) *convnet.inject*(*3*) *remove-weights.simps*(*3*)
valid-net.simps ‹*is1* ◁ *input-sizes m1*› ‹*is2* ◁ *input-sizes m2*› *output-size.simps*(*3*)
by (*metis base-input-def*)+

In the Pool layer tensor entries get multiplied:

have *lookup-prod*: *Tensor.lookup* (*tensors-from-net* (*Pool m1 m2*) $ *j*) *is*
= *Tensor.lookup* (*tensors-from-net m1* $ *j*) *is1* ∗ *Tensor.lookup* (*tensors-from-net m2* $ *j*) *is2*
proof −
have *j-small*: *j* < *dim-vec* (*tensors-from-net m1*) *j* < *dim-vec* (*tensors-from-net m2*)
by (*metis Pool.prems*(*1*) *Pool.prems*(*3*) *convnet.distinct*(*3*) *convnet.inject*(*3*) *convnet.simps*(*9*)
output-size.simps(*3*) *output-size-correct-tensors remove-weights.simps*(*3*) *valid-net.cases*)+
then have *0*:*tensors-from-net* (*Pool m1 m2*) $ *j* = *tensors-from-net m1* $ *j* ⊗ *tensors-from-net m2* $ *j*
unfolding *tensors-from-net.simps* **using** *j-small index-component-mult* **by** *blast*
have *Tensor.dims* (*tensors-from-net m1* $ *j*) = *input-sizes m1*
Tensor.dims (*tensors-from-net m2* $ *j*) = *input-sizes m2*
using *dims-tensors-from-net j-small nth-mem* **by** (*simp-all add*: *vec-setI*)
then have *is12-valid*:
is1 ◁ *Tensor.dims* (*tensors-from-net m1* $ *j*)
is2 ◁ *Tensor.dims* (*tensors-from-net m2* $ *j*)
using *is12-def* **by** *presburger*+
then show *?thesis*
unfolding *0* **using** *lookup-tensor-prod*[*OF is12-valid*] *is12-def* **by** *auto*
qed

Output values get multiplied in the Pool layer as well:

have *evaluate-net* (*Pool m1 m2*) (*base-input* (*Pool m1 m2*) *is*) $ *j*
= *evaluate-net m1* (*base-input m1 is1*) $ *j* ∗ *evaluate-net m2* (*base-input m2 is2*) $ *j*
proof −
have *valid-net' m1 valid-net' m2*
using *remove-weights.simps valid-net.simps Pool.prems*
by (*metis convnet.distinct*(*3*) *convnet.distinct*(*5*) *convnet.inject*(*3*))+
have *input-sizes m1* = *map dim-vec* (*base-input m1 is1*)
input-sizes m2 = *map dim-vec* (*base-input m2 is2*)
using *base-input-def base-input-length base-input-def is12-def* **by** *auto*
have *j* < *dim-vec* (*evaluate-net m1* (*base-input m1 is1*)) *j* < *dim-vec* (*evaluate-net m2* (*base-input m2 is2*))
using *Pool.prems* ‹*input-sizes m1* = *map dim-vec* (*base-input m1 is1*)› ‹*valid-net' m1*›
output-size-correct **by** (*auto*,*metis Pool.prems*(*1*) *Pool.prems*(*3*) ‹*input-sizes m2* = *map dim-vec* (*base-input m2 is2*)›
convnet.distinct(*3*) *convnet.distinct*(*5*) *convnet.inject*(*3*) *output-size.simps*(*3*) *output-size-correct*

remove-weights.simps(*3*) *valid-net.cases*)
then show *?thesis* **unfolding** *evaluate-net.simps* **unfolding** *base-input-def*
using *is12-def*(*1*) *is12-def*(*2*) *valid-index-length* **by** (*simp add*: *append-eq-conv-conj drop-map*
drop-zip index-component-mult input-sizes-remove-weights take-map take-zip)
qed

then show *?case* **using** *lookup-prod IH base-input-def* **by** *auto*
qed

lemma *insert-remove-weights*:
obtains *w* **where** *m* = *insert-weights* (*remove-weights m*) *w*
proof (*induction m arbitrary*:*thesis*)
case (*Input m thesis*)
then show *?case* **by** *simp*
next
case (*Conv A m thesis*)
then obtain *w* **where** *m* = *insert-weights* (*remove-weights m*) *w* **by** *auto*
then have *1*:*remove-weights* (*Conv A m*) = *Conv* (*dim-row A*, *dim-col A*) (*remove-weights m*) **by** *simp*
have *Conv A m* = *insert-weights* (*remove-weights* (*Conv A m*)) (λ*i*. *if i*<*dim-row A* ∗*dim-col A then flatten-matrix A i else w* (*i*−*dim-row A* ∗*dim-col A*))
unfolding *1 insert-weights.simps*
using *extract-matrix-flatten-matrix*[*of A*] *extract-matrix-cong*[*of dim-row A dim-col A*
λ*i*. *if i* < *dim-row A* ∗ *dim-col A then flatten-matrix A i else w* (*i* − *dim-row A* ∗ *dim-col A*) *flatten-matrix A*]
using ‹*m* = *insert-weights* (*remove-weights m*) *w*› **by** *fastforce*
then show *?case* **using** *Conv.prems* **by** *blast*
next
case (*Pool m1 m2*)
then obtain *w1 w2* **where** *m1* = *insert-weights* (*remove-weights m1*) *w1 m2* = *insert-weights* (*remove-weights m2*) *w2* **by** *metis*
then have *Pool m1 m2* = *insert-weights* (*remove-weights* (*Pool m1 m2*)) (λ*i*. *if i*<*count-weights* (*remove-weights m1*) *then w1 i else w2* (*i* − *count-weights* (*remove-weights m1*)))
unfolding *remove-weights.simps insert-weights.simps*
using *insert-weights-cong*[*of* - λ*i*. *if i* < *count-weights* (*remove-weights m1*) *then w1 i else w2* (*i* − *count-weights* (*remove-weights m1*)) *w1*] **by** *fastforce*
then show *?case* **unfolding** *Pool* **using** *Pool.prems* **by** *blast*
qed

lemma *remove-insert-weights*:
shows *remove-weights* (*insert-weights m w*) = *m*
proof (*induction m arbitrary*:*w*)
case *Input*
then show *?case* **by** *simp*
next
case (*Conv r12 m*)

then obtain *r1 r2* **where** *r12 = (r1, r2)* **by** *fastforce*
then have *remove-weights (insert-weights m w) = m* **using** *Conv.IH* **by** *blast*
then have *remove-weights (insert-weights (Conv (r1,r2) m) w) = Conv (r1,r2) m*
unfolding *insert-weights.simps remove-weights.simps*
using *extract-matrix-def Conv.IH dim-extract-matrix(1)* **by** (*metis dim-col-mat(1)*)
then show *?case* **using** ‹*r12 = (r1, r2)*› **by** *blast*
next
case (*Pool m1 m2 w*)
then show *?case* **unfolding** *insert-weights.simps remove-weights.simps* **using** *Pool.IH* **by** *blast*
qed

lemma *finite-valid-index*: *finite* {*is*. *is* ◁ *ds*}
proof (*induction ds*)
case *Nil*
then show *?case* **by** (*metis List.finite-set finite-subset length-0-conv list.set-intros(1) mem-Collect-eq subsetI valid-index-length*)
next
case (*Cons d ds*)
have {*is*. *is* ◁ *d* # *ds*} ⊆ (⋃*i*<*d*. {*i* # *is* |*is*. *is* ◁ *ds*})
proof (*rule subsetI*)
fix *is* **assume** *is* ∈ {*is*. *is* ◁ *d* # *ds*}
then have *is* ◁ *d* # *ds* **by** *auto*
then obtain *i is′* **where** *is = i* # *is′* **by** *blast*
then have *i*<*d* **using** ‹*is* ◁ *d* # *ds*› **by** *blast*
have *is′* ◁ *ds* **using** ‹*is = i* # *is′*› ‹*is* ◁ *d* # *ds*› **by** *blast*
have *is* ∈ {*i* # *is* |*is*. *is* ◁ *ds*} **by** (*simp add*: ‹*is = i* # *is′*› ‹*is′* ◁ *ds*›)
then show *is* ∈ (⋃*i*<*d*. {*i* # *is* |*is*. *is* ◁ *ds*}) **using** ‹*i* < *d*› **by** *blast*
qed
moreover have ⋀*i*. *finite* {*i* # *is* |*is*. *is* ◁ *ds*} **by** (*simp add*: *Cons.IH*)
ultimately show *finite* {*is*. *is* ◁ *d* # *ds*} **by** (*simp add*: *finite-subset*)
qed

lemma *setsum-valid-index-split*:
(∑ *is* | *is* ◁ *ds1* @ *ds2*. *f is*) = (∑ *is1* | *is1* ◁ *ds1*. (∑ *is2* | *is2* ◁ *ds2*. *f* (*is1* @ *is2*)))
proof −
have *1*:((λ(*is1*, *is2*). *is1* @ *is2*) ‘ ({*is1*. *is1* ◁ *ds1*} × {*is2*. *is2* ◁ *ds2*})) = {*is*. *is* ◁ *ds1* @ *ds2*} (**is** *?A = ?B*)
proof (*rule subset-antisym*; *rule subsetI*)
fix *x* **assume** *x* ∈ *?A*
then show *x* ∈ *?B* **using** *valid-index-append* **by** *auto*
next
fix *x* **assume** *x* ∈ *?B*
then have *x* ◁ *ds1* @ *ds2* **by** *auto*
then obtain *x1 x2* **where** *x = x1* @ *x2 x1* ◁ *ds1 x2* ◁ *ds2* **by** (*metis valid-index-split*)

then have (*x1*, *x2*) ∈ ({*is1*. *is1* ◁ *ds1*} × {*is2*. *is2* ◁ *ds2*}) **by** *auto*
then show *x* ∈ *?A* **using** *imageI* ‹*x* = *x1* @ *x2*› **by** *blast*
qed
have *2*:*inj-on* (λ(*is1*, *is2*). *is1* @ *is2*) ({*is1*. *is1* ◁ *ds1*} × {*is2*. *is2* ◁ *ds2*})
by (*simp add*: *inj-on-def valid-index-length*)
show *?thesis*
unfolding *Groups-Big.comm-monoid-add-class.sum.cartesian-product*[*of* λ*is1 is2*. *f* (*is1* @ *is2*)]
using *Groups-Big.comm-monoid-add-class.sum.reindex*[*OF 2*, *of f*] *1 2 SigmaE prod.simps*(*2*) *sum.reindex-cong* **by** (*simp add*: *split-def*)
qed

lemma *prod-lessThan-split*:
fixes *g* :: *nat* ⇒ *real* **shows** *prod g* {..<*n*+*m*} = *prod g* {..<*n*} ∗ *prod* (λ*x*. *g* (*x*+*n*)) {..<*m*}
using *Groups-Big.comm-monoid-mult-class.prod.union-inter-neutral*[*of* {..<*n*} {*n*..<*n*+*m*} *g*, *unfolded ivl-disj-un-one*(*2*)[*OF le-add1*], *OF finite-lessThan finite-atLeastLessThan*]
by (*metis* (*no-types*) *add.commute add.left-neutral atLeast0LessThan empty-iff ivl-disj-int-one*(*2*) *prod-shift-bounds-nat-ivl*)

lemma *evaluate-net-from-tensors*:
assumes *valid-net'* *m*
and *map dim-vec inputs* = *input-sizes m*
and *j* < *output-size'* *m*
shows *evaluate-net m inputs* $ *j*
= (∑ *is*∈{*is*. *is* ◁ *input-sizes m*}. (∏ *k*<*length inputs*. *inputs* ! *k* $ (*is*!*k*)) ∗ *Tensor.lookup* (*tensors-from-net m* $ *j*) *is*)
using *assms* **proof** (*induction m arbitrary*:*j is inputs*)
case (*Input M*)
then have *length inputs* = *1 input-sizes* (*Input M*) = [*M*] **by** *auto*
{
fix *is* **assume** *is* ◁ *input-sizes* (*Input M*)
then have *length is* = *1* **by** (*simp add*: *valid-index-length*)
then have *is* = [*hd is*] **by** (*metis One-nat-def length-0-conv length-Suc-conv list.sel*(*1*))
then have *Tensor.lookup* (*tensors-from-net* (*Input M*) $ *j*) *is* = (*if hd is*=*j then 1 else 0*)
by (*metis Input.prems*(*3*) ‹*input-sizes* (*Input M*) = [*M*]› ‹*is* ◁ *input-sizes* (*Input M*)› *list.distinct*(*1*)
lookup-unit-vec nth-Cons-0 output-size.simps(*1*) *remove-weights.simps*(*1*) *tensors-from-net.simps*(*1*) *valid-indexE index-vec*)
then have (∏ *k*<*length inputs*. *inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net* (*Input M*) $ *j*) *is* =
(*if is*=[*j*] *then* (∏ *k*<*length inputs*. *inputs* ! *k* $ (*is* ! *k*)) *else 0*) **using** ‹*is* = [*hd is*]› **by** *auto*
}
then have (∑ *is* | *is* ◁ *input-sizes* (*Input M*). (∏ *k*<*length inputs*. *inputs* ! *k* $ (*is* ! *k*)) ∗ *lookup* (*tensors-from-net* (*Input M*) $ *j*) *is*)

$= (\sum is \mid is \lhd input\text{-}sizes\ (Input\ M).\ (if\ is{=}[j]\ then\ (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k))\ else\ 0))$ **by** *auto*
also have $(\sum is \mid is \lhd input\text{-}sizes\ (Input\ M).\ (if\ is{=}[j]\ then\ (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k))\ else\ 0))$
$= (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ ([j]\ !\ k))$ **unfolding** *sum.delta*[*OF finite-valid-index*]
using *Input.prems*(3) *valid-index.Cons valid-index.Nil* **by** *auto*
also have $... = inputs\ !\ 0\ \$\ j$ **using** ‹*length inputs = 1*› **by** (*simp add*: *prod-lessThan-Suc*)
also have $... = evaluate\text{-}net\ (Input\ M)\ inputs\ \$\ j$ **unfolding** *evaluate-net.simps*
by (*metis* ‹*length inputs = 1*› *hd-conv-nth list.size*(3) *zero-neq-one*)
finally show *?case* **by** *auto*
next
case (*Conv A m j*)
have $j < dim\text{-}row\ A$ **using** *Conv.prems*(3) **by** *auto*
have $0{:}\bigwedge is.\ is \lhd input\text{-}sizes\ (Conv\ A\ m) \Longrightarrow$
$(\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ (Conv\ A\ m)\ \$\ j)\ is =$
$(\sum i = 0..{<}dim\text{-}vec\ (tensors\text{-}from\text{-}net\ m).\ row\ A\ j\ \$\ i * ((\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m\ \$\ i)\ is))$
proof −
fix *is* **assume** $is \lhd input\text{-}sizes\ (Conv\ A\ m)$
then have $is \lhd input\text{-}sizes\ m$ **by** *simp*
have $0{:}lookup\ (tensors\text{-}from\text{-}net\ (Conv\ A\ m)\ \$\ j)\ is =$
$(\sum i = 0..{<}dim\text{-}vec\ (tensors\text{-}from\text{-}net\ m).\ row\ A\ j\ \$\ i * lookup\ (tensors\text{-}from\text{-}net\ m\ \$\ i)\ is)$
unfolding *tensors-from-net.simps mat-tensorlist-mult-def index-vec*[*OF* ‹*j < dim-row A*›]
lookup-tensor-from-lookup[*OF* ‹*is* ⊲ *input-sizes m*›] *index-mult-mat-vec*[*OF* ‹*j < dim-row A*›] *scalar-prod-def*
using *index-map-vec* **by** *auto*
show $(\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ (Conv\ A\ m)\ \$\ j)\ is$
$= (\sum i = 0..{<}dim\text{-}vec\ (tensors\text{-}from\text{-}net\ m).\ row\ A\ j\ \$\ i * ((\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m\ \$\ i)\ is))$
unfolding *0 sum-distrib-left* **by** (*simp add*: *semiring-normalization-rules*(19))
qed
have *valid-net' m* **by** (*metis Conv.prems*(1) *convnet.distinct*(1) *convnet.distinct*(5) *convnet.inject*(2) *remove-weights.simps*(2) *valid-net.simps*)
have *map dim-vec inputs = input-sizes m* **by** (*simp add*: *Conv.prems*(2))
have *output-size' m = dim-vec* (*tensors-from-net m*) **by** (*simp add*: ‹*valid-net' m*› *output-size-correct-tensors*)
have $1{:}\bigwedge i.\ i{<}dim\text{-}vec\ (tensors\text{-}from\text{-}net\ m) \Longrightarrow (\sum is \mid is \lhd input\text{-}sizes\ (Conv\ A\ m).\ ((\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m\ \$\ i)\ is)) = evaluate\text{-}net\ m\ inputs\ \$\ i$ **unfolding** *input-sizes.simps*
using *Conv.IH* ‹*valid-net' m*› ‹*map dim-vec inputs = input-sizes m*› ‹*output-size' m = dim-vec* (*tensors-from-net m*)› **by** *simp*

have $(\sum is \mid is \lhd input\text{-}sizes\ (Conv\ A\ m).\ (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ (Conv\ A\ m)\ \$\ j)\ is)$
$= (\sum i = 0..{<}dim\text{-}vec\ (tensors\text{-}from\text{-}net\ m).\ (\sum is \mid is \lhd input\text{-}sizes\ (Conv$

A m). *row A j $ i* * ((∏ *k*<*length inputs*. *inputs* ! *k* $ (*is* ! *k*)) * *lookup* (*tensors-from-net m* $ *i*) *is*)))
using *Groups-Big.comm-monoid-add-class.sum.commute 0* **by** *auto*
also have ... = (∑ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* * (∑ *is* | *is* ◁ *input-sizes* (*Conv A m*). ((∏ *k*<*length inputs*. *inputs* ! *k* $ (*is* ! *k*)) * *lookup* (*tensors-from-net m* $ *i*) *is*)))
by (*simp add*: *sum-distrib-left*)
also have ... = (∑ *i* = *0*..<*dim-vec* (*tensors-from-net m*). *row A j* $ *i* * *evaluate-net m inputs* $ *i*) **using** *1* **by** *auto*
also have ... = *row A j* · *evaluate-net m inputs*
by (*metis* (*full-types*) ‹*map dim-vec inputs* = *input-sizes m*› ‹*output-size*′ *m* = *dim-vec* (*tensors-from-net m*)›
‹*valid-net*′ *m*› *output-size-correct scalar-prod-def*)
also have ... = (*A* *$_v$ *evaluate-net m inputs*) $ *j* **by** (*simp add*: ‹*j* < *dim-row A*›)
also have ... = *evaluate-net* (*Conv A m*) *inputs* $ *j* **by** *simp*
finally show *?case* **by** *auto*
next
case (*Pool m1 m2 j*)
have *valid-net*′ *m1 valid-net*′ *m2*
by (*metis Pool.prems*(*1*) *convnet.distinct*(*3*) *convnet.inject*(*3*) *convnet.simps*(*9*) *remove-weights.simps*(*3*) *valid-net.simps*)+
have *j* < *output-size*′ *m2 j* < *output-size*′ *m1*
apply (*metis Pool.prems*(*1*) *Pool.prems*(*3*) *convnet.distinct*(*3*) *convnet.inject*(*3*) *convnet.simps*(*9*)
output-size.simps(*3*) *remove-weights.simps*(*3*) *valid-net.simps*) **using** *Pool.prems* **by** *auto*
then have *j* < *dim-vec* (*tensors-from-net m1*) *j* < *dim-vec* (*tensors-from-net m2*)
by (*simp-all add*: ‹*valid-net*′ *m1*› ‹*valid-net*′ *m2*› *output-size-correct-tensors*)

def *inputs1* == *take* (*length* (*input-sizes m1*)) *inputs*
def *inputs2* == *drop* (*length* (*input-sizes m1*)) *inputs*
have *map dim-vec inputs1* = *input-sizes m1 map dim-vec inputs2* = *input-sizes m2*
apply (*metis Pool.prems*(*2*) *append-eq-conv-conj input-sizes.simps*(*3*) *inputs1-def take-map*)
by (*metis Pool.prems*(*2*) *append-eq-conv-conj drop-map input-sizes.simps*(*3*) *inputs2-def*)
have *inputs* = *inputs1* @ *inputs2* **by** (*simp add*: *inputs1-def inputs2-def*)
{
fix *is1 is2* **assume** *is1* ◁ *input-sizes m1 is2* ◁ *input-sizes m2*
have *length is1* = *length inputs1*
using ‹*is1* ◁ *input-sizes m1*› ‹*map dim-vec inputs1* = *input-sizes m1*› *valid-index-length* **by** *fastforce*
have *length is2* = *length inputs2*
using ‹*is2* ◁ *input-sizes m2*› ‹*map dim-vec inputs2* = *input-sizes m2*› *valid-index-length* **by** *fastforce*
have *1*:(∏ *k*<*length inputs1*. (*inputs1* @ *inputs2*) ! *k* $ ((*is1* @ *is2*) ! *k*)) =

$(\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k))$
using ‹*length is1 = length inputs1*› ‹*length is2 = length inputs2*›
nth-append **by** (*metis* (*no-types*, *lifting*) *lessThan-iff prod.cong*)
have *2*:$(\prod x{<}length\ inputs2.\ (inputs1\ @\ inputs2)\ !\ (x + length\ inputs1)\ \$\ ((is1\ @\ is2)\ !\ (x + length\ inputs1))) =$
$(\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k))$
using ‹*length is1 = length inputs1*› ‹*length is2 = length inputs2*›
by (*metis* (*no-types*, *lifting*) *add.commute nth-append-length-plus*)
have $(\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ ((is1\ @\ is2)\ !\ k)) = (\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k))$
unfolding ‹*inputs = inputs1 @ inputs2*› *length-append prod-lessThan-split*
using *1 2* **by** *metis*
}
note *1 = this*
{
fix *is1 is2* **assume** *is1 ◁ input-sizes m1 is2 ◁ input-sizes m2*
then have *is1 ◁ dims* (*tensors-from-net m1 $ j*) *is2 ◁ dims* (*tensors-from-net m2 $ j*)
using ‹*j < dim-vec* (*tensors-from-net m1*)› ‹*j < dim-vec* (*tensors-from-net m2*)› *dims-tensors-from-net vec-setI* **by** *force+*
have *lookup* (*tensors-from-net* (*Pool m1 m2*) $ *j*) (*is1 @ is2*) = *lookup* (*tensors-from-net m1 $ j*) *is1 * lookup* (*tensors-from-net m2 $ j*) *is2*
unfolding *tensors-from-net.simps index-component-mult*[*OF* ‹*j < dim-vec* (*tensors-from-net m1*)› ‹*j < dim-vec* (*tensors-from-net m2*)›]
lookup-tensor-prod[*OF* ‹*is1 ◁ dims* (*tensors-from-net m1 $ j*)› ‹*is2 ◁ dims* (*tensors-from-net m2 $ j*)›] **by** *metis*
}
note *2 = this*

have *j-le-eval*:*j < dim-vec* (*evaluate-net m1* (*take* (*length* (*input-sizes m1*)) *inputs*))
j < dim-vec (*evaluate-net m2* (*drop* (*length* (*input-sizes m1*)) *inputs*))
using ‹*j < output-size' m1*› ‹*map dim-vec inputs1 = input-sizes m1*› ‹*valid-net' m1*› *inputs1-def output-size-correct*
using ‹*j < output-size' m2*› ‹*map dim-vec inputs2 = input-sizes m2*› ‹*valid-net' m2*› *inputs2-def* **by** *auto*
have $(\sum is \mid is \lhd input\text{-}sizes\ (Pool\ m1\ m2).\ (\prod k{<}length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ (Pool\ m1\ m2)\ \$\ j)\ is) = (\sum is1 \mid is1 \lhd input\text{-}sizes\ m1.\ \sum is2 \mid is2 \lhd input\text{-}sizes\ m2.\ (\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m1\ \$\ j)\ is1 * lookup\ (tensors\text{-}from\text{-}net\ m2\ \$\ j)\ is2)$
unfolding *input-sizes.simps setsum-valid-index-split* **using** *1 2*
using *mem-Collect-eq sum.cong* **by** (*simp add*: *mult.assoc*)
also have ... $= (\sum is1 \mid is1 \lhd input\text{-}sizes\ m1.\ (\prod k{<}length\ inputs1.\ inputs1\ !\ k\ \$\ (is1\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m1\ \$\ j)\ is1) * (\sum is2 \mid is2 \lhd input\text{-}sizes\ m2.\ (\prod k{<}length\ inputs2.\ inputs2\ !\ k\ \$\ (is2\ !\ k)) * lookup\ (tensors\text{-}from\text{-}net\ m2\ \$\ j)\ is2)$

unfolding *sum-product* **by** (*rule sum.cong*, *metis*, *rule sum.cong*, *metis*, *simp*)
also have ... = *evaluate-net* (*Pool m1 m2*) *inputs* $ *j* **unfolding** *evaluate-net.simps index-component-mult*[*OF j-le-eval*]
using *Pool.IH*(*1*)[*OF* ‹*valid-net' m1*› - ‹*j* < *output-size' m1*›] *Pool.IH*(*2*)[*OF* ‹*valid-net' m2*› - ‹*j* < *output-size' m2*›]
using ‹*map dim-vec inputs1 = input-sizes m1*› ‹*map dim-vec inputs2 = input-sizes m2*› *inputs1-def inputs2-def* **by** *auto*
finally show *?case* **by** *metis*
qed

lemma *tensors-from-net-eqI*:
assumes *valid-net' m1 valid-net' m2 input-sizes m1 = input-sizes m2*
assumes ⋀*inputs*. *input-sizes m1 = map dim-vec inputs* ⟹ *evaluate-net m1 inputs = evaluate-net m2 inputs*
shows *tensors-from-net m1 = tensors-from-net m2*
proof −
have *map dim-vec* (*map* 0_v (*input-sizes m2*)) = *input-sizes m2*
map dim-vec (*map* 0_v (*input-sizes m1*)) = *input-sizes m1* **by** (*simp-all add*: *nth-equalityI*)
then have *output-size' m1 = output-size' m2* **using**
output-size-correct[*OF* ‹*valid-net' m1*› ‹*map dim-vec* (*map* 0_v (*input-sizes m1*)) = *input-sizes m1*›]
output-size-correct[*OF* ‹*valid-net' m2*› ‹*map dim-vec* (*map* 0_v (*input-sizes m2*)) = *input-sizes m2*›]
assms(*3*) *assms*(*4*)
by (*metis* (*no-types*))
have ⋀*is*. *base-input m1 is = base-input m2 is*
unfolding *base-input-def* ‹*input-sizes m1 = input-sizes m2*› **by** *metis*
show *?thesis* **by** (*rule eq-vecI*, *rule tensor-lookup-eqI*; *metis*
lookup-tensors-from-net[*OF* ‹*valid-net' m1*›, *unfolded* ‹⋀*is*. *base-input m1 is = base-input m2 is*› ‹*output-size' m1 = output-size' m2*›]
lookup-tensors-from-net[*OF* ‹*valid-net' m2*›] *assms*(*3*) *base-input-length*
assms(*1*) *assms*(*2*) *dims-tensors-from-net output-size-correct-tensors vec-setI*
‹*output-size' m1 = output-size' m2*› *assms*(*4*))
qed

end

24 Concrete Matrices

theory *DL-Concrete-Matrices*
imports *HOL.Real Jordan-Normal-Form.Matrix DL-Missing-Matrix*
begin

The following definition allows non-square-matrices, mat_one (mat_one n) only allows square matrices.

definition *eye-matrix*::*nat* ⇒ *nat* ⇒ *real mat*
where *eye-matrix nr nc* = *mat nr nc* (λ(*r*, *c*). *if r=c then 1 else 0*)

lemma *eye-matrix-dim*: *dim-row* (*eye-matrix nr nc*) = *nr dim-col* (*eye-matrix nr nc*) = *nc* **by** (*simp-all add*: *eye-matrix-def*)

lemma *row-eye-matrix*:
assumes *i* < *nr*
shows *row* (*eye-matrix nr nc*) *i* = *unit-vec nc i*
by (*rule eq-vecI*, *simp add*: *assms eye-matrix-def unit-vec-def*, *simp add*: *eye-matrix-dim*(*2*))

lemma *unit-eq-0*[*simp*]:
assumes *i*: *i* ≥ *n*
shows *unit-vec n i* = 0_v *n*
by (*rule eq-vecI*, *insert i*, *auto simp*: *unit-vec-def*)

lemma *mult-eye-matrix*:
assumes *i* < *nr*
shows (*eye-matrix nr* (*dim-vec v*) $*_v$ *v*) $ *i* = (*if i*<*dim-vec v then v* $ *i else 0*)
(**is** *?a* $ *i* = *?b*)
proof −
have *?a* $ *i* = *row* (*eye-matrix nr* (*dim-vec v*)) *i* · *v* **using** *index-mult-mat-vec assms eye-matrix-dim* **by** *auto*
also have ... = *unit-vec* (*dim-vec v*) *i* · *v* **using** *row-eye-matrix assms* **by** *auto*
also have ... = *?b* **using** *scalar-prod-left-unit carrier-vecI unit-eq-0 scalar-prod-left-zero* **by** *fastforce*
finally show *?thesis* **by** *auto*
qed

definition *all1-vec*::*nat* ⇒ *real vec*
where *all1-vec n* = *vec n* (λ*i*. *1*)

definition *all1-matrix*::*nat* ⇒ *nat* ⇒ *real mat*
where *all1-matrix nr nc* = *mat nr nc* (λ(*r*, *c*). *1*)

lemma *all1-matrix-dim*: *dim-row* (*all1-matrix nr nc*) = *nr dim-col* (*all1-matrix nr nc*) = *nc*
by (*simp-all add*: *all1-matrix-def*)

lemma *row-all1-matrix*:
assumes *i* < *nr*
shows *row* (*all1-matrix nr nc*) *i* = *all1-vec nc*
apply (*rule eq-vecI*)
apply (*simp add*: *all1-matrix-def all1-vec-def assms*)
by (*simp add*: *all1-matrix-def all1-vec-def*)

lemma *all1-vec-scalar-prod*:
shows *all1-vec* (*length xs*) · (*vec-of-list xs*) = *sum-list xs*
proof −
have *all1-vec* (*length xs*) · (*vec-of-list xs*) = (∑ *i* = *0*..<*dim-vec* (*vec-of-list xs*). *vec-of-list xs* $ *i*)

unfolding *scalar-prod-def* **by** (*metis* (*no-types*, *lifting*) *all1-vec-def mult-cancel-right1 sum-ivl-cong*
vec.abs-eq dim-vec index-vec vec-of-list.abs-eq)
also have ... = ($\sum$ *i* = *0*..<*length xs*. *xs* ! *i*) **using** *vec.abs-eq dim-vec vec-of-list.abs-eq*
by (*metis sum-ivl-cong index-vec*)
also have ... = *sum-list xs* **by** (*simp add*: *sum-list-sum-nth*)
finally show *?thesis* **by** *auto*
qed

lemma *mult-all1-matrix*:
assumes *i* < *nr*
shows ((*all1-matrix nr* (*dim-vec v*)) $*_v$ *v*) $ *i* = *sum-list* (*list-of-vec v*) (**is** *?a* $ *i* = *sum-list* (*list-of-vec v*))
proof −
have *?a* $ *i* = *row* (*all1-matrix nr* (*dim-vec v*)) *i* · *v* **using** *index-mult-mat-vec assms all1-matrix-dim* **by** *auto*
also have ... = *sum-list* (*list-of-vec v*) **unfolding** *row-all1-matrix*[*OF assms*] **using** *all1-vec-scalar-prod*[*of list-of-vec v*]
by (*metis vec.abs-eq dim-vec vec-list vec-of-list.abs-eq*)
finally show *?thesis* **by** *auto*
qed

definition *copy-first-matrix*::*nat* ⇒ *nat* ⇒ *real mat*
where *copy-first-matrix nr nc* = *mat nr nc* (λ(*r*, *c*). *if c* = *0 then 1 else 0*)

lemma *copy-first-matrix-dim*: *dim-row* (*copy-first-matrix nr nc*) = *nr dim-col* (*copy-first-matrix nr nc*) = *nc*
by (*simp-all add*: *copy-first-matrix-def*)

lemma *row-copy-first-matrix*:
assumes *i* < *nr*
shows *row* (*copy-first-matrix nr nc*) *i* = *unit-vec nc 0*
apply (*rule eq-vecI*)
apply (*auto simp add*: *copy-first-matrix-def assms*)[*1*]
by (*simp add*: *copy-first-matrix-def*)

lemma *mult-copy-first-matrix*:
assumes *i* < *nr* **and** *dim-vec v* > *0*
shows (*copy-first-matrix nr* (*dim-vec v*) $*_v$ *v*) $ *i* = *v* $ *0* (**is** *?a* $ *i* = *v* $ *0*)
proof −
have *?a* $ *i* = *row* (*copy-first-matrix nr* (*dim-vec v*)) *i* · *v* **using** *index-mult-mat-vec assms copy-first-matrix-dim* **by** *auto*
also have ... = *unit-vec* (*dim-vec v*) *0* · *v* **using** *row-copy-first-matrix assms* **by** *auto*
also have ... = *v* $ *0* **using** *assms*(*2*) *scalar-prod-left-unit carrier-dim-vec* **by** *blast*
finally show *?thesis* **by** *auto*

qed

end

25 Missing Lemmas of Finite_Set

theory *DL-Missing-Finite-Set*
imports *Main*
begin

lemma *card-even*[*simp*]: *card* $\{a \in Collect\ even.\ a < 2 * n\} = n$
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have $\{a \in Collect\ even.\ a < 2 * Suc\ n\} = insert\ (2*n)\ \{a \in Collect\ even.\ a < 2 * n\}$
    using *le-eq-less-or-eq less-Suc-eq-le subset-antisym* **by** *force*
  show *?case*
    unfolding ‹$\{a \in Collect\ even.\ a < 2 * Suc\ n\} = insert\ (2*n)\ \{a \in Collect\ even.\ a < 2 * n\}$›
    using *Suc card-insert-disjoint*[*of* $\{a \in Collect\ even.\ a < 2 * n\}$ *2*n*]
    by (*simp add*: *finite-M-bounded-by-nat less-not-refl2*)
qed

lemma *card-odd*[*simp*]: *card* $\{a \in Collect\ odd.\ a < 2 * n\} = n$
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have $\{a \in Collect\ odd.\ a < 2 * Suc\ n\} = insert\ (2*n+1)\ \{a \in Collect\ odd.\ a < 2 * n\}$
    using *le-eq-less-or-eq less-Suc-eq-le subset-antisym* **by** *force*
  show *?case*
    unfolding ‹$\{a \in Collect\ odd.\ a < 2 * Suc\ n\} = insert\ (2*n+1)\ \{a \in Collect\ odd.\ a < 2 * n\}$›
    using *Suc card-insert-disjoint*[*of* $\{a \in Collect\ even.\ a < 2 * n\}$ *2*n*]
    by (*simp add*: *finite-M-bounded-by-nat less-not-refl2*)
qed

end

26 Deep Network Model

theory *DL-Deep-Model*
imports *DL-Network Tensor-Matricization DL-Submatrix DL-Concrete-Matrices*

DL-Missing-Finite-Set DL-Missing-Sublist Jordan-Normal-Form.Determinant
begin

hide-const(**open**) *Polynomial.order*

fun *deep-model* **and** *deep-model'* **where**
deep-model' Y [] = *Input Y* |
deep-model' Y (*r* # *rs*) = *Pool* (*deep-model Y r rs*) (*deep-model Y r rs*) |
deep-model Y r rs = *Conv* (*Y*,*r*) (*deep-model' r rs*)

abbreviation *deep-model'-l rs* == *deep-model'* (*rs*!*0*) (*tl rs*)
abbreviation *deep-model-l rs* == *deep-model* (*rs*!*0*) (*rs*!*1*) (*tl* (*tl rs*))

lemma *valid-deep-model*: *valid-net* (*deep-model Y r rs*)
apply (*induction rs arbitrary*: *Y r*)
apply (*simp add*: *valid-net.intros*(*1*) *valid-net.intros*(*2*))
using *valid-net.intros*(*2*) *valid-net.intros*(*3*) **by** *auto*

lemma *valid-deep-model'*: *valid-net* (*deep-model' r rs*)
apply (*induction rs arbitrary*: *r*)
apply (*simp add*: *valid-net.intros*(*1*))
by (*metis deep-model'.elims deep-model'.simps*(*2*) *deep-model.elims output-size.simps valid-net.simps*)

lemma *input-sizes-deep-model'*:
assumes *length rs* ≥ *1*
shows *input-sizes* (*deep-model'-l rs*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
using *assms* **proof** (*induction butlast rs arbitrary*:*rs*)
  case *Nil*
  then have *rs* = [*rs*!*0*]
    by (*metis One-nat-def diff-diff-cancel diff-zero length-0-conv length-Suc-conv length-butlast nth-Cons-0*)
  then have *input-sizes* (*deep-model'-l rs*) = [*last rs*]
    by (*metis deep-model'.simps*(*1*) *input-sizes.simps*(*1*) *last.simps list.sel*(*3*))
  then show *input-sizes* (*deep-model'-l rs*) = *replicate* (*2* ^ (*length rs* − *1*)) (*last rs*)
    by (*metis One-nat-def* ‹[] = *butlast rs*› *empty-replicate length-butlast list.size*(*3*) *power-0 replicate.simps*(*2*))
next
  case (*Cons r rs' rs*)
  then have *IH*: *input-sizes* (*deep-model'-l* (*tl rs*)) = *replicate* (*2* ^ (*length* (*tl rs*) − *1*)) (*last rs*)
    by (*metis* (*no-types*, *lifting*) *One-nat-def butlast-tl diff-is-0-eq' last-tl length-Cons length-butlast length-tl list.sel*(*3*) *list.size*(*3*) *nat-le-linear not-one-le-zero*)
  have *rs* = *r* # (*tl rs*) **by** (*metis Cons.hyps*(*2*) *Cons.prems One-nat-def append-Cons append-butlast-last-id length-greater-0-conv less-le-trans list.sel*(*3*) *zero-less-Suc*)
  then have *deep-model'-l rs* = *Pool* (*deep-model-l rs*) (*deep-model-l rs*)
    by (*metis Cons.hyps*(*2*) *One-nat-def butlast.simps*(*2*) *deep-model'.elims list.sel*(*3*) *list.simps*(*3*) *nth-Cons-0 nth-Cons-Suc*)

then have *input-sizes* (*deep-model'-l rs*) = *input-sizes* (*deep-model-l rs*) @ *input-sizes* (*deep-model-l rs*)
using *input-sizes.simps*(3) **by** *metis*
also have ... = *input-sizes* (*deep-model'-l* (*tl rs*)) @ *input-sizes* (*deep-model'-l* (*tl rs*))
by (*metis* (*no-types*, *lifting*) *Cons.hyps*(2) *One-nat-def deep-model.elims input-sizes.simps*(2) *length-Cons length-butlast length-greater-0-conv length-tl list.sel*(2) *list.sel*(3) *list.size*(3) *nth-tl one-neq-zero*)
also have ... = *replicate* (2 ^ (*length* (*tl rs*) − 1)) (*last rs*) @ *replicate* (2 ^ (*length* (*tl rs*) − 1)) (*last rs*)
using *IH* **by** *auto*
also have ... = *replicate* (2 ^ (*length rs* − 1)) (*last rs*)
using *replicate-add*[*of* 2 ^ (*length* (*tl rs*) − 1) 2 ^ (*length* (*tl rs*) − 1) *last rs*]
by (*metis Cons.hyps*(2) *One-nat-def butlast-tl length-butlast list.sel*(3) *list.size*(4) *mult-2-right power-add power-one-right*)
finally show *?case* **by** *auto*
qed

lemma *input-sizes-deep-model*:
assumes *length rs* ≥ 2
shows *input-sizes* (*deep-model-l rs*) = *replicate* (2^(*length rs* − 2)) (*last rs*)
proof −
have *input-sizes* (*deep-model-l rs*) = *input-sizes* (*deep-model'-l* (*tl rs*))
by (*metis One-nat-def Suc-1 assms hd-Cons-tl deep-model.elims input-sizes.simps*(2) *length-Cons length-greater-0-conv lessI linorder-not-le list.size*(3) *not-numeral-le-zero nth-tl*)
also have ... = *replicate* (2^(*length rs* − 2)) (*last rs*) **using** *input-sizes-deep-model'*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-1 Suc-eq-plus1 assms diff-diff-left hd-Cons-tl last-tl length-Cons length-tl linorder-not-le list.size*(3) *not-less-eq not-numeral-le-zero numeral-le-one-iff semiring-norm*(69))
finally show *?thesis* **by** *auto*
qed

lemma *evaluate-net-Conv-id*:
assumes *valid-net' m*
and *input-sizes m* = *map dim-vec input*
and *j*<*nr*
shows *evaluate-net* (*Conv* (*eye-matrix nr* (*output-size' m*)) *m*) *input* $ *j*
= (*if j*<*output-size' m then evaluate-net m input* $ *j else* 0)
unfolding *evaluate-net.simps output-size-correct*[*OF assms*(1) *assms*(2)[*symmetric*]]
using *mult-eye-matrix*[*OF* ‹*j*<*nr*›, *of evaluate-net m input*, *unfolded dim-vec-of-list*]
by *metis*

lemma *tensors-from-net-Conv-id*:
assumes *valid-net' m*
and *i*<*nr*

shows *tensors-from-net* (*Conv* (*eye-matrix nr* (*output-size′ m*)) *m*) $ *i*
= (*if i<output-size′ m then tensors-from-net m* $ *i else tensor0* (*input-sizes m*))
(**is** *?a* $ *i* = *?b*)
proof (*rule tensor-lookup-eqI*)
have *Tensor.dims* (*?a* $ *i*) = *input-sizes m* **by** (*metis assms*(*1*) *assms*(*2*) *dims-tensors-from-net*
eye-matrix-dim(*1*) *eye-matrix-dim*(*2*) *input-sizes.simps*(*2*) *output-size.simps*(*2*)
output-size-correct-tensors remove-weights.simps(*2*) *valid-net.intros*(*2*) *vec-setI*)
moreover have *Tensor.dims* (*?b*) = *input-sizes m* **using** *dims-tensors-from-net*
output-size-correct-tensors[*OF assms*(*1*)] *dims-tensor0* **by** (*simp add*: *vec-setI*)
ultimately show *Tensor.dims* (*?a* $ *i*) = *Tensor.dims* (*?b*) **by** *auto*

def *Convm* == *Conv* (*eye-matrix nr* (*output-size′ m*)) *m*
fix *is*
assume *is* ◁ *Tensor.dims* (*?a*$*i*)
then have *is* ◁ *input-sizes m* **using** ‹*Tensor.dims* (*?a*$*i*) = *input-sizes m*› **by** *auto*
have *valid-net′ Convm* **by** (*simp add*: *assms eye-matrix-dim valid-net.intros*(*2*) *Convm-def*)
have *base-input m is* = *base-input Convm is* **by** (*simp add*: *Convm-def base-input-def*)
have *i* < *output-size′ Convm* **unfolding** *Convm-def remove-weights.simps output-size.simps*
eye-matrix-dim **using** *assms* **by** *metis*
have *is* ◁ *input-sizes* (*Conv* (*eye-matrix nr* (*output-size′ m*)) *m*)
by (*metis* ‹*is* ◁ *input-sizes m*› *input-sizes.simps*(*2*))
then have *f1*: *lookup* (*tensors-from-net* (*Conv* (*eye-matrix nr* (*output-size′ m*)) *m*) $ *i*) *is* = *evaluate-net* (*Conv* (*eye-matrix nr* (*output-size′ m*)) *m*) (*base-input* (*Conv* (*eye-matrix nr* (*output-size′ m*)) *m*) *is*) $ *i*
using *Convm-def* ‹*i* < *output-size′ Convm*› ‹*valid-net′ Convm*› *lookup-tensors-from-net* **by** *blast*
have *lookup* (*tensor0* (*input-sizes m*)) *is* = (*0*::*real*)
by (*meson* ‹*is* ◁ *input-sizes m*› *lookup-tensor0*)
then show *Tensor.lookup* (*?a* $ *i*) *is* = *Tensor.lookup ?b is*
using *Convm-def* ‹*base-input m is* = *base-input Convm is*› ‹*is* ◁ *input-sizes m*› *assms*(*1*) *assms*(*2*)
base-input-length evaluate-net-Conv-id f1 lookup-tensors-from-net **by** *auto*
qed

lemma *evaluate-net-Conv-copy-first*:
assumes *valid-net′ m*
and *input-sizes m* = *map dim-vec input*
and *j*<*nr*
and *output-size′ m* > *0*
shows *evaluate-net* (*Conv* (*copy-first-matrix nr* (*output-size′ m*)) *m*) *input* $ *j*
= *evaluate-net m input* $ *0*
unfolding *evaluate-net.simps output-size-correct*[*OF assms*(*1*) *assms*(*2*)[*symmetric*]]
using *mult-copy-first-matrix*[*OF* ‹*j*<*nr*›, *of evaluate-net m input*, *unfolded dim-vec-of-list*]
assms(*3*) *copy-first-matrix-dim*(*1*) **by** (*metis* ‹*output-size′ m* = *dim-vec* (*evaluate-net m input*)› *assms*(*4*))

lemma *tensors-from-net-Conv-copy-first*:
assumes *valid-net' m*
and *i<nr*
and *output-size' m > 0*
shows *tensors-from-net (Conv (copy-first-matrix nr (output-size' m)) m) $ i =*
tensors-from-net m $ 0
*(***is** *?a $ i = ?b)*
proof *(rule tensor-lookup-eqI)*
have *Tensor.dims (?a$i) = input-sizes m*
by *(metis assms(1) assms(2) copy-first-matrix-dim(1) copy-first-matrix-dim(2)*
dims-tensors-from-net
input-sizes.simps(2) output-size.simps(2) output-size-correct-tensors remove-weights.simps(2)
valid-net.intros(2) vec-setI)
moreover have *Tensor.dims (?b) = input-sizes m* **using** *dims-tensors-from-net*
output-size-correct-tensors[OF assms(1)] **using** *assms(3)* **by** *(simp add: vec-setI)*
ultimately show *Tensor.dims (?a$i) = Tensor.dims (?b)* **by** *auto*

def *Convm == Conv (copy-first-matrix nr (output-size' m)) m*
fix *is*
assume *is ◁ Tensor.dims (?a$i)*
then have *is ◁ input-sizes m* **using** *‹Tensor.dims (?a$i) = input-sizes m›* **by**
auto
have *valid-net' Convm* **by** *(simp add: assms copy-first-matrix-dim valid-net.intros(2)*
Convm-def)
have *base-input m is = base-input Convm is* **by** *(simp add: Convm-def base-input-def)*
have *i < output-size' Convm* **unfolding** *Convm-def remove-weights.simps output-size.simps*
copy-first-matrix-dim **using** *assms* **by** *metis*
show *Tensor.lookup (?a $ i) is = Tensor.lookup ?b is*
by *(metis Convm-def ‹base-input m is = base-input Convm is› ‹i < output-size'*
Convm›
‹is ◁ input-sizes m› ‹valid-net' Convm› assms(1) assms(2) assms(3) base-input-length
evaluate-net-Conv-copy-first input-sizes.simps(2) lookup-tensors-from-net)
qed

lemma *evaluate-net-Conv-all1*:
assumes *valid-net' m*
and *input-sizes m = map dim-vec input*
and *i<nr*
shows *evaluate-net (Conv (all1-matrix nr (output-size' m)) m) input $ i*
= Groups-List.sum-list (list-of-vec (evaluate-net m input))
unfolding *evaluate-net.simps output-size-correct[OF assms(1) assms(2)[symmetric]]*
using *mult-all1-matrix[OF ‹i<nr›, of evaluate-net m input, unfolded dim-vec-of-list]*
assms(3) all1-matrix-dim(1) **by** *metis*

lemma *tensors-from-net-Conv-all1*:
assumes *valid-net' m*
and *i<nr*
shows *tensors-from-net (Conv (all1-matrix nr (output-size' m)) m) $ i*
= listsum (input-sizes m) (list-of-vec (tensors-from-net m))

```
  (is ?a $ i = ?b)
proof (rule tensor-lookup-eqI)
  have i < dim-vec ?a by (metis assms all1-matrix-dim output-size.simps(2)
    output-size-correct-tensors remove-weights.simps(2) valid-net.intros(2))
  then show Tensor.dims (?a $ i) = Tensor.dims (?b)
    using dims-tensors-from-net input-sizes.simps(2) listsum-dims
    by (metis index-vec-of-list in-set-conv-nth length-list-of-vec vec-list vec-setI)

  def Convm == Conv (all1-matrix nr (output-size' m)) m
  fix is assume is ◁ Tensor.dims (?a $ i)
  then have is ◁ input-sizes m
    using ‹i < dim-vec ?a› dims-tensors-from-net input-sizes.simps(2) by (metis
vec-setI)
  then have is ◁ input-sizes Convm by (simp add: Convm-def)
 have valid-net' Convm by (simp add: Convm-def assms all1-matrix-dim valid-net.intros(2))
  have i< output-size' Convm using Convm-def ‹i < dim-vec ?a› ‹valid-net'
Convm›
    output-size-correct-tensors by presburger
 have base-input Convm is = base-input m is unfolding base-input-def Convm-def
input-sizes.simps by metis
 have Tensor.lookup (?a $ i) is = evaluate-net Convm (base-input Convm is) $ i
  using lookup-tensors-from-net[OF ‹valid-net' Convm› ‹is ◁ input-sizes Convm›
‹i< output-size' Convm›]
    by (metis Convm-def )
 also have ... = monoid-add-class.sum-list (list-of-vec (evaluate-net m (base-input
Convm is)))
  using evaluate-net-Conv-all1 Convm-def ‹is ◁ input-sizes Convm› assms base-input-length
‹i < nr›
    by simp
 also have ... = monoid-add-class.sum-list (list-of-vec (map-vec (λA.  lookup A
is)(tensors-from-net m)))
    unfolding ‹base-input Convm is = base-input m is›
    using lookup-tensors-from-net[OF ‹valid-net' m› ‹is ◁ input-sizes m›]
      base-input-length[OF ‹is ◁ input-sizes m›] output-size-correct[OF assms(1)]
output-size-correct-tensors[OF assms(1)]
   eq-vecI[of evaluate-net m (base-input m is) map-vec (λA. lookup A is) (tensors-from-net
m)] index-map-vec(1) index-map-vec(2)
    by force
 also have ... = monoid-add-class.sum-list (map (λA.  lookup A is) (list-of-vec
(tensors-from-net m)))
  using eq-vecI[of vec-of-list (list-of-vec (map-vec (λA.  lookup A is)(tensors-from-net
m)))
   vec-of-list (map (λA.  lookup A is) (list-of-vec (tensors-from-net m)))] dim-vec-of-list
    nth-list-of-vec length-map list-vec nth-map  index-map-vec(1) index-map-vec(2)
vec-list
    by (metis (no-types, lifting))
 also have ... = Tensor.lookup ?b is  using dims-tensors-from-net set-list-of-vec
    using lookup-listsum[OF ‹is ◁ input-sizes m›, of list-of-vec (tensors-from-net
m)]
```

by *metis*
finally show *Tensor.lookup* (*?a* $ *i*) *is* = *Tensor.lookup ?b is* **by** *blast*
qed

fun *witness* **and** *witness'* **where**
witness' Y [] = *Input Y* |
witness' Y (*r* # *rs*) = *Pool* (*witness Y r rs*) (*witness Y r rs*) |
witness Y r rs = *Conv* ((*if length rs* = *0 then eye-matrix else* (*if length rs* = *1 then all1-matrix else copy-first-matrix*)) *Y r*) (*witness' r rs*)

abbreviation *witness-l rs* == *witness* (*rs*!*0*) (*rs*!*1*) (*tl* (*tl rs*))
abbreviation *witness'-l rs* == *witness'* (*rs*!*0*) (*tl rs*)

lemma *witness-is-deep-model*: *remove-weights* (*witness Y r rs*) = *deep-model Y r rs*
proof (*induction rs arbitrary*: *Y r*)
case *Nil*
then show *?case* **unfolding** *witness.simps witness'.simps deep-model.simps deep-model'.simps*
by (*simp add*: *eye-matrix-dim*)
next
case (*Cons r' rs Y r*)
have *dim-row* ((*if length* (*r'* # *rs*) = *0 then eye-matrix else* (*if length* (*r'* # *rs*) = *1 then all1-matrix else copy-first-matrix*)) *Y r*) = *Y*
dim-col ((*if length* (*r'* # *rs*) = *0 then eye-matrix else* (*if length* (*r'* # *rs*) = *1 then all1-matrix else copy-first-matrix*)) *Y r*) = *r*
by (*simp-all add*: *all1-matrix-dim copy-first-matrix-dim*)
then show *?case* **unfolding** *witness.simps* **unfolding** *witness'.simps* **unfolding** *remove-weights.simps*
using *Cons* **by** *simp*
qed

lemma *witness'-is-deep-model*: *remove-weights* (*witness' Y rs*) = *deep-model' Y rs*
proof (*induction rs arbitrary*: *Y*)
case *Nil*
then show *?case* **unfolding** *witness.simps witness'.simps deep-model.simps deep-model'.simps*
by (*simp add*: *eye-matrix-dim*)
next
case (*Cons r rs Y*)
have *dim-row* ((*if length rs* = *0 then eye-matrix else* (*if length rs* = *1 then all1-matrix else copy-first-matrix*)) *Y r*) = *Y*
dim-col ((*if length rs* = *0 then eye-matrix else* (*if length rs* = *1 then all1-matrix else copy-first-matrix*)) *Y r*) = *r*
by (*simp-all add*: *all1-matrix-dim copy-first-matrix-dim eye-matrix-dim*)
then show *?case* **unfolding** *witness'.simps* **unfolding** *witness.simps* **unfolding** *remove-weights.simps*
using *Cons* **by** *simp*
qed

lemma *witness-valid*: *valid-net'* (*witness Y r rs*)
using *valid-deep-model witness-is-deep-model* **by** *auto*

lemma *witness'-valid*: *valid-net'* (*witness' Y rs*)
using *valid-deep-model' witness'-is-deep-model* **by** *auto*

lemma *witness-l0'*: *witness' Y* [*M*] =
(*Pool*
(*Conv* (*eye-matrix Y M*) (*Input M*))
(*Conv* (*eye-matrix Y M*) (*Input M*))
)
unfolding *witness'.simps witness.simps* **by** *simp*

lemma *witness-l1*: *witness Y r0* [*M*] =
Conv (*all1-matrix Y r0*) (*witness' r0* [*M*])
unfolding *witness'.simps* **by** *simp*

lemma *tensors-ht-l0*:
assumes *j*<*r0*
shows *tensors-from-net* (*Conv* (*eye-matrix r0 M*) (*Input M*)) $ *j*
= (*if j*<*M then unit-vec M j else tensor0* [*M*])
by (*metis assms input-sizes.simps*(*1*) *output-size.simps*(*1*) *remove-weights.simps*(*1*)
tensors-from-net.simps(*1*)
tensors-from-net-Conv-id valid-net.intros(*1*) *index-vec*)

lemma *tensor-prod-unit-vec*:
unit-vec M j ⊗ *unit-vec M j* = *tensor-from-lookup* [*M*,*M*] (λ*is*. *if is*=[*j*,*j*] *then 1*
else 0) (**is** *?A*=*?B*)
proof (*rule tensor-lookup-eqI*)
show *Tensor.dims ?A* = *Tensor.dims ?B*
by (*metis append-Cons self-append-conv2 dims-unit-vec dims-tensor-prod dims-tensor-from-lookup*)
fix *is* **assume** *is-valid*:*is* ◁ *Tensor.dims* (*unit-vec M j* ⊗ *unit-vec M j*)
then have *is* ◁ [*M*,*M*] **by** (*metis append-Cons self-append-conv2 dims-unit-vec*
dims-tensor-prod)
then obtain *i1 i2* **where** *is-split*: *is* = [*i1*, *i2*] *i1* < *M i2* < *M* **using** *list.distinct*(*1*)
by *blast*
then have [*i1*] ◁ *Tensor.dims* (*unit-vec M j*) [*i2*] ◁ *Tensor.dims* (*unit-vec M*
j)
by (*simp-all add*: *valid-index.Cons valid-index.Nil dims-unit-vec*)
have *is* = [*i1*] @ [*i2*] **by** (*simp add*: *is-split*(*1*))
show *Tensor.lookup ?A is* = *Tensor.lookup ?B is*
unfolding ‹*is* = [*i1*] @ [*i2*]›
lookup-tensor-prod[*OF* ‹[*i1*] ◁ *Tensor.dims* (*unit-vec M j*)› ‹[*i2*] ◁ *Tensor.dims*
(*unit-vec M j*)›]
lookup-tensor-from-lookup[*OF* ‹*is* ◁ [*M*, *M*]›, *unfolded* ‹*is* = [*i1*] @ [*i2*]›]
lookup-unit-vec[*OF* ‹*i1* < *M*›] *lookup-unit-vec*[*OF* ‹*i2* < *M*›] **by** *fastforce*
qed

lemma *tensors-ht-l0'*:

assumes *j<r0*
shows *tensors-from-net (witness' r0 [M]) $ j*
*= (if j<M then unit-vec M j ⊗ unit-vec M j else tensor0 [M,M]) (***is** *- = ?b)*
proof *−*
have *valid-net' (Conv (eye-matrix r0 M) (Input M))*
by *(metis convnet.inject(3) list.discI witness'.elims witness-l0' witness-valid)*
have *j-le:j < dim-vec (tensors-from-net (Conv (eye-matrix r0 M) (Input M)))*
using *output-size-correct-tensors[OF ‹valid-net' (Conv (eye-matrix r0 M) (Input M))›,*
unfolded remove-weights.simps output-size.simps eye-matrix-dim]
assms **by** *simp*
show *?thesis*
unfolding *tensors-from-net.simps(3) witness-l0' index-component-mult[OF j-le j-le] tensors-ht-l0[OF assms]*
by *auto*
qed

lemma *lookup-tensors-ht-l0'*:
assumes *j<r0*
and *is ◁ [M,M]*
shows *(Tensor.lookup (tensors-from-net (witness' r0 [M]) $ j)) is = (if is=[j,j] then 1 else 0)*

proof *(cases j<M)*
assume *j<M*
show *?thesis* **unfolding** *tensors-ht-l0'[OF assms(1)] tensor-prod-unit-vec*
apply *(cases is = [j,j])* **using** *‹j<M› assms(2)*
by *(simp-all add:lookup-tensor-from-lookup)*
next
assume *¬j<M*
then have *is ≠ [j, j]* **using** *assms(2)* **using** *list.distinct(1) nth-Cons-0 valid-index.simps* **by** *blast*
show *?thesis* **unfolding** *tensors-ht-l0'[OF assms(1)] tensor-prod-unit-vec*
using *‹¬j<M›* **by** *(simp add: lookup-tensor0[OF assms(2)] ‹is ≠ [j, j]›)*
qed

lemma *lookup-tensors-ht-l1*:
assumes *j < r1*
and *is ◁ [M,M]*
shows *Tensor.lookup (tensors-from-net (witness r1 r0 [M]) $ j) is*
= (if is!0 = is!1 ∧ is!0<r0 then 1 else 0)
proof *−*
have *witness-l0'-valid: valid-net' (witness' r0 [M])* **unfolding** *witness-l0'*
by *(simp add: eye-matrix-dim valid-net.intros)*
have *input-sizes (witness' r0 [M]) = [M,M]* **unfolding** *witness-l0'* **by** *simp*
have *output-size' (witness' r0 [M]) = r0* **unfolding** *witness-l0'* **using** *witness-l0'-valid*
by *(simp add: eye-matrix-dim)*
have *dim-vec (tensors-from-net (witness' r0 [M])) = r0*
using *‹output-size' (witness' r0 [M]) = r0› witness-l0'-valid output-size-correct-tensors*

by *fastforce*
have *all0-but1*:$\bigwedge i.\ i{\neq}is!0 \Longrightarrow i{<}r0 \Longrightarrow$ *Tensor.lookup* (*tensors-from-net* (*witness'* *r0* [*M*]) $ *i*) *is* = *0*
using *lookup-tensors-ht-l0'* ‹*is* ◁ [*M*, *M*]› **by** *auto*

have *tensors-from-net* (*witness r1 r0* [*M*]) $ *j* =
Tensor-Plus.listsum [*M*,*M*] (*list-of-vec* (*tensors-from-net* (*witness'* *r0* [*M*])))
unfolding *witness-l1* **using** *tensors-from-net-Conv-all1*[*OF* *witness-l0'-valid* *assms*(*1*)]
witness-l0' ‹*output-size'* (*witness'* *r0* [*M*]) = *r0*› **by** *simp*
then have *Tensor.lookup* (*tensors-from-net* (*witness r1 r0* [*M*]) $ *j*) *is*
= *monoid-add-class.sum-list* (*map* (λ*A*. *Tensor.lookup A is*) (*list-of-vec* (*tensors-from-net* (*witness'* *r0* [*M*]))))
using *lookup-listsum*[*OF* ‹*is* ◁ [*M*, *M*]› ‹*input-sizes* (*witness'* *r0* [*M*]) = [*M*, *M*]›
dims-tensors-from-net **by** (*metis set-list-of-vec*)
also have ... = *monoid-add-class.sum-list* (*map* (λ*i*. *lookup* (*tensors-from-net* (*witness'* *r0* [*M*]) $ *i*) *is*) [*0*..<*r0*])
using *map-map*[*of* (λ*A*. *Tensor.lookup A is*) λ*i*. (*tensors-from-net* (*witness'* *r0* [*M*]) $ *i*) [*0*..<*r0*]]
using *list-of-vec-map* ‹*dim-vec* (*tensors-from-net* (*witness'* *r0* [*M*])) = *r0*› **by** (*metis* (*mono-tags*, *lifting*) *comp-apply map-eq-conv*)
also have ... = (∑ *i*<*r0*. *Tensor.lookup* ((*tensors-from-net* (*witness'* *r0* [*M*])) $ *i*) *is*)
using *sum-set-upt-conv-sum-list-nat atLeast0LessThan* **by** (*metis atLeast-upt*)
also have ... = (*if is*!*0* = *is*!*1* ∧ *is*!*0*<*r0 then 1 else 0*)
proof (*cases is*!*0*<*r0*)
case *True*
have *finite* {*0*..<*r0*} **by** *auto*
have *is*!*0* ∈ {*0*..<*r0*} **using** *True* **by** *auto*
have (∑ *i*<*r0*. *Tensor.lookup* ((*tensors-from-net* (*witness'* *r0* [*M*])) $ *i*) *is*)
= *Tensor.lookup* (*tensors-from-net* (*witness'* *r0* [*M*]) $ (*is*!*0*)) *is*
using ‹*dim-vec* (*tensors-from-net* (*witness'* *r0* [*M*])) = *r0*›
using *sum.remove*[*OF* ‹*finite* {*0*..<*r0*}› ‹*is*!*0* ∈ {*0*..<*r0*}›,
of λ*i*. (*Tensor.lookup* (*tensors-from-net* (*witness'* *r0* [*M*])$*i*) *is*)]
using *all0-but1 atLeast0LessThan* **by** *force*
then show *?thesis* **using** *lookup-tensors-ht-l0'* ‹*is* ! *0* < *r0*› ‹*is* ◁ [*M*, *M*]› **by** *fastforce*
next
case *False*
then show *?thesis* **using** *all0-but1 atLeast0LessThan sum.neutral* **by** *force*
qed
finally show *?thesis* **by** *auto*
qed

lemma *length-output-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*

shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
using *output-size-correct-tensors valid-deep-model*
deep-model.elims output-size.simps(*2*) **by** (*metis assms*)

lemma *length-output-deep-model′*:
assumes *remove-weights m* = *deep-model′-l rs*
shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
using *output-size-correct-tensors valid-deep-model′*
deep-model′.elims output-size.simps **by** (*metis assms deep-model.elims*)

lemma *length-output-witness*:
dim-vec (*tensors-from-net* (*witness-l rs*)) = *rs* ! *0*
using *length-output-deep-model witness-is-deep-model* **by** *blast*

lemma *length-output-witness′*:
dim-vec (*tensors-from-net* (*witness′-l rs*)) = *rs* ! *0*
using *length-output-deep-model′ witness′-is-deep-model* **by** *blast*

lemma *dims-output-deep-model*:
assumes *length rs* ≥ *2*
and ⋀*r*. *r*∈*set rs* ⟹ *r* > *0*
and *j* < *rs*!*0*
and *remove-weights m* = *deep-model-l rs*
shows *Tensor.dims* (*tensors-from-net m* $ *j*) = *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
using *dims-tensors-from-net input-sizes-deep-model*[*OF assms*(*1*)] *output-size-correct-tensors valid-deep-model*
assms(*3*) *assms*(*4*) *input-sizes-remove-weights length-output-witness witness-is-deep-model*
by (*metis vec-setI*)

lemma *dims-output-witness*:
assumes *length rs* ≥ *2*
and ⋀*r*. *r*∈*set rs* ⟹ *r* > *0*
and *j* < *rs*!*0*
shows *Tensor.dims* (*tensors-from-net* (*witness-l rs*) $ *j*) = *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
using *dims-output-deep-model witness-is-deep-model assms* **by** *blast*

lemma *dims-output-deep-model′*:
assumes *length rs* ≥ *1*
and ⋀*r*. *r*∈*set rs* ⟹ *r* > *0*
and *j* < *rs*!*0*
and *remove-weights m* = *deep-model′-l rs*
shows *Tensor.dims* (*tensors-from-net m* $ *j*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
proof −
have *dim-vec* (*tensors-from-net m*) > *j*
using *length-output-deep-model′* ‹*remove-weights m* = *deep-model′-l rs*› ‹*j* < *rs*!*0*› **by** *auto*

then have *Tensor.dims* (*tensors-from-net m $ j*) = *input-sizes m*
using *dims-tensors-from-net*[*of - m*] *output-size-correct-tensors*
vec-setI **by** *metis*
then show *?thesis*
using *assms*(*1*) *input-sizes-deep-model′*
input-sizes-remove-weights[*of m*, *unfolded* ‹*remove-weights m = deep-model′-l rs*›] **by** *auto*
qed

lemma *dims-output-witness′*:
assumes *length rs ≥ 1*
and ⋀*r*. *r*∈*set rs* ⟹ *r > 0*
and *j < rs*!*0*
shows *Tensor.dims* (*tensors-from-net* (*witness′-l rs*) $ *j*) = *replicate* (*2*^(*length rs − 1*)) (*last rs*)
using *dims-output-deep-model′ assms witness′-is-deep-model* **by** *blast*

abbreviation *ten2mat* == *matricize* {*n*. *even n*}
abbreviation *mat2ten* == *dematricize* {*n*. *even n*}

locale *deep-model-correct-params* =
fixes *rs*::*nat list*
assumes *deep*:*length rs ≥ 3*
and *no-zeros*:⋀*r*. *r*∈*set rs* ⟹ *0 < r*
begin

definition *r* = *min* (*last rs*) (*last* (*butlast rs*))
definition *N-half* = *2*^(*length rs − 3*)
definition *weight-space-dim* = *count-weights*(*deep-model-l rs*)

end

locale *deep-model-correct-params-y* = *deep-model-correct-params* +
fixes *y*::*nat*
assumes *y-valid*:*y < rs* ! *0*
begin

definition *A ws* = *tensors-from-net* (*insert-weights* (*deep-model-l rs*) *ws*) $ *y*
definition *A′ ws* = *ten2mat* (*A ws*)

lemma *dims-tensor-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *dims* (*tensors-from-net m $ y*) = *replicate* (*2 ∗ N-half*) (*last rs*)
proof −
have *dims* (*tensors-from-net m $ y*) = *replicate* (*2* ^ (*length rs − 2*)) (*last rs*)
using *dims-output-deep-model*[*OF - no-zeros y-valid assms*] **using** *less-imp-le-nat Suc-le-lessD deep numeral-3-eq-3*

by *auto*
then show *?thesis* **using** *N-half-def* **by** (*metis One-nat-def Suc-1 Suc-eq-plus1 Suc-le-lessD deep diff-diff-left less-numeral-extra*(*3*) *numeral-3-eq-3 realpow-num-eq-if zero-less-diff*)
qed

lemma *order-tensor-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *order* (*tensors-from-net m* $ *y*) = *2* ∗ *N-half*
using *dims-tensor-deep-model* **by** (*simp add*: *assms*)

lemma *dims-A*:
shows *Tensor.dims* (*A ws*) = *replicate* (*2* ∗ *N-half*) (*last rs*)
unfolding *A-def*
using *dims-tensor-deep-model remove-insert-weights* **by** *blast*

lemma *order-A*:
shows *order* (*A ws*) = *2* ∗ *N-half* **using** *dims-A length-replicate* **by** *auto*

lemma *dims-A′*:
shows *dim-row* (*A′ ws*) = *prod-list* (*nths* (*Tensor.dims* (*A ws*)) {*n*. *even n*})
and *dim-col* (*A′ ws*) = *prod-list* (*nths* (*Tensor.dims* (*A ws*)) {*n*. *odd n*})
unfolding *A′-def matricize-def* **by** (*simp-all add*: *A-def Collect-neg-eq*)

lemma *dims-A′-pow*:
shows *dim-row* (*A′ ws*) = (*last rs*) ^ *N-half dim-col* (*A′ ws*) = (*last rs*) ^ *N-half*
unfolding *dims-A′ dims-A nths-replicate set-le-in card-even card-odd prod-list-replicate*
by *simp-all*

definition *Aw* = *tensors-from-net* (*witness-l rs*) $ *y*
definition *Aw′* = *ten2mat Aw*

definition *witness-weights* = (*SOME ws*. *witness-l rs* = *insert-weights* (*deep-model-l rs*) *ws*)

lemma *witness-weights*:*witness-l rs* = *insert-weights* (*deep-model-l rs*) *witness-weights*
proof −
have *0*:∃ *x*. *witness-l rs* = *insert-weights* (*deep-model-l rs*) *x*
unfolding *weight-space-dim-def* **using** *insert-remove-weights witness-is-deep-model*
by *metis*
show *witness-l rs* = *insert-weights* (*deep-model-l rs*) *witness-weights*
unfolding *witness-weights-def* **using** *someI-ex*[*OF 0*] **by** *blast*
qed

lemma *Aw-def′*: *Aw* = *A witness-weights* **unfolding** *Aw-def A-def* **using** *witness-weights*
by *auto*

lemma *Aw'-def'*: *Aw' = A' witness-weights* **unfolding** *Aw'-def A'-def Aw-def'* **by** *auto*

lemma *dims-Aw*: *Tensor.dims Aw = replicate (2 * N-half) (last rs)*
unfolding *Aw-def'* **using** *dims-A* **by** *auto*

lemma *order-Aw*: *order Aw = 2 * N-half*
unfolding *Aw-def'* **using** *order-A* **by** *auto*

lemma *dims-Aw'*:
dim-row Aw' = prod-list (nths (Tensor.dims Aw) {n. even n})
dim-col Aw' = prod-list (nths (Tensor.dims Aw) {n. odd n})
unfolding *Aw'-def' Aw-def'* **using** *dims-A'* **by** *auto*

lemma *dims-Aw'-pow*: *dim-row Aw' = (last rs) ^ N-half dim-col Aw' = (last rs) ^ N-half*
unfolding *Aw'-def' Aw-def'* **using** *dims-A'-pow* **by** *auto*

lemma *witness-tensor*:
assumes *is ◁ Tensor.dims Aw*
shows *Tensor.lookup Aw is*
= (if nths is {n. even n} = nths is {n. odd n} ∧ (∀ i∈set is. i < last (butlast rs)) then 1 else 0)
using *assms deep no-zeros y-valid* **unfolding** *Aw-def* **proof** (*induction butlast* (*butlast* (*butlast rs*)) *arbitrary*:*rs is y*)
case *Nil*
have *length rs = 3*
by (*rule antisym*, *metis Nil.hyps One-nat-def Suc-1 Suc-eq-plus1 add-2-eq-Suc' diff-diff-left*
length-butlast less-numeral-extra(*3*) *list.size*(*3*) *not-le numeral-3-eq-3 zero-less-diff*, *metis ‹3 ≤ length rs›*)
then have *rs = [rs!0, rs!1, rs!2]* **by** (*metis* (*no-types*, *lifting*) *Cons-nth-drop-Suc One-nat-def Suc-eq-plus1*
append-Nil id-take-nth-drop length-0-conv length-tl lessI list.sel(*3*) *list.size*(*4*) *not-le numeral-3-eq-3*
numeral-le-one-iff one-add-one semiring-norm(*70*) *take-0 zero-less-Suc*)
have *input-sizes* (*witness-l* [*rs ! 0*, *rs ! 1*, *rs ! 2*]) = [*rs!2*, *rs!2*]
using *witness.simps witness'.simps input-sizes.simps* **by** *auto*
then have *Tensor.dims* (*tensors-from-net* (*witness-l rs*) $ *y*) = [*rs!2*, *rs!2*]
using *dims-tensors-from-net*[*of tensors-from-net* (*witness-l rs*) $ *y witness-l rs*]
Nil.prems(*4*) *length-output-witness ‹rs = [rs ! 0, rs ! 1, rs ! 2]› vec-setI* **by** *metis*
then have *is ◁* [*rs!2*, *rs!2*] **using** *Nil.prems* **by** *metis*
then have *Tensor.lookup* ((*tensors-from-net* (*witness-l rs*))$*y*) *is*
= (if is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1 then 1 else 0)
using *Nil.prems*(*4*) *‹rs = [rs ! 0, rs ! 1, rs ! 2]›* **by** (*metis list.sel*(*3*) *lookup-tensors-ht-l1*)
have *is ! 0 = is ! 1 ∧ is ! 0 < rs ! 1*
⟷ nths is {n. even n} = nths is {n. odd n} ∧ (∀ i∈set is. i < last (butlast

rs))
proof −
have *length is = 2* **by** (*metis One-nat-def Suc-eq-plus1* ‹*is* ◁ [*rs* ! *2*, *rs* ! *2*]› *list.size*(*3*) *list.size*(*4*) *numeral-2-eq-2 valid-index-length*)
have *nths is* {*n*. *even n*} = [*is*!*0*]
apply (*rule nths-only-one*)
using *subset-antisym less-2-cases* ‹*length is = 2*› **by** *fastforce*
have *nths is* {*n*. *odd n*} = [*is*!*1*]
apply (*rule nths-only-one*)
using *subset-antisym less-2-cases* ‹*length is = 2*› **by** *fastforce*
have *last* (*butlast rs*) = *rs*!*1* **by** (*metis One-nat-def Suc-eq-plus1* ‹*rs* = [*rs* ! *0*, *rs* ! *1*, *rs* ! *2*]›
append-butlast-last-id last-conv-nth length-butlast length-tl lessI list.sel(*3*) *list.simps*(*3*)
list.size(*3*) *list.size*(*4*) *nat.simps*(*3*) *nth-append*)
show *?thesis* **unfolding** ‹*last* (*butlast rs*) = *rs*!*1*›
apply (*rule iffI*; *rule conjI*)
apply (*simp add*: ‹*nths is* (*Collect even*) = [*is* ! *0*]› ‹*nths is* {*n*. *odd n*} = [*is* ! *1*]›)
apply (*metis* ‹*length is = 2*› *One-nat-def in-set-conv-nth less-2-cases*)
apply (*simp add*: ‹*nths is* (*Collect even*) = [*is* ! *0*]› ‹*nths is* {*n*. *odd n*} = [*is* ! *1*]›)
apply (*simp add*: ‹*length is = 2*›)
done
qed
then show *?case* **unfolding** ‹*Tensor.lookup* (*tensors-from-net* (*witness-l rs*) $ *y*) *is* = (*if is* ! *0* = *is* ! *1* ∧ *is* ! *0* < *rs* ! *1 then 1 else 0*)›
using *witness-is-deep-model witness-valid* ‹*rs* = [*rs* ! *0*, *rs* ! *1*, *rs* ! *2*]› **by** *auto*
next
case (*Cons r rs' rs is j*)

We prove the Induction Hypothesis for "tl rs" and j=0:

have *rs* = *r* # *tl rs* **by** (*metis Cons.hyps*(*2*) *append-butlast-last-id butlast.simps*(*1*) *hd-append2 list.collapse list.discI list.sel*(*1*))
have *1*:*rs'* = *butlast* (*butlast* (*butlast* (*tl rs*))) **by** (*metis Cons.hyps*(*2*) *butlast-tl list.sel*(*3*))
have *2*:*3* ≤ *length* (*tl rs*) **by** (*metis* (*no-types*, *lifting*) *Cons.hyps*(*2*) *Cons.prems*(*2*)
Nitpick.size-list-simp(*2*) *One-nat-def Suc-eq-plus1* ‹*rs* = *r* # *tl rs*› ‹*rs'* = *butlast* (*butlast* (*butlast* (*tl rs*)))›
diff-diff-left diff-self-eq-0 gr0-conv-Suc le-Suc-eq length-butlast length-tl less-numeral-extra(*3*) *list.simps*(*3*) *numeral-3-eq-3*)
have *3*:⋀*r*. *r* ∈ *set* (*tl rs*) ⟹ *0* < *r* **by** (*metis Cons.prems*(*3*) *list.sel*(*2*) *list.set-sel*(*2*))
have *4*:*0* < (*tl rs*) ! *0* **using** *2 3* **by** *auto*
have *IH*: ⋀*is'*. *is'* ◁ *Tensor.dims* (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*)
⟹ *Tensor.lookup* (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*) *is'* =
(*if nths is'* (*Collect even*) = *nths is'* {*n*. *odd n*} ∧ (∀ *i*∈*set is'*. *i* < *last* (*butlast* (*tl rs*))) *then 1 else 0*)
using *1 2 3 4 Cons.hyps*(*1*) **by** *blast*

The list "is" can be split in two parts:

have *is* ◁ *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
using *Cons.prems*(*3*) *dims-output-witness 2* **by** (*metis* (*no-types*, *lifting*) *Cons.prems*(*1*) *Cons.prems*(*3*)
Cons.prems(*4*) *Nitpick.size-list-simp*(*2*) *One-nat-def diff-diff-left diff-is-0-eq length-tl*
nat-le-linear not-numeral-le-zero numeral-le-one-iff one-add-one semiring-norm(*70*))
then have *is* ◁ *replicate* (*2*^(*length* (*tl rs*) − *2*)) (*last rs*) @ *replicate* (*2*^(*length* (*tl rs*) − *2*)) (*last rs*)
using *Cons.prems dims-output-witness* **by** (*metis 2 Nitpick.size-list-simp*(*2*) *One-nat-def*
diff-diff-left length-tl mult-2 not-numeral-le-zero numeral-le-one-iff one-add-one
power.simps(*2*) *replicate-add semiring-norm*(*70*))
then obtain *is1 is2* **where** *is* = *is1* @ *is2* **and**
is1-replicate: *is1* ◁ *replicate* (*2*^(*length* (*tl rs*) − *2*)) (*last rs*) **and**
is2-replicate: *is2* ◁ *replicate* (*2*^(*length* (*tl rs*) − *2*)) (*last rs*) **by** (*metis valid-index-split*)
then have
is1-valid: *is1* ◁ *Tensor.dims* (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*) (**is** *?is1*) **and**
is2-valid: *is2* ◁ *Tensor.dims* (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*) (**is** *?is2*)
proof −
have *last* (*tl rs*) = *last rs* **by** (*metis 2* ‹*rs* = *r* # *tl rs*› *last-ConsR list.size*(*3*) *not-numeral-le-zero*)
then show *?is1 ?is2* **using** *dims-output-witness*[*of tl rs*]
using *dims-output-witness*[*of tl rs*] *2 3 is1-replicate is2-replicate* ‹*last* (*tl rs*) = *last rs*› **by** *auto*
qed

A shorthand for the condition to find a "1" in the tensor:

let *?cond* = λ*is rs*. *nths is* {*n*. *even n*} = *nths is* {*n*. *odd n*} ∧ (∀ *i*∈*set is*. *i* < *last* (*butlast rs*))

We can use the IH on our newly created is1 and is2:

have *IH-is12*:
Tensor.lookup (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*) *is1* =
(*if* (*?cond is1* (*tl rs*)) *then 1 else 0*)
Tensor.lookup (*tensors-from-net* (*witness-l* (*tl rs*)) $ *0*) *is2* =
(*if* (*?cond is2* (*tl rs*)) *then 1 else 0*)
using *IH is1-valid is2-valid* **by** *fast*+

In the induction step we have to add two layers: first the Pool layer, then the Conv layer.

The Pool layer connects the two subtrees. Therefore the two conditions on is1 and is2 become one, and we have to prove that they are equivalent:

have *?cond is1* (*tl rs*) ∧ *?cond is2* (*tl rs*) ⟷ *?cond is rs*
proof −
have *length is1* = *2* ^ (*length* (*tl rs*) − *2*)

```
      length is2 = 2 ^ (length (tl rs) − 2)
    using is1-replicate is2-replicate by (simp-all add: valid-index-length)
  then have even (length is1) even (length is2)
      by (metis Cons.hyps(2) One-nat-def add-gr-0 diff-diff-left even-numeral
even-power
    length-butlast length-tl list.size(4) one-add-one zero-less-Suc)+
  then have {j. j + length is1 ∈ {n. even n}} = {n. even n}
          {j. j + length is1 ∈ {n. odd n}} = {n. odd n} by simp-all
  have length (nths is2 (Collect even)) = length (nths is2 (Collect odd))
    using length-nths-even ‹even (length is2)› by blast
   have cond1-iff: (nths is1 (Collect even) = nths is1 {n. odd n} ∧ nths is2
(Collect even) = nths is2 {n. odd n})
       = (nths is (Collect even) = nths is {n. odd n})
      unfolding ‹is = is1 @ is2› nths-append
      ‹{j. j + length is1 ∈ {n. odd n}} = {n. odd n}› ‹{j. j + length is1 ∈ {n.
even n}} = {n. even n}›
      by (simp add: ‹length (nths is2 (Collect even)) = length (nths is2 (Collect
odd))›)
   have last (butlast (tl rs)) = last (butlast rs) using Nitpick.size-list-simp(2)
‹even (length is1)›
    ‹length is1 = 2 ^ (length (tl rs) − 2)› butlast-tl last-tl length-butlast length-tl
not-less-eq zero-less-diff
    by (metis (full-types) Cons.hyps(2) length-Cons less-nat-zero-code)
  have cond2-iff: (∀ i∈set is1. i < last (butlast (tl rs))) ∧ (∀ i∈set is2. i < last
(butlast (tl rs))) ⟷ (∀ i∈set is. i < last (butlast rs))
   unfolding ‹last (butlast (tl rs)) = last (butlast rs)› ‹is = is1 @ is2› set-append
by blast
  then show ?thesis using cond1-iff cond2-iff by blast
 qed
```

Now we can make the Pool layer step:

```
 have lookup-witness': Tensor.lookup ((tensors-from-net (witness' (rs ! 1) (tl (tl
rs)))) $ 0) is =
  (if ?cond is rs then 1 else 0)
 proof −
  have lookup-prod: Tensor.lookup ((tensors-from-net (witness-l (tl rs)) $ 0) ⊗
(tensors-from-net (witness-l (tl rs))) $ 0) is =
    (if ?cond is rs then 1 else 0)
    using ‹?cond is1 (tl rs) ∧ ?cond is2 (tl rs) ⟷ ?cond is rs›
   unfolding ‹is = is1 @ is2› lookup-tensor-prod[OF is1-valid is2-valid] IH-is12
    by auto
  have witness-l-tl: witness-l (tl rs) = witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))
    by (metis One-nat-def Suc-1 ‹rs = r # tl rs› nth-Cons-Suc)
  have tl-tl:(tl (tl rs)) = ((rs ! 2) # tl (tl (tl rs)))
  proof −
   have length (tl (tl rs)) ≠ 0
   by (metis One-nat-def Suc-eq-plus1 diff-diff-left diff-is-0-eq length-tl not-less-eq-eq
     Cons.prems(2) numeral-3-eq-3)
    then have tl (tl rs) ≠ []
```

```
      by fastforce
    then show ?thesis
      by (metis list.exhaust-sel nth-Cons-0 nth-Cons-Suc numeral-2-eq-2 tl-Nil)
  qed
  have length-gt0:dim-vec (tensors-from-net (witness (rs ! 1) (rs ! 2) (tl (tl (tl
rs))))) > 0
    using output-size-correct-tensors[of witness (rs ! 1) (rs ! 2) (tl (tl (tl rs)))]
    witness-is-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))]
    valid-deep-model[of rs ! 1 rs ! 2 tl (tl (tl rs))] output-size.simps witness.simps
  by (metis 2 3 One-nat-def ‹rs = r # tl rs› deep-model.elims length-greater-0-conv
list.size(3)
    not-numeral-le-zero nth-Cons-Suc nth-mem)
  then have tensors-from-net (witness' (rs ! 1) ((rs ! 2) # tl (tl (tl rs)))) $ 0
      = (tensors-from-net (witness-l (tl rs)) $ 0) ⊗ (tensors-from-net (witness-l
(tl rs)) $ 0)
  unfolding witness'.simps tensors-from-net.simps witness-l-tl using index-component-mult
by blast
  then show ?thesis using lookup-prod tl-tl by simp
 qed
```

Then we can make the Conv layer step:

```
 show ?case
 proof −
  have valid-net' (witness' (rs ! 1) (tl (tl rs))) by (simp add: witness'-valid)
  have output-size' (witness' (rs ! 1) (tl (tl rs))) = rs ! 1
  by (metis 2 Nitpick.size-list-simp(2) diff-diff-left diff-is-0-eq hd-Cons-tl deep-model'.simps(2)
deep-model.elims length-tl not-less-eq-eq numeral-2-eq-2 numeral-3-eq-3 one-add-one
output-size.simps(2) output-size.simps(3) tl-Nil witness'-is-deep-model)
  have if-resolve:(if length (tl (tl rs)) = 0 then eye-matrix else if length (tl (tl
rs)) = 1 then all1-matrix else copy-first-matrix) = copy-first-matrix
  by (metis 2 Cons.prems(2) Nitpick.size-list-simp(2) One-nat-def Suc-n-not-le-n
not-numeral-le-zero numeral-3-eq-3)
  have tensors-from-net (Conv (copy-first-matrix (rs ! 0) (rs ! 1)) (witness' (rs
! 1) (tl (tl rs)))) $ j =
    tensors-from-net (witness' (rs ! 1) (tl (tl rs))) $ 0
    using tensors-from-net-Conv-copy-first[OF ‹valid-net' (witness' (rs ! 1) (tl
(tl rs)))› ‹j < rs ! 0›, unfolded ‹output-size' (witness' (rs ! 1) (tl (tl rs))) = rs !
1›]
    using 4 One-nat-def ‹rs = r # tl rs› nth-Cons-Suc by metis
  then show ?thesis unfolding witness.simps if-resolve ‹output-size' (witness'
(rs ! 1) (tl (tl rs))) = rs ! 1›
    using lookup-witness' ‹valid-net' (witness' (rs ! 1) (tl (tl rs)))› hd-conv-nth
output-size-correct-tensors
    by fastforce
 qed
qed

lemma witness-matricization:
assumes i < dim-row Aw' and j < dim-col Aw'
```

shows Aw' $$ (i, j)
$=$ (*if* $i{=}j \wedge (\forall i0{\in}set\ (digit\text{-}encode\ (nths\ (Tensor.dims\ Aw)\ \{n.\ even\ n\})\ i).\ i0 < last\ (butlast\ rs))$ *then 1 else 0*)
proof −
def *is* == *weave* $\{n.\ even\ n\}$
(*digit-encode* (*nths* (*Tensor.dims Aw*) $\{n.\ even\ n\}$) *i*)
(*digit-encode* (*nths* (*Tensor.dims Aw*) $\{n.\ odd\ n\}$) *j*)
have *lookup-eq*: Aw' $$ $(i, j) = Tensor.lookup\ Aw\ is$
using *Aw'-def matricize-def dims-Aw'*(*1*)[*symmetric*, *unfolded A-def*] *dims-Aw'*(*2*)[*symmetric*, *unfolded A-def Collect-neg-eq*]
index-mat(*1*)[*OF* ‹$i <$ *dim-row* Aw'› ‹$j <$ *dim-col* Aw'›] *is-def Collect-neg-eq case-prod-conv*
by (*metis* (*no-types*) *Aw'-def Collect-neg-eq case-prod-conv is-def matricize-def*)
have *is* ◁ *Tensor.dims Aw*
using *is-def valid-index-weave A-def Collect-neg-eq assms digit-encode-valid-index dims-Aw'* **by** *metis*

have *even* (*order Aw*)
unfolding *Aw-def* **using** *assms dims-output-witness even-numeral le-eq-less-or-eq numeral-2-eq-2 numeral-3-eq-3 deep no-zeros y-valid* **by** *fastforce*

have *nths-dimsAw*: *nths* (*Tensor.dims Aw*) (*Collect even*) = *nths* (*Tensor.dims Aw*) $\{n.\ odd\ n\}$
proof −
have *0*:*Tensor.dims* (*tensors-from-net* (*witness-l rs*) $ *y*) = *replicate* (*2* ^ (*length rs* − *2*)) (*last rs*)
using *dims-output-witness*[*OF - no-zeros y-valid*] **using** *deep* **by** *linarith*
show *?thesis* **unfolding** *A-def*
using *nths-replicate*
by (*metis* (*no-types*, *lifting*) *0 Aw-def* ‹*even* (*order Aw*)› *length-replicate length-nths-even*)
qed

have $i = j \longleftrightarrow$ *nths is* (*Collect even*) = *nths is* $\{n.\ odd\ n\}$
proof
have *eq-lengths*: *length* (*digit-encode* (*nths* (*Tensor.dims Aw*) (*Collect even*)) *i*)
= *length* (*digit-encode* (*nths* (*Tensor.dims Aw*) $\{n.\ odd\ n\}$) *j*)
unfolding *length-digit-encode* **by** (*metis* ‹*even* (*order Aw*)› *length-nths-even*)

then show $i = j \Longrightarrow$ *nths is* (*Collect even*) = *nths is* $\{n.\ odd\ n\}$ **unfolding** *is-def*
using *nths-weave*[*of digit-encode* (*nths* (*Tensor.dims Aw*) (*Collect even*)) *i Collect even digit-encode* (*nths* (*Tensor.dims Aw*) $\{n.\ odd\ n\}$) *j*, *unfolded eq-lengths*, *unfolded Collect-neg-eq*[*symmetric*] *card-even mult-2*[*symmetric*] *card-odd*] *nths-dimsAw* **by** *simp*
show *nths is* (*Collect even*) = *nths is* $\{n.\ odd\ n\} \Longrightarrow i = j$ **unfolding** *is-def*
using *nths-weave*[*of digit-encode* (*nths* (*Tensor.dims Aw*) (*Collect even*)) *i Collect even digit-encode* (*nths* (*Tensor.dims Aw*) $\{n.\ odd\ n\}$) *j*, *unfolded*

eq-lengths, *unfolded Collect-neg-eq*[*symmetric*] *card-even mult-2*[*symmetric*] *card-odd*]
using *Divides.mod-less* ‹*nths* (*Tensor.dims Aw*) (*Collect even*) = *nths* (*Tensor.dims Aw*) {*n*. *odd n*}›
deep no-zeros y-valid assms digit-decode-encode dims-Aw' **by** *metis*
qed

have *i=j* ⟹ *set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*) = *set is*
unfolding *is-def nths-dimsAw*
using *set-weave*[*of* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *odd n*}) *j*) *Collect even*
(*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *odd n*}) *j*),
unfolded mult-2[*symmetric*] *card-even Collect-neg-eq*[*symmetric*] *card-odd*]
Un-absorb card-even card-odd mult-2 **by** *blast*
then show *?thesis* **unfolding** *lookup-eq*
using *witness-tensor*[*OF* ‹*is* ◁ *Tensor.dims Aw*›]
by (*simp add*: *A-def* ‹(*i* = *j*) = (*nths is* (*Collect even*) = *nths is* {*n*. *odd n*})›)
qed

definition *rows-with-1* = {*i*. (∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*). *i0* < *last* (*butlast rs*))}

lemma *card-low-digits*:
assumes *m*>*0* ⋀*d*. *d*∈*set ds* ⟹ *m* ≤ *d*
shows *card* {*i*. *i*<*prod-list ds* ∧ (∀ *i0*∈*set* (*digit-encode ds i*). *i0* < *m*)} = *m* ^ (*length ds*)
using *assms* **proof** (*induction ds*)
case *Nil*
then show *?case* **using** *prod-list.Nil* **by** *simp*
next
case (*Cons d ds*)
def *low-digits* == λ*ds i*. *i* < *prod-list ds* ∧ (∀ *i0*∈*set* (*digit-encode ds i*). *i0* < *m*)
have *card* {*i*. *low-digits ds i*} = *m* ^ (*length ds*) **unfolding** *low-digits-def*
by (*simp add*: *Cons.IH Cons.prems*(*1*) *Cons.prems*(*2*))
have *card* {*i*. *low-digits* (*d* # *ds*) *i*} = *card* ({..<*m*} × {*i*. *low-digits ds i*})
proof −
def *f* == λ*p*. *fst p* + *d* ∗ *snd p*
have *inj-on f* ({..<*m*} × {*i*. *low-digits ds i*})
proof (*rule inj-onI*)
fix *x y* **assume** *x* ∈ {..<*m*} × {*i*. *low-digits ds i*} *y* ∈ {..<*m*} × {*i*. *low-digits ds i*} *f x* = *f y*
then have *fst x*<*m fst y*<*m* **by** *auto*
then have *fst x*<*d fst y*<*d* **using** *Cons*(*3*) **by** (*meson list.set-intros*(*1*) *not-le order-trans*)+
then have *f x mod d* = *fst x f y mod d* = *fst y* **unfolding** *f-def* **by** *simp-all*
have *f x div d* = *snd x f y div d* = *snd y* **using** ‹*f x* = *f y*› ‹*f x mod d* = *fst x*› ‹*fst y* < *d*› *f-def* **by** *auto*

show $x = y$ **using** ‹$f\ x = f\ y$› ‹$f\ x\ div\ d = snd\ x$› ‹$f\ x\ mod\ d = fst\ x$› ‹$f\ y\ div\ d = snd\ y$› ‹$f\ y\ mod\ d = fst\ y$› *prod-eqI* **by** *fastforce*
qed
have $f\ `\ (\{..<m\} \times \{i.\ low\text{-}digits\ ds\ i\}) = \{i.\ low\text{-}digits\ (d\ \#\ ds)\ i\}$
proof (*rule subset-antisym*; *rule subsetI*)
fix x **assume** $x \in f\ `\ (\{..<m\} \times \{i.\ low\text{-}digits\ ds\ i\})$
then obtain *i0 i1* **where** $x = i0 + d * i1\ i0 < m\ low\text{-}digits\ ds\ i1$ **using** *f-def* **by** *force*
then have $i0<d$ **using** *Cons*(*3*) **by** (*meson list.set-intros*(*1*) *not-le order-trans*)
show $x \in \{i.\ low\text{-}digits\ (d\ \#\ ds)\ i\}$ **unfolding** *low-digits-def*
proof (*rule*; *rule conjI*)
have $i1 < prod\text{-}list\ ds\ \forall i0{\in}set\ (digit\text{-}encode\ ds\ i1).\ i0 < m$
using ‹*low-digits ds i1*› *low-digits-def* **by** *auto*
show $x < prod\text{-}list\ (d\ \#\ ds)$ **unfolding** *prod-list.Cons* ‹$x = i0 + d * i1$› **using** ‹$i0<d$› ‹$i1 < prod\text{-}list\ ds$›
proof −
have $d \neq 0$
by (*metis* ‹$i0 < d$› *gr-implies-not0*)
then have $(i0 + d * i1)\ div\ (d * prod\text{-}list\ ds) = 0$
by (*simp add*: *Divides.div-mult2-eq* ‹$i0 < d$› ‹$i1 < prod\text{-}list\ ds$›)
then show $i0 + d * i1 < d * prod\text{-}list\ ds$
by (*metis* (*no-types*) ‹$i0 < d$› ‹$i1 < prod\text{-}list\ ds$› *div-eq-0-iff gr-implies-not0 no-zero-divisors*)
qed
show $\forall i0{\in}set\ (digit\text{-}encode\ (d\ \#\ ds)\ x).\ i0 < m$
using ‹$\forall i0{\in}set\ (digit\text{-}encode\ ds\ i1).\ i0 < m$› ‹$i0 < d$› ‹$i0 < m$› ‹$x = i0 + d * i1$› **by** *auto*
qed
next
fix x **assume** $x \in \{i.\ low\text{-}digits\ (d\ \#\ ds)\ i\}$
then have $x < prod\text{-}list\ (d\ \#\ ds)\ \forall i0{\in}set\ (digit\text{-}encode\ (d\ \#\ ds)\ x).\ i0 < m$ **using** *low-digits-def* **by** *auto*
have $x\ mod\ d < m$ **using** ‹$\forall i0{\in}set\ (digit\text{-}encode\ (d\ \#\ ds)\ x).\ i0 < m$›[*unfolded digit-encode.simps*] **by** *simp*
have $x\ div\ d < prod\text{-}list\ ds$ **using** ‹$x < prod\text{-}list\ (d\ \#\ ds)$›[*unfolded prod-list.Cons*]
by (*metis Divides.div-mult2-eq div-eq-0-iff gr-implies-not0 mult-0-right*)
have $\forall i0{\in}set\ (digit\text{-}encode\ ds\ (x\ div\ d)).\ i0 < m$ **by** (*simp add*: ‹$\forall i0{\in}set\ (digit\text{-}encode\ (d\ \#\ ds)\ x).\ i0 < m$›)
have $f\ ((x\ mod\ d),(x\ div\ d)) = x$ **by** (*simp add*: *f-def*)
show $x \in f\ `\ (\{..<m\} \times \{i.\ low\text{-}digits\ ds\ i\})$ **by** (*metis SigmaI* ‹$\forall i0{\in}set\ (digit\text{-}encode\ ds\ (x\ div\ d)).\ i0 < m$› ‹$f\ (x\ mod\ d,\ x\ div\ d) = x$› ‹$x\ div\ d < prod\text{-}list\ ds$› ‹$x\ mod\ d < m$› *image-eqI lessThan-iff low-digits-def mem-Collect-eq*)
qed
then have *bij-betw f* $(\{..<m\} \times \{i.\ low\text{-}digits\ ds\ i\})\ \{i.\ low\text{-}digits\ (d\ \#\ ds)\ i\}$
by (*simp add*: ‹*inj-on f* $(\{..<m\} \times \{i.\ low\text{-}digits\ ds\ i\})$›*bij-betw-def*)
then show *?thesis* **by** (*simp add*: *bij-betw-same-card*)
qed
then show *?case* **unfolding** ‹$card\ \{i.\ low\text{-}digits\ ds\ i\} = m\ \hat{}\ (length\ ds)$› *card-cartesian-product* **using** *low-digits-def* **by** *simp*

qed

lemma *card-rows-with-1*: *card* {*i*∈*rows-with-1*. *i*<*dim-row Aw'*} = *r* ^ *N-half*
proof −
have *1*:{*i*∈*rows-with-1*. *i*<*dim-row Aw'*} = {*i*. *i* < *prod-list* (*nths* (*Tensor.dims Aw*) (*Collect even*)) ∧
(∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) (*Collect even*)) *i*). *i0* < *r*)} (**is** *?A* = *?B*)
proof (*rule subset-antisym*; *rule subsetI*)
fix *i* **assume** *i* ∈ *?A*
then have *i* < *dim-row Aw'* ∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*). *i0* < *last* (*butlast rs*)
using *rows-with-1-def* **by** *auto*
then have *i* < *prod-list* (*nths* (*dims Aw*) (*Collect even*)) **using** *dims-Aw'* **by** *linarith*
then have *digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ◁ *nths* (*dims Aw*) (*Collect even*)
using *digit-encode-valid-index* **by** *auto*
have ∀ *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*). *i0* < *r*
proof
fix *i0* **assume** *1*:*i0* ∈ *set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)
then obtain *k* **where** *k* < *length* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)
digit-encode (*nths* (*dims Aw*) (*Collect even*)) *i* ! *k* = *i0* **by** (*meson in-set-conv-nth*)
have *i0* < *last* (*butlast rs*)
using ‹∀ *i0*∈*set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*). *i0* < *last* (*butlast rs*)› *1* **by** *blast*
have *set* (*nths* (*dims Aw*) (*Collect even*)) ⊆ {*last rs*} **unfolding** *dims-Aw* **using** *subset-eq* **by** *fastforce*
then have *nths* (*dims Aw*) (*Collect even*) ! *k* = *last rs*
using ‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ◁ *nths* (*dims Aw*) (*Collect even*)›
‹*k* < *length* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)›
nth-mem valid-index-length **by** *auto*
then have *i0* < *last rs*
using *valid-index-lt* ‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ! *k* = *i0*›
‹*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i* ◁ *nths* (*dims Aw*) (*Collect even*)›
‹*k* < *length* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) *i*)› *valid-index-length* **by** *fastforce*
then show *i0* < *r* **unfolding** *r-def* **by** (*simp add*: ‹*i0* < *last* (*butlast rs*)›)
qed
then show *i* ∈ *?B* **using** ‹*i* < *prod-list* (*nths* (*dims Aw*) (*Collect even*))› **by** *blast*
next
fix *i* **assume** *i*∈*?B*
then show *i*∈*?A* **by** (*simp add*: *dims-Aw' r-def rows-with-1-def*)

qed
have *2*:$\bigwedge d.\ d \in set\ (nths\ (Tensor.dims\ Aw)\ (Collect\ even)) \Longrightarrow r \leq d$
proof −
fix *d* **assume** $d \in set\ (nths\ (Tensor.dims\ Aw)\ (Collect\ even))$
then have $d \in set\ (Tensor.dims\ Aw)$ **using** *in-set-nthsD* **by** *fast*
then have $d = last\ rs$ **using** *dims-Aw* **by** *simp*
then show $r \leq d$ **by** (*simp add*: *r-def*)
qed
have *3*:$0 < r$ **unfolding** *r-def* **by** (*metis deep diff-diff-cancel diff-zero dual-order.trans in-set-butlastD last-in-set length-butlast list.size*(*3*) *min-def nat-le-linear no-zeros not-numeral-le-zero numeral-le-one-iff rel-simps*(*3*))
have *4*: $length\ (nths\ (Tensor.dims\ Aw)\ (Collect\ even)) = N\text{-}half$
unfolding *length-nths order-Aw* **using** *card-even*[*of N-half*]
by (*metis* (*mono-tags*, *lifting*) *Collect-cong*)
then show *?thesis* **using** *card-low-digits*[*of r nths* (*Tensor.dims Aw*) (*Collect even*)] *1 2 3 4* **by** *metis*
qed

lemma *infinite-rows-with-1*: *infinite rows-with-1*
proof −
def *listpr* == *prod-list* (*nths* (*Tensor.dims Aw*) {*n*. *even n*})
have $\bigwedge i.\ listpr\ dvd\ i \Longrightarrow i \in rows\text{-}with\text{-}1$
proof −
fix *i* **assume** *dvd-i*: *listpr dvd i*
{
fix *i0*::*nat*
assume *i0*∈*set* (*digit-encode* (*nths* (*Tensor.dims Aw*) {*n*. *even n*}) *i*)
then have *i0*=*0* **using** *digit-encode-0 dvd-i listpr-def* **by** *auto*
then have $i0 < last\ (butlast\ rs)$ **using** *deep no-zeros*
by (*metis Nitpick.size-list-simp*(*2*) *One-nat-def Suc-le-lessD in-set-butlastD last-in-set length-butlast length-tl not-numeral-less-zero numeral-2-eq-2 numeral-3-eq-3 numeral-le-one-iff semiring-norm*(*70*))
}
then show *i*∈*rows-with-1* **by** (*simp add*: *rows-with-1-def*)
qed
have *0*:*Tensor.dims Aw* = *replicate* (*2* ^ (*length rs* − *2*)) (*last rs*) **unfolding** *Aw-def*
using *dims-output-witness*[*OF - no-zeros y-valid*] **using** *deep* **by** *linarith*
then have *listpr* > *0* **unfolding** *listpr-def 0*
by (*metis 0 deep last-in-set length-greater-0-conv less-le-trans no-zeros dims-Aw'-pow*(*1*) *dims-Aw'*(*1*) *zero-less-numeral zero-less-power*)
then have *inj* (*op* ∗ *listpr*) **by** (*metis injI mult-left-cancel neq0-conv*)
then show *?thesis* **using** ‹$\bigwedge i.\ listpr\ dvd\ i \Longrightarrow i \in rows\text{-}with\text{-}1$›
by (*meson dvd-triv-left image-subset-iff infinite-iff-countable-subset*)
qed

lemma *witness-submatrix*: $submatrix\ Aw'\ rows\text{-}with\text{-}1\ rows\text{-}with\text{-}1 = 1_m\ (r\ \hat{}\ N\text{-}half)$

proof
show *dim-row* (*submatrix Aw' rows-with-1 rows-with-1*) = *dim-row* (*1*$_m$ (*r* ^ *N-half*))
unfolding *index-one-mat*(*2*) *dim-submatrix*(*1*)
by (*metis* (*full-types*) *set-le-in card-rows-with-1*)
show *dim-col* (*submatrix Aw' rows-with-1 rows-with-1*) = *dim-col* (*1*$_m$ (*r* ^ *N-half*))
by (*metis* ‹*dim-row* (*submatrix Aw' rows-with-1 rows-with-1*) = *dim-row* (*1*$_m$ (*r* ^ *N-half*))› *dim-submatrix*(*1*) *dim-submatrix*(*2*) *index-one-mat*(*2*) *index-one-mat*(*3*) *dims-Aw'-pow*)
show ⋀*i j*. *i* < *dim-row* (*1*$_m$ (*r* ^ *N-half*)) ⟹ *j* < *dim-col* (*1*$_m$ (*r* ^ *N-half*)) ⟹ *submatrix Aw' rows-with-1 rows-with-1* $$ (*i*, *j*) = *1*$_m$ (*r* ^ *N-half*) $$ (*i*, *j*)
proof −
fix *i j* **assume** *i* < *dim-row* (*1*$_m$ (*r* ^ *N-half*)) *j* < *dim-col* (*1*$_m$ (*r* ^ *N-half*))
then have *i* < *r* ^ *N-half j* < *r* ^ *N-half* **by** *auto*
then have *i* < *card* {*i* ∈ *rows-with-1*. *i* < *dim-row Aw'*} *j* < *card* {*i* ∈ *rows-with-1*. *i* < *dim-col Aw'*}
using *card-rows-with-1 dims-Aw'-pow* **by** *auto*
then have *pick rows-with-1 i* < *dim-row Aw' pick rows-with-1 j* < *dim-col Aw'*
using *card-le-pick-inf*[*OF infinite-rows-with-1*, *of dim-row Aw' i*]
using *card-le-pick-inf*[*OF infinite-rows-with-1*, *of dim-col Aw' j*] **by** *force*+
have ∀ *i0*∈*set* (*digit-encode* (*nths* (*dims Aw*) (*Collect even*)) (*pick rows-with-1 i*)). *i0* < *last* (*butlast rs*)
using *infinite-rows-with-1 pick-in-set-inf rows-with-1-def* **by** *auto*
then have *Aw'* $$ (*pick rows-with-1 i*, *pick rows-with-1 j*) = (*if pick rows-with-1 i* = *pick rows-with-1 j then 1 else 0*)
using *witness-matricization*[*OF* ‹*pick rows-with-1 i* < *dim-row Aw'*› ‹*pick rows-with-1 j* < *dim-col Aw'*›] **by** *simp*
then have *submatrix Aw' rows-with-1 rows-with-1* $$ (*i*, *j*) = (*if pick rows-with-1 i* = *pick rows-with-1 j then 1 else 0*)
using *submatrix-index* **by** (*metis* (*no-types*, *lifting*)
‹*dim-col* (*submatrix Aw' rows-with-1 rows-with-1*) = *dim-col* (*1*$_m$ (*r* ^ *N-half*))›
‹*dim-row* (*submatrix Aw' rows-with-1 rows-with-1*) = *dim-row* (*1*$_m$ (*r* ^ *N-half*))›
‹*i* < *dim-row* (*1*$_m$ (*r* ^ *N-half*))› ‹*j* < *r* ^ *N-half*› *dim-submatrix*(*1*) *dim-submatrix*(*2*) *index-one-mat*(*3*))
then have *submatrix Aw' rows-with-1 rows-with-1* $$ (*i*, *j*) = (*if i* = *j then 1 else 0*)
using *pick-eq-iff-inf*[*OF infinite-rows-with-1*] **by** *auto*
then show *submatrix Aw' rows-with-1 rows-with-1* $$ (*i*, *j*) = *1*$_m$ (*r* ^ *N-half*) $$ (*i*, *j*)
by (*simp add*: ‹*i* < *r* ^ *N-half*› ‹*j* < *r* ^ *N-half*›)
qed
qed

lemma *witness-det*: *det* (*submatrix Aw' rows-with-1 rows-with-1*) ≠ *0* **unfolding** *witness-submatrix* **by** *simp*

end

end

27 Less common functions on lists

theory *PP-More-List2*
imports
Main
begin

definition *strip-while* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list*
where
strip-while P = *rev* ∘ *dropWhile P* ∘ *rev*

lemma *strip-while-rev* [*simp*]:
strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
by (*simp add*: *strip-while-def*)

lemma *strip-while-Nil* [*simp*]:
strip-while P [] = []
by (*simp add*: *strip-while-def*)

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
by (*simp add*: *strip-while-def*)

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
by (*simp add*: *strip-while-def*)

lemma *strip-while-Cons* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-eq-Nil* [*simp*]:
strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)
by (*simp add*: *strip-while-def*)

lemma *strip-while-eq-Cons-rec*:
strip-while P (*x* # *xs*) = *x* # *strip-while P xs* ⟷ ¬ (*P x* ∧ (∀ *x*∈*set xs*. *P x*))
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-not-last* [*simp*]:
¬ *P* (*last xs*) ⟹ *strip-while P xs* = *xs*
by (*cases xs rule*: *rev-cases*) *simp-all*

lemma *split-strip-while-append*:
fixes *xs* :: *'a list*

obtains *ys zs* :: *'a list*
where *strip-while P xs* = *ys* **and** ∀ *x*∈*set zs*. *P x* **and** *xs* = *ys* @ *zs*
proof (*rule that*)
show *strip-while P xs* = *strip-while P xs* **..**
show ∀ *x*∈*set* (*rev* (*takeWhile P* (*rev xs*))). *P x* **by** (*simp add*: *takeWhile-eq-all-conv* [*symmetric*])
have *rev xs* = *rev* (*strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*)))
by (*simp add*: *strip-while-def*)
then show *xs* = *strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*))
by (*simp only*: *rev-is-rev-conv*)
qed

lemma *strip-while-snoc* [*simp*]:
strip-while P (*xs* @ [*x*]) = (*if P x then strip-while P xs else xs* @ [*x*])
by (*simp add*: *strip-while-def*)

lemma *strip-while-map*:
strip-while P (*map f xs*) = *map f* (*strip-while* (*P* ∘ *f*) *xs*)
by (*simp add*: *strip-while-def rev-map dropWhile-map*)

definition *no-leading* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool*
where
no-leading P xs ⟷ (*xs* ≠ [] ⟶ ¬ *P* (*hd xs*))

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
by (*simp add*: *no-leading-def*)

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading P (*x* # *xs*) ⟷ ¬ *P x*
by (*simp add*: *no-leading-def*)

lemma *no-leading-append* [*simp*]:
no-leading P (*xs* @ *ys*) ⟷ *no-leading P xs* ∧ (*xs* = [] ⟶ *no-leading P ys*)
by (*induct xs*) *simp-all*

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (*dropWhile P xs*)
by (*induct xs*) *simp-all*

lemma *dropWhile-eq-obtain-leading*:
assumes *dropWhile P xs* = *ys*
obtains *zs* **where** *xs* = *zs* @ *ys* **and** ⋀*z*. *z* ∈ *set zs* ⟹ *P z* **and** *no-leading P ys*
proof −
from *assms* **have** ∃ *zs*. *xs* = *zs* @ *ys* ∧ (∀ *z* ∈ *set zs*. *P z*) ∧ *no-leading P ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*

```
  next
    case (Cons x xs ys)
    show ?case proof (cases P x)
      case True with Cons.hyps [of ys] Cons.prems
      have ∃ zs. xs = zs @ ys ∧ (∀ a∈set zs. P a) ∧ no-leading P ys
        by simp
      then obtain zs where xs = zs @ ys and ⋀z. z ∈ set zs ⟹ P z
        and ∗: no-leading P ys
        by blast
      with True have x # xs = (x # zs) @ ys and ⋀z. z ∈ set (x # zs) ⟹ P z
        by auto
      with ∗ show ?thesis
        by blast    next
      case False
      with Cons show ?thesis by (cases ys) simp-all
    qed
  qed
  with that show thesis
    by blast
qed

lemma dropWhile-idem-iff:
  dropWhile P xs = xs ⟷ no-leading P xs
  by (cases xs) (auto elim: dropWhile-eq-obtain-leading)


abbreviation no-trailing :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  no-trailing P xs ≡ no-leading P (rev xs)

lemma no-trailing-unfold:
  no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))
  by (induct xs) simp-all

lemma no-trailing-Nil [simp, intro!]:
  no-trailing P []
  by simp

lemma no-trailing-Cons [simp]:
  no-trailing P (x # xs) ⟷ no-trailing P xs ∧ (xs = [] ⟶ ¬ P x)
  by simp

lemma no-trailing-append-Cons [simp]:
  no-trailing P (xs @ y # ys) ⟷ no-trailing P (y # ys)
  by simp

lemma no-trailing-strip-while [simp]:
  no-trailing P (strip-while P xs)
  by (induct xs rule: rev-induct) simp-all
```

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs = ys*
obtains *zs* **where** *xs = ys @ zs* **and** $\bigwedge z.\ z \in set\ zs \Longrightarrow P\ z$ **and** *no-trailing P ys*
proof −
from *assms* **have** *rev (rev (dropWhile P (rev xs))) = rev ys*
by (*simp add*: *strip-while-def*)
then have *dropWhile P (rev xs) = rev ys*
by *simp*
then obtain *zs* **where** *A*: *rev xs = zs @ rev ys* **and** *B*: $\bigwedge z.\ z \in set\ zs \Longrightarrow P\ z$
and *C*: *no-trailing P ys*
using *dropWhile-eq-obtain-leading* **by** *blast*
from *A* **have** *rev (rev xs) = rev (zs @ rev ys)*
by *simp*
then have *xs = ys @ rev zs*
by *simp*
moreover from *B* **have** $\bigwedge z.\ z \in set\ (rev\ zs) \Longrightarrow P\ z$
by *simp*
ultimately show *thesis* **using** *that C* **by** *blast*
qed

lemma *strip-while-idem-iff*:
strip-while P xs = xs $\longleftrightarrow$ *no-trailing P xs*
proof −
def *ys* ≡ *rev xs*
moreover have *strip-while P (rev ys) = rev ys* $\longleftrightarrow$ *no-trailing P (rev ys)*
by (*simp add*: *dropWhile-idem-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma *no-trailing-map*:
no-trailing P (map f xs) = no-trailing (P ∘ f) xs
by (*simp add*: *last-map no-trailing-unfold*)

lemma *no-trailing-upt* [*simp*]:
no-trailing P [n..<m] $\longleftrightarrow$ *(n < m* $\longrightarrow$ ¬ *P (m − 1))*
by (*auto simp add*: *no-trailing-unfold*)

definition *nth-default* :: *'a* ⇒ *'a list* ⇒ *nat* ⇒ *'a*
where
nth-default dflt xs n = (if n < length xs then xs ! n else dflt)

lemma *nth-default-nth*:
n < length xs ⟹ *nth-default dflt xs n = xs ! n*
by (*simp add*: *nth-default-def*)

lemma *nth-default-beyond*:

length xs $\leq$ *n* $\Longrightarrow$ *nth-default dflt xs n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* $\Rightarrow$ *x* | *Suc n'* $\Rightarrow$ *nth-default dflt xs n'*)
by (*simp add*: *nth-default-def split*: *nat.split*)

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-append*:
nth-default dflt (*xs* @ *ys*) *n* =
(*if n* < *length xs then nth xs n else nth-default dflt ys* (*n* − *length xs*))
by (*auto simp add*: *nth-default-def nth-append*)

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
by (*simp add*: *fun-eq-iff nth-default-append*) (*simp add*: *nth-default-def*)

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
by (*auto simp add*: *nth-default-def fun-eq-iff nth-append*)

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = *dflt* $\longleftrightarrow$ (*k* < *length xs* $\longrightarrow$ *xs* ! *k* = *dflt*)
by (*simp add*: *nth-default-def*)

lemma *in-enumerate-iff-nth-default-eq*:
x $\neq$ *dflt* $\Longrightarrow$ (*n*, *x*) $\in$ *set* (*enumerate 0 xs*) $\longleftrightarrow$ *nth-default dflt xs n* = *x*
by (*auto simp add*: *nth-default-def in-set-conv-nth enumerate-eq-zip*)

lemma *last-conv-nth-default*:
assumes *xs* $\neq$ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
using *assms* **by** (*simp add*: *nth-default-def last-conv-nth*)

lemma *nth-default-map-eq*:
f dflt' = dflt $\Longrightarrow$ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt' xs n*)
by (*simp add*: *nth-default-def*)

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* $\neq$ *dflt*}
by (*simp add*: *nth-default-def*)

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k* $\neq$ *dflt*} = *map fst* (*filter* (λ(-, *x*). *x* $\neq$ *dflt*) (*enumerate 0 xs*))
by (*rule sorted-distinct-set-unique*) (*auto simp add*: *nth-default-def in-set-conv-nth sorted-filter distinct-map-filter enumerate-eq-zip intro*: *rev-image-eqI*)

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
proof −
have ∗: *map* (*nth-default x xs*) [*0*..<*length xs*] = *map* (*List.nth xs*) [*0*..<*length xs*]
by (*rule map-cong*) (*simp-all add*: *nth-default-nth*)
show *?thesis* **by** (*simp add*: ∗ *map-nth*)
qed

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
by (*auto simp add*: *nth-default-def* [*abs-def*] *in-set-conv-nth*)

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
proof −
have *length* (*dropWhile P* (*rev xs*)) + *length* (*takeWhile P* (*rev xs*)) = *length xs*
by (*subst add.commute*)
(*simp add*: *arg-cong* [**where** *f*=*length*, *OF takeWhile-dropWhile-id*, *unfolded length-append*])
then show *?thesis* **using** *assms*
by (*simp add*: *strip-while-def rev-nth dropWhile-nth*)
qed

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
unfolding *strip-while-def o-def length-rev*
by (*subst* (*2*) *length-rev*[*symmetric*])
(*simp add*: *strip-while-def length-dropWhile-le del*: *length-rev*)

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
by (*induct xs rule*: *rev-induct*) *auto*

```
lemma nth-default-eq-iff:
  nth-default dflt xs = nth-default dflt ys
    ⟷ strip-while (HOL.eq dflt) xs = strip-while (HOL.eq dflt) ys (is ?P ⟷
?Q)
proof
  let ?xs = strip-while (HOL.eq dflt) xs and ?ys = strip-while (HOL.eq dflt) ys
  assume ?P
  then have eq: nth-default dflt ?xs = nth-default dflt ?ys
    by simp
  have len: length ?xs = length ?ys
  proof (rule ccontr)
    assume len: length ?xs ≠ length ?ys
    { fix xs ys :: 'a list
     let ?xs = strip-while (HOL.eq dflt) xs and ?ys = strip-while (HOL.eq dflt) ys
      assume eq: nth-default dflt ?xs = nth-default dflt ?ys
      assume len: length ?xs < length ?ys
      then have length ?ys > 0 by arith
      then have ?ys ≠ [] by simp
      with last-conv-nth-default [of ?ys dflt]
      have last ?ys = nth-default dflt ?ys (length ?ys − 1)
        by auto
      moreover from ‹?ys ≠ []› no-trailing-strip-while [of HOL.eq dflt ys]
        have last ?ys ≠ dflt by (simp add: no-trailing-unfold)
      ultimately have nth-default dflt ?xs (length ?ys − 1) ≠ dflt
        using eq by simp
      moreover from len have length ?ys − 1 ≥ length ?xs by simp
      ultimately have False by (simp only: nth-default-beyond) simp
    }
    from this [of xs ys] this [of ys xs] len eq show False
      by (auto simp only: linorder-class.neq-iff)
  qed
  then show ?Q
  proof (rule nth-equalityI [rule-format])
    fix n
    assume n < length ?xs
    moreover with len have n < length ?ys
      by simp
    ultimately have xs: nth-default dflt ?xs n = ?xs ! n
      and ys: nth-default dflt ?ys n = ?ys ! n
      by (simp-all only: nth-default-nth)
    with eq show ?xs ! n = ?ys ! n
      by simp
  qed
next
  assume ?Q
   then have nth-default dflt (strip-while (HOL.eq dflt) xs) = nth-default dflt
(strip-while (HOL.eq dflt) ys)
    by simp
  then show ?P
```

by *simp*
qed

end

28 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

theory *PP-Poly-Mapping*
imports
HOL−Library.Groups-Big-Fun
HOL−Library.Fun-Lexorder
PP-More-List2
begin

28.1 Preliminary: auxiliary operations for 'almost everywhere zero'

A central notion for polynomials are functions being 'almost everywhere zero'. For these we provide some auxiliary definitions and lemmas.

lemma *finite-mult-not-eq-zero-leftI*:
fixes $f :: 'b \Rightarrow 'a :: mult\text{-}zero$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows *finite* $\{a.\ g\ a * f\ a \neq 0\}$
proof −
have $\{a.\ g\ a * f\ a \neq 0\} \subseteq \{a.\ f\ a \neq 0\}$ **by** *auto*
then show *?thesis* **using** *assms* **by** (*rule finite-subset*)
qed

lemma *finite-mult-not-eq-zero-rightI*:
fixes $f :: 'b \Rightarrow 'a :: mult\text{-}zero$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows *finite* $\{a.\ f\ a * g\ a \neq 0\}$
proof −
have $\{a.\ f\ a * g\ a \neq 0\} \subseteq \{a.\ f\ a \neq 0\}$ **by** *auto*
then show *?thesis* **using** *assms* **by** (*rule finite-subset*)
qed

lemma *finite-mult-not-eq-zero-prodI*:
fixes $f\ g :: 'a \Rightarrow 'b :: semiring\text{-}0$
assumes *finite* $\{a.\ f\ a \neq 0\}$ (**is** *finite ?F*)
assumes *finite* $\{b.\ g\ b \neq 0\}$ (**is** *finite ?G*)
shows *finite* $\{(a, b).\ f\ a * g\ b \neq 0\}$
proof −
from *assms* **have** *finite* $(?F \times ?G)$
by *blast*
then have *finite* $\{(a, b).\ f\ a \neq 0 \wedge g\ b \neq 0\}$
by *simp*

```
  then show ?thesis
    by (rule rev-finite-subset) auto
qed

lemma finite-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add ⇒ 'b::semiring-0
  assumes finite {a. f a ≠ 0} (is finite ?F)
  assumes finite {b. g b ≠ 0} (is finite ?G)
  shows finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0} (is finite ?FG)
proof −
  from assms have finite (?F × ?G)
    by (simp add: finite-cartesian-product-iff)
  then have finite (case-prod plus ‘ (?F × ?G))
    by (rule finite-imageI)
  also have case-prod plus ‘ (?F × ?G) = ?FG
    by auto
  finally show ?thesis
    by simp
qed

lemma finite-mult-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add ⇒ 'b::semiring-0
  assumes finite {a. f a ≠ 0}
  assumes finite {b. g b ≠ 0}
  shows finite {a + b | a b. f a * g b ≠ 0}
proof −
  from assms
  have finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0}
    by (rule finite-not-eq-zero-sumI)
  then show ?thesis
    by (rule rev-finite-subset) (auto dest: mult-not-zero)
qed

lemma finite-Sum-any-not-eq-zero-weakenI:
  assumes finite {a. ∃ b. f a b ≠ 0}
  shows finite {a. Sum-any (f a) ≠ 0}
proof −
  have {a. Sum-any (f a) ≠ 0} ⊆ {a. ∃ b. f a b ≠ 0}
    by (auto elim: Sum-any.not-neutral-obtains-not-neutral)
  then show ?thesis using assms by (rule finite-subset)
qed

context zero
begin

definition when :: 'a ⇒ bool ⇒ 'a (infixl when 20)
where
  (a when P) = (if P then a else 0)
```

Case distinctions always complicate matters, particularly when nested.

The *op when* operation allows to minimise these if $0::'a$ is the false-case value and makes proof obligations much more readable.

lemma *when* [*simp*]:
  $P \Longrightarrow (a\ when\ P) = a$
  $\neg\ P \Longrightarrow (a\ when\ P) = 0$
  by (*simp-all add*: *when-def*)

lemma *when-simps* [*simp*]:
  $(a\ when\ True) = a$
  $(a\ when\ False) = 0$
  by *simp-all*

lemma *when-cong*:
  assumes $P \longleftrightarrow Q$
    and $Q \Longrightarrow a = b$
  shows $(a\ when\ P) = (b\ when\ Q)$
  using *assms* **by** (*simp add*: *when-def*)

lemma *zero-when* [*simp*]:
  $(0\ when\ P) = 0$
  by (*simp add*: *when-def*)

lemma *when-when*:
  $(a\ when\ P\ when\ Q) = (a\ when\ P \wedge Q)$
  by (*cases Q*) *simp-all*

lemma *when-commute*:
  $(a\ when\ Q\ when\ P) = (a\ when\ P\ when\ Q)$
  by (*simp add*: *when-when conj-commute*)

lemma *when-neq-zero* [*simp*]:
  $(a\ when\ P) \neq 0 \longleftrightarrow P \wedge a \neq 0$
  by (*cases P*) *simp-all*

end

context *monoid-add*
begin

lemma *when-add-distrib*:
  $(a + b\ when\ P) = (a\ when\ P) + (b\ when\ P)$
  by (*simp add*: *when-def*)

end

context *semiring-1*
begin

lemma *zero-power-eq*:

0 ^ n = (1 when n = 0)
by (*simp add*: *power-0-left*)

end

context *comm-monoid-add*
begin

lemma *Sum-any-when-equal* [*simp*]:
(∑ *a*. (*f a when a = b*)) = *f b*
by (*simp add*: *when-def*)

lemma *Sum-any-when-equal'* [*simp*]:
(∑ *a*. (*f a when b = a*)) = *f b*
by (*simp add*: *when-def*)

lemma *Sum-any-when-independent*:
(∑ *a*. *g a when P*) = ((∑ *a*. *g a*) *when P*)
by (*cases P*) *simp-all*

lemma *Sum-any-when-dependent-prod-right*:
(∑ (*a*, *b*). *g a when b = h a*) = (∑ *a*. *g a*)
proof −
have *inj-on* (λ*a*. (*a*, *h a*)) {*a*. *g a* ≠ *0*}
by (*rule inj-onI*) *auto*
then show *?thesis* **unfolding** *Sum-any.expand-set*
by (*rule sum.reindex-cong*) *auto*
qed

lemma *Sum-any-when-dependent-prod-left*:
(∑ (*a*, *b*). *g b when a = h b*) = (∑ *b*. *g b*)
proof −
have (∑ (*a*, *b*). *g b when a = h b*) = (∑ (*b*, *a*). *g b when a = h b*)
by (*rule Sum-any.reindex-cong* [*of prod.swap*]) (*simp-all add*: *fun-eq-iff*)
then show *?thesis* **by** (*simp add*: *Sum-any-when-dependent-prod-right*)
qed

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
(*a* − *b when P*) = (*a when P*) − (*b when P*)
by (*simp add*: *when-def*)

end

context *group-add*

begin

lemma *when-uminus-distrib*:
(− *a when P*) = − (*a when P*)
by (*simp add*: *when-def*)

end

context *mult-zero*
begin

lemma *mult-when*:
a ∗ (*b when P*) = (*a* ∗ *b when P*)
by (*cases P*) *simp-all*

lemma *when-mult*:
(*a when P*) ∗ *b* = (*a* ∗ *b when P*)
by (*cases P*) *simp-all*

end

28.2 Type definition

The following type is of central importance:

typedef (**overloaded**) ('*a*, '*b*) *poly-mapping* ((- ⇒$_0$ /-) [*1*, *0*] *0*) =
{*f* :: '*a* ⇒ '*b*::*zero*. *finite* {*x*. *f x* ≠ *0*}}
morphisms *lookup Abs-poly-mapping*
proof −
have (λ-::'*a*. (*0* :: '*b*)) ∈ *?poly-mapping* **by** *simp*
then show *?thesis* **by** (*blast intro*!: *exI*)
qed

lemma *lookup-Abs-poly-mapping*:
finite {*x*. *f x* ≠ *0*} ⟹ *lookup* (*Abs-poly-mapping f*) = *f*
using *Abs-poly-mapping-inverse* [*of f*] **by** *simp*

lemma *finite-lookup* [*simp*]:
finite {*k*. *lookup f k* ≠ *0*}
using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *finite-lookup-nat* [*simp*]:
fixes *f* :: '*a* ⇒$_0$ *nat*
shows *finite* {*k*. *0* < *lookup f k*}
using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *poly-mapping-eqI*:
assumes ⋀*k*. *lookup f k* = *lookup g k*
shows *f* = *g*
using *assms* **unfolding** *poly-mapping.lookup-inject* [*symmetric*] **by** *auto*

We model the universe of functions being 'almost everywhere zero' by means of a separate type $'a \Rightarrow_0 'b$. For convenience we provide a suggestive infix syntax which is a variant of the usual function space syntax. Conversion between both types happens through the morphisms

lookup::$('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow 'b$

Abs-poly-mapping::$('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow_0 'b$

satisfying

Abs-poly-mapping (*lookup* *?x*) = *?x*

finite $\{x.\ ?f\ x \neq (0::?'b)\} \Longrightarrow$ *lookup* (*Abs-poly-mapping* *?f*) = *?f*

Luckily, we have rarely to deal with those low-level morphisms explicitly but rely on Isabelle's *lifting* package with its method *transfer* and its specification tool *lift-definition*.

setup-lifting *type-definition-poly-mapping*

$'a \Rightarrow_0 'b$ serves distinctive purposes:

1. A clever nesting as $(nat \Rightarrow_0 nat) \Rightarrow_0 'a$ later in theory *MPoly* gives a suitable representation type for polynomials 'almost for free': Interpreting $nat \Rightarrow_0 nat$ as a mapping from variable identifiers to exponents yields monomials, and the whole type maps monomials to coefficients. Lets call this the *ultimate interpretation*.

2. A further more specialised type isomorphic to $nat \Rightarrow_0 'a$ is apt to direct implementation using code generation [**?**].

Note that despite the names 'mapping' and 'lookup' suggest something implementation-near, it is best to keep $'a \Rightarrow_0 'b$ as an abstract *algebraic* type providing operations like 'addition', 'multiplication' without any notion of key-order, data structures etc. This implementations-specific notions are easily introduced later for particular implementations but do not provide any gain for specifying logical properties of polynomials.

28.3 Additive structure

The additive structure covers the usual operations *0*, + and (unary and binary) −. Recalling the ultimate interpretation, it is obvious that these have just lift the corresponding operations on values to mappings.

Isabelle has a rich hierarchy of algebraic type classes, and in such situations of pointwise lifting a typical pattern is to have instantiations for a considerable number of type classes.

The operations themselves are specified using *lift-definition*, where the proofs of the 'almost everywhere zero' property can be significantly involved.

The *lookup* operation is supposed to be usable explicitly (unless in other situations where the morphisms between types are somehow internal to the *lifting* package). Hence it is good style to provide explicit rewrite rules how *lookup* acts on operations immediately.

instantiation *poly-mapping* :: (*type*, *zero*) *zero*
begin

lift-definition *zero-poly-mapping* :: $'a \Rightarrow_0 'b$
  is $\lambda k.\ 0$
  by *simp*

instance ..

end

lemma *lookup-zero* [*simp*]:
  lookup 0 k = 0
  by *transfer rule*

instantiation *poly-mapping* :: (*type*, *monoid-add*) *monoid-add*
begin

lift-definition *plus-poly-mapping* ::
    $('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b$
  is $\lambda f1\ f2\ k.\ f1\ k + f2\ k$
proof −
  fix *f1 f2* :: $'a \Rightarrow 'b$
  assume *finite* $\{k.\ f1\ k \neq 0\}$
    and *finite* $\{k.\ f2\ k \neq 0\}$
  then have *finite* $(\{k.\ f1\ k \neq 0\} \cup \{k.\ f2\ k \neq 0\})$ **by** *auto*
  moreover have $\{x.\ f1\ x + f2\ x \neq 0\} \subseteq \{k.\ f1\ k \neq 0\} \cup \{k.\ f2\ k \neq 0\}$
    by *auto*
  ultimately show *finite* $\{x.\ f1\ x + f2\ x \neq 0\}$
    by (*blast intro*: *finite-subset*)
qed

instance
  by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

end

lemma *lookup-add*:
  lookup $(f + g)\ k = lookup\ f\ k + lookup\ g\ k$
  by *transfer rule*

instance *poly-mapping* :: (*type*, *comm-monoid-add*) *comm-monoid-add*

by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

instantiation *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ '*a* ⇒$_0$ '*b*
is λ*f1 f2 k*. *f1 k* − *f2 k*
proof −
fix *f1 f2* :: '*a* ⇒ '*b*
assume *finite* {*k*. *f1 k* ≠ *0*}
and *finite* {*k*. *f2 k* ≠ *0*}
then have *finite* ({*k*. *f1 k* ≠ *0*} ∪ {*k*. *f2 k* ≠ *0*}) **by** *auto*
moreover have {*x*. *f1 x* − *f2 x* ≠ *0*} ⊆ {*k*. *f1 k* ≠ *0*} ∪ {*k*. *f2 k* ≠ *0*}
by *auto*
ultimately show *finite* {*x*. *f1 x* − *f2 x* ≠ *0*} **by** (*blast intro*: *finite-subset*)
qed

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff diff-diff-add*)+

end

instantiation *poly-mapping* :: (*type*, *ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ '*a* ⇒$_0$ '*b*
is *uminus*
by *simp*

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

end

lemma *lookup-uminus* [*simp*]:
lookup (− *f*) *k* = − *lookup f k*
by *transfer simp*

lemma *lookup-minus*:
lookup (*f* − *g*) *k* = *lookup f k* − *lookup g k*
by *transfer rule*

28.4 Multiplicative structure

instantiation *poly-mapping* :: (*zero*, *zero-neq-one*) *zero-neq-one*
begin

lift-definition *one-poly-mapping* :: $'a \Rightarrow_0 'b$
is $\lambda k.\ 1\ when\ k = 0$
by *simp*

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff*)

end

lemma *lookup-one*:
$lookup\ 1\ k = (1\ when\ k = 0)$
by *transfer rule*

lemma *lookup-one-zero* [*simp*]:
$lookup\ 1\ 0 = 1$
by *transfer simp*

definition *prod-fun* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a$::*monoid-add* $\Rightarrow 'b$::*semiring-0*
where
$prod\text{-}fun\ f1\ f2\ k = (\sum l.\ f1\ l * (\sum q.\ (f2\ q\ when\ k = l + q)))$

lemma *prod-fun-unfold-prod*:
fixes $f\ g :: 'a :: monoid\text{-}add \Rightarrow 'b$::*semiring-0*
assumes *fin-f*: $finite\ \{a.\ f\ a \neq 0\}$
assumes *fin-g*: $finite\ \{b.\ g\ b \neq 0\}$
shows $prod\text{-}fun\ f\ g\ k = (\sum (a,\ b).\ f\ a * g\ b\ when\ k = a + b)$
proof −
let $?C = \{a.\ f\ a \neq 0\} \times \{b.\ g\ b \neq 0\}$
from *fin-f fin-g* **have** $finite\ ?C$ **by** *blast*
moreover have $\{a.\ \exists b.\ (f\ a * g\ b\ when\ k = a + b) \neq 0\} \times$
$\{b.\ \exists a.\ (f\ a * g\ b\ when\ k = a + b) \neq 0\} \subseteq \{a.\ f\ a \neq 0\} \times \{b.\ g\ b \neq 0\}$
by *auto*
ultimately show *?thesis* **using** *fin-g*
by (*auto simp add*: *prod-fun-def*
Sum-any.cartesian-product [*of* $\{a.\ f\ a \neq 0\} \times \{b.\ g\ b \neq 0\}$] *Sum-any-right-distrib*
mult-when)
qed

lemma *finite-prod-fun*:
fixes *f1 f2* :: $'a :: monoid\text{-}add \Rightarrow 'b :: semiring\text{-}0$
assumes *fin1*: $finite\ \{l.\ f1\ l \neq 0\}$
and *fin2*: $finite\ \{q.\ f2\ q \neq 0\}$
shows $finite\ \{k.\ prod\text{-}fun\ f1\ f2\ k \neq 0\}$
proof −
have ∗: $finite\ \{k.\ (\exists l.\ f1\ l \neq 0 \wedge (\exists q.\ f2\ q \neq 0 \wedge k = l + q))\}$
using *assms* **by** *simp*
{ fix *k l*
have $\{q.\ (f2\ q\ when\ k = l + q) \neq 0\} \subseteq \{q.\ f2\ q \neq 0 \wedge k = l + q\}$ **by** *auto*

with *fin2* **have** *sum f2* $\{q.\ f2\ q \neq 0 \wedge k = l + q\} = (\sum q.\ (f2\ q\ when\ k = l + q))$
by (*simp add*: *Sum-any.expand-superset* [*of* $\{q.\ f2\ q \neq 0 \wedge k = l + q\}$]) **}**
note *aux* = *this*
have $\{k.\ (\sum l.\ f1\ l * sum\ f2\ \{q.\ f2\ q \neq 0 \wedge k = l + q\}) \neq 0\}$
$\subseteq \{k.\ (\exists l.\ f1\ l * sum\ f2\ \{q.\ f2\ q \neq 0 \wedge k = l + q\} \neq 0)\}$
by (*auto elim*!: *Sum-any.not-neutral-obtains-not-neutral*)
also have $\ldots \subseteq \{k.\ (\exists l.\ f1\ l \neq 0 \wedge sum\ f2\ \{q.\ f2\ q \neq 0 \wedge k = l + q\} \neq 0)\}$
by (*auto dest*: *mult-not-zero*)
also have $\ldots \subseteq \{k.\ (\exists l.\ f1\ l \neq 0 \wedge (\exists q.\ f2\ q \neq 0 \wedge k = l + q))\}$
by (*auto elim*!: *sum.not-neutral-contains-not-neutral*)
finally have *finite* $\{k.\ (\sum l.\ f1\ l * sum\ f2\ \{q.\ f2\ q \neq 0 \wedge k = l + q\}) \neq 0\}$
using $*$ **by** (*rule finite-subset*)
with *aux* **have** *finite* $\{k.\ (\sum l.\ f1\ l * (\sum q.\ (f2\ q\ when\ k = l + q))) \neq 0\}$
by *simp*
with *fin2* **show** *?thesis*
by (*simp add*: *prod-fun-def*)
qed

instantiation *poly-mapping* :: (*monoid-add*, *semiring-0*) *semiring-0*
begin

lift-definition *times-poly-mapping* :: $('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b$
is *prod-fun*
by(*rule finite-prod-fun*)

instance
proof
fix *a b c* :: $'a \Rightarrow_0 'b$
show $a * b * c = a * (b * c)$
proof *transfer*
fix *f g h* :: $'a \Rightarrow 'b$
assume *fin-f*: *finite* $\{a.\ f\ a \neq 0\}$ (**is** *finite ?F*)
assume *fin-g*: *finite* $\{b.\ g\ b \neq 0\}$ (**is** *finite ?G*)
assume *fin-h*: *finite* $\{c.\ h\ c \neq 0\}$ (**is** *finite ?H*)
from *fin-f fin-g* **have** *fin-fg*: *finite* $\{(a, b).\ f\ a * g\ b \neq 0\}$ (**is** *finite ?FG*)
by (*rule finite-mult-not-eq-zero-prodI*)
from *fin-g fin-h* **have** *fin-gh*: *finite* $\{(b, c).\ g\ b * h\ c \neq 0\}$ (**is** *finite ?GH*)
by (*rule finite-mult-not-eq-zero-prodI*)
from *fin-f fin-g* **have** *fin-fg'*: *finite* $\{a + b \mid a\ b.\ f\ a * g\ b \neq 0\}$ (**is** *finite ?FG'*)
by (*rule finite-mult-not-eq-zero-sumI*)
then have *fin-fg''*: *finite* $\{d.\ (\sum (a, b).\ f\ a * g\ b\ when\ d = a + b) \neq 0\}$
by (*auto intro*: *finite-Sum-any-not-eq-zero-weakenI*)
from *fin-g fin-h* **have** *fin-gh'*: *finite* $\{b + c \mid b\ c.\ g\ b * h\ c \neq 0\}$ (**is** *finite ?GH'*)
by (*rule finite-mult-not-eq-zero-sumI*)
then have *fin-gh''*: *finite* $\{d.\ (\sum (b, c).\ g\ b * h\ c\ when\ d = b + c) \neq 0\}$
by (*auto intro*: *finite-Sum-any-not-eq-zero-weakenI*)
show *prod-fun* (*prod-fun f g*) *h* = *prod-fun f* (*prod-fun g h*) (**is** *?lhs* = *?rhs*)

proof
fix k
from *fin-f fin-g fin-h fin-fg''*
have *?lhs k* = $(\sum (ab,\ c).\ (\sum (a,\ b).\ f\ a * g\ b$ *when* $ab = a + b) * h\ c$ *when* $k = ab + c)$
by (*simp add*: *prod-fun-unfold-prod*)
also have $\ldots = (\sum (ab,\ c).\ (\sum (a,\ b).\ f\ a * g\ b * h\ c$ *when* $k = ab + c$ *when* $ab = a + b))$
apply (*subst Sum-any-left-distrib*)
using *fin-fg* **apply** (*simp add*: *split-def*)
apply (*subst Sum-any-when-independent* [*symmetric*])
apply (*simp add*: *when-when when-mult mult-when split-def conj-commute*)
done
also have $\ldots = (\sum (ab,\ c,\ a,\ b).\ f\ a * g\ b * h\ c$ *when* $k = ab + c$ *when* $ab = a + b)$
apply (*subst Sum-any.cartesian-product2* [*of* $(?FG' \times ?H) \times ?FG$])
apply (*auto simp add*: *finite-cartesian-product-iff fin-fg fin-fg' fin-h dest*: *mult-not-zero*)
done
also have $\ldots = (\sum (ab,\ c,\ a,\ b).\ f\ a * g\ b * h\ c$ *when* $k = a + b + c$ *when* $ab = a + b)$
by (*rule Sum-any.cong*) (*simp add*: *split-def when-def*)
also have $\ldots = (\sum (ab,\ cab).\ (case\ cab\ of\ (c,\ a,\ b) \Rightarrow f\ a * g\ b * h\ c$ *when* $k = a + b + c$
when $ab = (case\ cab\ of\ (c,\ a,\ b) \Rightarrow a + b))$
by (*simp add*: *split-def*)
also have $\ldots = (\sum (c,\ a,\ b).\ f\ a * g\ b * h\ c$ *when* $k = a + b + c)$
by (*simp add*: *Sum-any-when-dependent-prod-left*)
also have $\ldots = (\sum (bc,\ cab).\ (case\ cab\ of\ (c,\ a,\ b) \Rightarrow f\ a * g\ b * h\ c$ *when* $k = a + b + c$
when $bc = (case\ cab\ of\ (c,\ a,\ b) \Rightarrow b + c))$
by (*simp add*: *Sum-any-when-dependent-prod-left*)
also have $\ldots = (\sum (bc,\ c,\ a,\ b).\ f\ a * g\ b * h\ c$ *when* $k = a + b + c$ *when* $bc = b + c)$
by (*simp add*: *split-def*)
also have $\ldots = (\sum (bc,\ c,\ a,\ b).\ f\ a * g\ b * h\ c$ *when* $bc = b + c$ *when* $k = a + bc)$
by (*rule Sum-any.cong*) (*simp add*: *split-def when-def ac-simps*)
also have $\ldots = (\sum (a,\ bc,\ b,\ c).\ f\ a * g\ b * h\ c$ *when* $bc = b + c$ *when* $k = a + bc)$
proof −
have *bij* $(\lambda(a,\ d,\ b,\ c).\ (d,\ c,\ a,\ b))$
by (*auto intro*!: *bijI injI surjI* [*of* - $\lambda(d,\ c,\ a,\ b).\ (a,\ d,\ b,\ c)$] *simp add*: *split-def*)
then show *?thesis*
by (*rule Sum-any.reindex-cong*) *auto*
qed
also have $\ldots = (\sum (a,\ bc).\ (\sum (b,\ c).\ f\ a * g\ b * h\ c$ *when* $bc = b + c$ *when* $k = a + bc))$

```
      apply (subst Sum-any.cartesian-product2 [of (?F × ?GH') × ?GH])
     apply (auto simp add: finite-cartesian-product-iff fin-f fin-gh fin-gh' ac-simps
dest: mult-not-zero)
      done
    also have ... = (∑(a, bc). f a * (∑(b, c). g b * h c when bc = b + c) when
k = a + bc)
      apply (subst Sum-any-right-distrib)
      using fin-gh apply (simp add: split-def)
      apply (subst Sum-any-when-independent [symmetric])
      apply (simp add: when-when when-mult mult-when split-def ac-simps)
      done
    also from fin-f fin-g fin-h fin-gh''
    have ... = ?rhs k
      by (simp add: prod-fun-unfold-prod)
    finally show ?lhs k = ?rhs k .
  qed
 qed
 show (a + b) * c = a * c + b * c
 proof transfer
  fix f g h :: 'a ⇒ 'b
  assume fin-f: finite {k. f k ≠ 0}
  assume fin-g: finite {k. g k ≠ 0}
  assume fin-h: finite {k. h k ≠ 0}
  show prod-fun (λk. f k + g k) h = (λk. prod-fun f h k + prod-fun g h k)
   apply (rule ext)
   apply (auto simp add: prod-fun-def algebra-simps)
   apply (subst Sum-any.distrib)
   using fin-f fin-g apply (auto intro: finite-mult-not-eq-zero-rightI)
   done
 qed
 show a * (b + c) = a * b + a * c
 proof transfer
  fix f g h :: 'a ⇒ 'b
  assume fin-f: finite {k. f k ≠ 0}
  assume fin-g: finite {k. g k ≠ 0}
  assume fin-h: finite {k. h k ≠ 0}
  show prod-fun f (λk. g k + h k) = (λk. prod-fun f g k + prod-fun f h k)
   apply (rule ext)
  apply (auto simp add: prod-fun-def Sum-any.distrib algebra-simps when-add-distrib)
   apply (subst Sum-any.distrib)
   apply (simp-all add: algebra-simps)
   apply (auto intro: fin-g fin-h)
   apply (subst Sum-any.distrib)
   apply (simp-all add: algebra-simps)
   using fin-f apply (rule finite-mult-not-eq-zero-rightI)
   using fin-f apply (rule finite-mult-not-eq-zero-rightI)
   done
 qed
 show 0 * a = 0
```

```
    by transfer (simp add: prod-fun-def [abs-def])
  show a * 0 = 0
    by transfer (simp add: prod-fun-def [abs-def])
qed

end

lemma lookup-mult:
  lookup (f * g) k = (∑ l. lookup f l * (∑ q. lookup g q when k = l + q))
  by transfer (simp add: prod-fun-def)

instance poly-mapping :: (comm-monoid-add, comm-semiring-0) comm-semiring-0
proof
  fix a b c :: 'a ⇒₀ 'b
  show a * b = b * a
  proof transfer
    fix f g :: 'a ⇒ 'b
    assume fin-f: finite {a. f a ≠ 0}
    assume fin-g: finite {b. g b ≠ 0}
    show prod-fun f g = prod-fun g f
    proof
      fix k
      have fin1: ⋀l. finite {a. (f a when k = l + a) ≠ 0}
        using fin-f by auto
      have fin2: ⋀l. finite {b. (g b when k = l + b) ≠ 0}
        using fin-g by auto
      from fin-f fin-g have finite {(a, b). f a ≠ 0 ∧ g b ≠ 0} (is finite ?AB)
        by simp
      show prod-fun f g k = prod-fun g f k
        apply (simp add: prod-fun-def)
        apply (subst Sum-any-right-distrib)
        apply (rule fin2)
        apply (subst Sum-any-right-distrib)
        apply (rule fin1)
        apply (subst Sum-any.commute [of ?AB])
        apply (fact ‹finite ?AB›)
        apply (auto simp add: mult-when ac-simps)
        done
    qed
  qed
  show (a + b) * c = a * c + b * c
  proof transfer
    fix f g h :: 'a ⇒ 'b
    assume fin-f: finite {k. f k ≠ 0}
    assume fin-g: finite {k. g k ≠ 0}
    assume fin-h: finite {k. h k ≠ 0}
    show prod-fun (λk. f k + g k) h = (λk. prod-fun f h k + prod-fun g h k)
      apply (auto simp add: prod-fun-def fun-eq-iff algebra-simps)
      apply (subst Sum-any.distrib)
```

```
    using fin-f apply (rule finite-mult-not-eq-zero-rightI)
    using fin-g apply (rule finite-mult-not-eq-zero-rightI)
    apply simp-all
    done
  qed
qed

instance poly-mapping :: (monoid-add, semiring-0-cancel) semiring-0-cancel
  ..

instance poly-mapping :: (comm-monoid-add, comm-semiring-0-cancel) comm-semiring-0-cancel
  ..

instance poly-mapping :: (monoid-add, semiring-1) semiring-1
proof
  fix a :: 'a ⇒₀ 'b
  show 1 * a = a
    by transfer (simp add: prod-fun-def [abs-def] when-mult)
  show a * 1 = a
    apply transfer
   apply (simp add: prod-fun-def [abs-def] Sum-any-right-distrib Sum-any-left-distrib
mult-when)
    apply (subst when-commute)
    apply simp
    done
qed

instance poly-mapping :: (comm-monoid-add, comm-semiring-1) comm-semiring-1
proof
  fix a :: 'a ⇒₀ 'b
  show 1 * a = a
    by transfer (simp add: prod-fun-def [abs-def])
qed

instance poly-mapping :: (monoid-add, semiring-1-cancel) semiring-1-cancel
  ..

instance poly-mapping :: (monoid-add, ring) ring
  ..

instance poly-mapping :: (comm-monoid-add, comm-ring) comm-ring
  ..

instance poly-mapping :: (monoid-add, ring-1) ring-1
  ..

instance poly-mapping :: (comm-monoid-add, comm-ring-1) comm-ring-1
  ..
```

28.5 Single-point mappings

lift-definition *single* :: *'a* ⇒ *'b* ⇒ *'a* ⇒$_0$ *'b*::*zero*
is *λk v k'. (v when k = k')*
by *simp*

lemma *inj-single* [*iff*]:
inj (single k)
proof (*rule injI*, *transfer*)
fix *k* :: *'b* **and** *a b* :: *'a*::*zero*
assume *(λk'. a when k = k') = (λk'. b when k = k')*
then have *(λk'. a when k = k') k = (λk'. b when k = k') k*
by (*rule arg-cong*)
then show *a = b* **by** *simp*
qed

lemma *lookup-single*:
lookup (single k v) k' = (v when k = k')
by *transfer rule*

lemma *lookup-single-eq* [*simp*]:
lookup (single k v) k = v
by *transfer simp*

lemma *lookup-single-not-eq*:
k ≠ k' ⟹ lookup (single k v) k' = 0
by *transfer simp*

lemma *single-zero* [*simp*]:
single k 0 = 0
by *transfer simp*

lemma *single-one* [*simp*]:
single 0 1 = 1
by *transfer simp*

lemma *single-add*:
single k (a + b) = single k a + single k b
by *transfer* (*simp add*: *fun-eq-iff when-add-distrib*)

lemma *single-uminus*:
single k (− a) = − single k a
by *transfer* (*simp add*: *fun-eq-iff when-uminus-distrib*)

lemma *single-diff*:
single k (a − b) = single k a − single k b
by *transfer* (*simp add*: *fun-eq-iff when-diff-distrib*)

lemma *single-numeral* [*simp*]:
single 0 (numeral n) = numeral n

by (*induct n*) (*simp-all only*: *numeral.simps numeral-add single-zero single-one single-add*)

lemma *lookup-numeral*:
lookup (*numeral n*) *k* = (*numeral n when k* = *0*)
proof −
have *lookup* (*numeral n*) *k* = *lookup* (*single 0* (*numeral n*)) *k*
by *simp*
then show *?thesis* **unfolding** *lookup-single* **by** *simp*
qed

lemma *single-of-nat* [*simp*]:
single 0 (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *single-add*)

lemma *lookup-of-nat*:
lookup (*of-nat n*) *k* = (*of-nat n when k* = *0*)
proof −
have *lookup* (*of-nat n*) *k* = *lookup* (*single 0* (*of-nat n*)) *k*
by *simp*
then show *?thesis* **unfolding** *lookup-single* **by** *simp*
qed

lemma *of-nat-single*:
of-nat = *single 0* ∘ *of-nat*
by (*simp add*: *fun-eq-iff*)

lemma *mult-single*:
single k a ∗ *single l b* = *single* (*k* + *l*) (*a* ∗ *b*)
proof *transfer*
fix *k l* :: *'a* **and** *a b* :: *'b*
show *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) = (λ*k'*. *a* ∗ *b when k* + *l* = *k'*)
proof
fix *k'*
have *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) *k'* = (∑ *n*. *a* ∗ *b when l* = *n when k'* = *k* + *n*)
by (*simp add*: *prod-fun-def Sum-any-right-distrib mult-when when-mult*)
also have ... = (∑ *n*. *a* ∗ *b when k'* = *k* + *n when l* = *n*)
by (*simp add*: *when-when conj-commute*)
also have ... = (*a* ∗ *b when k'* = *k* + *l*)
by *simp*
also have ... = (*a* ∗ *b when k* + *l* = *k'*)
by (*simp add*: *when-def*)
finally show *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) *k'* =
(λ*k'*. *a* ∗ *b when k* + *l* = *k'*) *k'* **.**
qed
qed

instance *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*
by *intro-classes* (*auto intro*: *inj-comp inj-of-nat simp add*: *of-nat-single*)

instance *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*
..

lemma *single-of-int* [*simp*]:
single 0 (*of-int k*) = *of-int k*
by (*cases k*) (*simp-all add*: *single-diff single-uminus*)

lemma *lookup-of-int*:
lookup (*of-int l*) *k* = (*of-int l when k* = *0*)
proof −
have *lookup* (*of-int l*) *k* = *lookup* (*single 0* (*of-int l*)) *k*
by *simp*
then show *?thesis* **unfolding** *lookup-single* **by** *simp*
qed

28.6 Integral domains

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-no-zero-divisors*) *ring-no-zero-divisors*
— The *linorder* constraint is a pragmatic device for the proof maybe it can be dropped
proof
fix *f g* :: *'a* ⇒$_0$ *'b*
assume *f* ≠ *0* **and** *g* ≠ *0*
then show *f* ∗ *g* ≠ *0*
proof *transfer*
fix *f g* :: *'a* ⇒ *'b*
def *F* ≡ {*a*. *f a* ≠ *0*}
moreover def *G* ≡ {*a*. *g a* ≠ *0*}
ultimately have [*simp*]:
⋀*a*. *f a* ≠ *0* ⟷ *a* ∈ *F*
⋀*b*. *g b* ≠ *0* ⟷ *b* ∈ *G*
by *simp-all*
assume *finite* {*a*. *f a* ≠ *0*}
then have [*simp*]: *finite F*
by *simp*
assume *finite* {*a*. *g a* ≠ *0*}
then have [*simp*]: *finite G*
by *simp*
assume *f* ≠ (λ*a*. *0*)
then obtain *a* **where** *f a* ≠ *0*
by (*auto simp add*: *fun-eq-iff*)
assume *g* ≠ (λ*b*. *0*)
then obtain *b* **where** *g b* ≠ *0*
by (*auto simp add*: *fun-eq-iff*)
from ‹*f a* ≠ *0*› **and** ‹*g b* ≠ *0*› **have** *F* ≠ {} **and** *G* ≠ {}

```
    by auto
  note Max-F = ⟨finite F⟩ ⟨F ≠ {}⟩
  note Max-G = ⟨finite G⟩ ⟨G ≠ {}⟩
  from Max-F and Max-G have [simp]:
    Max F ∈ F
    Max G ∈ G
    by auto
  from Max-F Max-G have [dest!]:
    ⋀a. a ∈ F ⟹ a ≤ Max F
    ⋀b. b ∈ G ⟹ b ≤ Max G
    by auto
  def q ≡ Max F + Max G
  have (∑(a, b). f a * g b when q = a + b) =
    (∑(a, b). f a * g b when q = a + b when a ∈ F ∧ b ∈ G)
   by (rule Sum-any.cong) (auto simp add: split-def when-def q-def intro: ccontr)
  also have ... =
    (∑(a, b). f a * g b when (Max F, Max G) = (a, b))
  proof (rule Sum-any.cong)
    fix ab :: 'a × 'a
    obtain a b where [simp]: ab = (a, b)
      by (cases ab) simp-all
    have [dest!]:
      a ≤ Max F ⟹ Max F ≠ a ⟹ a < Max F
      b ≤ Max G ⟹ Max G ≠ b ⟹ b < Max G
      by auto
    show (case ab of (a, b) ⇒ f a * g b when q = a + b when a ∈ F ∧ b ∈ G) =
      (case ab of (a, b) ⇒ f a * g b when (Max F, Max G) = (a, b))
      by (auto simp add: split-def when-def q-def dest: add-strict-mono [of a Max
F b Max G])
  qed
  also have ... = (∑ ab. (case ab of (a, b) ⇒ f a * g b) when
    (Max F, Max G) = ab)
    unfolding split-def when-def by auto
  also have ... ≠ 0
    by simp
  finally have prod-fun f g q ≠ 0
    by (simp add: prod-fun-unfold-prod)
  then show prod-fun f g ≠ (λk. 0)
    by (auto simp add: fun-eq-iff)
 qed
qed

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, ring-1-no-zero-divisors)
ring-1-no-zero-divisors
 ..

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, idom) idom
 ..
```

28.7 Mapping order

instantiation *poly-mapping* :: (*linorder*, {*zero*, *linorder*}) *linorder*
begin

lift-definition *less-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ *bool*
is *less-fun*
.

lift-definition *less-eq-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ *bool*
is λ*f g*. *less-fun f g* ∨ *f* = *g*
.

instance proof (*rule class.Orderings.linorder.of-class.intro*)
show *class.linorder* (*less-eq* :: (- ⇒$_0$ -) ⇒ -) *less*
proof (*rule linorder-strictI*, *rule order-strictI*)
fix *f g h* :: '*a* ⇒$_0$ '*b*
show *f* ≤ *g* ⟷ *f* < *g* ∨ *f* = *g*
by *transfer* (*rule refl*)
show ¬ *f* < *f*
by *transfer* (*rule less-fun-irrefl*)
show *f* < *g* ∨ *f* = *g* ∨ *g* < *f*
proof *transfer*
fix *f g* :: '*a* ⇒ '*b*
assume *finite* {*k*. *f k* ≠ *0*} **and** *finite* {*k*. *g k* ≠ *0*}
then have *finite* ({*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*})
by *simp*
moreover have {*k*. *f k* ≠ *g k*} ⊆ {*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*}
by *auto*
ultimately have *finite* {*k*. *f k* ≠ *g k*}
by (*rule rev-finite-subset*)
then show *less-fun f g* ∨ *f* = *g* ∨ *less-fun g f*
by (*rule less-fun-trichotomy*)
qed
assume *f* < *g* **then show** ¬ *g* < *f*
by *transfer* (*rule less-fun-asym*)
note ‹*f* < *g*› **moreover assume** *g* < *h*
ultimately show *f* < *h*
by *transfer* (*rule less-fun-trans*)
qed
qed

end

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *linorder*}) *ordered-ab-semigroup-add*
proof (*intro-classes*, *transfer*)
fix *f g h* :: '*a* ⇒ '*b*
assume ∗: *less-fun f g* ∨ *f* = *g*
{ assume *less-fun f g*

then obtain k **where** $f\ k < g\ k$ $(\bigwedge k'.\ k' < k \Longrightarrow f\ k' = g\ k')$
by (*blast elim*!: *less-funE*)
then have $h\ k + f\ k < h\ k + g\ k$ $(\bigwedge k'.\ k' < k \Longrightarrow h\ k' + f\ k' = h\ k' + g\ k')$
by *simp-all*
then have *less-fun* $(\lambda k.\ h\ k + f\ k)$ $(\lambda k.\ h\ k + g\ k)$
by (*blast intro*: *less-funI*)
}
with $*$ **show** *less-fun* $(\lambda k.\ h\ k + f\ k)$ $(\lambda k.\ h\ k + g\ k) \vee (\lambda k.\ h\ k + f\ k) = (\lambda k.\ h\ k + g\ k)$
by (*auto simp add*: *fun-eq-iff*)
qed

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *linordered-cancel-ab-semigroup-add*
..

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-comm-monoid-add*
..

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-cancel-comm-monoid-add*
..

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
..

For pragmatism we leave out the final elements in the hierarchy: *linordered-ring*, *linordered-ring-strict*, *linordered-idom*; remember that the order instance is a mere technical device, not a deeper algebraic property.

28.8 Fundamental mapping notions

lift-definition *keys* :: $('a \Rightarrow_0 'b{::}zero) \Rightarrow 'a\ set$
is $\lambda f.\ \{k.\ f\ k \neq 0\}$ **.**

lift-definition *range* :: $('a \Rightarrow_0 'b{::}zero) \Rightarrow 'b\ set$
is $\lambda f :: 'a \Rightarrow 'b.\ Set.range\ f - \{0\}$ **.**

lemma *finite-keys* [*simp*]:
finite (*keys f*)
by *transfer*

lemma *not-in-keys-iff-lookup-eq-zero* [*simp*]:
$k \notin keys\ f \longleftrightarrow lookup\ f\ k = 0$
by *transfer simp*

lemma *lookup-not-eq-zero-eq-in-keys* [*simp*]:
$lookup\ f\ k \neq 0 \longleftrightarrow k \in keys\ f$

by *transfer simp*

lemma *lookup-eq-zero-in-keys-contradict* [*dest*]:
lookup f k = *0* ⟹ ¬ *k* ∈ *keys f*
by *simp*

lemma *finite-range* [*simp*]: *finite* (*PP-Poly-Mapping.range p*)
proof *transfer*
fix *f* :: *'b* ⇒ *'a*
assume ∗: *finite* {*x*. *f x* ≠ *0*}
have *Set.range f* − {*0*} ⊆ *f* ‘ {*x*. *f x* ≠ *0*}
by *auto*
thus *finite* (*Set.range f* − {*0*})
by(*rule finite-subset*)(*rule finite-imageI*[*OF* ∗])
qed

lemma *in-keys-lookup-in-range* [*simp*]:
k ∈ *keys f* ⟹ *lookup f k* ∈ *range f*
by *transfer simp*

lemma *keys-zero* [*simp*]:
keys 0 = {}
by *transfer simp*

lemma *range-zero* [*simp*]:
range 0 = {}
by *transfer auto*

lemma *keys-add-subset*:
keys (*f* + *g*) ⊆ *keys f* ∪ *keys g*
by *transfer auto*

lemma *keys-one* [*simp*]:
keys 1 = {*0*}
by *transfer simp*

lemma *range-one* [*simp*]:
range 1 = {*1*}
by *transfer* (*auto simp add*: *when-def*)

lemma *keys-single* [*simp*]:
keys (*single k v*) = (*if v* = *0 then* {} *else* {*k*})
by *transfer simp*

lemma *range-single* [*simp*]:
range (*single k v*) = (*if v* = *0 then* {} *else* {*v*})
by *transfer* (*auto simp add*: *when-def*)

lemma *keys-mult*:

keys (*f* ∗ *g*) ⊆ {*a* + *b* | *a b*. *a* ∈ *keys f* ∧ *b* ∈ *keys g*}
apply *transfer*
apply (*auto simp add*: *prod-fun-def dest*!: *mult-not-zero elim*!: *Sum-any.not-neutral-obtains-not-neutral*)
apply *blast*
done

lemma *setsum-keys-plus-distrib*:
assumes *hom-0*: ⋀*k*. *f k 0* = *0*
and *hom-plus*: ⋀*k*. *k* ∈ *PP-Poly-Mapping.keys p* ∪ *PP-Poly-Mapping.keys q* ⟹ *f k* (*PP-Poly-Mapping.lookup p k* + *PP-Poly-Mapping.lookup q k*) = *f k* (*PP-Poly-Mapping.lookup p k*) + *f k* (*PP-Poly-Mapping.lookup q k*)
shows
(∑ *k*∈*PP-Poly-Mapping.keys* (*p* + *q*). *f k* (*PP-Poly-Mapping.lookup* (*p* + *q*) *k*)) =
(∑ *k*∈*PP-Poly-Mapping.keys p*. *f k* (*PP-Poly-Mapping.lookup p k*)) +
(∑ *k*∈*PP-Poly-Mapping.keys q*. *f k* (*PP-Poly-Mapping.lookup q k*))
(**is** *?lhs* = *?p* + *?q*)
proof −
let *?A* = *PP-Poly-Mapping.keys p* ∪ *PP-Poly-Mapping.keys q*
have *?lhs* = (∑ *k*∈*?A*. *f k* (*PP-Poly-Mapping.lookup p k* + *PP-Poly-Mapping.lookup q k*))
apply(*rule sum.mono-neutral-cong-left*)
apply(*simp-all add*: *PP-Poly-Mapping.keys-add-subset*)
apply(*transfer fixing*: *f*)
apply(*auto simp add*: *hom-0*)[*1*]
apply(*transfer fixing*: *f*)
apply(*auto simp add*: *hom-0*)[*1*]
done
also have ... = (∑ *k*∈*?A*. *f k* (*PP-Poly-Mapping.lookup p k*) + *f k* (*PP-Poly-Mapping.lookup q k*))
by(*rule sum.cong*)(*simp-all add*: *hom-plus*)
also have ... = (∑ *k*∈*?A*. *f k* (*PP-Poly-Mapping.lookup p k*)) + (∑ *k*∈*?A*. *f k* (*PP-Poly-Mapping.lookup q k*))
(**is** - = *?p′* + *?q′*)
by(*simp add*: *sum.distrib*)
also have *?p′* = *?p*
by(*rule sum.mono-neutral-right*)(*auto simp add*: *hom-0*)
also have *?q′* = *?q*
by(*rule sum.mono-neutral-right*)(*auto simp add*: *hom-0*)
finally show *?thesis* **.**
qed

28.9 Degree

definition *degree* :: (*nat* ⇒$_0$ *′a*::*zero*) ⇒ *nat*
where
degree f = *Max* (*insert 0* (*Suc* ‘ *keys f*))

lemma *degree-zero* [*simp*]:

```
  degree 0 = 0
  unfolding degree-def by transfer simp

lemma degree-one [simp]:
  degree 1 = 1
  unfolding degree-def by transfer simp

lemma degree-single-zero [simp]:
  degree (single k 0) = 0
  unfolding degree-def by transfer simp

lemma degree-single-not-zero [simp]:
  v ≠ 0 ⟹ degree (single k v) = Suc k
  unfolding degree-def by transfer simp

lemma degree-zero-iff [simp]:
  degree f = 0 ⟷ f = 0
unfolding degree-def proof transfer
  fix f :: nat ⇒ 'a
  assume finite {n. f n ≠ 0}
  then have fin: finite (insert 0 (Suc ` {n. f n ≠ 0})) by auto
  show Max (insert 0 (Suc ` {n. f n ≠ 0})) = 0 ⟷ f = (λn. 0) (is ?P ⟷
?Q)
  proof
    assume ?P
    have {n. f n ≠ 0} = {}
    proof (rule ccontr)
      assume {n. f n ≠ 0} ≠ {}
      then obtain n where n ∈ {n. f n ≠ 0} by blast
      then have {n. f n ≠ 0} = insert n {n. f n ≠ 0} by auto
     then have Suc ` {n. f n ≠ 0} = insert (Suc n) (Suc ` {n. f n ≠ 0}) by auto
      with ‹?P› have Max (insert 0 (insert (Suc n) (Suc ` {n. f n ≠ 0}))) = 0
by simp
      then have Max (insert (Suc n) (insert 0 (Suc ` {n. f n ≠ 0}))) = 0
        by (simp add: insert-commute)
      with fin have max (Suc n) (Max (insert 0 (Suc ` {n. f n ≠ 0}))) = 0
        by simp
      then show False by simp
    qed
    then show ?Q by (simp add: fun-eq-iff)
  next
    assume ?Q then show ?P by simp
  qed
qed

lemma degree-greater-zero-in-keys:
  assumes 0 < degree f
  shows degree f − 1 ∈ keys f
proof −
```

```
  from assms have keys f ≠ {}
    by (auto simp add: degree-def)
  then show ?thesis unfolding degree-def
    by (simp add: mono-Max-commute [symmetric] mono-Suc)
qed

lemma in-keys-less-degree:
  n ∈ keys f ⟹ n < degree f
unfolding degree-def by transfer (auto simp add: Max-gr-iff)

lemma beyond-degree-lookup-zero:
  degree f ≤ n ⟹ lookup f n = 0
  unfolding degree-def by transfer auto

lemma degree-add:
  degree (f + g) ≤ max (degree f) (PP-Poly-Mapping.degree g)
unfolding degree-def proof transfer
  fix f g :: nat ⇒ 'a
  assume f: finite {x. f x ≠ 0}
  assume g: finite {x. g x ≠ 0}
  let ?f = Max (insert 0 (Suc ` {k. f k ≠ 0}))
  let ?g = Max (insert 0 (Suc ` {k. g k ≠ 0}))
  have Max (insert 0 (Suc ` {k. f k + g k ≠ 0})) ≤ Max (insert 0 (Suc ` ({k. f
k ≠ 0} ∪ {k. g k ≠ 0})))
    by (rule Max.antimono) (insert f g, auto)
  also have ... = max ?f ?g
    using f g by (simp-all add: image-Un Max-Un [symmetric])
  finally show Max (insert 0 (Suc ` {k. f k + g k ≠ 0}))
    ≤ max (Max (insert 0 (Suc ` {k. f k ≠ 0}))) (Max (insert 0 (Suc ` {k. g k ≠
0})))
    .
qed

lemma sorted-list-of-set-keys:
  sorted-list-of-set (keys f) = filter (λk. k ∈ keys f) [0..<degree f] (is - = ?r)
proof −
  have keys f = set ?r
    by (auto dest: in-keys-less-degree)
  moreover have sorted-list-of-set (set ?r) = ?r
    unfolding sorted-list-of-set-sort-remdups
    by (simp add: remdups-filter filter-sort [symmetric])
  ultimately show ?thesis by simp
qed
```

28.10 Inductive structure

```
lift-definition update :: 'a ⇒ 'b ⇒ ('a ⇒₀ 'b::zero) ⇒ 'a ⇒₀ 'b
  is λk v f. f(k := v)
proof −
```

```
  fix f :: 'a ⇒ 'b and k' v
  assume finite {k. f k ≠ 0}
  then have finite (insert k' {k. f k ≠ 0})
    by simp
  then show finite {k. (f(k' := v)) k ≠ 0}
    by (rule rev-finite-subset) auto
qed

lemma update-induct [case-names const update]:
  assumes const': P 0
  assumes update': ⋀f a b. a ∉ keys f ⟹ b ≠ 0 ⟹ P f ⟹ P (update a b f)
  shows P f
proof −
  obtain g where f = Abs-poly-mapping g and finite {a. g a ≠ 0}
    by (cases f) simp-all
  def Q-subst: Q ≡ λg. P (Abs-poly-mapping g)
  from ‹finite {a. g a ≠ 0}› have Q g
  proof (induct g rule: finite-update-induct)
    case const with const' Q-subst show ?case
      by (simp add: zero-poly-mapping.abs-eq)
  next
    case (update a b g)
    from ‹finite {a. g a ≠ 0}› ‹g a = 0› have a ∉ keys (Abs-poly-mapping g)
      by (simp add: Abs-poly-mapping-inverse keys.rep-eq)
    moreover note ‹b ≠ 0›
    moreover from ‹Q g› have P (Abs-poly-mapping g)
      by (simp add: Q-subst)
    ultimately have P (update a b (Abs-poly-mapping g))
      by (rule update')
    also from ‹finite {a. g a ≠ 0}›
    have update a b (Abs-poly-mapping g) = Abs-poly-mapping (g(a := b))
      by (simp add: update.abs-eq eq-onp-same-args)
    finally show ?case
      by (simp add: Q-subst fun-upd-def)
  qed
  then show ?thesis by (simp add: Q-subst ‹f = Abs-poly-mapping g›)
qed

lemma lookup-update:
  lookup (update k v f) k' = (if k = k' then v else lookup f k')
  by transfer simp

lemma keys-update:
  keys (update k v f) = (if v = 0 then keys f − {k} else insert k (keys f))
  by transfer auto
```

28.11 Quasi-functorial structure

```
lift-definition map :: ('b::zero ⇒ 'c::zero)
```

$\Rightarrow$ ('a $\Rightarrow_0$ 'b) $\Rightarrow$ ('a $\Rightarrow_0$ 'c::*zero*)
is $\lambda g\ f\ k.\ g\ (f\ k)$ *when* $f\ k \neq 0$
by *simp*

context
fixes f :: 'b $\Rightarrow$ 'a
assumes *inj-f*: *inj f*
begin

lift-definition *map-key* :: ('a $\Rightarrow_0$ 'c::*zero*) $\Rightarrow$ 'b $\Rightarrow_0$ 'c
is $\lambda p.\ p \circ f$
proof −
fix g :: 'c $\Rightarrow$ 'd **and** p :: 'a $\Rightarrow$ 'c
assume *finite* $\{x.\ p\ x \neq 0\}$
hence *finite* (f ‘ $\{y.\ p\ (f\ y) \neq 0\}$)
by(*rule finite-subset*[*rotated*]) *auto*
thus *finite* $\{x.\ (p \circ f)\ x \neq 0\}$ **unfolding** *o-def*
by(*rule finite-imageD*)(*rule subset-inj-on*[*OF inj-f*], *simp*)
qed

end

lemma *map-key-compose*:
assumes [*transfer-rule*]: *inj f inj g*
shows *map-key f* (*map-key g p*) = *map-key* ($g \circ f$) *p*
proof −
from *assms* **have** [*transfer-rule*]: *inj* ($g \circ f$)
by(*simp add*: *inj-comp*)
show *?thesis* **by** *transfer*(*simp add*: *o-assoc*)
qed

lemma *map-key-id*:
map-key ($\lambda x.\ x$) *p* = *p*
proof −
have [*transfer-rule*]: *inj* ($\lambda x.\ x$) **by** *simp*
show *?thesis* **by** *transfer*(*simp add*: *o-def*)
qed

context
fixes f :: 'a $\Rightarrow$ 'b
assumes *inj-f* [*transfer-rule*]: *inj f*
begin

lemma *map-key-map*:
map-key f (*map g p*) = *map g* (*map-key f p*)
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *map-key-plus*:
map-key f (*p* + *q*) = *map-key f p* + *map-key f q*

by *transfer* (*simp add*: *fun-eq-iff*)

lemma *keys-map-key*:
keys (*map-key f p*) = *f* −‘ *keys p*
by *transfer auto*

lemma *map-key-zero* [*simp*]:
map-key f 0 = *0*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *map-key-single* [*simp*]:
map-key f (*single* (*f k*) *v*) = *single k v*
by *transfer* (*simp add*: *fun-eq-iff inj-onD* [*OF inj-f*] *when-def*)

end

lemma *mult-map-scale-conv-mult*: *map* (*op* ∗ *s*) *p* = *single 0 s* ∗ *p*
proof(*transfer fixing*: *s*)
fix *p* :: ′*a* ⇒ ′*b*
assume ∗: *finite* {*x*. *p x* ≠ *0*}
{ fix *x*
have *prod-fun* (λ*k*′. *s when 0* = *k*′) *p x* =
(∑ *l* :: ′*a*. *if l* = *0 then s* ∗ (∑ *q*. *p q when x* = *q*) *else 0*)
by(*auto simp add*: *prod-fun-def when-def intro*: *Sum-any.cong simp del*: *Sum-any.delta*)
also have ... = (λ*k*. *s* ∗ *p k when p k* ≠ *0*) *x* **by**(*simp add*: *when-def*)
also note *calculation* **}**
then show (λ*k*. *s* ∗ *p k when p k* ≠ *0*) = *prod-fun* (λ*k*′. *s when 0* = *k*′) *p*
by(*simp add*: *fun-eq-iff*)
qed

lemma *map-single* [*simp*]:
(*c* = *0* ⟹ *f 0* = *0*) ⟹ *map f* (*single x c*) = *single x* (*f c*)
by *transfer*(*auto simp add*: *fun-eq-iff when-def*)

lemma *map-eq-zero-iff*: *map f p* = *0* ⟷ (∀ *k* ∈ *keys p*. *f* (*lookup p k*) = *0*)
by *transfer*(*auto simp add*: *fun-eq-iff when-def*)

28.12 Canonical dense representation of *nat* ⇒$_0$ ′*a*

abbreviation *no-trailing-zeros* :: ′*a* :: *zero list* ⇒ *bool*
where
no-trailing-zeros ≡ *no-trailing* (*op* = *0*)

lift-definition *nth* :: ′*a list* ⇒ (*nat* ⇒$_0$ ′*a*::*zero*)
is *nth-default 0*
by (*fact finite-nth-default-neq-default*)

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [*simp*]:

```
  nth [] = 0
  by transfer (simp add: fun-eq-iff)

lemma nth-singleton [simp]:
  nth [v] = single 0 v
proof (transfer, rule ext)
  fix n :: nat and v :: 'a
  show nth-default 0 [v] n = (v when 0 = n)
    by (auto simp add: nth-default-def nth-append)
qed

lemma nth-replicate [simp]:
  nth (replicate n 0 @ [v]) = single n v
proof (transfer, rule ext)
  fix m n :: nat and v :: 'a
  show nth-default 0 (replicate n 0 @ [v]) m = (v when n = m)
    by (auto simp add: nth-default-def nth-append)
qed

lemma nth-strip-while [simp]:
  nth (strip-while (op = 0) xs) = nth xs
  by transfer (fact nth-default-strip-while-dflt)

lemma nth-strip-while' [simp]:
  nth (strip-while (λk. k = 0) xs) = nth xs
  by (subst eq-commute) (fact nth-strip-while)

lemma nth-eq-iff:
  nth xs = nth ys ⟷ strip-while (HOL.eq 0) xs = strip-while (HOL.eq 0) ys
  by transfer (simp add: nth-default-eq-iff)

lemma lookup-nth [simp]:
  lookup (nth xs) = nth-default 0 xs
  by (fact nth.rep-eq)

lemma keys-nth [simp]:
  keys (nth xs) =  fst ‘ {(n, v) ∈ set (enumerate 0 xs). v ≠ 0}
proof transfer
  fix xs :: 'a list
  { fix n
    assume nth-default 0 xs n ≠ 0
    then have n < length xs and xs ! n ≠ 0
      by (auto simp add: nth-default-def split: if-splits)
    then have (n, xs ! n) ∈ {(n, v). (n, v) ∈ set (enumerate 0 xs) ∧ v ≠ 0} (is
?x ∈ ?A)
      by (auto simp add: in-set-conv-nth enumerate-eq-zip)
    then have fst ?x ∈ fst ‘ ?A
      by blast
    then have n ∈ fst ‘ {(n, v). (n, v) ∈ set (enumerate 0 xs) ∧ v ≠ 0}
```

```
    by simp
  }
  then show {k. nth-default 0 xs k ≠ 0} = fst ‘ {(n, v). (n, v) ∈ set (enumerate
0 xs) ∧ v ≠ 0}
    by (auto simp add: in-enumerate-iff-nth-default-eq)
qed

lemma range-nth [simp]:
  range (nth xs) = set xs − {0}
  by transfer simp

lemma degree-nth:
  no-trailing-zeros xs ⟹ degree (nth xs) = length xs
unfolding degree-def proof transfer
  fix xs :: 'a list
  assume ∗: no-trailing-zeros xs
  let ?A = {n. nth-default 0 xs n ≠ 0}
  let ?f = nth-default 0 xs
  let ?bound = Max (insert 0 (Suc ‘ {n. ?f n ≠ 0}))
  show ?bound = length xs
  proof (cases xs = [])
    case False
    with ∗ obtain n where n: n < length xs xs ! n ≠ 0
      by (fastforce simp add: no-trailing-unfold last-conv-nth neq-Nil-conv)
    then have ?bound = Max (Suc ‘ {k. (k < length xs ⟶ xs ! k ≠ (0::'a)) ∧ k
< length xs})
      by (subst Max-insert) (auto simp add: nth-default-def)
    also let ?A = {k. k < length xs ∧ xs ! k ≠ 0}
    have {k. (k < length xs ⟶ xs ! k ≠ (0::'a)) ∧ k < length xs} = ?A by auto
    also have Max (Suc ‘ ?A) = Suc (Max ?A) using n
      by (subst mono-Max-commute [where f = Suc, symmetric]) (auto simp add:
mono-Suc)
    also {
      have Max ?A ∈ ?A using n Max-in [of ?A] by fastforce
      hence Suc (Max ?A) ≤ length xs by simp
      moreover from ∗ False have length xs − 1 ∈ ?A
        by(auto simp add: no-trailing-unfold last-conv-nth)
      hence length xs − 1 ≤ Max ?A using Max-ge[of ?A length xs − 1] by auto
      hence length xs ≤ Suc (Max ?A) by simp
      ultimately have Suc (Max ?A) = length xs by simp }
    finally show ?thesis .
  qed simp
qed

lemma nth-trailing-zeros [simp]:
  nth (xs @ replicate n 0) = nth xs
  by transfer simp

lemma nth-idem:
```

nth (*List.map* (*lookup f*) [*0*..<*degree f*]) = *f*
unfolding *degree-def* **by** *transfer*
(*auto simp add*: *nth-default-def fun-eq-iff not-less*)

lemma *nth-idem-bound*:
assumes *degree f* ≤ *n*
shows *nth* (*List.map* (*lookup f*) [*0*..<*n*]) = *f*
proof −
from *assms* **obtain** *m* **where** *n* = *degree f* + *m*
by (*blast dest*: *le-Suc-ex*)
then have [*0*..<*n*] = [*0*..<*degree f*] @ [*degree f*..<*degree f* + *m*]
by (*simp add*: *upt-add-eq-append* [*of 0*])
moreover have *List.map* (*lookup f*) [*degree f*..<*degree f* + *m*] = *replicate m 0*
by (*rule replicate-eqI*) (*auto simp add*: *beyond-degree-lookup-zero*)
ultimately show *?thesis* **by** (*simp add*: *nth-idem*)
qed

28.13 Canonical sparse representation of *'a* ⇒$_0$ *'b*

lift-definition *the-value* :: (*'a* × *'b*) *list* ⇒ *'a* ⇒$_0$ *'b*::*zero*
is λ*xs k*. *case map-of xs k of None* ⇒ *0* | *Some v* ⇒ *v*
proof −
fix *xs* :: (*'a* × *'b*) *list*
have *fin*: *finite* {*k*. ∃ *v*. *map-of xs k* = *Some v*}
using *finite-dom-map-of* [*of xs*] **unfolding** *dom-def* **by** *auto*
then show *finite* {*k*. (*case map-of xs k of None* ⇒ *0* | *Some v* ⇒ *v*) ≠ *0*}
using *fin* **by** (*simp split*: *option.split*)
qed

definition *items* :: (*'a*::*linorder* ⇒$_0$ *'b*::*zero*) ⇒ (*'a* × *'b*) *list*
where
items f = *List.map* (λ*k*. (*k*, *lookup f k*)) (*sorted-list-of-set* (*keys f*))

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *OAList* will support arbitrary linear orders *linorder* as keys, not just natural numbers *nat*.

lemma *the-value-items* [*simp*]:
the-value (*items f*) = *f*
unfolding *items-def*
by *transfer* (*simp add*: *fun-eq-iff map-of-map-restrict restrict-map-def*)

lemma *lookup-the-value*:
lookup (*the-value xs*) *k* = (*case map-of xs k of None* ⇒ *0* | *Some v* ⇒ *v*)
by *transfer rule*

lemma *items-the-value*:
assumes *sorted* (*List.map fst xs*) **and** *distinct* (*List.map fst xs*) **and** *0* ∉ *snd* ‘ *set xs*

shows *items* (*the-value xs*) = *xs*
proof −
from *assms* **have** *sorted-list-of-set* (*set* (*List.map fst xs*)) = *List.map fst xs*
unfolding *sorted-list-of-set-sort-remdups* **by** (*simp add*: *distinct-remdups-id sorted-sort-id*)
moreover from *assms* **have** *keys* (*the-value xs*) = *fst* ‘ *set xs*
by *transfer* (*auto simp add*: *image-def split*: *option.split dest*: *set-map-of-compr*)
ultimately show *?thesis*
unfolding *items-def* **using** *assms*
by (*auto simp add*: *lookup-the-value intro*: *map-idI*)
qed

lemma *the-value-Nil* [*simp*]:
the-value [] = *0*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *the-value-Cons* [*simp*]:
the-value (*x* # *xs*) = *update* (*fst x*) (*snd x*) (*the-value xs*)
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *items-zero* [*simp*]:
items 0 = []
unfolding *items-def* **by** *simp*

lemma *items-one* [*simp*]:
items 1 = [(*0*, *1*)]
unfolding *items-def* **by** *transfer simp*

lemma *items-single* [*simp*]:
items (*single k v*) = (*if v* = *0 then* [] *else* [(*k*, *v*)])
unfolding *items-def* **by** *simp*

lemma *in-set-items-iff* [*simp*]:
(*k*, *v*) ∈ *set* (*items f*) ⟷ *k* ∈ *keys f* ∧ *lookup f k* = *v*
unfolding *items-def* **by** *transfer auto*

28.14 Size estimation

context
fixes *f* :: *'a* ⇒ *nat*
and *g* :: *'b* :: *zero* ⇒ *nat*
begin

definition *poly-mapping-size* :: (*'a* ⇒$_0$ *'b*) ⇒ *nat*
where
poly-mapping-size m = *g 0* + (∑ *k* ∈ *keys m*. *Suc* (*f k* + *g* (*lookup m k*)))

lemma *poly-mapping-size-0* [*simp*]:
poly-mapping-size 0 = *g 0*

by (*simp add*: *poly-mapping-size-def*)

lemma *poly-mapping-size-single* [*simp*]:
poly-mapping-size (*single k v*) = (*if v* = *0 then g 0 else g 0* + *f k* + *g v* + *1*)
unfolding *poly-mapping-size-def* **by** *transfer simp*

lemma *keys-less-poly-mapping-size*:
k ∈ *keys m* ⟹ *f k* + *g* (*lookup m k*) < *poly-mapping-size m*
unfolding *poly-mapping-size-def*
proof *transfer*
fix *k* :: *'a* **and** *m* :: *'a* ⇒ *'b* **and** *f* :: *'a* ⇒ *nat* **and** *g*
let *?keys* = {*k*. *m k* ≠ *0*}
assume ∗: *finite ?keys k* ∈ *?keys*
then have *f k* + *g* (*m k*) = (∑ *k'* ∈ *?keys*. *f k'* + *g* (*m k'*) *when k'* = *k*)
by (*simp add*: *sum.delta when-def*)
also have ... < (∑ *k'* ∈ *?keys*. *Suc* (*f k'* + *g* (*m k'*))) **using** ∗
by (*intro sum-strict-mono*) (*auto simp add*: *when-def*)
also have ... ≤ *g 0* + ... **by** *simp*
finally have *f k* + *g* (*m k*) < ... .
then show *f k* + *g* (*m k*) < *g 0* + (∑ *k* | *m k* ≠ *0*. *Suc* (*f k* + *g* (*m k*)))
by *simp*
qed

lemma *lookup-le-poly-mapping-size*:
g (*lookup m k*) ≤ *poly-mapping-size m*
proof (*cases k* ∈ *keys m*)
case *True*
with *keys-less-poly-mapping-size* [*of k m*]
show *?thesis* **by** *simp*
qed (*simp add*: *poly-mapping-size-def*)

lemma *poly-mapping-size-estimation*:
k ∈ *keys m* ⟹ *y* ≤ *f k* + *g* (*lookup m k*) ⟹ *y* < *poly-mapping-size m*
using *keys-less-poly-mapping-size* **by** (*auto intro*: *le-less-trans*)

lemma *poly-mapping-size-estimation2*:
assumes *v* ∈ *range m* **and** *y* ≤ *g v*
shows *y* < *poly-mapping-size m*
proof −
from *assms* **obtain** *k* **where** ∗: *lookup m k* = *v v* ≠ *0*
by *transfer blast*
from ∗ **have** *k* ∈ *keys m*
by *auto*
then show *?thesis*
proof (*rule poly-mapping-size-estimation*)
from *assms* ∗ **have** *y* ≤ *g* (*lookup m k*)
by *simp*
then show *y* ≤ *f k* + *g* (*lookup m k*)
by *simp*

qed
qed

end

lemma *poly-mapping-size-one* [*simp*]:
poly-mapping-size f g 1 = g 0 + f 0 + g 1 + 1
unfolding *poly-mapping-size-def* **by** *transfer simp*

lemma *poly-mapping-size-cong* [*fundef-cong*]:
$m = m' \Longrightarrow g\ 0 = g'\ 0 \Longrightarrow (\bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k = f'\ k)$
$\Longrightarrow (\bigwedge v.\ v \in range\ m' \Longrightarrow g\ v = g'\ v)$
$\Longrightarrow$ *poly-mapping-size f g m = poly-mapping-size f' g' m'*
by (*auto simp add*: *poly-mapping-size-def intro*!: *sum.cong*)

instantiation *poly-mapping* :: (*type*, *zero*) *size*
begin

definition *size* = *poly-mapping-size* (λ-. *0*) (λ-. *0*)

instance ..

end

28.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::
$('a \Rightarrow 'b :: zero \Rightarrow 'c :: zero) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c)$
is $\lambda f\ p\ k.$ (*if k* $\in$ *keys p then f k* (*lookup p k*) *else 0*)
by *simp*

lemma *mapp-cong* [*fundef-cong*]:
$[\![\ m = m';\ \bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k\ (lookup\ m'\ k) = f'\ k\ (lookup\ m'\ k)\ ]\!]$
$\Longrightarrow$ *mapp f m = mapp f' m'*
by *transfer* (*auto simp add*: *fun-eq-iff*)

lemma *lookup-mapp*:
lookup (*mapp f p*) *k* = (*f k* (*lookup p k*) *when k* $\in$ *keys p*)
unfolding *when-def* **by** *transfer simp*

lemma *keys-mapp-subset*: *keys* (*mapp f p*) $\subseteq$ *keys p*
by *transfer auto*

hide-const (**open**) *lookup single update keys range map map-key degree nth the-value items foldr mapp*

end

29 An abstract type for multivariate polynomials

theory *PP-MPoly*
imports *PP-Poly-Mapping*
begin

29.1 Abstract type definition

typedef (**overloaded**) *'a mpoly* =
  UNIV :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*zero*) *set*
  morphisms *mapping-of MPoly*
 ..

setup-lifting *type-definition-mpoly*

thm *mapping-of-inverse* **thm** *MPoly-inverse*
thm *mapping-of-inject* **thm** *MPoly-inject*
thm *mapping-of-induct* **thm** *MPoly-induct*
thm *mapping-of-cases* **thm** *MPoly-cases*

29.2 Additive structure

instantiation *mpoly* :: (*zero*) *zero*
begin

lift-definition *zero-mpoly* :: *'a mpoly*
  is *0* :: (*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a* .

instance ..

end

instantiation *mpoly* :: (*monoid-add*) *monoid-add*
begin

lift-definition *plus-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
  is *Groups.plus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - .

instance
  by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff add.assoc*)+

end

instance *mpoly* :: (*comm-monoid-add*) *comm-monoid-add*
  by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

instantiation *mpoly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-mpoly* :: *'a mpoly ⇒ 'a mpoly ⇒ 'a mpoly*
  is *Groups.minus* :: *((nat ⇒₀ nat) ⇒₀ 'a) ⇒ -* .

instance
  by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff diff-diff-add*)+

end

instantiation *mpoly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-mpoly* :: *'a mpoly ⇒ 'a mpoly*
  is *Groups.uminus* :: *((nat ⇒₀ nat) ⇒₀ 'a) ⇒ -* .

instance
  by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff add-uminus-conv-diff*)+

end

29.3 Multiplication by a coefficient

lift-definition *smult* :: *'a*::{*times*,*zero*} *⇒ 'a mpoly ⇒ 'a mpoly*
  is *λa. PP-Poly-Mapping.map* (*Groups.times a*) :: *((nat ⇒₀ nat) ⇒₀ 'a) ⇒ -* .

29.4 Multiplicative structure

instantiation *mpoly* :: (*zero-neq-one*) *zero-neq-one*
begin

lift-definition *one-mpoly* :: *'a mpoly*
  is *1* :: *((nat ⇒₀ nat) ⇒₀ 'a)* .

instance
  by *intro-classes* (*transfer*, *simp*)

end

instantiation *mpoly* :: (*semiring-0*) *semiring-0*
begin

lift-definition *times-mpoly* :: *'a mpoly ⇒ 'a mpoly ⇒ 'a mpoly*
  is *Groups.times* :: *((nat ⇒₀ nat) ⇒₀ 'a) ⇒ -* .

instance
  by *intro-classes* (*transfer*, *simp add*: *algebra-simps*)+

end

instance *mpoly* :: (*comm-semiring-0*) *comm-semiring-0*

by *intro-classes* (*transfer*, *simp add*: *algebra-simps*)+

instance *mpoly* :: (*semiring-0-cancel*) *semiring-0-cancel*
..

instance *mpoly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel*
..

instance *mpoly* :: (*semiring-1*) *semiring-1*
by *intro-classes* (*transfer*, *simp*)+

instance *mpoly* :: (*comm-semiring-1*) *comm-semiring-1*
by *intro-classes* (*transfer*, *simp*)+

instance *mpoly* :: (*semiring-1-cancel*) *semiring-1-cancel*
..

instance *mpoly* :: (*ring*) *ring*
..

instance *mpoly* :: (*comm-ring*) *comm-ring*
..

instance *mpoly* :: (*ring-1*) *ring-1*
..

instance *mpoly* :: (*comm-ring-1*) *comm-ring-1*
..

29.5 Monomials

Terminology is not unique here, so we use the notions as follows: A "monomial" and a "coefficient" together give a "term". These notions are significant in connection with "leading", "leading term", "leading coefficient" and "leading monomial", which all rely on a monomial order.

lift-definition *monom* :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow$ *'a*::*zero* $\Rightarrow$ *'a mpoly*
is *PP-Poly-Mapping.single* :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow$ - .

lemma *mapping-of-monom* [*simp*]:
mapping-of (*monom m a*) = *PP-Poly-Mapping.single m a*
by(*fact monom.rep-eq*)

lemma *monom-zero* [*simp*]:
monom 0 0 = *0*
by *transfer simp*

lemma *monom-one* [*simp*]:

monom 0 1 = 1
by *transfer simp*

lemma *monom-add*:
monom m (a + b) = monom m a + monom m b
by *transfer* (*simp add*: *single-add*)

lemma *monom-uminus*:
monom m (− a) = − monom m a
by *transfer* (*simp add*: *single-uminus*)

lemma *monom-diff*:
monom m (a − b) = monom m a − monom m b
by *transfer* (*simp add*: *single-diff*)

lemma *monom-numeral* [*simp*]:
monom 0 (numeral n) = numeral n
by (*induct n*) (*simp-all only*: *numeral.simps numeral-add monom-zero monom-one monom-add*)

lemma *monom-of-nat* [*simp*]:
monom 0 (of-nat n) = of-nat n
by (*induct n*) (*simp-all add*: *monom-add*)

lemma *of-nat-monom*:
of-nat = monom 0 ∘ of-nat
by (*simp add*: *fun-eq-iff*)

lemma *inj-monom* [*iff*]:
inj (*monom m*)
proof (*rule injI*, *transfer*)
fix *a b* :: *'a* **and** *m* :: *nat* ⇒$_0$ *nat*
assume *PP-Poly-Mapping.single m a = PP-Poly-Mapping.single m b*
with *injD* [*of PP-Poly-Mapping.single m a b*]
show *a = b* **by** *simp*
qed

lemma *mult-monom*: *monom x a ∗ monom y b = monom (x + y) (a ∗ b)*
by *transfer* (*simp add*: *PP-Poly-Mapping.mult-single*)
— FIXME: why does transfer need so much backtracking until it finds the right goal?

instance *mpoly* :: (*semiring-char-0*) *semiring-char-0*
by *intro-classes* (*auto simp add*: *of-nat-monom inj-of-nat intro*: *inj-comp*)

instance *mpoly* :: (*ring-char-0*) *ring-char-0*
..

lemma *monom-of-int* [*simp*]:

monom 0 (of-int k) = of-int k
apply (*cases k*)
apply *simp-all*
unfolding *monom-diff monom-uminus*
apply *simp*
done

29.6 Integral domains

instance *mpoly* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
by *intro-classes* (*transfer*, *simp*)

instance *mpoly* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors*
..

instance *mpoly* :: (*idom*) *idom*
..

29.7 Monom coefficient lookup

definition *coeff* :: *'a*::*zero mpoly* ⇒ (*nat* ⇒$_0$ *nat*) ⇒ *'a*
where
coeff p = PP-Poly-Mapping.lookup (*mapping-of p*)

29.8 Insertion morphism

definition *insertion-fun-natural* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒ *nat*) ⇒ *'a*) ⇒ *'a*::*comm-semiring-1*
where
insertion-fun-natural f p = (∑ *m*. *p m* ∗ (∏ *v*. *f v* ^ *m v*))

definition *insertion-fun* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒$_0$ *nat*) ⇒ *'a*) ⇒ *'a*::*comm-semiring-1*
where
insertion-fun f p = (∑ *m*. *p m* ∗ (∏ *v*. *f v* ^ *PP-Poly-Mapping.lookup m v*))

N.b. have been unable to relate this to *insertion-fun-natural* using lifting!

lift-definition *insertion-aux* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ *'a*::*comm-semiring-1*
is *insertion-fun* **.**

lift-definition *insertion* :: (*nat* ⇒ *'a*) ⇒ *'a mpoly* ⇒ *'a*::*comm-semiring-1*
is *insertion-aux* **.**

lemma *aux*:
PP-Poly-Mapping.lookup f = (λ-. *0*) ⟷ *f* = *0*
apply *transfer* **apply** *simp* **done**

lemma *insertion-trivial* [*simp*]:
insertion (λ-. *0*) *p* = *coeff p 0*
proof −
{ fix *f* :: (*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*
have *insertion-aux* (λ-. *0*) *f* = *PP-Poly-Mapping.lookup f 0*

```
    apply (simp add: insertion-aux-def insertion-fun-def power-Sum-any [symmetric])
     apply (simp add: zero-power-eq mult-when aux)
     done
  }
  then show ?thesis by (simp add: coeff-def insertion-def)
qed

lemma insertion-zero [simp]:
  insertion f 0 = 0
  by transfer (simp add: insertion-aux-def insertion-fun-def)

lemma insertion-fun-add:
  fixes f p q
  shows insertion-fun f (PP-Poly-Mapping.lookup (p + q)) =
    insertion-fun f (PP-Poly-Mapping.lookup p) +
      insertion-fun f (PP-Poly-Mapping.lookup q)
  unfolding insertion-fun-def
  apply (subst Sum-any.distrib [symmetric])
  apply (simp-all add: plus-poly-mapping.rep-eq algebra-simps)
  apply (rule finite-mult-not-eq-zero-rightI)
  apply simp
  apply (rule finite-mult-not-eq-zero-rightI)
  apply simp
  done

lemma insertion-add:
  insertion f (p + q) = insertion f p + insertion f q
  by transfer (simp add: insertion-aux-def insertion-fun-add)

lemma insertion-one [simp]:
  insertion f 1 = 1
 by transfer (simp add: insertion-aux-def insertion-fun-def one-poly-mapping.rep-eq
when-mult)

lemma insertion-fun-mult:
  fixes f p q
  shows insertion-fun f (PP-Poly-Mapping.lookup (p * q)) =
    insertion-fun f (PP-Poly-Mapping.lookup p) *
      insertion-fun f (PP-Poly-Mapping.lookup q)
proof -
  { fix m :: nat ⇒₀ nat
    have finite {v. PP-Poly-Mapping.lookup m v ≠ 0}
      by simp
    then have finite {v. f v ^ PP-Poly-Mapping.lookup m v ≠ 1}
      by (rule rev-finite-subset) (auto intro: ccontr)
  }
  moreover def g ≡ λm. (∏ v. f v ^ PP-Poly-Mapping.lookup m v)
  ultimately have *: ⋀a b. g (a + b) = g a * g b
    by (simp add: plus-poly-mapping.rep-eq power-add Prod-any.distrib)
```

have *bij*: *bij* $(\lambda(l,\ n,\ m).\ (m,\ l,\ n))$
by (*auto intro*!: *bijI injI simp add*: *image-def*)
let *?P* = $\{l.$ *PP-Poly-Mapping.lookup p l* $\neq$ *0*$\}$
let *?Q* = $\{n.$ *PP-Poly-Mapping.lookup q n* $\neq$ *0*$\}$
let *?PQ* = $\{l + n \mid l\ n.$ *l* $\in$ *PP-Poly-Mapping.keys p* $\wedge$ *n* $\in$ *PP-Poly-Mapping.keys q*$\}$
have *finite* $\{l + n \mid l\ n.$ *PP-Poly-Mapping.lookup p l* $\neq$ *0* $\wedge$ *PP-Poly-Mapping.lookup q n* $\neq$ *0*$\}$
by (*rule finite-not-eq-zero-sumI*) *simp-all*
then have *fin-PQ*: *finite ?PQ*
by *simp*
have $(\sum m.$ *PP-Poly-Mapping.lookup* $(p * q)$ $m * g\ m) =$
$(\sum m.\ (\sum l.$ *PP-Poly-Mapping.lookup p l* $*\ (\sum n.$ *PP-Poly-Mapping.lookup q n when m* $= l + n)) * g\ m)$
by (*simp add*: *times-poly-mapping.rep-eq prod-fun-def*)
also have $\ldots = (\sum m.\ (\sum l.\ (\sum n.\ g\ m * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$)$ *when m* $= l + n)))$
apply (*subst Sum-any-left-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-rightI*)
apply (*subst Sum-any-right-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-rightI*)
apply (*subst Sum-any-left-distrib*)
apply (*auto intro*: *finite-mult-not-eq-zero-leftI*)
apply (*simp add*: *ac-simps mult-when*)
done
also have $\ldots = (\sum m.\ (\sum (l,\ n).\ g\ m * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$)$ *when m* $= l + n))$
apply (*subst* (*2*) *Sum-any.cartesian-product* [*of ?P* $\times$ *?Q*])
apply (*auto dest*!: *mult-not-zero*)
done
also have $\ldots = (\sum (m,\ l,\ n).\ g\ m * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$)$ *when m* $= l + n)$
apply (*subst Sum-any.cartesian-product* [*of ?PQ* $\times$ (*?P* $\times$ *?Q*)])
apply (*auto dest*!: *mult-not-zero simp add*: *fin-PQ*)
apply *auto*
done
also have $\ldots = (\sum (l,\ n,\ m).\ g\ m * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$)$ *when m* $= l + n)$
using *bij* **by** (*rule Sum-any.reindex-cong* [*of* $\lambda(l,\ n,\ m).\ (m,\ l,\ n)$]) (*simp add*: *fun-eq-iff*)
also have $\ldots = (\sum (l,\ n).\ \sum m.\ g\ m * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$)$ *when m* $= l + n)$
apply (*subst Sum-any.cartesian-product2* [*of* (*?P* $\times$ *?Q*) $\times$ *?PQ*])
apply (*auto dest*!: *mult-not-zero simp add*: *fin-PQ*)
apply *auto*
done
also have $\ldots = (\sum (l,\ n).\ (g\ l * g\ n) * ($*PP-Poly-Mapping.lookup p l* $*$ *PP-Poly-Mapping.lookup q n*$))$
by (*simp add*: $*$)

also have ... = $(\sum l.\ \sum n.\ (g\ l * g\ n) * (PP\text{-}Poly\text{-}Mapping.lookup\ p\ l * PP\text{-}Poly\text{-}Mapping.lookup\ q\ n))$
apply (*subst Sum-any.cartesian-product* [*of ?P* × *?Q*])
apply (*auto dest*!: *mult-not-zero*)
done
also have ... = $(\sum l.\ \sum n.\ (PP\text{-}Poly\text{-}Mapping.lookup\ p\ l * g\ l) * (PP\text{-}Poly\text{-}Mapping.lookup\ q\ n * g\ n))$
by (*simp add*: *ac-simps*)
also have ... =
$(\sum m.\ PP\text{-}Poly\text{-}Mapping.lookup\ p\ m * g\ m) *$
$(\sum m.\ PP\text{-}Poly\text{-}Mapping.lookup\ q\ m * g\ m)$
by (*rule Sum-any-product* [*symmetric*]) (*auto intro*: *finite-mult-not-eq-zero-rightI*)
finally show *?thesis* **by** (*simp add*: *insertion-fun-def g-def*)
qed

lemma *insertion-mult*:
insertion f $(p * q)$ = *insertion f p * insertion f q*
by *transfer* (*simp add*: *insertion-aux-def insertion-fun-mult*)

29.9 Degree

lift-definition *degree* :: *'a::zero mpoly* ⇒ *nat* ⇒ *nat*
is λ*p v*. *Max* (*insert 0* ((λ*m*. *PP-Poly-Mapping.lookup m v*) ‘ *PP-Poly-Mapping.keys p*)) .

lift-definition *total-degree* :: *'a::zero mpoly* ⇒ *nat*
is λ*p*. *Max* (*insert 0* ((λ*m*. *sum* (*PP-Poly-Mapping.lookup m*) (*PP-Poly-Mapping.keys m*)) ‘ *PP-Poly-Mapping.keys p*)) .

lemma *degree-zero* [*simp*]:
degree 0 v = *0*
by *transfer simp*

lemma *total-degree-zero* [*simp*]:
total-degree 0 = *0*
by *transfer simp*

lemma *degree-one* [*simp*]:
degree 1 v = *0*
by *transfer simp*

lemma *total-degree-one* [*simp*]:
total-degree 1 = *0*
by *transfer simp*

29.10 Pseudo-division of polynomials

lemma *smult-conv-mult*: *smult s p* = *monom 0 s * p*

by *transfer* (*simp add*: *mult-map-scale-conv-mult*)

lemma *smult-monom* [*simp*]:
fixes *c* :: - :: *mult-zero*
shows *smult c* (*monom x c'*) = *monom x* (*c* ∗ *c'*)
by *transfer simp*

lemma *smult-0* [*simp*]:
fixes *p* :: - :: *mult-zero mpoly*
shows *smult 0 p* = *0*
by *transfer*(*simp add*: *map-eq-zero-iff*)

lemma *mult-smult-left*: *smult s p* ∗ *q* = *smult s* (*p* ∗ *q*)
by(*simp add*: *smult-conv-mult mult.assoc*)

lift-definition *sdiv* :: *'a*::*ring-div* ⇒ *'a mpoly* ⇒ *'a mpoly*
is λ*a*. *PP-Poly-Mapping.map* (λ*b*. *b div a*) :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ -
.

'Polynomial division' is only possible on univariate polynomials $K[x]$ over a field K, all other kinds of polynomials only allow pseudo-division [1]p.40/41":

∀ *x y* :: *'a mpoly*. *y* ≠ *0* ⇒ ∃ *a q r*. *smult a x* = *q* ∗ *y* + *r*

The introduction of pseudo-division below generalises `~~/src/HOL/Computational_Algebra/Polynomial.thy`. [1] Winkler, Polynomial Algorithms, 1996. The generalisation raises issues addressed by Wenda Li and commented below. Florian replied to the issues conjecturing, that the abstract mpoly needs not be aware of the issues, in case these are only concerned with executability.

definition *pseudo-divmod-rel*
:: *'a*::*ring-div* => *'a mpoly* => *'a mpoly* => *'a mpoly* => *'a mpoly* => *bool*
where
pseudo-divmod-rel a x y q r ⟷
smult a x = *q* ∗ *y* + *r* ∧ (*if y* = *0 then q* = *0 else r* = *0* ∨ *degree r* < *degree y*)

definition *pdiv* :: *'a*::*ring-div mpoly* ⇒ *'a mpoly* ⇒ (*'a* × *'a mpoly*) (**infixl** *pdiv 70*)
where
x pdiv y = (*THE* (*a*, *q*). ∃ *r*. *pseudo-divmod-rel a x y q r*)

definition *pmod* :: *'a*::*ring-div mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *pmod 70*)
where
x pmod y = (*THE r*. ∃ *a q*. *pseudo-divmod-rel a x y q r*)

definition *pdivmod* :: *'a*::*ring-div mpoly* ⇒ *'a mpoly* ⇒ (*'a* × *'a mpoly*) × *'a mpoly*
where
pdivmod p q = (*p pdiv q*, *p pmod q*)

lemma *pdiv-code*:
p pdiv q = fst (pdivmod p q)
by (*simp add*: *pdivmod-def*)

lemma *pmod-code*:
p pmod q = snd (pdivmod p q)
by (*simp add*: *pdivmod-def*)

definition *div* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *div 70*)
where
x div y = (*THE q'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*q'* = *smult* (*inverse a*) *q*))

definition *mod* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *mod 70*)
where
x mod y = (*THE r'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*r'* = *smult* (*inverse a*) *r*))

definition *divmod* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* × *'a mpoly*
where
divmod p q = (*p div q*, *p mod q*)

lemma *div-poly-code*:
p div q = fst (divmod p q)
by (*simp add*: *divmod-def*)

lemma *mod-poly-code*:
p mod q = snd (divmod p q)
by (*simp add*: *divmod-def*)

29.11 Primitive poly, etc

lift-definition *coeffs* :: *'a* :: *zero mpoly* ⇒ *'a set*
is *PP-Poly-Mapping.range* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - .

lemma *finite-coeffs* [*simp*]: *finite* (*coeffs p*)
by *transfer simp*

[1]p.82 A "primitive"' polynomial has coefficients with GCD equal to 1. A polynomial is factored into "content" and "primitive part" for many different purposes.

definition *primitive* :: *'a*::{*ring-div*,*semiring-Gcd*} *mpoly* ⇒ *bool*
where
primitive p ⟷ *Gcd* (*coeffs p*) = *1*

definition *content-primitive* :: *'a*::{*ring-div*,*GCD.Gcd*} *mpoly* ⇒ *'a* × *'a mpoly*
where
content-primitive p = (
let d = *Gcd* (*coeffs p*)
in (*d*, *sdiv d p*))

value *let p* = *M* [*1*,*2*,*3*] (*4*::*int*) + *M* [*2*,*0*,*4*] *6* + *M* [*2*,*0*,*5*] *8*
in content-primitive p

end

theory *PP-More-MPoly*
imports *PP-MPoly*
begin

abbreviation *lookup* == *PP-Poly-Mapping.lookup*
abbreviation *keys* == *PP-Poly-Mapping.keys*

30 MPpoly Mapping extenion

lemma *lookup-Abs-poly-mapping-when-finite*:
assumes *finite S*
shows *lookup* (*Abs-poly-mapping* (λ*x*. *f x when x*∈*S*)) = (λ*x*. *f x when x*∈*S*)
proof −
have *finite* {*x*. (*f x when x*∈*S*) ≠ *0*} **using** *assms* **by** *auto*
then show *?thesis* **using** *lookup-Abs-poly-mapping* **by** *fast*
qed

definition *remove-key*::*'a* ⇒ (*'a* ⇒$_0$ *'b*::*monoid-add*) ⇒ (*'a* ⇒$_0$ *'b*) **where**
remove-key k0 f = *Abs-poly-mapping* (λ*k*. *lookup f k when k* ≠ *k0*)

lemma *remove-key-lookup*:
lookup (*remove-key k0 f*) *k* = (*lookup f k when k* ≠ *k0*)
unfolding *remove-key-def* **using** *finite-subset* **by** (*simp add*: *lookup-Abs-poly-mapping*)

lemma *remove-key-keys*: *keys f* − {*k*} = *keys* (*remove-key k f*) (**is** *?A* = *?B*)
proof (*rule antisym*; *rule subsetI*)
fix *x* **assume** *x* ∈ *?A*
then show *x* ∈ *?B* **using** *remove-key-lookup lookup-not-eq-zero-eq-in-keys DiffD1 DiffD2 insertCI*
by (*metis* (*mono-tags*, *lifting*) *when-def*)
next
fix *x* **assume** *x* ∈ *?B*
then have *lookup* (*remove-key k f*) *x* ≠ *0* **by** *blast*
then show *x* ∈ *?A* **using** *remove-key-lookup lookup-not-eq-zero-eq-in-keys*
by (*simp add*: *remove-key-lookup*)
qed

lemma *remove-key-sum*: *remove-key k f + PP-Poly-Mapping.single k (lookup f k) = f*
proof −
{
fix *k′*
have *rem*:(*lookup f k′ when k′ ≠ k*) = *lookup* (*remove-key k f*) *k′*
using *when-def* **by** (*simp add*: *remove-key-lookup*)
have *sin*:(*lookup f k when k′=k*) = *lookup* (*PP-Poly-Mapping.single k* (*lookup f k*)) *k′*
by (*simp add*: *lookup-single-not-eq when-def*)
have *lookup f k′* = (*lookup f k′ when k′ ≠ k*) + ((*lookup f k*) *when k′=k*)
unfolding *when-def* **by** *fastforce*
with *rem sin* **have** *lookup f k′* = *lookup* ((*remove-key k f*) + *PP-Poly-Mapping.single k* (*lookup f k*)) *k′*
using *lookup-add* **by** *metis*
}
then show *?thesis* **by** (*metis poly-mapping-eqI*)
qed

lemma *remove-key-single*[*simp*]: *remove-key v* (*PP-Poly-Mapping.single v n*) = *0*
proof −
have *0*:⋀*k*. (*lookup* (*PP-Poly-Mapping.single v n*) *k when k ≠ v*) = *0* **by** (*simp add*: *lookup-single-not-eq when-def*)
show *?thesis* **unfolding** *remove-key-def 0* **by** (*simp add*: *zero-poly-mapping-def*)
qed

lemma *remove-key-add*: *remove-key v m + remove-key v m′* = *remove-key v* (*m + m′*)
by (*rule poly-mapping-eqI*; *simp add*: *lookup-add remove-key-lookup when-add-distrib*)

lemma *poly-mapping-induct* [*case-names single sum*]:
fixes *P*::(*′a*, *′b*::*monoid-add*) *poly-mapping* ⇒ *bool*
assumes *single*:⋀*k v*. *P* (*PP-Poly-Mapping.single k v*)
and *sum*:(⋀*f g k v*. *P f* ⟹ *P g* ⟹ *g* = (*PP-Poly-Mapping.single k v*) ⟹ *k* ∉ *keys f* ⟹ *P* (*f*+*g*))
shows *P f* **using** *finite-keys*[*of f*]
proof (*induction keys f arbitrary*: *f rule*: *finite-induct*)
case (*empty*)
then show *?case* **using** *single*[*of - 0*] **by** (*metis* (*full-types*) *aux empty-iff not-in-keys-iff-lookup-eq-zero single-zero*)
next
case (*insert k K f*)
obtain *f1 f2* **where** *f12-def*: *f1* = *remove-key k f f2* = *PP-Poly-Mapping.single k* (*lookup f k*) **by** *blast*
have *P f1*
proof −
have *Suc* (*card* (*keys f1*)) = *card* (*keys f*)

using *remove-key-keys finite-keys f12-def*(*1*) **by** (*metis* (*no-types*) *Diff-insert-absorb card-insert-disjoint insert.hyps*(*2*) *insert.hyps*(*4*))
then show *?thesis* **using** *insert lessI* **by** (*metis Diff-insert-absorb f12-def*(*1*) *remove-key-keys*)
qed
have *P f2* **by** (*simp add*: *single f12-def*(*2*))
have *f1 + f2 = f* **using** *remove-key-sum f12-def* **by** *auto*
have *k ∉ keys f1* **using** *remove-key-keys f12-def* **by** *fast*
then show *?case* **using** ‹*P f1*› ‹*P f2*› *sum*[*of f1 f2 k lookup f k*] ‹*f1 + f2 = f*› *f12-def* **by** *auto*
qed

lemma *map-lookup*:
assumes *g 0 = 0*
shows *lookup* (*PP-Poly-Mapping.map g f*) *x* = *g* ((*lookup f*) *x*)
proof −
have (*g* (*lookup f x*) *when lookup f x* ≠ *0*) = *g* (*lookup f x*)
by (*metis* (*mono-tags*, *lifting*) *assms when-def*)
then have (*g* (*lookup f x*) *when x* ∈ *keys f*) = *g* (*lookup f x*)
using *lookup-not-eq-zero-eq-in-keys* **by** *simp*
then show *?thesis* **unfolding** *PP-Poly-Mapping.map-def map-fun-def*
by (*simp add*:*lookup-Abs-poly-mapping*)
qed

lemma *keys-add*:
assumes *keys f* ∩ *keys g* = {}
shows *keys f* ∪ *keys g* = *keys* (*f*+*g*)
proof
have *keys f* ⊆ *keys* (*f*+*g*)
proof
fix *x* **assume** *x*∈*keys f*
then have *lookup* (*f*+*g*) *x* = *lookup f x* **by** (*metis add.right-neutral assms disjoint-iff-not-equal not-in-keys-iff-lookup-eq-zero plus-poly-mapping.rep-eq*)
then show *x*∈*keys* (*f*+*g*) **using** ‹*x*∈*keys f*› **by** (*metis not-in-keys-iff-lookup-eq-zero*)
qed
moreover have *keys g* ⊆ *keys* (*f*+*g*)
proof
fix *x* **assume** *x*∈*keys g*
then have *lookup* (*f*+*g*) *x* = *lookup g x* **by** (*metis IntI add.left-neutral assms empty-iff not-in-keys-iff-lookup-eq-zero plus-poly-mapping.rep-eq*)
then show *x*∈*keys* (*f*+*g*) **using** ‹*x*∈*keys g*› **by** (*metis not-in-keys-iff-lookup-eq-zero*)
qed
ultimately show *keys f* ∪ *keys g* ⊆ *keys* (*f*+*g*) **by** *simp*
next
show *keys* (*f* + *g*) ⊆ *keys f* ∪ *keys g* **by** (*simp add*: *keys-add-subset*)
qed

lemma *fun-when*:

$f\ 0 = 0 \Longrightarrow f\ (a\ when\ P) = (f\ a\ when\ P)$ **by** (*simp add*: *when-def*)

31 MPoly extension

lemma *coeff-all-0*:$(\bigwedge m.\ coeff\ p\ m = 0) \Longrightarrow p=0$
by (*metis aux coeff-def mapping-of-inject zero-mpoly.rep-eq*)

definition *vars*::*'a*::*zero mpoly* ⇒ *nat set* **where**
vars p = *UNION* (*keys* (*mapping-of p*)) (λ*m*. *keys m*)

lemma *vars-finite*: *finite* (*vars p*) **unfolding** *vars-def* **by** *auto*

lemma *vars-monom-single*: *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*) ⊆ {*v*}
proof
fix *w* **assume** *w* ∈ *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*)
then have *w* = *v* **using** *vars-def* **by** (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq monom.rep-eq*)
then show *w* ∈ {*v*} **by** *auto*
qed

lemma *vars-monom-keys*:
assumes *a*≠*0*
shows *vars* (*monom m a*) = *keys m*
proof (*rule antisym*; *rule subsetI*)
fix *w* **assume** *w* ∈ *vars* (*monom m a*)
then have *lookup m w* ≠ *0* **using** *vars-def* **by** (*metis UN-E lookup-eq-zero-in-keys-contradict lookup-single-not-eq monom.rep-eq*)
then show *w* ∈ *keys m* **by** (*meson lookup-not-eq-zero-eq-in-keys*)
next
fix *w* **assume** *w* ∈ *keys m*
then have *lookup m w* ≠ *0* **by** (*meson lookup-not-eq-zero-eq-in-keys*)
then show *w* ∈ *vars* (*monom m a*) **unfolding** *vars-def* **using** *assms* **by** (*metis UN-iff lookup-not-eq-zero-eq-in-keys lookup-single-eq monom.rep-eq*)
qed

lemma *vars-monom-subset*:
shows *vars* (*monom m a*) ⊆ *keys m*
by (*cases a=0*; *simp add*: *vars-def vars-monom-keys*)

lemma *vars-monom-single-cases*: *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*)
= (*if k=0* ∨ *a=0 then* {} *else* {*v*})
proof(*cases k=0*)
assume *k=0*
then have (*PP-Poly-Mapping.single v k*) = *0* **by** *simp*
then have *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*) = {}
by (*metis* (*mono-tags*, *lifting*) *single-zero singleton-inject subset-singletonD vars-monom-single zero-neq-one*)
then show *?thesis* **using** ‹*k=0*› **by** *auto*
next

assume *k≠0*
then show *?thesis*
proof (*cases a=0*)
assume *a=0*
then have *monom* (*PP-Poly-Mapping.single v k*) *a = 0* **by** (*metis monom.abs-eq monom-zero single-zero*)
then show *?thesis* **by** (*metis* (*mono-tags*, *hide-lams*) ‹*k ≠ 0*› ‹*a=0*› *monom.abs-eq single-zero singleton-inject subset-singletonD vars-monom-single*)
next
assume *a≠0*
then have *v ∈ vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*) **by** (*simp add*: ‹*k ≠ 0*› *vars-def*)
then show *?thesis* **using** ‹*a≠0*› ‹*k ≠ 0*› *vars-monom-single* **by** *fastforce*
qed
qed

lemma *vars-monom*:
assumes *a≠0*
shows *vars* (*monom m* (*1*::*'a*::*zero-neq-one*)) = *vars* (*monom m* (*a*::*'a*))
unfolding *vars-monom-keys*[*OF assms*] **using** *vars-monom-keys*[*of 1*] *one-neq-zero* **by** *blast*

lemma *vars-add*: *vars* (*p1 + p2*) ⊆ *vars p1* ∪ *vars p2*
proof
fix *w* **assume** *w ∈ vars* (*p1 + p2*)
then obtain *m* **where** *w ∈ keys m m ∈ keys* (*mapping-of* (*p1 + p2*)) **by** (*metis UN-E vars-def*)
then have *m ∈ keys* (*mapping-of* (*p1*)) ∪ *keys* (*mapping-of* (*p2*)) **by** (*metis keys-add-subset plus-mpoly.rep-eq subsetCE*)
then show *w ∈ vars p1* ∪ *vars p2* **using** *vars-def* ‹*w ∈ keys m*› **by** *fastforce*
qed

lemma *vars-mult*: *vars* (*p*∗*q*) ⊆ *vars p* ∪ *vars q*
proof
fix *x* **assume** *x*∈*vars* (*p*∗*q*)
then obtain *m* **where** *m*∈*keys* (*mapping-of* (*p*∗*q*)) *x*∈*keys m*
using *vars-def* **by** *blast*
then have *m*∈*keys* (*mapping-of p* ∗ *mapping-of q*)
by (*simp add*: *times-mpoly.rep-eq*)
then obtain *a b* **where** *m*=*a* + *b a ∈ keys* (*mapping-of p*) *b ∈ keys* (*mapping-of q*)
using *keys-mult* **by** *blast*
then have *x ∈ keys a* ∪ *keys b* **by** (*metis* ‹*x ∈ keys m*› *keys-add-subset subsetCE*)
then show *x ∈ vars p* ∪ *vars q* **unfolding** *vars-def*
using ‹*a ∈ keys* (*mapping-of p*)› ‹*b ∈ keys* (*mapping-of q*)› **by** *blast*
qed

lemma *vars-add-monom*:
assumes *p2 = monom m a m ∉ keys* (*mapping-of p1*)

shows *vars (p1 + p2) = vars p1 ∪ vars p2*
proof −
have *keys (mapping-of p2) ⊆ {m}* **using** *monom-def keys-single assms* **by** *auto*
have *keys (mapping-of (p1+p2)) = keys (mapping-of p1) ∪ keys (mapping-of p2)*
using *keys-add* **by** (*metis Int-insert-right-if0 ‹keys (mapping-of p2) ⊆ {m}› assms(2) inf-bot-right plus-mpoly.rep-eq subset-singletonD*)
then show *?thesis* **unfolding** *vars-def* **by** *simp*
qed

lemma *vars-setsum*: *finite S ⟹ vars (∑ m∈S. f m) ⊆ (⋃ m∈S. vars (f m))*
proof (*induction S rule:finite-induct*)
case *empty*
then show *?case* **by** (*metis UN-empty eq-iff monom-zero sum.empty single-zero vars-monom-single-cases*)
next
case (*insert s S*)
then have *vars (sum f (insert s S)) = vars (f s + sum f S)* **by** (*metis sum.insert*)
also have *... ⊆ vars (f s) ∪ vars (sum f S)* **by** (*simp add*: *vars-add*)
also have *... ⊆ (⋃ m∈insert s S. vars (f m))* **using** *insert.IH* **by** *auto*
finally show *?case* **by** *metis*
qed

lemma *coeff-monom*: *coeff (monom m a) m′ = (a when m′=m)*
by (*simp add*: *coeff-def lookup-single-not-eq when-def*)

lemma *coeff-add*: *coeff p m + coeff q m = coeff (p+q) m*
by (*simp add*: *coeff-def lookup-add plus-mpoly.rep-eq*)

lemma *coeff-eq*: *coeff p = coeff q ⟷ p=q* **by** (*simp add*: *coeff-def lookup-inject mapping-of-inject*)

lemma *coeff-monom-mult*: *coeff ((monom m′ a) * q) (m′ + m) = a * coeff q m*
unfolding *coeff-def PP-MPoly.times-mpoly.rep-eq lookup-mult mapping-of-monom lookup-single when-mult*
Sum-any-when-equal′ Groups.cancel-semigroup-add-class.add-left-cancel **by** *metis*

lemma *one-term-is-monomial*:
assumes *card (keys (mapping-of p)) ≤ 1*
obtains *m* **where** *p = monom m (coeff p m)*
proof (*cases keys (mapping-of p) = {}*)
case *True*
then show *?thesis* **using** *aux coeff-def empty-iff mapping-of-inject mapping-of-monom not-in-keys-iff-lookup-eq-zero single-zero* **by** (*metis* (*no-types*) *that*)
next
case *False*
then obtain *m* **where** *keys (mapping-of p) = {m}* **using** *assms* **by** (*metis One-nat-def Suc-leI antisym card-0-eq card-eq-SucD finite-keys neq0-conv*)
have *p = monom m (coeff p m)*

unfolding *mapping-of-inject*[*symmetric*]
by (*rule poly-mapping-eqI*, *metis* (*no-types*, *lifting*) ‹*keys* (*mapping-of p*) = {*m*}›
coeff-def keys-single lookup-single-eq mapping-of-monom not-in-keys-iff-lookup-eq-zero singletonD)
then show *?thesis* **..**
qed

definition *remove-term*::(*nat* ⇒$_0$ *nat*) ⇒ *'a*::*zero mpoly* ⇒ *'a mpoly* **where**
remove-term m0 p = *MPoly* (*Abs-poly-mapping* (λ*m*. *coeff p m when m* ≠ *m0*))

lemma *remove-term-coeff*: *coeff* (*remove-term m0 p*) *m* = (*coeff p m when m* ≠ *m0*)
proof −
have {*m*. (*coeff p m when m* ≠ *m0*) ≠ *0*} ⊆ {*m*. *coeff p m* ≠ *0*} **by** *auto*
then have *finite* {*m*. (*coeff p m when m* ≠ *m0*) ≠ *0*} **unfolding** *coeff-def* **using** *finite-subset* **by** *auto*
then have *lookup* (*Abs-poly-mapping* (λ*m*. *coeff p m when m* ≠ *m0*)) *m* = (*coeff p m when m* ≠ *m0*) **using** *lookup-Abs-poly-mapping* **by** *fastforce*
then show *?thesis* **unfolding** *remove-term-def* **using** *coeff-def* **by** (*metis* (*mono-tags*, *lifting*) *Quotient-mpoly Quotient-rep-abs-fold-unmap*)
qed

lemma *coeff-keys*: *m* ∈ *keys* (*mapping-of p*) ⟷ *coeff p m* ≠ *0* **by** (*simp add*: *coeff-def*)

lemma *remove-term-keys*:
shows *keys* (*mapping-of p*) − {*m*} = *keys* (*mapping-of* (*remove-term m p*)) (**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix *m'* **assume** *m'*∈*?A*
then show *m'* ∈ *?B* **by** (*simp add*: *coeff-keys remove-term-coeff*)
qed
show *?B* ⊆ *?A*
proof
fix *m'* **assume** *m'*∈ *?B*
then show *m'* ∈ *?A* **by** (*simp add*: *coeff-keys remove-term-coeff*)
qed
qed

lemma *remove-term-sum*: *remove-term m p* + *monom m* (*coeff p m*) = *p*
proof −
have *coeff p* = (λ*m'*. (*coeff p m' when m'* ≠ *m*) + ((*coeff p m*) *when m'*=*m*))
unfolding *when-def* **by** *fastforce*
moreover have *coeff* (*remove-term m p* + *monom m* (*coeff p m*)) = ...

using *remove-term-coeff coeff-monom coeff-add* **by** (*metis* (*no-types*))
ultimately show *?thesis* **using** *coeff-eq* **by** *auto*
qed

lemma *mpoly-induct* [*case-names monom sum*]:
assumes *monom*:$\bigwedge m\ a.\ P\ (monom\ m\ a)$
and *sum*:$(\bigwedge p1\ p2\ m\ a.\ P\ p1 \Longrightarrow P\ p2 \Longrightarrow p2 = (monom\ m\ a) \Longrightarrow m \notin keys\ (mapping\text{-}of\ p1) \Longrightarrow P\ (p1+p2))$
shows *P p* **using** *assms*
using *poly-mapping-induct*[*of* $\lambda p :: (nat \Rightarrow_0 nat) \Rightarrow_0 'a.\ P\ (MPoly\ p)$] *MPoly-induct monom.abs-eq plus-mpoly.abs-eq*
by (*metis* (*no-types*) *MPoly-inverse UNIV-I*)

lemma *monom-pow*:*monom* (*PP-Poly-Mapping.single v n0*) *a* ^ *n* = *monom* (*PP-Poly-Mapping.single v* (*n0*∗*n*)) (*a* ^ *n*)
apply (*induction n*)
apply *auto*
by (*metis* (*no-types*, *lifting*) *mult-monom single-add*)

lemma *insertion-fun-single*: *insertion-fun f* (λ*m*. (*a when* (*PP-Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) = *m*)) = *a* ∗ *f v* ^ *n* (**is** *?i* = -)
proof −
have *setsum-single*:$\bigwedge\ a\ f.\ (\sum m{\in}\{a\}.\ f\ m) = f\ a$
by (*metis add.right-neutral empty-Diff finite.emptyI sum.empty sum.insert-remove*)

have *1*:*?i* = $(\sum m.\ (a\ when\ PP\text{-}Poly\text{-}Mapping.single\ v\ n = m) * (\prod v.\ f\ v\ \hat{}\ lookup\ m\ v))$
unfolding *insertion-fun-def* **by** *metis*
have $\forall m.\ m \neq PP\text{-}Poly\text{-}Mapping.single\ v\ n \longrightarrow (a\ when\ PP\text{-}Poly\text{-}Mapping.single\ v\ n = m) = 0$ **by** *simp*

have $(\sum m{\in}\{PP\text{-}Poly\text{-}Mapping.single\ v\ n\}.\ (a\ when\ PP\text{-}Poly\text{-}Mapping.single\ v\ n = m) * (\prod v.\ f\ v\ \hat{}\ lookup\ m\ v)) = ?i$
unfolding *1 when-mult* **unfolding** *when-def* **by** *auto*
then have *2*:*?i* = *a* ∗ $(\prod va.\ f\ va\ \hat{}\ lookup\ (PP\text{-}Poly\text{-}Mapping.single\ v\ n)\ va)$
unfolding *setsum-single*[*of* $\lambda m.\ (a\ when\ PP\text{-}Poly\text{-}Mapping.single\ v\ n = m) * (\prod v.\ f\ v\ \hat{}\ lookup\ m\ v)$ *PP-Poly-Mapping.single k v*]
by *auto*
have $\forall v0.\ v0 \neq v \longrightarrow lookup\ (PP\text{-}Poly\text{-}Mapping.single\ v\ n)\ v0 = 0$ **by** (*simp add*: *lookup-single-not-eq*)
then have $\forall va.\ va \neq v \longrightarrow f\ va\ \hat{}\ lookup\ (PP\text{-}Poly\text{-}Mapping.single\ v\ n)\ va = 1$ **by** *simp*
then have *a* ∗ $(\prod va{\in}\{v\}.\ f\ va\ \hat{}\ lookup\ (PP\text{-}Poly\text{-}Mapping.single\ v\ n)\ va) = ?i$ **unfolding** *2*
using *Prod-any.expand-superset*[*of* $\{v\}$ $\lambda va.\ f\ va\ \hat{}\ lookup\ (PP\text{-}Poly\text{-}Mapping.single\ v\ n)\ va$, *simplified*]
by *fastforce*
then show *?thesis* **by** *simp*
qed

lemma *insertion-single*[*simp*]: *insertion f* (*monom* (*PP-Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) *a*) = *a* ∗ *f v* ^ *n*
using *insertion-fun-single Sum-any.strong-cong insertion.rep-eq insertion-aux.rep-eq insertion-fun-def*
mapping-of-monom single.rep-eq **by** (*metis* (*no-types*, *lifting*))

lemma *insertion-fun-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒ '*a*::*comm-ring-1*)
assumes ⋀*m v*. *p m* ≠ *0* ⟹ *lookup m v* ≠ *0* ⟹ *f v* = *g v*
shows *insertion-fun f p* = *insertion-fun g p*
proof −
{
fix *m*::*nat*⇒$_0$*nat*
assume *p m* ≠ *0*
then have (∏ *v*. *f v* ^ *lookup m v*) = (∏ *v*. *g v* ^ *lookup m v*)
using *assms* **by** (*metis power-0*)
}
then show *?thesis* **unfolding** *insertion-fun-def* **by** (*metis* (*no-types*, *lifting*) *mult-not-zero*)
qed

lemma *insertion-aux-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒$_0$ '*a*::*comm-ring-1*)
assumes ⋀*m v*. *lookup p m* ≠ *0* ⟹ *lookup m v* ≠ *0* ⟹ *f v* = *g v*
shows *insertion-aux f p* = *insertion-aux g p*
using *insertion-fun-irrelevant-vars*[*of lookup p f g*] *assms*
by (*metis insertion-aux.rep-eq*)

lemma *insertion-irrelevant-vars*:
fixes *p*::'*a*::*comm-ring-1 mpoly*
assumes ⋀*v*. *v*∈*vars p* ⟹ *f v* = *g v*
shows *insertion f p* = *insertion g p*
proof −
{
fix *m v* **assume** *lookup* (*mapping-of p*) *m* ≠ *0 lookup m v* ≠ *0*
then have *v* ∈ *vars p* **unfolding** *vars-def* **by** (*meson UN-I lookup-not-eq-zero-eq-in-keys*)
then have *f v* = *g v* **using** *assms* **by** *auto*
}
then show *?thesis*
unfolding *insertion-def* **using** *insertion-aux-irrelevant-vars*[*of mapping-of p*]
by (*metis insertion.rep-eq insertion-def*)
qed

32 Nested MPoly

definition *reduce-nested-mpoly*::'*a*::*comm-ring-1 mpoly mpoly* ⇒ '*a mpoly* **where**
reduce-nested-mpoly pp = *insertion* (λ*v*. *monom* (*PP-Poly-Mapping.single v 1*) *1*) *pp*

lemma *reduce-nested-mpoly-sum*:
fixes *p1*::*'a*::*comm-ring-1 mpoly mpoly*
shows *reduce-nested-mpoly* (*p1* + *p2*) = *reduce-nested-mpoly p1* + *reduce-nested-mpoly p2*
by (*simp add*: *insertion-add reduce-nested-mpoly-def*)

lemma *reduce-nested-mpoly-prod*:
fixes *p1*::*'a*::*comm-ring-1 mpoly mpoly*
shows *reduce-nested-mpoly* (*p1* ∗ *p2*) = *reduce-nested-mpoly p1* ∗ *reduce-nested-mpoly p2*
by (*simp add*: *insertion-mult reduce-nested-mpoly-def*)

lemma *reduce-nested-mpoly-0*:
shows *reduce-nested-mpoly 0* = *0* **by** (*simp add*: *reduce-nested-mpoly-def*)

lemma *insertion-nested-poly*:
fixes *pp*::*'a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*insertion* (λ*v*. *monom 0* (*f v*)) *pp*) = *insertion f* (*reduce-nested-mpoly pp*)
proof (*induction pp rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
case (*single v n*)
show *?case* **unfolding** *reduce-nested-mpoly-def*
apply (*simp add*: *insertion-mult monom-pow*)
using *monom-pow*[*of 0 0 f v n*] **apply** *simp*
using *insertion-single*[*of f 0 0*] **by** *auto*
next
case (*sum m1 m2 k v*)
then have *insertion f* (*insertion* (λ*v*. *monom 0* (*f v*)) (*monom m1 a* ∗ *monom m2 1*))
= *insertion f* (*reduce-nested-mpoly* (*monom m1 a* ∗ *monom m2 1*))
unfolding *reduce-nested-mpoly-prod insertion-mult* **by** *metis*
then show *?case* **using** *mult-monom*[*of m1 a m2 1*] **by** *auto*
qed
next
case (*sum p1 p2 m a*)
then show *?case* **by** (*simp add*: *reduce-nested-mpoly-sum insertion-add*)
qed

definition *extract-var*::*'a*::*comm-ring-1 mpoly* ⇒ *nat* ⇒ *'a*::*comm-ring-1 mpoly mpoly* **where**
extract-var p v = (∑ *m*. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))

lemma *extract-var-finite-set*:
assumes {*m'*. *coeff p m'* ≠ *0*} ⊆ *S*

assumes *finite S*
shows *extract-var p v* = (∑ *m*∈*S*. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
proof −
{
fix *m′* **assume** *coeff p m′* = *0*
then have *monom* (*remove-key v m′*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m′ v*)) (*coeff p m′*)) = *0*
using *monom.abs-eq monom-zero single-zero* **by** *metis*
}
then have *0*:{*a*. *monom* (*remove-key v a*) (*monom* (*PP-Poly-Mapping.single v* (*lookup a v*)) (*coeff p a*)) ≠ *0*} ⊆ *S*
using ‹{*m′*. *coeff p m′* ≠ *0*} ⊆ *S*› **by** *fastforce*
then show *?thesis*
unfolding *extract-var-def* **using** *Sum-any.expand-superset* [*OF* ‹*finite S*› *0*]
by *metis*
qed

lemma *extract-var-non-zero-coeff*: *extract-var p v* = (∑ *m*∈{*m′*. *coeff p m′* ≠ *0*}. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
using *extract-var-finite-set coeff-def finite-lookup order-refl* **by** (*metis* (*no-types*, *lifting*) *Collect-cong sum.cong*)

lemma *extract-var-sum*: *extract-var* (*p*+*p′*) *v* = *extract-var p v* + *extract-var p′ v*
proof −
def *S* == {*m*. *coeff p m* ≠ *0*} ∪ {*m*. *coeff p′ m* ≠ *0*} ∪ {*m*. *coeff* (*p*+*p′*) *m* ≠ *0*}
have *subsets*:{*m*. *coeff p m* ≠ *0*} ⊆ *S* {*m*. *coeff p′ m* ≠ *0*} ⊆ *S* {*m*. *coeff* (*p*+*p′*) *m* ≠ *0*} ⊆ *S* **unfolding** *S-def* **by** *auto*
have *finite S* **unfolding** *S-def* **using** *coeff-def finite-lookup* **by** (*metis* (*mono-tags*) *Collect-disj-eq finite-Collect-disjI*)
then show *?thesis* **unfolding**
extract-var-finite-set[*OF subsets*(*1*) ‹*finite S*›]
extract-var-finite-set[*OF subsets*(*2*) ‹*finite S*›]
extract-var-finite-set[*OF subsets*(*3*) ‹*finite S*›]
coeff-add[*symmetric*] *monom-add sum.distrib*
by *metis*
qed

lemma *extract-var-monom*:
shows *extract-var* (*monom m a*) *v* = *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) *a*)
proof (*cases a* = *0*)
assume *a* ≠ *0*
have *0*:{*m′*. *coeff* (*monom m a*) *m′* ≠ *0*} = {*m*}
unfolding *coeff-monom* **using** ‹*a* ≠ *0*› **by** *auto*

```
  show ?thesis
    unfolding extract-var-non-zero-coeff unfolding 0 unfolding coeff-monom
      using sum.insert[OF finite.emptyI, unfolded sum.empty add.right-neutral]
when-def
    by auto
next
  assume a = 0
  have 0:{m'. coeff (monom m a) m' ≠ 0} = {}
    unfolding coeff-monom using ‹a = 0› by auto
  show ?thesis unfolding extract-var-non-zero-coeff 0
     using ‹a = 0› monom.abs-eq monom-zero sum.empty single-zero by (metis
(no-types, lifting))
qed


lemma extract-var-monom-mult:
shows extract-var (monom (m+m') (a*b)) v = extract-var (monom m a) v *
extract-var (monom m' b) v
unfolding extract-var-monom remove-key-add lookup-add single-add mult-monom
by auto

lemma extract-var-single: extract-var (monom (PP-Poly-Mapping.single v n) a)
v = monom 0 (monom (PP-Poly-Mapping.single v n) a)
unfolding extract-var-monom by simp

lemma extract-var-single':
assumes v ≠ v'
shows extract-var (monom (PP-Poly-Mapping.single v n) a) v' = monom (PP-Poly-Mapping.single
v n) (monom 0 a)
unfolding extract-var-monom using assms by (metis add.right-neutral lookup-single-not-eq
remove-key-sum single-zero)

lemma reduce-nested-mpoly-extract-var:
fixes p::'a::comm-ring-1 mpoly
shows reduce-nested-mpoly (extract-var p v) = p
proof (induction p rule:mpoly-induct)
  case (monom m a)
  then show ?case
  proof (induction m arbitrary:a rule:poly-mapping-induct)
    case (single v' n)
    show ?case
    proof (cases v' = v)
      case True
      then show ?thesis
        by (metis (no-types, lifting) insertion-single mult.right-neutral power-0
        reduce-nested-mpoly-def single-zero extract-var-single)
    next
      case False
    then show ?thesis unfolding extract-var-single'[OF False] reduce-nested-mpoly-def
```

insertion-single
by (*simp add*: *monom-pow mult-monom*)
qed
next
case (*sum m m' v n a*)
then show *?case*
using *extract-var-monom-mult*[*of m m' a 1*] *reduce-nested-mpoly-prod* **by** (*metis mult.right-neutral mult-monom*)
qed
next
case (*sum p1 p2 m a*)
then show *?case* **unfolding** *extract-var-sum reduce-nested-mpoly-sum* **by** *auto*
qed

lemma *vars-extract-var-subset*: *vars* (*extract-var p v*) ⊆ *vars p*
proof
have *finite* {*m'*. *coeff p m'* ≠ *0*} **by** (*simp add*: *coeff-def*)
fix *x* **assume** *x* ∈ *vars* (*extract-var p v*)
then have *x* ∈ *vars* (∑ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
unfolding *extract-var-non-zero-coeff* **by** *metis*
then have *x* ∈ (⋃ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *vars* (*monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*))))
using *vars-setsum*[*OF* ‹*finite* {*m'*. *coeff p m'* ≠ *0*}›] **by** *auto*
then obtain *m* **where** *m*∈{*m'*. *coeff p m'* ≠ *0*} *x* ∈ *vars* (*monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
by *blast*
show *x* ∈ *vars p* **by** (*metis* (*mono-tags*, *lifting*) *DiffD1 UN-I* ‹*m* ∈ {*m'*. *coeff p m'* ≠ *0*}›
‹*x* ∈ *vars* (*monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))›
coeff-keys mem-Collect-eq remove-key-keys subsetCE vars-def vars-monom-subset)
qed

lemma *v-not-in-vars-extract-var*: *v* ∉ *vars* (*extract-var p v*)
proof −
have *finite* {*m'*. *coeff p m'* ≠ *0*} **by** (*simp add*: *coeff-def*)
have ⋀*m*. *m*∈{*m'*. *coeff p m'* ≠ *0*} ⟹ *v* ∉ *vars* (*monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
by (*metis Diff-iff remove-key-keys singletonI subsetCE vars-monom-subset*)
then have *v* ∉ (⋃ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *vars* (*monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*))))
by *simp*
then show *?thesis*
unfolding *extract-var-non-zero-coeff* **using** *vars-setsum*[*OF* ‹*finite* {*m'*. *coeff p m'* ≠ *0*}›] **by** *blast*
qed

lemma *vars-coeff-extract-var*: *vars* (*coeff* (*extract-var p v*) *j*) ⊆ {*v*}
proof (*induction p rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case* **unfolding** *extract-var-monom coeff-monom vars-monom-single-cases*
by (*metis monom-zero single-zero vars-monom-single when-def*)
next
case (*sum p1 p2 m a*)
then show *?case* **unfolding** *extract-var-sum coeff-add*[*symmetric*]
by (*metis* (*no-types*, *lifting*) *Un-insert-right insert-absorb2 subset-insertI2 subset-singletonD sup-bot.right-neutral vars-add*)
qed

definition *replace-coeff*
where *replace-coeff f p* = *MPoly* (*Abs-poly-mapping* (λ*m*. *f* (*lookup* (*mapping-of p*) *m*)))

lemma *coeff-replace-coeff*:
assumes *f 0* = *0*
shows *coeff* (*replace-coeff f p*) *m* = *f* (*coeff p m*)
proof −
have *0*:*finite* {*m*. *f* (*lookup* (*mapping-of p*) *m*) ≠ *0*}
unfolding *coeff-def*[*symmetric*] **by** (*metis* (*mono-tags*, *lifting*) *Collect-mono assms*(*1*) *coeff-def finite-lookup finite-subset*)+
then show *?thesis* **unfolding** *replace-coeff-def coeff-def* **using** *lookup-Abs-poly-mapping*[*OF 0*]
by (*metis* (*mono-tags*, *lifting*) *Quotient-mpoly Quotient-rep-abs-fold-unmap*)
qed

lemma *replace-coeff-monom*:
assumes *f 0* = *0*
shows *replace-coeff f* (*monom m a*) = *monom m* (*f a*)
unfolding *replace-coeff-def*
unfolding *mapping-of-inject*[*symmetric*] *lookup-inject*[*symmetric*] **apply** (*rule HOL.ext*)
unfolding *lookup-single mapping-of-monom fun-when*[*of f*, *OF* ‹*f 0* = *0*›]
by (*metis coeff-def coeff-monom lookup-single lookup-single-not-eq monom.abs-eq single.abs-eq*)

lemma *replace-coeff-add*:
assumes *f 0* = *0*
assumes ⋀*a b*. *f* (*a*+*b*) = *f a* + *f b*
shows *replace-coeff f* (*p1* + *p2*) = *replace-coeff f p1* + *replace-coeff f p2*
proof −
have *finite* {*m*. *f* (*lookup* (*mapping-of p1*) *m*) ≠ *0*}
finite {*m*. *f* (*lookup* (*mapping-of p2*) *m*) ≠ *0*}
unfolding *coeff-def*[*symmetric*] **by** (*metis* (*mono-tags*, *lifting*) *Collect-mono assms*(*1*) *coeff-def finite-lookup finite-subset*)+
then show *?thesis*
unfolding *replace-coeff-def PP-MPoly.plus-mpoly.rep-eq* **unfolding** *PP-Poly-Mapping.plus-poly-mapping.re*

unfolding *assms*(2) *PP-MPoly.plus-mpoly.abs-eq* **using** *PP-Poly-Mapping.plus-poly-mapping.abs-eq*[*unfold*
eq-onp-def] **by** *fastforce*
qed

lemma *insertion-replace-coeff*:
fixes *pp*::'*a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*replace-coeff* (*insertion f*) *pp*) = *insertion f* (*reduce-nested-mpoly*
pp)
proof (*induction pp rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
case (*single v n*)
show *?case* **unfolding** *reduce-nested-mpoly-def* **unfolding** *replace-coeff-monom*[*of*
insertion f, *OF insertion-zero*]
insertion-single insertion-mult **using** *insertion-single* **by** (*simp add*: *monom-pow*)
next
case (*sum m1 m2 k v*)
have *replace-coeff* (*insertion f*) (*monom m1 a* ∗ *monom m2 1*) = *replace-coeff*
(*insertion f*) (*monom m1 a*) ∗ *replace-coeff* (*insertion f*) (*monom m2 1*)
by (*simp add*: *mult-monom replace-coeff-monom*)
then have *insertion f* (*replace-coeff* (*insertion f*) (*monom m1 a* ∗ *monom m2*
1)) = *insertion f* (*reduce-nested-mpoly* (*monom m1 a* ∗ *monom m2 1*))
unfolding *reduce-nested-mpoly-prod insertion-mult*
by (*simp add*: *insertion-mult sum.IH*(*1*) *sum.IH*(*2*))
then show *?case* **using** *mult-monom*[*of m1 a m2 1*] **by** *auto*
qed
next
case (*sum p1 p2 m a*)
then show *?case* **using** *reduce-nested-mpoly-sum insertion-add*
replace-coeff-add[*of insertion f*, *OF insertion-zero insertion-add*] **by** *metis*
qed

lemma *replace-coeff-extract-var-cong*:
assumes *f v* = *g v*
shows *replace-coeff* (*insertion f*) (*extract-var p v*) = *replace-coeff* (*insertion g*)
(*extract-var p v*)
by (*induction p rule*:*mpoly-induct*;*simp add*: *assms extract-var-monom replace-coeff-monom*
extract-var-sum insertion-add replace-coeff-add)

lemma *vars-replace-coeff*:
assumes *f* 0 = 0
shows *vars* (*replace-coeff f p*) ⊆ *vars p*
unfolding *vars-def* **apply** (*rule subsetI*) **unfolding** *mem-simps*(*8*) *coeff-keys*
using *assms coeff-replace-coeff* **by** (*metis coeff-keys*)

end

33 Polynomials representing the Deep Network Model

theory *DL-Deep-Model-Poly*
imports *DL-Deep-Model PP-More-MPoly Jordan-Normal-Form.Determinant*
begin

definition *polyfun N f* = (∃ *p*. *vars p* ⊆ *N* ∧ (∀ *x*. *insertion x p* = *f x*))

lemma *polyfunI*: (⋀*P*. (⋀*p*. *vars p* ⊆ *N* ⟹ (⋀*x*. *insertion x p* = *f x*) ⟹ *P*) ⟹ *P*) ⟹ *polyfun N f*
unfolding *polyfun-def* **by** *metis*

lemma *polyfun-subset*: *N*⊆*N'* ⟹ *polyfun N f* ⟹ *polyfun N' f*
unfolding *polyfun-def* **by** *blast*

lemma *polyfun-const*: *polyfun N* (λ-. *c*)
proof −
have ⋀*x*. *insertion x* (*monom 0 c*) = *c* **using** *insertion-single* **by** (*metis insertion-one monom-one mult.commute mult.right-neutral single-zero*)
then show *?thesis* **unfolding** *polyfun-def* **by** (*metis* (*full-types*) *empty-iff keys-single single-zero subsetI subset-antisym vars-monom-subset*)
qed

lemma *polyfun-add*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* + *g x*)
proof −
obtain *p1 p2* **where** *vars p1* ⊆ *N* ∀ *x*. *insertion x p1* = *f x*
vars p2 ⊆ *N* ∀ *x*. *insertion x p2* = *g x*
using *polyfun-def assms* **by** *metis*
then have *vars* (*p1* + *p2*) ⊆ *N* ∀ *x*. *insertion x* (*p1* + *p2*) = *f x* + *g x*
using *vars-add* **using** *Un-iff subsetCE subsetI* **apply** *blast*
by (*simp add*: ‹∀ *x*. *insertion x p1* = *f x*› ‹∀ *x*. *insertion x p2* = *g x*› *insertion-add*)
then show *?thesis* **using** *polyfun-def* **by** *blast*
qed

lemma *polyfun-mult*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* ∗ *g x*)
proof −
obtain *p1 p2* **where** *vars p1* ⊆ *N* ∀ *x*. *insertion x p1* = *f x*
vars p2 ⊆ *N* ∀ *x*. *insertion x p2* = *g x*
using *polyfun-def assms* **by** *metis*
then have *vars* (*p1* ∗ *p2*) ⊆ *N* ∀ *x*. *insertion x* (*p1* ∗ *p2*) = *f x* ∗ *g x*
using *vars-mult* **using** *Un-iff subsetCE subsetI* **apply** *blast*
by (*simp add*: ‹∀ *x*. *insertion x p1* = *f x*› ‹∀ *x*. *insertion x p2* = *g x*› *insertion-mult*)
then show *?thesis* **using** *polyfun-def* **by** *blast*
qed

lemma *polyfun-Sum*:
assumes *finite I*
assumes $\bigwedge i.\ i{\in}I \Longrightarrow$ *polyfun N* (*f i*)
shows *polyfun N* ($\lambda x.\ \sum i{\in}I.\ f\ i\ x$)
using *assms*
apply (*induction I rule*:*finite-induct*)
apply (*simp add*: *polyfun-const*)
using *comm-monoid-add-class.sum.insert polyfun-add* **by** *fastforce*

lemma *polyfun-Prod*:
assumes *finite I*
assumes $\bigwedge i.\ i{\in}I \Longrightarrow$ *polyfun N* (*f i*)
shows *polyfun N* ($\lambda x.\ \prod i{\in}I.\ f\ i\ x$)
using *assms*
apply (*induction I rule*:*finite-induct*)
apply (*simp add*: *polyfun-const*)
using *comm-monoid-add-class.sum.insert polyfun-mult* **by** *fastforce*

lemma *polyfun-single*:
assumes $i{\in}N$
shows *polyfun N* ($\lambda x.\ x\ i$)
proof −
have $\forall f.$ *insertion f* (*monom* (*PP-Poly-Mapping.single i 1*) *1*) = *f i* **using** *insertion-single* **by** *simp*
then show *?thesis* **unfolding** *polyfun-def*
using *vars-monom-single*[*of i 1 1*] *One-nat-def assms singletonD subset-eq*
by *blast*
qed

lemma *polyfun-det*:
assumes $\bigwedge x.\ (A\ x) \in$ *carrier-mat n n*
assumes $\bigwedge x\ i\ j.\ i{<}n \Longrightarrow j{<}n \Longrightarrow$ *polyfun N* ($\lambda x.\ (A\ x)$ \$\$ (i,j))
shows *polyfun N* ($\lambda x.$ *det* (*A x*))
proof −
{
fix *p* **assume** $p{\in}$ {*p*. *p permutes* {*0*..<*n*}}
then have *p permutes* {*0*..<*n*} **by** *auto*
then have $\bigwedge x.\ x < n \Longrightarrow p\ x < n$ **using** *permutes-in-image* **by** *auto*
then have *polyfun N* ($\lambda x.\ \prod i = 0..{<}n.\ A\ x$ \$\$ (i, $p\ i$))
using *polyfun-Prod*[*of* {*0*..<*n*} *N* $\lambda i\ x.\ A\ x$ \$\$ (i, $p\ i$)] *assms* **by** *simp*
then have *polyfun N* ($\lambda x.$ *signof p* $*$ ($\prod i = 0..{<}n.\ A\ x$ \$\$ (i, $p\ i$))) **using** *polyfun-const polyfun-mult* **by** *blast*
}
moreover have *finite* {*i*. *i permutes* {*0*..<*n*}} **by** (*simp add*: *finite-permutations*)
ultimately show *?thesis* **unfolding** *det-def'*[*OF assms*(*1*)]
using *polyfun-Sum*[*OF* ‹*finite* {*i*. *i permutes* {*0*..<*n*}}›, *of N* $\lambda p\ x.$ *signof p* $*$ ($\prod i = 0..{<}n.\ A\ x$ \$\$ (i, $p\ i$))]
by *blast*
qed

lemma *polyfun-extract-matrix*:
assumes $i<m$ $j<n$
shows *polyfun* $\{..<a + (m * n + c)\}$ $(\lambda f.$ *extract-matrix* $(\lambda i.$ f $(i + a))$ m n \$\$ $(i,j))$
unfolding *index-extract-matrix*[*OF assms*] **apply** (*rule polyfun-single*) **using** *two-digit-le*[*OF assms*] **by** *simp*

lemma *polyfun-mult-mat-vec*:
assumes $\bigwedge x.$ v $x \in$ *carrier-vec* n
assumes $\bigwedge j.$ $j<n \Longrightarrow$ *polyfun* N $(\lambda x.$ v x \$ $j)$
assumes $\bigwedge x.$ A $x \in$ *carrier-mat* m n
assumes $\bigwedge i$ $j.$ $i<m \Longrightarrow j<n \Longrightarrow$ *polyfun* N $(\lambda x.$ A x \$\$ $(i,j))$
assumes $j < m$
shows *polyfun* N $(\lambda x.$ $((A$ $x)$ $*_v$ $(v$ $x))$ \$ $j)$
proof −
have $\bigwedge x.$ $j <$ *dim-row* $(A$ $x)$ **using** ‹$j < m$› *assms*(*3*) *carrier-matD*(*1*) **by** *force*
have $\bigwedge x.$ $n =$ *dim-vec* $(v$ $x)$ **using** *assms*(*1*) *carrier-vecD* **by** *fastforce*
{
fix i **assume** $i \in \{0..<n\}$
then have $i < n$ **by** *auto*
{
fix x
have $i <$ *dim-vec* $(v$ $x)$ **using** *assms*(*1*) *carrier-vecD* ‹$i<n$› **by** *fastforce*
have $j <$ *dim-row* $(A$ $x)$ **using** ‹$j < m$› *assms*(*3*) *carrier-matD*(*1*) **by** *force*
have *dim-col* $(A$ $x) =$ *dim-vec* $(v$ $x)$ **by** (*metis assms*(*1*) *assms*(*3*) *carrier-matD*(*2*) *carrier-vecD*)
then have *row* $(A$ $x)$ j \$ $i = A$ x \$\$ (j,i) $i<n$ **using** ‹$j <$ *dim-row* $(A$ $x)$› ‹$i<n$› **by** (*simp-all add*: ‹$i <$ *dim-vec* $(v$ $x)$›)
}
then have *polyfun* N $(\lambda x.$ *row* $(A$ $x)$ j \$ $i * v$ x \$ $i)$
using *polyfun-mult assms*(*4*)[*OF* ‹$j < m$›] *assms*(*2*) **by** *fastforce*
}
then show *?thesis* **unfolding** *index-mult-mat-vec*[*OF* ‹$\bigwedge x.$ $j <$ *dim-row* $(A$ $x)$›] *scalar-prod-def*
using *polyfun-Sum*[*of* $\{0..<n\}$ N λi $x.$ *row* $(A$ $x)$ j \$ $i * v$ x \$ i] *finite-atLeastLessThan*[*of* *0* n] ‹$\bigwedge x.$ $n =$ *dim-vec* $(v$ $x)$›
by *simp*
qed

lemma *polyfun-evaluate-net-plus-a*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes $j <$ *output-size m*
shows *polyfun* $\{..<a +$ *count-weights m*$\}$ $(\lambda f.$ *evaluate-net* (*insert-weights m* $(\lambda i.$ f $(i + a)))$ *inputs* \$ $j)$
using *assms* **proof** (*induction m arbitrary*:*inputs j a*)
case (*Input*)

then show *?case* **unfolding** *insert-weights.simps evaluate-net.simps* **using** *polyfun-const*
by *metis*
next
case (*Conv x m*)
then obtain *x1 x2* **where** *x*=(*x1*,*x2*) **by** *fastforce*
show *?case* **unfolding** ‹*x*=(*x1*,*x2*)› *insert-weights.simps evaluate-net.simps drop-map*
unfolding *list-of-vec-index*
proof (*rule polyfun-mult-mat-vec*)
{
fix *f*
have *1*:*valid-net'* (*insert-weights m* (λ*i*. *f* (*i* + *x1* ∗ *x2*)))
using ‹*valid-net* (*Conv x m*)› *valid-net.simps* **by** (*metis*
convnet.distinct(*1*) *convnet.distinct*(*5*) *convnet.inject*(*2*) *remove-insert-weights*)
have *2*:*map dim-vec inputs* = *input-sizes* (*insert-weights m* (λ*i*. *f* (*i* + *x1* ∗
x2)))
using *input-sizes-remove-weights remove-insert-weights*
by (*simp add*: *Conv.prems*(*1*))
have *dim-vec* (*evaluate-net* (*insert-weights m* (λ*i*. *f* (*i* + *x1* ∗ *x2*))) *inputs*)
= *output-size m*
using *output-size-correct*[*OF 1 2*] **using** *remove-insert-weights* **by** *auto*
then show *evaluate-net* (*insert-weights m* (λ*i*. *f* (*i* + *x1* ∗ *x2*))) *inputs* ∈
carrier-vec (*output-size m*)
using *carrier-vec-def* **by** (*metis* (*full-types*) *mem-Collect-eq*)
}

have *map dim-vec inputs* = *input-sizes m* **by** (*simp add*: *Conv.prems*(*1*))
have *valid-net m* **using** *Conv.prems*(*2*) *valid-net.cases* **by** *fastforce*
show ⋀*j*. *j* < *output-size m* ⟹ *polyfun* {..<*a* + *count-weights* (*Conv* (*x1*,
x2) *m*)}
(λ*f*. *evaluate-net* (*insert-weights m* (λ*i*. *f* (*i* + *x1* ∗ *x2* + *a*))) *inputs* $ *j*)
unfolding *vec-of-list-index count-weights.simps*
using *Conv*(*1*)[*OF* ‹*map dim-vec inputs* = *input-sizes m*› ‹*valid-net m*›, *of* -
x1 ∗ *x2* + *a*]
unfolding *semigroup-add-class.add.assoc ab-semigroup-add-class.add.commute*[*of*
x1 ∗ *x2 a*]
by *blast*

have *output-size m* = *x2* **using** *Conv.prems*(*2*) ‹*x* = (*x1*, *x2*)› *valid-net.cases*
by *fastforce*
show ⋀*f*. *extract-matrix* (λ*i*. *f* (*i* + *a*)) *x1 x2* ∈ *carrier-mat x1* (*output-size*
m) **unfolding** ‹*output-size m* = *x2*› **using** *dim-extract-matrix*
using *carrier-matI* **by** (*metis* (*no-types*, *lifting*))

show ⋀*i j*. *i* < *x1* ⟹ *j* < *output-size m* ⟹ *polyfun* {..<*a* + *count-weights*
(*Conv* (*x1*, *x2*) *m*)} (λ*f*. *extract-matrix* (λ*i*. *f* (*i* + *a*)) *x1 x2* $$ (*i*, *j*))
unfolding ‹*output-size m* = *x2*› *count-weights.simps* **using** *polyfun-extract-matrix*[*of*
- *x1* - *x2 a count-weights m*] **by** *blast*

show *j* < *x1* **using** *Conv.prems*(*3*) ‹*x* = (*x1*, *x2*)› **by** *auto*

qed
next
case (*Pool m1 m2 inputs j a*)
have *A2*:⋀*f*. *map dim-vec* (*take* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *f* (*i* + *a*))))) *inputs*) = *input-sizes m1*
by (*metis Pool.prems*(*1*) *append-eq-conv-conj input-sizes.simps*(*3*) *input-sizes-remove-weights remove-insert-weights take-map*)
have *B2*:⋀*f*. *map dim-vec* (*drop* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *f* (*i* + *a*))))) *inputs*) = *input-sizes m2*
using *Pool.prems*(*1*) *append-eq-conv-conj input-sizes.simps*(*3*) *input-sizes-remove-weights remove-insert-weights* **by** (*metis drop-map*)
have *A3*:*valid-net m1* **and** *B3*:*valid-net m2* **using** ‹*valid-net* (*Pool m1 m2*)› *valid-net.simps* **by** *blast*+
have *output-size* (*Pool m1 m2*) = *output-size m2* **unfolding** *output-size.simps*
using ‹*valid-net* (*Pool m1 m2*)› *valid-net.cases* **by** *fastforce*
then have *A4*:*j* < *output-size m1* **and** *B4*:*j* < *output-size m2* **using** ‹*j* < *output-size* (*Pool m1 m2*)› **by** *simp-all*

let *?net1* = λ*f*. *evaluate-net* (*insert-weights m1* (λ*i*. *f* (*i* + *a*)))
(*take* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *f* (*i* + *a*))))) *inputs*)
let *?net2* = λ*f*. *evaluate-net* (*insert-weights m2* (λ*i*. *f* (*i* + *count-weights m1* + *a*)))
(*drop* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *f* (*i* + *a*))))) *inputs*)
have *length1*: ⋀*f*. *output-size m1* = *dim-vec* (*?net1 f*)
by (*metis A2 A3 input-sizes-remove-weights output-size-correct remove-insert-weights*)
then have *jlength1*:⋀*f*. *j* < *dim-vec* (*?net1 f*) **using** *A4* **by** *metis*
have *length2*: ⋀*f*. *output-size m2* = *dim-vec* (*?net2 f*)
by (*metis B2 B3 input-sizes-remove-weights output-size-correct remove-insert-weights*)
then have *jlength2*:⋀*f*. *j* < *dim-vec* (*?net2 f*) **using** *B4* **by** *metis*
have *cong1*:⋀*xf*. (λ*f*. *evaluate-net* (*insert-weights m1* (λ*i*. *f* (*i* + *a*)))
(*take* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *xf* (*i* + *a*))))) *inputs*) $ *j*)
= (λ*f*. *?net1 f* $ *j*)
using *input-sizes-remove-weights remove-insert-weights* **by** *auto*
have *cong2*:⋀*xf*. (λ*f*. *evaluate-net* (*insert-weights m2* (λ*i*. *f* (*i* + (*a* + *count-weights m1*))))
(*drop* (*length* (*input-sizes* (*insert-weights m1* (λ*i*. *xf* (*i* + *a*))))) *inputs*) $ *j*)
= (λ*f*. *?net2 f* $ *j*)
unfolding *semigroup-add-class.add.assoc*[*symmetric*] *ab-semigroup-add-class.add.commute*[*of a count-weights m1*]
using *input-sizes-remove-weights remove-insert-weights* **by** *auto*

show *?case* **unfolding** *insert-weights.simps evaluate-net.simps index-component-mult*[*OF jlength1 jlength2*] *count-weights.simps*
apply (*rule polyfun-mult*)
using *Pool.IH*(*1*)[*OF A2 A3 A4*, *of a*, *unfolded cong1*]
using *Pool.IH*(*2*)[*OF B2 B3 B4*, *of a* + *count-weights m1*, *unfolded cong2 semigroup-add-class.add.assoc*[*of a*]]
using *polyfun-subset*[*of* {..<*a* + *count-weights m1*} {..<*a* + (*count-weights m1* + *count-weights m2*)}]

by *auto*
qed

lemma *polyfun-evaluate-net*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes *j* < *output-size m*
shows *polyfun* {..<*count-weights m*} (λ*f*. *evaluate-net* (*insert-weights m f*) *inputs* $ *j*)
using *polyfun-evaluate-net-plus-a*[**where** *a*=*0*, *OF assms*] **by** *simp*

lemma *polyfun-tensors-from-net*:
assumes *valid-net m*
assumes *is* ◁ *input-sizes m*
assumes *j* < *output-size m*
shows *polyfun* {..<*count-weights m*} (λ*f*. *Tensor.lookup* (*tensors-from-net* (*insert-weights m f*) $ *j*) *is*)
proof −
have *1*:⋀*f*. *valid-net'* (*insert-weights m f*) **by** (*simp add*: *assms*(*1*) *remove-insert-weights*)
have *input-sizes*:⋀*f*. *input-sizes* (*insert-weights m f*) = *input-sizes m*
unfolding *input-sizes-remove-weights* **by** (*simp add*: *remove-insert-weights*)
have *2*:⋀*f*. *is* ◁ *input-sizes* (*insert-weights m f*)
unfolding *input-sizes* **using** *assms*(*2*) **by** *blast*
have *3*:⋀*f*. *j* < *output-size'* (*insert-weights m f*)
by (*simp add*: *assms*(*3*) *remove-insert-weights*)
have ⋀*f1 f2*. *base-input* (*insert-weights m f1*) *is* = *base-input* (*insert-weights m f2*) *is*
unfolding *base-input-def* **by** (*simp add*: *input-sizes*)
then have ⋀*xf*. (λ*f*. *evaluate-net* (*insert-weights m f*) (*base-input* (*insert-weights m xf*) *is*) $ *j*)
= (λ*f*. *evaluate-net* (*insert-weights m f*) (*base-input* (*insert-weights m f*) *is*) $ *j*)
by *metis*
then show *?thesis* **unfolding** *lookup-tensors-from-net*[*OF 1 2 3*]
using *polyfun-evaluate-net*[*OF base-input-length*[*OF 2*, *unfolded input-sizes*, *symmetric*] *assms*(*1*) *assms*(*3*)]
by *fastforce*
qed

lemma *polyfun-matricize*:
assumes ⋀*x*. *dims* (*T x*) = *ds*
assumes ⋀*is*. *is* ◁ *ds* ⟹ *polyfun N* (λ*x*. *Tensor.lookup* (*T x*) *is*)
assumes ⋀*x*. *dim-row* (*matricize I* (*T x*)) = *nr*
assumes ⋀*x*. *dim-col* (*matricize I* (*T x*)) = *nc*
assumes *i* < *nr*
assumes *j* < *nc*
shows *polyfun N* (λ*x*. *matricize I* (*T x*) $$ (*i*,*j*))
proof −
let *?weave* = λ *x*. (*weave I*

(*digit-encode* (*nths ds I*) *i*)
(*digit-encode* (*nths ds* (−*I*)) *j*))
have *1*:⋀*x*. *matricize I* (*T x*) $$ (*i*,*j*) = *Tensor.lookup* (*T x*) (*?weave x*) **unfolding** *matricize-def*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) *case-prod-conv*
dim-col-mat(*1*) *dim-row-mat*(*1*) *index-mat*(*1*) *matricize-def*)
have ⋀*x*. *?weave x* ◁ *ds*
using *valid-index-weave*(*1*) *assms*(*2*) *digit-encode-valid-index dim-row-mat*(*1*) *matricize-def*
using *assms digit-encode-valid-index matricize-def* **by** (*metis dim-col-mat*(*1*))
then have *polyfun N* (λ*x*. *Tensor.lookup* (*T x*) (*?weave x*)) **using** *assms*(*2*) **by** *simp*
then show *?thesis* **unfolding** *1* **using** *assms*(*1*) **by** *blast*
qed

lemma (¬ (*a*::*nat*) < *b*) = (*a* ≥ *b*)
by (*metis not-le*)

lemma *polyfun-submatrix*:
assumes ⋀*x*. (*A x*) ∈ *carrier-mat m n*
assumes ⋀*x i j*. *i*<*m* ⟹ *j*<*n* ⟹ *polyfun N* (λ*x*. (*A x*) $$ (*i*,*j*))
assumes *i* < *card* {*i*. *i* < *m* ∧ *i* ∈ *I*}
assumes *j* < *card* {*j*. *j* < *n* ∧ *j* ∈ *J*}
assumes *infinite I infinite J*
shows *polyfun N* (λ*x*. (*submatrix* (*A x*) *I J*) $$ (*i*,*j*))
proof −
have *1*:⋀*x*. (*submatrix* (*A x*) *I J*) $$ (*i*,*j*) = (*A x*) $$ (*pick I i*, *pick J j*)
using *submatrix-index* **by** (*metis* (*no-types*, *lifting*) *Collect-cong assms*(*1*) *assms*(*3*) *assms*(*4*) *carrier-matD*(*1*) *carrier-matD*(*2*))
have *pick I i* < *m pick J j* < *n* **using** *card-le-pick-inf*[*OF* ‹*infinite I*›] *card-le-pick-inf*[*OF* ‹*infinite J*›]
‹*i* < *card* {*i*. *i* < *m* ∧ *i* ∈ *I*}›[*unfolded set-le-in*] ‹*j* < *card* {*j*. *j* < *n* ∧ *j* ∈ *J*}›[*unfolded set-le-in*] *not-less* **by** *metis*+
then show *?thesis* **unfolding** *1* **by** (*simp add*: *assms*(*2*))
qed

context *deep-model-correct-params-y*
begin

definition *witness-submatrix* **where**
witness-submatrix f = *submatrix* (*A*′ *f*) *rows-with-1 rows-with-1*

lemma *polyfun-tensor-deep-model*:
assumes *is* ◁ *input-sizes* (*deep-model-l rs*)
shows *polyfun* {..<*weight-space-dim*}
(λ*f*. *Tensor.lookup* (*tensors-from-net* (*insert-weights* (*deep-model-l rs*) *f*) $ *y*) *is*)
proof −

have *1*:$\bigwedge f$. *remove-weights* (*insert-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*
then have *y* < *output-size* (*deep-model-l rs*) **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*
have *0*:{..<*weight-space-dim*} = *set* [*0*..<*weight-space-dim*] **by** *auto*
then show *?thesis* **unfolding** *weight-space-dim-def* **using** *polyfun-tensors-from-net assms*(*1*) *valid-deep-model*
‹*y* < *output-size* (*deep-model-l rs*)› **by** *metis*
qed

lemma *input-sizes-deep-model*: *input-sizes* (*deep-model-l rs*) = *replicate* (*2* ∗ *N-half*) (*last rs*)
unfolding *N-half-def* **using** *input-sizes-deep-model deep*
by (*metis* (*no-types*, *lifting*) *Nitpick.size-list-simp*(*2*) *One-nat-def Suc-1 Suc-le-lessD diff-Suc-Suc length-tl less-imp-le-nat list.size*(*3*) *not-less-eq numeral-3-eq-3 realpow-num-eq-if*)

lemma *polyfun-matrix-deep-model*:
assumes *i*<(*last rs*) ^ *N-half*
assumes *j*<(*last rs*) ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (λ*f*. *A'* *f* $$ (*i*,*j*))
proof −
have *0*:*y* < *output-size* (*deep-model-l rs*) **using** *valid-deep-model y-valid length-output-deep-model* **by** *force*
have *1*:$\bigwedge f$. *remove-weights* (*insert-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*
have *2*:($\bigwedge f$ *is*. *is* ◁ *replicate* (*2* ∗ *N-half*) (*last rs*) ⟹
polyfun {..<*weight-space-dim*} (λ*x*. *Tensor.lookup* (*A x*) *is*))
unfolding *A-def* **using** *polyfun-tensor-deep-model*[*unfolded input-sizes-deep-model*] *0* **by** *blast*
show *?thesis*
unfolding *A'-def A-def* **apply** (*rule polyfun-matricize*)
using *dims-tensor-deep-model*[*OF 1*] *2*[*unfolded A-def*]
using *dims-A'-pow*[*unfolded A'-def A-def*] ‹*i*<(*last rs*) ^ *N-half*› ‹*j*<(*last rs*) ^ *N-half*›
by *auto*
qed

lemma *polyfun-submatrix-deep-model*:
assumes *i* < *r* ^ *N-half*
assumes *j* < *r* ^ *N-half*
shows *polyfun* {..<*weight-space-dim*} (λ*f*. *witness-submatrix f* $$ (*i*,*j*))
unfolding *witness-submatrix-def*
proof (*rule polyfun-submatrix*)
have *1*:$\bigwedge f$. *remove-weights* (*insert-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*
show $\bigwedge f$. *A'* *f* ∈ *carrier-mat* ((*last rs*) ^ *N-half*) ((*last rs*) ^ *N-half*)

using *1 dims-A′-pow* **using** *weight-space-dim-def* **by** *auto*
show $\bigwedge f\ i\ j.\ i < last\ rs\ \hat{}\ N\text{-}half \Longrightarrow j < last\ rs\ \hat{}\ N\text{-}half \Longrightarrow$
polyfun {..<*weight-space-dim*} ($\lambda f.\ A'\ f\ \$\$\ (i, j)$)
using *polyfun-matrix-deep-model weight-space-dim-def* **by** *force*
show $i < card\ \{i.\ i < last\ rs\ \hat{}\ N\text{-}half \wedge i \in rows\text{-}with\text{-}1\}$
using *assms*(*1*) *card-rows-with-1 dims-Aw′-pow set-le-in* **by** *metis*
show $j < card\ \{i.\ i < last\ rs\ \hat{}\ N\text{-}half \wedge i \in rows\text{-}with\text{-}1\}$
using *assms*(*2*) *card-rows-with-1 dims-Aw′-pow set-le-in* **by** *metis*
show *infinite rows-with-1 infinite rows-with-1* **by** (*simp-all add*: *infinite-rows-with-1*)
qed

lemma *polyfun-det-deep-model*:
shows *polyfun* {..<*weight-space-dim*} ($\lambda f.\ det\ (witness\text{-}submatrix\ f)$)
proof (*rule polyfun-det*)
fix *f*
have *remove-weights* (*insert-weights* (*deep-model-l rs*) *f*) = *deep-model-l rs*
using *remove-insert-weights* **by** *metis*

show *witness-submatrix f* ∈ *carrier-mat* (*r* ^ *N-half*) (*r* ^ *N-half*)
unfolding *witness-submatrix-def* **apply** (*rule carrier-matI*) **unfolding** *dim-submatrix*[*unfolded set-le-in*]
unfolding *dims-A′-pow*[*unfolded weight-space-dim-def*] **using** *card-rows-with-1 dims-Aw′-pow* **by** *simp-all*
show $\bigwedge i\ j.\ i < r\ \hat{}\ N\text{-}half \Longrightarrow j < r\ \hat{}\ N\text{-}half \Longrightarrow$ *polyfun* {..<*weight-space-dim*} ($\lambda f.$ *witness-submatrix f* $\$\$\ (i, j)$)
using *polyfun-submatrix-deep-model* **by** *blast*
qed

end

end

34 Alternative Lebesgue Measure Definition

theory *Lebesgue-Functional*
imports *HOL−Analysis.Lebesgue-Measure HOL.Topological-Spaces*
begin

Lebesgue_Measure.lborel is defined on the typeclass euclidean_space, which does not allow the space dimension to be dependent on a variable. As the Lebesgue measure of higher dimensions is the product measure of the one dimensional Lebesgue measure, we can easily define a more flexible version of the Lebesgue measure as follows. This version of the Lebesgue measure measures sets of functions from nat to real whose values are undefined for arguments higher than n. These "Extensional Function Spaces" are defined in HOL/Library/FuncSet.

definition *lborel-f* :: *nat* ⇒ (*nat* ⇒ *real*) *measure* **where**
lborel-f n = ($\Pi_M\ b$∈{..<*n*}. *lborel*)

lemma *product-sigma-finite-interval*: *product-sigma-finite* ($\lambda b.$ *interval-measure* ($\lambda x.$ *x*))
unfolding *product-sigma-finite-def* **using** *sigma-finite-interval-measure* **by** *auto*

lemma *l-borel-f-1*: *distr* (*lborel-f 1*) *lborel* ($\lambda x.$ *x 0*) = *lborel*
unfolding *lborel-f-def*
using *product-sigma-finite.distr-singleton*[*OF product-sigma-finite-interval*, *of 0*]
lborel-eq-real lessThan-Suc **by** *auto*

lemma *space-lborel-f*: *space* (*lborel-f n*) = Pi_E {..<*n*} (λ-. *UNIV*) **unfolding** *lborel-f-def*
unfolding *space-PiM space-lborel space-borel* **by** *metis*

lemma *space-lborel-f-subset*: *space* (*lborel-f n*) ⊆ *space* (*lborel-f* (*Suc n*))
unfolding *space-lborel-f* **by** (*rule subsetI*, *rule PiE-I*, *blast*,
metis PiE-E Suc-n-not-le-n le-cases lessThan-subset-iff subsetCE)

lemma *space-lborel-add-dim*:
assumes *f* ∈ *space* (*lborel-f n*)
shows *f*(*n*:=*x*) ∈ *space* (*lborel-f* (*Suc n*))
unfolding *space-lborel-f*
using *assms lessThan-Suc space-lborel-f* **by** *auto*

lemma *integral-lborel-f*:
assumes *f* ∈ *borel-measurable* (*lborel-f* (*Suc n*))
shows *integral*^{N} (*lborel-f* (*Suc n*)) *f* = $\int^+ y.$ $\int^+ x.$ *f* (*x*(*n* := *y*)) *∂lborel-f n* *∂lborel*
unfolding *lborel-f-def*
using *product-sigma-finite.product-nn-integral-insert-rev*[*OF product-sigma-finite-interval*, *of* {..<*n*}, *OF finite-lessThan* -]
assms[*unfolded lborel-f-def*] *lborel-eq-real* **by** (*simp add*: *lessThan-Suc*)

lemma *emeasure-lborel-f-Suc*:
assumes *A* ∈ *sets* (*lborel-f* (*Suc n*))
assumes ⋀*y*. {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *A*} ∈ *sets* (*lborel-f n*)
shows *emeasure* (*lborel-f* (*Suc n*)) *A* = $\int^+ y.$ *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *A*} *∂lborel*
proof −
{
fix *x y* **assume** *x*∈*space* (*lborel-f n*)
then have (*indicator A*) (*x*(*n* := *y*)) = (*indicator* {*x*∈*space* (*lborel-f n*). *x*(*n* := *y*) ∈ *A*}) *x* **using** *indicator-def*
by (*metis* (*no-types*, *lifting*) *mem-Collect-eq*)
}
then show *?thesis*
unfolding *nn-integral-indicator*[*OF assms*(*1*), *symmetric*] *nn-integral-indicator*[*OF assms*(*2*), *symmetric*]
integral-lborel-f[*OF borel-measurable-indicator*, *OF assms*(*1*)]

using *nn-integral-cong* **by** (*metis* (*no-types*, *lifting*))
qed

lemma *lborel-f-measurable-add-dim*: $(\lambda f.\ f(n := x)) \in$ *measurable* (*lborel-f* n) (*lborel-f* (*Suc* n))
proof −
have $x \in$ *space lborel* **by** *simp*
have *0*:$(\lambda(f,\ y).\ f(n := y)) \circ (\lambda xa.\ (xa,\ x)) = (\lambda f.\ f(n := x))$ **unfolding** *comp-def* **using** *case-prod-conv* **by** *fast*
show *?thesis* **unfolding** *lborel-f-def*
using *measurable-comp*[*OF measurable-Pair2'*[*of x lborel* Pi_M {..<n} ($\lambda b.$ *lborel*), *OF* ‹$x \in$ *space lborel*›]
measurable-add-dim[*of n* {..<n} $\lambda b.$ *lborel*], *unfolded 0*] *lessThan-Suc* **by** *auto*
qed

lemma *sets-lborel-f-sub-dim*:
assumes $A \in$ *sets* (*lborel-f* (*Suc* n))
shows $\{x.\ x(n := y) \in A\} \cap$ *space* (*lborel-f* n) $\in$ *sets* (*lborel-f* n)
proof −
have $(\lambda f.\ f(n := y)) -` A \cap$ *space* (*lborel-f* n) $\in$ *sets* (*lborel-f* n)
using *measurable-sets*[*OF lborel-f-measurable-add-dim assms*] **by** *auto*
moreover have $(\lambda f.\ f(n := y)) -` A = \{x.\ x(n := y) \in A\}$ **by** *auto*
finally show *?thesis* **by** *metis*
qed

lemma *lborel-f-measurable-restrict*:
assumes $m \leq n$
shows $(\lambda f.$ *restrict* f {..<m}) $\in$ *measurable* (*lborel-f* n) (*lborel-f* m)
using *measurable-restrict-subset lborel-f-def assms* **by** *auto*

lemma *lborel-measurable-sub-dim*: $(\lambda f.$ *restrict* f {..<n}) $\in$ *measurable* (*lborel-f* (*Suc* n)) (*lborel-f* n)
using *lborel-f-measurable-restrict*[*of n Suc n*] **by** *linarith*

lemma *measurable-lborel-component* [*measurable*]:
assumes k<n
shows $(\lambda x.\ x\ k) \in$ *borel-measurable* (*lborel-f* n)
unfolding *lborel-f-def* **using** *assms measurable-PiM-component-rev* **by** *simp-all*

end

theory *PP-Univariate*
imports *PP-MPoly PP-More-MPoly HOL−Computational-Algebra.Polynomial*
begin

This file connects univariate MPolys to the theory of univariate polynomials from `~~/src/HOL/Computational_Algebra/Polynomial.thy`.

definition *poly-to-mpoly*::*nat* ⇒ *'a*::*comm-monoid-add poly* ⇒ *'a mpoly*
where *poly-to-mpoly v p* = *MPoly* (*Abs-poly-mapping* ($\lambda m.$ (*coeff p* (*PP-Poly-Mapping.lookup*

m v)) *when PP-Poly-Mapping.keys m* ⊆ {*v*}))

lemma *poly-to-mpoly-finite*: *finite* {*m*::*nat* ⇒$_0$ *nat*. (*coeff p* (*PP-Poly-Mapping.lookup m v*) *when PP-Poly-Mapping.keys m* ⊆ {*v*}) ≠ *0*} (**is** *finite ?M*)
proof −
have *?M* ⊆ *PP-Poly-Mapping.single v* ‘ {*x*. *Polynomial.coeff p x* ≠ *0*}
proof
fix *m* **assume** *m* ∈ *?M*
then have ⋀*v'*. *v'*≠*v* ⟹ *PP-Poly-Mapping.lookup m v'* = *0* **by** *fastforce*
then have *m* = *PP-Poly-Mapping.single v* (*PP-Poly-Mapping.lookup m v*)
using *PP-Poly-Mapping.poly-mapping-eqI* **by** (*metis* (*full-types*) *lookup-single-eq lookup-single-not-eq*)
then show *m* ∈ (*PP-Poly-Mapping.single v*) ‘ {*x*. *Polynomial.coeff p x* ≠ *0*}
using ‹*m* ∈ *?M*› **by** *auto*
qed
then show *?thesis* **using** *finite-surj*[*OF MOST-coeff-eq-0*[*unfolded eventually-cofinite*]] **by** *blast*
qed

lemma *coeff-poly-to-mpoly*: *PP-MPoly.coeff* (*poly-to-mpoly v p*) (*PP-Poly-Mapping.single v k*) = *Polynomial.coeff p k*
unfolding *poly-to-mpoly-def coeff-def MPoly-inverse*[*OF Set.UNIV-I*] *lookup-Abs-poly-mapping*[*OF poly-to-mpoly-finite*]
using *empty-subsetI keys-single lookup-single order-refl when-simps*(*1*) **by** *simp*

definition *mpoly-to-poly*::*nat* ⇒ *'a*::*comm-monoid-add mpoly* ⇒ *'a poly*
where *mpoly-to-poly v p* = *Abs-poly* (λ*k*. *PP-MPoly.coeff p* (*PP-Poly-Mapping.single v k*))

lemma *coeff-mpoly-to-poly*[*simp*]: *Polynomial.coeff* (*mpoly-to-poly v p*) *k* = *PP-MPoly.coeff p* (*PP-Poly-Mapping.single v k*)
proof −
have *0*:*PP-Poly-Mapping.single v* ‘ {*x*. *PP-Poly-Mapping.lookup* (*mapping-of p*) (*PP-Poly-Mapping.single v x*) ≠ *0*}
⊆ {*k*. *PP-Poly-Mapping.lookup* (*mapping-of p*) *k* ≠ *0*}
by *auto*
have ∀$_\infty$ *k*. *PP-MPoly.coeff p* (*PP-Poly-Mapping.single v k*) = *0* **unfolding** *coeff-def eventually-cofinite*
using *finite-imageD*[*OF finite-subset*[*OF 0 PP-Poly-Mapping.finite-lookup*]] *inj-single* **by** (*metis inj-eq inj-onI*)
then show *?thesis*
unfolding *mpoly-to-poly-def* **by** (*simp add*: *Abs-poly-inverse*)
qed

lemma *mpoly-to-poly-inverse*:
assumes *vars p* ⊆ {*v*}
shows *poly-to-mpoly v* (*mpoly-to-poly v p*) = *p*
proof −
def *f* == (λ*m*. *Polynomial.coeff* (*mpoly-to-poly v p*) (*PP-Poly-Mapping.lookup*

```
m v) when PP-Poly-Mapping.keys m ⊆ {v})
  have finite {m. f m ≠ 0} unfolding f-def using poly-to-mpoly-finite by blast
  have Abs-poly-mapping f = mapping-of p
  proof (rule PP-Poly-Mapping.poly-mapping-eqI)
    fix m
   show PP-Poly-Mapping.lookup (Abs-poly-mapping f) m = PP-Poly-Mapping.lookup
(mapping-of p) m
    proof (cases PP-Poly-Mapping.keys m ⊆ {v})
      assume PP-Poly-Mapping.keys m ⊆ {v}
     then show ?thesis unfolding PP-Poly-Mapping.lookup-Abs-poly-mapping[OF
‹finite {m. f m ≠ 0}›] unfolding f-def
        unfolding coeff-mpoly-to-poly coeff-def using when-simps(1) apply simp
        using keys-single lookup-not-eq-zero-eq-in-keys lookup-single-eq
        lookup-single-not-eq poly-mapping-eqI subset-singletonD
        by (metis (no-types, lifting) aux lookup-eq-zero-in-keys-contradict)
    next
      assume ¬PP-Poly-Mapping.keys m ⊆ {v}
     then show ?thesis unfolding PP-Poly-Mapping.lookup-Abs-poly-mapping[OF
‹finite {m. f m ≠ 0}›] unfolding f-def
       using ‹vars p ⊆ {v}› unfolding vars-def by (metis (no-types, lifting) UN-I
lookup-not-eq-zero-eq-in-keys subsetCE subsetI when-def)
    qed
  qed
  then show ?thesis
    unfolding poly-to-mpoly-def f-def  by (simp add: mapping-of-inverse)
qed

lemma poly-to-mpoly-inverse: mpoly-to-poly v (poly-to-mpoly v p) = p
  unfolding mpoly-to-poly-def coeff-poly-to-mpoly by (simp add: coeff-inverse)

lemma poly-to-mpoly0: poly-to-mpoly v 0 = 0
proof −
  have ⋀m. (Polynomial.coeff 0 (PP-Poly-Mapping.lookup m v) when PP-Poly-Mapping.keys
m ⊆ {v}) = 0 by simp
  have Abs-poly-mapping (λm. Polynomial.coeff 0 (PP-Poly-Mapping.lookup m v)
when PP-Poly-Mapping.keys m ⊆ {v}) = 0
   apply (rule PP-Poly-Mapping.poly-mapping-eqI) unfolding lookup-Abs-poly-mapping[OF
poly-to-mpoly-finite] by auto
  then show ?thesis using poly-to-mpoly-def zero-mpoly.abs-eq by (metis (no-types))
qed

lemma mpoly-to-poly-add: mpoly-to-poly v (p1 + p2) = mpoly-to-poly v p1 +
mpoly-to-poly v p2
  unfolding Polynomial.plus-poly.abs-eq PP-More-MPoly.coeff-add coeff-mpoly-to-poly
  using mpoly-to-poly-def by auto

lemma poly-eq-insertion:
assumes vars p ⊆ {v}
shows poly (mpoly-to-poly v p) x = insertion (λv. x) p
```

using *assms* **proof** (*induction p rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*cases a=0*)
case *True*
then show *?thesis*
by (*metis PP-MPoly.monom.abs-eq insertion-zero monom-zero poly-0 poly-to-mpoly0 poly-to-mpoly-inverse single-zero*)
next
case *False*
then have *PP-Poly-Mapping.keys m* ⊆ {*v*} **using** *monom* **unfolding** *vars-def PP-MPoly.mapping-of-monom keys-single* **by** *simp*
then have ⋀*v'*. *v'*≠*v* ⟹ *PP-Poly-Mapping.lookup m v' = 0* **unfolding** *vars-def* **by** *auto*
then have *m = PP-Poly-Mapping.single v* (*PP-Poly-Mapping.lookup m v*)
by (*metis lookup-single-eq lookup-single-not-eq poly-mapping-eqI*)
then have *0*:*insertion* (λ*v*. *x*) (*PP-MPoly.monom m a*) = *a* ∗ *x* ^ (*PP-Poly-Mapping.lookup m v*)
using *insertion-single* **by** *metis*
have ⋀*k*. *PP-Poly-Mapping.single v k = m* ⟷ *PP-Poly-Mapping.lookup m v = k*
using ‹*m = PP-Poly-Mapping.single v* (*PP-Poly-Mapping.lookup m v*)› **by** *auto*
then have *monom a* (*PP-Poly-Mapping.lookup m v*) = (*Abs-poly* (λ*k*. *if PP-Poly-Mapping.single v k = m then a else 0*))
by (*simp add*: *Polynomial.monom.abs-eq*)
then show *?thesis* **unfolding** *mpoly-to-poly-def PP-More-MPoly.coeff-monom 0 when-def* **by** (*metis poly-monom*)
qed
next
case (*sum p1 p2 m a*)
then have *poly* (*mpoly-to-poly v p1*) *x = insertion* (λ*v*. *x*) *p1*
poly (*mpoly-to-poly v p2*) *x = insertion* (λ*v*. *x*) *p2*
by (*simp-all add*: *vars-add-monom*)
then show *?case* **unfolding** *insertion-add mpoly-to-poly-add* **by** *simp*
qed

Using the new connection between MPoly and univariate polynomials, we can transfer:

lemma *univariate-mpoly-roots-finite*:
fixes *p*::′*a*::*idom mpoly*
assumes *vars p* ⊆ {*v*} *p* ≠ *0*
shows *finite* {*x*. *insertion* (λ*v*. *x*) *p = 0*}
using *poly-roots-finite*[*of mpoly-to-poly v p*, *unfolded poly-eq-insertion*[*OF* ‹*vars p* ⊆ {*v*}›]]
using *assms*(*1*) *assms*(*2*) *mpoly-to-poly-inverse poly-to-mpoly0* **by** *fastforce*

end

35 Lebesgue Measure of Polynomial Zero Sets

theory *Lebesgue-Zero-Set*
imports *PP-MPoly PP-More-MPoly Lebesgue-Functional PP-Univariate*
begin

lemma *measurable-insertion* [*measurable*]:
assumes *vars p* ⊆ {..<*n*}
shows (λ*f*. *insertion f p*) ∈ *borel-measurable* (*lborel-f n*)
using *assms* **proof** (*induction p rule*:*mpoly-induct*)
case (*monom m a*)
then show *?case*
proof (*cases a* = *0*)
case *True*
show *?thesis* **unfolding** *insertion-single* ‹*a* = *0*› *PP-MPoly.monom.abs-eq single-zero*
zero-mpoly.abs-eq[*symmetric*] *insertion-zero* **by** *measurable*
next
case *False*
have *PP-Poly-Mapping.keys m* ⊆ {..<*n*} **using** *monom* **by** (*simp add*: *False vars-monom-keys*)
then show *?thesis* **using** ‹*a*≠*0*›
proof (*induction m arbitrary*:*a rule*:*poly-mapping-induct*)
case (*single x i a*)
then show *?case*
proof (*cases i* = *0*)
case *True*
show *?thesis* **unfolding** *insertion-single* ‹*i* = *0*› **by** *simp*
next
case *False*
then show *?thesis* **unfolding** *insertion-single* **apply** *measurable*
using *vars-monom-single-cases single False insert-subset lessThan-iff* ‹*a*≠*0*› **by** *fastforce*
qed
next
case (*sum m1 m2 x i*)
then have *PP-Poly-Mapping.keys m1* ∩ *PP-Poly-Mapping.keys m2* = {} **by** *simp*
then have *PP-Poly-Mapping.keys m1* ∪ *PP-Poly-Mapping.keys m2* = *PP-Poly-Mapping.keys* (*m1* + *m2*) **using** *keys-add* **by** *metis*
then have *1*:*PP-Poly-Mapping.keys m1* ⊆ {..<*n*} **and** *2*:*PP-Poly-Mapping.keys m2* ⊆ {..<*n*} **using** *sum.prems* **by** *auto*
show *?case* **unfolding** *PP-MPoly.mult-monom*[*of m1 a m2 1*,*simplified*,*symmetric*] *insertion-mult* **using** *sum.IH*(*1*)[*OF 1* ‹*a*≠*0*›] *sum.IH*(*2*)[*OF 2*, *of 1*, *simplified*] **by** *measurable*
qed
qed
next
case (*sum p1 p2 m a*)

then have $(\lambda f.\ insertion\ f\ p1) \in borel\text{-}measurable\ (lborel\text{-}f\ n)$
$(\lambda f.\ insertion\ f\ p2) \in borel\text{-}measurable\ (lborel\text{-}f\ n)$
using *vars-add-monom*[*OF sum.hyps*] *le-sup-iff* **by** *blast+*
then show *?case* **unfolding** *insertion-add* **by** *measurable*
qed

This proof follows Richard Caron and Tim Traynor, "The zero set of a polynomial" http://www1.uwindsor.ca/math/sites/uwindsor.ca.math/files/05-03.pdf

lemma *lebesgue-mpoly-zero-set*:
fixes *p*::*real mpoly*
assumes $p \neq 0$ *vars* $p \subseteq \{..<n\}$
shows $\{f \in space\ (lborel\text{-}f\ n).\ insertion\ f\ p = 0\} \in null\text{-}sets\ (lborel\text{-}f\ n)$
using *assms* **proof** (*induction n arbitrary*:*p*)
case *0*
then have *vars p* = {} **by** *simp* **then have** $\bigwedge f.\ insertion\ f\ p = PP\text{-}MPoly.coeff\ p\ 0$
unfolding *insertion-trivial*[*symmetric*] **using** *insertion-irrelevant-vars* **by** *blast*
have $\bigwedge m.\ m \neq 0 \Longrightarrow PP\text{-}MPoly.coeff\ p\ m = 0$
proof (*rule ccontr*)
fix *m*::*nat* ⇒$_0$ *nat* **assume** $m \neq 0$ *PP-MPoly.coeff p m* ≠ *0*
then obtain *v* **where** *PP-Poly-Mapping.lookup m v* ≠ *0* **using** *aux* **by** *auto*
then have $v \in vars\ p$ **unfolding** *PP-More-MPoly.vars-def* **using** ‹*PP-MPoly.coeff p m* ≠ *0*›
by (*meson UN-I coeff-keys lookup-not-eq-zero-eq-in-keys*)
then show *False* **using** ‹*vars p* = {}› **by** *auto*
qed
then have *PP-MPoly.coeff p 0* ≠ *0* **using** ‹*p* ≠ *0*›
by (*metis coeff-all-0*)
then have $\{f.\ insertion\ f\ p = 0\} = \{\}$ **using** ‹$\bigwedge f.\ insertion\ f\ p = PP\text{-}MPoly.coeff\ p\ 0$› **by** *auto*
then show *?case* **by** *auto*
next
case (*Suc n p*)

Show that N is finite:

then have *extract-var p n* ≠ *0* **using** *reduce-nested-mpoly-0*
by (*metis reduce-nested-mpoly-extract-var*)
let *?q* = $\lambda j.$ *PP-MPoly.coeff* (*extract-var p n*) *j*
obtain *j* **where** *?q j* ≠ *0* **using** ‹*extract-var p n* ≠ *0*›
by (*metis coeff-all-0*)
then have *finite* $\{x.\ insertion\ (\lambda\text{-}.\ x)\ (?q\ j) = 0\}$
using *univariate-mpoly-roots-finite*[*OF vars-coeff-extract-var*] **by** *metis*
then have *finite* $(\bigcap j.\ \{x.\ insertion\ (\lambda\text{-}.\ x)\ (?q\ j) = 0\})$ **by** *auto*
moreover have $\{x.\ \forall j.\ insertion\ (\lambda\text{-}.\ x)\ (?q\ j) = 0\} = (\bigcap j.\ \{x.\ insertion\ (\lambda v.\ x)\ (?q\ j) = 0\})$ **by** *blast*
ultimately have *finite* $\{x.\ \forall j.\ insertion\ (\lambda\text{-}.\ x)\ (?q\ j) = 0\}$ **by** *metis*

def *p-fix1* == $\lambda x1.$ *replace-coeff* (*insertion* (λ-. *x1*)) (*extract-var p n*)
def $N == \{x1.$ *p-fix1 x1* = *0*$\}$
have $N \subseteq \{x. \forall j.$ *insertion* (λ-. *x*) (*?q j*) = *0*$\}$
proof
fix *x* **assume** $x \in N$
then have *p-fix1 x* = *0* **using** *N-def* **by** *auto*
then have $\bigwedge m.$ *PP-MPoly.coeff* (*p-fix1 x*) *m* = *0* **by** (*metis PP-More-MPoly.coeff-monom monom-zero when-def*)
have $\bigwedge j.$ *insertion* (λ-. *x*) (*?q j*) = *0*
using ‹$\bigwedge m.$ *PP-MPoly.coeff* (*p-fix1 x*) *m* = *0*›[*unfolded p-fix1-def coeff-replace-coeff*[*of insertion* (λ-. *x*), *OF insertion-zero*]]
by *metis*
then show *x* ∈ $\{x. \forall j.$ *insertion* (λ-. *x*) (*PP-MPoly.coeff* (*extract-var p n*) *j*) = *0*$\}$ **by** *blast*
qed
then have *finite N* **by** (*simp add*: ‹*finite* $\{x. \forall j.$ *insertion* (λ-. *x*) (*PP-MPoly.coeff* (*extract-var p n*) *j*) = *0*$\}$› *finite-subset*)

Use the IH:

def *A* == $\{f \in$ *space* (*lborel-f* (*Suc n*)). *insertion f p* = *0*$\}$

have $\bigwedge x1.$ *vars* (*p-fix1 x1*) ⊆ $\{..<n\}$
proof −
fix *x1*
have *vars* (*extract-var p n*) ⊆ $\{..<n\}$
using ‹*vars p* ⊆ $\{..<$*Suc n*$\}$› *lessThan-Suc v-not-in-vars-extract-var vars-extract-var-subset* **by** *fastforce*
then show *vars* (*p-fix1 x1*) ⊆ $\{..<n\}$ **unfolding** *p-fix1-def*
using *vars-replace-coeff*[*of insertion* (λ-. *x1*), *OF insertion-zero*] **by** *blast*
qed
have *set-eq*:$\bigwedge x1.$ $\{x \in$ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*$\}$ = $\{f \in$ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*$\}$
proof −
fix *x1*
show $\{x \in$ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*$\}$ = $\{f \in$ *space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*$\}$
proof (*rule subset-antisym*;*rule subsetI*)
fix *x* **assume** *x* ∈ $\{x \in$ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*$\}$
then have *insertion* (*x*(*n* := *x1*)) *p* = *0* *x* ∈ *space* (*lborel-f n*)
using *A-def* **by** *auto*
then have *insertion x* (*p-fix1 x1*) = *0* **unfolding** *p-fix1-def*
unfolding *replace-coeff-extract-var-cong*[*of* λ-. *x1 n x*(*n* := *x1*) *p*, *OF fun-upd-same*[*symmetric*]]
using *insertion-replace-coeff*[*of x*(*n* := *x1*)]
using *insertion-irrelevant-vars*[*of replace-coeff* (*insertion* (*x*(*n* := *x1*))) (*extract-var p n*) *x x*(*n* := *x1*)]
vars-replace-coeff fun-upd-other insertion-zero reduce-nested-mpoly-extract-var subset-eq
v-not-in-vars-extract-var **by** *metis*

then show $x \in \{f{\in}space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\}$ **using** ‹$x \in space\ (lborel\text{-}f\ n)$› **by** *blast*
next
fix f **assume** $f \in \{f{\in}space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\}$
then have $f{\in}space\ (lborel\text{-}f\ n)\ insertion\ f\ (p\text{-}fix1\ x1) = 0$ **by** *auto*
have *insertion* $(f(n := x1))\ p = 0$ **using** ‹*insertion f* $(p\text{-}fix1\ x1) = 0$›[*unfolded p-fix1-def*]
insertion-replace-coeff insertion-irrelevant-vars replace-coeff-extract-var-cong
by (*metis* (*no-types*, *lifting*) ‹*insertion f* $(p\text{-}fix1\ x1) = 0$› ‹*vars* $(p\text{-}fix1\ x1) \subseteq \{..<n\}$›
fun-upd-other fun-upd-same lessThan-iff order-less-irrefl p-fix1-def reduce-nested-mpoly-extract-var subsetCE)
then have $f(n := x1) \in A$ **unfolding** *A-def* **using** *space-lborel-add-dim*
using ‹$f \in space\ (lborel\text{-}f\ n)$› *lborel-f-def mem-Collect-eq* **by** *blast*
then show $f \in \{f{\in}space\ (lborel\text{-}f\ n).\ f(n := x1) \in A\}$ **using** ‹$f \in space\ (lborel\text{-}f\ n)$› **by** *auto*
qed
qed

have $\bigwedge x1.\ x1{\in}N \Longrightarrow \{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} \in sets\ (lborel\text{-}f\ n)$
and *emeasure-in-N*: $\bigwedge x1.\ x1{\in}N \Longrightarrow emeasure\ (lborel\text{-}f\ n)\ \{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} = emeasure\ (lborel\text{-}f\ n)\ (space\ (lborel\text{-}f\ n))$
proof −
fix *x1* **assume** *x1*$\in$*N*
then have *p-fix1 x1 = 0* **using** *N-def* **by** *auto*
then have $\bigwedge f.\ insertion\ f\ (p\text{-}fix1\ x1) = 0$ **using** *insertion-zero* **by** *auto*
then have $\{f \in space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\} = space\ (lborel\text{-}f\ n)$ **by** *simp*
show $\{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} \in sets\ (lborel\text{-}f\ n)$ **unfolding** *set-eq*
by (*simp add*: ‹$\{f \in space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\} = space\ (lborel\text{-}f\ n)$›)
show $emeasure\ (lborel\text{-}f\ n)\ \{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} = emeasure\ (lborel\text{-}f\ n)\ (space\ (lborel\text{-}f\ n))$
unfolding *set-eq*
by (*simp add*: ‹$\{f \in space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\} = space\ (lborel\text{-}f\ n)$›)
qed

have *emeasure-not-in-N*: $\bigwedge x1.\ x1{\notin}N \Longrightarrow emeasure\ (lborel\text{-}f\ n)\ \{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} = 0$
and $\bigwedge x1.\ x1{\notin}N \Longrightarrow \{x{\in}space\ (lborel\text{-}f\ n).\ x(n := x1) \in A\} \in sets\ (lborel\text{-}f\ n)$
proof −
fix *x1* **assume** *x1*$\notin$*N*
then have *p-fix1 x1* $\neq$ *0* **using** *p-fix1-def N-def* **by** *auto*
then have $emeasure\ (lborel\text{-}f\ n)\ \{f{\in}space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\} = 0$
$\{f{\in}space\ (lborel\text{-}f\ n).\ insertion\ f\ (p\text{-}fix1\ x1) = 0\} \in sets\ (lborel\text{-}f\ n)$

using *Suc.IH*[*OF* ‹*p-fix1 x1* ≠ *0*›] ‹⋀*x1*. *vars* (*p-fix1 x1*) ⊆ {..<*n*}› **by** *auto*
then show *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *0*
{*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)
using ‹{*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} ∈ *sets* (*lborel-f n*)›
‹*emeasure* (*lborel-f n*) {*f*∈*space* (*lborel-f n*). *insertion f* (*p-fix1 x1*) = *0*} = *0*›
using *set-eq*
by *auto*
qed

have *N* ∈ *null-sets lborel* **using** ‹*finite N*› *finite-imp-null-set-lborel* **by** *blast*
have *ae-zero*: *AE x1 in lborel*. *emeasure* (*lborel-f n*) {*x* ∈ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *0*
apply (*rule AE-I'*[*OF* ‹*N* ∈ *null-sets lborel*›])
using ‹⋀*x1*. *x1*∉*N* ⟹ *emeasure* (*lborel-f n*) {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} = *0*›
by *force*

have *measurable*: (λ*x1*. *emeasure* (*lborel-f n*) {*x* ∈ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*}) ∈ *borel-measurable lborel*
proof (*rule borel-measurableI*)
let *?f* = (λ*x1*. *emeasure* (*lborel-f n*) {*x* ∈ *space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*})
fix *S*::*ennreal set* **assume** *open S*
have *0*:*0*∈*S* ⟹ − *N* ⊆ *?f* −‘ *S*
using *emeasure-not-in-N* **by** *auto*
have *1*:*emeasure* (*lborel-f n*) (*space* (*lborel-f n*)) ∈ *S* ⟹ *N* ⊆ *?f* −‘ *S*
using *emeasure-in-N* **by** *auto*
have *2*:*0*∉*S* ⟹ *?f* −‘ *S* ⊆ *N* **using** *emeasure-not-in-N* **by** *fastforce*
have *3*:*emeasure* (*lborel-f n*) (*space* (*lborel-f n*)) ∉ *S* ⟹ *?f* −‘ *S* ⊆ −*N* **using** *emeasure-in-N* **by** *auto*
have *?f* −‘ *S* = {} ∨ *?f* −‘ *S* = *N* ∨ *?f* −‘ *S* = *UNIV* ∨ *?f* −‘ *S* = − *N*
apply (*cases 0*∈*S*; *cases emeasure* (*lborel-f n*) (*space* (*lborel-f n*)) ∉ *S*)
using *0 1 2 3* **by** *auto*
then show *?f* −‘ *S* ∩ *space lborel* ∈ *sets lborel*
using ‹*finite N*› *finite-imp-null-set-lborel borel-comp null-setsD2 sets-lborel* **by** *fastforce*
qed

have *A* ∈ *sets* (*lborel-f* (*Suc n*)) **unfolding** *A-def*
using *pred-eq-const1*[*OF measurable-insertion*[*OF* ‹*vars p* ⊆ {..<*Suc n*}›]] *pred-def* **by** *force*
then have *in-sets*: {*f* ∈ *space* (*lborel-f* (*Suc n*)). *insertion f p* = *0*} ∈ *sets* (*lborel-f* (*Suc n*)) **using** *A-def* **by** *metis*
have ⋀*x1*. {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)
using ‹⋀*x1*. *x1*∈*N* ⟹ {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)›
‹⋀*x1*. *x1*∉*N* ⟹ {*x*∈*space* (*lborel-f n*). *x*(*n* := *x1*) ∈ *A*} ∈ *sets* (*lborel-f n*)› **by** *auto*
have *emeasure* (*lborel-f* (*Suc n*)) *A* = ∫⁺ *y*. *emeasure* (*lborel-f n*) {*x* ∈ *space*

(*lborel-f n*). $x(n := y) \in A\}$ *∂lborel*
using *emeasure-lborel-f-Suc*[*OF* ‹$A \in$ *sets* (*lborel-f* (*Suc n*))›]
‹$\bigwedge x1.$ $\{x \in$*space* (*lborel-f n*). $x(n := x1) \in A\} \in$ *sets* (*lborel-f n*)› **by** *blast*
also have ... = *0*
using *nn-integral-0-iff-AE*[*OF measurable*] *ae-zero* **by** *blast*
finally have *emeasure* (*lborel-f* (*Suc n*)) *A* = *0* **by** *auto*
then show *?case* **unfolding** *null-sets-def* **using** *in-sets A-def* **by** *blast*
qed

end

36 Rank and Submatrices

theory *DL-Rank-Submatrix*
imports *DL-Rank DL-Submatrix DL-Missing-Matrix*
begin

lemma *row-submatrix-UNIV*:
assumes *i* < *card* $\{i.$ *i* < *dim-row A* ∧ *i* ∈ *I*$\}$
shows *row* (*submatrix A I UNIV*) *i* = *row A* (*pick I i*)
proof (*rule eq-vecI*)
show *dim-eq*:*dim-vec* (*row* (*submatrix A I UNIV*) *i*) = *dim-vec* (*row A* (*pick I i*))
unfolding *carrier-vecD*[*OF row-carrier*] *dim-submatrix* **by** *auto*
fix *j* **assume** *j* < *dim-vec* (*row A* (*pick I i*))
then have *j* < *dim-col* (*submatrix A I UNIV*) *j* < *dim-col A j* < *card* $\{j.$ *j* < *dim-col A* ∧ *j* ∈ *UNIV*$\}$ **using** *dim-eq* **by** *auto*
show *row* (*submatrix A I UNIV*) *i* $ *j* = *row A* (*pick I i*) $ *j*
unfolding *row-def index-vec*[*OF* ‹*j* < *dim-col* (*submatrix A I UNIV*)›] *index-vec*[*OF* ‹*j* < *dim-col A*›]
using *submatrix-index*[*OF assms* ‹*j* < *card* $\{j.$ *j* < *dim-col A* ∧ *j* ∈ *UNIV*$\}$›]
using *pick-UNIV* **by** *auto*
qed

lemma *distinct-cols-submatrix-UNIV*:
assumes *distinct* (*cols* (*submatrix A I UNIV*))
shows *distinct* (*cols A*)
using *assms* **proof** (*rule contrapos-pp*)
assume ¬ *distinct* (*cols A*)
then obtain *i j* **where** *i* < *dim-col A j* < *dim-col A* (*cols A*)!*i* = (*cols A*)!*j* *i*≠*j*
using *distinct-conv-nth cols-length* **by** *metis*
have *i* < *dim-col* (*submatrix A I UNIV*) *j* < *dim-col* (*submatrix A I UNIV*)
unfolding *dim-submatrix* **using** ‹*i* < *dim-col A*› ‹*j* < *dim-col A*›**by** *simp-all*
then have *i* < *length* (*cols* (*submatrix A I UNIV*)) *j* < *length* (*cols* (*submatrix A I UNIV*))
unfolding *cols-length* **by** *simp-all*
have (*cols* (*submatrix A I UNIV*))!*i* = (*cols* (*submatrix A I UNIV*))!*j*
proof (*rule eq-vecI*)
show *dim-vec* (*cols* (*submatrix A I UNIV*) ! *i*) = *dim-vec* (*cols* (*submatrix A*

I UNIV) ! *j*)
by (*simp add*: ‹*i* < *dim-col* (*submatrix A I UNIV*)› ‹*j* < *dim-col* (*submatrix A I UNIV*)›)
fix *k* **assume** *k* < *dim-vec* (*cols* (*submatrix A I UNIV*) ! *j*)
then have *k* < *dim-row* (*submatrix A I UNIV*)
using ‹*j* < *length* (*cols* (*submatrix A I UNIV*))› **by** *auto*
then have *k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*} **using** *dim-submatrix*(*1*)
by *metis*
have *i-transfer*:*cols* (*submatrix A I UNIV*) ! *i* $ *k* = (*cols A*) ! *i* $ (*pick I k*)
unfolding *cols-nth*[*OF* ‹*i* < *dim-col* (*submatrix A I UNIV*)›] *col-def index-vec*[*OF* ‹*k* < *dim-row* (*submatrix A I UNIV*)›]
unfolding *submatrix-index*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}› ‹*i* < *dim-col* (*submatrix A I UNIV*)›[*unfolded dim-submatrix*]]
unfolding *pick-UNIV cols-nth*[*OF* ‹*i* < *dim-col A*›] *col-def index-vec*[*OF pick-le*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}›]]
by *metis*
have *j-transfer*:*cols* (*submatrix A I UNIV*) ! *j* $ *k* = (*cols A*) ! *j* $ (*pick I k*)
unfolding *cols-nth*[*OF* ‹*j* < *dim-col* (*submatrix A I UNIV*)›] *col-def index-vec*[*OF* ‹*k* < *dim-row* (*submatrix A I UNIV*)›]
unfolding *submatrix-index*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}› ‹*j* < *dim-col* (*submatrix A I UNIV*)›[*unfolded dim-submatrix*]]
unfolding *pick-UNIV cols-nth*[*OF* ‹*j* < *dim-col A*›] *col-def index-vec*[*OF pick-le*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}›]]
by *metis*
show *cols* (*submatrix A I UNIV*) ! *i* $ *k* = *cols* (*submatrix A I UNIV*) ! *j* $ *k*
using ‹*cols A* ! *i* = *cols A* ! *j*› *i-transfer j-transfer* **by** *auto*
qed
then show ¬ *distinct* (*cols* (*submatrix A I UNIV*)) **unfolding** *distinct-conv-nth*
using ‹*i* < *length* (*cols* (*submatrix A I UNIV*))› ‹*j* < *length* (*cols* (*submatrix A I UNIV*))› ‹*i* ≠ *j*› **by** *blast*
qed

lemma *cols-submatrix-subset*: *set* (*cols* (*submatrix A UNIV J*)) ⊆ *set* (*cols A*)
proof
fix *c* **assume** *c* ∈ *set* (*cols* (*submatrix A UNIV J*))
then obtain *j* **where** *j* < *length* (*cols* (*submatrix A UNIV J*)) *cols* (*submatrix A UNIV J*) ! *j* = *c*
by (*meson in-set-conv-nth*)
then have *j* < *dim-col* (*submatrix A UNIV J*) **by** *simp*
then have *j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*} **by** (*simp add*: *dim-submatrix*(*2*))
have *cols* (*submatrix A UNIV J*) ! *j* = *cols A* ! (*pick J j*)
unfolding *cols-nth*[*OF* ‹*j* < *dim-col* (*submatrix A UNIV J*)›] *cols-nth*[*OF pick-le*[*OF* ‹*j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}›]]
proof (*rule eq-vecI*)
show *dim-vec* (*col* (*submatrix A UNIV J*) *j*) = *dim-vec* (*col A* (*pick J j*))
unfolding *dim-col dim-submatrix* **by** *auto*
fix *i* **assume** *i* < *dim-vec* (*col A* (*pick J j*))
then have *i* < *dim-row A* **by** *simp*
then have *i* < *dim-row* (*submatrix A UNIV J*) **using** ‹*dim-vec* (*col* (*submatrix*

A UNIV J) *j*) = *dim-vec* (*col A* (*pick J j*))› **by** *auto*
show *col* (*submatrix A UNIV J*) *j* $ *i* = *col A* (*pick J j*) $ *i*
unfolding *col-def index-vec*[*OF* ‹*i* < *dim-row* (*submatrix A UNIV J*)›] *index-vec*[*OF* ‹*i* < *dim-row A*›]
using *submatrix-index* **by** (*metis* (*no-types*, *lifting*) ‹*dim-vec* (*col* (*submatrix A UNIV J*) *j*) = *dim-vec* (*col A* (*pick J j*))›
‹*i* < *dim-vec* (*col A* (*pick J j*))› ‹*j* < *dim-col* (*submatrix A UNIV J*)› *dim-col dim-submatrix*(*1*) *dim-submatrix*(*2*) *pick-UNIV*)
qed
then show *c* ∈ *set* (*cols A*)
using ‹*cols* (*submatrix A UNIV J*) ! *j* = *c*›
using *pick-le*[*OF* ‹*j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}›] **by** (*metis cols-length nth-mem*)
qed

lemma (**in** *vec-space*) *lin-dep-submatrix-UNIV*:
assumes *A* ∈ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols* (*submatrix A I UNIV*))
shows *LinearCombinations.module.lin-dep class-ring* (*module-vec TYPE*('*a*) (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*})) (*set* (*cols* (*submatrix A I UNIV*)))
(**is** *LinearCombinations.module.lin-dep class-ring ?M* (*set ?S'*))
proof −
obtain *v* **where** *2*:*v* ∈ *carrier-vec nc* **and** *3*:*v* ≠ 0_v *nc* **and** *A* $*_v$ *v* = 0_v *n*
using *vec-space.lin-depE*[*OF assms*(*1*) *assms*(*2*) *distinct-cols-submatrix-UNIV*[*OF assms*(*3*)]] **by** *auto*
have *1*: *submatrix A I UNIV* ∈ *carrier-mat* (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*}) *nc*
apply (*rule carrier-matI*) **unfolding** *dim-submatrix* **using** ‹*A* ∈ *carrier-mat n nc*› **by** *auto*
have *4*:*submatrix A I UNIV* $*_v$ *v* = 0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*})
proof (*rule eq-vecI*)
show *dim-eq*:*dim-vec* (*submatrix A I UNIV* $*_v$ *v*) = *dim-vec* (0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*})) **using** *1* **by** *auto*
fix *i* **assume** *i* < *dim-vec* (0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*}))
then have *i-le*:*i* < *card* {*i*. *i* < *n* ∧ *i* ∈ *I*} **by** *auto*
have (*submatrix A I UNIV* $*_v$ *v*) $ *i* = *row* (*submatrix A I UNIV*) *i* · *v* **using** *dim-eq i-le* **by** *auto*
also have ... = *row A* (*pick I i*) · *v* **using** *row-submatrix-UNIV*
by (*metis* (*no-types*, *lifting*) *dim-eq dim-mult-mat-vec dim-submatrix*(*1*) ‹*i* < *dim-vec* (0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*}))›)
also have ... = *0*
using ‹*A* $*_v$ *v* = 0_v *n*› *i-le*[*THEN pick-le*] **by** (*metis assms*(*1*) *index-mult-mat-vec carrier-matD*(*1*) *index-zero-vec*(*1*))
also have ... = 0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*}) $ *i* **by** (*simp add*: *i-le*)
finally show (*submatrix A I UNIV* $*_v$ *v*) $ *i* = 0_v (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*}) $ *i* **by** *metis*
qed
show *?thesis* **using** *vec-space.lin-depI*[*OF 1 2 3 4*] **using** *assms*(*3*) **by** *auto*
qed

lemma (**in** *vec-space*) *rank-gt-minor*:
assumes $A \in$ *carrier-mat n nc*
assumes *det* (*submatrix A I J*) $\neq$ *0*
shows *card* $\{j.\ j < nc \wedge j \in J\} \leq$ *rank A*
proof −
have *square*:*dim-row* (*submatrix A I J*) = *dim-col* (*submatrix A I J*)
using *det-def* ‹*det* (*submatrix A I J*) $\neq$ *0*› **by** *metis*
then have *full-rank*:*vec-space.rank* (*dim-row* (*submatrix A I J*)) (*submatrix A I J*) = *dim-row* (*submatrix A I J*)
using *vec-space.low-rank-det-zero assms*(*2*) *carrier-matI* **by** *auto*
then have *distinct*:*distinct* (*cols* (*submatrix A I J*)) **using** *vec-space.non-distinct-low-rank*
using *square less-irrefl carrier-matI* **by** *fastforce*
then have *indpt*:*LinearCombinations.module.lin-indpt class-ring* (*module-vec TYPE*('*a*) (*dim-row* (*submatrix A I J*))) (*set* (*cols* (*submatrix A I J*)))
using *vec-space.full-rank-lin-indpt*[*OF - full-rank distinct*] *square* **by** *fastforce*

have *distinct2*: *distinct* (*cols* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)) **using** *submatrix-split distinct* **by** *metis*
have *indpt2*:*LinearCombinations.module.lin-indpt class-ring* (*module-vec TYPE*('*a*) (*card* $\{i.\ i < n \wedge i \in I\}$)) (*set* (*cols* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)))
using *submatrix-split dim-submatrix*(*1*) *indpt* **by** (*metis* (*full-types*) *assms*(*1*) *carrier-matD*(*1*))

have *submatrix A UNIV J* $\in$ *carrier-mat n* (*dim-col* (*submatrix A UNIV J*))
apply (*rule carrier-matI*) **unfolding** *dim-submatrix*(*1*) **using** ‹*A* $\in$ *carrier-mat n nc*› *carrier-matD* **by** *simp-all*
have *lin-indpt* (*set* (*cols* (*submatrix A UNIV J*)))
using *indpt2 vec-space.lin-dep-submatrix-UNIV*[*OF* ‹*submatrix A UNIV J* $\in$ *carrier-mat n* (*dim-col* (*submatrix A UNIV J*))› *- distinct2*] **by** *blast*
have *distinct3*:*distinct* (*cols* (*submatrix A UNIV J*)) **by** (*metis distinct distinct-cols-submatrix-UNIV submatrix-split*)
show *?thesis* **using**
rank-ge-card-indpt[*OF* ‹*A* $\in$ *carrier-mat n nc*› *cols-submatrix-subset* ‹*lin-indpt* (*set* (*cols* (*submatrix A UNIV J*)))›,
unfolded distinct-card[*OF distinct3*, *unfolded cols-length dim-submatrix*], *unfolded carrier-matD*(*2*)[*OF* ‹*A* $\in$ *carrier-mat n nc*›]]
by *blast*
qed

end

37 Shallow Network Model

theory *DL-Shallow-Model*
imports *DL-Network Tensor-Rank*
begin

fun *shallow-model'* **where**

shallow-model′ Z M 0 = Conv (Z,M) (Input M) |
shallow-model′ Z M (Suc N) = Pool (shallow-model′ Z M 0) (shallow-model′ Z M N)

definition *shallow-model* **where**
shallow-model Y Z M N = Conv (Y,Z) (shallow-model′ Z M N)

lemma *valid-shallow-model′*: *valid-net (shallow-model′ Z M N)*
apply *(induction N)* **unfolding** *shallow-model′.simps*
by *(simp add: valid-net.intros, metis shallow-model′.elims shallow-model′.simps(1) valid-net.intros output-size.simps)*

lemma *output-size-shallow-model′*: *output-size (shallow-model′ Z M N) = Z*
apply *(induction N)* **unfolding** *shallow-model′.simps* **using** *output-size.simps*
by *simp-all*

lemma *valid-shallow-model*: *valid-net (shallow-model Y Z M N)*
unfolding *shallow-model-def* **using** *valid-shallow-model′ valid-net.intros output-size.simps output-size-shallow-model′* **by** *metis*

lemma *output-size-shallow-model*: *output-size (shallow-model Y Z M N) = Y*
unfolding *shallow-model-def* **using** *output-size-shallow-model′ output-size.simps*
by *simp*

lemma *input-sizes-shallow-model*: *input-sizes (shallow-model Y Z M N) = replicate (Suc N) M*
apply *(induction N)* **unfolding** *shallow-model-def input-sizes.simps* **by** *simp-all*

lemma *cprank-max1-shallow-model′*:
assumes *y < output-size (shallow-model′ Z M N)*
shows *cprank-max1 (tensors-from-net (insert-weights (shallow-model′ Z M N) w) $ y)*
using *assms* **proof** *(induction N arbitrary:w)*
case *0*
then have *input-sizes (insert-weights (shallow-model′ Z M 0) w) = [M]*
unfolding *shallow-model-def shallow-model′.simps insert-weights.simps input-sizes.simps* **by** *metis*
then have *dims (tensors-from-net (insert-weights (shallow-model′ Z M 0) w) $ y) = [M]*
using *dims-tensors-from-net[OF vec-setI] 0.prems(1) output-size-correct-tensors remove-insert-weights valid-shallow-model′* **by** *metis*
then show *?case*
using *order1* **by** *(metis One-nat-def eq-imp-le length-Cons list.size(3))*
next
case *(Suc N)*
have *y-le-IH:y < dim-vec (tensors-from-net (insert-weights (shallow-model′ Z M N) (λi. w (i + (count-weights (shallow-model′ Z M 0))))))*
using *output-size-correct-tensors[of insert-weights (shallow-model′ Z M N) (λi. w (i + (count-weights (shallow-model′ Z M 0))))),*

unfolded remove-insert-weights, *OF valid-shallow-model'*]
using *Suc.prems*(*1*) *output-size-shallow-model'* **by** *auto*
have *cprank-max1-IH*:*cprank-max1* (*tensors-from-net* (*insert-weights* (*shallow-model'* *Z M N*) (λ*i*. *w* (*i* + (*count-weights* (*shallow-model' Z M 0*))))) $ *y*)
using *Suc.IH Suc.prems*(*1*) *output-size-shallow-model'* **by** *auto*
have *y-le-0*:*y* < *dim-vec* (*tensors-from-net* (*insert-weights* (*shallow-model' Z M 0*) *w*))
by (*metis assms output-size-correct-tensors output-size-shallow-model' remove-insert-weights valid-shallow-model'*)
have *cprank-max1-0*:*cprank-max1* (*tensors-from-net* (*insert-weights* (*shallow-model' Z M 0*) *w*) $ *y*)
proof −
have *input-sizes* (*insert-weights* (*shallow-model' Z M 0*) *w*) = [*M*]
unfolding *shallow-model-def shallow-model'.simps insert-weights.simps input-sizes.simps* **by** *metis*
then show *?thesis* **using** *order1 dims-tensors-from-net*[*OF vec-setI*] *One-nat-def eq-imp-le length-Cons list.size*(*3*) *y-le-0* **by** *metis*
qed
then show *?case* **unfolding** *shallow-model'.simps*(*2*) *insert-weights.simps tensors-from-net.simps*
using *cprank-max1-IH cprank-max1-0 cprank-max1-prod index-component-mult y-le-0 y-le-IH* **by** *fastforce*
qed

lemma *cprank-shallow-model*:
assumes *remove-weights m* = *shallow-model Y Z M N*
assumes *y* < *Y*
shows *cprank* (*tensors-from-net m* $ *y*) ≤ *Z*
proof −
obtain *w* **where** *m* = *insert-weights* (*shallow-model Y Z M N*) *w* **by** (*metis assms*(*1*) *insert-remove-weights*)
have *cprank-max Z* (*tensors-from-net m* $ *y*)
proof −
have *dim-extract*: *dim-row* (*extract-matrix w Y Z*) = *Y*
using *dim-extract-matrix*(*1*) **by** *force*
have *dimc-extract-matrix*: *dim-col* (*extract-matrix w Y Z*) = *Z*
using *dim-extract-matrix*(*2*) **by** *force*
have *input-sizes*: (*input-sizes* (*insert-weights* (*shallow-model' Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*)))) = (*input-sizes* (*shallow-model' Z M N*))
using *input-sizes-remove-weights remove-insert-weights* **by** *auto*
have *0*:*tensors-from-net m* $ *y* = *Tensor-Plus.listsum* (*input-sizes* (*shallow-model'* *Z M N*))
(*map* (λ*j*. (*extract-matrix w Y Z*) $$ (*y*, *j*) · (*tensors-from-net* (*insert-weights* (*shallow-model' Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*)))) $ *j*) [*0*..<*Z*])
unfolding ‹*m* = *insert-weights* (*shallow-model Y Z M N*) *w*› *shallow-model-def insert-weights.simps tensors-from-net.simps*
using *nth-mat-tensorlist-mult dims-tensors-from-net assms*(*2*) *dim-extract output-size-correct-tensors*[*of insert-weights* (*shallow-model' Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*)), *unfolded remove-insert-weights*, *OF valid-shallow-model'*]

dimc-extract-matrix output-size-shallow-model′ input-sizes **by** *auto*

def *Bs* == *map* (λ*j*. *extract-matrix w Y Z* $$ (*y*, *j*) · *tensors-from-net* (*insert-weights* (*shallow-model′ Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*))) $ *j*) [*0*..<*Z*]

have ⋀*B*. *B* ∈ *set Bs* ⟹ *cprank-max1 B* ⋀*B*. *B* ∈ *set Bs* ⟹ *dims B* = *input-sizes* (*shallow-model′ Z M N*)
proof −
fix *B* **assume** *B* ∈ *set Bs*
then obtain *j* **where** *B* = *Bs* ! *j j* < *length Bs* **by** (*metis in-set-conv-nth*)
then have *j* < *Z* **using** *length-map Bs-def* **by** *simp*
have *1*:*cprank-max1* (*tensors-from-net* (*insert-weights* (*shallow-model′ Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*))) $ *j*)
using ‹*j* < *Z*› *output-size-shallow-model′ cprank-max1-shallow-model′* **by** *auto*
then have *cprank-max1* (*extract-matrix w Y Z* $$ (*y*, *j*) · *tensors-from-net* (*insert-weights* (*shallow-model′ Z M N*) (λ*i*. *w* (*i* + *Y* ∗ *Z*))) $ *j*)
using *smult-prod-extract1 cprank-max1-order0*[*OF 1*, *of extract-matrix w Y Z* $$ (*y*, *j*) · *1*]
by (*metis dims-smult mult.left-neutral order-tensor-one*)
then show *cprank-max1 B* **by** (*simp add*: *Bs-def* ‹*B* = *Bs* ! *j*› ‹*j* < *Z*›)
show *dims B* = *input-sizes* (*shallow-model′ Z M N*) **unfolding** ‹*B* = *Bs* ! *j*› *Bs-def*
nth-map[*of j* [*0*..<*Z*], *unfolded length-upt Nat.diff-0*, *OF* ‹*j* < *Z*›] *dims-smult input-sizes*[*symmetric*]
by (*rule dims-tensors-from-net*; *rule vec-setI*[**where** *i*=*j*], *simp add*:‹*j* < *Z*›, *metis* (*no-types*) ‹*j* < *Z*› *output-size-correct-tensors output-size-shallow-model′ remove-insert-weights valid-shallow-model′*)
qed
then show *?thesis* **unfolding** *0* **using** *cprank-maxI length-map Bs-def* **by** (*metis* (*no-types*, *lifting*) *diff-zero length-upt*)
qed
then show *?thesis* **unfolding** *cprank-def* **by** (*simp add*: *Least-le*)
qed

end

38 Fundamental Theorem of Network Capacity

theory *DL-Fundamental-Theorem-Network-Capacity*
imports *DL-Rank-CP-Rank DL-Deep-Model-Poly Lebesgue-Zero-Set DL-Rank-Submatrix HOL−Analysis.Complete-Measure DL-Shallow-Model*
begin

context *deep-model-correct-params-y*
begin

definition *polynomial-f w* = *det* (*submatrix* (*matricize* {*n*. *even n*} (*A w*)) *rows-with-1*

rows-with-1)

lemma *polyfun-polynomial*:
shows *polyfun* {..<*weight-space-dim*} *polynomial-f*
unfolding *polynomial-f-def* **using** *polyfun-det-deep-model* **unfolding** *witness-submatrix-def A'-def* **.**

definition *polynomial-p* = (*SOME p*. *vars p* ⊆ {..<*weight-space-dim*} ∧ (∀ *x*. *insertion x p* = *polynomial-f x*))

lemma
polynomial-p-not-0: *polynomial-p*≠*0* **and**
vars-polynomial-p: *vars polynomial-p* ⊆ {..<*weight-space-dim*} **and**
polynomial-pf: ⋀*w*. *insertion w polynomial-p* = *polynomial-f w*
proof −
have *vars polynomial-p* ⊆ {..<*weight-space-dim*} ∧ (∀ *x*. *insertion x polynomial-p* = *polynomial-f x*) **unfolding** *polynomial-p-def*
using *someI-ex*[*OF polyfun-polynomial*[*unfolded polyfun-def*]] **.**
then show *vars polynomial-p* ⊆ {..<*weight-space-dim*} ⋀*w*. *insertion w polynomial-p* = *polynomial-f w* **by** *auto*
show *polynomial-p*≠*0* **using** *A'-def Aw'-def'* ‹⋀*w*. *insertion w polynomial-p* = *polynomial-f w*› *polynomial-f-def witness-det* **by** *auto*
qed

lemma *if-polynomial-0-rank*:
assumes *polynomial-f w* ≠ *0*
shows *r* ^ *N-half* ≤ *cprank* (*A w*)
proof −
have *r* ^ *N-half* = *dim-row* (*submatrix* (*matricize* {*n*. *even n*} (*A w*)) *rows-with-1 rows-with-1*)
by (*metis* (*full-types*) *Aw'-def card-rows-with-1 dim-submatrix*(*1*) *dims-A dims-Aw dims-matricize*(*1*) *set-le-in*)
also have ... ≤ *mrank* (*matricize* {*n*. *even n*} (*A w*))
using *assms vec-space.rank-gt-minor*[*OF carrier-matI*[*OF dims-A'-pow*, *unfolded weight-space-dim-def*]]
by (*metis* (*full-types*) *A'-def dim-submatrix*(*1*) *dims-A'-pow*(*1*) *polynomial-f-def*)
also have ... ≤ *cprank* (*A w*) **using** *matrix-rank-le-cp-rank* **by** *blast*
finally show *?thesis* **.**
qed

lemma *if-polynomial-0-evaluate*:
assumes *polynomial-f wd* ≠ *0*
assumes ∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶ *evaluate-net* (*insert-weights* (*deep-model-l rs*) *wd*) *inputs*
= *evaluate-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-half*−*1*)) *ws*) *inputs*
shows *Z* ≥ *r* ^ *N-half*
proof −

have *valid1*:*valid-net'* (*insert-weights* (*deep-model-l rs*) *wd*)
using *remove-insert-weights valid-deep-model* **by** *presburger*
have *valid2*:*valid-net'* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-half*−*1*)) *ws*)
by (*simp add*: *remove-insert-weights valid-shallow-model*)
have *input-sizes*: *input-sizes* (*insert-weights* (*deep-model-l rs*) *wd*)
= *input-sizes* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2* ∗ *N-half* − *1*)) *ws*)
by (*metis N-half-def Suc-mult-two-diff-one input-sizes-remove-weights input-sizes-shallow-model local.input-sizes-deep-model power-eq-0-iff remove-insert-weights zero-neq-numeral*)
have *tensors-from-net* (*insert-weights* (*deep-model-l rs*) *wd*)
= *tensors-from-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-half* −*1*)) *ws*)
using *tensors-from-net-eqI*[*OF valid1 valid2 input-sizes*, *unfolded input-sizes-remove-weights remove-insert-weights*]
using *assms* **by** *blast*
then show *?thesis*
using *if-polynomial-0-rank assms*
by (*metis A-def assms*(*1*) *cprank-shallow-model less-le-trans not-le remove-insert-weights y-valid*)
qed

lemma *if-polynomial-0-evaluate-notex*:
assumes *polynomial-f wd* ≠ *0*
shows ¬(∃ *weights-shallow Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
evaluate-net (*insert-weights* (*deep-model-l rs*) *wd*) *inputs*
= *evaluate-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-half*−*1*)) *ws*) *inputs*))
using *assms if-polynomial-0-evaluate not-le* **by** *blast*

theorem *fundamental-theorem-network-capacity*:
AE x in lborel-f weight-space-dim. *r* ^ *N-half* ≤ *cprank* (*A x*)
using *AE-I'*[*OF lebesgue-mpoly-zero-set*[*OF polynomial-p-not-0 vars-polynomial-p*]]
by (*metis* (*mono-tags*, *lifting*) *Collect-mono if-polynomial-0-rank polynomial-pf*)

theorem *fundamental-theorem-network-capacity-v2*:
shows *AE wd in lborel-f weight-space-dim*.
¬(∃ *ws Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
evaluate-net (*insert-weights* (*deep-model-l rs*) *wd*) *inputs*
= *evaluate-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*∗*N-half*−*1*)) *ws*) *inputs*))
apply (*rule AE-I'*[*OF lebesgue-mpoly-zero-set*[*OF polynomial-p-not-0 vars-polynomial-p*], *unfolded polynomial-pf*])
apply (*rule subsetI*) **unfolding** *mem-Collect-eq*
using *if-polynomial-0-evaluate-notex* **by** *metis*

end
end

References

[1] A. Bentkamp. An Isabelle Formalization of the Expressiveness of Deep Learning. Master's thesis, Universität des Saarlandes, 2016. http://matryoshka.gforge.inria.fr/bentkamp_msc_thesis.pdf.

[2] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis. In V. Feldman, A. Rakhlin, and O. Shamir, editors, *Conference on Learning Theory (COLT 2016)*, volume 49 of *JMLR Workshop and Conference Proceedings*, pages 698–728. JMLR.org, 2016.