

# Expressiveness of Deep Learning

Alexander Bentkamp

December 17, 2016

## 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

<b>1</b>	<b>Tensor</b>	<b>3</b>
<b>2</b>	<b>Subtensors</b>	<b>9</b>
<b>3</b>	<b>Tensor Addition</b>	<b>13</b>
<b>4</b>	<b>Tensor Scalar Multiplication</b>	<b>18</b>
<b>5</b>	<b>Tensor Product</b>	<b>21</b>
<b>6</b>	<b>Unit Vectors as Tensors</b>	<b>29</b>
<b>7</b>	<b>Tensor CP-Rank</b>	<b>30</b>
<b>8</b>	<b>Missing Lemmas of Vector_Space</b>	<b>35</b>
<b>9</b>	<b>Missing Lemmas of VS_Connect</b>	<b>36</b>
<b>10</b>	<b>Missing Lemmas of List</b>	<b>37</b>
<b>11</b>	<b>Matrix Rank</b>	<b>38</b>

<b>12 Subadditivity of rank</b>	<b>45</b>
<b>13 Missing Lemmas of Sublist</b>	<b>53</b>
<b>14 Pick</b>	<b>55</b>
<b>15 Sublist</b>	<b>60</b>
<b>16 Tensor Matricization</b>	<b>63</b>
<b>17 Submatrices</b>	<b>78</b>
<b>18 Submatrix</b>	<b>78</b>
<b>19 CP-Rank and Matrix Rank</b>	<b>79</b>
<b>20 Missing Lemmas of Matrix</b>	<b>82</b>
<b>21 Matrix to Vector Conversion</b>	<b>83</b>
<b>22 Deep Learning Networks</b>	<b>84</b>
<b>23 Concrete Matrices</b>	<b>99</b>
<b>24 Missing Lemmas of Finite_Set</b>	<b>102</b>
<b>25 Deep Network Model</b>	<b>102</b>
<b>26 Material to be moved to HOL finally</b>	<b>126</b>
26.1 Already taken over into Isabelle repository . . . . .	126
26.2 Not yet taken over into Isabelle repository . . . . .	126
<b>27 Less common functions on lists</b>	<b>127</b>
<b>28 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view</b>	<b>134</b>
28.1 Preliminary: auxiliary operations for ‘almost everywhere zero’	134
28.2 Type definition . . . . .	138
28.3 Additive structure . . . . .	139
28.4 Multiplicative structure . . . . .	141
28.5 Single-point mappings . . . . .	148
28.6 Integral domains . . . . .	150
28.7 Mapping order . . . . .	152
28.8 Fundamental mapping notions . . . . .	153
28.9 Degree . . . . .	155
28.10 Inductive structure . . . . .	157
28.11 Quasi-functorial structure . . . . .	158

28.12	Canonical dense representation of $nat \Rightarrow_0 'a$	160
28.13	Canonical sparse representation of $'a \Rightarrow_0 'b$	163
28.14	Size estimation	164
28.15	Further mapping operations and properties	166
<b>29</b>	<b>An abstract type for multivariate polynomials</b>	<b>167</b>
29.1	Abstract type definition	167
29.2	Additive structure	167
29.3	Multiplication by a coefficient	168
29.4	Multiplicative structure	168
29.5	Monomials	169
29.6	Integral domains	171
29.7	Monom coefficient lookup	171
29.8	Insertion morphism	171
29.9	Degree	174
29.10	Monomials	174
29.11	Pseudo-division of polynomials	175
29.12	Primitive poly, etc	176
<b>30</b>	<b>MPpoly Mapping extenion</b>	<b>177</b>
<b>31</b>	<b>MPoly extension</b>	<b>180</b>
<b>32</b>	<b>Nested MPoly</b>	<b>186</b>
<b>33</b>	<b>Polynomials representing the Deep Network Model</b>	<b>192</b>
<b>34</b>	<b>Alternative Lebesgue Measure Definition</b>	<b>201</b>
<b>35</b>	<b>Lebesgue Measure of Polynomial Zero Sets</b>	<b>206</b>
<b>36</b>	<b>Ranks of Submatrices</b>	<b>211</b>
<b>37</b>	<b>Shallow Network Model</b>	<b>217</b>
<b>38</b>	<b>Missing Lemmas of Complete_Measure</b>	<b>220</b>
<b>39</b>	<b>Fundamental Theorem of Network Capacity</b>	<b>220</b>

## 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**  $\bigwedge m. m < \text{length } ds \implies is!m < ds!m$

**shows** is  $\triangleleft$  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  $\triangleleft$  ds' **using** Cons **by** (metis length-Cons less-irrefl linorder-neqE-nat not-less-eq nth-Cons-Suc)

**then show** ?case **using** Cons.prem2 (ds = d # ds') valid-index.Cons **by** fastforce

**qed**

**lemma** valid-index-append:

**assumes** is1-valid:is1  $\triangleleft$  ds1 **and** is2-valid:is2  $\triangleleft$  ds2

**shows** is1 @ is2  $\triangleleft$  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))

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

**fun** lookup-base::nat list  $\Rightarrow$  'a list  $\Rightarrow$  nat list  $\Rightarrow$  '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  $\Rightarrow$  nat list  $\Rightarrow$  'a **where**

lookup A = lookup-base (dims A) (vec A)

**fun** tensor-vec-from-lookup::nat list  $\Rightarrow$  (nat list  $\Rightarrow$  'a)  $\Rightarrow$  '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  
( $\lambda i. \text{tensor-vec-from-lookup } ds (\lambda is. e (i \# is))$ ) [0.. $d$ ])

**definition** tensor-from-lookup::nat list  $\Rightarrow$  (nat list  $\Rightarrow$  'a)  $\Rightarrow$  'a tensor **where**

tensor-from-lookup ds e = tensor-from-vec ds (tensor-vec-from-lookup ds e)

**lemma** concat-parts-leg:

```

assumes  $a * d \leq \text{length } v$ 
shows  $\text{concat } (\text{map } (\text{fixed-length-sublist } v \ d) \ [0..<a]) = \text{take } (a*d) \ v$ 
using assms proof (induction a)
  case 0
  then show ?case by simp
next
  case (Suc a)
  then have  $\text{concat } (\text{map } (\text{fixed-length-sublist } v \ d) \ [0..<a]) = \text{take } (a * d) \ v$  by
auto
  then have  $\text{concat } (\text{map } (\text{fixed-length-sublist } v \ d) \ [0..<\text{Suc } a]) =$ 
 $\text{take } (a * d) \ v \ @ \ \text{fixed-length-sublist } v \ d \ a$  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  $a * d = \text{length } v$ 
shows  $\text{concat } (\text{map } (\text{fixed-length-sublist } v \ d) \ [0..<a]) = v$ 
by (simp add: concat-parts-leq assms)

```

```

lemma tensor-lookup-base:
assumes  $\text{length } v = \text{prod-list } ds$ 
and  $\bigwedge is. is \triangleleft ds \implies \text{lookup-base } ds \ v \ is = e \ is$ 
shows  $\text{tensor-vec-from-lookup } ds \ e = v$ 
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  $a * \text{prod-list } ds = \text{length } v$  by auto
  {
    fix i assume  $i < a$ 
    then have  $\text{prod-list } ds * (i+1) \leq \text{length } v$  using  $\langle a * \text{prod-list } ds = \text{length } v \rangle$ 
using discrete mult.commute mult-le-mono1 by metis
    have  $\bigwedge is'. is' \triangleleft ds \implies e \ (i \# is') = \text{lookup-base } ds \ (\text{fixed-length-sublist } v$ 
 $(\text{prod-list } ds) \ i) \ is'$ 
    using  $\langle i < a \rangle$  by (metis Cons.prem(2) Tensor.lookup-base-Cons valid-index.simps)
    then have  $\text{tensor-vec-from-lookup } ds \ (\lambda is'. e \ (i \# is')) = \text{fixed-length-sublist } v$ 
 $(\text{prod-list } ds) \ i$ 
    using Cons using  $\langle \text{prod-list } ds * (i + 1) \leq \text{length } v \rangle$  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 \langle a * prod-list ds = length v \rangle
atLeastLessThan-iff map-eq-conv set-upt Cons by (metis (no-types, lifting))
qed

```

**lemma** *tensor-lookup*:  
**assumes**  $\bigwedge is. is \triangleleft \text{dims } A \implies \text{lookup } A \text{ } is = e \text{ } is$   
**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**  $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = l$   
**shows**  $\text{length } (\text{concat } xss) = \text{length } xss * l$   
**using** *assms* **by** (*induction xss; auto*)

**lemma** *concat-equal-length-map*:  
**assumes**  $\bigwedge i. i < a \implies \text{length } (f \ i) = d$   
**shows**  $\text{length } (\text{concat } (\text{map } (\lambda i. f \ i) [0..<a])) = a * d$   
**using** *assms* **by** (*induction a; auto*)

**lemma** *concat-parts*:  
**assumes**  $\bigwedge xs. xs \in \text{set } xss \implies \text{length } xs = d \text{ and } i < \text{length } xss$   
**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**  $\text{length } (\text{concat } xss) = \text{length } xs * d$  **by** (*simp add: Cons.prem1*)  
*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.prem1*)  
  **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.prem1  
**by** *auto*  
  **qed**  
**qed**

**lemma** *concat-parts'*:  
**assumes**  $\bigwedge i. i < a \implies \text{length } (f \ i) = d$   
**and**  $i < a$   
**shows** *fixed-length-sublist* (concat (map (lambda 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 ( $\bigwedge i. i < a \implies \text{length } (f i) = d$ ) by auto
then have  $\text{length } (\text{concat } (\text{map } f [0..<a])) = a * d$  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.prem1) <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:  
 $\text{length } (\text{tensor-vec-from-lookup } ds e) = \text{prod-list } ds$   
**by** (induction ds arbitrary:e; auto simp add: concat-equal-length-map)

**lemma** lookup-tensor-vec:  
**assumes**  $is \triangleleft ds$   
**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  $\lambda i. \text{tensor-vec-from-lookup}$   
 ds  $\lambda is. e (i \# is)$ ] prod-list ds i] <i < d>)  
**qed**

**lemma** lookup-tensor-from-lookup:  
**assumes**  $is \triangleleft ds$   
**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*:  $\text{dims } (\text{tensor-from-lookup } ds \ e) = ds$   
**unfolding** *tensor-from-lookup-def*  
**by** (*simp add: length-tensor-vec-from-lookup*)

**lemma** *tensor-lookup-cong*:  
**assumes** *tensor-from-lookup ds e<sub>1</sub> = tensor-from-lookup ds e<sub>2</sub>*  
**and** *is < ds*  
**shows** *e<sub>1</sub> is = e<sub>2</sub> is* **using** *assms lookup-tensor-from-lookup* **by** *metis*

**lemma** *tensor-from-lookup-eqI*:  
**assumes**  $\bigwedge is. is < ds \implies e_1 \ is = e_2 \ is$   
**shows** *tensor-from-lookup ds e<sub>1</sub> = tensor-from-lookup ds e<sub>2</sub>*  
**by** (*metis assms lookup-tensor-vec length-tensor-vec-from-lookup tensor-lookup-base tensor-from-lookup-def*)

**lemma** *tensor-lookup-eqI*:  
**assumes**  $\text{dims } A = \text{dims } B$  **and**  $\bigwedge is. is < (\text{dims } A) \implies \text{lookup } A \ is = \text{lookup } B \ is$   
**shows**  $A = B$  **by** (*metis assms(1) assms(2) tensor-lookup*)

**end**

## 2 Subtensors

**theory** *Tensor-Subtensor*  
**imports** *Tensor*  
**begin**

**definition** *subtensor*:: $'a \ \text{tensor} \Rightarrow \text{nat} \Rightarrow 'a \ \text{tensor}$  **where**  
*subtensor A i = tensor-from-vec (tl (dims A)) (fixed-length-sublist (vec A) (prod-list (tl (dims A)))) i*

**definition** *subtensor-combine*:: $\text{nat list} \Rightarrow 'a \ \text{tensor list} \Rightarrow 'a \ \text{tensor}$  **where**  
*subtensor-combine ds As = tensor-from-vec (length As # ds) (concat (map vec As))*

**lemma** *length-fixed-length-sublist[simp]*:  
**assumes**  $(\text{Suc } i) * l \leq \text{length } xs$   
**shows**  $\text{length } (\text{fixed-length-sublist } xs \ l \ i) = l$   
**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**  $\text{dims } A \neq []$  **and**  $i < \text{hd } (\text{dims } A)$   
**shows**  $\text{vec } (\text{subtensor } A \ i) = \text{fixed-length-sublist } (\text{vec } A) (\text{prod-list } (\text{tl } (\text{dims } A))) \ i$   
**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**  $\text{dims } A \neq []$  **and**  $i < \text{hd } (\text{dims } A)$   
**shows**  $\text{dims } (\text{subtensor } A \ i) = \text{tl } (\text{dims } A)$   
**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**  $\text{dims } A \neq []$   
**shows**  $\text{subtensor-combine } (\text{tl } (\text{dims } A)) (\text{map } (\text{subtensor } A) [0..<\text{hd } (\text{dims } A)]) = A$   
**proof** –  
**have** *length-vec-A*:  $\text{hd } (\text{dims } A) * \text{prod-list } (\text{tl } (\text{dims } A)) = \text{length } (\text{Tensor.vec } A)$   
**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**  $i < \text{hd } (\text{dims } A)$   
**then have**  $(\text{Suc } i) * (\text{prod-list } (\text{tl } (\text{dims } A))) \leq \text{length } (\text{vec } A)$   
**by** (*metis Suc-leI length-vec-A mult-le-mono1*)  
**then have**  $(\text{vec } \circ (\lambda i. \text{tensor-from-vec } (\text{tl } (\text{dims } A)) (\text{?subtensor-vec } i))) \ i =$   
*?subtensor-vec i*  
**by** *simp*  
}  
**then have**  $1 : \text{map } (\text{Tensor.vec } \circ (\lambda i. \text{tensor-from-vec } (\text{tl } (\text{dims } A)) (\text{?subtensor-vec } i))) [0..<\text{hd } (\text{dims } A)]$   
=  $\text{map } \text{?subtensor-vec } [0..<\text{hd } (\text{dims } A)]$  **by** *auto*  
**then have**  $\text{subtensor-combine } (\text{tl } (\text{dims } A)) (\text{map } (\lambda i. \text{subtensor } A \ i) [0..<\text{hd } (\text{dims } A)]) = A$   
**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**  $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$   
**shows** *subtensor-combine-dims[simp]*:  $\text{dims } (\text{subtensor-combine } ds \ As) = \text{length } As \ \# \ ds$  (**is** *?D*)  
**and** *subtensor-combine-vec[simp]*:  $\text{vec } (\text{subtensor-combine } ds \ As) = \text{concat } (\text{map } \text{vec } As)$  (**is** *?V*)  
**proof** –  
**have**  $\bigwedge v. v \in \text{set } (\text{map } \text{Tensor.vec } As) \implies \text{length } v = \text{prod-list } ds$  **using** *assms length-vec* **by** *fastforce*  
**then have**  $\text{length } As * \text{prod-list } ds = \text{length } (\text{concat } (\text{map } \text{Tensor.vec } As))$  **using** *concat-equal-length*  
**by** (*metis length-map*)  
**then show** *?D ?V* **unfolding** *subtensor-combine-def* **by** *simp+*  
**qed**

**lemma** *subtensor-subtensor-combine*:  
**assumes**  $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$  **and**  $i < \text{length } As$   
**shows**  $\text{subtensor } (\text{subtensor-combine } ds \ As) \ i = As \ ! \ i$   
**proof** –  
  **have** *fixed-length-sublist*  $(\text{concat } (\text{map } \text{vec } As)) \ (\text{prod-list } ds) \ i = \text{vec } (As \ ! \ i)$   
  **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**  $(\text{metis } \text{assms } \text{list.sel}(3) \ \text{nth-mem } \text{tensor-from-vec-simp})$   
**qed**

**lemma** *subtensor-induct*[*case-names order-0 order-step*]:  
**assumes** *order-0*:  $\bigwedge A. \text{dims } A = [] \implies P \ A$   
**and** *order-step*:  $\bigwedge A. \text{dims } A \neq [] \implies (\bigwedge i. i < \text{hd } (\text{dims } A) \implies P \ (\text{subtensor } A \ i)) \implies P \ A$   
**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**  $(\text{metis } \text{dims-subtensor } \text{list.sel}(3))$   
**qed**

**lemma** *subtensor-combine-induct*[*case-names order-0 order-step*]:  
**assumes** *order-0*:  $\bigwedge A. \text{dims } A = [] \implies P \ A$   
**and** *order-step*:  $\bigwedge As \ ds. (\bigwedge A. A \in \text{set } As \implies P \ A) \implies (\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds) \implies P \ (\text{subtensor-combine } ds \ As)$   
**shows**  $P \ A$   
**proof** (*induction A rule:subtensor-induct*)  
  **case** (*order-0 A*)  
  **then show** *?case* **by**  $(\text{simp add: } \text{assms}(1))$   
**next**  
  **case** (*order-step A*)  
  **have**  $P \ (\text{subtensor-combine } (\text{tl } (\text{dims } A)) \ (\text{map } (\text{subtensor } A) \ [0..<\text{hd } (\text{dims } A)]))$   
  **apply**  $(\text{rule } \text{assms}(2))$   
  **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**  $i \# is \triangleleft \text{dims } A$   
**shows**  $\text{lookup } (\text{subtensor } A \ i) \ is = \text{lookup } A \ (i \# is)$

```

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:subtens (subtens-combine ds As) i = As ! i
  by (metis list.discI list.sel(1) order-step.hyps order-step.prems subtens-combine-dims
subtens-subtens-combine valid-index-dimsE)
  have 1:dims (subtens-combine ds As) = length As # ds
  using order-step subtens-combine-def subtens-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-subtens lookup-def list.discI list.sel(1)
list.sel(3) valid-index-dimsE vec-subtens by (metis 0 1)

```

**qed**

**lemma** *lookup-subtens*:

**assumes** *is*  $\triangleleft$  *dims A*

**shows** *lookup A is = hd (vec (fold ( $\lambda i A.$  *subtens 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-subtens lookup-subtens1 fold-simps(2) list.discI list.sel(1) list.sel(3)*

*valid-indexE* **by** (*metis (no-types, lifting)*)

**qed**

**lemma** *subtens-eqI*:

**assumes** *dims A*  $\neq$  []

**and** *dims-eq:dims A = dims B*

**and**  $\bigwedge i. i < \text{hd}(\text{dims } A) \implies \text{subtens } A \ i = \text{subtens } B \ i$

**shows** *A=B*

**proof** –

{

**fix** *is* **assume** *is*  $\triangleleft$  *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-subtens1 assms* **by** (*metis ( $\triangleleft$  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 Option Tensor-Subtensor
begin

```

```

definition vec-plus a b = map ( $\lambda(x,y). \text{plus } x \ y$ ) (zip a b)

```

```

definition plus-base::'a::semigroup-add tensor  $\Rightarrow$  'a tensor  $\Rightarrow$  '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  $\Longrightarrow$  dims (A + B) = dims A unfolding
plus-def plus-base-def

```

```

  using dims-tensor length-vec length-map map-fst-vec vec-plus-def by (metis (full-types))

```

```

lemma plus-dim2[simp]: dims A = dims B  $\Longrightarrow$  dims (A + B) = dims B using
plus-dim1 by metis

```

```

lemma plus-base: dims A = dims B  $\Longrightarrow$  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-vec take-map
take-vec 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-vec
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  $\neq$  []

```

```

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.prem1 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.prem1] unfolding vec-plus-def using
order-0.prem1 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.prem1)
subtensor-plus valid-index.cases)
    using 1 ⟨is = i # is'⟩ order-step.prem1 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-semigroup-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-from-vec d (vec0 (prod-list d))*

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

**lemma** *lookup-is-in-vec*: *is  $\triangleleft$  (dims A)  $\implies$  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.prem by auto*  
    **have** *2:is'  $\triangleleft$  dims (subtensor A i) using  $\langle is = i \# is' \rangle$  dims-subtensor order-step.prem*  
**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.prem*  
**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**  $is \triangleleft ds$   
**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**  $listsum::nat\ list \Rightarrow 'a::monoid-add\ tensor\ list \Rightarrow 'a\ tensor$  **where**  
 $listsum\ ds\ As = foldr\ (op\ +)\ As\ (tensor0\ ds)$

**definition**  $listsum'::'a::monoid-add\ tensor\ list \Rightarrow 'a\ tensor$  **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**  $\bigwedge A. A \in set\ As \Longrightarrow dims\ A = ds$   
**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  $is < dims\ (subtensor\ (tensor0\ ds)\ i)$ 
  then have  $i \# is < dims\ (tensor0\ ds)$  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 < dims (subtensor (tensor0 ds) i) dims-tensor0
lookup-tensor0
    by metis
qed

```

```

lemma subtensor-listsum:
assumes  $\bigwedge A. A \in set\ As \implies dims\ A = ds$ 
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  $\bigwedge A. A \in set\ As \implies A = tensor0\ ds$ 
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  $\bigwedge i. i \neq j \implies i < \text{length } As \implies As!i = \text{tensor0 } ds$ 
and  $\text{dims } (As!j) = ds$ 
and  $j < \text{length } As$ 
shows  $\text{listsum } ds \ As = As!j$ 
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  $\bigwedge i. i < \text{length } As \implies As ! i = \text{tensor0 } ds$  using Cons using
Suc-less-eq length-Cons list.sel(3) nat.simps(3) nth-tl by fastforce
          then have  $\text{listsum } ds \ As = \text{tensor0 } ds$  using listsum0 by (metis in-set-conv-nth)
          then show ?thesis by (metis 0 Cons.prem(2) listsum-Cons nth-Cons-0 tensor-add-0-right)
        next
          case (Suc j')
            then have  $\text{listsum } ds \ As = As!j'$  by (metis (no-types, lifting) Cons.IH
Cons.prem(1) Cons.prem(2) Cons.prem(3) Suc-less-eq length-Cons less-Suc-eq
list.sel(3) not-less-eq nth-tl)
            then show ?thesis by (metis Cons.prem(1) Cons.prem(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
      qed

```

```

lemma lookup-listsum:
assumes  $is \triangleleft ds$ 
and  $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$ 
shows  $\text{lookup } (\text{listsum } ds \ As) \ is = (\sum A \leftarrow As. \text{lookup } A \ is)$ 
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**  $vec-smult::'a::ring \Rightarrow 'a\ list \Rightarrow 'a\ list$  **where**  
 $vec-smult\ \alpha\ \beta = map\ (op\ *\ \alpha)\ \beta$

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

**lemma**  $vec-smult-Cons$ :  
**shows**  $vec-smult\ \alpha\ (a\ \# as) = (\alpha * a)\ \# vec-smult\ \alpha\ as$  **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**  $vec-smult\ \alpha\ (vec-plus\ as\ bs) = vec-plus\ (vec-smult\ \alpha\ as)\ (vec-smult\ \alpha\ bs)$   
**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**  $0:vec-smult\ \alpha\ (vec-plus\ (a\ \# as')\ bs) = (\alpha*(a+b))\ \# vec-smult\ \alpha\ (vec-plus\ as'\ bs')$   
**unfolding**  $vec-smult-def\ vec-plus-def$  **using**  $Cons.IH[of\ bs']$  **by**  $simp$   
**have**  $length\ bs' = length\ as'$  **using**  $Cons.prem\ (bs = b\ \# bs')$  **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**  $length-vec-smult$ :  $length\ (vec-smult\ \alpha\ v) = length\ v$  **unfolding**  $vec-smult-def$   
**by**  $simp$

**definition**  $smult::'a::ring \Rightarrow 'a\ tensor \Rightarrow 'a\ tensor$  **(infixl  $\cdot$  70)** **where**  
 $smult\ \alpha\ A = (tensor-from-vec\ (dims\ A)\ (vec-smult\ \alpha\ (vec\ A)))$

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

**lemma**  $dims-smult[simp]$ :  $dims\ (\alpha \cdot A) = dims\ A$

**and** *vec-smult*[simp]:  $\text{vec } (\alpha \cdot A) = \text{map } (\text{op} * \alpha) (\text{vec } A)$   
**unfolding** *smult-def vec-smult-def* **by** (*simp add: length-vec*)**+**

**lemma** *tensor-smult-distr-right*:  $(\alpha + \beta) \cdot A = \alpha \cdot A + \beta \cdot A$   
**unfolding** *plus-def plus-base-def*  
**by** (*auto; metis smult-def vec-smult-def vec-smult-distr-right*)

**lemma** *tensor-smult-distr-left*:  $\text{dims } A = \text{dims } B \implies \alpha \cdot (A + B) = \alpha \cdot A + \alpha \cdot B$

**proof** –

**assume** *a1*:  $\text{dims } A = \text{dims } B$   
**then have** *f2*:  $\text{length } (\text{vec-plus } (\text{vec } A) (\text{vec } B)) = \text{length } (\text{vec } A)$   
**by** (*simp add: length-vec vec-plus-def*)  
**have** *f3*:  $\text{dims } (\text{tensor-from-vec } (\text{dims } B) (\text{vec-smult } \alpha (\text{vec } A))) = \text{dims } B$   
**using** *a1* **by** (*simp add: length-vec vec-smult-def*)  
**have** *f4*:  $\text{vec } (\alpha \cdot A) = \text{vec-smult } \alpha (\text{vec } A)$   
**by** (*simp add: vec-smult-def*)  
**have**  $\text{length } (\text{vec-smult } \alpha (\text{vec } B)) = \text{length } (\text{vec } B)$   
**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**  $\text{length } xs = l * c \ i < c$

**shows**  $\text{fixed-length-sublist } (\text{vec-smult } \alpha \ xs) \ l \ i = \text{vec-smult } \alpha \ (\text{fixed-length-sublist } xs \ l \ i)$

**unfolding** *fixed-length-sublist-def vec-smult-def* **by** (*simp add: drop-map take-map*)

**lemma** *smult-subtensor*:

**assumes**  $\text{dims } A \neq [] \ i < \text{hd } (\text{dims } A)$

**shows**  $\alpha \cdot \text{subtensor } A \ i = \text{subtensor } (\alpha \cdot A) \ i$

**proof** (*rule tensor-eqI*)

**show**  $\text{dims } (\alpha \cdot \text{subtensor } A \ i) = \text{dims } (\text{subtensor } (\alpha \cdot A) \ i)$

**using** *dims-smult dims-subtensor assms(1) assms(2)* **by** *simp*

**show**  $\text{vec } (\alpha \cdot \text{subtensor } A \ i) = \text{vec } (\text{subtensor } (\alpha \cdot A) \ i)$

**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**  $is \triangleleft \text{dims } A$

**shows**  $\text{lookup } (\alpha \cdot A) \ is = \alpha * \text{lookup } A \ is$

**using** *assms* **proof** (*induction A arbitrary:is rule:subtensor-induct*)

**case** (*order-0 A is*)

**then have**  $\text{length } (\text{vec } A) = 1$  **by** (*simp add: length-vec*)

```

then have  $hd (vec-smult \alpha (vec A)) = \alpha * hd (vec A)$  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  $lookup (\alpha \cdot subtensor A i) is' = \alpha * lookup (subtensor A i) is'$ 
    by (metis (no-types, lifting) dims-subtensor list.sel(1) list.sel(3) order-step.IH order-step.hyps order-step.premis valid-index-dimsE)
  then show ?case using smult-subtensor ⟨is = i # is'⟩ dims-smult lookup-subtensor1

   $list.sel(1) order-step.hyps order-step.premis valid-index-dimsE$ 
  by metis
qed

```

```

lemma tensor-smult-assoc:
fixes  $A::'a::ring\ tensor$ 
shows  $\alpha \cdot (\beta \cdot A) = (\alpha * \beta) \cdot A$ 
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:  $A * B = tensor-from-vec (dims A @ dims B) (concat$ 
    ( $map (\lambda a. vec-smult a (vec B)) (vec A)$ ))

```

```

  abbreviation tensor-prod-otimes ::  $'a\ tensor \Rightarrow 'a\ tensor \Rightarrow 'a\ tensor$  (infixl
     $\otimes$  70)
    where  $A \otimes B \equiv A * B$ 

```

```

  lemma vec-tensor-prod[simp]:  $vec (A \otimes B) = concat (map (\lambda a. vec-smult a (vec$ 
     $B)) (vec A))$  (is  $?V$ )

```

```

  and dims-tensor-prod[simp]:  $dims (A \otimes B) = dims A @ dims B$  (is  $?D$ )

```

```

  proof -

```

```

    have  $length (concat (map (\lambda a. vec-smult a (vec B)) (vec A))) = prod-list (dims$ 
     $A @ dims B)$ 

```

```

    proof -

```

```

      have  $\bigwedge xs. xs \in set (map (\lambda a. vec-smult a (vec B)) (vec A)) \implies length xs =$ 
     $length (vec B)$ 

```

**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**  $\text{concat} (\text{map } f (\text{concat } xss)) = \text{concat} (\text{map} (\lambda xs. \text{concat} (\text{map } f xs)) xss)$   
**by** (*induction xss; auto*)

**lemma** *subtensor-combine-tensor-prod*:

**assumes**  $\bigwedge A. A \in \text{set } As \implies \text{dims } A = ds$

**shows**  $\text{subtensor-combine } ds \ As \otimes B = \text{subtensor-combine} (ds @ \text{dims } B) (\text{map} (\lambda A. A \otimes B) As)$

**proof** –

**let** *?f* =  $\lambda a. \text{vec-smult } a (\text{Tensor.vec } B)$

**let** *?xss* =  $\text{map } \text{Tensor.vec } As$

**have** *1*:  $\text{prod-list} (\text{length } As \# ds) = \text{length} (\text{concat } ?xss)$  **by** (*metis assms length-vec subtensor-combine-dims subtensor-combine-vec*)

**have** *2*:  $\bigwedge A. A \in \text{set } As \implies \text{prod-list} (\text{dims } A @ \text{dims } B) = \text{length} (\text{concat} (\text{map } ?f (\text{Tensor.vec } A)))$

**by** (*metis dims-tensor-prod length-vec vec-tensor-prod*)

**have** *3*:  $\text{length } As \# ds @ \text{dims } B = (\text{length} (\text{map} (\lambda A. \text{tensor-from-vec} (\text{dims } A @ \text{dims } B)$

$(\text{concat} (\text{map} (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)))) As) \# ds @ \text{dims } B)$  **by** *simp*

**have** *4*:  $(\text{concat} (\text{map} (\lambda xs. \text{concat} (\text{map} (\lambda a. \text{vec-smult } a (\text{vec } B)) xs)) (\text{map } \text{vec } As)))$

$= (\text{concat} (\text{map } \text{vec} (\text{map} (\lambda A. \text{tensor-from-vec} (\text{dims } A @ \text{dims } B) (\text{concat} (\text{map} (\lambda a. \text{vec-smult } a (\text{vec } B)) (\text{vec } A)))) As)))$

**unfolding** *map-map[unfolded comp-def]* **using** *vec-tensor* **by** (*metis (no-types, lifting) 2 map-eq-conv*)

**have**  $\text{subtensor-combine } ds \ As \otimes B = \text{tensor-from-vec} (\text{length } As \# ds @ \text{dims } B) (\text{concat} (\text{map } ?f (\text{concat } ?xss)))$

**unfolding** *subtensor-combine-def tensor-prod-def* **using** *1* **by** *auto*

**also have**  $\dots = \text{tensor-from-vec} (\text{length } As \# ds @ \text{dims } B) (\text{concat} (\text{map} (\lambda xs. \text{concat} (\text{map } ?f xs)) ?xss))$

**using** *tensorprod-subtensor-base[of ?f ?xss]* **by** *auto*

**also have**  $\dots = \text{subtensor-combine} (ds @ \text{dims } B) (\text{map} (\lambda A. A \otimes B) As)$

**unfolding** *subtensor-combine-def tensor-prod-def* **using** *3 4* **by** *metis*

**finally show** *?thesis* **by** *metis*

**qed**

**lemma** *subtensor-tensor-prod*:

**assumes**  $\text{dims } A \neq []$  **and**  $i < \text{hd} (\text{dims } A)$

**shows**  $\text{subtensor} (A \otimes B) i = \text{subtensor } A \ i \otimes B$

**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:  $i < \text{length} (\text{map} (\lambda A. A \otimes B) As)$  using order-step by (simp add:
order-step.hyps order-step.prem1)
  have 2:  $(\bigwedge A. A \in \text{set} (\text{map} (\lambda A. A \otimes B) As)) \implies \text{dims } A = ds @ \text{dims } B$ 
using order-step by auto
  have subtensor (subtensor-combine ds As  $\otimes B$ )  $i = \text{subtensor} (\text{subtensor-combine}
(ds @ \text{dims } B) (\text{map} (\lambda A. A \otimes B) As)) i$ 
    using subtensor-combine-tensor-prod order-step by metis
  also have ... = As !  $i \otimes B$ 
    using order-step subtensor-subtensor-combine[of (map (lambda A. A otimes B) As) ds
@ dims B i] 1 2 by auto
  also have ... = subtensor (subtensor-combine ds As)  $i \otimes B$ 
    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:  $is1 \triangleleft \text{dims } A$  and is2-valid:  $is2 \triangleleft \text{dims } B$ 
shows lookup (A otimes 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 otimes 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 otimes 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.prem1(1)
by auto
next
  case (order-step A is1)
  then obtain i is1' where i # is1' = is1 by blast
  have lookup (subtensor A i otimes 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 otimes B) (is1 @ is2) = lookup A is1 * lookup B is2
    using lookup-subtensor1[of i is1' A] lookup-subtensor1[of i is1' @ is2 A otimes 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.prem1(1) valid-index-append valid-index-dimsE
  by metis
qed

```

```

lemma valid-index-split:
  assumes  $is \triangleleft ds1 @ ds2$ 
  obtains  $is1 is2$  where  $is1 @ is2 = is$   $is1 \triangleleft ds1$   $is2 \triangleleft ds2$ 
proof
  assume  $a: \bigwedge is1 is2. is1 @ is2 = is \implies is1 \triangleleft ds1 \implies is2 \triangleleft ds2 \implies thesis$ 
  have  $length-is: length is = length ds1 + length ds2$  using valid-index-length
using  $assms$  by auto
  show  $take (length ds1) is \triangleleft ds1$ 
  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  $drop (length ds1) is \triangleleft ds2$ 
  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  $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ 
proof (rule tensor-lookup-eqI, simp)
  fix  $is$  assume  $is \triangleleft dims ((A \otimes B) \otimes C)$ 
  obtain  $is1 is23$  where  $is1 \triangleleft dims A$   $is23 \triangleleft dims (B \otimes C)$   $is1 @ is23 = is$ 
  by (metis (mono-tags, lifting)  $\langle is \triangleleft dims ((A \otimes B) \otimes C) \rangle$  Tensor-Product.dims-tensor-prod
append-assoc valid-index-split)
  obtain  $is2 is3$  where  $is2 \triangleleft dims B$   $is3 \triangleleft dims C$   $is2 @ is3 = is23$ 
  by (metis  $\langle is23 \triangleleft dims (local.tensor-prod-otimes B C) \rangle$  dims-tensor-prod
valid-index-split)
  def  $is12 == is1 @ is2$ 
  have  $is12 \triangleleft dims (A \otimes B)$  by (simp add:  $\langle is1 \triangleleft dims A \rangle \langle is2 \triangleleft dims B \rangle$ 
 $is12-def$  valid-index-append)
  have  $is12 @ is3 = is$  by (simp add:  $\langle is1 @ is23 = is \rangle \langle is2 @ is3 = is23 \rangle$ 
 $is12-def$ )
  show  $lookup ((A \otimes B) \otimes C) is = lookup (A \otimes (B \otimes C)) is$ 
  unfolding lookup-tensor-prod[OF  $\langle is1 \triangleleft dims A \rangle \langle is23 \triangleleft dims (B \otimes C) \rangle$ ,
unfolded  $\langle is1 @ is23 = is \rangle$ ]
  lookup-tensor-prod[OF  $\langle is12 \triangleleft dims (A \otimes B) \rangle \langle is3 \triangleleft dims C \rangle$ , unfolded
 $\langle is12 @ is3 = is \rangle$ ]
  using  $\langle is1 \triangleleft dims A \rangle \langle is2 @ is3 = is23 \rangle \langle is2 \triangleleft dims B \rangle \langle is3 \triangleleft dims C \rangle$ 
 $is12-def$  mult.assoc by fastforce
  qed
  qed

```

end

lemma tensor-prod-distr-left:

**assumes**  $\text{dims } A = \text{dims } B$   
**shows**  $(A + B) \otimes C = (A \otimes C) + (B \otimes C)$   
**proof** –  
**have**  $\bigwedge is. is \triangleleft \text{dims } A @ \text{dims } C \implies \text{lookup } ((A + B) \otimes C) \text{ is} = \text{lookup } (A \otimes C + B \otimes C) \text{ is}$   
**proof** –  
**fix**  $is$  **assume**  $is \triangleleft \text{dims } A @ \text{dims } C$   
**obtain**  $is1 \ is2$  **where**  $is = is1 @ is2$   $is1 \triangleleft \text{dims } A$   $is2 \triangleleft \text{dims } C$  **using** *valid-index-split* **using**  $\langle is \triangleleft \text{dims } A @ \text{dims } C \rangle$  **by** *blast*  
**then show**  $\text{lookup } ((A + B) \otimes C) \text{ is} = \text{lookup } ((A \otimes C) + (B \otimes C)) \text{ is}$   
**using** *lookup-plus*  
 $\langle is1 \triangleleft \text{dims } A \rangle \langle is2 \triangleleft \text{dims } C \rangle$  *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distrib(2) valid-index-append*  
**by** *fastforce*  
**qed**  
**moreover have**  $\text{tensor-from-lookup } (\text{dims } A @ \text{dims } C) (\text{lookup } ((A + B) \otimes C)) = (A + B) \otimes C$   
 $\text{tensor-from-lookup } (\text{dims } A @ \text{dims } C) (\text{lookup } ((A \otimes C) + (B \otimes C))) = (A \otimes C) + (B \otimes C)$   
**by**  $(\text{metis } (\text{no-types, lifting}) \text{ assms plus-dim1 dims-tensor-prod tensor-lookup})+$   
**ultimately show** *?thesis* **using** *tensor-from-lookup-eq1*  
**by**  $(\text{metis } \langle \bigwedge is. is \triangleleft \text{dims } A @ \text{dims } C \implies \text{lookup } ((A + B) \otimes C) \text{ is} = \text{lookup } (A \otimes C + B \otimes C) \text{ is} \rangle)$   
**qed**

**lemma** *tensor-prod-distr-right*:

**assumes**  $\text{dims } A = \text{dims } B$   
**shows**  $C \otimes (A + B) = (C \otimes A) + (C \otimes B)$   
**proof** –  
**have**  $\bigwedge is. is \triangleleft \text{dims } C @ \text{dims } A \implies \text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } (C \otimes A + C \otimes B) \text{ is}$   
**proof** –  
**fix**  $is$  **assume**  $is \triangleleft \text{dims } C @ \text{dims } A$   
**obtain**  $is1 \ is2$  **where**  $is = is1 @ is2$   $is1 \triangleleft \text{dims } C$   $is2 \triangleleft \text{dims } A$  **using** *valid-index-split* **using**  $\langle is \triangleleft \text{dims } C @ \text{dims } A \rangle$  **by** *blast*  
**then show**  $\text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } ((C \otimes A) + (C \otimes B)) \text{ is}$   
**using** *lookup-plus*  
**using**  $\langle is2 \triangleleft \text{dims } A \rangle \langle is1 \triangleleft \text{dims } C \rangle$  *assms plus-dim1 dims-tensor-prod lookup-tensor-prod ring-class.ring-distrib(1) valid-index-append*  
**by** *fastforce*  
**qed**  
**moreover have**  $\text{tensor-from-lookup } (\text{dims } C @ \text{dims } A) (\text{lookup } (C \otimes (A + B))) = C \otimes (A + B)$   
 $\text{tensor-from-lookup } (\text{dims } C @ \text{dims } A) (\text{lookup } ((C \otimes A) + (C \otimes B))) = (C \otimes A) + (C \otimes B)$   
**by**  $(\text{metis } (\text{no-types, lifting}) \text{ assms plus-dim1 dims-tensor-prod tensor-lookup})+$   
**ultimately show** *?thesis* **using** *tensor-from-lookup-eq1*  
**by**  $(\text{metis } \langle \bigwedge is. is \triangleleft \text{dims } C @ \text{dims } A \implies \text{lookup } (C \otimes (A + B)) \text{ is} = \text{lookup } (C \otimes A + C \otimes B) \text{ is} \rangle)$

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**  $\text{dims } (a \cdot (A \otimes B)) = \text{dims } ((a \cdot A) \otimes B)$  **by** *simp*

**fix** *is* **assume**  $is \triangleleft \text{dims } (a \cdot (A \otimes B))$

**then have**  $is \triangleleft \text{dims } (A \otimes B)$  **by** *auto*

**then obtain** *is1 is2* **where**  $is1 \triangleleft \text{dims } A$   $is2 \triangleleft \text{dims } B$   $is = is1 @ is2$  **by**  
(*metis dims-tensor-prod valid-index-split*)

**then have**  $is1 \triangleleft \text{dims } (a \cdot A)$  **by** *auto*

**show**  $\text{lookup } (a \cdot (A \otimes B)) \text{ is} = \text{lookup } (a \cdot A \otimes B) \text{ is}$

**using** *lookup-tensor-prod*[OF  $\langle is1 \triangleleft \text{dims } A \rangle \langle is2 \triangleleft \text{dims } B \rangle$ ] *lookup-tensor-prod*[OF  
 $\langle is1 \triangleleft \text{dims } (a \cdot A) \rangle \langle is2 \triangleleft \text{dims } B \rangle$ ]

*lookup-smult*[OF  $\langle is \triangleleft \text{dims } (A \otimes B) \rangle$ ] *lookup-smult*[OF  $\langle is1 \triangleleft \text{dims } A \rangle \langle is$   
 $= is1 @ is2 \rangle$ ] **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**  $is \triangleleft dims (a \cdot (A \otimes B))$   
**then have**  $is \triangleleft dims (A \otimes B)$  **by** *auto*  
**then obtain**  $is1\ is2$  **where**  $is1 \triangleleft dims\ A\ is2 \triangleleft dims\ B$   $is = is1\ @\ is2$  **by**  
(*metis dims-tensor-prod valid-index-split*)  
**then have**  $is2 \triangleleft dims (a \cdot B)$  **by** *auto*  
**show**  $lookup (a \cdot (A \otimes B))\ is = lookup (A \otimes (a \cdot B))\ is$   
**using**  $lookup\ tensor\ prod[OF\ \langle is1 \triangleleft dims\ A \rangle\ \langle is2 \triangleleft dims\ B \rangle]\ lookup\ tensor\ prod[OF$   
 $\langle is1 \triangleleft dims\ A \rangle\ \langle is2 \triangleleft dims\ (a \cdot B) \rangle]$   
 $lookup\ smult[OF\ \langle is \triangleleft dims\ (A \otimes B) \rangle]\ lookup\ smult[OF\ \langle is2 \triangleleft dims\ B \rangle]\ \langle is$   
 $= is1\ @\ is2 \rangle$  **by** *simp*  
**qed**

**lemma** *order-0-multiple-of-one:*

**assumes**  $order\ A = 0$

**obtains**  $a$  **where**  $A = a \cdot 1$

**proof**

**assume**  $(\bigwedge a. A = a \cdot 1 \implies thesis)$

**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**  $\langle vec\ A = [a] \rangle$  **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**  $is \triangleleft dims (A \otimes tensor0\ ds)$

**then obtain**  $is1\ is2$  **where**  $is1 \triangleleft dims\ A\ is2 \triangleleft dims (tensor0\ ds)$   $is = is1\ @$   
 $is2$

**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) \langle is \triangleleft dims (A \otimes tensor0\ ds) \rangle dims-tensor0*)

*dims-tensor-prod lookup-tensor0 lookup-tensor-prod mult-zero-right*)  
**qed**

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

**lemma** *subtensor-prod-with-vec*:  
**assumes**  $\text{order } A = 1$   $i < \text{hd } (\text{dims } A)$   
**shows**  $\text{subtensor } (A \otimes B) \text{ } i = \text{lookup } A \text{ } [i] \cdot B$   
**proof** (*rule tensor-lookup-eqI*)  
**have**  $\text{dims } (A \otimes B) \neq []$  **using** *assms(1)* **by** *auto*  
**have**  $\text{hd } (\text{dims } A) = \text{hd } (\text{dims } (A \otimes B))$   
**by** (*metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(1)*)  
**show**  $\text{dims } (\text{subtensor } (A \otimes B) \text{ } i) = \text{dims } (\text{lookup } A \text{ } [i] \cdot B)$   
**unfolding** *dims-smult dims-subtensor[OF ‹dims (A ‹otimes B)› ‹neq []› ‹i < hd (dims A)›][unfolding ‹hd (dims A) = hd (dims (A ‹otimes B))›]*  
**by** (*metis One-nat-def Suc-length-conv append-self-conv2 assms(1) dims-tensor-prod length-0-conv list.sel(3)*)  
**next**  
**fix** *is* **assume**  $is \triangleleft \text{dims } (\text{subtensor } (A \otimes B) \text{ } i)$   
**have**  $\text{dims } (A \otimes B) \neq []$  **using** *assms(1)* **by** *auto*  
**have**  $\text{hd } (\text{dims } A) = \text{hd } (\text{dims } (A \otimes B))$   
**by** (*metis One-nat-def Suc-length-conv append-Cons assms(1) dims-tensor-prod list.sel(1)*)  
**then have**  $is \triangleleft \text{dims } B$   
**using**  $\langle is \triangleleft \text{dims } (\text{subtensor } (A \otimes B) \text{ } i) \rangle$  [*unfolding dims-subtensor[OF ‹dims (A ‹otimes B)› ‹neq []› ‹i < hd (dims A)›][unfolding ‹hd (dims A) = hd (dims (A ‹otimes 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**  $[i] \triangleleft \text{dims } A$  **using** *assms* **by** (*metis One-nat-def Suc-length-conv length-0-conv list.sel(1) valid-index.Nil valid-index.simps*)  
**then have**  $i \# is \triangleleft \text{dims } (A \otimes B)$  **using**  $\langle is \triangleleft \text{dims } (\text{subtensor } (A \otimes B) \text{ } i) \rangle$  *dims-subtensor valid-index.Cons* **by** *auto*  
**then show**  $\text{lookup } (\text{subtensor } (A \otimes B) \text{ } i) \text{ } is = \text{lookup } (\text{lookup } A \text{ } [i] \cdot B) \text{ } is$   
**unfolding** *lookup-subtensor1[OF ‹i ‹# is ‹triangleleft dims (A ‹otimes B)›]*  
**using** *lookup-tensor-prod[OF ‹[i] ‹triangleleft dims A› ‹is ‹triangleleft dims B›]* *lookup-smult ‹is ‹triangleleft 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  $\Rightarrow$  nat  $\Rightarrow$  'a::ring-1 tensor
where unit-vec n i = tensor-from-lookup [n] ( $\lambda x.$  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]  $\triangleleft$  [n] by (simp add: assms valid-index.Cons valid-index.Nil)
  then have lookup (unit-vec n i) [j] = ( $\lambda x.$  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  $\otimes$  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] ( $\lambda x.$  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  $\neq$  []
shows listsum (dims A) (map ( $\lambda i.$  unit-vec (hd (dims A)) i  $\otimes$  subtensor A i)
[0.. $\text{hd}$  (dims A)]) = A (is ?LS = A)
proof -
  let ?f =  $\lambda i.$  unit-vec (hd (dims A)) i  $\otimes$  subtensor A i

```

```

have correct-dims:  $\bigwedge B. B \in \text{set } (\text{map } ?f [0..<hd (dims A)]) \implies \text{dims } B = \text{dims } A$ 
proof –
  fix B
  assume  $B \in \text{set } (\text{map } ?f [0..<hd (dims A)])$ 
  then obtain  $i$  where  $B : B = ?f i$  and  $i < hd (dims A)$  by auto
  then have  $\text{dims } (\text{subtensor } A i) = tl (dims A)$  using dims-subtensor using
assms by blast
  then show  $\text{dims } B = \text{dims } A$  unfolding B
  by (metis append-Cons assms dims-tensor-prod dims-unit-vec list.exhaust-sel
self-append-conv2)
  qed
  have  $\bigwedge j. j < hd (dims A) \implies \text{subtensor } ?LS j = \text{subtensor } A j$ 
  proof –
    fix j
    assume  $j < hd (dims A)$ 
    have  $1 : \text{subtensor } ?LS j = \text{listsum } (tl (dims A)) (\text{map } (\lambda A. \text{subtensor } A j) (\text{map } ?f [0..<hd (dims A)]))$ 
    using subtensor-listsum[of ( $\text{map } (\lambda i. ?f i) [0..<hd (dims A)]$ )]  $\text{dims } A j$ , OF
correct-dims assms  $\langle j < hd (dims A) \rangle$ 
    by linarith
    also have  $\dots = \text{listsum } (tl (dims A)) (\text{map } (\lambda i. \text{subtensor } (?f i) j) [0..<hd (dims A)])$ 
    proof –
      have  $\text{map } (\lambda A. \text{subtensor } A j) (\text{map } ?f [0..<hd (dims A)]) = \text{map } (\lambda i. \text{subtensor } (?f i) j) [0..<hd (dims A)]$ 
      unfolding map-map[of ( $\lambda A. \text{subtensor } A j$ )]  $?f [0..<hd (dims A)]$  by simp
      with 1 show ?thesis by metis
    qed
    also have  $\dots = \text{map } (\lambda i. \text{if } i = j \text{ then } \text{subtensor } A i \text{ else } \text{tensor0 } (\text{dims } (\text{subtensor } A i))) [0..<hd (dims A)] ! j$ 
    unfolding subtensor-prod-with-unit-vec[OF  $\langle j < hd (dims A) \rangle$ ]
    using listsum-all-0-but-one[of  $j$  ( $\text{map } (\lambda i. \text{if } i = j \text{ then } \text{subtensor } A i \text{ else } \text{tensor0 } (\text{dims } (\text{subtensor } A i))) [0..<hd (dims A)]$ )]  $tl (dims A)$ 
    by (simp add:  $\langle j < hd (dims A) \rangle$  assms)
    also have  $\dots = \text{subtensor } A j$  by (simp add:  $\langle j < hd (dims A) \rangle$ )
    finally show  $\text{subtensor } ?LS j = \text{subtensor } A j$  by auto
  qed
  moreover have  $\text{dims } ?LS = \text{dims } A$  using correct-dims listsum-dims by blast
  ultimately show ?thesis using subtensor-eqI by (metis (no-types, lifting)
assms)
qed

```

## 7 Tensor CP-Rank

```

theory Tensor-Rank
imports Tensor-Unit-Vec

```

**begin**

**inductive** *cprank-max1::'a::ring-1 tensor*  $\Rightarrow$  *bool* **where**

*order1: order*  $A \leq 1 \Rightarrow$  *cprank-max1*  $A$  |

*higher-order: order*  $A = 1 \Rightarrow$  *cprank-max1*  $B \Rightarrow$  *cprank-max1*  $(A \otimes B)$

**lemma** *cprank-max1-order0: cprank-max1*  $B \Rightarrow$  *order*  $A = 0 \Rightarrow$  *cprank-max1*  $(A \otimes B)$

**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*  $A \leq 0 \Rightarrow$  *cprank-max1*  $B \Rightarrow$  *cprank-max1*  $(A \otimes B)$

**by** (*simp* *add: cprank-max1-order0*)

**lemma** *cprank-max1-prod: cprank-max1*  $A \Rightarrow$  *cprank-max1*  $B \Rightarrow$  *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**  $\bigwedge B. B \in \text{set } Bs \Rightarrow$  *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**  $A::'a::\text{comm-ring-1}$  *tensor*

**assumes** *cprank-max1*  $A$

**obtains**  $Bs$   $a$  **where**  $\bigwedge B. B \in \text{set } Bs \Rightarrow$  *order*  $B = 1$   $a \cdot \text{prod-list } Bs = A$

**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**  $A = a \cdot \text{prod-list } []$  **using** *order-0-multiple-of-one* **using** *prod-list.Nil* **by** *auto*

**then show** *?thesis* **using** *length-pos-if-in-set* *order1.premis* **by** *fastforce*

**next**

**case** *False*

```

    then have order A = 1 using order1 by linarith
    then have A = 1 · 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.premis set-ConsD)
  qed
next
  case (higher-order A B)
  then obtain Bs b where (∧B'. B' ∈ set Bs ⇒ order B' = 1) b · prod-list Bs
= B by metis
  then have (∧B. B ∈ set (A # Bs) ⇒ order B = 1) using higher-order by
auto
  have A ⊗ B = b · (A ⊗ prod-list Bs) using smult-prod-extract2 ⟨b · prod-list
Bs = B⟩ by metis
  then show ?case by (metis ⟨∧Ba. Ba ∈ set (A # Bs) ⇒ order Ba = 1⟩
higher-order.premis 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.prem*s **by** *fastforce*  
**next**  
**case** (*cprank-max-Suc A B j*)  
**then obtain** *BS* **where** *BS-def:( $\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B$ ) ( $\bigwedge B'. B' \in \text{set } BS \implies \text{dims } B' = \text{dims } B$ )*  
*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.prem*s  
*set-ConsD*  
**by** (*metis plus-dim1 cprank-max-Suc.hyps(1)*)  
**qed**

**lemma** *cprank-maxI*:  
**assumes**  $\bigwedge B. B \in \text{set } BS \implies \text{cprank-max1 } B$   
**and**  $\bigwedge B. B \in \text{set } BS \implies \text{dims } B = ds$   
**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  $\implies$  A = tensor0 (dims A)* **by** (*metis dims-tensor0 length-0-conv cprank-max0 cprank-maxE*)

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

**lemma** *cprank-max-prod-order1*:

**assumes**  $order\ A = 1$   
**and**  $cprank-max\ n\ B$   
**shows**  $cprank-max\ n\ (A \otimes B)$   
**proof** –  
**obtain**  $CS$  **where**  $(\bigwedge C. C \in set\ CS \implies cprank-max1\ C)$   
**and**  $(\bigwedge C. C \in set\ CS \implies dims\ C = dims\ B)$   
**and**  $listsum\ (dims\ B)\ CS = B$   
**and**  $length\ CS = n$   
**using**  $assms(2)\ cprank-maxE$  **by**  $metis$   
**def**  $CS' == map\ (\lambda C. A \otimes C)\ CS$   
**then have**  $\bigwedge C'. C' \in set\ CS' \implies cprank-max1\ C'$   
**using**  $assms(1)\ higher-order\ \langle \bigwedge C. C \in set\ CS \implies cprank-max1\ C \rangle\ imageE$   
 $set-map$  **by**  $auto$   
**have**  $listsum\ (dims\ A @ dims\ B)\ CS' = A \otimes B$  **using**  $CS'-def\ \langle Tensor-Plus.listsum\ (dims\ B)\ CS = B \rangle$   
**using**  $\langle \bigwedge Ca. Ca \in set\ CS \implies dims\ Ca = dims\ B \rangle\ listsum-prod-distr-right$  **by**  
 $fastforce$   
**then show**  $?thesis$  **by**  $(metis\ (mono-tags,\ lifting)\ CS'-def\ \langle \bigwedge C'. C' \in set\ CS' \implies cprank-max1\ C' \rangle\ \langle \bigwedge Ca. Ca \in set\ CS \implies dims\ Ca = dims\ B \rangle\ \langle length\ CS = n \rangle\ dims-tensor-prod\ imageE\ length-map\ cprank-maxI\ set-map)$   
**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**  $Bs == map\ (\lambda i. unit-vec\ (hd\ (dims\ A))\ i \otimes subtensor\ A\ i)\ [0..<hd\ (dims\ A)]$

**have**  $\bigwedge B. B \in set\ Bs \implies dims\ A = dims\ B$

**proof** –

**fix**  $B$  **assume**  $B \in set\ Bs$

**obtain**  $i$  **where**  $i < hd\ (dims\ A)\ Bs!i=B$  **using**  $Bs-def\ \langle B \in set\ Bs \rangle$  **by**  $auto$

**then have**  $dims\ (unit-vec\ (hd\ (dims\ A))\ i \otimes subtensor\ A\ i) = dims\ A$

**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**  $Bs-def\ \langle Bs\ !\ i = B \rangle\ \langle i < hd\ (dims\ A) \rangle$

**by**  $auto$

**qed**

**have**  $\bigwedge B. B \in set\ Bs \implies cprank-max\ (prod-list\ (tl\ (dims\ A)))\ B$

**proof** –

**fix**  $B$  **assume**  $B \in set\ Bs$

**obtain**  $i$  **where**  $i < hd\ (dims\ A)\ Bs!i=B$  **using**  $Bs-def\ \langle B \in set\ Bs \rangle$  **by**  $auto$

**then have**  $cprank-max\ (prod-list\ (tl\ (dims\ A)))\ (unit-vec\ (hd\ (dims\ A))\ i \otimes subtensor\ A\ i)$

**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 ≠ 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 {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 F (vs (span A))*  
**proof** –  
  **have** *vectorspace F (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 F (span A) V* **using** *assms span-is-submodule* **by** *simp*  
  **have** *LinearCombinations.module.span F (vs (span A)) A = carrier (vs (span A))*  
    **using** *span-li-not-depend(1)[OF ⟨A ⊆ span A⟩ ⟨submodule F (span A) V⟩]* **by** *auto*  
  **then show** *?thesis unfolding vectorspace.fin-dim-def[OF ⟨vectorspace F (span-vs A)⟩]*  
    **using** *List.finite-set ⟨A ⊆ span A⟩ ⟨vectorspace F (vs (span A))⟩*  
      *vec-vs vectorspace.carrier-vs-is-self[OF ⟨vectorspace F (span-vs A)⟩]* **using** *assms(1)* **by** *auto*  
**qed**

**lemma** (in *vec-space*) *fin-dim-span-cols*:  
**assumes** *A* ∈ *carrier<sub>m</sub> n nc*  
**shows** *vectorspace.fin-dim F (vs (span (set (cols A))))*  
**using** *fin-dim-span cols-dim List.finite-set assms mat-carrierD(1) vec-module-simps(3)*  
**by** *force*

end

## 10 Missing Lemmas of List

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

```
lemma nth-map-zip:
assumes  $i < \text{length } xs$ 
assumes  $i < \text{length } ys$ 
shows  $\text{map } f (\text{zip } xs \ ys) ! i = f (xs ! i, ys ! i)$ 
using nth-zip nth-map length-zip by (simp add: assms(1) assms(2))
```

```
lemma nth-map-zip2:
assumes  $i < \text{length } (\text{map } f (\text{zip } xs \ ys))$ 
shows  $\text{map } f (\text{zip } xs \ ys) ! i = f (xs ! i, ys ! i)$ 
using nth-zip nth-map length-zip assms by simp
```

```
fun find-first where
find-first  $a [] = \text{undefined}$  |
find-first  $a (x \# xs) = (\text{if } x = a \text{ then } 0 \text{ else } \text{Suc } (\text{find-first } a \ xs))$ 
```

```
lemma find-first-le:
assumes  $a \in \text{set } xs$ 
shows  $\text{find-first } a \ xs < \text{length } xs$ 
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  $a \in \text{set } xs$ 
shows  $xs ! (\text{find-first } a \ xs) = a$ 
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  $i < \text{length } xs$ 
```

```

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  $S \subseteq \text{carrier } V$ 
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  $S \subseteq \text{carrier } (\text{span-vs } S)$  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  $S \subseteq \text{carrier } V$ 

```

**and** *finite*  $S$   
**and** *maximal*  $U$  ( $\lambda T. T \subseteq S \wedge \text{lin-indpt } T$ )  
**shows**  $\text{vectorspace.dim } K (\text{span-vs } S) = \text{card } U$   
**proof** –  
**have** *lin-indpt*  $U \subseteq S$  **by** (*metis* *assms*(3) *maximal-def*)+  
**then have**  $U \subseteq \text{span } S$  **using** *in-own-span*[*OF* *assms*(1)] **by** *blast*  
**then have** *lin-indpt*: *LinearCombinations.module.lin-indpt*  $K (\text{span-vs } S) U$   
**using** *module.span-li-not-depend*(2)[*OF*  $\langle U \subseteq \text{span } S \rangle$ ]  $\langle \text{lin-indpt } U \rangle$  *assms*(1)  
*span-is-submodule* **by** *blast*  
**have**  $\text{span } U = \text{span } S$   
**proof** (*rule ccontr*)  
**assume**  $\text{span } U \neq \text{span } S$   
**have**  $\text{span } U \subseteq \text{span } S$  **using** *span-is-monotone*  $\langle U \subseteq S \rangle$  **by** *metis*  
**then have**  $\neg S \subseteq \text{span } U$  **by** (*meson*  $\langle U \subseteq S \rangle$   $\langle \text{span } U \neq \text{span } S \rangle$  *assms*(1))  
*span-is-submodule*  
*span-is-subset subset-antisym subset-trans*  
**then obtain**  $s$  **where**  $s \in S$   $s \notin \text{span } U$  **by** *blast*  
**then have** *lin-indpt*  $(U \cup \{s\})$  **using** *lindep-span*  
**by** (*meson*  $\langle U \subseteq S \rangle$   $\langle \text{lin-indpt } U \rangle$  *assms*(1) *lin-dep-iff-in-span set-rev-mp*  
*span-mem subset-trans*)  
**have**  $s \notin U$  **using**  $\langle U \subseteq S \rangle$   $\langle s \notin \text{span } U \rangle$  *assms*(1) *span-mem* **by** *auto*  
**then have**  $(U \cup \{s\}) \subseteq S \wedge \text{lin-indpt } (U \cup \{s\})$  **using**  $\langle U \subseteq S \rangle$   $\langle \text{lin-indpt } (U \cup \{s\}) \rangle$   $\langle s \in S \rangle$  **by** *auto*  
**then have**  $\neg \text{maximal } U$  ( $\lambda T. T \subseteq S \wedge \text{lin-indpt } T$ )  
**unfolding** *maximal-def* **using** *Un-subset-iff*  $\langle s \notin U \rangle$  *insert-subset order-refl*  
**by** *auto*  
**then show** *False* **using** *assms* **by** *metis*  
**qed**  
**then have** *span*:*LinearCombinations.module.span*  $K (\text{vs } (\text{span } S)) U = \text{span } S$   
**using** *module.span-li-not-depend*[*OF*  $\langle U \subseteq \text{span } S \rangle$ ]  
**by** (*simp add*: *LinearCombinations.module.span-is-submodule* *assms*(1) *module-axioms*)  
**have** *vectorspace*  $K (\text{vs } (\text{span } S))$   
**using** *field.field-axioms* *vectorspace-def* *submodule-is-module*[*OF* *span-is-submodule*[*OF* *assms*(1)]] **by** *metis*  
**then have** *vectorspace.basis*  $K (\text{vs } (\text{span } S)) U$  **using** *vectorspace.basis-def*[*OF*  $\langle \text{vectorspace } K (\text{vs } (\text{span } S)) \rangle$ ]  
**by** (*simp add*: *span*  $\langle U \subseteq \text{span } S \rangle$  *lin-indpt*)  
**then show** *?thesis*  
**using**  $\langle U \subseteq S \rangle$   $\langle \text{vectorspace } K (\text{vs } (\text{span } S)) \rangle$  *assms*(2) *infinite-super* *vectorspace.dim-basis* **by** *blast*  
**qed**

**definition** (*in* *vec-space*) *rank* :: 'a mat  $\Rightarrow$  nat  
**where**  $\text{rank } A = \text{vectorspace.dim } F (\text{span-vs } (\text{set } (\text{cols } A)))$

**lemma** (*in* *vec-space*) *rank-card-indpt*:  
**assumes**  $A \in \text{carrier}_m n nc$   
**assumes** *maximal*  $S$  ( $\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T$ )  
**shows**  $\text{rank } A = \text{card } S$

**proof** –  
**have**  $\text{set } (\text{cols } A) \subseteq \text{carrier}_v \ n$  **using**  $\text{cols-dim } \text{assms}(1)$  **by** *blast*  
**have**  $\text{finite } (\text{set } (\text{cols } A))$  **by** *blast*  
**show**  $?thesis$  **using**  $\text{dim-span}[OF \langle \text{set } (\text{cols } A) \subseteq \text{carrier}_v \ n \rangle \langle \text{finite } (\text{set } (\text{cols } A)) \rangle \text{assms}(2)]$   
**unfolding**  $\text{rank-def}$  **by** *blast*  
**qed**

**lemma** *maximal-exists-superset*:

**assumes**  $\text{finite } S$   
**assumes**  $\text{maxc}: \bigwedge A. P \ A \implies A \subseteq S \ \text{and } P \ B$   
**shows**  $\exists A. \text{finite } A \wedge \text{maximal } A \ P \wedge B \subseteq A$   
**proof** –  
**have**  $\text{finite } (S - B)$  **using**  $\text{assms}(1) \ \text{assms}(3) \ \text{infinite-super } \text{maxc}$  **by** *blast*  
**then show**  $?thesis$  **using**  $\langle P \ B \rangle$   
**proof** (*induction*  $S - B$  *arbitrary*: $B$  *rule*: *finite-psubset-induct*)  
**case** ( $\text{psubset } B$ )  
**then show**  $?case$   
**proof** (*cases*  $\text{maximal } B \ P$ )  
**case** *True*  
**then show**  $?thesis$  **using**  $\text{order-refl } \text{psubset.hyps}$  **by** ( $\text{metis } \text{assms}(1) \ \text{maxc}$   $\text{psubset.prem1 rev-finite-subset}$ )  
**next**  
**case** *False*  
**then obtain**  $B'$  **where**  $B \subset B' \ P \ B'$  **using**  $\text{maximal-def } \text{psubset.prem1}$  **by** ( $\text{metis } \text{dual-order.order-iff-strict}$ )  
**then have**  $B' \subseteq S \ B \subseteq S$  **using**  $\text{maxc } \langle P \ B \rangle$  **by** *auto*  
**then have**  $S - B' \subset S - B$  **using**  $\langle B \subset B' \rangle$  **by** *blast*  
**then show**  $?thesis$  **using**  $\text{psubset}(2)[OF \langle S - B' \subset S - B \rangle \langle P \ B' \rangle]$  **using**  $\langle B \subset B' \rangle$  **by** *fast*  
**qed**  
**qed**  
**qed**

**lemma** (*in*  $\text{vec-space}$ ) *rank-ge-card-indpt*:

**assumes**  $A \in \text{carrier}_m \ n \ nc$   
**assumes**  $U \subseteq \text{set } (\text{cols } A)$   
**assumes**  $\text{lin-indpt } U$   
**shows**  $\text{rank } A \geq \text{card } U$   
**proof** –  
**obtain**  $S$  **where**  $\text{maximal } S \ (\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T) \ U \subseteq S \ \text{finite } S$   
**using**  $\text{maximal-exists-superset}[of \ \text{set } (\text{cols } A) \ (\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T) \ U]$   
**using**  $\text{List.finite-set } \text{assms}(2) \ \text{assms}(3) \ \text{maximal-exists-superset}$  **by** *blast*  
**then show**  $?thesis$   
**unfolding**  $\text{rank-card-indpt}[OF \langle A \in \text{carrier}_m \ n \ nc \rangle \langle \text{maximal } S \ (\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T) \rangle]$   
**using**  $\text{card-mono}$  **by** *blast*  
**qed**

**lemma** (in *vec-space*) *lin-indpt-full-rank*:  
**assumes**  $A \in \text{carrier}_m \ n \ nc$   
**assumes** *distinct* (cols  $A$ )  
**assumes** *lin-indpt* (set (cols  $A$ ))  
**shows**  $\text{rank } A = nc$   
**proof** –  
  **have** *maximal* (set (cols  $A$ )) ( $\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T$ )  
  **by** (*simp add: assms(3) maximal-def subset-antisym*)  
  **then have**  $\text{rank } A = \text{card } (\text{set } (\text{cols } A))$  **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**  $A \in \text{carrier}_m \ n \ nc$   
**shows**  $\text{rank } A \leq nc$   
**proof** –  
  **obtain**  $S$  **where** *maximal*  $S$  ( $\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T$ )  
  **using** *maximal-exists*[of ( $\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T$ ) *card* (set (cols  $A$ )) {}]  
  **by** (*meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2 rev-finite-subset*)  
  **then have**  $\text{card } S \leq \text{card } (\text{set } (\text{cols } A))$  **by** (*simp add: card-mono maximal-def*)  
  **then have**  $\text{card } S \leq nc$   
  **using** *assms(1) cols-length card-length mat-carrierD(2)* **by** (*metis dual-order.trans*)  
  **then show** *?thesis*  
  **using** *rank-card-indpt*[OF  $\langle A \in \text{carrier}_m \ n \ nc \rangle \langle \text{maximal } S \ (\lambda T. T \subseteq \text{set } (\text{cols } A) \wedge \text{lin-indpt } T) \rangle]$   
  **by** *simp*  
**qed**

**lemma** (in *vec-space*) *full-rank-lin-indpt*:  
**assumes**  $A \in \text{carrier}_m \ n \ nc$   
**assumes**  $\text{rank } A = nc$   
**assumes** *distinct* (cols  $A$ )  
**shows** *lin-indpt* (set (cols  $A$ ))  
**proof** –  
  **have**  $1: \text{set } (\text{cols } A) \subseteq \text{carrier}_v \ n$  **using** *assms(1) cols-dim* **by** *blast*  
  **have**  $2: \text{finite } (\text{set } (\text{cols } A))$  **by** *simp*  
  **have**  $\text{card } (\text{set } (\text{cols } A)) = nc$   
  **using** *assms(1) assms(3) distinct-card* **by** *fastforce*  
  **have**  $3: \text{vector\_space.dim } F \ (\text{span-}vs \ (\text{set } (\text{cols } A))) = \text{card } (\text{set } (\text{cols } A))$   
  **using**  $\langle \text{rank } A = nc \rangle$ [*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**  $A \in \text{carrier}_m n nc$   
**assumes** *distinct* (cols  $A$ )  
**shows**  $A \otimes_{mv} (\text{vec } nc (\lambda i. a (\text{col } A i))) = \text{lincomb } a (\text{set } (\text{cols } A))$   
**proof** (rule *vec-eqI*)  
  **have** *finite* (set (cols  $A$ )) **using** *assms(1)* **by** *simp*  
  **then show**  $\dim_v (A \otimes_{mv} (\text{vec } nc (\lambda i. a (\text{col } A i)))) = \dim_v (\text{lincomb } a (\text{set } (\text{cols } A)))$   
  **using** *assms cols-dim vec-space.lincomb-dim* **by** (metis *dim-mat-mult-vec mat-carrierD(1)*)  
  **fix**  $i$  **assume**  $i < \dim_v (\text{lincomb } a (\text{set } (\text{cols } A)))$   
  **then have**  $i < n$  **using**  $\langle \dim_v (A \otimes_{mv} (\text{vec } nc (\lambda i. a (\text{col } A i)))) = \dim_v (\text{lincomb } a (\text{set } (\text{cols } A))) \rangle$  *assms* **by** *auto*  
  **have**  $\text{set } (\text{cols } A) \subseteq \text{carrier}_v n$  **using** *cols-dim*  $\langle A \in \text{carrier}_m n nc \rangle$  *mat-carrierD(1)*  
**by** *blast*  
  **have** *bij-betw* (*nth* (cols  $A$ )  $\{..<\text{length } (\text{cols } A)\}$ ) (set (cols  $A$ ))  
  **unfolding** *bij-betw-def* **by** (rule *conjI*, *simp add: inj-on-nth*  $\langle \text{distinct } (\text{cols } A) \rangle$ ;  
  *metis subset-antisym in-set-conv-nth lessThan-iff rev-image-eqI subsetI*  
  *image-subsetI lessThan-iff nth-mem*)  
  **then have**  $(\sum_{x \in \text{set } (\text{cols } A)}. a x * x \$ i) =$   
   $(\sum_{j \in \{..<\text{length } (\text{cols } A)\}}. a (\text{cols } A ! j) * (\text{cols } A ! j) \$ i)$   
  **using** *bij-betw-imp-surj-on bij-betw-imp-inj-on* **by** (metis (*no-types, lifting*)  
*sum.reindex-cong*)  
  **also have**  $\dots = (\sum_{j \in \{..<\text{length } (\text{cols } A)\}}. a (\text{col } A j) * (\text{cols } A ! j) \$ i)$   
  **using** *assms(1) assms(2) find-first-unique[OF*  $\langle \text{distinct } (\text{cols } A) \rangle$   $\langle i < n \rangle$  **by**  
*auto*  
  **also have**  $\dots = (\sum_{j \in \{..<\text{length } (\text{cols } A)\}}. (\text{cols } A ! j) \$ i * a (\text{col } A j))$  **by**  
*(metis mult-commute-abs)*  
  **also have**  $\dots = (\sum_{j \in \{..<\text{length } (\text{cols } A)\}}. \text{row } A i \$ j * a (\text{col } A j))$  **using**  $\langle i$   
 $< n \rangle$  *assms(1) assms(2)* **by** *auto*  
  **finally show**  $(A \otimes_{mv} (\text{vec } nc (\lambda i. a (\text{col } A i)))) \$ i = \text{lincomb } a (\text{set } (\text{cols } A))$   
 $\$ i$   
  **unfolding** *lincomb-index* $[OF \langle i < n \rangle \langle \text{set } (\text{cols } A) \subseteq \text{carrier}_v n \rangle]$   
  **unfolding** *mat-mult-vec-def scalar-prod-def*  
  **using**  $\langle i < n \rangle$  *assms(1) atLeast0LessThan lessThan-def mat-carrierD(1) vec-index-vec*  
*sum.cong* **by** *auto*  
**qed**

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

*assms(1) assms(3)] by auto*  
**qed**

**lemma** (in *vec-space*) *lin-depI*:  
**assumes**  $A \in \text{carrier}_m n nc$   
**assumes**  $v \in \text{carrier}_v nc v \neq \mathbf{0}_v nc A \otimes_{mv} v = \mathbf{0}_v n$   
**assumes** *distinct (cols A)*  
**shows** *lin-dep (set (cols A))*  
**proof** –  
  **have** 1: *finite (set (cols A))* **by** *simp*  
  **have** 2:  $\text{set (cols A)} \subseteq \text{set (cols A)}$  **by** *auto*  
  **have** 3:  $(\lambda a. v \$ \text{find-first } a \text{ (cols A)}) \in \text{set (cols A)} \rightarrow \text{UNIV}$  **by** *simp*  
  **obtain**  $i$  **where**  $v \$ i \neq 0 i < nc$   
  **using**  $\langle v \neq \mathbf{0}_v nc \rangle$   
  **by** (*metis assms(2) vec-dim-vec vec-elemsD vec-eq-iff vec-zero-def vec-zero-index(1)*)  
  **then have**  $i < \text{dim}_c A$  **using** *assms(1)* **by** *blast*  
  **have** 4:  $\text{col } A \ i \in \text{set (cols A)}$   
  **using** *cols-nth[OF <i < dim\_c A>] <i < dim\_c A> in-set-conv-nth* **by** *fastforce*  
  **have** 5:  $v \$ \text{find-first (col } A \ i) \text{ (cols A)} \neq 0$   
  **using** *find-first-unique[OF <distinct (cols A)>] cols-nth[OF <i < dim\_c A>] <i <*  
*nc> <v \\$ i ≠ 0>*  
  **assms(1)** **by** *auto*  
  **have** 6: *lincomb*  $(\lambda a. v \$ \text{find-first } a \text{ (cols A)}) \text{ (set (cols A))} = \mathbf{0}_v n$   
  **using** *assms(1) assms(2) assms(4) assms(5) lincomb-eq-mat-mult* **by** *auto*  
  **show** *?thesis* **using** *lin-dep-crit[OF 1 2 3 4 5 6]* **by** *metis*  
**qed**

**lemma** (in *vec-space*) *lin-depE*:  
**assumes**  $A \in \text{carrier}_m n nc$   
**assumes** *lin-dep (set (cols A))*  
**assumes** *distinct (cols A)*  
**obtains**  $v$  **where**  $v \in \text{carrier}_v nc v \neq \mathbf{0}_v nc A \otimes_{mv} v = \mathbf{0}_v n$   
**proof** –  
  **have** *finite (set (cols A))* **by** *simp*  
  **obtain**  $a w$  **where**  $a \in \text{set (cols A)} \rightarrow \text{UNIV}$  *lincomb a (set (cols A)) = 0\_v n w*  
 $\in \text{set (cols A)}$   $a w \neq 0$   
  **using** *finite-lin-dep[OF <finite (set (cols A))> <lin-dep (set (cols A))>]*  
  **using** *assms(1) cols-dim mat-carrierD(1)* **by** *blast*  
  **def**  $v == \text{vec } nc \ (\lambda i. a \text{ (col } A \ i))$   
  **have** 1:  $v \in \text{carrier}_v nc$  **by** (*simp add: v-def*)  
  **have** 2:  $v \neq \mathbf{0}_v nc$   
  **proof** –  
  **obtain**  $i$  **where**  $w = \text{col } A \ i \ i < \text{length (cols A)}$   
  **by** (*metis <w ∈ set (cols A)> cols-length cols-nth in-set-conv-nth*)  
  **have**  $v \$ i \neq 0$   
  **unfolding** *v-def*  
  **using**  $\langle a w \neq 0 \rangle$  [*unfolded <w = col A i> vec-index-vec[OF <i < length (cols A)>]*]  
  *assms(1) cols-length mat-carrierD(2)* **by** (*metis (no-types) <A ∈ carrier\_m n*

```

nc)
  ⟨ $\bigwedge f. \text{vec}(\text{length}(\text{cols } A)) f \$ i = f i \rangle \langle a(\text{col } A \ i) \neq 0 \rangle \text{cols-length mat-carrierD}(2)$ 
  then show ?thesis using ⟨ $i < \text{length}(\text{cols } A)$ ⟩ assms(1) by auto
qed
have  $\exists A \otimes_{mv} v = \mathbf{0}_v \ n$  unfolding v-def
  using ⟨lincomb a (set (cols A)) =  $\mathbf{0}_v \ n$  mat-mult-eq-lincomb[OF  $\langle A \in \text{carrier}_m$ 
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  $A \in \text{carrier}_m \ n \ n$ 
and  $\neg \text{distinct}(\text{cols } A)$ 
shows  $\text{rank } A < n$ 
proof -
  obtain S where maximal S ( $\lambda T. T \subseteq \text{set}(\text{cols } A) \wedge \text{lin-indpt } T$ )
  using maximal-exists[of ( $\lambda T. T \subseteq \text{set}(\text{cols } A) \wedge \text{lin-indpt } T$ ) card (set (cols
A)) {}]
  by (meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2
rev-finite-subset)
  then have  $\text{card } S \leq \text{card}(\text{set}(\text{cols } A))$  by (simp add: card-mono maximal-def)
  then have  $\text{card } S < n$ 
  using assms(1) cols-length card-length ( $\neg \text{distinct}(\text{cols } A)$ ) card-distinct mat-carrierD(2)
nat-less-le
  by (metis dual-order.antisym dual-order.trans)
  then show ?thesis
  using rank-card-indpt[OF ( $A \in \text{carrier}_m \ n \ n$ ) maximal S ( $\lambda T. T \subseteq \text{set}(\text{cols}$ 
A) \wedge \text{lin-indpt } T)⟩]
  by simp
qed

```

The theorem "det non-zero  $\longleftrightarrow$  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  $A \in \text{carrier}_m \ n \ n$ 
and  $\text{det } A = 0$ 
shows  $\text{rank } A < n$ 
proof (rule ccontr)
  assume  $\neg \text{rank } A < n$ 
  then have  $\text{rank } A = n$  using rank-le-nc assms le-neq-implies-less by blast
  obtain v where  $v \in \text{carrier}_v \ n \ v \neq \mathbf{0}_v \ n \ A \otimes_{mv} v = \mathbf{0}_v \ n$ 
  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) ⟨ $v \in \text{carrier}_v \ n \ v \neq \mathbf{0}_v \ n \ \langle A$ 
 $\otimes_{mv} v = \mathbf{0}_v \ n \rangle$ ⟩] True by blast

```

```

next
  case False
  then show False using non-distinct-low-rank  $\langle \text{rank } A = n \rangle \langle \neg \text{rank } A < n \rangle$ 
assms(1) by blast
qed
qed

```

```

lemma det-identical-cols:
  assumes  $A \in \text{carrier}_m \ n \ n$ 
  and  $ij: i \neq j$ 
  and  $i: i < n$  and  $j: j < n$ 
  and  $r: \text{col } A \ i = \text{col } A \ j$ 
  shows  $\det A = 0$ 
  using det-identical-rows det-transpose
  by (metis  $A \ i \ j \ \text{mat-carrierD}(2) \ \text{mat-transpose-closed} \ r \ \text{row-transpose}$ )

```

```

lemma (in vec-space) low-rank-det-zero:
  assumes  $A \in \text{carrier}_m \ n \ n$ 
  and  $\det A \neq 0$ 
  shows  $\text{rank } A = n$ 
  proof -
    have distinct (cols  $A$ )
    proof (rule ccontr)
      assume  $\neg \text{distinct} \ (\text{cols } A)$ 
      then obtain  $i \ j$  where  $i \neq j \ (\text{cols } A) \ ! \ i = (\text{cols } A) \ ! \ j \ i < \text{length} \ (\text{cols } A) \ j < \text{length} \ (\text{cols } A)$ 
      using distinct-conv-nth by blast
      then have  $\text{col } A \ i = \text{col } A \ j \ i < n \ j < n$  using assms(1) by auto
      then have  $\det A = 0$  using det-identical-cols using  $\langle i \neq j \rangle$  assms(1) by blast
      then show False using  $\langle \det A \neq 0 \rangle$  by auto
    qed
    have  $\bigwedge v. v \in \text{carrier}_v \ n \implies v \neq \mathbf{0}_v \ n \implies A \otimes_{mv} v \neq \mathbf{0}_v \ n$ 
      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)] -  $\langle \text{distinct} \ (\text{cols } A) \rangle$  by auto
    then show ?thesis using lin-indpt-full-rank[OF assms(1)]  $\langle \text{distinct} \ (\text{cols } A) \rangle$  by metis
  qed

```

```

lemma (in vec-space) det-rank-iff:
  assumes  $A \in \text{carrier}_m \ n \ n$ 
  shows  $\det A \neq 0 \iff \text{rank } A = n$ 
  using assms det-zero-low-rank low-rank-det-zero by force

```

## 12 Subadditivity of rank

Subadditivity is the property of rank, that  $\text{rank}(A + B) \leq \text{rank } A + \text{rank } B$ .

**lemma** (in module) *lincomb-add*:

**assumes** *finite* ( $b1 \cup b2$ )

**assumes**  $b1 \cup b2 \subseteq \text{carrier } M$

**assumes**  $x1 = \text{lincomb } a1 \ b1 \ a1 \in (b1 \rightarrow \text{carrier } R)$

**assumes**  $x2 = \text{lincomb } a2 \ b2 \ a2 \in (b2 \rightarrow \text{carrier } R)$

**assumes**  $x = x1 \oplus_M x2$

**shows**  $\text{lincomb } (\lambda v. (\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0})) \ v \oplus (\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \ v) \ (b1 \cup b2) = x$

**proof** –

**have** *finite* ( $b1 \cup (b2 - b1)$ ) *finite* ( $b2 \cup (b1 - b2)$ )

$b1 \cup (b2 - b1) \subseteq \text{carrier } M \ b2 \cup (b1 - b2) \subseteq \text{carrier } M$

$b1 \cap (b2 - b1) = \{\} \ b2 \cap (b1 - b2) = \{\}$

$(\lambda b. \mathbf{0}_R) \in b2 - b1 \rightarrow \text{carrier } R \ (\lambda b. \mathbf{0}_R) \in b1 - b2 \rightarrow \text{carrier } R$

**using** *finite* ( $b1 \cup b2$ )  $\langle b1 \cup b2 \subseteq \text{carrier } M \rangle \langle a2 \in (b2 \rightarrow \text{carrier } R) \rangle$  **by** *auto*

**have**  $\text{lincomb } (\lambda b. \mathbf{0}_R) \ (b2 - b1) = \mathbf{0}_M \ \text{lincomb } (\lambda b. \mathbf{0}_R) \ (b1 - b2) = \mathbf{0}_M$

**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**  $x1 = \text{lincomb } (\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0}) \ (b1 \cup b2)$

$x2 = \text{lincomb } (\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \ (b1 \cup b2)$

**using** *lincomb-union2*[*OF* *finite* ( $b1 \cup (b2 - b1)$ )  $\langle b1 \cup (b2 - b1) \subseteq \text{carrier } M \rangle \langle b1 \cap (b2 - b1) = \{\} \rangle \langle a1 \in (b1 \rightarrow \text{carrier } R) \rangle \langle (\lambda b. \mathbf{0}_R) \in b2 - b1 \rightarrow \text{carrier } R \rangle$ ]

*lincomb-union2*[*OF* *finite* ( $b2 \cup (b1 - b2)$ )  $\langle b2 \cup (b1 - b2) \subseteq \text{carrier } M \rangle \langle b2 \cap (b1 - b2) = \{\} \rangle \langle a2 \in (b2 \rightarrow \text{carrier } R) \rangle \langle (\lambda b. \mathbf{0}_R) \in b1 - b2 \rightarrow \text{carrier } R \rangle$ ]

**using** *assms(2)* *assms(3)* *assms(4)* *assms(5)* *assms(6)* **by** (*simp-all* *add:Un-commute*)

**have**  $(\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0}) \in (b1 \cup b2) \rightarrow \text{carrier } R$

$(\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \in (b1 \cup b2) \rightarrow \text{carrier } R$  **using** *assms(4)*

*assms(6)* **by** *auto*

**show**  $\text{lincomb } (\lambda v. (\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0})) \ v \oplus (\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \ v) \ (b1 \cup b2) = x$

**using** *lincomb-sum*[*OF* *finite* ( $b1 \cup b2$ )  $\langle b1 \cup b2 \subseteq \text{carrier } M \rangle$

$\langle (\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0}) \in (b1 \cup b2) \rightarrow \text{carrier } R \rangle \langle (\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \in (b1 \cup b2) \rightarrow \text{carrier } R \rangle$ ]

$\langle x1 = \text{lincomb } (\lambda v. \text{if } v \in b1 \text{ then } a1 \ v \ \text{else } \mathbf{0}) \ (b1 \cup b2) \rangle \langle x2 = \text{lincomb } (\lambda v. \text{if } v \in b2 \text{ then } a2 \ v \ \text{else } \mathbf{0}) \ (b1 \cup b2) \rangle$  *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**  $\text{vectorspace.dim } K \ (vs \ (\text{subspace-sum } W1 \ W2)) \leq \text{vectorspace.dim } K \ (vs \ W1) + \text{vectorspace.dim } K \ (vs \ W2)$

**proof** –

**have** *vectorspace*  $K \ (vs \ W1)$  *vectorspace*  $K \ (vs \ W2)$  *submodule*  $K \ W1 \ V$  *submodule*  $K \ W2 \ V$

**by** (*simp* *add:*  $\langle \text{subspace } K \ W1 \ V \rangle \langle \text{subspace } K \ W2 \ V \rangle$  *subspace-is-vs*)**+**

```

obtain  $b1\ b2$  where  $\text{vectorspace.basis } K\ (vs\ W1)\ b1\ \text{vectorspace.basis } K\ (vs\ W2)\ b2\ \text{finite } b1\ \text{finite } b2$ 
using  $\text{vectorspace.finite-basis-exists}[OF\ \langle\text{vectorspace } K\ (vs\ W1)\ \rangle\ \langle\text{vectorspace.fin-dim } K\ (vs\ W1)\ \rangle]$ 
using  $\text{vectorspace.finite-basis-exists}[OF\ \langle\text{vectorspace } K\ (vs\ W2)\ \rangle\ \langle\text{vectorspace.fin-dim } K\ (vs\ W2)\ \rangle]$ 
by  $\text{blast}$ 
then have  $\text{LinearCombinations.module.gen-set } K\ (vs\ W1)\ b1\ \text{LinearCombinations.module.gen-set } K\ (vs\ W2)\ b2$ 
using  $\langle\text{vectorspace } K\ (vs\ W1)\ \rangle\ \langle\text{vectorspace } K\ (vs\ W2)\ \rangle\ \text{vectorspace.basis-def}$ 
by  $\text{blast+}$ 
then have  $\text{span } b1 = W1\ \text{span } b2 = W2$ 
using  $\text{module.span-li-not-depend}(1)\ \langle\text{submodule } K\ W1\ V\ \rangle\ \langle\text{submodule } K\ W2\ V\ \rangle$ 
 $\langle\text{vectorspace } K\ (vs\ W1)\ \rangle\ \langle\text{vectorspace.basis } K\ (vs\ W1)\ b1\ \rangle\ \langle\text{vectorspace } K\ (vs\ W2)\ \rangle$ 
 $\langle\text{vectorspace.basis } K\ (vs\ W2)\ b2\ \rangle\ \text{vectorspace.basis-def}$  by  $\text{force+}$ 
have  $W1 \subseteq \text{carrier } V\ W2 \subseteq \text{carrier } V$  using  $\langle\text{subspace } K\ W1\ V\ \rangle\ \langle\text{subspace } K\ W2\ V\ \rangle\ \text{subspace-def}\ \text{submodule-def}$  by  $\text{metis+}$ 
have  $b1 \subseteq \text{carrier } V$ 
using  $\langle\text{vectorspace.basis } K\ (vs\ W1)\ b1\ \rangle\ \langle\text{vectorspace } K\ (vs\ W1)\ \rangle\ \text{vectorspace.basis-def}$ 

 $\langle W1 \subseteq \text{carrier } V \rangle$  by  $\text{fastforce}$ 
have  $b2 \subseteq \text{carrier } V$ 
using  $\langle\text{vectorspace.basis } K\ (vs\ W2)\ b2\ \rangle\ \langle\text{vectorspace } K\ (vs\ W2)\ \rangle\ \text{vectorspace.basis-def}$ 

 $\langle W2 \subseteq \text{carrier } V \rangle$  by  $\text{fastforce}$ 
have  $\text{finite } (b1 \cup b2)\ b1 \cup b2 \subseteq \text{carrier } V$ 
by  $(\text{simp-all add: } \langle\text{finite } b1\ \rangle\ \langle\text{finite } b2\ \rangle\ \langle b2 \subseteq \text{carrier } V \rangle\ \langle b1 \subseteq \text{carrier } V \rangle)$ 
have  $\text{subspace-sum } W1\ W2 \subseteq \text{span } (b1 \cup b2)$ 
proof  $(\text{rule subsetI})$ 
fix  $x$  assume  $x \in \text{subspace-sum } W1\ W2$ 
obtain  $x1\ x2$  where  $x1 \in W1\ x2 \in W2\ x = x1 \oplus_V x2$ 
using  $\text{imageE}[OF\ \langle x \in \text{subspace-sum } W1\ W2 \rangle[\text{unfolded submodule-sum-def}]]$ 
by  $(\text{metis } (\text{no-types, lifting})\ \text{BNF-Def.Collect-case-prodD}\ \text{split-def})$ 
obtain  $a1$  where  $x1 = \text{lincomb } a1\ b1\ a1 \in (b1 \rightarrow \text{carrier } K)$ 
using  $\langle\text{span } b1 = W1\ \rangle\ \text{finite-span}[OF\ \langle\text{finite } b1\ \rangle\ \langle b1 \subseteq \text{carrier } V \rangle]\ \langle x1 \in W1 \rangle$ 
by  $\text{auto}$ 
obtain  $a2$  where  $x2 = \text{lincomb } a2\ b2\ a2 \in (b2 \rightarrow \text{carrier } K)$ 
using  $\langle\text{span } b2 = W2\ \rangle\ \text{finite-span}[OF\ \langle\text{finite } b2\ \rangle\ \langle b2 \subseteq \text{carrier } V \rangle]\ \langle x2 \in W2 \rangle$ 
by  $\text{auto}$ 
obtain  $a$  where  $x = \text{lincomb } a\ (b1 \cup b2)$  using  $\text{lincomb-add}[OF\ \langle\text{finite } (b1 \cup b2)\ \rangle\ \langle b1 \cup b2 \subseteq \text{carrier } V \rangle$ 
 $\langle x1 = \text{lincomb } a1\ b1 \rangle\ \langle a1 \in (b1 \rightarrow \text{carrier } K) \rangle\ \langle x2 = \text{lincomb } a2\ b2 \rangle\ \langle a2 \in (b2 \rightarrow \text{carrier } K) \rangle\ \langle x = x1 \oplus_V x2 \rangle]$  by  $\text{blast}$ 
then show  $x \in \text{span } (b1 \cup b2)$  using  $\text{finite-span}[OF\ \langle\text{finite } (b1 \cup b2)\ \rangle\ \langle (b1 \cup b2) \subseteq \text{carrier } V \rangle]$ 
using  $\langle b1 \subseteq \text{carrier } V \rangle\ \langle b2 \subseteq \text{carrier } V \rangle\ \langle\text{span } b1 = W1\ \rangle\ \langle\text{span } b2 = W2\ \rangle\ \langle x \in \text{subspace-sum } W1\ W2 \rangle\ \text{span-union-is-sum}$  by  $\text{auto}$ 

```

```

qed
have  $b1 \subseteq W1$   $b2 \subseteq W2$ 
  using  $\langle$ vector space  $K$  (vs  $W1$ ) $\rangle$   $\langle$ vector space  $K$  (vs  $W2$ ) $\rangle$   $\langle$ vector space.basis  $K$ 
(vs  $W1$ )  $b1$  $\rangle$ 
   $\langle$ vector space.basis  $K$  (vs  $W2$ )  $b2$  $\rangle$  vector space.basis-def local.carrier-vs-is-self by
blast+
then have  $b1 \cup b2 \subseteq$  subspace-sum  $W1$   $W2$  using  $\langle$ submodule  $K$   $W1$   $V$  $\rangle$   $\langle$ submodule
 $K$   $W2$   $V$  $\rangle$  in-sum
  by (metis  $assms(1)$   $assms(3)$  dual-order.trans sup-least vector space.vsum-comm
vector space-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:  $\langle$ submodule  $K$   $W1$ 
 $V$  $\rangle$   $\langle$ submodule  $K$   $W2$   $V$  $\rangle$  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  $\langle$  $b1 \cup b2 \subseteq$  subspace-sum  $W1$   $W2$  $\rangle$ 
 $\langle$ submodule  $K$  (subspace-sum  $W1$   $W2$ )  $V$  $\rangle$ ]
    by (simp add:  $\langle$ subspace-sum  $W1$   $W2$   $\subseteq$  span ( $b1 \cup b2$ ) $\rangle$ )
  show subspace-sum  $W1$   $W2$   $\supseteq$  LinearCombinations.module.span  $K$  (vs (subspace-sum
 $W1$   $W2$ )) ( $b1 \cup b2$ )
    using  $\langle$  $b1 \cup b2 \subseteq$  subspace-sum  $W1$   $W2$  $\rangle$  by (metis (full-types) LinearCombi-
nations.module.span-is-subset2
LinearCombinations.module.submodule-is-module  $\langle$ submodule  $K$  (subspace-sum
 $W1$   $W2$ )  $V$  $\rangle$  local.carrier-vs-is-self submodule-def)
qed
have vector space  $K$  (vs (subspace-sum  $W1$   $W2$ )) using  $assms(1)$   $assms(3)$ 
subspace-def sum-is-subspace vector space.subspace-is-vs by blast
then have vector space.dim  $K$  (vs (subspace-sum  $W1$   $W2$ ))  $\leq$  card ( $b1 \cup b2$ )
  using vector space.gen-ge-dim[OF  $\langle$ vector space  $K$  (vs (subspace-sum  $W1$   $W2$ )) $\rangle$ 
 $\langle$ finite ( $b1 \cup b2$ ) $\rangle$ ]
   $\langle$  $b1 \cup b2 \subseteq$  subspace-sum  $W1$   $W2$  $\rangle$ 
   $\langle$ subspace-sum  $W1$   $W2$  = LinearCombinations.module.span  $K$  (vs (subspace-sum
 $W1$   $W2$ )) ( $b1 \cup b2$ ) $\rangle$ 
  local.carrier-vs-is-self by blast
also have ...  $\leq$  card  $b1$  + card  $b2$  by (simp add: card-Un-le)
also have ... = vector space.dim  $K$  (vs  $W1$ ) + vector space.dim  $K$  (vs  $W2$ )
  by (metis  $\langle$ finite  $b1$  $\rangle$   $\langle$ finite  $b2$  $\rangle$   $\langle$ vector space  $K$  (vs  $W1$ ) $\rangle$   $\langle$ vector space  $K$  (vs
 $W2$ ) $\rangle$ 
   $\langle$ vector space.basis  $K$  (vs  $W1$ )  $b1$  $\rangle$   $\langle$ vector space.basis  $K$  (vs  $W2$ )  $b2$  $\rangle$  vector space.dim-basis)
finally show ?thesis by auto
qed

```

```

lemma (in module) nested-submodules:
assumes submodule  $R$   $W$   $M$ 
assumes submodule  $R$   $X$   $M$ 
assumes  $X \subseteq W$ 
shows submodule  $R$   $X$  (md  $W$ )

```

**unfolding** *submodule-def*  
**using**  $\langle X \subseteq W \rangle$  *submodule-is-module*[*OF*  $\langle \text{submodule } R \ W \ M \rangle$ ] **using**  $\langle \text{submodule } R \ X \ M \rangle$ [*unfolded submodule-def*] **by** *auto*

**lemma** (**in** *vectorspace*) *nested-subspaces*:  
**assumes** *subspace*  $K \ W \ V$   
**assumes** *subspace*  $K \ X \ V$   
**assumes**  $X \subseteq W$   
**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*  $K \ (vs \ X) \leq dim$   
**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**  $b \subseteq carrier \ V$  *LinearCombinations.module.lin-indpt*  $K \ (vs \ X) \ b$   
**using** *vectorspace.basis-def*[*OF*  $\langle \text{vectorspace } K \ (vs \ X) \rangle$ ]  $\langle \text{subspace } K \ X \ V \rangle$ [*unfolded subspace-def submodule-def*] **by** *auto*  
**then have** *lin-indpt*  $b$   
**by** (*metis* *LinearCombinations.module.span-li-not-depend(2)*  $\langle \text{vectorspace } K \ (vs \ X) \rangle$   $\langle \text{vectorspace.basis } K \ (vs \ X) \ b \rangle$   
*assms(1) is-module local.carrier-vs-is-self submodule-def vectorspace.basis-def*)  
**show** *?thesis* **using** *li-le-dim(2)*[*OF*  $\langle \text{fin-dim} \rangle$   $\langle b \subseteq carrier \ V \rangle$   $\langle \text{lin-indpt } b \rangle$ ]  
**using**  $\langle b \subseteq carrier \ V \rangle$   $\langle \text{lin-indpt } b \rangle$   $\langle \text{vectorspace } K \ (vs \ X) \rangle$   $\langle \text{vectorspace.basis } K \ (vs \ X) \ b \rangle$  *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*  $K \ (vs \ (\text{subspace-sum } W1 \ W2))$   
**proof** –  
**obtain**  $b1$  **where** *finite*  $b1 \ b1 \subseteq W1$  *LinearCombinations.module.gen-set*  $K \ (vs \ W1) \ b1$   
**using** *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*  
**obtain**  $b2$  **where** *finite*  $b2 \ b2 \subseteq W2$  *LinearCombinations.module.gen-set*  $K \ (vs \ W2) \ b2$   
**using** *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*  
**have**  $1: \text{finite } (b1 \cup b2)$  **by** (*simp add:*  $\langle \text{finite } b1 \rangle$   $\langle \text{finite } b2 \rangle$ )  
**have**  $2: b1 \cup b2 \subseteq \text{subspace-sum } W1 \ W2$   
**by** (*metis* (*no-types, lifting*)  $\langle b1 \subseteq W1 \rangle$   $\langle b2 \subseteq W2 \rangle$  *assms(1) assms(2)*  
*le-sup-iff subset-Un-eq vectorspace.in-sum-vs vectorspace.vsum-comm vectorspace-axioms*)  
**have**  $3: \text{LinearCombinations.module.gen-set } K \ (vs \ (\text{subspace-sum } W1 \ W2)) \ (b1$

```

∪ 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 ∈ span (b1 ∪ b2) unfolding ⟨x = x1 ⊕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**  $A \in \text{carrier}_m \ n \ nc$

**assumes**  $B \in \text{carrier}_m \ n \ nc$

**shows**  $\text{rank} (A \oplus_m B) \leq \text{rank} A + \text{rank} B$

**proof** –

**def** W1 == span (set (cols A))

**def** W2 == span (set (cols B))

**have** set (cols (A ⊕<sub>m</sub> B)) ⊆ subspace-sum W1 W2

**proof**

**fix** x **assume** x ∈ set (cols (A ⊕<sub>m</sub> B))

**obtain** i **where** x = col (A ⊕<sub>m</sub> B) i i < length (cols (A ⊕<sub>m</sub> B))

**using**  $\langle x \in \text{set} (\text{cols} (A \oplus_m B)) \rangle$  *nth-find-first cols-nth find-first-le* **by** (*metis cols-length*)  
**then have**  $x = \text{col } A \ i \oplus_v \text{col } B \ i$  **using**  $\langle i < \text{length} (\text{cols} (A \oplus_m B)) \rangle$  *assms(1) assms(2)* **by auto**  
**have**  $\text{col } A \ i \in \text{span} (\text{set} (\text{cols } A))$   $\text{col } B \ i \in \text{span} (\text{set} (\text{cols } B))$   
**using**  $\langle i < \text{length} (\text{cols} (A \oplus_m B)) \rangle$  *assms(1) assms(2) in-set-conv-nth*  
**by** (*metis cols-dim cols-length cols-nth mat-carrierD(1) mat-carrierD(2) mat-index-add(3) span-mem*)  
**then show**  $x \in \text{subspace-sum } W1 \ W2$   
**unfolding** *W1-def W2-def*  $\langle x = \text{col } A \ i \oplus_v \text{col } B \ i \rangle$  *submodule-sum-def* **by blast**  
**qed**  
**have** *subspace F (subspace-sum W1 W2) V*  
**by** (*metis W1-def W2-def assms(1) assms(2) cols-dim mat-carrierD(1) span-is-submodule subspace-def sum-is-submodule vec-vs*)  
**then have**  $\text{span} (\text{set} (\text{cols} (A \oplus_m B))) \subseteq \text{subspace-sum } W1 \ W2$   
**by** (*simp add:  $\langle \text{set} (\text{cols} (A \oplus_m B)) \subseteq \text{subspace-sum } W1 \ W2 \rangle$  span-is-subset*)  
**have** *subspace F (span (set (cols (A ⊕<sub>m</sub> B)))) V* **by** (*metis assms(2) cols-dim mat-add-closed mat-carrierD(1) span-is-subspace*)  
**have** *subspace:subspace F (span (set (cols (A ⊕<sub>m</sub> B)))) (vs (subspace-sum W1 W2))*  
**using** *nested-subspaces[OF  $\langle \text{subspace } F \text{ (subspace-sum } W1 \ W2) \ V \rangle$   $\langle \text{subspace } F \text{ (span (set (cols (A ⊕<sub>m</sub> B)))) } \ V \rangle$   $\langle \text{span (set (cols (A ⊕<sub>m</sub> B))) } \subseteq \text{subspace-sum } W1 \ W2 \rangle$ ]*.  
**have** *vectorspace.fin-dim F (vs W1) vectorspace.fin-dim F (vs W2) subspace F W1 V subspace F W2 V*  
**using** *span-is-subspace W1-def W2-def assms(1) assms(2) cols-dim mat-carrierD fin-dim-span-cols* **by auto**  
**then have** *fin-dim: vectorspace.fin-dim F (vs (subspace-sum W1 W2))* **using** *fin-dim-subspace-sum* **by auto**  
**have** *vectorspace.fin-dim F (span-vs (set (cols (A ⊕<sub>m</sub> B))))* **using** *assms(2) mat-add-closed vec-space.fin-dim-span-cols* **by blast**  
**then have**  $\text{rank} (A \oplus_m B) \leq \text{vectorspace.dim } F \text{ (vs (subspace-sum } W1 \ W2))$   
**unfolding** *rank-def*  
**using** *vectorspace.subspace-dim[OF subspace-is-vs[OF  $\langle \text{subspace } F \text{ (subspace-sum } W1 \ W2) \ V \rangle$   $\langle \text{subspace } F \text{ (span (set (cols (A ⊕<sub>m</sub> B)))) } \ V \rangle$   $\langle \text{span (set (cols (A ⊕<sub>m</sub> B))) } \subseteq \text{subspace-sum } W1 \ W2 \rangle$ ]* *subspace fin-dim* **by auto**  
**also have**  $\text{vectorspace.dim } F \text{ (vs (subspace-sum } W1 \ W2))} \leq \text{rank } A + \text{rank } B$   
**unfolding** *rank-def*  
**using** *W1-def W2-def  $\langle \text{subspace } F \ W1 \ V \rangle$   $\langle \text{subspace } F \ W2 \ V \rangle$   $\langle \text{vectorspace.fin-dim } F \text{ (vs } W1) \rangle$   $\langle \text{vectorspace.fin-dim } F \text{ (vs } W2) \rangle$  *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*:  $\text{vectorspace.dim } F (\text{span-vs } \{\}) = 0$   
**proof** –  
**have** *vectorspace*  $F (\text{span-vs } \{\})$  **using** *field.field-axioms span-is-submodule submodule-is-module vectorspace-def* **by** *auto*  
**have**  $\{\} \subseteq \text{carrier}_v n \wedge \text{lin-indpt } \{\}$   
**by** (*metis (no-types) empty-subsetI fin-dim finite-basis-exists subset-li-is-li vec-vs vectorspace.basis-def*)  
**then have** *vectorspace.basis*  $F (\text{span-vs } \{\}) \{\}$  **using** *vectorspace.basis-def*  
**by** (*simp add: (vectorspace F (vs (span { }))) span-is-submodule span-li-not-depend(1) span-li-not-depend(2) vectorspace.basis-def*)  
**then show** *?thesis* **using**  $(\text{vectorspace } F (\text{vs } (\text{span } \{\})))$  *vectorspace.dim-basis*  
**by** *fastforce*  
**qed**

**lemma** (in *vec-space*) *rank-0I*:  $\text{rank } (\mathbf{0}_m \ n \ nc) = 0$   
**proof** –  
**have**  $\text{set } (\text{cols } (\mathbf{0}_m \ n \ nc)) \subseteq \{\mathbf{0}_v \ n\}$   
**by** (*metis col-zero cols-length cols-nth in-set-conv-nth insertCI mat-index-zero(3) subsetI*)  
**have**  $\text{set } (\text{cols } (\mathbf{0}_m \ n \ nc::'a \ \text{mat})) = \{\} \vee \text{set } (\text{cols } (\mathbf{0}_m \ n \ nc)) = \{\mathbf{0}_v \ n::'a \ \text{vec}\}$   
**by** (*meson (set (cols (0\_m n nc)) ⊆ {0\_v n}) subset-singletonD*)  
**then have**  $\text{span } (\text{set } (\text{cols } (\mathbf{0}_m \ n \ nc))) = \{\mathbf{0}_v \ n\}$   
**by** (*metis (no-types) span-empty span-zero vectorspace.span-empty vectorspace-axioms*)  
**then show** *?thesis* **unfolding** *rank-def*  $(\text{span } (\text{set } (\text{cols } (\mathbf{0}_m \ n \ nc)))) = \{\mathbf{0}_v \ n\}$   
**using** *span-empty dim-zero-vs* **by** *simp*  
**qed**

**lemma** (in *vec-space*) *rank-le-1-product-entries*:  
**fixes**  $f \ g::\text{nat} \Rightarrow 'a$   
**assumes**  $A \in \text{carrier}_m \ n \ nc$   
**assumes**  $\bigwedge r \ c. r < \text{dim}_r \ A \implies c < \text{dim}_c \ A \implies A \ \$\$ (r,c) = f \ r * g \ c$   
**shows**  $\text{rank } A \leq 1$   
**proof** –  
**have**  $\text{set } (\text{cols } A) \subseteq \text{span } \{\text{vec } n \ f\}$   
**proof**  
**fix**  $v$  **assume**  $v \in \text{set } (\text{cols } A)$   
**then obtain**  $c$  **where**  $c < \text{dim}_c \ A \ v = \text{col } A \ c$  **by** (*metis cols-length cols-nth in-set-conv-nth*)  
**have**  $g \ c \odot_v \ \text{vec } n \ f = v$   
**proof** (*rule vec-eqI*)  
**show**  $\text{dim}_v (g \ c \odot_v \ \text{Matrix.vec } n \ f) = \text{dim}_v \ v$  **using**  $(v = \text{col } A \ c)$  *assms(1)*  
**by** *auto*  
**fix**  $r$  **assume**  $r < \text{dim}_v \ v$   
**then have**  $r < \text{dim}_v (\text{Matrix.vec } n \ f)$  **using**  $(\text{dim}_v (g \ c \odot_v \ \text{Matrix.vec } n \ f) = \text{dim}_v \ v)$  **by** *auto*  
**then have**  $r < n \ r < \text{dim}_r \ A$  **using** *vec-index-scalar-mult(2)*  $(A \in \text{carrier}_m \ n \ nc)$  **by** *auto*

```

    show (g c ⊙v Matrix.vec n f) $ r = v $ r
      unfolding ⟨v = col A c⟩ col-def vec-index-scalar-mult(1)[OF ⟨r < dimv
(Matrix.vec n f)⟩]
        vec-index-vec[OF ⟨r < n⟩] vec-index-vec[OF ⟨r < dimr A⟩] by (simp add:
⟨c < dimc A⟩ ⟨r < dimr A⟩ assms(2))
      qed
    then show v ∈ span {vec n f} using submodule.smult-closed[OF span-is-submodule]
cF
      using UNIV-I empty-subsetI insert-subset span-self vec-dim vec-module-simps(4)
    by auto
      qed
    have vectorspace F (vs (span {Matrix.vec n f})) using span-is-subspace[THEN
subspace-is-vs, of {vec n f}] by auto
    have submodule F (span {Matrix.vec n f}) V by (simp add: span-is-submodule)
    have subspace F (span (set (cols A))) (vs (span {Matrix.vec n f}))
      using vectorspace.span-is-subspace[OF ⟨vectorspace F (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 F (span
{Matrix.vec n f}) V⟩]]
        ⟨set (cols A) ⊆ span {vec n f}⟩ by auto
    have fin-dim:vectorspace.fin-dim F (vs (span {Matrix.vec n f}))
      vectorspace.fin-dim F (vs (span {Matrix.vec n f})) (carrier := span (set (cols
A)))
      using fin-dim-span fin-dim-span-cols ⟨A ∈ carrierm n nc⟩ by auto
    have vectorspace.dim F (vs (span {Matrix.vec n f})) ≤ 1
      using vectorspace.dim-le1I[OF ⟨vectorspace F (vs (span {Matrix.vec n f}))⟩]
span-mem span-li-not-depend(1)[OF - ⟨submodule F (span {Matrix.vec n f})
V⟩] by simp
    then show ?thesis unfolding rank-def using vectorspace.subspace-dim[OF
⟨vectorspace F (vs (span {Matrix.vec n f}))⟩ ⟨subspace F (span (set (cols A)))
(vs (span {Matrix.vec n f}))⟩]
      fin-dim(1) fin-dim(2)] by simp
    qed
  end
end

```

## 13 Missing Lemmas of Sublist

```

theory DL-Missing-Sublist
imports Main
begin

```

**lemma** *sublist-only-one*:

**assumes**  $\{i. i < \text{length } xs \wedge i \in I\} = \{j\}$

**shows**  $\text{sublist } xs \ I = [xs!j]$

**proof** –

**have**  $\text{set } (\text{sublist } xs \ I) = \{xs!j\}$

**unfolding** *set-sublist* **using** *subset-antisym* *assms* **by** *fastforce*

**moreover have**  $\text{length } (\text{sublist } xs \ I) = 1$

**unfolding** *length-sublist 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** *sublist-replicate:*

*sublist (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]*

*sublist-append Suc sublist-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]*

*sublist-append Suc sublist-singleton 0*

**by** *simp*

**qed**

**qed**

**lemma** *length-sublist-even:*

**assumes** *even (length xs)*

**shows** *length (sublist xs (Collect even)) = length (sublist 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*:  $\text{length } (\text{sublist } (\text{drop } 2 \text{ } xs) (\text{Collect even})) = \text{length } (\text{sublist } (\text{drop } 2 \text{ } xs) \{a. \text{ odd } a\})$  **by** *simp*

**have**  $\text{length } (\text{take } 2 \text{ } xs) = 2$  **using** *Suc.hyps(2)* **by** *auto*

**then have** *plus-odd*:  $\{j. j + \text{length } (\text{take } 2 \text{ } xs) \in \text{Collect odd}\} = \text{Collect odd}$  **and**  
*plus-even*:  $\{j. j + \text{length } (\text{take } 2 \text{ } xs) \in \text{Collect even}\} = \text{Collect even}$  **by**  
*simp-all*

**have** *sublist-take2*:  $\text{sublist } (\text{take } 2 \text{ } xs) (\text{Collect even}) = [\text{take } 2 \text{ } xs ! 0]$  *sublist*  
 $(\text{take } 2 \text{ } xs) (\text{Collect odd}) = [\text{take } 2 \text{ } xs ! 1]$

**using**  $\langle \text{length } (\text{take } 2 \text{ } xs) = 2 \rangle$  *less-2-cases* *sublist-only-one*[*of take 2 xs Collect even 0*]  
*sublist-only-one*[*of take 2 xs Collect odd 1*]  
**by** *fastforce+*

**then have**  $\text{length } (\text{sublist } (\text{take } 2 \text{ } xs @ \text{drop } 2 \text{ } xs) (\text{Collect even}))$   
 $= \text{length } (\text{sublist } (\text{take } 2 \text{ } xs @ \text{drop } 2 \text{ } xs) \{a. \text{ odd } a\})$

**unfolding** *sublist-append* *length-append* *plus-odd* *plus-even* *sublist-take2* *length-drop2*  
**by** *auto*

**then show** *?case* **using** *append-take-drop-id*[*of 2 xs*] **by** *simp*

**qed**

**lemma** *sublist-map*:  
 $\text{sublist } (\text{map } f \text{ } xs) A = \text{map } f (\text{sublist } xs A)$

**proof** (*induction xs arbitrary:A*)

**case** *Nil*

**then show** *?case* **by** *simp*

**next**

**case** (*Cons x xs*)

**then show** *?case*

**by** (*simp add: sublist-Cons*)

**qed**

## 14 Pick

**fun** *pick* ::  $\text{nat set} \Rightarrow \text{nat} \Rightarrow \text{nat}$  **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**  $\text{pick } S \ n \in S$

**proof** (*cases n*)

**show**  $n = 0 \implies \text{pick } S \ n \in S$

**unfolding** *pick.simps* **using**  $\langle \text{infinite } S \rangle$  *LeastI* *pick.simps(1)* **by** (*metis Collect-mem-eq not-finite-existsD*)

**next**

**fix** *n'* **assume**  $n = \text{Suc } n'$

**obtain** *a* **where**  $a \in S \wedge a > \text{pick } S \ n'$  **using** *assms* **by** (*metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff*)

**show**  $\text{pick } S \ n \in S$  **unfolding**  $\langle n = \text{Suc } n' \rangle$  *pick.simps(2)*

**using** *LeastI*[of  $\lambda a. a \in S \wedge \text{pick } S \ n' < a \ a, OF \ \langle a \in S \wedge a > \text{pick } S \ n' \rangle$ ] **by**  
*blast*  
**qed**

**lemma** *pick-mono-inf*:  
**assumes** *infinite S*  
**shows**  $m < n \implies \text{pick } S \ m < \text{pick } S \ n$   
**using** *assms proof (induction n)*  
**case** *0*  
**then show** *?case by auto*  
**next**  
**case** *(Suc n)*  
**then obtain** *a where  $a \in S \wedge \text{pick } S \ n < a$  by (metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff)*  
**then have**  $\text{pick } S \ n < \text{pick } S \ (\text{Suc } n)$  **unfolding** *pick.simps*  
**using** *LeastI*[of  $\lambda a. a \in S \wedge \text{pick } S \ n < a \ a, OF \ \langle a \in S \wedge a > \text{pick } S \ n \rangle$ ] **by**  
*simp*  
**then show** *?case using Suc.IH Suc.prem(1) assms dual-order.strict-trans less-Suc-eq*  
**by auto**  
**qed**

**lemma** *pick-eq-iff-inf*:  
**assumes** *infinite S*  
**shows**  $x = y \iff \text{pick } S \ x = \text{pick } S \ y$   
**by (metis assms nat-neq-iff pick-mono-inf)**

**lemma** *card-le-pick-inf*:  
**assumes** *infinite S*  
**and**  $\text{pick } S \ n \geq i$   
**shows**  $\text{card } \{a \in S. a < i\} \leq n$   
**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**  $\text{card } \{a \in S. a < \text{pick } S \ n\} \leq n$  **using** *Suc by blast*  
**have**  $\{a \in S. a < i\} \subseteq \{a \in S. a < \text{pick } S \ (\text{Suc } n)\}$  **using** *Suc.prem(2) by*  
*auto*  
**have**  $\{a \in S. a < \text{pick } S \ (\text{Suc } n)\} = \{a \in S. a < \text{pick } S \ n\} \cup \{\text{pick } S \ n\}$   
**apply (rule subset-antisym; rule subsetI)**  
**using** *not-less-Least UnCI mem-Collect-eq nat-neq-iff singleton-conv*  
*pick-mono-inf*[*OF Suc.prem(1), of n Suc n*] *pick-in-set-inf*[*OF Suc.prem(1),*  
*of n*] **by fastforce+**  
**then have**  $\text{card } \{a \in S. a < i\} \leq \text{card } \{a \in S. a < \text{pick } S \ n\} + \text{card } \{\text{pick } S \ n\}$**

```

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.premis, of n Suc n] pick-in-set-inf[OF Suc.premis, 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  $\in S$  and
  card-pick-le: card  $\{a \in S. a < pick S n\} = n$  and
  pick-mono-le:  $m < n \implies pick S m < pick S n$ 
using assms proof (induction n)
  assume  $0 < card S$ 
  then obtain x where  $x \in S$  by fastforce
  then show pick S 0  $\in S$  unfolding pick.simps by (meson LeastI)
  then show card  $\{a \in S. a < pick S 0\} = 0$  using not-less-Least by auto
  show  $m < 0 \implies pick S m < pick S 0$  by auto
next
  fix n
  assume  $n < card S \implies pick S n \in S$ 
  and  $n < card S \implies card \{a \in S. a < pick S n\} = n$ 
  and  $Suc n < card S$ 
  and  $m < n \implies n < card S \implies pick S m < pick S n$ 
  then have card  $\{a \in S. a < pick S n\} = n$  pick S n  $\in S$  by linarith+
  have card  $\{a \in S. a > pick S n\} > 0$ 
  proof -

```

**have**  $S = \{a \in S. a < \text{pick } S \ n\} \cup \{a \in S. a \geq \text{pick } S \ n\}$  **by** *fastforce*  
**then have**  $\text{card } \{a \in S. a \geq \text{pick } S \ n\} > 1$   
**using**  $\langle \text{Suc } n < \text{card } S \rangle \langle \text{card } \{a \in S. a < \text{pick } S \ n\} = n \rangle$   
 $\text{card-Un-le}[\text{of } \{a \in S. a < \text{pick } S \ n\} \{a \in S. \text{pick } S \ n \leq a\}]$  **by** *force*  
**then have**  $0: \{a \in S. a \geq \text{pick } S \ n\} \subseteq \{\text{pick } S \ n\} \cup \{a \in S. a > \text{pick } S \ n\}$   
**by** *auto*  
**have**  $1: \text{finite } (\{\text{pick } S \ n\} \cup \{a \in S. \text{pick } S \ n < a\})$   
**unfolding** *finite-Un* **using** *Collect-mem-eq* *assms* *card-infinite* *conjI* **by** *force*  
**have**  $1 < \text{card } \{\text{pick } S \ n\} + \text{card } \{a \in S. \text{pick } S \ n < a\}$   
**using** *card-mono[OF 1 0]* *card-Un-le* [*of*  $\{\text{pick } S \ n\} \{a \in S. a > \text{pick } S \ n\}$ ]  
 $\langle \text{card } \{a \in S. a \geq \text{pick } S \ n\} > 1 \rangle$   
**by** *linarith*  
**then show** *?thesis* **by** *simp*  
**qed**  
**then show**  $\text{pick } S \ (\text{Suc } n) \in S$  **unfolding** *pick.simps*  
**by** (*metis* (*no-types*, *lifting*) *Collect-empty-eq* *LeastI* *card-0-eq* *card-infinite* *less-numeral-extra*(3))  
**have**  $\text{pick } S \ (\text{Suc } n) > \text{pick } S \ n$   
**by** (*metis* (*no-types*, *lifting*) *pick.simps*(2)  $\langle \text{card } \{a \in S. a > \text{pick } S \ n\} > 0 \rangle$  *Collect-empty-eq* *LeastI* *card-0-eq* *card-infinite* *less-numeral-extra*(3))  
**then show**  $m < \text{Suc } n \implies \text{pick } S \ m < \text{pick } S \ (\text{Suc } n)$   
**using**  $\langle m < n \implies n < \text{card } S \implies \text{pick } S \ m < \text{pick } S \ n \rangle$   
**using**  $\langle \text{Suc } n < \text{card } S \rangle$  *dual-order.strict-trans* *less-Suc-eq* **by** *auto*  
**then show**  $\text{card } \{a \in S. a < \text{pick } S \ (\text{Suc } n)\} = \text{Suc } n$   
**proof** –  
**have**  $\{a \in S. a < \text{pick } S \ (\text{Suc } n)\} = \{a \in S. a < \text{pick } S \ n\} \cup \{\text{pick } S \ n\}$   
**apply** (*rule subset-antisym*; *rule subsetI*)  
**using** *pick.simps* *not-less-Least*  $\langle \text{pick } S \ (\text{Suc } n) > \text{pick } S \ n \rangle \langle \text{pick } S \ n \in S \rangle$   
**by** *fastforce* +  
**then have**  $\text{card } \{a \in S. a < \text{pick } S \ (\text{Suc } n)\} = \text{card } \{a \in S. a < \text{pick } S \ n\}$   
 $+ \text{card } \{\text{pick } S \ n\}$  **using** *card-Un-disjoint* **by** *auto*  
**then show** *?thesis* **by** (*metis* *One-nat-def* *Suc-eq-plus1*  $\langle \text{card } \{a \in S. a < \text{pick } S \ n\} = n \rangle$  *card-empty* *card-insert-if* *empty-iff* *finite.emptyI*)  
**qed**  
**qed**

**lemma** *card-le-pick-le*:

**assumes**  $n < \text{card } S$

**and**  $\text{pick } S \ n \geq i$

**shows**  $\text{card } \{a \in S. a < i\} \leq n$

**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**  $(\text{Suc } n)$

**have**  $\text{card } \{a \in S. a < \text{pick } S \ n\} \leq n$  **using** *Suc* **by** (*simp* *add*: *less-eq-Suc-le* *nat-less-le*)

**have**  $\{a \in S. a < i\} \subseteq \{a \in S. a < \text{pick } S \text{ (Suc } n)\}$  **using** *Suc.prem(2)* **by** *auto*  
**have**  $\{a \in S. a < \text{pick } S \text{ (Suc } n)\} = \{a \in S. a < \text{pick } S \text{ } n\} \cup \{\text{pick } S \text{ } n\}$   
**apply** (*rule subset-antisym; rule subsetI*)  
**using** *pick.simps not-less-Least pick-mono-le[OF Suc.prem(1), of n, OF lessI]*  
*pick-in-set-le[of n S] Suc* **by** *fastforce+*  
**then have**  $\text{card } \{a \in S. a < i\} \leq \text{card } \{a \in S. a < \text{pick } S \text{ } n\} + \text{card } \{\text{pick } S \text{ } n\}$   
**using** *card-Un-disjoint card-mono[OF - ⟨{a ∈ S. a < i} ⊆ {a ∈ S. a < pick S (Suc n)}⟩]* **by** *simp*  
**then show**  $?case$  **using**  $\langle \text{card } \{a \in S. a < \text{pick } S \text{ } n\} \leq n \rangle$  **by** *auto*  
**qed**

**lemma**

**assumes**  $n < \text{card } S \vee \text{infinite } S$

**shows**

*pick-in-set: pick S n ∈ S* **and**

*card-le-pick:  $i \leq \text{pick } S \text{ } n \implies \text{card } \{a \in S. a < i\} \leq n$*  **and**

*card-pick:  $\text{card } \{a \in S. a < \text{pick } S \text{ } n\} = n$*  **and**

*pick-mono:  $m < n \implies \text{pick } S \text{ } m < \text{pick } S \text{ } n$*

**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**  $1: \text{pick } I \text{ (card } \{a \in I. a < i\}) = i$  **by** (*metis (mono-tags, lifting)*)

*Least-equality Suc.IH order-refl*)

**have**  $\{a \in I. a < \text{Suc } i\} = \{a \in I. a < i\} \cup \{i\}$  **using** *True* **by** *auto*

**then have**  $2: \text{card } \{a \in I. a < \text{Suc } i\} = \text{Suc } (\text{card } \{a \in I. a < i\})$  **by** *auto*

**then show**  $?thesis$  **unfolding**  $2$  *pick.simps 1* **using** *Suc-le-eq* **by** *auto*

**next**

**case** *False*

**then have**  $1: \{a \in I. a < \text{Suc } i\} = \{a \in I. a < i\}$  **using** *Collect-cong less-Suc-eq*

**by** *auto*

**have**  $2: \bigwedge a. (a \in I \wedge \text{Suc } i \leq a) = (a \in I \wedge i \leq a)$  **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*:  $i \in I \implies \text{pick } I (\text{card } \{a \in I. a < i\}) = i$   
**unfolding** *pick-card* **using** *Least-equality order-refl* **by** (*metis (no-types, lifting)*)

## 15 Sublist

**lemma** *nth-sublist-card*:  
**assumes**  $j < \text{length } xs$   
**and**  $j \in J$   
**shows**  $\text{sublist } xs J ! \text{card } \{j0. j0 < j \wedge j0 \in J\} = xs!j$   
**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**  $\{j0. j0 < j \wedge j0 \in J\} \subset \{i. i < \text{length } xs \wedge i \in J\}$   
**using** *True snoc.prem(2)* **by** *auto*  
**then have**  $\text{card } \{j0. j0 < j \wedge j0 \in J\} < \text{length } (\text{sublist } xs J)$  **unfolding**  
*length-sublist*  
**using** *psubset-card-mono[of {i. i < length xs \wedge i \in J}]* **by** *simp*  
**then show** *?thesis* **unfolding** *sublist-append nth-append* **by** (*simp add: True snoc.IH snoc.prem(2)*)  
**next**  
**case** *False*  
**then have**  $\text{length } xs = j$   
**using** *length-append-singleton less-antisym snoc.prem(1)* **by** *auto*  
**then show** *?thesis* **unfolding** *sublist-append nth-append length-sublist*  $\langle \text{length } xs = j \rangle$   
**by** (*simp add: snoc.prem(2)*)  
**qed**  
**qed**

**lemma** *pick-reduce-set*:  
**assumes**  $i < \text{card } \{a. a < m \wedge a \in I\}$   
**shows**  $\text{pick } I i = \text{pick } \{a. a < m \wedge a \in I\} i$   
**using** *assms* **proof** (*induction i*)  
**let**  $?L = \text{LEAST } a. a \in \{a. a < m \wedge a \in I\}$   
**case** *0*  
**then have**  $\{a. a < m \wedge a \in I\} \neq \{\}$  **using** *card-empty less-numeral-extra(3)*  
**by** *fastforce*  
**then have**  $?L \in I ?L < m$  **by** (*metis (mono-tags, lifting) Collect-empty-eq LeastI mem-Collect-eq*)  
**have**  $\bigwedge x. x \in \{a. a < m \wedge a \in I\} \implies ?L \leq x$  **by** (*simp add: Least-le*)  
**have**  $\bigwedge x. x \in I \implies ?L \leq x$   
**by** (*metis (mono-tags) (?L < m) (\bigwedge x. x \in \{a. a < m \wedge a \in I\} \implies ?L \leq x)*  
*dual-order.strict-trans2 le-cases mem-Collect-eq*)  
**then show** *?case* **unfolding** *pick.simps* **using** *Least-equality*[*of*  $\lambda x. x \in I$ , *OF*

$\langle ?L \in I \rangle$  **by** *blast*  
**next**  
**case** (*Suc i*)  
**let**  $?L = \text{LEAST } x. x \in \{a. a < m \wedge a \in I\} \wedge \text{pick } I \ i < x$   
**have**  $0 : \text{pick } \{a. a < m \wedge a \in I\} \ i = \text{pick } I \ i$  **using** *Suc-lessD Suc* **by** *linarith*  
**then have**  $?L \in \{a. a < m \wedge a \in I\} \ \text{pick } I \ i < ?L$   
**using** *LeastI*[of  $\lambda a. a \in \{a. a < m \wedge a \in I\} \wedge \text{pick } I \ i < a$ ] **using** *Suc.prem*  
*pick-in-set-le pick-mono-le* **by** *fastforce+*  
**then have**  $?L \in I$  **by** *blast*  
**show**  $?case$  **unfolding** *pick.simps 0* **using** *Least-equality*[of  $\lambda a. a \in I \wedge \text{pick } I \ i < a \ ?L$ ]  
**by** (*metis (no-types, lifting) Least-le*  $\langle ?L \in \{a. a < m \wedge a \in I\} \ \langle \text{pick } I \ i < ?L \rangle$  *mem-Collect-eq not-le not-less-iff-gr-or-eq order.trans*)  
**qed**

**lemma** *nth-sublist*:

**assumes**  $i < \text{card } \{i. i < \text{length } xs \wedge i \in I\}$   
**shows**  $\text{sublist } xs \ I \ ! \ i = xs \ ! \ \text{pick } I \ i$   
**proof** –  
**have**  $\{a \in \{i. i < \text{length } xs \wedge i \in I\}. a < \text{pick } \{i. i < \text{length } xs \wedge i \in I\} \ i\}$   
 $= \{a. a < \text{pick } \{i. i < \text{length } xs \wedge i \in I\} \ i \wedge a \in I\}$   
**using** *assms pick-in-set* **by** *fastforce*  
**then have**  $\text{card } \{a. a < \text{pick } \{i. i < \text{length } xs \wedge i \in I\} \ i \wedge a \in I\} = i$   
**using** *card-pick-le*[OF *assms*] **by** *simp*  
**then have**  $\text{sublist } xs \ I \ ! \ i = xs \ ! \ \text{pick } \{i. i < \text{length } xs \wedge i \in I\} \ i$   
**using** *nth-sublist-card*[**where**  $j = \text{pick } \{i. i < \text{length } xs \wedge i \in I\} \ i$ , 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*:  $\text{pick } UNIV \ j = j$

**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**  $n < \text{card } \{a. a < i \wedge a \in S\}$   
**shows**  $\text{pick } S \ n < i$   
**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-sublists*:

**fixes**  $f :: 'a \Rightarrow 'b :: \text{comm-monoid-mult}$

**shows**  $\text{prod-list } (\text{map } f \ xs) = \text{prod-list } (\text{map } f \ (\text{sublist } xs \ A)) * \text{prod-list } (\text{map } f \ (\text{sublist } xs \ (-A)))$

```

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 sublist-append sublist-singleton
  snoc
  by (cases (length xs)∈A; simp;metis mult.assoc mult.commute)
qed

lemma sublist-zip: sublist (zip xs ys) I = zip (sublist xs I) (sublist ys I)
proof (rule nth-equalityI)
  show length (sublist (zip xs ys) I) = length (zip (sublist xs I) (sublist ys I))
  proof (cases length xs ≤ length ys)
    case True
    then have {i. i < length xs ∧ i ∈ I} ⊆ {i. i < length ys ∧ i ∈ I} by (simp
  add: Collect-mono less-le-trans)
    then have card {i. i < length xs ∧ i ∈ I} ≤ card {i. i < length ys ∧ i ∈ I}
    by (metis (mono-tags, lifting) card-mono finite-nat-set-iff-bounded mem-Collect-eq)
    then show ?thesis unfolding length-sublist length-zip using True using
  min-def by linarith
    next
    case False
    then have {i. i < length ys ∧ i ∈ I} ⊆ {i. i < length xs ∧ i ∈ I} by (simp
  add: Collect-mono less-le-trans)
    then have card {i. i < length ys ∧ i ∈ I} ≤ card {i. i < length xs ∧ i ∈ I}
    by (metis (mono-tags, lifting) card-mono finite-nat-set-iff-bounded mem-Collect-eq)
    then show ?thesis unfolding length-sublist length-zip using False using
  min-def by linarith
  qed
  show ∀i < length (sublist (zip xs ys) I). sublist (zip xs ys) I ! i = zip (sublist xs
  I) (sublist ys I) ! i
  proof (rule allI; rule impI)
    fix i assume i < length (sublist (zip xs ys) I)
    then have i < length (sublist xs I) i < length (sublist ys I)
    by (simp-all add: ⟨length (sublist (zip xs ys) I) = length (zip (sublist xs I)
  (sublist ys I))⟩)
    show sublist (zip xs ys) I ! i = zip (sublist xs I) (sublist ys I) ! i
    unfolding nth-sublist[OF ⟨i < length (sublist (zip xs ys) I)⟩[unfolded length-sublist]]

    unfolding nth-zip[OF ⟨i < length (sublist xs I)⟩ ⟨i < length (sublist ys I)⟩]
    unfolding nth-zip[OF pick-le[OF ⟨i < length (sublist xs I)⟩][unfolded length-sublist]]

    pick-le[OF ⟨i < length (sublist ys I)⟩[unfolded length-sublist]]
    by (metis (full-types) ⟨i < length (sublist xs I)⟩ ⟨i < length (sublist ys I)⟩
  length-sublist nth-sublist)
  qed
qed

```

end

## 16 Tensor Matricization

**theory** *Tensor-Matricization*

**imports** *Tensor-Plus*

*../Jordan-Normal-Form/Matrix DL-Missing-Sublist*

**begin**

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

**fun** *digit-encode* :: *nat list*  $\Rightarrow$  *nat*  $\Rightarrow$  *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*  $\triangleleft$  *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*  $\triangleleft$  *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**  $a < \text{prod-list } ds$   
**shows**  $\text{digit-encode } ds \ a \triangleleft ds$   
**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**  $\text{length } (\text{digit-encode } ds \ a) = \text{length } ds$   
**by** (*induction ds arbitrary:a; simp-all*)

**lemma** *digit-encode-0*:  
 $\text{prod-list } ds \ \text{dvd } a \implies \text{digit-encode } ds \ a = \text{replicate } (\text{length } ds) \ 0$   
**proof** (*induction ds arbitrary:a*)  
  **case** *Nil*  
  **then show** *?case* **by** *simp*  
**next**  
  **case** (*Cons d ds a*)  
  **then have**  $\text{prod-list } ds \ \text{dvd } (a \ \text{div } d)$  **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**

**definition** *weave* ::  $\text{nat set} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list}$  **where**  
 $\text{weave } A \ xs \ ys = \text{map } (\lambda i. \ \text{if } i \in A \ \text{then } xs!(\text{card } \{a \in A. \ a < i\}) \ \text{else } ys!(\text{card } \{a \in -A. \ a < i\})) \ [0..<\text{length } xs + \text{length } ys]$

**lemma** *length-weave*:  
**shows**  $\text{length } (\text{weave } A \ xs \ ys) = \text{length } xs + \text{length } ys$   
**unfolding** *weave-def length-map* **by** *simp*

**lemma** *nth-weave*:  
**assumes**  $i < \text{length } (\text{weave } A \ xs \ ys)$   
**shows**  $\text{weave } A \ xs \ ys \ ! \ i = (\text{if } i \in A \ \text{then } xs!(\text{card } \{a \in A. \ a < i\}) \ \text{else } ys!(\text{card } \{a \in -A. \ a < i\}))$   
**proof** –  
  **have**  $i < \text{length } xs + \text{length } ys$  **using** *length-weave* **using** *assms* **by** *metis*  
  **then have**  $i < \text{length } [0..<\text{length } xs + \text{length } ys]$  **by** *auto*  
  **then have**  $[0..<\text{length } xs + \text{length } ys] \ ! \ i = i$

```

    by (metis ⟨i < length xs + length ys⟩ add.left-neutral nth-upt)
  then show ?thesis
    unfolding weave-def nth-map[OF ⟨i < length [0.. $\text{length } xs + \text{length } ys$ ⟩] 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  $\notin$  A
assumes length ys = card {a $\in$ -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  $\forall i < \text{length (weave A xs (ys @ [y]))}. \text{weave A xs (ys @ [y]) ! } i = (\text{weave A xs ys @ [y]) ! } i$ 
    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  $\in$  -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
          case False
            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  $\notin$  A
              have i < length xs + length ys by (metis ⟨i < length (weave A xs ys)⟩ length-weave)
              then have {a  $\in$  -A. a < i}  $\subset$  {a $\in$ -A. a < length xs + length ys}
                using assms(1) ⟨i < length xs + length ys⟩ ⟨i  $\notin$  A⟩ by auto
              then have card {a  $\in$  -A. a < i} < card {a $\in$ -A. a < length xs + length ys}
                using psubset-card-mono[of {a $\in$ -A. a < length xs + length ys} {a  $\in$  -A. a < i}] by simp
              then have (ys @ [y]) ! card {a  $\in$  -A. a < i} = ys ! card {a  $\in$  -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** *valid-index-list-all2-iff*:  $is \triangleleft ds \iff list\text{-all2 } (op <) is ds$   
**by** (*metis list-all2-conv-all-nth list-all2-nthD valid-indexI valid-index-length valid-index-lt*)

**lemma** *sublist-nth*:

**assumes**  $n \in A \ n < \text{length } xs$

**shows**  $sublist\ xs\ A \ ! \ (\text{card } \{i. i < n \wedge i \in A\}) = xs \ ! \ n$

**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** *sublist-append[of xs [x] A] nth-append*

**using** *length-sublist[of xs A] sublist-singleton snoc.prem1* **by** *auto*

**next**

**case** *False*

**then have**  $n < \text{length } xs$  **using** *snoc* **by** *auto*

**then have**  $0 : sublist\ xs\ A \ ! \ \text{card } \{i. i < n \wedge i \in A\} = xs \ ! \ n$  **using** *snoc* **by**

*auto*

**have**  $\{i. i < n \wedge i \in A\} \subset \{i. i < \text{length } xs \wedge i \in A\}$  **using**  $\langle n < \text{length } xs \rangle$   
*snoc* **by** *force*

**then have**  $\text{card } \{i. i < n \wedge i \in A\} < \text{length } (sublist\ xs\ A)$  **unfolding**  
*length-sublist*

**by** (*simp add: psubset-card-mono*)

**then show** *?thesis* **unfolding** *sublist-append[of xs [x] A] nth-append* **using** *0*

**by** (*simp add: ‹n < length xs›*)

**qed**

**qed**

**lemma** *list-all2-sublist*:

**assumes**  $list\text{-all2 } P (sublist\ xs\ A) (sublist\ ys\ A)$

**and**  $list\text{-all2 } P (sublist\ xs\ (-A)) (sublist\ ys\ (-A))$

**shows**  $list\text{-all2 } P\ xs\ ys$

**proof** –

**have**  $\text{length } xs = \text{length } ys$

**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 (sublist ys (-A)) > length (sublist xs (-A))
    unfolding length-sublist 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 (sublist ys A) > length (sublist xs A)
    unfolding length-sublist 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 (sublist xs (-A)) > length (sublist ys (-A))
      unfolding length-sublist 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 (sublist xs A) > length (sublist ys A)
    unfolding length-sublist 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  $\text{card } \{i. i < n \wedge i \in A\} < \text{length } (\text{sublist } xs \ A)$  unfolding
length-sublist
  by (simp add: psubset-card-mono)
  show ?thesis using sublist-nth[OF ⟨n∈A⟩ ⟨n < length xs⟩] sublist-nth[OF
⟨n∈A⟩ ⟨n < length ys⟩]
  list-all2-nthD[OF assms(1), of card {i. i < n ∧ i ∈ A}] length-sublist
  by (simp add: ⟨card {i. i < n ∧ i ∈ A} < length (sublist xs A)⟩)
next
  case False then have n∈-A by auto
  have  $\{i. i < n \wedge i \in -A\} \subset \{i. i < \text{length } xs \wedge i \in -A\}$  using ⟨n < length
xs⟩ ⟨n∈-A⟩ by force
  then have  $\text{card } \{i. i < n \wedge i \in -A\} < \text{length } (\text{sublist } xs \ (-A))$  unfolding
length-sublist
  by (simp add: psubset-card-mono)
  show ?thesis using sublist-nth[OF ⟨n∈-A⟩ ⟨n < length xs⟩] sublist-nth[OF
⟨n∈-A⟩ ⟨n < length ys⟩]
  list-all2-nthD[OF assms(2), of card {i. i < n ∧ i ∈ -A}] length-sublist
  using ⟨card {i. i < n ∧ i ∈ -A} < length (sublist xs (-A))⟩ by auto
next
  qed
  qed
  then show ?thesis using ⟨length xs = length ys⟩ list-all2-all-nthI by blast
qed

```

**lemma** *sublist-weave*:

```

assumes  $\text{length } xs = \text{card } \{a \in A. a < \text{length } xs + \text{length } ys\}$ 
assumes  $\text{length } ys = \text{card } \{a \in (-A). a < \text{length } xs + \text{length } ys\}$ 
shows  $\text{sublist } (\text{weave } A \ xs \ ys) \ A = xs \wedge \text{sublist } (\text{weave } A \ xs \ ys) \ (-A) = ys$ 
using assms proof (induction length xs + length ys arbitrary: xs ys)
  case 0
  then show ?case
  unfolding weave-def sublist-map by simp
next
  case (Suc l)
  then show ?case
  proof (cases l∈A)
    case True
    then have  $l \in \{a \in A. a < \text{length } xs + \text{length } ys\}$  using Suc.hyps mem-Collect-eq
zero-less-Suc by auto
    then have  $\text{length } xs > 0$  using Suc by fastforce
    then obtain  $xs' \ x$  where  $xs = xs' \ @ \ [x]$  by (metis append-butlast-last-id
length-greater-0-conv)
    then have  $l = \text{length } xs' + \text{length } ys$  using Suc.hyps by simp
    have  $\text{length-}xs':\text{length } xs' = \text{card } \{a \in A. a < \text{length } xs' + \text{length } ys\}$ 

```

**proof** –  
**have**  $\{a \in A. a < \text{length } xs + \text{length } ys\} = \{a \in A. a < \text{length } xs' + \text{length } ys\} \cup \{l\}$   
**using**  $\langle xs = xs' @ [x] \rangle \langle l \in \{a \in A. a < \text{length } xs + \text{length } ys\} \rangle \langle l = \text{length } xs' + \text{length } ys \rangle$   
**by force**  
**then have**  $\text{card } \{a \in A. a < \text{length } xs + \text{length } ys\} = \text{card } \{a \in A. a < \text{length } xs' + \text{length } ys\} + 1$   
**using**  $\langle l = \text{length } xs' + \text{length } ys \rangle$  **by fastforce**  
**then show** *?thesis* **by** (*metis One-nat-def Suc.premis(1)*)  $\langle xs = xs' @ [x] \rangle$   
*add-right-imp-eq*  
*length-Cons length-append list.size(3)*  
**qed**  
**have**  $\text{length-ys} : \text{length } ys = \text{card } \{a \in -A. a < \text{length } xs' + \text{length } ys\}$   
**proof** –  
**have**  $l \notin \{a \in -A. a < \text{length } xs + \text{length } ys\}$  **using**  $\langle l \in A \rangle \langle l = \text{length } xs' + \text{length } ys \rangle$  **by blast**  
**have**  $\{a \in -A. a < \text{length } xs + \text{length } ys\} = \{a \in -A. a < \text{length } xs' + \text{length } ys\}$   
**apply** (*rule subset-antisym*)  
**using**  $\langle l = \text{length } xs' + \text{length } ys \rangle \langle \text{Suc } l = \text{length } xs + \text{length } ys \rangle \langle l \notin \{a \in -A. a < \text{length } xs + \text{length } ys\} \rangle$   
**apply** (*metis (no-types, lifting) Collect-mono less-Suc-eq mem-Collect-eq*)  
**using** *Collect-mono Suc.hyps(2)*  $\langle l = \text{length } xs' + \text{length } ys \rangle$  **by auto**  
**then show** *?thesis* **using** *Suc.premis(2)* **by auto**  
**qed**  
**have**  $\text{length } xs' + \text{length } ys \in A$  **using**  $\langle l \in A \rangle \langle l = \text{length } xs' + \text{length } ys \rangle$  **by blast**

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

**have**  $\text{sublist } (\text{weave } A \ xs \ ys) \ (-A) = \text{sublist } (\text{weave } A \ xs' \ ys \ @ \ [x]) \ (-A)$   
**unfolding**  
 $\langle xs = xs' @ [x] \rangle$  **using** *weave-append1[OF length xs' + length ys ∈ A]*  
*length-xs'* **by metis**  
**also have**  $\dots = \text{sublist } (\text{weave } A \ xs' \ ys) \ (-A) \ @ \ \text{sublist } [x] \ \{a. a + (\text{length } xs' + \text{length } ys) \in (-A)\}$

**using** *sublist-append length-weave* **by** *metis*  
**also have** ... = *sublist (weave A xs' ys) (-A)*  
**using** *sublist-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]* **by**  
*presburger*  
**finally show** *?thesis using (sublist (weave A xs ys) A = xs)* **by** *auto*  
**next**  
**case** *False*  
**then have**  $l \notin \{a \in A. a < \text{length } xs + \text{length } ys\}$  **using** *Suc.hyps mem-Collect-eq*  
*zero-less-Suc* **by** *auto*  
**then have**  $\text{length } ys > 0$  **using** *Suc* **by** *fastforce*  
**then obtain**  $ys' y$  **where**  $ys = ys' @ [y]$  **by** (*metis append-butlast-last-id*  
*length-greater-0-conv*)  
**then have**  $l = \text{length } xs + \text{length } ys'$  **using** *Suc.hyps* **by** *simp*  
**have**  $\text{length-ys}':\text{length } ys' = \text{card } \{a \in -A. a < \text{length } xs + \text{length } ys'\}$   
**proof** -  
**have**  $\{a \in -A. a < \text{length } xs + \text{length } ys\} = \{a \in -A. a < \text{length } xs +$   
 $\text{length } ys'\} \cup \{l\}$   
**using**  $\langle ys = ys' @ [y] \rangle \langle l \notin \{a \in A. a < \text{length } xs + \text{length } ys\} \rangle \langle l = \text{length}$   
 $xs + \text{length } ys' \rangle$   
**by** *force*  
**then have**  $\text{card } \{a \in -A. a < \text{length } xs + \text{length } ys\} = \text{card } \{a \in -A. a <$   
 $\text{length } xs + \text{length } ys'\} + 1$   
**using**  $\langle l = \text{length } xs + \text{length } ys' \rangle$  **by** *fastforce*  
**then show** *?thesis* **by** (*metis One-nat-def Suc.prem(2) (ys = ys' @ [y])*  
*add-right-imp-eq*  
*length-Cons length-append list.size(3)*)  
**qed**  
**have**  $\text{length-xs}:\text{length } xs = \text{card } \{a \in A. a < \text{length } xs + \text{length } ys'\}$   
**proof** -  
**have**  $l \notin \{a \in A. a < \text{length } xs + \text{length } ys\}$  **using**  $\langle l \notin A \rangle \langle l = \text{length } xs +$   
 $\text{length } ys' \rangle$  **by** *blast*  
**have**  $\{a \in A. a < \text{length } xs + \text{length } ys\} = \{a \in A. a < \text{length } xs + \text{length}$   
 $ys'\}$   
**apply** (*rule subset-antisym*)  
**using**  $\langle l = \text{length } xs + \text{length } ys' \rangle \langle \text{Suc } l = \text{length } xs + \text{length } ys \rangle \langle l \notin \{a \in$   
 $A. a < \text{length } xs + \text{length } ys'\} \rangle$   
**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.prem(1)* **by** *auto*  
**qed**  
**have**  $\text{length } xs + \text{length } ys' \notin A$  **using**  $\langle l \notin A \rangle \langle l = \text{length } xs + \text{length } ys' \rangle$  **by**  
*blast*  
  
**then have** *sublist (weave A xs ys) A = sublist (weave A xs ys' @ [y]) A*  
**unfolding**  
 $\langle ys = ys' @ [y] \rangle$  **using** *weave-append2[OF (length xs + length ys' ∉ A)*  
*length-ys']* **by** *metis*

**also have** ... = *sublist* (weave  $A$   $xs$   $ys'$ )  $A$  @ *sublist*  $[y]$  { $a$ .  $a + (\text{length } xs + \text{length } ys') \in A$ }  
**using** *sublist-append length-weave* **by** *metis*  
**also have** ... = *sublist* (weave  $A$   $xs$   $ys'$ )  $A$   
**using** *sublist-singleton*  $\langle \text{length } xs + \text{length } ys' \notin A \rangle$  **by** *auto*  
**also have** ... =  $xs$   
**using** *Suc.hyps*(1)[*OF*  $\langle l = \text{length } xs + \text{length } ys' \rangle$  *length-xs length-ys'*] **by**  
*auto*  
**finally have** *sublist* (weave  $A$   $xs$   $ys$ )  $A = xs$  **by** *auto*

**have** *sublist* (weave  $A$   $xs$   $ys$ )  $(-A) = \text{sublist}$  (weave  $A$   $xs$   $ys'$  @  $[y]$ )  $(-A)$   
**unfolding**  
 $\langle ys = ys' @ [y] \rangle$  **using** *weave-append2*[*OF*  $\langle \text{length } xs + \text{length } ys' \notin A \rangle$   
*length-ys'*] **by** *metis*  
**also have** ... = *sublist* (weave  $A$   $xs$   $ys'$ )  $(-A)$  @ *sublist*  $[y]$  { $a$ .  $a + (\text{length } xs + \text{length } ys') \in (-A)$ }  
**using** *sublist-append length-weave* **by** *metis*  
**also have** ... = *sublist* (weave  $A$   $xs$   $ys'$ )  $(-A)$  @  $[y]$   
**using** *sublist-singleton*  $\langle \text{length } xs + \text{length } ys' \notin A \rangle$  **by** *auto*  
**also have** ... =  $ys$   
**using** *Suc.hyps*(1)[*OF*  $\langle l = \text{length } xs + \text{length } ys' \rangle$  *length-xs length-ys'*]  $\langle ys = ys' @ [y] \rangle$  **by** *simp*  
**finally show** ?thesis **using**  $\langle \text{sublist}$  (weave  $A$   $xs$   $ys$ )  $A = xs \rangle$  **by** *auto*  
**qed**  
**qed**

**lemma** *set-weave*:

**assumes**  $\text{length } xs = \text{card } \{a \in A. a < \text{length } xs + \text{length } ys\}$

**assumes**  $\text{length } ys = \text{card } \{a \in -A. a < \text{length } xs + \text{length } ys\}$

**shows**  $\text{set} (\text{weave } A \text{ } xs \text{ } ys) = \text{set } xs \cup \text{set } ys$

**proof**

**show**  $\text{set} (\text{weave } A \text{ } xs \text{ } ys) \subseteq \text{set } xs \cup \text{set } ys$

**proof**

**fix**  $x$  **assume**  $x \in \text{set} (\text{weave } A \text{ } xs \text{ } ys)$

**then obtain**  $i$  **where**  $\text{weave } A \text{ } xs \text{ } ys ! i = x$   $i < \text{length} (\text{weave } A \text{ } xs \text{ } ys)$  **by**  
*(meson in-set-conv-nth)*

**show**  $x \in \text{set } xs \cup \text{set } ys$

**proof** (*cases*  $i \in A$ )

**case** *True*

**then have**  $i \in \{a \in A. a < \text{length } xs + \text{length } ys\}$  **unfolding** *length-weave*

**by** (*metis*  $\langle i < \text{length} (\text{weave } A \text{ } xs \text{ } ys) \rangle$  *length-weave mem-Collect-eq*)

**then have**  $\{a \in A. a < i\} \subset \{a \in A. a < \text{length } xs + \text{length } ys\}$

**using** *Collect-mono*  $\langle i < \text{length} (\text{weave } A \text{ } xs \text{ } ys) \rangle$ [*unfolded length-weave*]

*le-Suc-ex less-imp-le-nat trans-less-add1*

*le-neq-trans less-irrefl mem-Collect-eq* **by** *auto*

**then have**  $\text{card } \{a \in A. a < i\} < \text{card } \{a \in A. a < \text{length } xs + \text{length } ys\}$

**by** (*simp add: psubset-card-mono*)

**then show**  $x \in \text{set } xs \cup \text{set } ys$

**unfolding** *nth-weave*[*OF*  $\langle i < \text{length} (\text{weave } A \text{ } xs \text{ } ys) \rangle$ ], *unfolded*  $\langle \text{weave } A \text{ } xs$

```

ys ! i = x] using True
  using UnI1 assms(1) nth-mem by auto
next
case False
have  $i \notin A \implies i \in \{a \in -A. a < \text{length } xs + \text{length } ys\}$  unfolding length-weave
  by (metis ComplI  $\langle i < \text{length } (\text{weave } A \text{ } xs \text{ } ys) \rangle$  length-weave mem-Collect-eq)
then have  $\{a \in -A. a < i\} \subset \{a \in -A. a < \text{length } xs + \text{length } ys\}$ 
  using Collect-mono  $\langle i < \text{length } (\text{weave } A \text{ } xs \text{ } ys) \rangle$  [unfolded length-weave]
le-Suc-ex less-imp-le-nat trans-less-add1
  le-neq-trans less-irrefl mem-Collect-eq using False by auto
then have  $\text{card } \{a \in -A. a < i\} < \text{card } \{a \in -A. a < \text{length } xs + \text{length } ys\}$ 
  by (simp add: psubset-card-mono)
then show  $x \in \text{set } xs \cup \text{set } ys$ 
  unfolding nth-weave[OF  $\langle i < \text{length } (\text{weave } A \text{ } xs \text{ } ys) \rangle$ , unfolded  $\langle \text{weave } A \text{ } xs \text{ } ys \rangle$ ] using False
  using UnI1 assms(2) nth-mem by auto
qed
qed
show  $\text{set } xs \cup \text{set } ys \subseteq \text{set } (\text{weave } A \text{ } xs \text{ } ys)$ 
  using sublist-weave[OF assms] by (metis Un-subset-iff set-sublist-subset)
qed

```

**lemma** *weave-complementary-sublists*[simp]:  
 $\text{weave } A (\text{sublist } xs \ A) (\text{sublist } xs \ (-A)) = xs$

**proof** (*induction xs rule:rev-induct*)

```

case Nil
then show ?case by (metis gen-length-def length-0-conv length-code length-weave
sublist-nil)
next
case (snoc x xs)
have length-xs:  $\text{length } xs = \text{length } (\text{sublist } xs \ A) + \text{length } (\text{sublist } xs \ (-A))$  by
(metis length-weave snoc.IH)
show ?case
proof (cases  $(\text{length } xs) \in A$ )
case True
have 0:  $\text{length } (\text{sublist } xs \ A) + \text{length } (\text{sublist } xs \ (-A)) \in A$  using length-xs
True by metis
have 1:  $\text{length } (\text{sublist } xs \ A) = \text{card } \{a \in A. a < \text{length } (\text{sublist } xs \ A) + \text{length } (\text{sublist } xs \ (-A))\}$ 
  using length-sublist[of xs A] by (metis (no-types, lifting) Collect-cong length-xs)
have 2:  $\text{sublist } (xs \ @ \ [x]) \ A = \text{sublist } xs \ A \ @ \ [x]$ 
  unfolding sublist-append[of xs [x] A] using sublist-singleton True by auto
have 3:  $\text{sublist } (xs \ @ \ [x]) \ (-A) = \text{sublist } xs \ (-A)$ 
  unfolding sublist-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:  $\text{length } (\text{sublist } xs \ A) + \text{length } (\text{sublist } xs \ (-A)) \notin A$  using length-xs

```

*False by metis*

```

have 1:length (sublist xs (-A)) = card {a ∈ -A. a < length (sublist xs A) +
length (sublist xs (-A))}
using length-sublist[of xs -A] by (metis (no-types, lifting) Collect-cong
length-xs)
have 2:sublist (xs @ [x]) A = sublist xs A
unfolding sublist-append[of xs [x] A] using sublist-singleton False by auto
have 3:sublist (xs @ [x]) (-A) = sublist xs (-A) @ [x]
unfolding sublist-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-sublist'*:

```

length (sublist xs I) = card {i ∈ I. i < length xs}
unfolding length-sublist by meson

```

**lemma** *valid-index-weave*:

```

assumes is1 < (sublist ds A)
and is2 < (sublist ds (-A))
shows weave A is1 is2 < ds
and sublist (weave A is1 is2) A = is1
and sublist (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-sublists by metis
have 1:length is1 = card {i ∈ A. i < length is1 + length is2} unfolding
length-ds
using length-sublist' assms(1) valid-index-length by auto
have 2:length is2 = card {i ∈ -A. i < length is1 + length is2} unfolding
length-ds
using length-sublist'[of ds -A] assms(2) valid-index-length by auto
show sublist (weave A is1 is2) A = is1 sublist (weave A is1 is2) (-A) = is2
using sublist-weave[OF 1 2] by blast+
then have sublist (weave A is1 is2) A < (sublist ds A)
sublist (weave A is1 is2) (-A) < (sublist ds (-A)) using assms by auto
then show weave A is1 is2 < ds using list-all2-sublist valid-index-list-all2-iff
by blast
qed

```

**definition** *matricize* :: nat set ⇒ 'a tensor ⇒ 'a mat **where**

```

matricize rmodes T = mat
(prod-list (sublist (Tensor.dims T) rmodes))
(prod-list (sublist (Tensor.dims T) (-rmodes)))
(λ(r, c). Tensor.lookup T (weave rmodes
(digit-encode (sublist (Tensor.dims T) rmodes) r)
(digit-encode (sublist (Tensor.dims T) (-rmodes)) c)
))

```

**definition** *dematricize*::*nat set*  $\Rightarrow$  '*a mat*  $\Rightarrow$  *nat list*  $\Rightarrow$  '*a tensor* **where**  
*dematricize rmodes A ds = tensor-from-lookup ds*  
 (*λis. A \$\$ (digit-decode (sublist ds rmodes) (sublist is rmodes),*  
     *digit-decode (sublist ds (-rmodes)) (sublist is (-rmodes)))*  
 )

**lemma** *dims-matricize*:  
*dim<sub>r</sub> (matricize rmodes T) = prod-list (sublist (Tensor.dims T) rmodes)*  
*dim<sub>c</sub> (matricize rmodes T) = prod-list (sublist (Tensor.dims T) (-rmodes))*  
**unfolding** *matricize-def using mat-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-sublist*:  
**assumes** *is*  $\triangleleft$  *ds*  
**shows** *sublist is A*  $\triangleleft$  *sublist ds A*  
**using** *assms proof (induction arbitrary:A rule:valid-index.induct)*  
**case** *Nil*  
**then show** *?case using sublist-nil valid-index.simps by blast*  
**next**  
**case** (*Cons is ds i d*)  
**then have** *sublist is {j. Suc j  $\in$  A}  $\triangleleft$  sublist ds {j. Suc j  $\in$  A}*  
**by** *simp*  
**then show** *?case unfolding sublist-Cons*  
**by** (*cases 0  $\in$  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*  $\triangleleft$  *Tensor.dims (dematricize rmodes (matricize rmodes T) (Tensor.dims T))*  
**then have** *is*  $\triangleleft$  *Tensor.dims T using 1 by auto*  
**let** *?rds = (sublist (Tensor.dims T) rmodes)*  
**let** *?cds = (sublist (Tensor.dims T) (-rmodes))*  
**have** *decode-r: digit-decode ?rds (sublist is rmodes) < prod-list ?rds*  
**by** (*simp add:  $\langle is \triangleleft Tensor.dims T \rangle$  valid-index-sublist digit-decode-lt*)  
**have** *decode-c: digit-decode ?cds (sublist is (-rmodes)) < prod-list ?cds*  
**by** (*simp add:  $\langle is \triangleleft Tensor.dims T \rangle$  valid-index-sublist digit-decode-lt*)  
**have** (*matricize rmodes T*) \$\$  
 (*digit-decode ?rds (sublist is rmodes),*  
   *digit-decode ?cds (sublist is (- rmodes))*) =

$Tensor.lookup\ T\ is$   
**unfolding** *matricize-def*  
**by** (*simp add: decode-r decode-c*  $\langle is \triangleleft Tensor.dims\ T \rangle\ valid-index-sublist$ )  
**then show**  $Tensor.lookup\ (dematricize\ rmodes\ (matricize\ rmodes\ T))\ (Tensor.dims\ T) = Tensor.lookup\ T\ is$   
**by** (*simp add: dematricize-def dims-tensor-from-lookup lookup-tensor-from-lookup* [*OF*  $\langle is \triangleleft Tensor.dims\ T \rangle$ ])  
**qed**

**lemma** *matricize-dematricize:*

**assumes**  $dim_r\ A = prod-list\ (sublist\ ds\ rmodes)$

**and**  $dim_c\ A = prod-list\ (sublist\ ds\ (-rmodes))$

**shows**  $matricize\ rmodes\ (dematricize\ rmodes\ A\ ds) = A$

**proof** (*rule mat-eqI*)

**show**  $dim_r\ (matricize\ rmodes\ (dematricize\ rmodes\ A\ ds)) = dim_r\ A$

**unfolding** *assms(1) dematricize-def dims-tensor-from-lookup matricize-def mat-dim-row-mat* **by** *metis*

**show**  $dim_c\ (matricize\ rmodes\ (dematricize\ rmodes\ A\ ds)) = dim_c\ A$

**unfolding** *assms(2) dematricize-def dims-tensor-from-lookup matricize-def mat-dim-col-mat* **by** *metis*

**fix**  $r\ c$  **assume**  $r < dim_r\ A\ c < dim_c\ A$

**have** *valid1:digit-encode*  $(sublist\ ds\ rmodes)\ r \triangleleft sublist\ ds\ rmodes$  **and**  
*valid2:digit-encode*  $(sublist\ ds\ (-rmodes))\ c \triangleleft sublist\ ds\ (-rmodes)$

**using**  $\langle r < dim_r\ A \rangle\ assms(1)\ \langle c < dim_c\ A \rangle\ assms(2)\ digit-encode-valid-index$

**by** *auto*

**have**  $0:Tensor.lookup\ (dematricize\ rmodes\ A\ ds)$

*(weave rmodes*

*(digit-encode*  $(sublist\ (Tensor.dims\ (dematricize\ rmodes\ A\ ds))\ rmodes)\ r$

*(digit-encode*  $(sublist\ (Tensor.dims\ (dematricize\ rmodes\ A\ ds))\ (-rmodes))$

*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*  $\langle c < dim_c\ A \rangle$  [*unfolded assms(2)*]]

*digit-decode-encode-lt* [*OF*  $\langle r < dim_r\ A \rangle$  [*unfolded assms(1)*]]

*valid-index-weave(2)* [*OF valid1 valid2*] *valid-index-weave(3)* [*OF valid1 valid2*]

**by** *presburger*

**from**  $\langle r < dim_r\ A \rangle$  **have** *r-le:*  $r < prod-list\ (sublist\ (Tensor.dims\ (dematricize\ rmodes\ A\ ds))\ rmodes)$

**by** (*metis*  $\langle dim_r\ (matricize\ rmodes\ (dematricize\ rmodes\ A\ ds)) = dim_r\ A \rangle$  *matricize-def mat-dim-row-mat(1)*)

**from**  $\langle c < dim_c\ A \rangle$  **have** *c-le:*  $c < prod-list\ (sublist\ (Tensor.dims\ (dematricize\ rmodes\ A\ ds))\ (-rmodes))$

**by** (*metis*  $\langle dim_c\ (matricize\ rmodes\ (dematricize\ rmodes\ A\ ds)) = dim_c\ A \rangle$  *matricize-def mat-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**  $\text{dims } A = \text{dims } B$   
**shows**  $\text{matricize } I A \oplus_m \text{matricize } I B = \text{matricize } I (A+B)$   
**proof** (*rule mat-eqI*)  
  **show**  $\text{dim}_r (\text{matricize } I A \oplus_m \text{matricize } I B) = \text{dim}_r (\text{matricize } I (A + B))$  **by**  
  (*simp add: assms dims-matricize(1)*)  
  **show**  $\text{dim}_c (\text{matricize } I A \oplus_m \text{matricize } I B) = \text{dim}_c (\text{matricize } I (A + B))$  **by**  
  (*simp add: assms dims-matricize(2)*)  
  **fix**  $i j$  **assume**  $ij\text{-le1}: i < \text{dim}_r (\text{matricize } I (A + B))$   $j < \text{dim}_c (\text{matricize } I (A + B))$   
  **then have**  
     $ij\text{-le2}: i < \text{prod-list } (\text{sublist } (\text{Tensor.dims } A) I)$   $j < \text{prod-list } (\text{sublist } (\text{Tensor.dims } A) (-I))$  **and**  
     $ij\text{-le3}: i < \text{prod-list } (\text{sublist } (\text{Tensor.dims } B) I)$   $j < \text{prod-list } (\text{sublist } (\text{Tensor.dims } B) (-I))$  **and**  
     $ij\text{-le4}: i < \text{prod-list } (\text{sublist } (\text{Tensor.dims } (A + B)) I)$   $j < \text{prod-list } (\text{sublist } (\text{Tensor.dims } (A + B)) (-I))$   
    **by** (*simp-all add: assms dims-matricize*)  
    **then have**  $ij\text{-le5}: i < \text{dim}_r (\text{matricize } I B)$   $j < \text{dim}_c (\text{matricize } I B)$   
    **by** (*simp-all add: assms dims-matricize*)  
    **show**  $(\text{matricize } I A \oplus_m \text{matricize } I B) \text{ \textit{\$} \textit{\$} } (i, j) = \text{matricize } I (A + B) \text{ \textit{\$} \textit{\$} } (i, j)$   
  **unfolding** *mat-index-add(1)*[*OF ij-le5*] **unfolding** *matricize-def* **unfolding**  
  *mat-index-mat*[*OF ij-le2*] *mat-index-mat*[*OF ij-le3*] *mat-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**  $\text{matricize } I (\text{tensor0 } ds) = \mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds)))$   
**proof** (*rule mat-eqI*)  
  **show**  $\text{dim}_r (\text{matricize } I (\text{tensor0 } ds)) = \text{dim}_r (\mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds))))$   
  **unfolding** *mat-zero-def mat-dim-row-mat* **by** (*simp add: dims-matricize(1)*)  
  **show**  $\text{dim}_c (\text{matricize } I (\text{tensor0 } ds)) = \text{dim}_c (\mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds))))$   
  **unfolding** *mat-zero-def mat-dim-row-mat* **by** (*simp add: dims-matricize(2)*)  
  **fix**  $i j$  **assume**  $ij\text{-le1}: i < \text{dim}_r (\mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds))))$   
     $j < \text{dim}_c (\mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds))))$   
  **then have**  $ij\text{-le2}: i < \text{dim}_r (\text{matricize } I (\text{tensor0 } ds))$   $j < \text{dim}_c (\text{matricize } I (\text{tensor0 } ds))$   
  **unfolding** *mat-zero-def mat-dim-row-mat* **by** (*simp-all add: dims-matricize*)  
  **show**  $\text{matricize } I (\text{tensor0 } ds) \text{ \textit{\$} \textit{\$} } (i, j) = \mathbf{0}_m (\text{dim}_r (\text{matricize } I (\text{tensor0 } ds))) (\text{dim}_c (\text{matricize } I (\text{tensor0 } ds))) \text{ \textit{\$} \textit{\$} } (i, j)$   
  **unfolding** *mat-zero-def mat-index-mat*[*OF ij-le2*] **unfolding** *matricize-def*

```

mat-index-mat[OF ij-le2[unfolding 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

```

## 17 Submatrices

```

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

```

## 18 Submatrix

**definition** *submatrix* :: 'a mat  $\Rightarrow$  nat set  $\Rightarrow$  nat set  $\Rightarrow$  'a mat **where**  
*submatrix* A I J = mat (card {i. i < dim<sub>r</sub> A  $\wedge$  i  $\in$  I}) (card {j. j < dim<sub>c</sub> A  $\wedge$  j  $\in$  J})  
( $\lambda$ (i,j). A \$\$ (pick I i, pick J j))

**lemma** *dim-submatrix*: dim<sub>r</sub> (submatrix A I J) = card {i. i < dim<sub>r</sub> A  $\wedge$  i  $\in$  I}  
dim<sub>c</sub> (submatrix A I J) = card {j. j < dim<sub>c</sub> A  $\wedge$  j  $\in$  J}  
**unfolding** *submatrix-def* **by** *simp-all*

**lemma** *submatrix-index*:  
**assumes** i < card {i. i < dim<sub>r</sub> A  $\wedge$  i  $\in$  I}  
**assumes** j < card {j. j < dim<sub>c</sub> A  $\wedge$  j  $\in$  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  $\wedge$  a  $\in$  I} = {a  $\in$  I. a < n} **by** *meson*

**lemma** *submatrix-index-card*:  
**assumes** i < dim<sub>r</sub> A j < dim<sub>c</sub> A i  $\in$  I j  $\in$  J  
**shows** submatrix A I J \$\$ (card {a  $\in$  I. a < i}, card {a  $\in$  J. a < j}) = A \$\$ (i, j)  
**proof** –  
**have** i = pick I (card {a  $\in$  I. a < i})  
j = pick J (card {a  $\in$  J. a < j}) **using** *pick-card-in-set assms* **by** *auto*  
**have** {a  $\in$  I. a < i}  $\subset$  {i. i < dim<sub>r</sub> A  $\wedge$  i  $\in$  I}  
{a  $\in$  J. a < j}  $\subset$  {j. j < dim<sub>c</sub> A  $\wedge$  j  $\in$  J}  
**unfolding** *set-le-in* **using**  $\langle$ i < dim<sub>r</sub> A $\rangle$   $\langle$ j < dim<sub>c</sub> A $\rangle$  *Collect-mono less-imp-le*  
*less-le-trans*  $\langle$ i  $\in$  I $\rangle$   $\langle$ j  $\in$  J $\rangle$  **by** *auto*  
**then have** card {a  $\in$  I. a < i} < card {i. i < dim<sub>r</sub> A  $\wedge$  i  $\in$  I}  
card {a  $\in$  J. a < j} < card {j. j < dim<sub>c</sub> A  $\wedge$  j  $\in$  J} **by** (*simp-all add:*  
*psubset-card-mono*)  
**then show** *?thesis*  
**using**  $\langle$ i = pick I (card {a  $\in$  I. a < i}) $\rangle$   $\langle$ j = pick J (card {a  $\in$  J. a < j}) $\rangle$   
*submatrix-index* **by** *fastforce*

qed

```
lemma submatrix-split: submatrix A I J = submatrix (submatrix A UNIV J) I
UNIV
proof (rule mat-eqI)
  show  $\dim_r$  (submatrix A I J) =  $\dim_r$  (submatrix (submatrix A UNIV J) I
UNIV)
  by (simp add: dim-submatrix(1))
  show  $\dim_c$  (submatrix A I J) =  $\dim_c$  (submatrix (submatrix A UNIV J) I UNIV)
  by (simp add: dim-submatrix(2))
  fix i j assume ij-le:i <  $\dim_r$  (submatrix (submatrix A UNIV J) I UNIV) j <
 $\dim_c$  (submatrix (submatrix A UNIV J) I UNIV)
  then have ij-le1:i < card {i. i <  $\dim_r$  A  $\wedge$  i  $\in$  I} j < card {i. i <  $\dim_c$  A  $\wedge$  i  $\in$ 
J}
  by (simp-all add: dim-submatrix)
  then have ij-le2:i < card {i. i <  $\dim_r$  (submatrix A UNIV J)  $\wedge$  i  $\in$  I} j < card
{i. i <  $\dim_c$  (submatrix A UNIV J)  $\wedge$  i  $\in$  UNIV}
  by (simp-all add: dim-submatrix)
  then have i-le3:pick I i < card {i. i <  $\dim_r$  A  $\wedge$  i  $\in$  UNIV}
  using ij-le1(1) pick-le by auto
  have j-le3: pick UNIV j < card {i. i <  $\dim_c$  A  $\wedge$  i  $\in$  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
```

## 19 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 ( $\dim_r$  A) A
```

```
no-notation normal-rel (infixl  $\triangleleft$  60)
```

```
lemma lookup-order1-prod:
assumes  $\bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1$ 
assumes is  $\triangleleft$  dims (prod-list Bs)
shows lookup (prod-list Bs) is = prod-list (map ( $\lambda(i,B). \text{lookup } B [i]$ ) (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.prem(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  $\bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1$  a · prod-list Bs
= A
  using cprank-max1-prod-listE assms by metis
  def row-factor ==  $\lambda ris. a * (\text{prod-list } (\text{map } (\lambda(i,B). \text{lookup } B [i]) (\text{zip } ris (\text{sublist } Bs I))))$ 
  def col-factor ==  $\lambda cis. (\text{prod-list } (\text{map } (\lambda(i,B). \text{lookup } B [i]) (\text{zip } cis (\text{sublist } Bs (-I)))))$ 
  have  $\bigwedge is. is < \text{dims } A \implies \text{lookup } A is = \text{row-factor } (\text{sublist } is I) * \text{col-factor } (\text{sublist } is (-I))$ 
  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  $\bigwedge B. B \in \text{set } Bs \implies \text{Tensor.order } B = 1$ ]
lookup-smult
      using <a · prod-list Bs = A> dims-smult by fastforce
    also have ... = a * (prod-list (map (λ(i,B). lookup B [i]) (sublist (zip is Bs)
I))) *
      (prod-list (map (λ(i,B). lookup B [i]) (sublist (zip is Bs)
(-I))))

```

```

    using prod-list-complementary-sublists by auto
    also have ... = row-factor (sublist is I) * col-factor (sublist is (-I))
    using sublist-zip row-factor-def col-factor-def by metis
    finally show lookup A is = row-factor (sublist is I) * col-factor (sublist is
(-I)) .
  qed
  def row-factor' == λr. row-factor (digit-encode (sublist (Tensor.dims A) I) r)
  def col-factor' == λc. col-factor (digit-encode (sublist (Tensor.dims A) (-I)) c)
  have ∧r c. r < dimr (matricize I A) ⇒ c < dimc (matricize I A) ⇒ matricize
I A $$ (r,c) = row-factor' r * col-factor' c
  proof -
    fix r c assume r < dimr (matricize I A) c < dimc (matricize I A)
    then have matricize I A $$ (r,c) = Tensor.lookup A (weave I
      (digit-encode (sublist (Tensor.dims A) I) r)
      (digit-encode (sublist (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 (sublist is I) *
col-factor (sublist is (-I))⟩
      valid-index-weave[OF
        digit-encode-valid-index[OF ⟨r < dimr (matricize I A)⟩[unfolding dims-matricize]]
        digit-encode-valid-index[OF ⟨c < dimc (matricize I A)⟩[unfolding 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) = 0m (dimr (matricize I (tensor0 ds))) (dimc
(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  $mrank (matricize I (A + B)) \leq mrank (matricize I A) + mrank (matricize I B)$ 
using matricize-add vec-space.rank-subadditive dims-matricize
mat-carrierI mat-index-add(2) ⟨dims A = dims B⟩ by metis
then show  $mrank (matricize I (A + B)) \leq Suc j$ 
using  $\langle mrank (matricize I A) \leq 1 \rangle \langle mrank (matricize I B) \leq j \rangle$  by linarith
qed

```

```

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

end

```

## 20 Missing Lemmas of Matrix

```

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

```

```

lemma dim-vec-of-list[simp] :  $dim_v (vec-of-list as) = length as$  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:  $i < length xs \implies (vec-of-list xs) \$ i = xs ! i$ 
by (metis vec.abs-eq vec-index-vec vec-of-list.abs-eq)

```

```

lemma nth-list-of-vec:  $i < dim_v v \implies (list-of-vec v) ! i = v \$ i$ 
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)

```

```

definition component-mult  $v w = vec (min (dim_v v) (dim_v w)) (\lambda i. v \$ i * w \$ i)$ 

```

```

definition vec-set:: $'a vec \Rightarrow 'a set (set_v)$ 
where  $vec-set v = vec-index v \setminus \{.. < dim_v v\}$ 

```

```

lemma index-component-mult:

```

**assumes**  $i < \dim_v v$   $i < \dim_v w$   
**shows**  $\text{component-mult } v \ w \ \$ \ i = v \ \$ \ i * w \ \$ \ i$   
**unfolding**  $\text{component-mult-def}$  **using**  $\text{assms vec-index-vec}$  **by**  $\text{auto}$

**lemma**  $\text{dim-component-mult}$ :  
 $\dim_v (\text{component-mult } v \ w) = \min (\dim_v v) (\dim_v w)$   
**unfolding**  $\text{component-mult-def}$  **using**  $\text{vec-index-vec}$  **by**  $\text{auto}$

**lemma**  $\text{vec-setE}$ :  
**assumes**  $a \in \text{set}_v v$   
**obtains**  $i$  **where**  $v \$ i = a$   $i < \dim_v v$  **using**  $\text{assms}$  **unfolding**  $\text{vec-set-def}$  **by**  $\text{blast}$

**lemma**  $\text{vec-setI}$ :  
**assumes**  $v \$ i = a$   $i < \dim_v v$   
**shows**  $a \in \text{set}_v v$  **using**  $\text{assms}$  **unfolding**  $\text{vec-set-def}$  **using**  $\text{image-eqI lessThan-iff}$   
**by**  $\text{blast}$

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

**lemma**  $\text{length-list-of-vec[simp]}$  :  $\text{length } (\text{list-of-vec } v) = \dim_v v$  **by**  $\text{transfer auto}$

**end**

## 21 Matrix to Vector Conversion

**theory**  $\text{DL-Flatten-Matrix}$   
**imports**  $\text{Real .. / Jordan-Normal-Form / Matrix}$   
**begin**

**definition**  $\text{extract-matrix}$  ::  $(\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ mat}$  **where**  
 $\text{extract-matrix } a \ m \ n = \text{mat } m \ n \ (\lambda(i,j). a (i*n + j))$

**definition**  $\text{flatten-matrix}$  ::  $'a \text{ mat} \Rightarrow (\text{nat} \Rightarrow 'a)$  **where**  
 $\text{flatten-matrix } A \ k = A \ \$\$ (k \ \text{div} \ \dim_c A, k \ \text{mod} \ \dim_c A)$

**lemma**  $\text{two-digit-le}$ :  
**fixes**  $i \ j :: \text{nat}$   
**assumes**  $i < m$   $j < n$   
**shows**  $i*n + j < m*n$   
**proof** –  
**have**  $(i*n + j) \ \text{div} \ n = i \ m*n \ \text{div} \ n = m$  **using**  $\text{assms}$  **by**  $\text{auto}$   
**then have**  $(i * n + j) \ \text{div} \ m \ \text{div} \ n = i \ \text{div} \ m$   
**by**  $(\text{metis } \text{Divides.div-mult2-eq } \text{mult.commute})$   
**then show**  $?thesis$   
**by**  $(\text{metis } \text{Divides.div-mult2-eq } (i < m) (m * n \ \text{div} \ n = m) \ \text{div-eq-0-iff not-less0})$   
**qed**

**lemma**  $\text{extract-matrix-cong}$ :

**assumes**  $\bigwedge i. i < m * n \implies a\ i = b\ i$   
**shows**  $extract\text{-}matrix\ a\ m\ n = extract\text{-}matrix\ b\ m\ n$   
**proof** –  
**have**  $\bigwedge i\ j. i < m \implies j < n \implies a\ (i*n + j) = b\ (i*n + j)$  **using** *two-digit-le*  
*assms* **by** *blast*  
**then show** *?thesis* **unfolding** *extract-matrix-def* **by** *auto*  
**qed**

**lemma** *extract-matrix-flatten-matrix*:  
 $extract\text{-}matrix\ (flatten\text{-}matrix\ A)\ (dim_r\ A)\ (dim_c\ A) = A$   
**unfolding** *extract-matrix-def* *flatten-matrix-def* **by** *auto*

**lemma** *flatten-matrix-extract-matrix*:  
**shows**  $\bigwedge k. k < m*n \implies flatten\text{-}matrix\ (extract\text{-}matrix\ a\ m\ n)\ k = a\ k$   
**unfolding** *extract-matrix-def* *flatten-matrix-def*  
**by** (*metis* (*no-types*, *lifting*) *Divides.div-mult2-eq* *case-prod-conv* *div-eq-0-iff* *mat-dim-col-mat(1)*)

*mat-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**  $extract\text{-}matrix\ a\ m\ n\ \$\$ (i,j) = a\ (i*n + j)$   
**unfolding** *extract-matrix-def* **using** *assms* **by** *simp*

**lemma** *dim-extract-matrix*:  
**shows**  $dim_r\ (extract\text{-}matrix\ as\ m\ n) = m$   
**and**  $dim_c\ (extract\text{-}matrix\ as\ m\ n) = n$   
**unfolding** *extract-matrix-def* **by** *simp-all*

end

## 22 Deep Learning Networks

**theory** *DL-Network*  
**imports** *Tensor-Product* *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**  $\otimes_1$  70)

**datatype**  $'a\ convnet = Input\ nat \mid Conv\ 'a\ 'a\ convnet \mid Pool\ 'a\ convnet\ 'a\ convnet$

**fun** *input-sizes*  $:: 'a\ convnet \Rightarrow nat\ list$  **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 (λi. w (i+r0\*r1))) |  
*insert-weights* (Pool m1 m2) w = Pool  
(insert-weights m1 w)  
(insert-weights m2 (λi. 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<sub>r</sub> A, dim<sub>c</sub> A) (*remove-weights* m) |  
*remove-weights* (Pool m1 m2) = Pool (*remove-weights* m1) (*remove-weights* m2)

**abbreviation** *output-size'* == (λm. *output-size* (*remove-weights* m))  
**abbreviation** *valid-net'* == (λm. *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 ⊗<sub>mv</sub> *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<sub>r</sub> A) (λj. tensor-from-lookup ds (λis. (A ⊗<sub>mv</sub> (map<sub>v</sub> (λT. Tensor.lookup T is) Ts)) \$j))

**lemma** *insert-weights-cong*:

```

assumes ( $\bigwedge i. i < \text{count-weights } m \implies w1\ i = w2\ i$ )
shows  $\text{insert-weights } m\ w1 = \text{insert-weights } m\ w2$ 
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  $2:\text{insert-weights } m\ (\lambda i. w1\ (i + r0 * r1)) = \text{insert-weights } m\ (\lambda i. w2\ (i + r0 * r1))$  using Conv
  using  $\langle r01 = (r0, r1) \rangle$  add.commute add-less-cancel-right count-weights.simps(2)
by fastforce
  then show ?case unfolding  $\langle r01 = (r0, r1) \rangle$  insert-weights.simps
  by (metis Conv.premis  $\langle r01 = (r0, r1) \rangle$  count-weights.simps(2) extract-matrix-cong trans-less-add1)
next
  case (Pool m1 m2)
  have  $1:\text{insert-weights } m1\ w1 = \text{insert-weights } m1\ w2$ 
  using Pool(1)[of w1 w2] Pool(3)[unfolded count-weights.simps] by simp
  have  $2:\text{insert-weights } m2\ (\lambda i. w1\ (i + \text{count-weights } m1)) = \text{insert-weights } m2\ (\lambda i. w2\ (i + \text{count-weights } m1))$ 
  using Pool(2)[of  $\lambda i. w1\ (i + \text{count-weights } m1)$   $\lambda i. w2\ (i + \text{count-weights } m1)$ ] 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  $T \in \text{set}_v\ (\text{mat-tensorlist-mult } A\ Ts\ ds)$ 
shows  $\text{Tensor.dims } T = ds$ 
proof –
  obtain j where  $T = \text{tensor-from-lookup } ds\ (\lambda is. (A \otimes_{mv}\ (\text{map}_v\ (\lambda T. \text{Tensor.lookup } T\ is)\ Ts)))\ \$j$ 
  using vec-setE[OF assms, unfolded mat-tensorlist-mult-def] by (metis vec-dim-vec 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  $\Rightarrow$  real tensor vec where
tensors-from-net (Input M) = Matrix.vec M  $(\lambda i. \text{unit-vec } M\ i)$  |
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  $\text{output-size}'\ m = \text{dim}_v\ (\text{tensors-from-net } m)$ 
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<sub>v</sub> inputs = input-sizes m

**shows** output-size' m = dim<sub>v</sub> (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-mat-mult-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<sub>v</sub> (take (length (input-sizes m1)) inputs) = input-sizes
m1

map dim<sub>v</sub> (drop (length (input-sizes m1)) inputs) = input-sizes m2

**using** Pool.prem(2) **by** (metis append-eq-conv-conj drop-map input-sizes.simps(3)
take-map)+

**ultimately have**

output-size' m1 = dim<sub>v</sub> (evaluate-net m1 (take (length (input-sizes m1)) in-
puts))

output-size' m2 = dim<sub>v</sub> (evaluate-net m2 (drop (length (input-sizes m1)) in-
puts))

**using** Pool.IH **by** blast+

**then show** ?case **unfolding** evaluate-net.simps remove-weights.simps output-size.simps

**by** (metis Pool.prem(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*:  $input\text{-}sizes\ m = input\text{-}sizes\ (remove\text{-}weights\ m)$

**by** (*induction m*; *simp*)

**lemma** *dims-tensors-from-net*:

**assumes**  $T \in set_v\ (tensors\text{-}from\text{-}net\ m)$

**shows**  $Tensor.dims\ T = input\text{-}sizes\ m$

**using** *assms* **proof** (*induction m arbitrary:T*)

**case** (*Input M*)

**then obtain**  $j$  **where**  $T = unit\text{-}vec\ M\ j$

**using** *vec-setE tensors-from-net.simps(1)* **by** (*metis vec-dim-vec 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\_v (tensors-from-net m1) i < dim\_v (tensors-from-net m2)*

**using** *tensors-from-net.simps vec-setE dim-component-mult* **by** (*metis min.strict-boundedE*)

**then obtain**  $T1\ T2$  **where**  $T = T1 \otimes T2$   $T1 \in set_v\ (tensors\text{-}from\text{-}net\ m1)$   $T2 \in set_v\ (tensors\text{-}from\text{-}net\ m2)$

**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**

**definition** *base-input* ::  $real\ mat\ convnet \Rightarrow nat\ list \Rightarrow real\ vec\ list$  **where**

*base-input m is = (map (\(n, i). unit\_v n i) (zip (input-sizes m) is))*

**lemma** *base-input-length*:

**assumes**  $is \triangleleft input\text{-}sizes\ m$

**shows**  $input\text{-}sizes\ m = map\ dim_v\ (base\text{-}input\ m\ is)$

**proof** (*rule nth-equalityI*)

**have**  $length\ (input\text{-}sizes\ m) = length\ is$  **using** *assms valid-index-length* **by** *auto*

**then show**  $length\ (input\text{-}sizes\ m) = length\ (map\ dim_v\ (base\text{-}input\ m\ is))$  **unfolding** *base-input-def* **by** *auto*

{

**fix**  $i$

**assume**  $i < length\ (input\text{-}sizes\ m)$

**then have**  $map\ (\lambda(n, i). unit_v\ n\ i)\ (zip\ (input\text{-}sizes\ m)\ is) ! i = unit_v\ (input\text{-}sizes\ m ! i)\ (is ! i)$

**using**  $\langle length\ (input\text{-}sizes\ m) = length\ is \rangle$  **by** *auto*

**then have**  $input\text{-}sizes\ m ! i = map\ dim_v\ (base\text{-}input\ m\ is) ! i$  **unfolding** *base-input-def* **using** *vec-unit-index(3)*

**using**  $\langle i < length\ (input\text{-}sizes\ m) \rangle \langle length\ (input\text{-}sizes\ m) = length\ (map\ dim_v$

```

(base-input m is))
  base-input-def assms length-map nth-map valid-index-lt by (simp add:
input-sizes-remove-weights)
}
then show  $\forall i < \text{length } (\text{input-sizes } m). \text{input-sizes } m ! i = \text{map } \text{dim}_v (\text{base-input }
m \text{ is}) ! i$  by auto
qed

```

**lemma** *nth-mat-tensorlist-mult*:

**assumes**  $\bigwedge A. A \in \text{set}_v \text{ Ts} \implies \text{dims } A = ds$

**assumes**  $i < \text{dim}_r A$

**assumes**  $\text{dim}_v \text{ Ts} = \text{dim}_c A$

**shows** *mat-tensorlist-mult*  $A \text{ Ts } ds \ \$ i = \text{listsum } ds (\text{map } (\lambda j. (A \ \$\$ (i,j)) \cdot \text{Ts } \$ j) [0..<\text{dim}_v \text{ Ts}])$

(**is**  $- = \text{listsum } ds \ ?\text{Ts}'$ )

**proof** (*rule tensor-lookup-eqI*)

**have**  $\text{dims-Ts}' : \bigwedge T. T \in \text{set } ?\text{Ts}' \implies \text{dims } T = ds$

**proof** –

**fix**  $T$  **assume**  $T \in \text{set } ?\text{Ts}'$

**then obtain**  $k$  **where**  $T = ?\text{Ts}' ! k$  **and**  $k < \text{length } ?\text{Ts}'$   $k < \text{dim}_v \text{ Ts}$  **using** *in-set-conv-nth* **by force**

**show**  $\text{dims } T = ds$  **unfolding**  $\langle T = ?\text{Ts}' ! k \rangle$  *nth-map[OF  $\langle k < \text{length } ?\text{Ts}' \rangle$ ]* *unfolding length-map*

**using** *assms(1)*  $\langle k < \text{dim}_v \text{ Ts} \rangle$

**by** (*simp add:  $\langle k < \text{length } (\text{map } (\lambda j. A \ \$\$ (i, j) \cdot \text{Ts } \$ j) [0..<\text{dim}_v \text{ Ts}])$* ) *vec-setI*)

**qed**

**then show** *dims-eq*:  $\text{dims } (\text{mat-tensorlist-mult } A \text{ Ts } ds \ \$ i) = \text{dims } (\text{Tensor-Plus.listsum } ds (\text{map } (\lambda j. A \ \$\$ (i, j) \cdot \text{Ts } \$ j) [0..<\text{dim}_v \text{ Ts}]))$

**using** *dims-mat-tensorlist-mult assms mat-tensorlist-mult-def listsum-dims*

**by** (*metis (no-types, lifting) vec-dim-vec vec-setI*)

**fix**  $is$  **assume** *is-valid*:  $is \triangleleft \text{dims } (\text{mat-tensorlist-mult } A \text{ Ts } ds \ \$ i)$

**then have**  $is \triangleleft ds$  **using** *dims-eq dims-Ts' listsum-dims* **by** (*metis (no-types, lifting)*)

**have** *summand-eq*:  $\bigwedge j. j \in \{0 ..<\text{dim}_v \text{ Ts}\} \implies \text{row } A \ i \ \$ j * (\text{map}_v (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}) \ \$ j = \text{lookup } (A \ \$\$ (i, j) \cdot \text{Ts } \$ j) \ is$

**using** *vec-index-vec  $\langle i < \text{dim}_r A \rangle$  row-def  $\langle \text{dim}_v \text{ Ts} = \text{dim}_c A \rangle$*

$\langle is \triangleleft ds \rangle$  *assms(1) lookup-smult atLeastLessThan-iff vec-index-map(1) vec-setI*

**by** *metis*

**have** *lookup*  $(\text{mat-tensorlist-mult } A \text{ Ts } ds \ \$ i) \ is = (A \otimes_{m_v} (\text{map}_v (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts})) \ \$ i$

**unfolding** *mat-tensorlist-mult-def* **using** *lookup-tensor-from-lookup[OF  $\langle is \triangleleft ds \rangle$ ]* **using**  $\langle i < \text{dim}_r A \rangle$  **by auto**

**also have**  $\dots = \text{row } A \ i \cdot \text{map}_v (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}$

**using**  $\langle i < \text{dim}_r A \rangle$  **by simp**

**also have**  $\dots = (\sum j \in \{0 ..<\text{dim}_v \text{ Ts}\}. \text{row } A \ i \ \$ j * (\text{map}_v (\lambda T. \text{Tensor.lookup } T \ is) \ \text{Ts}))$

$T \text{ is } Ts) \$ j)$   
**unfolding** *scalar-prod-def rows-nth*[*OF*  $\langle i < \dim_r A \rangle$ ] **by** *simp*  
**also have**  $\dots = (\sum j \in \{0..< \dim_v Ts\}. \text{lookup } (A \$\$ (i, j) \cdot Ts \$ j) \text{ is})$  **using**  
*summand-eq* **by** *force*  
**also have**  $\dots = (\sum A \leftarrow ?Ts'. \text{lookup } A \text{ is})$  **unfolding** *map-map*  
*Groups-List.sum-set-upt-conv-sum-list-nat*[*symmetric*] *atLeastLessThan-upt*[*symmetric*]  
**by** *auto*  
**also have**  $\dots = \text{lookup } (\text{listsum } ds \ ?Ts') \text{ is}$  **using** *lookup-listsum*[*OF*  $\langle is \triangleleft ds \rangle$ ]  
*dims-Ts'* **by** *fastforce*  
**finally show**  $\text{lookup } (\text{mat-tensorlist-mult } A \ Ts \ ds \ \$ \ i) \text{ is} = \text{lookup } (\text{listsum } ds$   
 $\ ?Ts') \text{ is}$  **by** *metis*  
**qed**

**lemma** *lookup-tensors-from-net*:

**assumes** *valid-net' m*  
**and**  $is \triangleleft \text{input-sizes } m$   
**and**  $j < \text{output-size}' m$   
**shows**  $\text{Tensor.lookup } (\text{tensors-from-net } m \ \$ \ j) \text{ is} = \text{evaluate-net } m \ (\text{base-input } m$   
 $\text{ is}) \ \$ \ j$   
**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**  $1:\text{tensors-from-net } (\text{Input } M) \ \$ \ j = \text{unit-vec } M \ j$  **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**  $2:\text{Tensor.lookup } (\text{tensors-from-net } (\text{Input } M) \ \$ \ j) \text{ is} = (\text{if } i=j \text{ then}$   
 $1 \text{ else } 0)$  **using** *lookup-unit-vec 1* **by** *metis*  
**have**  $\text{evaluate-net } (\text{Input } M) \ (\text{map } (\lambda(n, i). \text{unit}_v \ n \ i) \ (\text{zip } (\text{input-sizes } (\text{Input}$   
 $M)) \text{ is})) = \text{unit}_v \ M \ i$  **using**  $\langle is = [i] \rangle$  **by** *auto*  
**then show**  $?case$  **using** 2  $\langle j < M \rangle$  *base-input-def* **by** (*simp add:*  $\langle i < M \rangle$ )  
**next**  
**case** (*Conv A m j is*)  
**have**  $is\text{-valid:}is \triangleleft \text{input-sizes } m$  **using** *Conv.prem*s **by** *simp*  
**have** *valid-net:valid-net'*  $m$  **using** *Conv.prem*s(1) **unfolding** *remove-weights.simps*  
  
**using** *valid-net.simps convnet.distinct*(1) *convnet.distinct*(5) *convnet.inject*(2)  
**by** *blast*  
**then have** *length-em*:  $\dim_v (\text{evaluate-net } m \ (\text{base-input } m \text{ is})) = \text{output-size}' m$   
**using** *output-size-correct base-input-length is-valid* **by** *metis*  
  
**have**  $IH':\text{map}_v (\lambda T. \text{Tensor.lookup } T \text{ is}) (\text{tensors-from-net } m) =$   
 $\text{evaluate-net } m \ (\text{base-input } m \text{ is})$   
**proof** (*rule vec-eqI*)  
**show** *equal-lengths*:  $\dim_v (\text{map}_v (\lambda T. \text{lookup } T \text{ is}) (\text{tensors-from-net } m))$   
 $= \dim_v (\text{evaluate-net } m \ (\text{base-input } m \text{ is}))$  **using** *length-em*  
**by** (*simp add: output-size-correct-tensors valid-net*)  
**show**  $\bigwedge i. i < \dim_v (\text{evaluate-net } m \ (\text{base-input } m \text{ is})) \implies$   
 $\text{map}_v (\lambda T. \text{lookup } T \text{ is}) (\text{tensors-from-net } m) \ \$ \ i = \text{evaluate-net } m \ (\text{base-input}$   
 $m \text{ is}) \ \$ \ i$

```

proof –
  fix  $i$ 
  assume  $i < \dim_v (\text{evaluate-net } m \text{ (base-input } m \text{ is)})$ 
  then have  $i < \text{output-size}' m$  using equal-lengths length-em by auto
  then show  $\text{map}_v (\lambda T. \text{lookup } T \text{ is}) (\text{tensors-from-net } m) \$ i$ 
    =  $\text{evaluate-net } m \text{ (base-input } m \text{ is)} \$ i$ 
    using Conv.IH is-valid equal-lengths valid-net base-input-def length-em
nth-map-upt
   $\text{length-map } \text{nth-map}$  by auto
  qed
qed

have  $\text{Tensor.lookup} ((\text{tensors-from-net } (\text{Conv } A \ m)) \$ j) \text{ is} =$ 
   $(A \otimes_{m_v} (\text{map}_v (\lambda T. \text{Tensor.lookup } T \text{ is}) (\text{tensors-from-net } m))) \$ j$ 
proof –
  have  $\dim_v (\text{tensors-from-net } (\text{Conv } A \ m)) = \text{output-size}' (\text{Conv } A \ m)$ 
  using Conv by (simp add: mat-tensorlist-mult-def)
  then have  $j < \dim_v (\text{tensors-from-net } (\text{Conv } A \ m))$  using Conv.premis by auto
  then have  $(\text{tensors-from-net } (\text{Conv } A \ m)) \$ j = \text{tensor-from-lookup} (\text{input-sizes}$ 
 $m)$ 
     $(\lambda \text{is}. (A \otimes_{m_v} (\text{map}_v (\lambda T. \text{Tensor.lookup } T \text{ is}) (\text{tensors-from-net } m))))$ 
 $\$ j)$ 
  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  $(A \otimes_{m_v} (\text{map}_v (\lambda T. \text{Tensor.lookup } T \text{ is}) (\text{tensors-from-net } m))) \$ j$ 
  =  $(A \otimes_{m_v} (\text{evaluate-net } m \text{ (base-input } m \text{ is)})) \$ j$  using IH' by auto
also have  $\dots = \text{evaluate-net } (\text{Conv } A \ m) \text{ (base-input } (\text{Conv } A \ m) \text{ is)} \$ j$ 
  unfolding base-input-def using evaluate-net.simps by auto
finally show ?case by auto
next
case  $(\text{Pool } m1 \ m2 \ j \ \text{is})$ 

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

  obtain  $\text{is1 } \text{is2}$  where  $\text{is12-def:is} = \text{is1} \ @ \ \text{is2}$   $\text{is1} \triangleleft \text{input-sizes } m1$   $\text{is2} \triangleleft$ 
 $\text{input-sizes } m2$ 
  by  $(\text{metis } \text{Pool.premis}(2) \ \text{input-sizes.simps}(3) \ \text{valid-index-split})$ 

  Apply the induction hypothesis to the subnets:

  have  $\text{IH:Tensor.lookup} (\text{tensors-from-net } m1 \ \$ j) \ \text{is1}$ 
  =  $\text{evaluate-net } m1 \ (\text{map } (\lambda(x, y). \text{unit}_v \ x \ y) \ (\text{zip } (\text{input-sizes } m1) \ \text{is1})) \$ j$ 
   $\text{Tensor.lookup} (\text{tensors-from-net } m2 \ \$ j) \ \text{is2}$ 
  =  $\text{evaluate-net } m2 \ (\text{map } (\lambda(x, y). \text{unit}_v \ x \ y) \ (\text{zip } (\text{input-sizes } m2) \ \text{is2})) \$ j$ 
  using Pool convnet.distinct(3) convnet.distinct(5) convnet.inject(3) remove-weights.simps(3)

   $\text{valid-net.simps} \ \langle \text{is1} \triangleleft \text{input-sizes } m1 \rangle \ \langle \text{is2} \triangleleft \text{input-sizes } m2 \rangle \ \text{output-size.simps}(3)$ 

  by  $(\text{metis } \text{base-input-def})+$ 

```

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 < dimv (tensors-from-net m1) j < dimv (tensors-from-net
m2)
  by (metis Pool.premis(1) Pool.premis(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.premis
  by (metis convnet.distinct(3) convnet.distinct(5) convnet.inject(3))+
  have input-sizes m1 = map dimv (base-input m1 is1)
    input-sizes m2 = map dimv (base-input m2 is2)
  using base-input-def base-input-length base-input-def is12-def by auto
  have j < dimv (evaluate-net m1 (base-input m1 is1)) j < dimv (evaluate-net
m2 (base-input m2 is2))
  using Pool.premis ⟨input-sizes m1 = map dimv (base-input m1 is1)⟩ ⟨valid-net'
m1⟩
    output-size-correct by (auto,metis Pool.premis(1) Pool.premis(3) ⟨input-sizes
m2 = map dimv (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**  $m = \text{insert-weights } (\text{remove-weights } m)$   $w$   
**proof** (*induction m arbitrary:thesis*)  
  **case** (*Input m thesis*)  
  **then show** ?case **by** *simp*  
**next**  
  **case** (*Conv A m thesis*)  
  **then obtain**  $w$  **where**  $m = \text{insert-weights } (\text{remove-weights } m)$   $w$  **by** *auto*  
  **then have**  $1:\text{remove-weights } (\text{Conv } A \ m) = \text{Conv } (\text{dim}_r \ A, \ \text{dim}_c \ A) (\text{remove-weights } m)$  **by** *simp*  
  **have**  $\text{Conv } A \ m = \text{insert-weights } (\text{remove-weights } (\text{Conv } A \ m))$  ( $\lambda i. \ \text{if } i < \text{dim}_r \ A \ * \ \text{dim}_c \ A \ \text{then } \text{flatten-matrix } A \ i \ \text{else } w \ (i - \text{dim}_r \ A \ * \ \text{dim}_c \ A)$ )  
  **unfolding** *1 insert-weights.simps*  
  **using** *extract-matrix-flatten-matrix[of A] extract-matrix-cong[of dim\_r A dim\_c A]*  
   $\lambda i. \ \text{if } i < \text{dim}_r \ A \ * \ \text{dim}_c \ A \ \text{then } \text{flatten-matrix } A \ i \ \text{else } w \ (i - \text{dim}_r \ A \ * \ \text{dim}_c \ A) \ \text{flatten-matrix } A]$   
  **using**  $\langle m = \text{insert-weights } (\text{remove-weights } m) \ w \rangle$  **by** *fastforce*  
  **then show** ?case **using** *Conv.prem*s **by** *blast*  
**next**  
  **case** (*Pool m1 m2*)  
  **then obtain**  $w1 \ w2$  **where**  $m1 = \text{insert-weights } (\text{remove-weights } m1)$   $w1$   $m2 = \text{insert-weights } (\text{remove-weights } m2)$   $w2$  **by** *metis*  
  **then have**  $\text{Pool } m1 \ m2 = \text{insert-weights } (\text{remove-weights } (\text{Pool } m1 \ m2))$  ( $\lambda i. \ \text{if } i < \text{count-weights } (\text{remove-weights } m1) \ \text{then } w1 \ i \ \text{else } w2 \ (i - \text{count-weights } (\text{remove-weights } m1)))$ )  
  **unfolding** *remove-weights.simps insert-weights.simps*  
  **using** *insert-weights-cong[of - \lambda i. if i < count-weights (remove-weights m1) w1] by fastforce*  
  **then show** ?case **unfolding** *Pool* **using** *Pool.prem*s **by** *blast*  
**qed**

**lemma** *remove-insert-weights*:  
**shows**  $\text{remove-weights } (\text{insert-weights } m \ w) = m$   
**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**  $\text{remove-weights } (\text{insert-weights } m \ w) = m$  **using** *Conv.IH* **by** *blast*  
  **then have**  $\text{remove-weights } (\text{insert-weights } (\text{Conv } (r1, r2) \ m) \ w) = \text{Conv } (r1, r2) \ m$   
  **unfolding** *insert-weights.simps remove-weights.simps*

```

    using extract-matrix-def Conv.IH dim-extract-matrix(1) by (metis mat-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  $\lambda is1$ 
 $is2. f (is1 @ is2)$ ]
  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  $g :: nat \Rightarrow real$  shows  $prod\ g\ \{.. $n+m$ \} = prod\ g\ \{.. $n$ \} * prod\ (\lambda 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  $map\ dim_v\ inputs = input-sizes\ m$ 
and  $j < output-size'\ m$ 
shows  $evaluate-net\ m\ inputs\ \$\ j$ 
  =  $(\sum is \in \{is.\ is \triangleleft input-sizes\ m\}. (\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is!k)) * Tensor.lookup\ (tensors-from-net\ m\ \$\ j)\ is)$ 
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  $is \triangleleft input-sizes\ (Input\ M)$ 
    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.prem(3)  $\langle input-sizes\ (Input\ M) = [M] \rangle \langle is \triangleleft input-sizes\ (Input\ M) \rangle list.distinct(1)$ 
lookup-unit-vec nth-Cons-0 output-size.simps(1) remove-weights.simps(1) tensors-from-net.simps(1) valid-indexE vec-index-vec)
    then have  $(\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors-from-net\ (Input\ M)\ \$\ j)\ is =$ 
       $(if\ is=[j]\ then\ (\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k))\ else\ 0)$  using
 $\langle is = [hd\ is] \rangle$  by auto
  }
  then have  $(\sum is \mid is \triangleleft input-sizes\ (Input\ M). (\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k)) * lookup\ (tensors-from-net\ (Input\ M)\ \$\ j)\ is)$ 
    =  $(\sum is \mid is \triangleleft input-sizes\ (Input\ M). (if\ is=[j]\ then\ (\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k))\ else\ 0))$  by auto
  also have  $(\sum is \mid is \triangleleft input-sizes\ (Input\ M). (if\ is=[j]\ then\ (\prod k < length\ inputs.\ inputs\ !\ k\ \$\ (is\ !\ k))\ else\ 0))$ 
    =  $(\prod k < length\ inputs.\ inputs\ !\ k\ \$\ ([j]\ !\ k))$  unfolding sum.delta[OF finite-valid-index]

```

```

    using Input.premis(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-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 < dimr A using Conv.premis(3) by auto
have 0:  $\bigwedge is. is \triangleleft input\text{-}sizes (Conv A m) \implies$ 
  ( $\prod k < length\ inputs. inputs ! k \$ (is ! k)$ ) * lookup (tensors-from-net (Conv A m)
  $ j) is =
  ( $\sum i = 0..<dim_v (tensors\text{-}from\text{-}net\ m). row\ A\ j\ \$\ i * ((\prod k < length\ inputs. inputs$ 
  ! k $ (is ! k)) * lookup (tensors-from-net m $ i) is))
proof -
  fix is assume is < input-sizes (Conv A m)
  then have is < input-sizes m by simp
  have 0: lookup (tensors-from-net (Conv A m) $ j) is =
    ( $\sum i = 0..<dim_v (tensors\text{-}from\text{-}net\ m). row\ A\ j\ \$\ i * lookup (tensors\text{-}from\text{-}net$ 
    m $ i) is)
  unfolding tensors-from-net.simps mat-tensorlist-mult-def vec-index-vec[OF (j
  < dimr A)]
    lookup-tensor-from-lookup[OF (is < input-sizes m)] index-mat-mult-vec[OF (j
  < dimr A)] scalar-prod-def
  using vec-index-map by auto
  show ( $\prod k < length\ inputs. inputs ! k \$ (is ! k)$ ) * lookup (tensors-from-net
  (Conv A m) $ j) is
    = ( $\sum i = 0..<dim_v (tensors\text{-}from\text{-}net\ m). row\ A\ j\ \$\ i * ((\prod k < length\ inputs.$ 
    inputs ! k $ (is ! k)) * lookup (tensors-from-net m $ i) is))
  unfolding 0 sum-distrib-left by (simp add: semiring-normalization-rules(19))
qed
have valid-net' m by (metis Conv.premis(1) convnet.distinct(1) convnet.distinct(5)
convnet.inject(2) remove-weights.simps(2) valid-net.simps)
have map dimv inputs = input-sizes m by (simp add: Conv.premis(2))
have output-size' m = dimv (tensors-from-net m) by (simp add: (valid-net' m)
output-size-correct-tensors)
have 1:  $\bigwedge i. i < dim_v (tensors\text{-}from\text{-}net\ m) \implies (\sum is \mid is \triangleleft input\text{-}sizes (Conv A$ 
m). (( $\prod k < length\ inputs. inputs ! k \$ (is ! k)$ ) * lookup (tensors-from-net m $ i)
is)) = evaluate-net m inputs $ i unfolding input-sizes.simps
  using Conv.IH (valid-net' m) (map dimv inputs = input-sizes m) (output-size'
m = dimv (tensors-from-net m)) by simp

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

```

```

is < input-sizes (Conv A m). (( $\prod$  k < length inputs. inputs ! k $ (is ! k)) * lookup
(tensors-from-net m $ i) is)))
  by (simp add: sum-distrib-left)
  also have ... = ( $\sum$  i = 0..<dim_v (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_v inputs = input-sizes m⟩ ⟨output-size' m =
dim_v (tensors-from-net m)⟩
  ⟨valid-net' m⟩ output-size-correct scalar-prod-def)
  also have ... = (A  $\otimes_{m_v}$  evaluate-net m inputs) $ j by (simp add: ⟨j < dim_r 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.premis(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.premis(1) Pool.premis(3) convnet.distinct(3) convnet.inject(3)
convnet.simps(9)
  output-size.simps(3) remove-weights.simps(3) valid-net.simps) using Pool.premis
by auto
  then have j < dim_v (tensors-from-net m1) j < dim_v (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_v inputs1 = input-sizes m1 map dim_v inputs2 = input-sizes m2
  apply (metis Pool.premis(2) append-eq-conv-conj input-sizes.simps(3) inputs1-def
take-map)
  by (metis Pool.premis(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_v inputs1 = input-sizes m1⟩ valid-index-length
by fastforce
  have length is2 = length inputs2
  using ⟨is2 < input-sizes m2⟩ ⟨map dim_v inputs2 = input-sizes m2⟩ valid-index-length
by fastforce
  have 1:( $\prod$  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 (∏ k < length inputs. inputs ! k $ ((is1 @ is2) ! k)) = (∏ k < length inputs1.
inputs1 ! k $ (is1 ! k)) * (∏ 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 < dimv (tensors-from-net m1)⟩ ⟨j < dimv (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 < dimv (tensors-from-net
m1)⟩ ⟨j < dimv (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 < dimv (evaluate-net m1 (take (length (input-sizes m1)) inputs))

      j < dimv (evaluate-net m2 (drop (length (input-sizes m1)) inputs))
    using ⟨j < output-size' m1⟩ ⟨map dimv inputs1 = input-sizes m1⟩ ⟨valid-net'
m1⟩ inputs1-def output-size-correct
    using ⟨j < output-size' m2⟩ ⟨map dimv inputs2 = input-sizes m2⟩ ⟨valid-net'
m2⟩ inputs2-def by auto
    have (∑ is | is < input-sizes (Pool m1 m2). (∏ k < length inputs. inputs ! k $
(is ! k)) * lookup (tensors-from-net (Pool m1 m2) $ j) is)
      = (∑ is1 | is1 < input-sizes m1. ∑ is2 | is2 < input-sizes m2.
      (∏ k < length inputs1. inputs1 ! k $ (is1 ! k)) * (∏ k < length inputs2.
inputs2 ! k $ (is2 ! k)) *
      lookup (tensors-from-net m1 $ j) is1 * lookup (tensors-from-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 ... = (∑ is1 | is1 < input-sizes m1. (∏ k < length inputs1. inputs1 ! k
$ (is1 ! k)) * lookup (tensors-from-net m1 $ j) is1) *
      (∑ is2 | is2 < input-sizes m2. (∏ k < length inputs2. inputs2 ! k $
(is2 ! k)) * lookup (tensors-from-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 dimv inputs1 = input-sizes m1⟩ ⟨map dimv 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**  $\bigwedge \text{inputs}. \text{input-sizes } m1 = \text{map } \text{dim}_v \text{ inputs} \implies \text{evaluate-net } m1 \text{ inputs} = \text{evaluate-net } m2 \text{ inputs}$

**shows** *tensors-from-net m1 = tensors-from-net m2*

**proof** –

**have**  $\text{map } \text{dim}_v (\text{map } \mathbf{0}_v (\text{input-sizes } m2)) = \text{input-sizes } m2$

$\text{map } \text{dim}_v (\text{map } \mathbf{0}_v (\text{input-sizes } m1)) = \text{input-sizes } m1$  **by** (*simp-all add: nth-equalityI*)

**then have** *output-size' m1 = output-size' m2 using*

$\text{output-size-correct}[OF \langle \text{valid-net}' m1 \rangle \langle \text{map } \text{dim}_v (\text{map } \mathbf{0}_v (\text{input-sizes } m1)) = \text{input-sizes } m1 \rangle]$

$\text{output-size-correct}[OF \langle \text{valid-net}' m2 \rangle \langle \text{map } \text{dim}_v (\text{map } \mathbf{0}_v (\text{input-sizes } m2)) = \text{input-sizes } m2 \rangle]$

*assms(3) assms(4)*

**by** (*metis (no-types)*)

**have**  $\bigwedge \text{is}. \text{base-input } m1 \text{ is} = \text{base-input } m2 \text{ is}$

**unfolding** *base-input-def*  $\langle \text{input-sizes } m1 = \text{input-sizes } m2 \rangle$  **by** *metis*

**show** *?thesis by* (*rule vec-eqI, rule tensor-lookup-eqI; metis*

$\text{lookup-tensors-from-net}[OF \langle \text{valid-net}' m1 \rangle, \text{unfolded } \langle \bigwedge \text{is}. \text{base-input } m1 \text{ is} = \text{base-input } m2 \text{ is} \rangle \langle \text{output-size}' m1 = \text{output-size}' m2 \rangle]$

$\text{lookup-tensors-from-net}[OF \langle \text{valid-net}' m2 \rangle] \text{assms}(3) \text{base-input-length}$

$\text{assms}(1) \text{assms}(2) \text{dims-tensors-from-net output-size-correct-tensors vec-setI}$

$\langle \text{output-size}' m1 = \text{output-size}' m2 \rangle \text{assms}(4))$

**qed**

**end**

## 23 Concrete Matrices

**theory** *DL-Concrete-Matrices*

**imports** *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  $\Rightarrow$  nat  $\Rightarrow$  real mat*

**where** *eye-matrix nr nc = mat nr nc ( $\lambda(r, c). \text{if } r=c \text{ then } 1 \text{ else } 0$ )*

**lemma** *eye-matrix-dim: dim<sub>r</sub> (eye-matrix nr nc) = nr dim<sub>c</sub> (eye-matrix nr nc) = nc by* (*simp-all add: eye-matrix-def*)

**lemma** *row-eye-matrix:*

**assumes**  $i < nr$

**shows**  $\text{row } (\text{eye-matrix } nr \ nc) \ i = \text{unit}_v \ nc \ i$

**by** (*rule vec-eqI, simp add: assms eye-matrix-def vec-unit-def, simp add: eye-matrix-dim(2)*)

**lemma** *unit-eq-0*[simp]:  
**assumes**  $i: i \geq n$   
**shows**  $\text{unit}_v \ n \ i = \mathbf{0}_v \ n$   
**apply** (rule *vec-eqI*)  
**apply** (metis (mono-tags, lifting) *i leD vec-dim-vec vec-index-vec vec-unit-def vec-zero-def*)  
**by** *simp*

**lemma** *mult-eye-matrix*:  
**assumes**  $i < nr$   
**shows**  $(\text{eye-matrix } nr \ (\text{dim}_v \ v) \otimes_{mv} \ v) \ \$ \ i = (\text{if } i < \text{dim}_v \ v \ \text{then } v \ \$ \ i \ \text{else } 0)$  (is  $?a \ \$ \ i = ?b$ )  
**proof** –  
**have**  $?a \ \$ \ i = \text{row } (\text{eye-matrix } nr \ (\text{dim}_v \ v)) \ i \cdot v$  **using** *index-mat-mult-vec*  
*assms eye-matrix-dim* **by** *auto*  
**also have**  $\dots = \text{unit}_v \ (\text{dim}_v \ v) \ i \cdot v$  **using** *row-eye-matrix* *assms* **by** *auto*  
**also have**  $\dots = ?b$  **using** *scalar-prod-left-unit vec-elemsI unit-eq-0 scalar-prod-left-zero*  
**by** *fastforce*  
**finally show**  $?thesis$  **by** *auto*  
**qed**

**definition** *all1-vec*:: $\text{nat} \Rightarrow \text{real } \text{vec}$   
**where**  $\text{all1-vec } n = \text{vec } n \ (\lambda i. \ 1)$

**definition** *all1-matrix*:: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{real } \text{mat}$   
**where**  $\text{all1-matrix } nr \ nc = \text{mat } nr \ nc \ (\lambda(r, c). \ 1)$

**lemma** *all1-matrix-dim*:  $\text{dim}_r \ (\text{all1-matrix } nr \ nc) = nr$   $\text{dim}_c \ (\text{all1-matrix } nr \ nc) = nc$   
**by** (*simp-all add: all1-matrix-def*)

**lemma** *row-all1-matrix*:  
**assumes**  $i < nr$   
**shows**  $\text{row } (\text{all1-matrix } nr \ nc) \ i = \text{all1-vec } nc$   
**apply** (rule *vec-eqI*)  
**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**  $\text{all1-vec } (\text{length } xs) \cdot (\text{vec-of-list } xs) = \text{sum-list } xs$   
**proof** –  
**have**  $\text{all1-vec } (\text{length } xs) \cdot (\text{vec-of-list } xs) = (\sum i = 0..<\text{dim}_v \ (\text{vec-of-list } xs). \ \text{vec-of-list } xs \ \$ \ i)$   
**unfolding** *scalar-prod-def* **by** (metis (*no-types, lifting*) *all1-vec-def mult-cancel-right1 sum-ivl-cong*  
*vec.abs-eq vec-dim-vec vec-index-vec vec-of-list.abs-eq*)  
**also have**  $\dots = (\sum i = 0..<\text{length } xs. \ xs \ ! \ i)$  **using** *vec.abs-eq vec-dim-vec*

*vec-of-list.abs-eq*  
 by (*metis sum-ivl-cong vec-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  $((\text{all1-matrix } nr \ (\text{dim}_v \ v)) \otimes_{mv} v) \$ i = \text{sum-list } (\text{list-of-vec } v) \ (\text{is } ?a \$ i = \text{sum-list } (\text{list-of-vec } v))$   
 proof –  
 have  $?a \$ i = \text{row } (\text{all1-matrix } nr \ (\text{dim}_v \ v)) \ i \cdot v$  using *index-mat-mult-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 vec-dim-vec vec-list vec-of-list.abs-eq*)  
 finally show *?thesis* by *auto*  
 qed

**definition** *copy-first-matrix::nat  $\Rightarrow$  nat  $\Rightarrow$  real mat*  
 where *copy-first-matrix nr nc = mat nr nc* ( $\lambda(r, c). \text{if } c = 0 \text{ then } 1 \text{ else } 0$ )

**lemma** *copy-first-matrix-dim*:  $\text{dim}_r (\text{copy-first-matrix } nr \ nc) = nr \ \text{dim}_c (\text{copy-first-matrix } nr \ nc) = nc$   
 by (*simp-all add: copy-first-matrix-def*)

**lemma** *row-copy-first-matrix*:  
 assumes  $i < nr$   
 shows  $\text{row } (\text{copy-first-matrix } nr \ nc) \ i = \text{unit}_v \ nc \ 0$   
 apply (*rule vec-eqI*)  
 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  $\text{dim}_v \ v > 0$   
 shows  $(\text{copy-first-matrix } nr \ (\text{dim}_v \ v) \otimes_{mv} v) \$ i = v \$ 0 \ (\text{is } ?a \$ i = v \$ 0)$   
 proof –  
 have  $?a \$ i = \text{row } (\text{copy-first-matrix } nr \ (\text{dim}_v \ v)) \ i \cdot v$  using *index-mat-mult-vec*  
*assms copy-first-matrix-dim* by *auto*  
 also have ... =  $\text{unit}_v \ (\text{dim}_v \ v) \ 0 \cdot v$  using *row-copy-first-matrix* *assms* by *auto*  
 also have ... =  $v \$ 0$  using *assms(2) scalar-prod-left-unit vec-elems* by *blast*  
 finally show *?thesis* by *auto*  
 qed

end

## 24 Missing Lemmas of Finite\_Set

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

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

end
```

## 25 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

fun deep-model and deep-model' where
```

$deep\text{-}model' Y [] = Input Y \mid$   
 $deep\text{-}model' Y (r \# rs) = Pool (deep\text{-}model Y r rs) (deep\text{-}model Y r rs) \mid$   
 $deep\text{-}model Y r rs = Conv (Y, r) (deep\text{-}model' r rs)$

**abbreviation**  $deep\text{-}model'\text{-}l rs == deep\text{-}model' (rs!0) (tl rs)$

**abbreviation**  $deep\text{-}model\text{-}l rs == deep\text{-}model (rs!0) (rs!1) (tl (tl rs))$

**lemma** *valid-deep-model*: *valid-net* ( $deep\text{-}model Y r rs$ )

**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* ( $deep\text{-}model' r rs$ )

**apply** (*induction*  $rs$  *arbitrary*:  $r$ )

**apply** (*simp add*: *valid-net.intros*(1))

**by** (*metis*  $deep\text{-}model'.elims deep\text{-}model'.simps$ (2)  $deep\text{-}model.elims output\text{-}size.simps$  *valid-net.simps*)

**lemma** *input-sizes-deep-model'*:

**assumes**  $length rs \geq 1$

**shows**  $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = replicate (2^{length rs - 1}) (last rs)$

**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**  $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = [last rs]$

**by** (*metis*  $deep\text{-}model'.simps$ (1) *input-sizes.simps*(1) *last.simps* *list.sel*(3))

**then show**  $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = replicate (2^{length rs - 1}) (last rs)$

**by** (*metis* *One-nat-def*  $\langle [] = butlast rs \rangle$  *empty-replicate* *length-butlast* *list.size*(3) *power-0 replicate.simps*(2))

**next**

**case** (*Cons*  $r rs' rs$ )

**then have**  $input\text{-}sizes (deep\text{-}model'\text{-}l (tl rs)) = replicate (2^{length (tl rs) - 1}) (last rs)$

**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.prem*s *One-nat-def* *append-Cons* *append-butlast-last-id* *length-greater-0-conv* *less-le-trans* *list.sel*(3) *zero-less-Suc*)

**then have**  $deep\text{-}model'\text{-}l rs = Pool (deep\text{-}model\text{-}l rs) (deep\text{-}model\text{-}l rs)$

**by** (*metis* *Cons.hyps*(2) *One-nat-def* *butlast.simps*(2)  $deep\text{-}model'.elims$  *list.sel*(3) *list.simps*(3) *nth-Cons-0* *nth-Cons-Suc*)

**then have**  $input\text{-}sizes (deep\text{-}model'\text{-}l rs) = input\text{-}sizes (deep\text{-}model\text{-}l rs) @ input\text{-}sizes (deep\text{-}model\text{-}l rs)$

**using** *input-sizes.simps*(3) **by** *metis*

**also have**  $\dots = input\text{-}sizes (deep\text{-}model'\text{-}l (tl rs)) @ input\text{-}sizes (deep\text{-}model'\text{-}l (tl rs))$

**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* ( $2 \wedge (\text{length } (tl \text{ } rs) - 1)$ ) (*last rs*) @ *replicate* ( $2 \wedge$   
 $(\text{length } (tl \text{ } rs) - 1)$ ) (*last rs*)  
**using** *IH* **by** *auto*  
**also have** ... = *replicate* ( $2 \wedge (\text{length } rs - 1)$ ) (*last rs*)  
**using** *replicate-add*[of  $2 \wedge (\text{length } (tl \text{ } rs) - 1)$   $2 \wedge (\text{length } (tl \text{ } rs) - 1)$  *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*  $\geq 2$   
**shows** *input-sizes* (*deep-model-l rs*) = *replicate* ( $2 \wedge (\text{length } rs - 2)$ ) (*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* ( $2 \wedge (\text{length } rs - 2)$ ) (*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<sub>v</sub> input*  
**and**  $j < nr$   
**shows** *evaluate-net* (*Conv* (*eye-matrix nr* (*output-size' m*)) *m*) *input* \$ *j*  
= (*if*  $j < \text{output-size}' m$  *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*  $\langle j < nr \rangle$ , 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*  $i < \text{output-size}' m$  *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 dimv 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 = dimv (evaluate-net

```

$m$  input)  $\rangle$   $assms(4)$ )

**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** *Conv* == *Conv* (*copy-first-matrix*  $nr$  ( $output-size' m$ ))  $m$

**fix**  $is$

**assume**  $is < \text{Tensor.dims } (?a \$ i)$

**then have**  $is < \text{input-sizes } m$  **using**  $\langle \text{Tensor.dims } (?a \$ i) = \text{input-sizes } m \rangle$  **by**  
*auto*

**have** *valid-net' Conv* **by** (*simp add: assms copy-first-matrix-dim valid-net.intros(2)*  
*Conv-def*)

**have** *base-input*  $m$   $is = \text{base-input } Conv$   $is$  **by** (*simp add: Conv-def base-input-def*)

**have**  $i < \text{output-size}' Conv$  **unfolding** *Conv-def* *remove-weights.simps* *output-size.simps*

*copy-first-matrix-dim* **using**  $assms$  **by** *metis*

**show** *Tensor.lookup* ( $?a$   $\$ i$ )  $is = \text{Tensor.lookup } ?b$   $is$

**by** (*metis* *Conv-def*  $\langle \text{base-input } m is = \text{base-input } Conv is \rangle$   $\langle i < \text{output-size}'$   
*Conv*

$\langle is < \text{input-sizes } m \rangle$   $\langle \text{valid-net}' Conv \rangle$   $assms(1)$   $assms(2)$   $assms(3)$  *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*  $m = \text{map } \text{dim}_v$  *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*  $\langle i < nr \rangle$ , *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<sub>v</sub> ?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** *Conv<sub>m</sub> == 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<sub>v</sub> ?a) dims-tensors-from-net input-sizes.simps(2)* **by** (*metis*  
*vec-setI*)

**then have** *is < input-sizes Conv<sub>m</sub>* **by** (*simp add: Conv<sub>m</sub>-def*)

**have** *valid-net' Conv<sub>m</sub>* **by** (*simp add: Conv<sub>m</sub>-def assms all1-matrix-dim valid-net.intros(2)*)

**have** *i < output-size' Conv<sub>m</sub>* **using** *Conv<sub>m</sub>-def (i < dim<sub>v</sub> ?a) (valid-net' Conv<sub>m</sub>)*

*output-size-correct-tensors* **by** *presburger*

**have** *base-input Conv<sub>m</sub> is = base-input m is* **unfolding** *base-input-def Conv<sub>m</sub>-def*  
*input-sizes.simps* **by** *metis*

**have** *Tensor.lookup (?a \$ i) is = evaluate-net Conv<sub>m</sub> (base-input Conv<sub>m</sub> is) \$ i*

**using** *lookup-tensors-from-net[OF (valid-net' Conv<sub>m</sub>) (is < input-sizes Conv<sub>m</sub>)*  
*(i < output-size' Conv<sub>m</sub>)]*

**by** (*metis Conv<sub>m</sub>-def*)

**also have**  $\dots =$  *monoid-add-class.sum-list (list-of-vec (evaluate-net m (base-input*  
*Conv<sub>m</sub> is)))*

**using** *evaluate-net-Conv-all1 Conv<sub>m</sub>-def (is < input-sizes Conv<sub>m</sub>) assms base-input-length*  
*(i < nr)*

**by** *simp*

**also have**  $\dots =$  *monoid-add-class.sum-list (list-of-vec (map<sub>v</sub> (λA. lookup A*  
*is)(tensors-from-net m)))*

**unfolding** *(base-input Conv<sub>m</sub> 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)]*

*vec-eqI[of evaluate-net m (base-input m is) map<sub>v</sub> (λA. lookup A is) (tensors-from-net*  
*m)] vec-index-map(1) vec-index-map(2)*

**by** *force*

**also have**  $\dots =$  *monoid-add-class.sum-list (map (λA. lookup A is) (list-of-vec*  
*(tensors-from-net m)))*

**using** *vec-eqI[of vec-of-list (list-of-vec (map<sub>v</sub> (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 vec-index-map(1) vec-index-map(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_r ((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_c ((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**  $\dim_r$  ((if length  $rs = 0$  then eye-matrix else (if length  $rs = 1$  then all1-matrix else copy-first-matrix))  $Y r$ ) =  $Y$   
 $\dim_c$  ((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) vec-index-vec)

**lemma** tensor-prod-unit-vec:  
unit-vec  $M j \otimes$  unit-vec  $M j =$  tensor-from-lookup  $[M, M]$  ( $\lambda is.$  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:  $is \triangleleft$  Tensor.dims (unit-vec  $M j \otimes$  unit-vec  $M j$ )  
**then have**  $is \triangleleft [M, M]$  **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**  $[i1] \triangleleft$  Tensor.dims (unit-vec  $M j$ )  $[i2] \triangleleft$  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**  $\langle is = [i1] @ [i2] \rangle$   
 $lookup\text{-}tensor\text{-}prod[OF\ \langle [i1] \triangleleft Tensor.dims\ (unit\text{-}vec\ M\ j) \rangle\ \langle [i2] \triangleleft Tensor.dims$   
 $(unit\text{-}vec\ M\ j) \rangle]$   
 $lookup\text{-}tensor\text{-}from\text{-}lookup[OF\ \langle is \triangleleft [M, M] \rangle, unfolded\ \langle is = [i1] @ [i2] \rangle]$   
 $lookup\text{-}unit\text{-}vec[OF\ \langle i1 < M \rangle] lookup\text{-}unit\text{-}vec[OF\ \langle i2 < M \rangle]$  **by** *fastforce*  
**qed**

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

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

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

**lemma** *list-of-vec-map*:  $list\text{-}of\text{-}vec\ xs = map\ (op\ \$\ xs)\ [0..<dim_v\ xs]$  **by** *transfer*  
*auto*

```

lemma lookup-tensors-ht-l1:
assumes  $j < r1$ 
and  $is \triangleleft [M, M]$ 
shows  $Tensor.lookup (tensors-from-net (witness\ r1\ r0\ [M])\ \$\ j)\ is$ 
   $= (if\ is!0 = is!1 \wedge is!0 < r0\ then\ 1\ else\ 0)$ 
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  $dim_v (tensors-from-net (witness'\ r0\ [M])) = r0$ 
    using  $\langle output-size' (witness'\ r0\ [M]) = r0 \rangle$   $witness-l0'-valid$  output-size-correct-tensors
by fastforce
  have  $all0-but1: \bigwedge i. i \neq is!0 \implies i < r0 \implies Tensor.lookup (tensors-from-net (witness'\ r0\ [M])\ \$\ i)\ is = 0$ 
    using  $lookup-tensors-ht-l0' \langle is \triangleleft [M, M] \rangle$  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)]
     $witness-l0' \langle output-size' (witness'\ r0\ [M]) = r0 \rangle$  by simp
  then have  $Tensor.lookup (tensors-from-net (witness\ r1\ r0\ [M])\ \$\ j)\ is$ 
     $= monoid-add-class.sum-list (map (\lambda A. Tensor.lookup\ A\ is)\ (list-of-vec\ (tensors-from-net$ 
     $(witness'\ r0\ [M])))$ 
    using  $lookup-listsum [OF \langle is \triangleleft [M, M] \rangle \langle input-sizes (witness'\ r0\ [M]) = [M,$ 
     $M \rangle$ 
     $dims-tensors-from-net$  by (metis set-list-of-vec)
  also have  $\dots = monoid-add-class.sum-list (map (\lambda i. lookup (tensors-from-net$ 
     $(witness'\ r0\ [M])\ \$\ i)\ is)\ [0..<r0])$ 
    using  $map-map[of\ (\lambda A. Tensor.lookup\ A\ is)\ \lambda i. (tensors-from-net (witness'\ r0$ 
     $[M])\ \$\ i)\ [0..<r0]]$ 
    using  $list-of-vec-map \langle dim_v (tensors-from-net (witness'\ r0\ [M])) = r0 \rangle$  by
    (metis (mono-tags, lifting) comp-apply map-eq-conv)
  also have  $\dots = (\sum i < r0. Tensor.lookup ((tensors-from-net (witness'\ r0\ [M])\ \$\ i)\ is)$ 
    using  $sum-set-upt-conv-sum-list-nat\ atLeast0LessThan$  by (metis atLeast-upt)
  also have  $\dots = (if\ is!0 = is!1 \wedge is!0 < r0\ then\ 1\ else\ 0)$ 
proof (cases is!0 < r0)
  case True
  have finite  $\{0..<r0\}$  by auto
  have  $is!0 \in \{0..<r0\}$  using True by auto
  have  $(\sum i < r0. Tensor.lookup ((tensors-from-net (witness'\ r0\ [M])\ \$\ i)\ is)$ 
     $= Tensor.lookup (tensors-from-net (witness'\ r0\ [M])\ \$\ (is!0))\ is$ 

```

```

using ⟨dimv (tensors-from-net (witness' r0 [M])) = r0⟩
using sum.remove[OF ⟨finite {0..<r0}⟩ ⟨is!0 ∈ {0..<r0}⟩,
  of λi. (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 dimv (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 dimv (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:
dimv (tensors-from-net (witness-l rs)) = rs ! 0
  using length-output-deep-model witness-is-deep-model by blast

```

```

lemma length-output-witness':
dimv (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**  $\text{Tensor.dims} (\text{tensors-from-net} (\text{witness-l } rs) \$ j) = \text{replicate} (2^{(\text{length } rs - 2)}) (\text{last } rs)$

**using** *dims-output-deep-model witness-is-deep-model assms* **by** *blast*

**lemma** *dims-output-deep-model'*:

**assumes**  $\text{length } rs \geq 1$

**and**  $\bigwedge r. r \in \text{set } rs \implies r > 0$

**and**  $j < rs!0$

**and**  $\text{remove-weights } m = \text{deep-model}'\text{-l } rs$

**shows**  $\text{Tensor.dims} (\text{tensors-from-net } m \$ j) = \text{replicate} (2^{(\text{length } rs - 1)}) (\text{last } rs)$

**proof** –

**have**  $\text{dim}_v (\text{tensors-from-net } m) > j$

**using** *length-output-deep-model' (remove-weights m = deep-model'-l rs) (j < rs!0)* **by** *auto*

**then have**  $\text{Tensor.dims} (\text{tensors-from-net } m \$ j) = \text{input-sizes } m$

**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**  $\text{length } rs \geq 1$

**and**  $\bigwedge r. r \in \text{set } rs \implies r > 0$

**and**  $j < rs!0$

**shows**  $\text{Tensor.dims} (\text{tensors-from-net} (\text{witness}'\text{-l } rs) \$ j) = \text{replicate} (2^{(\text{length } rs - 1)}) (\text{last } rs)$

**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**  $r = \text{min} (\text{last } rs) (\text{last } (\text{butlast } rs))$

**definition**  $N\text{-half} = 2^{(\text{length } rs - 3)}$

**definition**  $\text{weight-space-dim} = \text{count-weights}(\text{deep-model-l } rs)$

**end**

**locale** *deep-model-correct-params-y* = *deep-model-correct-params* +

**fixes** *y::nat*

**assumes**  $y\text{-valid}:y < rs \ ! \ 0$   
**begin**

**definition**  $A \ ws = \text{tensors-from-net } (\text{insert-weights } (\text{deep-model-l } rs) \ ws) \ \$ \ y$

**definition**  $A' \ ws = \text{ten2mat } (A \ ws)$

**lemma**  $\text{dims-tensor-deep-model}$ :

**assumes**  $\text{remove-weights } m = \text{deep-model-l } rs$

**shows**  $\text{dims } (\text{tensors-from-net } m \ \$ \ y) = \text{replicate } (2 * N\text{-half}) \ (\text{last } rs)$

**proof** –

**have**  $\text{dims } (\text{tensors-from-net } m \ \$ \ y) = \text{replicate } (2 \ ^ \ (\text{length } rs - 2)) \ (\text{last } rs)$

**using**  $\text{dims-output-deep-model}[OF - \text{no-zeros } y\text{-valid } \text{assms}]$  **using**  $\text{less-imp-le-nat}$   
 $\text{Suc-le-lessD}$   $\text{deep numeral-3-eq-3}$

**by**  $\text{auto}$

**then show**  $?thesis$  **using**  $N\text{-half-def}$  **by**  $(\text{metis } \text{One-nat-def } \text{Suc-1 } \text{Suc-eq-plus1}$   
 $\text{Suc-le-lessD } \text{deep}$

$\text{diff-diff-left less-numeral-extra}(3) \ \text{numeral-3-eq-3 } \text{realpow-num-eq-if zero-less-diff})$

**qed**

**lemma**  $\text{order-tensor-deep-model}$ :

**assumes**  $\text{remove-weights } m = \text{deep-model-l } rs$

**shows**  $\text{order } (\text{tensors-from-net } m \ \$ \ y) = 2 * N\text{-half}$

**using**  $\text{dims-tensor-deep-model}$  **by**  $(\text{simp add: } \text{assms})$

**lemma**  $\text{dims-A}$ :

**shows**  $\text{Tensor.dims } (A \ ws) = \text{replicate } (2 * N\text{-half}) \ (\text{last } rs)$

**unfolding**  $A\text{-def}$

**using**  $\text{dims-tensor-deep-model}$   $\text{remove-insert-weights}$  **by**  $\text{blast}$

**lemma**  $\text{order-A}$ :

**shows**  $\text{order } (A \ ws) = 2 * N\text{-half}$  **using**  $\text{dims-A}$   $\text{length-replicate}$  **by**  $\text{auto}$

**lemma**  $\text{dims-A'}$ :

**shows**  $\text{dim}_r (A' \ ws) = \text{prod-list } (\text{sublist } (\text{Tensor.dims } (A \ ws)) \ \{n. \ \text{even } n\})$

**and**  $\text{dim}_c (A' \ ws) = \text{prod-list } (\text{sublist } (\text{Tensor.dims } (A \ ws)) \ \{n. \ \text{odd } n\})$

**unfolding**  $A'\text{-def}$   $\text{matricize-def}$  **by**  $(\text{simp-all add: } A\text{-def } \text{Collect-neg-eq})$

**lemma**  $\text{dims-A'-pow}$ :

**shows**  $\text{dim}_r (A' \ ws) = (\text{last } rs) \ ^ \ N\text{-half}$   $\text{dim}_c (A' \ ws) = (\text{last } rs) \ ^ \ N\text{-half}$

**unfolding**  $\text{dims-A'}$   $\text{dims-A}$   $\text{sublist-replicate}$   $\text{set-le-in card-even card-odd prod-list-replicate}$   
**by**  $\text{simp-all}$

**definition**  $Aw = \text{tensors-from-net } (\text{witness-l } rs) \ \$ \ y$

**definition**  $Aw' = \text{ten2mat } Aw$

**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**  $0:\exists x.$  *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* ( $2 * N\text{-half}$ ) (*last rs*)  
**unfolding** *Aw-def'* **using** *dims-A* **by** *auto*

**lemma** *order-Aw*: *order Aw* =  $2 * N\text{-half}$   
**unfolding** *Aw-def'* **using** *order-A* **by** *auto*

**lemma** *dims-Aw'*:

*dim<sub>r</sub> Aw'* = *prod-list* (*sublist* (*Tensor.dims Aw*) {*n. even n*})

*dim<sub>c</sub> Aw'* = *prod-list* (*sublist* (*Tensor.dims Aw*) {*n. odd n*})

**unfolding** *Aw'-def' Aw-def'* **using** *dims-A'* **by** *auto*

**lemma** *dims-Aw'-pow*: *dim<sub>r</sub> Aw'* = (*last rs*)  <sup>$\wedge N\text{-half}$</sup>  *dim<sub>c</sub> Aw'* = (*last rs*)  <sup>$\wedge N\text{-half}$</sup>

**unfolding** *Aw'-def' Aw-def'* **using** *dims-A'-pow* **by** *auto*

**lemma** *witness-tensor*:

**assumes** *is*  $\triangleleft$  *Tensor.dims Aw*

**shows** *Tensor.lookup Aw is*

= (*if* *sublist is* {*n. even n*} = *sublist is* {*n. odd n*}  $\wedge$  ( $\forall i \in \text{set } is. i < \text{last}(\text{butlast } rs)$ ) *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* ( $3 \leq \text{length } rs$ ))

**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.premis(4) length-output-witness ⟨rs = [rs ! 0, rs ! 1, rs ! 2]⟩ vec-setI by
metis
  then have is < [rs!2, rs!2] using Nil.premis 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.premis(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
    ↔ sublist is {n. even n} = sublist 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 sublist is {n. even n} = [is!0]
  apply (rule sublist-only-one)
  using subset-antisym less-2-cases ⟨length is = 2⟩ by fastforce
  have sublist is {n. odd n} = [is!1]
  apply (rule sublist-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: ⟨sublist is (Collect even) = [is ! 0]⟩ ⟨sublist is {n. odd n}
= [is ! 1]⟩)
  apply (metis ⟨length is = 2⟩ One-nat-def in-set-conv-nth less-2-cases)
  apply (simp add: ⟨sublist is (Collect even) = [is ! 0]⟩ ⟨sublist 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.prem(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.prem(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 sublist is' (Collect even) = sublist 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.prem(3) dims-output-witness 2 by (metis (no-types, lifting) Cons.prem(1)
Cons.prem(3)
Cons.prem(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.prem 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 = λis rs. sublist is {n. even n} = sublist 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 (sublist is2 (Collect even)) = length (sublist is2 (Collect odd))
    using length-sublist-even ⟨even (length is2)⟩ by blast
  have cond1-iff: (sublist is1 (Collect even) = sublist is1 {n. odd n} ∧ sublist
is2 (Collect even) = sublist is2 {n. odd n})
    = (sublist is (Collect even) = sublist is {n. odd n})
  unfolding ⟨is = is1 @ is2⟩ sublist-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 (sublist is2 (Collect even)) = length (sublist 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*:  $\text{Tensor.lookup } ((\text{tensors-from-net } (\text{witness-l } (\text{tl } rs)) \$ 0) \otimes (\text{tensors-from-net } (\text{witness-l } (\text{tl } rs))) \$ 0) \text{ is } =$   
*(if ?cond is rs then 1 else 0)*  
**using**  $\langle ?\text{cond is1 } (\text{tl } rs) \wedge ?\text{cond is2 } (\text{tl } rs) \longleftrightarrow ?\text{cond is } rs \rangle$   
**unfolding**  $\langle \text{is} = \text{is1} @ \text{is2} \rangle \text{ lookup-tensor-prod}[OF \text{ is1-valid is2-valid}] IH\text{-is12}$

**by** *auto*  
**have** *witness-l-tl*:  $\text{witness-l } (\text{tl } rs) = \text{witness } (rs ! 1) (rs ! 2) (\text{tl } (\text{tl } (\text{tl } rs)))$   
**by** *(metis One-nat-def Suc-1 ‹rs = r # tl rs› nth-Cons-Suc)*  
**have** *tl-tl*:  $(\text{tl } (\text{tl } rs)) = ((rs ! 2) \# \text{tl } (\text{tl } (\text{tl } rs)))$   
**proof** –  
**have** *length*  $(\text{tl } (\text{tl } rs)) \neq 0$   
**by** *(metis One-nat-def Suc-eq-plus1 diff-diff-left diff-is-0-eq length-tl not-less-eq-eq*

*Cons.premis(2) numeral-3-eq-3)*  
**then** **have**  $\text{tl } (\text{tl } rs) \neq []$   
**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<sub>v</sub>*  $(\text{tensors-from-net } (\text{witness } (rs ! 1) (rs ! 2) (\text{tl } (\text{tl } (\text{tl } rs)))))) > 0$   
**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**  $\text{tensors-from-net } (\text{witness}' (rs ! 1) ((rs ! 2) \# \text{tl } (\text{tl } (\text{tl } rs)))) \$ 0$   
 $= (\text{tensors-from-net } (\text{witness-l } (\text{tl } rs)) \$ 0) \otimes (\text{tensors-from-net } (\text{witness-l } (\text{tl } rs)) \$ 0)$   
**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'*  $(\text{witness}' (rs ! 1) (\text{tl } (\text{tl } rs)))$  **by** *(simp add: witness'-valid)*  
**have** *output-size'*  $(\text{witness}' (rs ! 1) (\text{tl } (\text{tl } rs))) = rs ! 1$   
**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*:  $(\text{if } \text{length } (\text{tl } (\text{tl } rs)) = 0 \text{ then eye-matrix else if } \text{length } (\text{tl } (\text{tl } rs)) = 1 \text{ then all1-matrix else copy-first-matrix}) = \text{copy-first-matrix}$   
**by** *(metis 2 Cons.premis(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 ‹ 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**  $i < \dim_r Aw'$  **and**  $j < \dim_c Aw'$

**shows**  $Aw' \text{ } \$\$ (i, j)$

= (if  $i=j \wedge (\forall i0 \in \text{set } (\text{digit-encode } (\text{sublist } (\text{Tensor.dims } Aw) \{n. \text{even } n\}) i). i0 < \text{last } (\text{butlast } rs))$  then 1 else 0)

**proof** –

```

def is == weave {n. even n}
  (digit-encode (sublist (Tensor.dims Aw) {n. even n}) i)
  (digit-encode (sublist (Tensor.dims Aw) {n. odd n}) j)
have lookup-eq:  $Aw' \text{ } \$\$ (i, j) = \text{Tensor.lookup } Aw \text{ } is$ 
using  $Aw' \text{-def } \text{matricize-def } \text{dims-Aw}'(1)[\text{symmetric, unfolded } A \text{-def}] \text{dims-Aw}'(2)[\text{symmetric,}$ 
 $\text{unfolded } A \text{-def } \text{Collect-neg-eq}]$ 
   $\text{mat-index-mat}(1)[OF \text{ } ‹i < \dim_r Aw'› ‹j < \dim_c Aw'›] \text{is-def } \text{Collect-neg-eq}$ 
 $\text{case-prod-conv}$ 
by (metis (no-types)  $Aw' \text{-def } \text{Collect-neg-eq } \text{case-prod-conv } \text{is-def } \text{matricize-def}$ )
have  $is \triangleleft \text{Tensor.dims } Aw$ 
using  $\text{is-def } \text{valid-index-weave } A \text{-def } \text{Collect-neg-eq } \text{assms } \text{digit-encode-valid-index}$ 
 $\text{dims-Aw}'$  by metis

```

**have** even (order Aw)

**unfolding**  $Aw \text{-def}$  **using**  $\text{assms } \text{dims-output-witness } \text{even-numeral } \text{le-eq-less-or-eq}$   
 $\text{numeral-2-eq-2 } \text{numeral-3-eq-3 } \text{deep } \text{no-zeros } y \text{-valid}$  **by** fastforce

**have**  $\text{sublist-dimsAw: } \text{sublist } (\text{Tensor.dims } Aw) (\text{Collect even}) = \text{sublist } (\text{Tensor.dims}$   
 $Aw) \{n. \text{odd } n\}$

**proof** –

**have**  $0 : \text{Tensor.dims } (\text{tensors-from-net } (\text{witness-l } rs) \$ y) = \text{replicate } (2 \wedge$   
 $(\text{length } rs - 2)) (\text{last } rs)$

**using**  $\text{dims-output-witness}[OF \text{ } \text{no-zeros } y \text{-valid}]$  **using** deep **by** linarith

**show** ?thesis **unfolding**  $A \text{-def}$

**using** sublist-replicate

**by** (metis (no-types, lifting)  $0 \text{ } Aw \text{-def } ‹\text{even } (\text{order } Aw)› \text{length-replicate}$   
 $\text{length-sublist-even}$ )

**qed**

**have**  $i = j \iff \text{sublist is (Collect even) = sublist is \{n. odd n\}}$   
**proof**  
**have**  $\text{eq-lengths: length (digit-encode (sublist (Tensor.dims Aw) (Collect even)))}$   
 $i)$   
 $= \text{length (digit-encode (sublist (Tensor.dims Aw) \{n. odd n\} j))}$   
**unfolding**  $\text{length-digit-encode by (metis \langle even (order Aw) \rangle length-sublist-even)}$   
  
**then show**  $i = j \implies \text{sublist is (Collect even) = sublist is \{n. odd n\}}$  **unfolding**  
 $\text{is-def}$   
**using**  $\text{sublist-weave[of digit-encode (sublist (Tensor.dims Aw) (Collect even))]}$   
 $i$   
 $\text{Collect even digit-encode (sublist (Tensor.dims Aw) \{n. odd n\} j, \text{un-}$   
 $\text{folded eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric]}$   
 $\text{card-odd]}$   
 $\text{sublist-dimsAw by simp}$   
**show**  $\text{sublist is (Collect even) = sublist is \{n. odd n\} \implies i = j}$  **unfolding**  
 $\text{is-def}$   
**using**  $\text{sublist-weave[of digit-encode (sublist (Tensor.dims Aw) (Collect even))]}$   
 $i$   
 $\text{Collect even digit-encode (sublist (Tensor.dims Aw) \{n. odd n\} j, \text{un-}$   
 $\text{folded eq-lengths, unfolded Collect-neg-eq[symmetric] card-even mult-2[symmetric]}$   
 $\text{card-odd]}$   
**using**  $\text{Divides.mod-less \langle sublist (Tensor.dims Aw) (Collect even) = sublist}$   
 $\text{(Tensor.dims Aw) \{n. odd n\} \rangle}$   
 $\text{deep no-zeros y-valid assms digit-decode-encode dims-Aw' by metis}$   
**qed**

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

**definition**  $\text{rows-with-1} = \{i. (\forall i0 \in \text{set (digit-encode (sublist (Tensor.dims Aw) \{n. even n\}) i). i0 < last (butlast rs)})\}$

**lemma**  $\text{card-low-digits:}$

**assumes**  $m > 0 \wedge d. d \in \text{set ds} \implies m \leq d$

```

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 == λds 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.prem1 Cons.prem2)
  have card {i. low-digits (d # ds) i} = card {..proof -
  def f == λp. fst p + d * snd p
  have inj-on f {..proof (rule inj-onI)
  fix x y assume x ∈ {..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 ' {..proof (rule subset-antisym; rule subsetI)
  fix x assume x ∈ f ' {..then obtain i0 i1 where x = i0 + d * i1 i0 < m low-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 ∈ {i. low-digits (d # ds) i} unfolding low-digits-def
  proof (rule; rule conjI)
  have i1 < prod-list ds ∀ i0 ∈ set (digit-encode ds i1). i0 < m
  using (low-digits ds i1) low-digits-def by auto
  show x < prod-list (d # ds) unfolding prod-list.Cons ⟨x = i0 + d * i1⟩
using ⟨i0 < d⟩ ⟨i1 < prod-list ds⟩
  proof -
  have d ≠ 0
  by (metis ⟨i0 < d⟩ gr-implies-not0)
  then have (i0 + d * i1) div (d * prod-list ds) = 0
  by (simp add: Divides.div-mult2-eq ⟨i0 < d⟩ ⟨i1 < prod-list ds⟩)
  then show i0 + d * i1 < d * prod-list ds
  by (metis (no-types) ⟨i0 < d⟩ ⟨i1 < prod-list ds⟩ div-eq-0-iff gr-implies-not0
no-zero-divisors)
  qed
  show ∀ i0 ∈ set (digit-encode (d # ds) x). i0 < m

```

```

      using ⟨∀ i0 ∈ set (digit-encode ds i1). i0 < m⟩ ⟨i0 < d⟩ ⟨i0 < m⟩ ⟨x = i0
+ d * i1⟩ by auto
    qed
  next
    fix x assume x ∈ {i. low-digits (d # ds) i}
    then have x < prod-list (d # ds) ∀ i0 ∈ set (digit-encode (d # ds) x). i0 <
m using low-digits-def by auto
    have x mod d < m using ⟨∀ i0 ∈ set (digit-encode (d # ds) x). i0 < m⟩[unfolded
digit-encode.simps] by simp
    have x div d < prod-list ds using ⟨x < prod-list (d # ds)⟩[unfolded prod-list.Cons]

      by (metis Divides.div-mult2-eq div-eq-0-iff gr-implies-not0 mult-0-right)
    have ∀ i0 ∈ set (digit-encode ds (x div d)). i0 < m by (simp add: ⟨∀ i0 ∈ set
(digit-encode (d # ds) x). i0 < m⟩)
    have f ((x mod d), (x div d)) = x by (simp add: f-def)
    show x ∈ f ‘ (⟨..<m⟩ × {i. low-digits ds i}) by (metis SigmaI ⟨∀ i0 ∈ set
(digit-encode ds (x div d)). i0 < m⟩ ⟨f (x mod d, x div d) = x⟩ ⟨x div d < prod-list
ds⟩ ⟨x mod d < m⟩ image-eqI lessThan-iff low-digits-def mem-Collect-eq)
    qed
    then have bij-betw f (⟨..<m⟩ × {i. low-digits ds i}) {i. low-digits (d # ds) i}
      by (simp add: ⟨inj-on f (⟨..<m⟩ × {i. low-digits ds i})⟩ bij-betw-def)
    then show ?thesis by (simp add: bij-betw-same-card)
  qed
  then show ?case unfolding ⟨card {i. low-digits ds i} = m ^ (length ds)⟩
card-cartesian-product using low-digits-def by simp
qed

```

**lemma** *card-rows-with-1*:  $\text{card } \{i \in \text{rows-with-1}. i < \dim_r Aw'\} = r \wedge N\text{-half}$

**proof** –

```

  have 1: {i ∈ rows-with-1. i < dim_r Aw'} = {i. i < prod-list (sublist (Tensor.dims
Aw) (Collect even))} ∧
    (∀ i0 ∈ set (digit-encode (sublist (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_r Aw' ∀ i0 ∈ set (digit-encode (sublist (Tensor.dims Aw) {n.
even n}) i). i0 < last (butlast rs)
      using rows-with-1-def by auto
    then have i < prod-list (sublist (dims Aw) (Collect even)) using dims-Aw'
by linarith
    then have digit-encode (sublist (dims Aw) (Collect even)) i ≺ sublist (dims
Aw) (Collect even)
      using digit-encode-valid-index by auto
    have ∀ i0 ∈ set (digit-encode (sublist (Tensor.dims Aw) {n. even n}) i). i0 < r
proof
      fix i0 assume 1: i0 ∈ set (digit-encode (sublist (dims Aw) (Collect even)) i)
      then obtain k where k < length (digit-encode (sublist (dims Aw) (Collect
even)) i)
        digit-encode (sublist (dims Aw) (Collect even)) i ! k = i0 by (meson

```

```

in-set-conv-nth)
  have i0 < last (butlast rs)
    using ⟨∀ i0 ∈ set (digit-encode (sublist (dims Aw) (Collect even))) i. i0 <
last (butlast rs)⟩ 1 by blast
  have set (sublist (dims Aw) (Collect even)) ⊆ {last rs} unfolding dims-Aw
using subset-eq by fastforce
  then have sublist (dims Aw) (Collect even) ! k = last rs
    using ⟨digit-encode (sublist (dims Aw) (Collect even)) i < sublist (dims
Aw) (Collect even)⟩
    ⟨k < length (digit-encode (sublist (dims Aw) (Collect even))) i⟩
    nth-mem valid-index-length by auto
  then have i0 < last rs
    using valid-index-lt ⟨digit-encode (sublist (dims Aw) (Collect even)) i ! k =
i0⟩
    ⟨digit-encode (sublist (dims Aw) (Collect even)) i < sublist (dims Aw)
(Collect even)⟩
    ⟨k < length (digit-encode (sublist (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 (sublist (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: ∧ d. d ∈ set (sublist (Tensor.dims Aw) (Collect even)) ⇒ r ≤ d
  proof -
    fix d assume d ∈ set (sublist (Tensor.dims Aw) (Collect even))
    then have d ∈ set (Tensor.dims Aw) using in-set-sublistD by fast
    then have d = last rs using dims-Aw by simp
    then show r ≤ 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 (sublist (Tensor.dims Aw) (Collect even)) = N-half
    unfolding length-sublist order-Aw using card-even[of N-half]
    by (metis (mono-tags, lifting) Collect-cong)
  then show ?thesis using card-low-digits[of r sublist (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 (sublist (Tensor.dims Aw) {n. even n})
  have ∧ i. listpr dvd i ⇒ i ∈ rows-with-1
  proof -

```

```

fix i assume dvd-i: listpr dvd i
{
  fix i0::nat
  assume i0∈set (digit-encode (sublist (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 ⟨∧i. listpr dvd i ⟹ i ∈ rows-with-1⟩
by (meson dvd-triv-left image-subset-iff infinite-iff-countable-subset)
qed

```

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

**proof**

```

show dimr (submatrix Aw' rows-with-1 rows-with-1) = dimr ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ ))
  unfolding mat-index-one(2) dim-submatrix(1)
  by (metis (full-types) set-le-in card-rows-with-1)
show dimc (submatrix Aw' rows-with-1 rows-with-1) = dimc ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ ))
  by (metis ⟨dimr (submatrix Aw' rows-with-1 rows-with-1) = dimr ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ ))⟩
dim-submatrix(1) dim-submatrix(2) mat-index-one(2) mat-index-one(3)
dims-Aw'-pow)
show ∧i j. i < dimr ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ )) ⟹
  j < dimc ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ )) ⟹ submatrix Aw' rows-with-1 rows-with-1
  $$ (i, j) =  $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ ) $$ (i, j)
proof -
  fix i j assume i < dimr ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ )) j < dimc ( $\mathbf{1}_m$  ( $r \wedge N\text{-half}$ ))
  then have i < r ^ N-half j < r ^ N-half by auto
  then have i < card {i ∈ rows-with-1. i < dimr Aw'} j < card {i ∈ rows-with-1.
i < dimc Aw'}
  using card-rows-with-1 dims-Aw'-pow by auto
  then have pick rows-with-1 i < dimr Aw' pick rows-with-1 j < dimc Aw'
  using card-le-pick-inf[OF infinite-rows-with-1, of dimr Aw' i]
  using card-le-pick-inf[OF infinite-rows-with-1, of dimc Aw' j] by force+
  have ∀ i0∈set (digit-encode (sublist (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 < dimr Aw⟩ ⟨pick
rows-with-1 j < dimc Aw⟩] by simp
  then have submatrix Aw' rows-with-1 rows-with-1 $$$ (i, j) = (if pick rows-with-1
i = pick rows-with-1 j then 1 else 0)
    using submatrix-index by (metis (no-types, lifting)
⟨dimc (submatrix Aw' rows-with-1 rows-with-1) = dimc (1m (r ^ N-half))⟩
⟨dimr (submatrix Aw' rows-with-1 rows-with-1) = dimr (1m (r ^ N-half))⟩
⟨i < dimr (1m (r ^ N-half))⟩ ⟨j < r ^ N-half⟩ dim-submatrix(1) dim-submatrix(2)
mat-index-one(3))
    then have submatrix Aw' rows-with-1 rows-with-1 $$$ (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 $$$ (i, j) = 1m (r ^ N-half)
$$$ (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)  $\neq 0$  **unfolding**  
*witness-submatrix* by simp

end

end

## 26 Material to be moved to HOL finally

**theory** *PP-Auxiliary*

**imports**

*Main*

**begin**

### 26.1 Already taken over into Isabelle repository

### 26.2 Not yet taken over into Isabelle repository

**lemma** (in *comm-monoid-set*) *mono-neutral-cong*:

assumes [*simp*]: *finite T finite S*

and \* [*rule-format*]:  $\forall i \in T - S. h i = z \ \forall i \in S - T. g i = z$

and *gh*:  $\bigwedge x. x \in S \cap T \implies g x = h x$  **shows**  $F g S = F h T$

**proof** –

have  $F g S = F g (S \cap T)$

by(*rule mono-neutral-right*)(*auto intro: \**)

also have  $\dots = F h (S \cap T)$  **using** *refl gh* **by**(*rule cong*)

also have  $\dots = F h T$

by(*rule mono-neutral-left*)(*auto intro: \**)

**finally show** *?thesis* .

qed

end

## 27 Less common functions on lists

**theory** *PP-More-List2*

**imports**

*Main*

*PP-Auxiliary*

**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**  $\forall x \in set\ zs. P\ x$  **and**  $xs = ys\ @\ zs$   
**proof** (rule that)  
**show**  $strip\_while\ P\ xs = strip\_while\ P\ xs ..$   
**show**  $\forall x \in set\ (rev\ (takeWhile\ P\ (rev\ xs))). P\ x$  **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*:  
 $strip\_while\ P\ (map\ f\ xs) = map\ f\ (strip\_while\ (P\ \circ\ f)\ xs)$   
**by** (simp add: strip-while-def rev-map dropWhile-map)

**definition** *no-leading* ::  $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$   
**where**  
 $no\_leading\ P\ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P\ (hd\ xs))$

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

**lemma** *no-leading-Cons* [simp, intro!]:  
 $no\_leading\ P\ (x\ \#\ xs) \longleftrightarrow \neg P\ x$   
**by** (simp add: no-leading-def)

**lemma** *no-leading-append* [simp]:  
 $no\_leading\ P\ (xs\ @\ ys) \longleftrightarrow no\_leading\ P\ xs \wedge (xs = [] \longrightarrow no\_leading\ P\ ys)$   
**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**  $\bigwedge z. z \in set\ zs \implies P\ z$  **and**  $no\_leading\ P\ ys$   
**proof** –  
**from** *assms* **have**  $\exists zs. xs = zs\ @\ ys \wedge (\forall z \in set\ zs. P\ z) \wedge no\_leading\ P\ ys$   
**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.prem
  have  $\exists zs. xs = zs @ ys \wedge (\forall a \in \text{set } zs. P a) \wedge \text{no-leading } P \text{ ys}$ 
  by simp
  then obtain zs where  $xs = zs @ ys$  and  $\bigwedge z. z \in \text{set } zs \implies P z$ 
  and *: no-leading P ys
  by blast
  with True have  $x \# xs = (x \# zs) @ ys$  and  $\bigwedge z. z \in \text{set } (x \# zs) \implies P z$ 
  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*:  
 $\text{dropWhile } P \text{ xs} = \text{xs} \iff \text{no-leading } P \text{ xs}$   
**by** (cases xs) (auto elim: dropWhile-eq-obtain-leading)

**abbreviation** *no-trailing* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool  
**where**  
 $\text{no-trailing } P \text{ xs} \equiv \text{no-leading } P (\text{rev } \text{xs})$

**lemma** *no-trailing-unfold*:  
 $\text{no-trailing } P \text{ xs} \iff (xs \neq [] \longrightarrow \neg P (\text{last } xs))$   
**by** (induct xs) simp-all

**lemma** *no-trailing-Nil* [simp, intro!]:  
 $\text{no-trailing } P []$   
**by** simp

**lemma** *no-trailing-Cons* [simp]:  
 $\text{no-trailing } P (x \# xs) \iff \text{no-trailing } P \text{ xs} \wedge (xs = [] \longrightarrow \neg P x)$   
**by** simp

**lemma** *no-trailing-append-Cons* [simp]:  
 $\text{no-trailing } P (xs @ y \# ys) \iff \text{no-trailing } P (y \# ys)$   
**by** simp

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

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

**lemma** *no-trailing-map*:  
*no-trailing*  $P (\text{map } f xs) = \text{no-trailing } (P \circ f) xs$   
**by** (*simp add: last-map no-trailing-unfold*)

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

**definition** *nth-default* ::  $'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$   
**where**  
*nth-default*  $dflt xs n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } dflt)$

**lemma** *nth-default-nth*:  
 $n < \text{length } xs \implies \text{nth-default } dflt xs n = xs ! n$   
**by** (*simp add: nth-default-def*)

**lemma** *nth-default-beyond*:

$length\ xs \leq n \implies nth\ default\ dflt\ xs\ n = dflt$   
**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*:  
 $nth\ default\ dflt\ (x \# xs)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ n' \Rightarrow nth\ default\ dflt\ xs\ n')$   
**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*:  
 $nth\ default\ dflt\ xs\ k = dflt \iff (k < length\ xs \implies xs\ !\ k = dflt)$   
**by** (*simp add: nth-default-def*)

**lemma** *in-enumerate-iff-nth-default-eq*:  
 $x \neq dflt \implies (n, x) \in set\ (enumerate\ 0\ xs) \iff nth\ default\ dflt\ xs\ n = x$   
**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*:  
 $f \text{ dflt}' = \text{dflt} \implies \text{nth-default dflt} (\text{map } f \text{ } xs) \text{ } n = f (\text{nth-default dflt}' \text{ } xs \text{ } n)$   
**by** (*simp add: nth-default-def*)

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

**lemma** *sorted-list-of-set-nth-default*:  
 $\text{sorted-list-of-set} \{k. \text{nth-default dflt } xs \text{ } k \neq \text{dflt}\} = \text{map fst} (\text{filter } (\lambda(-, x). x \neq \text{dflt}) (\text{enumerate } 0 \text{ } xs))$   
**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*:  
 $\text{map} (\text{nth-default } x \text{ } xs) [0..<\text{length } xs] = xs$   
**proof** –  
**have**  $*$ :  $\text{map} (\text{nth-default } x \text{ } xs) [0..<\text{length } xs] = \text{map} (\text{List.nth } xs) [0..<\text{length } xs]$   
**by** (*rule map-cong*) (*simp-all add: nth-default-nth*)  
**show** *?thesis* **by** (*simp add: \* map-nth*)  
**qed**

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

**lemma** *nth-strip-while*:  
**assumes**  $n < \text{length} (\text{strip-while } P \text{ } xs)$   
**shows**  $\text{strip-while } P \text{ } xs ! n = xs ! n$   
**proof** –  
**have**  $\text{length} (\text{dropWhile } P (\text{rev } xs)) + \text{length} (\text{takeWhile } P (\text{rev } xs)) = \text{length } xs$   
**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*:  
 $\text{length} (\text{strip-while } P \text{ } xs) \leq \text{length } xs$   
**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*]:  
 $\text{nth-default dflt} (\text{strip-while} (\text{op} = \text{dflt}) \text{ } xs) = \text{nth-default dflt } xs$   
**by** (*induct xs rule: rev-induct*) *auto*

**lemma** *nth-default-eq-iff*:  
 $nth\text{-default}\ dflt\ xs = nth\text{-default}\ dflt\ ys$   
 $\longleftrightarrow strip\text{-while}\ (HOL.eq\ dflt)\ xs = strip\text{-while}\ (HOL.eq\ dflt)\ ys$  (**is**  $?P \longleftrightarrow ?Q$ )

**proof**  
**let**  $?xs = strip\text{-while}\ (HOL.eq\ dflt)\ xs$  **and**  $?ys = strip\text{-while}\ (HOL.eq\ dflt)\ ys$   
**assume**  $?P$   
**then have**  $eq: nth\text{-default}\ dflt\ ?xs = nth\text{-default}\ dflt\ ?ys$   
**by** *simp*  
**have**  $len: length\ ?xs = length\ ?ys$   
**proof** (*rule ccontr*)  
**assume**  $len: length\ ?xs \neq length\ ?ys$   
**{ fix**  $xs\ ys :: 'a\ list$   
**let**  $?xs = strip\text{-while}\ (HOL.eq\ dflt)\ xs$  **and**  $?ys = strip\text{-while}\ (HOL.eq\ dflt)\ ys$   
**assume**  $eq: nth\text{-default}\ dflt\ ?xs = nth\text{-default}\ dflt\ ?ys$   
**assume**  $len: length\ ?xs < length\ ?ys$   
**then have**  $length\ ?ys > 0$  **by** *arith*  
**then have**  $?ys \neq []$  **by** *simp*  
**with** *last-conv-nth-default* [*of ?ys dflt*]  
**have**  $last\ ?ys = nth\text{-default}\ dflt\ ?ys\ (length\ ?ys - 1)$   
**by** *auto*  
**moreover from**  $\langle ?ys \neq [] \rangle$  *no-trailing-strip-while* [*of HOL.eq dflt ys*]  
**have**  $last\ ?ys \neq dflt$  **by** (*simp add: no-trailing-unfold*)  
**ultimately have**  $nth\text{-default}\ dflt\ ?xs\ (length\ ?ys - 1) \neq dflt$   
**using**  $eq$  **by** *simp*  
**moreover from**  $len$  **have**  $length\ ?ys - 1 \geq length\ ?xs$  **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**  $xs: nth\text{-default}\ dflt\ ?xs\ n = ?xs\ !\ n$   
**and**  $ys: nth\text{-default}\ dflt\ ?ys\ n = ?ys\ !\ n$   
**by** (*simp-all only: nth-default-nth*)  
**with**  $eq$  **show**  $?xs\ !\ n = ?ys\ !\ n$   
**by** *simp*

**qed**  
**next**  
**assume**  $?Q$   
**then have**  $nth\text{-default}\ dflt\ (strip\text{-while}\ (HOL.eq\ dflt)\ xs) = nth\text{-default}\ dflt\ (strip\text{-while}\ (HOL.eq\ dflt)\ ys)$   
**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
  ~~/src/HOL/Library/Groups-Big-Fun
  ~~/src/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 ⇒ 'a :: mult-zero
  assumes finite {a. f a ≠ 0}
  shows finite {a. g a * f a ≠ 0}
proof -
  have {a. g a * f a ≠ 0} ⊆ {a. f a ≠ 0} by auto
  then show ?thesis using assms by (rule finite-subset)
qed

```

```

lemma finite-mult-not-eq-zero-rightI:
  fixes f :: 'b ⇒ 'a :: mult-zero
  assumes finite {a. f a ≠ 0}
  shows finite {a. f a * g a ≠ 0}
proof -
  have {a. f a * g a ≠ 0} ⊆ {a. f a ≠ 0} by auto
  then show ?thesis using assms by (rule finite-subset)
qed

```

```

lemma finite-mult-not-eq-zero-prodI:
  fixes f g :: 'a ⇒ '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). f a * g b ≠ 0}
proof -
  from assms have finite (?F × ?G)
    by blast
  then have finite {(a, b). f a ≠ 0 ∧ g b ≠ 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  $\Rightarrow$  'b::semiring-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 | a b. f a  $\neq$  0  $\wedge$  g b  $\neq$  0} (is finite ?FG)
proof -
  from assms have finite (?F  $\times$  ?G)
    by (simp add: finite-cartesian-product-iff)
  then have finite (case-prod plus ' (?F  $\times$  ?G))
    by (rule finite-imageI)
  also have case-prod plus ' (?F  $\times$  ?G) = ?FG
    by auto
  finally show ?thesis
    by simp
qed

```

```

lemma finite-mult-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add  $\Rightarrow$  'b::semiring-0
  assumes finite {a. f a  $\neq$  0}
  assumes finite {b. g b  $\neq$  0}
  shows finite {a + b | a b. f a * g b  $\neq$  0}
proof -
  from assms
  have finite {a + b | a b. f a  $\neq$  0  $\wedge$  g b  $\neq$  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.  $\exists$  b. f a b  $\neq$  0}
  shows finite {a. Sum-any (f a)  $\neq$  0}
proof -
  have {a. Sum-any (f a)  $\neq$  0}  $\subseteq$  {a.  $\exists$  b. f a b  $\neq$  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  $\Rightarrow$  bool  $\Rightarrow$  '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]*:  
 $P \implies (a \text{ when } P) = a$   
 $\neg P \implies (a \text{ when } P) = 0$   
**by** (*simp-all add: when-def*)

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

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

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

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

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

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

**end**

**context** *monoid-add*  
**begin**

**lemma** *when-add-distrib*:  
 $(a + b \text{ when } P) = (a \text{ when } P) + (b \text{ when } P)$   
**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 (λa. (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*:

$(- a \text{ when } P) = - (a \text{ when } P)$   
**by** (*simp add: when-def*)

**end**

**context** *mult-zero*

**begin**

**lemma** *mult-when*:

$a * (b \text{ when } P) = (a * b \text{ when } P)$   
**by** (*cases P*) *simp-all*

**lemma** *when-mult*:

$(a \text{ when } P) * b = (a * b \text{ when } P)$   
**by** (*cases P*) *simp-all*

**end**

## 28.2 Type definition

The following type is of central importance:

**typedef** (**overloaded**) (*'a, 'b poly-mapping*  $((- \Rightarrow_0 /-) [1, 0] 0) =$   
 $\{f :: 'a \Rightarrow 'b :: \text{zero. finite } \{x. f x \neq 0\}\}$   
**morphisms** *lookup Abs-poly-mapping*

**proof** –

**have**  $(\lambda :: 'a. (0 :: 'b)) \in ?\text{poly-mapping}$  **by** *simp*  
**then show** *?thesis* **by** (*blast intro!: exI*)

**qed**

**lemma** *lookup-Abs-poly-mapping*:

$\text{finite } \{x. f x \neq 0\} \implies \text{lookup } (\text{Abs-poly-mapping } f) = f$   
**using** *Abs-poly-mapping-inverse [of f]* **by** *simp*

**lemma** *finite-lookup [simp]*:

$\text{finite } \{k. \text{lookup } f k \neq 0\}$   
**using** *poly-mapping.lookup [of f]* **by** *simp*

**lemma** *finite-lookup-nat [simp]*:

**fixes**  $f :: 'a \Rightarrow_0 \text{nat}$   
**shows**  $\text{finite } \{k. 0 < \text{lookup } f k\}$   
**using** *poly-mapping.lookup [of f]* **by** *simp*

**lemma** *poly-mapping-eqI*:

**assumes**  $\bigwedge k. \text{lookup } f k = \text{lookup } g k$   
**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

$$\text{lookup}::('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow 'b$$

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

satisfying

$$\text{Abs-poly-mapping} (\text{lookup } ?x) = ?x$$

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

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  $(\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a$  later in theory *MPoly* gives a suitable representation type for polynomials ‘almost for free’: Interpreting  $\text{nat} \Rightarrow_0 \text{nat}$  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  $\text{nat} \Rightarrow_0 'a$  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  $\Rightarrow_0$  '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 diff-diff-add)+

end

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

lift-definition uminus-poly-mapping :: ('a  $\Rightarrow_0$  'b)  $\Rightarrow$  'a  $\Rightarrow_0$  '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*:  
 $lookup\ 1\ k = (1\ \text{when}\ k = 0)$   
 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**  
 $prod\text{-}fun\ f1\ f2\ k = (\sum l. f1\ l * (\sum q. (f2\ q\ \text{when}\ k = l + q)))$

**lemma** *prod-fun-unfold-prod*:  
 fixes  $f\ g :: 'a :: monoid\text{-}add \Rightarrow 'b :: semiring\text{-}0$   
 assumes *fin-f*: *finite* { $a. f\ a \neq 0$ }  
 assumes *fin-g*: *finite* { $b. g\ b \neq 0$ }  
 shows  $prod\text{-}fun\ f\ g\ k = (\sum (a, b). f\ a * g\ b\ \text{when}\ k = a + b)$

**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  $\{a. \exists b. (f\ a * g\ b\ \text{when}\ k = a + b) \neq 0\} \times$   
 $\{b. \exists a. (f\ a * g\ b\ \text{when}\ k = a + b) \neq 0\} \subseteq \{a. f\ a \neq 0\} \times \{b. g\ b \neq 0\}$   
 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  $\{q. (f2\ q\ \text{when}\ k = l + q) \neq 0\} \subseteq \{q. f2\ q \neq 0 \wedge k = l + q\}$  by *auto*

**with** *fin2* **have**  $\text{sum } f2 \{q. f2 \ q \neq 0 \wedge k = l + q\} = (\sum q. (f2 \ q \ \text{when } k = l + q))$   
**by** (*simp add: Sum-any.expand-superset [of {q. f2 q ≠ 0 ∧ k = l + q}]*) }  
**note** *aux = this*  
**have**  $\{k. (\sum l. f1 \ l * \text{sum } f2 \{q. f2 \ q \neq 0 \wedge k = l + q\}) \neq 0\}$   
 $\subseteq \{k. (\exists l. f1 \ l * \text{sum } f2 \{q. f2 \ q \neq 0 \wedge k = l + q\} \neq 0)\}$   
**by** (*auto elim!: Sum-any.not-neutral-obtains-not-neutral*)  
**also have**  $\dots \subseteq \{k. (\exists l. f1 \ l \neq 0 \wedge \text{sum } f2 \{q. f2 \ q \neq 0 \wedge k = l + q\} \neq 0)\}$   
**by** (*auto dest: mult-not-zero*)  
**also have**  $\dots \subseteq \{k. (\exists l. f1 \ l \neq 0 \wedge (\exists q. f2 \ q \neq 0 \wedge k = l + q))\}$   
**by** (*auto elim!: sum.not-neutral-contains-not-neutral*)  
**finally have**  $\text{finite } \{k. (\sum l. f1 \ l * \text{sum } f2 \{q. f2 \ q \neq 0 \wedge k = l + q\}) \neq 0\}$   
**using** \* **by** (*rule finite-subset*)  
**with** *aux* **have**  $\text{finite } \{k. (\sum l. f1 \ l * (\sum q. (f2 \ q \ \text{when } k = l + q))) \neq 0\}$   
**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*:  $\text{finite } \{a. f \ a \neq 0\}$  (**is** *finite ?F*)  
**assume** *fin-g*:  $\text{finite } \{b. g \ b \neq 0\}$  (**is** *finite ?G*)  
**assume** *fin-h*:  $\text{finite } \{c. h \ c \neq 0\}$  (**is** *finite ?H*)  
**from** *fin-f fin-g* **have** *fin-fg*:  $\text{finite } \{(a, b). f \ a * g \ b \neq 0\}$  (**is** *finite ?FG*)  
**by** (*rule finite-mult-not-eq-zero-prodI*)  
**from** *fin-g fin-h* **have** *fin-gh*:  $\text{finite } \{(b, c). g \ b * h \ c \neq 0\}$  (**is** *finite ?GH*)  
**by** (*rule finite-mult-not-eq-zero-prodI*)  
**from** *fin-f fin-g* **have** *fin-fg'*:  $\text{finite } \{a + b \mid a \ b. f \ a * g \ b \neq 0\}$  (**is** *finite ?FG'*)  
**by** (*rule finite-mult-not-eq-zero-sumI*)  
**then have** *fin-fg''*:  $\text{finite } \{d. (\sum (a, b). f \ a * g \ b \ \text{when } d = a + b) \neq 0\}$   
**by** (*auto intro: finite-Sum-any-not-eq-zero-weakenI*)  
**from** *fin-g fin-h* **have** *fin-gh'*:  $\text{finite } \{b + c \mid b \ c. g \ b * h \ c \neq 0\}$  (**is** *finite ?GH'*)  
**by** (*rule finite-mult-not-eq-zero-sumI*)  
**then have** *fin-gh''*:  $\text{finite } \{d. (\sum (b, c). g \ b * h \ c \ \text{when } d = b + c) \neq 0\}$   
**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  $\text{fin-f fin-g fin-h fin-fg''}$ 
  have  $?lhs\ k = (\sum (ab, c). (\sum (a, b). f\ a * g\ b\ \text{when}\ ab = a + b) * h\ c\ \text{when}\ k = ab + c)$ 
    by ( $\text{simp add: prod-fun-unfold-prod}$ )
  also have  $\dots = (\sum (ab, c). (\sum (a, b). f\ a * g\ b * h\ c\ \text{when}\ k = ab + c\ \text{when}\ ab = a + b))$ 
    apply ( $\text{subst Sum-any-left-distrib}$ )
    using  $\text{fin-fg}$  apply ( $\text{simp add: split-def}$ )
    apply ( $\text{subst Sum-any-when-independent [symmetric]}$ )
    apply ( $\text{simp add: when-when when-mult mult-when split-def conj-commute}$ )
    done
  also have  $\dots = (\sum (ab, c, a, b). f\ a * g\ b * h\ c\ \text{when}\ k = ab + c\ \text{when}\ ab = a + b)$ 
    apply ( $\text{subst Sum-any.cartesian-product2 [of (?FG' \times ?H) \times ?FG]}$ )
    apply ( $\text{auto simp add: finite-cartesian-product-iff fin-fg fin-fg' fin-h dest: mult-not-zero}$ )
    done
  also have  $\dots = (\sum (ab, c, a, b). f\ a * g\ b * h\ c\ \text{when}\ k = a + b + c\ \text{when}\ ab = a + b)$ 
    by ( $\text{rule Sum-any.cong}$ ) ( $\text{simp add: split-def when-def}$ )
  also have  $\dots = (\sum (ab, cab). (\text{case cab of } (c, a, b) \Rightarrow f\ a * g\ b * h\ c\ \text{when}\ k = a + b + c)$ 
     $\text{when } ab = (\text{case cab of } (c, a, b) \Rightarrow a + b))$ 
    by ( $\text{simp add: split-def}$ )
  also have  $\dots = (\sum (c, a, b). f\ a * g\ b * h\ c\ \text{when}\ k = a + b + c)$ 
    by ( $\text{simp add: Sum-any-when-dependent-prod-left}$ )
  also have  $\dots = (\sum (bc, cab). (\text{case cab of } (c, a, b) \Rightarrow f\ a * g\ b * h\ c\ \text{when}\ k = a + b + c)$ 
     $\text{when } bc = (\text{case cab of } (c, a, b) \Rightarrow b + c))$ 
    by ( $\text{simp add: Sum-any-when-dependent-prod-left}$ )
  also have  $\dots = (\sum (bc, c, a, b). f\ a * g\ b * h\ c\ \text{when}\ k = a + b + c\ \text{when}\ bc = b + c)$ 
    by ( $\text{simp add: split-def}$ )
  also have  $\dots = (\sum (bc, c, a, b). f\ a * g\ b * h\ c\ \text{when}\ bc = b + c\ \text{when}\ k = a + bc)$ 
    by ( $\text{rule Sum-any.cong}$ ) ( $\text{simp add: split-def when-def ac-simps}$ )
  also have  $\dots = (\sum (a, bc, b, c). f\ a * g\ b * h\ c\ \text{when}\ bc = b + c\ \text{when}\ k = a + bc)$ 
    proof -
      have  $\text{bij } (\lambda(a, d, b, c). (d, c, a, b))$ 
        by ( $\text{auto intro!: bijI injI surjI [of - \lambda(d, c, a, b). (a, d, b, c)] simp add: split-def}$ )
      then show  $?thesis$ 
        by ( $\text{rule Sum-any.reindex-cong}$ )  $\text{auto}$ 
    qed
  also have  $\dots = (\sum (a, bc). (\sum (b, c). f\ a * g\ b * h\ c\ \text{when}\ bc = b + c\ \text{when}\ k = a + bc))$ 

```

```

    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 = ( $\sum l. \text{lookup } f \ l * (\sum q. \text{lookup } g \ q \text{ when } k = l + q)$ )
  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  $\Rightarrow_0$  'b
  show  $a * b = b * a$ 
  proof transfer
    fix f g :: 'a  $\Rightarrow$  'b
    assume fin-f: finite {a. f a  $\neq$  0}
    assume fin-g: finite {b. g b  $\neq$  0}
    show prod-fun f g = prod-fun g f
    proof
      fix k
      have fin1:  $\bigwedge l. \text{finite } \{a. (f \ a \text{ when } k = l + a) \neq 0\}$ 
        using fin-f by auto
      have fin2:  $\bigwedge l. \text{finite } \{b. (g \ b \text{ when } k = l + b) \neq 0\}$ 
        using fin-g by auto
      from fin-f fin-g have finite {(a, b). f a  $\neq$  0  $\wedge$  g b  $\neq$  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  $\Rightarrow$  'b
    assume fin-f: finite {k. f k  $\neq$  0}
    assume fin-g: finite {k. g k  $\neq$  0}
    assume fin-h: finite {k. h k  $\neq$  0}
    show prod-fun ( $\lambda k. f \ k + g \ k$ ) h = ( $\lambda k. \text{prod-fun } f \ h \ k + \text{prod-fun } g \ h \ k$ )
      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  $\Rightarrow_0$  '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  $\Rightarrow_0$  '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**  $single :: 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_0 'b :: zero$   
is  $\lambda k v k'. (v \text{ when } k = k')$   
by *simp*

**lemma** *inj-single [iff]*:  
 $inj (single k)$

**proof** (*rule injI, transfer*)  
fix  $k :: 'b$  and  $a b :: 'a :: zero$   
assume  $(\lambda k'. a \text{ when } k = k') = (\lambda k'. b \text{ when } k = k')$   
then have  $(\lambda k'. a \text{ when } k = k') k = (\lambda k'. b \text{ when } k = k') k$   
by (*rule arg-cong*)  
then show  $a = b$  by *simp*  
qed

**lemma** *lookup-single*:  
 $lookup (single k v) k' = (v \text{ when } k = k')$   
by *transfer rule*

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

**lemma** *lookup-single-not-eq*:  
 $k \neq k' \implies lookup (single k v) k' = 0$   
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]*:  
 $single 0 (\text{numeral } n) = \text{numeral } n$

**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 (λk'. a when k = k') (λk'. b when l = k') = (λk'. a \* b when k + l = k')*

**proof**

**fix** *k'*

**have** *prod-fun (λk'. a when k = k') (λk'. 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 (λk'. a when k = k') (λk'. b when l = k') k' = (λk'. 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  $\Rightarrow$  'b  
 assume *f  $\neq$  0* and *g  $\neq$  0*  
 then show *f \* g  $\neq$  0*  
**proof** *transfer*  
 fix *f g* :: 'a  $\Rightarrow$  'b  
 def *F*  $\equiv$  {*a. f a  $\neq$  0*}  
 moreover def *G*  $\equiv$  {*a. g a  $\neq$  0*}  
 ultimately have [*simp*]:  
 $\bigwedge a. f a \neq 0 \longleftrightarrow a \in F$   
 $\bigwedge b. g b \neq 0 \longleftrightarrow b \in G$   
 by *simp-all*  
 assume *finite {a. f a  $\neq$  0}*  
 then have [*simp*]: *finite F*  
 by *simp*  
 assume *finite {a. g a  $\neq$  0}*  
 then have [*simp*]: *finite G*  
 by *simp*  
 assume *f  $\neq$  ( $\lambda a. 0$ )*  
 then obtain *a* where *f a  $\neq$  0*  
 by (*auto simp add: fun-eq-iff*)  
 assume *g  $\neq$  ( $\lambda b. 0$ )*  
 then obtain *b* where *g b  $\neq$  0*  
 by (*auto simp add: fun-eq-iff*)  
 from *(f a  $\neq$  0)* and *(g b  $\neq$  0)* have *F  $\neq$  {}* and *G  $\neq$  {}*

```

    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
  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*  $\Rightarrow_0$  *b*)  $\Rightarrow$  (*a*  $\Rightarrow_0$  *b*)  $\Rightarrow$  *bool*  
**is** *less-fun*

.

**lift-definition** *less-eq-poly-mapping* :: (*a*  $\Rightarrow_0$  *b*)  $\Rightarrow$  (*a*  $\Rightarrow_0$  *b*)  $\Rightarrow$  *bool*  
**is**  $\lambda f g. \text{less-fun } f g \vee f = g$

.

**instance proof** (*rule class.Orderings.linorder.of-class.intro*)

**show** *class.linorder* (*less-eq* :: (*a*  $\Rightarrow_0$  *b*)  $\Rightarrow$  *bool*) *less*

**proof** (*rule linorder-strictI*, *rule order-strictI*)

**fix** *f g h* :: *a*  $\Rightarrow_0$  *b*

**show**  $f \leq g \longleftrightarrow f < g \vee f = g$

**by** *transfer* (*rule refl*)

**show**  $\neg f < f$

**by** *transfer* (*rule less-fun-irrefl*)

**show**  $f < g \vee f = g \vee g < f$

**proof** *transfer*

**fix** *f g* :: *a*  $\Rightarrow$  *b*

**assume** *finite* {*k. f k*  $\neq 0$ } **and** *finite* {*k. g k*  $\neq 0$ }

**then have** *finite* ({*k. f k*  $\neq 0$ }  $\cup$  {*k. g k*  $\neq 0$ })

**by** *simp*

**moreover have** {*k. f k*  $\neq g k$ }  $\subseteq$  {*k. f k*  $\neq 0$ }  $\cup$  {*k. g k*  $\neq 0$ }

**by** *auto*

**ultimately have** *finite* {*k. f k*  $\neq g k$ }

**by** (*rule rev-finite-subset*)

**then show** *less-fun* *f g*  $\vee f = g \vee$  *less-fun* *g f*

**by** (*rule less-fun-trichotomy*)

**qed**

**assume**  $f < g$  **then show**  $\neg g < f$

**by** *transfer* (*rule less-fun-asy*)

**note**  $\langle f < g \rangle$  **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*  $\Rightarrow$  *b*

**assume** \*: *less-fun* *f g*  $\vee f = g$

{ **assume** *less-fun* *f g*

```

then obtain  $k$  where  $f\ k < g\ k$  ( $\bigwedge k'. k' < k \implies f\ k' = g\ k'$ )
  by (blast elim!: less-funE)
then have  $h\ k + f\ k < h\ k + g\ k$  ( $\bigwedge k'. k' < k \implies h\ k' + f\ k' = h\ k' + g\ k'$ )
  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]:
   $k \notin keys\ f \iff lookup\ f\ k = 0$ 
  by transfer simp

```

```

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

```

**by** *transfer simp*

**lemma** *lookup-eq-zero-in-keys-contradict* [*dest*]:  
 $lookup\ f\ k = 0 \implies \neg k \in keys\ f$   
**by** *simp*

**lemma** *finite-range* [*simp*]: *finite* (*PP-Poly-Mapping.range* *p*)

**proof** *transfer*  
**fix** *f* :: 'b  $\Rightarrow$  'a  
**assume** \*: *finite* {*x*. *f* *x*  $\neq$  0}  
**have** *Set.range* *f* - {0}  $\subseteq$  *f* ' {*x*. *f* *x*  $\neq$  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*]:  
 $k \in keys\ f \implies lookup\ f\ k \in range\ f$   
**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*:  
 $keys\ (f + g) \subseteq keys\ f \cup keys\ g$   
**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:  $\bigwedge k. f k 0 = 0$ 
and hom-plus:  $\bigwedge k. k \in PP\text{-Poly-Mapping.keys } p \cup PP\text{-Poly-Mapping.keys } q$ 
 $\implies f k (PP\text{-Poly-Mapping.lookup } p k + PP\text{-Poly-Mapping.lookup } q k) = f k$ 
 $(PP\text{-Poly-Mapping.lookup } p k) + f k (PP\text{-Poly-Mapping.lookup } q k)$ 
shows
 $(\sum_{k \in PP\text{-Poly-Mapping.keys } (p + q)}. f k (PP\text{-Poly-Mapping.lookup } (p + q) k))$ 
 $=$ 
 $(\sum_{k \in PP\text{-Poly-Mapping.keys } p}. f k (PP\text{-Poly-Mapping.lookup } p k)) +$ 
 $(\sum_{k \in PP\text{-Poly-Mapping.keys } q}. f k (PP\text{-Poly-Mapping.lookup } q k))$ 
(is ?lhs = ?p + ?q)

```

**proof** –

```

let ?A =  $PP\text{-Poly-Mapping.keys } p \cup PP\text{-Poly-Mapping.keys } q$ 
have ?lhs =  $(\sum_{k \in ?A}. f k (PP\text{-Poly-Mapping.lookup } p k + PP\text{-Poly-Mapping.lookup } q k))$ 

```

```

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 ... =  $(\sum_{k \in ?A}. f k (PP\text{-Poly-Mapping.lookup } p k) + f k (PP\text{-Poly-Mapping.lookup } q k))$ 

```

```

by(rule sum.cong)(simp-all add: hom-plus)

```

```

also have ... =  $(\sum_{k \in ?A}. f k (PP\text{-Poly-Mapping.lookup } p k)) + (\sum_{k \in ?A}. f k$ 
 $(PP\text{-Poly-Mapping.lookup } q k))$ 

```

```

(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* ::  $(nat \Rightarrow_0 'a::zero) \Rightarrow nat$

**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':  $\bigwedge f a b. a \notin \text{keys } f \implies b \neq 0 \implies P f \implies P (\text{update } a b f)$ 
  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**  $map\text{-key}\ f\ (map\text{-key}\ g\ p) = map\text{-key}\ (g \circ f)\ p$

**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*:

$map\text{-key}\ (\lambda x. x)\ p = p$

**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*:

$map\text{-key}\ f\ (map\ g\ p) = map\ g\ (map\text{-key}\ f\ p)$   
**by** *transfer (simp add: fun-eq-iff)*

**lemma** *map-key-plus*:

$map\text{-key}\ f\ (p + q) = map\text{-key}\ f\ p + map\text{-key}\ f\ q$

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

**lemma** *keys-map-key*:

$keys (map\text{-}key\ f\ p) = f \text{ -' } keys\ p$

**by** *transfer auto*

**lemma** *map-key-zero [simp]*:

$map\text{-}key\ f\ 0 = 0$

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

**lemma** *map-key-single [simp]*:

$map\text{-}key\ f\ (single\ (f\ k)\ v) = single\ k\ v$

**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**  $p :: 'a \Rightarrow 'b$

**assume**  $*$ : *finite*  $\{x. p\ x \neq 0\}$

**{** **fix**  $x$

**have** *prod-fun*  $(\lambda k'. s\ \text{when}\ 0 = k')\ p\ x =$

$(\sum l :: 'a. \text{if}\ l = 0\ \text{then}\ s\ * (\sum q. p\ q\ \text{when}\ x = q)\ \text{else}\ 0)$

**by**(*auto simp add: prod-fun-def when-def intro: Sum-any.cong simp del:*

*Sum-any.delta*)

**also** **have**  $\dots = (\lambda k. s\ *\ p\ k\ \text{when}\ p\ k \neq 0)\ x$  **by**(*simp add: when-def*)

**also** **note** *calculation* }

**then** **show**  $(\lambda k. s\ *\ p\ k\ \text{when}\ p\ k \neq 0) = \text{prod-fun}\ (\lambda k'. s\ \text{when}\ 0 = k')\ p$

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

**qed**

**lemma** *map-single [simp]*:

$(c = 0 \implies f\ 0 = 0) \implies map\ f\ (single\ x\ c) = single\ x\ (f\ c)$

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

**lemma** *map-eq-zero-iff*:  $map\ f\ p = 0 \iff (\forall k \in keys\ p. f\ (lookup\ p\ k) = 0)$

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

## 28.12 Canonical dense representation of $nat \Rightarrow_0 'a$

**abbreviation** *no-trailing-zeros* ::  $'a :: zero\ list \Rightarrow bool$

**where**

$no\text{-trailing-zeros} \equiv no\text{-trailing}\ (op = 0)$

**lift-definition** *nth* ::  $'a\ list \Rightarrow (nat \Rightarrow_0 'a)::zero)$

**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 ( $\lambda k. k = 0$ ) xs) = nth xs
by (subst eq-commute) (fact nth-strip-while)

lemma nth-eq-iff:
  nth xs = nth ys  $\longleftrightarrow$  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)  $\in$  set (enumerate 0 xs). v  $\neq$  0}
proof transfer
  fix xs :: 'a list
  { fix n
    assume nth-default 0 xs n  $\neq$  0
    then have n < length xs and xs ! n  $\neq$  0
    by (auto simp add: nth-default-def split: if-splits)
    then have (n, xs ! n)  $\in$  {(n, v). (n, v)  $\in$  set (enumerate 0 xs)  $\wedge$  v  $\neq$  0} (is
?x  $\in$  ?A)
    by (auto simp add: in-set-conv-nth enumerate-eq-zip)
    then have fst ?x  $\in$  fst ' ?A
    by blast
    then have n  $\in$  fst ' {(n, v). (n, v)  $\in$  set (enumerate 0 xs)  $\wedge$  v  $\neq$  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 $'a \Rightarrow_0 'b$

**lift-definition** *the-value* :: ( $'a \times 'b$ ) *list*  $\Rightarrow$   $'a \Rightarrow_0 'b::zero$   
**is**  $\lambda xs k. \text{case map-of } xs \ k \ \text{of } None \Rightarrow 0 \mid Some \ v \Rightarrow v$   
**proof** –  
**fix** *xs* :: ( $'a \times 'b$ ) *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* :: ( $'a::linorder \Rightarrow_0 'b::zero$ )  $\Rightarrow$  ( $'a \times 'b$ ) *list*  
**where**  
*items f = List.map* ( $\lambda k. (k, \text{lookup } f \ k)$ ) (*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 *OAL-ist* 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*:  
 $\text{lookup} (\text{the-value } xs) \ k = (\text{case map-of } xs \ k \ \text{of } None \Rightarrow 0 \mid Some \ v \Rightarrow v)$   
**by** *transfer rule*

**lemma** *items-the-value*:  
**assumes** *sorted* (*List.map fst xs*) **and** *distinct* (*List.map fst xs*) **and**  $0 \notin \text{snd } \text{set } xs$

**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 ⇒<sub>0</sub> '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*:

$k \in \text{keys } m \implies f k + g (\text{lookup } m k) < \text{poly-mapping-size } m$

**unfolding** *poly-mapping-size-def*

**proof** transfer

**fix**  $k :: 'a$  **and**  $m :: 'a \Rightarrow 'b$  **and**  $f :: 'a \Rightarrow \text{nat}$  **and**  $g$

**let**  $?keys = \{k. m k \neq 0\}$

**assume** \*: finite  $?keys$   $k \in ?keys$

**then have**  $f k + g (m k) = (\sum k' \in ?keys. f k' + g (m k'))$  when  $k' = k$

by (simp add: sum.delta when-def)

**also have**  $\dots < (\sum k' \in ?keys. \text{Suc } (f k' + g (m k')))$  **using** \*

by (intro sum-strict-mono) (auto simp add: when-def)

**also have**  $\dots \leq g 0 + \dots$  **by** simp

**finally have**  $f k + g (m k) < \dots$

**then show**  $f k + g (m k) < g 0 + (\sum k \mid m k \neq 0. \text{Suc } (f k + g (m k)))$

by simp

**qed**

**lemma** *lookup-le-poly-mapping-size*:

$g (\text{lookup } m k) \leq \text{poly-mapping-size } m$

**proof** (cases  $k \in \text{keys } m$ )

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*:

$k \in \text{keys } m \implies y \leq f k + g (\text{lookup } m k) \implies y < \text{poly-mapping-size } m$

**using** *keys-less-poly-mapping-size* **by** (auto intro: le-less-trans)

**lemma** *poly-mapping-size-estimation2*:

**assumes**  $v \in \text{range } m$  **and**  $y \leq g v$

**shows**  $y < \text{poly-mapping-size } m$

**proof** –

**from** *assms* **obtain**  $k$  **where** \*: lookup  $m k = v$   $v \neq 0$

by transfer blast

**from** \* **have**  $k \in \text{keys } m$

by auto

**then show** ?thesis

**proof** (rule *poly-mapping-size-estimation*)

**from** *assms* \* **have**  $y \leq g (\text{lookup } m k)$

by simp

**then show**  $y \leq f k + g (\text{lookup } m k)$

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*]:  
 $m = m' \implies g\ 0 = g'\ 0 \implies (\bigwedge k. k \in \text{keys } m' \implies f\ k = f'\ k)$   
 $\implies (\bigwedge v. v \in \text{range } m' \implies g\ v = g'\ v)$   
 $\implies \text{poly-mapping-size } f\ g\ m = \text{poly-mapping-size } f'\ g'\ m'$   
**by** (*auto simp add: poly-mapping-size-def intro!: sum.cong*)

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

**definition** *size* = *poly-mapping-size* ( $\lambda\cdot. 0$ ) ( $\lambda\cdot. 0$ )

**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* ::  
 $('a \implies 'b :: \text{zero} \implies 'c :: \text{zero}) \implies ('a \implies_0 'b) \implies ('a \implies_0 'c)$   
**is**  $\lambda f\ p\ k. (\text{if } k \in \text{keys } p \text{ then } f\ k\ (\text{lookup } p\ k) \text{ else } 0)$   
**by** *simp*

**lemma** *mapp-cong* [*fundef-cong*]:  
 $\llbracket m = m'; \bigwedge k. k \in \text{keys } m' \implies f\ k\ (\text{lookup } m'\ k) = f'\ k\ (\text{lookup } m'\ k) \rrbracket$   
 $\implies \text{mapp } f\ m = \text{mapp } f'\ m'$   
**by** *transfer (auto simp add: fun-eq-iff)*

**lemma** *lookup-mapp*:  
*lookup* (*mapp f p*) *k* = (*f k* (*lookup p k*) *when*  $k \in \text{keys } p$ )  
**unfolding** *when-def* **by** *transfer simp*

**lemma** *keys-mapp-subset*:  $\text{keys } (\text{mapp } f\ p) \subseteq \text{keys } p$   
**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 ~~/src/HOL/GCD
begin
```

### 29.1 Abstract type definition

```
typedef (overloaded) 'a mpoly =
  UNIV :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  '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  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a .
```

```
instance ..
```

```
end
```

```
instantiation mpoly :: (monoid-add) monoid-add
begin
```

```
lift-definition plus-mpoly :: 'a mpoly  $\Rightarrow$  'a mpoly  $\Rightarrow$  'a mpoly
  is Groups.plus :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a)  $\Rightarrow$  - .
```

```
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  $\Rightarrow$  'a mpoly  $\Rightarrow$  'a mpoly  
**is** *Groups.minus* :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a)  $\Rightarrow$  - .

**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  $\Rightarrow$  'a mpoly  
**is** *Groups.uminus* :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a)  $\Rightarrow$  - .

**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}  $\Rightarrow$  'a mpoly  $\Rightarrow$  'a mpoly  
**is**  $\lambda a$ . *PP-Poly-Mapping.map* (*Groups.times a*) :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a)  $\Rightarrow$  - .

### 29.4 Multiplicative structure

**instantiation** *mpoly* :: (zero-neq-one) zero-neq-one  
**begin**

**lift-definition** *one-mpoly* :: 'a mpoly  
**is** *1* :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a) .

**instance**

**by** *intro-classes* (*transfer*, *simp*)

**end**

**instantiation** *mpoly* :: (semiring-0) semiring-0  
**begin**

**lift-definition** *times-mpoly* :: 'a mpoly  $\Rightarrow$  'a mpoly  $\Rightarrow$  'a mpoly  
**is** *Groups.times* :: ((nat  $\Rightarrow_0$  nat)  $\Rightarrow_0$  'a)  $\Rightarrow$  - .

**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* ::  $(nat \Rightarrow_0 nat) \Rightarrow 'a::zero \Rightarrow 'a \text{ mpoly}$   
**is** *PP-Poly-Mapping.single* ::  $(nat \Rightarrow_0 nat) \Rightarrow - .$

**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 ⇒<sub>0</sub> 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*  $\Rightarrow$  (*nat*  $\Rightarrow_0$  *nat*)  $\Rightarrow$  '*a*  
**where**  
*coeff* *p* = *PP-Poly-Mapping.lookup* (*mapping-of* *p*)

## 29.8 Insertion morphism

**definition** *insertion-fun-natural* :: (*nat*  $\Rightarrow$  '*a*)  $\Rightarrow$  ((*nat*  $\Rightarrow$  *nat*)  $\Rightarrow$  '*a*)  $\Rightarrow$  '*a*::*comm-semiring-1*  
**where**  
*insertion-fun-natural* *f* *p* = ( $\sum$  *m*. *p* *m* \* ( $\prod$  *v*. *f* *v* ^ *m* *v*))

**definition** *insertion-fun* :: (*nat*  $\Rightarrow$  '*a*)  $\Rightarrow$  ((*nat*  $\Rightarrow_0$  *nat*)  $\Rightarrow$  '*a*)  $\Rightarrow$  '*a*::*comm-semiring-1*  
**where**  
*insertion-fun* *f* *p* = ( $\sum$  *m*. *p* *m* \* ( $\prod$  *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*  $\Rightarrow$  '*a*)  $\Rightarrow$  ((*nat*  $\Rightarrow_0$  *nat*)  $\Rightarrow_0$  '*a*)  $\Rightarrow$  '*a*::*comm-semiring-1*  
 is *insertion-fun* .

**lift-definition** *insertion* :: (*nat*  $\Rightarrow$  '*a*)  $\Rightarrow$  '*a* *mpoly*  $\Rightarrow$  '*a*::*comm-semiring-1*  
 is *insertion-aux* .

**lemma** *aux*:  
*PP-Poly-Mapping.lookup* *f* = ( $\lambda$ -. *0*)  $\longleftrightarrow$  *f* = *0*  
**apply** *transfer* **apply** *simp* **done**

**lemma** *insertion-trivial* [*simp*]:  
*insertion* ( $\lambda$ -. *0*) *p* = *coeff* *p* *0*

**proof** –

{ **fix** *f* :: (*nat*  $\Rightarrow_0$  *nat*)  $\Rightarrow_0$  '*a*  
**have** *insertion-aux* ( $\lambda$ -. *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 ⇒<sub>0</sub> 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** *\**:  $\bigwedge a b. g (a + b) = g a * g b$   
**by** (*simp add: plus-poly-mapping.rep-eq power-add Prod-any.distrib*)  
**have** *bij*: *bij (λ(l, n, m). (m, l, n))*  
**by** (*auto intro!: bijI injI simp add: image-def*)  
**let** *?P = {l. PP-Poly-Mapping.lookup p l ≠ 0}*  
**let** *?Q = {n. PP-Poly-Mapping.lookup q n ≠ 0}*

```

let ?PQ = {l + n | l n. l ∈ PP-Poly-Mapping.keys p ∧ n ∈ PP-Poly-Mapping.keys
q}
have finite {l + n | l n. PP-Poly-Mapping.lookup p l ≠ 0 ∧ PP-Poly-Mapping.lookup
q n ≠ 0}
  by (rule finite-not-eq-zero-sumI) simp-all
then have fin-PQ: finite ?PQ
  by simp
have (∑ m. PP-Poly-Mapping.lookup (p * q) m * g m) =
  (∑ m. (∑ l. PP-Poly-Mapping.lookup p l * (∑ 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 ... = (∑ 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-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 ... = (∑ m. (∑ (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 × ?Q])
  apply (auto dest!: mult-not-zero)
done
also have ... = (∑ (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 × (?P × ?Q)])
  apply (auto dest!: mult-not-zero simp add: fin-PQ)
  apply auto
done
also have ... = (∑ (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 λ(l, n, m). (m, l, n)]) (simp add:
fun-eq-iff)
also have ... = (∑ (l, n). ∑ 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 × ?Q) × ?PQ])
  apply (auto dest!: mult-not-zero simp add: fin-PQ)
  apply auto
done
also have ... = (∑ (l, n). (g l * g n) * (PP-Poly-Mapping.lookup p l * PP-Poly-Mapping.lookup
q n))
  by (simp add: *)
also have ... = (∑ l. ∑ n. (g l * g n) * (PP-Poly-Mapping.lookup p l *
PP-Poly-Mapping.lookup q n))
  apply (subst Sum-any.cartesian-product [of ?P × ?Q])
  apply (auto dest!: mult-not-zero)

```

**done**  
**also have**  $\dots = (\sum l. \sum n. (PP\text{-Poly-Mapping.lookup } p \ l * g \ l) * (PP\text{-Poly-Mapping.lookup } q \ n * g \ n))$   
**by** (*simp add: ac-simps*)  
**also have**  $\dots =$   
 $(\sum m. PP\text{-Poly-Mapping.lookup } p \ m * g \ m) *$   
 $(\sum m. PP\text{-Poly-Mapping.lookup } q \ m * g \ m)$   
**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* ::  $'a::zero \ mpoly \Rightarrow nat \Rightarrow nat$   
**is**  $\lambda p \ v. Max \ (insert \ 0 \ ((\lambda m. PP\text{-Poly-Mapping.lookup } m \ v) \ ' PP\text{-Poly-Mapping.keys } p))$  .

**lift-definition** *total-degree* ::  $'a::zero \ mpoly \Rightarrow nat$   
**is**  $\lambda p. Max \ (insert \ 0 \ ((\lambda m. sum \ (PP\text{-Poly-Mapping.lookup } m) \ (PP\text{-Poly-Mapping.keys } m)) \ ' PP\text{-Poly-Mapping.keys } p))$  .

**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 Monomials

**lemma** *mapping-of-monom* [*simp*]:  
 $mapping-of \ (monom \ m \ a) = PP\text{-Poly-Mapping.single } m \ a$   
**by** (*fact monom.rep-eq*)

Naive construction of monomials

**definition**  $M :: \text{nat list} \Rightarrow 'a::\text{zero} \Rightarrow 'a \text{ mpoly}$   
**where**  
 $M \text{ ms} = \text{monom} (\text{PP-Poly-Mapping.nth ms})$

**declare**  $[[\text{code abort: monom}]]$   
**value**  $M [1,2,3] (2::\text{int}) + M [2,0,1] 3 + M [2,0,1] 7$

## 29.11 Pseudo-division of polynomials

**lemma** *smult-conv-mult*:  $\text{smult } s \text{ } p = \text{monom } 0 \text{ } s * p$   
**by transfer** (*simp add: mult-map-scale-conv-mult*)

**lemma** *smult-monom* [*simp*]:  
**fixes**  $c :: - :: \text{mult-zero}$   
**shows**  $\text{smult } c (\text{monom } x \text{ } c') = \text{monom } x (c * c')$   
**by transfer** *simp*

**lemma** *smult-0* [*simp*]:  
**fixes**  $p :: - :: \text{mult-zero mpoly}$   
**shows**  $\text{smult } 0 \text{ } p = 0$   
**by transfer**(*simp add: map-eq-zero-iff*)

**lemma** *mult-smult-left*:  $\text{smult } s \text{ } p * q = \text{smult } s (p * q)$   
**by**(*simp add: smult-conv-mult mult.assoc*)

**lift-definition** *sdiv* ::  $'a::\text{ring-div} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly}$   
**is**  $\lambda a. \text{PP-Poly-Mapping.map } (\lambda b. b \text{ div } a) :: ((\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a) \Rightarrow -$   
 $.$

‘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”:

$\forall x \text{ } y :: 'a \text{ mpoly}. y \neq 0 \Rightarrow \exists a \text{ } q \text{ } r. \text{smult } a \text{ } x = q * y + r$

The introduction of pseudo-division below generalises HOL/Library/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*  
 $:: 'a::\text{ring-div} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow \text{bool}$   
**where**  
 $\text{pseudo-divmod-rel } a \text{ } x \text{ } y \text{ } q \text{ } r \iff$   
 $\text{smult } a \text{ } x = q * y + r \wedge (\text{if } y = 0 \text{ then } q = 0 \text{ else } r = 0 \vee \text{degree } r < \text{degree } y)$

**definition** *pddiv* ::  $'a::\text{ring-div mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow ('a \times 'a \text{ mpoly})$  (**infixl** *pddiv* 70)  
**where**

$x \text{ pdiv } y = (\text{THE } (a, q). \exists r. \text{pseudo-divmod-rel } a \ x \ y \ q \ r)$

**definition**  $\text{pmod} :: 'a::\text{ring-div mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly}$  (**infixl**  $\text{pmod}$  70)  
**where**

$x \text{ pmod } y = (\text{THE } r. \exists a \ q. \text{pseudo-divmod-rel } a \ x \ y \ q \ r)$

**definition**  $\text{pdivmod} :: 'a::\text{ring-div mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow ('a \times 'a \text{ mpoly}) \times 'a \text{ mpoly}$   
**where**

$\text{pdivmod } p \ q = (p \ \text{pdiv } q, p \ \text{pmod } q)$

**lemma**  $\text{pdiv-code}$ :

$p \ \text{pdiv } q = \text{fst } (\text{pdivmod } p \ q)$   
**by** ( $\text{simp add: pdivmod-def}$ )

**lemma**  $\text{pmod-code}$ :

$p \ \text{pmod } q = \text{snd } (\text{pdivmod } p \ q)$   
**by** ( $\text{simp add: pdivmod-def}$ )

**definition**  $\text{div} :: 'a::\{\text{ring-div,field}\} \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly}$  (**infixl**  $\text{div}$  70)  
**where**

$x \ \text{div } y = (\text{THE } q'. \exists a \ q \ r. (\text{pseudo-divmod-rel } a \ x \ y \ q \ r) \wedge (q' = \text{smult } (\text{inverse } a) \ q))$

**definition**  $\text{mod} :: 'a::\{\text{ring-div,field}\} \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly}$  (**infixl**  $\text{mod}$  70)  
**where**

$x \ \text{mod } y = (\text{THE } r'. \exists a \ q \ r. (\text{pseudo-divmod-rel } a \ x \ y \ q \ r) \wedge (r' = \text{smult } (\text{inverse } a) \ r))$

**definition**  $\text{divmod} :: 'a::\{\text{ring-div,field}\} \text{ mpoly} \Rightarrow 'a \text{ mpoly} \Rightarrow 'a \text{ mpoly} \times 'a \text{ mpoly}$   
**where**

$\text{divmod } p \ q = (p \ \text{div } q, p \ \text{mod } q)$

**lemma**  $\text{div-poly-code}$ :

$p \ \text{div } q = \text{fst } (\text{divmod } p \ q)$   
**by** ( $\text{simp add: divmod-def}$ )

**lemma**  $\text{mod-poly-code}$ :

$p \ \text{mod } q = \text{snd } (\text{divmod } p \ q)$   
**by** ( $\text{simp add: divmod-def}$ )

## 29.12 Primitive poly, etc

**lift-definition**  $\text{coeffs} :: 'a :: \text{zero mpoly} \Rightarrow 'a \ \text{set}$   
**is**  $\text{PP-Poly-Mapping.range} :: ((\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a) \Rightarrow - .$

**lemma**  $\text{finite-coeffs}$  [ $\text{simp}$ ]:  $\text{finite } (\text{coeffs } p)$

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  $\Rightarrow$  bool

**where**

*primitive*  $p \longleftrightarrow \text{Gcd}(\text{coeffs } p) = 1$

**definition** *content-primitive* :: 'a::{ring-div,GCD.Gcd} mpoly  $\Rightarrow$  'a  $\times$  'a mpoly

**where**

*content-primitive*  $p =$   
  let  $d = \text{Gcd}(\text{coeffs } p)$   
  in  $(d, \text{sdiv } d \ p)$

**value** let  $p = M [1,2,3] (4::\text{int}) + M [2,0,4] 6 + M [2,0,5] 8$

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 extension

**lemma** *lookup-Abs-poly-mapping-when-finite*:

**assumes** *finite*  $S$

**shows** *lookup* (*Abs-poly-mapping*  $(\lambda x. f \ x \ \text{when } x \in S)$ ) =  $(\lambda x. f \ x \ \text{when } x \in S)$

**proof** –

**have** *finite*  $\{x. (f \ x \ \text{when } x \in S) \neq 0\}$  **using** *assms* **by** *auto*

**then show** *?thesis* **using** *lookup-Abs-poly-mapping* **by** *fast*

**qed**

**definition** *remove-key*::'a  $\Rightarrow$  ('a  $\Rightarrow_0$  'b::monoid-add)  $\Rightarrow$  ('a  $\Rightarrow_0$  'b) **where**

*remove-key*  $k0 \ f = \text{Abs-poly-mapping} (\lambda k. \text{lookup } f \ k \ \text{when } k \neq k0)$

**lemma** *remove-key-lookup*:

*lookup* (*remove-key*  $k0 \ f$ )  $k = (\text{lookup } f \ k \ \text{when } k \neq k0)$

**unfolding** *remove-key-def* **using** *finite-subset* **by** (*simp add: lookup-Abs-poly-mapping*)

**lemma** *remove-key-keys*: *keys*  $f - \{k\} = \text{keys} (\text{remove-key } k \ f)$  (**is**  $?A = ?B$ )

**proof** (*rule antisym*; *rule subsetI*)

**fix**  $x$  **assume**  $x \in ?A$

```

then show  $x \in ?B$  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  $x \in ?B$ 
  then have lookup (remove-key k f) x  $\neq$  0 by blast
  then show  $x \in ?A$  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  $k' \neq k$ ) = 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  $k' \neq k$ ) + ((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:  $\bigwedge k. (lookup (PP-Poly-Mapping.single v n) k when  $k \neq v$ ) = 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  $P::('a, 'b::monoid-add) poly-mapping \Rightarrow bool$ 
assumes single:  $\bigwedge k v. P (PP-Poly-Mapping.single k v)$ 
and sum: ( $\bigwedge f g k v. P f \Longrightarrow P g \Longrightarrow g = (PP-Poly-Mapping.single k v) \Longrightarrow k \notin keys f \Longrightarrow P (f+g)$ )
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**  $\text{lookup } (f+g) \ x = \text{lookup } g \ x$  **by** (*metis IntI add.left-neutral*  
*assms empty-iff not-in-keys-iff-lookup-eq-zero plus-poly-mapping.rep-eq*)  
**then show**  $x \in \text{keys } (f+g)$  **using**  $(x \in \text{keys } g)$  **by** (*metis not-in-keys-iff-lookup-eq-zero*)  
**qed**  
**ultimately show**  $\text{keys } f \cup \text{keys } g \subseteq \text{keys } (f+g)$  **by** *simp*  
**next**  
**show**  $\text{keys } (f + g) \subseteq \text{keys } f \cup \text{keys } g$  **by** (*simp add: keys-add-subset*)  
**qed**

**lemma** *fun-when:*

$f \ 0 = 0 \implies f \ (a \ \text{when } P) = (f \ a \ \text{when } P)$  **by** (*simp add: when-def*)

## 31 MPoly extension

**lemma** *coeff-all-0:*  $(\bigwedge m. \text{coeff } p \ m = 0) \implies p=0$

**by** (*metis aux coeff-def mapping-of-inject zero-mpoly.rep-eq*)

**definition** *vars::'a::zero mpoly  $\Rightarrow$  nat set* **where**

*vars*  $p = \text{UNION } (\text{keys } (\text{mapping-of } p)) \ (\lambda m. \text{keys } m)$

**lemma** *vars-finite:* *finite (vars p)* **unfolding** *vars-def* **by** *auto*

**lemma** *vars-monom-single:*  $\text{vars } (\text{monom } (PP\text{-Poly-Mapping.single } v \ k) \ a) \subseteq \{v\}$

**proof**

**fix**  $w$  **assume**  $w \in \text{vars } (\text{monom } (PP\text{-Poly-Mapping.single } v \ k) \ a)$

**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**  $w \in \{v\}$  **by** *auto*

**qed**

**lemma** *vars-monom-keys:*

**assumes**  $a \neq 0$

**shows**  $\text{vars } (\text{monom } m \ a) = \text{keys } m$

**proof** (*rule antisym; rule subsetI*)

**fix**  $w$  **assume**  $w \in \text{vars } (\text{monom } m \ a)$

**then have**  $\text{lookup } m \ w \neq 0$  **using** *vars-def* **by** (*metis UN-E lookup-eq-zero-in-keys-contradict*  
*lookup-single-not-eq monom.rep-eq*)

**then show**  $w \in \text{keys } m$  **by** (*meson lookup-not-eq-zero-eq-in-keys*)

**next**

**fix**  $w$  **assume**  $w \in \text{keys } m$

**then have**  $\text{lookup } m \ w \neq 0$  **by** (*meson lookup-not-eq-zero-eq-in-keys*)

**then show**  $w \in \text{vars } (\text{monom } m \ a)$  **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**  $\text{vars } (\text{monom } m \ a) \subseteq \text{keys } m$

**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  $\vee$  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**  $m=a + b$   $a \in \text{keys } (mapping-of\ p)$   $b \in \text{keys } (mapping-of\ q)$   
**using** *keys-mult by blast*  
**then have**  $x \in \text{keys } a \cup \text{keys } b$  **by** (*metis  $\langle x \in \text{keys } m \rangle$  keys-add-subset subsetCE*)  
**then show**  $x \in \text{vars } p \cup \text{vars } q$  **unfolding** *vars-def*  
**using**  $\langle a \in \text{keys } (mapping-of\ p) \rangle \langle b \in \text{keys } (mapping-of\ q) \rangle$  **by** *blast*  
**qed**

**lemma** *vars-add-monom:*  
**assumes**  $p2 = \text{monom } m\ a$   $m \notin \text{keys } (mapping-of\ p1)$   
**shows**  $\text{vars } (p1 + p2) = \text{vars } p1 \cup \text{vars } p2$   
**proof** –  
**have**  $\text{keys } (mapping-of\ p2) \subseteq \{m\}$  **using** *monom-def keys-single assms by auto*  
**have**  $\text{keys } (mapping-of\ (p1+p2)) = \text{keys } (mapping-of\ p1) \cup \text{keys } (mapping-of\ p2)$   
**using** *keys-add by (metis Int-insert-right-iff0  $\langle \text{keys } (mapping-of\ p2) \subseteq \{m\} \rangle$  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  $\implies$  vars  $(\sum_{m \in S} f\ m) \subseteq (\bigcup_{m \in S} \text{vars } (f\ m))$*   
**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**  $\text{vars } (\text{sum } f\ (\text{insert } s\ S)) = \text{vars } (f\ s + \text{sum } f\ S)$  **by** (*metis sum.insert*)  
**also have**  $\dots \subseteq \text{vars } (f\ s) \cup \text{vars } (\text{sum } f\ S)$  **by** (*simp add: vars-add*)  
**also have**  $\dots \subseteq (\bigcup_{m \in \text{insert } s\ S} \text{vars } (f\ m))$  **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  $\iff$  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**  $\text{card } (\text{keys } (\text{mapping-of } p)) \leq 1$   
**obtains**  $m$  **where**  $p = \text{monom } m \text{ (coeff } p \text{ } m)$   
**proof** ( $\text{cases } \text{keys } (\text{mapping-of } p) = \{\}$ )  
  **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**  $\text{keys } (\text{mapping-of } p) = \{m\}$  **using** *assms* **by** (*metis One-nat-def Suc-leI antisym card-0-eq card-eq-SucD finite-keys neq0-conv*)  
  **have**  $p = \text{monom } m \text{ (coeff } p \text{ } m)$   
  **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  $\Rightarrow_0$  nat)  $\Rightarrow$  'a::zero mpoly  $\Rightarrow$  'a mpoly* **where**  
*remove-term*  $m0 \text{ } p = \text{MPoly } (\text{Abs-poly-mapping } (\lambda m. \text{coeff } p \text{ } m \text{ when } m \neq m0))$

**lemma** *remove-term-coeff*:  $\text{coeff } (\text{remove-term } m0 \text{ } p) \text{ } m = (\text{coeff } p \text{ } m \text{ when } m \neq m0)$   
**proof** –  
  **have**  $\{m. (\text{coeff } p \text{ } m \text{ when } m \neq m0) \neq 0\} \subseteq \{m. \text{coeff } p \text{ } m \neq 0\}$  **by** *auto*  
  **then have** *finite*  $\{m. (\text{coeff } p \text{ } m \text{ when } m \neq m0) \neq 0\}$  **unfolding** *coeff-def*  
**using** *finite-subset* **by** *auto*  
  **then have** *lookup*  $(\text{Abs-poly-mapping } (\lambda m. \text{coeff } p \text{ } m \text{ when } m \neq m0)) \text{ } m = (\text{coeff } p \text{ } m \text{ when } m \neq m0)$  **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*:  $m \in \text{keys } (\text{mapping-of } p) \iff \text{coeff } p \text{ } m \neq 0$  **by** (*simp add: coeff-def*)

**lemma** *remove-term-keys*:  
**shows**  $\text{keys } (\text{mapping-of } p) - \{m\} = \text{keys } (\text{mapping-of } (\text{remove-term } m \text{ } p))$  (**is**  $?A = ?B$ )  
**proof**  
  **show**  $?A \subseteq ?B$   
  **proof**  
    **fix**  $m'$  **assume**  $m' \in ?A$

```

    then show  $m' \in ?B$  by (simp add: coeff-keys remove-term-coeff)
  qed
  show  $?B \subseteq ?A$ 
  proof
    fix  $m'$  assume  $m' \in ?B$ 
    then show  $m' \in ?A$  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  $p$  =  $(\lambda m'. (\text{coeff } p \ m' \text{ when } m' \neq m) + ((\text{coeff } p \ m) \text{ when } m'=m))$ 
  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:  $\bigwedge m a. P (\text{monom } m \ a)$ 
  and sum:  $(\bigwedge p1 \ p2 \ m \ a. P \ p1 \ \Longrightarrow \ P \ p2 \ \Longrightarrow \ p2 = (\text{monom } m \ a) \ \Longrightarrow \ m \notin \text{keys}$ 
  (mapping-of  $p1$ )  $\Longrightarrow \ P (p1+p2))$ 
  shows  $P \ p$  using assms
  using poly-mapping-induct[of  $\lambda p :: (\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a. P (\text{MPoly } p)$ ] 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  $v \ n0$ )  $a \ ^n = \text{monom}$ 
(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  $f \ (\lambda m. (a \text{ when } (\text{PP-Poly-Mapping.single}$ 
 $(v::\text{nat}) \ (n::\text{nat})) = m)) = a * f \ v \ ^n$  (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 =  $(\sum m. (a \text{ when } \text{PP-Poly-Mapping.single } v \ n = m) * (\prod v. f \ v \ ^{\text{lookup } m \ v}))$ 
  unfolding insertion-fun-def by metis
  have  $\forall m. m \neq \text{PP-Poly-Mapping.single } v \ n \longrightarrow (a \text{ when } \text{PP-Poly-Mapping.single}$ 
 $v \ n = m) = 0$  by simp

```

**have**  $(\sum m \in \{PP\text{-Poly-Mapping.single } v \ n\}. (a \text{ when } PP\text{-Poly-Mapping.single } v \ n = m) * (\prod v. f \ v \ \wedge \ \text{lookup } m \ v)) = ?i$   
**unfolding** *1 when-mult unfolding when-def by auto*  
**then have**  $2: ?i = a * (\prod va. f \ va \ \wedge \ \text{lookup } (PP\text{-Poly-Mapping.single } v \ n) \ va)$   
**unfolding** *setsum-single[of  $\lambda m. (a \text{ when } PP\text{-Poly-Mapping.single } v \ n = m)$ ]*  
 $* (\prod v. f \ v \ \wedge \ \text{lookup } m \ v) \ PP\text{-Poly-Mapping.single } k \ v]$   
**by auto**  
**have**  $\forall v0. v0 \neq v \longrightarrow \text{lookup } (PP\text{-Poly-Mapping.single } v \ n) \ v0 = 0$  **by** *(simp add: lookup-single-not-eq)*  
**then have**  $\forall va. va \neq v \longrightarrow f \ va \ \wedge \ \text{lookup } (PP\text{-Poly-Mapping.single } v \ n) \ va = 1$   
**by simp**  
**then have**  $a * (\prod va \in \{v\}. f \ va \ \wedge \ \text{lookup } (PP\text{-Poly-Mapping.single } v \ n) \ va) =$   
 $?i$  **unfolding** *2*  
**using** *Prod-any.expand-superset[of  $\{v\}$ ]*  $\lambda va. f \ va \ \wedge \ \text{lookup } (PP\text{-Poly-Mapping.single } v \ n) \ va, \text{ simplified}]$   
**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**  $p::(nat \Rightarrow_0 \ nat) \Rightarrow 'a::comm-ring-1$   
**assumes**  $\bigwedge m \ v. p \ m \neq 0 \implies \text{lookup } m \ v \neq 0 \implies f \ v = g \ v$   
**shows** *insertion-fun f p = insertion-fun g p*  
**proof** –  
 $\{$   
**fix**  $m::nat \Rightarrow_0 \ nat$   
**assume**  $p \ m \neq 0$   
**then have**  $(\prod v. f \ v \ \wedge \ \text{lookup } m \ v) = (\prod v. g \ v \ \wedge \ \text{lookup } m \ v)$   
**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**  $p::(nat \Rightarrow_0 \ nat) \Rightarrow_0 'a::comm-ring-1$   
**assumes**  $\bigwedge m \ v. \text{lookup } p \ m \neq 0 \implies \text{lookup } m \ v \neq 0 \implies f \ v = g \ v$   
**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  $\bigwedge v. v \in \text{vars } p \implies f v = g v$ 
shows  $\text{insertion } f p = \text{insertion } g p$ 
proof -
  {
    fix  $m v$  assume  $\text{lookup } (\text{mapping-of } p) m \neq 0 \text{ lookup } m v \neq 0$ 
    then have  $v \in \text{vars } p$  unfolding  $\text{vars-def}$  by  $(\text{meson } UN-I \text{ lookup-not-eq-zero-eq-in-keys})$ 
    then have  $f v = g v$  using  $\text{assms}$  by  $\text{auto}$ 
  }
then show  $?thesis$ 
unfolding  $\text{insertion-def}$  using  $\text{insertion-aux-irrelevant-vars}$ [of  $\text{mapping-of } p$ ]

  by  $(\text{metis } \text{insertion.rep-eq } \text{insertion-def})$ 
qed

```

## 32 Nested MPoly

**definition**  $\text{reduce-nested-mpoly}::'a::\text{comm-ring-1 } \text{mpoly } \text{mpoly} \Rightarrow 'a \text{ mpoly}$  **where**  
 $\text{reduce-nested-mpoly } pp = \text{insertion } (\lambda v. \text{monom } (PP\text{-Poly-Mapping.single } v \ 1))$   
 $1) pp$

**lemma**  $\text{reduce-nested-mpoly-sum}$ :  
**fixes**  $p1::'a::\text{comm-ring-1 } \text{mpoly } \text{mpoly}$   
**shows**  $\text{reduce-nested-mpoly } (p1 + p2) = \text{reduce-nested-mpoly } p1 + \text{reduce-nested-mpoly } p2$   
**by**  $(\text{simp add: insertion-add } \text{reduce-nested-mpoly-def})$

**lemma**  $\text{reduce-nested-mpoly-prod}$ :  
**fixes**  $p1::'a::\text{comm-ring-1 } \text{mpoly } \text{mpoly}$   
**shows**  $\text{reduce-nested-mpoly } (p1 * p2) = \text{reduce-nested-mpoly } p1 * \text{reduce-nested-mpoly } p2$   
**by**  $(\text{simp add: insertion-mult } \text{reduce-nested-mpoly-def})$

**lemma**  $\text{reduce-nested-mpoly-0}$ :  
**shows**  $\text{reduce-nested-mpoly } 0 = 0$  **by**  $(\text{simp add: } \text{reduce-nested-mpoly-def})$

**lemma**  $\text{insertion-nested-poly}$ :  
**fixes**  $pp::'a::\text{comm-ring-1 } \text{mpoly } \text{mpoly}$   
**shows**  $\text{insertion } f (\text{insertion } (\lambda v. \text{monom } 0 (f v)) pp) = \text{insertion } f (\text{reduce-nested-mpoly } pp)$   
**proof**  $(\text{induction } pp \text{ rule:mpoly-induct})$   
**case**  $(\text{monom } m a)$   
**then show**  $?case$   
**proof**  $(\text{induction } m \text{ arbitrary:a rule:poly-mapping-induct})$   
**case**  $(\text{single } v n)$   
**show**  $?case$  **unfolding**  $\text{reduce-nested-mpoly-def}$   
**apply**  $(\text{simp add: insertion-mult monom-pow})$   
**using**  $\text{monom-pow}$ [of  $0 \ 0 \ f \ v \ n$ ] **apply**  $\text{simp}$   
**using**  $\text{insertion-single}$ [of  $f \ 0 \ 0$ ] **by**  $\text{auto}$   
**next**

```

      case (sum m1 m2 k v)
    then have insertion f (insertion (λv. 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**  $extract\text{-}var::'a::comm\text{-}ring\text{-}1\ mpoly \Rightarrow nat \Rightarrow 'a::comm\text{-}ring\text{-}1\ mpoly$   
 $mpoly$  where  
 $extract\text{-}var\ p\ v = (\sum m. monom (remove\text{-}key\ v\ m) (monom (PP\text{-}Poly\text{-}Mapping.single$   
 $v (lookup\ m\ v)) (coeff\ p\ m)))$

**lemma**  $extract\text{-}var\text{-}finite\text{-}set$ :  
**assumes**  $\{m'. coeff\ p\ m' \neq 0\} \subseteq S$   
**assumes**  $finite\ S$   
**shows**  $extract\text{-}var\ p\ v = (\sum m \in S. monom (remove\text{-}key\ v\ m) (monom (PP\text{-}Poly\text{-}Mapping.single$   
 $v (lookup\ m\ v)) (coeff\ p\ m)))$   
**proof** –  
 {  
**fix**  $m'$  **assume**  $coeff\ p\ m' = 0$   
**then have**  $monom (remove\text{-}key\ v\ m') (monom (PP\text{-}Poly\text{-}Mapping.single\ v$   
 $(lookup\ m'\ v)) (coeff\ p\ m')) = 0$   
**using**  $monom.abs\text{-}eq\ monom\text{-}zero\ single\text{-}zero$  **by**  $metis$   
 }  
**then have**  $0:\{a. monom (remove\text{-}key\ v\ a) (monom (PP\text{-}Poly\text{-}Mapping.single\ v$   
 $(lookup\ a\ v)) (coeff\ p\ a)) \neq 0\} \subseteq S$   
**using**  $\{m'. coeff\ p\ m' \neq 0\} \subseteq S$  **by**  $fastforce$   
**then show**  $?thesis$   
**unfolding**  $extract\text{-}var\text{-}def$  **using**  $Sum\text{-}any.expand\text{-}superset [OF \langle finite\ S \rangle 0]$   
**by**  $metis$   
**qed**

**lemma**  $extract\text{-}var\text{-}non\text{-}zero\text{-}coeff$ :  $extract\text{-}var\ p\ v = (\sum m \in \{m'. coeff\ p\ m' \neq 0\}.$   
 $monom (remove\text{-}key\ v\ m) (monom (PP\text{-}Poly\text{-}Mapping.single\ v (lookup\ m\ v)) (coeff$   
 $p\ m)))$   
**using**  $extract\text{-}var\text{-}finite\text{-}set\ coeff\text{-}def\ finite\text{-}lookup\ order\text{-}refl$  **by**  $(metis (no\text{-}types,$   
 $lifting) Collect\text{-}cong\ sum.cong)$

**lemma**  $extract\text{-}var\text{-}sum$ :  $extract\text{-}var (p+p')\ v = extract\text{-}var\ p\ v + extract\text{-}var\ p'\ v$   
**proof** –  
**def**  $S == \{m. coeff\ p\ m \neq 0\} \cup \{m. coeff\ p'\ m \neq 0\} \cup \{m. coeff (p+p')\ m \neq$   
 $0\}$   
**have**  $subsets:\{m. coeff\ p\ m \neq 0\} \subseteq S \{m. coeff\ p'\ m \neq 0\} \subseteq S \{m. coeff (p+p')$   
 $m \neq 0\} \subseteq S$  **unfolding**  $S\text{-}def$  **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**  $p::'a::\text{comm-ring-1}$  *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*  $p$   $v$ )  $\subseteq$  *vars*  $p$   
**proof**  
  **have** *finite*  $\{m'. \text{coeff } p \ m' \neq 0\}$  **by** (*simp add: coeff-def*)  
  **fix**  $x$  **assume**  $x \in \text{vars} (\text{extract-var } p \ v)$   
  **then have**  $x \in \text{vars} (\sum m \in \{m'. \text{coeff } p \ m' \neq 0\}. \text{monom } (\text{remove-key } v \ m)$   
(*monom* (*PP-Poly-Mapping.single*  $v$  (*lookup*  $m$   $v$ )) (*coeff*  $p$   $m$ )))  
  **unfolding** *extract-var-non-zero-coeff* **by** *metis*  
  **then have**  $x \in (\bigcup m \in \{m'. \text{coeff } p \ m' \neq 0\}. \text{vars } (\text{monom } (\text{remove-key } v \ m)$   
(*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**  $m \in \{m'. \text{coeff } p \ m' \neq 0\}$   $x \in \text{vars} (\text{monom } (\text{remove-key}$   
 $v \ m)$  (*monom* (*PP-Poly-Mapping.single*  $v$  (*lookup*  $m$   $v$ )) (*coeff*  $p$   $m$ )))  
  **by** *blast*  
  **show**  $x \in \text{vars } p$  **by** (*metis* (*mono-tags, lifting*) *DiffD1 UN-I ‹m ∈ {m'. coeff p*  
 $m' \neq 0\}$ )

$\langle x \in \text{vars } (\text{monom } (\text{remove-key } v \ m) \ (\text{monom } (\text{PP-Poly-Mapping.single } v \ (\text{lookup } m \ v)) \ (\text{coeff } p \ m)))) \rangle$   
 $\text{coeff-keys mem-Collect-eq remove-key-keys subsetCE vars-def vars-monom-subset}$   
**qed**

**lemma** *v-not-in-vars-extract-var*:  $v \notin \text{vars } (\text{extract-var } p \ v)$

**proof** –

**have**  $\text{finite } \{m'. \text{coeff } p \ m' \neq 0\}$  **by** (*simp add: coeff-def*)  
**have**  $\bigwedge m. m \in \{m'. \text{coeff } p \ m' \neq 0\} \implies v \notin \text{vars } (\text{monom } (\text{remove-key } v \ m) \ (\text{monom } (\text{PP-Poly-Mapping.single } v \ (\text{lookup } m \ v)) \ (\text{coeff } p \ m)))$   
**by** (*metis Diff-iff remove-key-keys singletonI subsetCE vars-monom-subset*)  
**then have**  $v \notin (\bigcup m \in \{m'. \text{coeff } p \ m' \neq 0\}. \text{vars } (\text{monom } (\text{remove-key } v \ m) \ (\text{monom } (\text{PP-Poly-Mapping.single } v \ (\text{lookup } m \ v)) \ (\text{coeff } p \ m))))$   
**by** *simp*  
**then show** *?thesis*  
**unfolding** *extract-var-non-zero-coeff* **using** *vars-setsum[OF \langle finite {m'. coeff p m' \neq 0} \rangle]* **by** *blast*  
**qed**

**lemma** *vars-coeff-extract-var*:  $\text{vars } (\text{coeff } (\text{extract-var } p \ v) \ j) \subseteq \{v\}$

**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**  $\text{replace-coeff } f \ p = \text{MPoly } (\text{Abs-poly-mapping } (\lambda m. f \ (\text{lookup } (\text{mapping-of } p) \ m)))$

**lemma** *coeff-replace-coeff*:

**assumes**  $f \ 0 = 0$

**shows**  $\text{coeff } (\text{replace-coeff } f \ p) \ m = f \ (\text{coeff } p \ m)$

**proof** –

**have**  $0 : \text{finite } \{m. f \ (\text{lookup } (\text{mapping-of } p) \ m) \neq 0\}$

**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**  $\text{replace-coeff } f \text{ (monom } m \ a) = \text{monom } m \ (f \ a)$   
**unfolding**  $\text{replace-coeff-def}$   
**unfolding**  $\text{mapping-of-inject[symmetric] lookup-inject[symmetric]}$  **apply** ( $\text{rule HOL.ext}$ )  
**unfolding**  $\text{lookup-single mapping-of-monom fun-when[of } f, \text{OF } \langle f \ 0 = 0 \rangle]$   
**by** ( $\text{metis coeff-def coeff-monom lookup-single lookup-single-not-eq monom.abs-eq single.abs-eq}$ )

**lemma**  $\text{replace-coeff-add}$ :

**assumes**  $f \ 0 = 0$

**assumes**  $\bigwedge a \ b. f \ (a+b) = f \ a + f \ b$

**shows**  $\text{replace-coeff } f \ (p1 + p2) = \text{replace-coeff } f \ p1 + \text{replace-coeff } f \ p2$

**proof** –

**have**  $\text{finite } \{m. f \ (\text{lookup } (\text{mapping-of } p1) \ m) \neq 0\}$

$\text{finite } \{m. f \ (\text{lookup } (\text{mapping-of } p2) \ m) \neq 0\}$

**unfolding**  $\text{coeff-def[symmetric]}$  **by** ( $\text{metis (mono-tags, lifting) Collect-mono assms(1) coeff-def finite-lookup finite-subset}+$ )

**then show**  $?thesis$

**unfolding**  $\text{replace-coeff-def PP-MPoly.plus-mpoly.rep-eq}$  **unfolding**  $\text{PP-Poly-Mapping.plus-poly-mapping.rep}$

**unfolding**  $\text{assms(2) PP-MPoly.plus-mpoly.abs-eq}$  **using**  $\text{PP-Poly-Mapping.plus-poly-mapping.abs-eq[unfold eq-onp-def]}$  **by**  $\text{fastforce}$

**qed**

**lemma**  $\text{insertion-replace-coeff}$ :

**fixes**  $pp::'a::\text{comm-ring-1 mpoly mpoly}$

**shows**  $\text{insertion } f \ (\text{replace-coeff } (\text{insertion } f) \ pp) = \text{insertion } f \ (\text{reduce-nested-mpoly } pp)$

**proof** ( $\text{induction } pp \ \text{rule:mpoly-induct}$ )

**case** ( $\text{monom } m \ a$ )

**then show**  $?case$

**proof** ( $\text{induction } m \ \text{arbitrary:a rule:poly-mapping-induct}$ )

**case** ( $\text{single } v \ n$ )

**show**  $?case$  **unfolding**  $\text{reduce-nested-mpoly-def}$  **unfolding**  $\text{replace-coeff-monom[of insertion } f, \text{OF insertion-zero}]$

$\text{insertion-single insertion-mult}$  **using**  $\text{insertion-single}$  **by** ( $\text{simp add: monom-pow}$ )

**next**

**case** ( $\text{sum } m1 \ m2 \ k \ v$ )

**have**  $\text{replace-coeff } (\text{insertion } f) \ (\text{monom } m1 \ a * \text{monom } m2 \ 1) = \text{replace-coeff } (\text{insertion } f) \ (\text{monom } m1 \ a) * \text{replace-coeff } (\text{insertion } f) \ (\text{monom } m2 \ 1)$

**by** ( $\text{simp add: mult-monom replace-coeff-monom}$ )

**then have**  $\text{insertion } f \ (\text{replace-coeff } (\text{insertion } f) \ (\text{monom } m1 \ a * \text{monom } m2 \ 1)) = \text{insertion } f \ (\text{reduce-nested-mpoly } (\text{monom } m1 \ a * \text{monom } m2 \ 1))$

**unfolding**  $\text{reduce-nested-mpoly-prod insertion-mult}$

**by** ( $\text{simp add: insertion-mult sum.IH(1) sum.IH(2)}$ )

**then show**  $?case$  **using**  $\text{mult-monom[of } m1 \ m2 \ 1]$  **by**  $\text{auto}$

**qed**

**next**

**case** ( $\text{sum } p1 \ p2 \ m \ a$ )

**then show**  $?case$  **using**  $\text{reduce-nested-mpoly-sum insertion-add}$

*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**  $\text{replace-coeff } (\text{insertion } f) (\text{extract-var } p v) = \text{replace-coeff } (\text{insertion } g) (\text{extract-var } p v)$

**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**  $\text{vars } (\text{replace-coeff } f p) \subseteq \text{vars } p$

**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*  $N f = (\exists p. \text{vars } p \subseteq N \wedge (\forall x. \text{insertion } x p = f x))$

**lemma** *polyfunI*:  $(\bigwedge P. (\bigwedge p. \text{vars } p \subseteq N \implies (\bigwedge x. \text{insertion } x p = f x) \implies P) \implies P) \implies \text{polyfun } N f$

**unfolding** *polyfun-def* **by** *metis*

**lemma** *polyfun-subset*:  $N \subseteq N' \implies \text{polyfun } N f \implies \text{polyfun } N' f$

**unfolding** *polyfun-def* **by** *blast*

**lemma** *polyfun-const*:  $\text{polyfun } N (\lambda-. c)$

**proof** –

**have**  $\bigwedge x. \text{insertion } x (\text{monom } 0 c) = c$  **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**  $\text{polyfun } N f \text{ polyfun } N g$

**shows**  $\text{polyfun } N (\lambda x. f x + g x)$

**proof** –

**obtain**  $p1 p2$  **where**  $\text{vars } p1 \subseteq N \forall x. \text{insertion } x p1 = f x$

$\text{vars } p2 \subseteq N \forall x. \text{insertion } x p2 = g x$

**using** *polyfun-def assms* **by** *metis*

**then have**  $\text{vars } (p1 + p2) \subseteq N \forall x. \text{insertion } x (p1 + p2) = f x + g x$

**using** *vars-add* **using** *Un-iff subsetCE subsetI* **apply** *blast*  
**by** (*simp add:  $\langle \forall x. \text{insertion } x \ p1 = f \ x \rangle \langle \forall x. \text{insertion } x \ p2 = g \ x \rangle$*  *insertion-add*)  
**then show** *?thesis* **using** *polyfun-def* **by** *blast*  
**qed**

**lemma** *polyfun-mult*:  
**assumes** *polyfun N f polyfun N g*  
**shows** *polyfun N  $(\lambda x. f \ x * g \ x)$*   
**proof** –  
**obtain** *p1 p2* **where** *vars p1  $\subseteq N \forall x. \text{insertion } x \ p1 = f \ x$*   
*vars p2  $\subseteq N \forall x. \text{insertion } x \ p2 = g \ x$*   
**using** *polyfun-def assms* **by** *metis*  
**then have** *vars  $(p1 * p2) \subseteq N \forall x. \text{insertion } x \ (p1 * p2) = f \ x * g \ x$*   
**using** *vars-mult* **using** *Un-iff subsetCE subsetI* **apply** *blast*  
**by** (*simp add:  $\langle \forall x. \text{insertion } x \ p1 = f \ x \rangle \langle \forall x. \text{insertion } x \ p2 = g \ x \rangle$*  *insertion-mult*)  
**then show** *?thesis* **using** *polyfun-def* **by** *blast*  
**qed**

**lemma** *polyfun-Sum*:  
**assumes** *finite I*  
**assumes**  $\bigwedge i. i \in I \implies \text{polyfun } N \ (f \ i)$   
**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**  $\bigwedge i. i \in I \implies \text{polyfun } N \ (f \ i)$   
**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**  $\forall f. \text{insertion } f \ (\text{monom } (PP\text{-Poly-Mapping.single } i \ 1) \ 1) = f \ i$  **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**  $\bigwedge x. (A \ x) \in \text{carrier}_m \ n \ n$

**assumes**  $\bigwedge x i j. i < n \implies j < n \implies \text{polyfun } N (\lambda x. (A x) \$\$ (i,j))$   
**shows**  $\text{polyfun } N (\lambda x. \det (A x))$   
**proof** –  
{  
  **fix**  $p$  **assume**  $p \in \{p. p \text{ permutes } \{0..<n\}\}$   
  **then have**  $p \text{ permutes } \{0..<n\}$  **by** *auto*  
  **then have**  $\bigwedge x. x < n \implies p x < n$  **using** *permutes-in-image* **by** *auto*  
  **then have**  $\text{polyfun } N (\lambda x. \prod i = 0..<n. A x \$\$ (i, p i))$   
  **using** *polyfun-Prod*[*of*  $\{0..<n\}$   $N \lambda i x. A x \$\$ (i, p i)$ ] *assms* **by** *simp*  
  **then have**  $\text{polyfun } N (\lambda x. \text{signof } p * (\prod i = 0..<n. A x \$\$ (i, p i)))$  **using**  
*polyfun-const polyfun-mult* **by** *blast*  
}  
**moreover have** *finite*  $\{i. i \text{ permutes } \{0..<n\}\}$  **by** (*simp add: finite-permutations*)  
**ultimately show** *?thesis unfolding det-def'*[*OF assms(1)*]  
  **using** *polyfun-Sum*[*OF*  $\{i. i \text{ permutes } \{0..<n\}\}$ , *of*  $N \lambda p x. \text{signof } p * (\prod i = 0..<n. A x \$\$ (i, p i))$ ]  
  **by** *blast*  
**qed**

**lemma** *polyfun-extract-matrix*:  
**assumes**  $i < m \ j < n$   
**shows**  $\text{polyfun } \{..<a + (m * n + c)\} (\lambda f. \text{extract-matrix } (\lambda i. f (i + a)) m n \$\$ (i,j))$   
**unfolding** *index-extract-matrix*[*OF assms*] **apply** (*rule polyfun-single*) **using** *two-digit-le*[*OF assms*] **by** *simp*

**lemma** *polyfun-mat-mult-vec*:  
**assumes**  $\bigwedge x. v x \in \text{carrier}_v n$   
**assumes**  $\bigwedge j. j < n \implies \text{polyfun } N (\lambda x. v x \$ j)$   
**assumes**  $\bigwedge x. A x \in \text{carrier}_m m n$   
**assumes**  $\bigwedge i j. i < m \implies j < n \implies \text{polyfun } N (\lambda x. A x \$\$ (i,j))$   
**assumes**  $j < m$   
**shows**  $\text{polyfun } N (\lambda x. ((A x) \otimes_{mv} (v x)) \$ j)$   
**proof** –  
  **have**  $\bigwedge x. j < \text{dim}_r (A x)$  **using**  $\langle j < m \rangle$  *assms(3)* *mat-carrierD(1)* **by** *force*  
  **have**  $\bigwedge x. n = \text{dim}_v (v x)$  **using** *assms(1)* *vec-elemsD* **by** *fastforce*  
  {  
    **fix**  $i$  **assume**  $i \in \{0..<n\}$   
    **then have**  $i < n$  **by** *auto*  
    {  
      **fix**  $x$   
      **have**  $i < \text{dim}_v (v x)$  **using** *assms(1)* *vec-elemsD*  $\langle i < n \rangle$  **by** *fastforce*  
      **have**  $j < \text{dim}_r (A x)$  **using**  $\langle j < m \rangle$  *assms(3)* *mat-carrierD(1)* **by** *force*  
      **have**  $\text{dim}_c (A x) = \text{dim}_v (v x)$  **by** (*metis assms(1) assms(3) mat-carrierD(2) vec-elemsD*)  
      **then have**  $\text{row } (A x) j \$ i = A x \$\$ (j,i) \ i < n$  **using**  $\langle j < \text{dim}_r (A x) \rangle \langle i < n \rangle$   
      **by** (*simp-all add:*  $\langle i < \text{dim}_v (v x) \rangle$ )  
    }  
    **then have**  $\text{polyfun } N (\lambda x. \text{row } (A x) j \$ i * v x \$ i)$

```

    using polyfun-mult assms(4)[OF ⟨j < m⟩] assms(2) by fastforce
  }
  then show ?thesis unfolding index-mat-mult-vec[OF ⟨∧x. j < dimr (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] ⟨∧x. n = dimv (v x)⟩
    by simp
qed

```

```

lemma polyfun-evaluate-net-plus-a:
assumes map dimv inputs = input-sizes m
assumes valid-net m
assumes j < output-size m
shows polyfun {..v inputs = input-sizes (insert-weights m (λi. f (i + x1 *
x2)))
        using input-sizes-remove-weights remove-insert-weights
        by (simp add: Conv.premis(1))
      have dimv (evaluate-net (insert-weights m (λi. 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 (λi. f (i + x1 * x2))) inputs ∈
carrierv (output-size m)
        using vec-carrier-def by (metis (full-types) mem-Collect-eq)
    }

  have map dimv inputs = input-sizes m by (simp add: Conv.premis(1))
  have valid-net m using Conv.premis(2) valid-net.cases by fastforce
  show ∧j. j < output-size m ⇒ polyfun {..

```

```

    using Conv(1)[OF ⟨map dimv 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.prem(2) ⟨x = (x1, x2)⟩ valid-net.cases
by fastforce
    show  $\bigwedge f. \text{extract-matrix } (\lambda i. f (i + a)) x1 x2 \in \text{carrier}_m x1 (\text{output-size } m)$ 
unfolding ⟨output-size m = x2⟩ using dim-extract-matrix
    using mat-carrierI by (metis (no-types, lifting))

    show  $\bigwedge i j. i < x1 \implies j < \text{output-size } m \implies \text{polyfun } \{.. < a + \text{count-weights}$ 
(Cov (x1, x2) m) }  $(\lambda f. \text{extract-matrix } (\lambda i. f (i + a)) x1 x2 \text{ } \text{\$ } (i, j))$ 
    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.prem(3) ⟨x = (x1, x2)⟩ by auto
qed
next
    case (Pool m1 m2 inputs j a)
    have A2:  $\bigwedge f. \text{map dim}_v (\text{take } (\text{length } (\text{input-sizes } (\text{insert-weights } m1 (\lambda i. f (i + a)))) \text{ inputs}) = \text{input-sizes } m1$ 
+ a)) by (metis Pool.prem(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights
remove-insert-weights take-map)
    have B2:  $\bigwedge f. \text{map dim}_v (\text{drop } (\text{length } (\text{input-sizes } (\text{insert-weights } m1 (\lambda i. f (i + a)))) \text{ inputs}) = \text{input-sizes } m2$ 
+ a)) by (metis Pool.prem(1) append-eq-conv-conj input-sizes.simps(3) input-sizes-remove-weights
remove-insert-weights 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 =  $\lambda f. \text{evaluate-net } (\text{insert-weights } m1 (\lambda i. f (i + a)))$ 
(take (length (input-sizes (insert-weights m1 (lambda i. f (i + a)))) inputs)
let ?net2 =  $\lambda f. \text{evaluate-net } (\text{insert-weights } m2 (\lambda i. f (i + \text{count-weights } m1 +$ 
a)))
(drop (length (input-sizes (insert-weights m1 (lambda i. f (i + a)))) inputs)
have length1:  $\bigwedge f. \text{output-size } m1 = \text{dim}_v (?net1 f)$ 
by (metis A2 A3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength1:  $\bigwedge f. j < \text{dim}_v (?net1 f)$  using A4 by metis
have length2:  $\bigwedge f. \text{output-size } m2 = \text{dim}_v (?net2 f)$ 
by (metis B2 B3 input-sizes-remove-weights output-size-correct remove-insert-weights)
then have jlength2:  $\bigwedge f. j < \text{dim}_v (?net2 f)$  using B4 by metis
have cong1:  $\bigwedge x f. (\lambda f. \text{evaluate-net } (\text{insert-weights } m1 (\lambda i. f (i + a)))$ 

```

```

      (take (length (input-sizes (insert-weights m1 ( $\lambda i$ .  $xf$  (i + a)))))) inputs) $ j)
    = ( $\lambda f$ . ?net1 f $ j)
  using input-sizes-remove-weights remove-insert-weights by auto
  have cong2: $\wedge x f$ . ( $\lambda f$ . evaluate-net (insert-weights m2 ( $\lambda i$ . f (i + (a + count-weights
m1))))))
    (drop (length (input-sizes (insert-weights m1 ( $\lambda i$ .  $xf$  (i + a)))))) inputs) $ j)
    = ( $\lambda f$ . ?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**  $map\ dim_v\ inputs = input-sizes\ m$   
**assumes** *valid-net m*  
**assumes**  $j < output-size\ m$   
**shows**  $polyfun\ \{.. $count-weights\ m\}\ (\lambda f$ . 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**  $is \triangleleft input-sizes\ m$   
**assumes**  $j < output-size\ m$   
**shows**  $polyfun\ \{.. $count-weights\ m\}\ (\lambda f$ . Tensor.lookup (tensors-from-net (insert-weights  
m f) $ j) is)$

**proof** –

**have** 1: $\wedge f$ . *valid-net'* (insert-weights m f) **by** (*simp add: assms(1) remove-insert-weights*)

**have**  $input-sizes:\wedge f$ .  $input-sizes\ (insert-weights\ m\ f) = input-sizes\ m$   
**unfolding** *input-sizes-remove-weights* **by** (*simp add: remove-insert-weights*)

**have** 2: $\wedge f$ .  $is \triangleleft input-sizes\ (insert-weights\ m\ f)$   
**unfolding** *input-sizes* **using** *assms(2)* **by** *blast*

**have** 3: $\wedge f$ .  $j < output-size'\ (insert-weights\ m\ f)$   
**by** (*simp add: assms(3) remove-insert-weights*)

**have**  $\wedge f1\ f2$ .  $base-input\ (insert-weights\ m\ f1)\ is = base-input\ (insert-weights\ m$   
 $f2)\ is$   
**unfolding** *base-input-def* **by** (*simp add: input-sizes*)

**then** **have**  $\wedge x f$ . ( $\lambda f$ . evaluate-net (insert-weights m f) (base-input (insert-weights  
m x f) is) \$ j)

$= (\lambda f. \text{evaluate-net } (\text{insert-weights } m f) (\text{base-input } (\text{insert-weights } m f) \text{ is}) \$$   
 $j)$   
**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**  $\bigwedge x. \text{dims } (T x) = ds$   
**assumes**  $\bigwedge is. is \triangleleft ds \implies \text{polyfun } N (\lambda x. \text{Tensor.lookup } (T x) \text{ is})$   
**assumes**  $\bigwedge x. \text{dim}_r (\text{matricize } I (T x)) = nr$   
**assumes**  $\bigwedge x. \text{dim}_c (\text{matricize } I (T x)) = nc$   
**assumes**  $i < nr$   
**assumes**  $j < nc$   
**shows**  $\text{polyfun } N (\lambda x. \text{matricize } I (T x) \$\$ (i,j))$   
**proof** –  
**let**  $?weave = \lambda x. (\text{weave } I$   
 $(\text{digit-encode } (\text{sublist } ds I) i)$   
 $(\text{digit-encode } (\text{sublist } ds (-I)) j))$   
**have**  $1: \bigwedge x. \text{matricize } I (T x) \$\$ (i,j) = \text{Tensor.lookup } (T x) (?weave x)$  **un-**  
**folding** *matricize-def*  
**by** (*metis (no-types, lifting) assms(1) assms(3) assms(4) assms(5) assms(6)*  
*case-prod-conv*  
 $\text{mat-dim-col-mat}(1) \text{mat-dim-row-mat}(1) \text{mat-index-mat}(1) \text{matricize-def}$ )  
**have**  $\bigwedge x. ?weave x \triangleleft ds$   
**using** *valid-index-weave(1) assms(2) digit-encode-valid-index mat-dim-row-mat(1)*  
*matricize-def*  
**using** *assms digit-encode-valid-index matricize-def* **by** (*metis mat-dim-col-mat(1)*)  
**then have**  $\text{polyfun } N (\lambda x. \text{Tensor.lookup } (T x) (?weave x))$  **using** *assms(2)* **by**  
*simp*  
**then show** *?thesis unfolding 1 using assms(1) by blast*  
**qed**

**lemma**  $(\neg (a::nat) < b) = (a \geq b)$   
**by** (*metis not-le*)

**lemma** *polyfun-submatrix*:  
**assumes**  $\bigwedge x. (A x) \in \text{carrier}_m m n$   
**assumes**  $\bigwedge x i j. i < m \implies j < n \implies \text{polyfun } N (\lambda x. (A x) \$\$ (i,j))$   
**assumes**  $i < \text{card } \{i. i < m \wedge i \in I\}$   
**assumes**  $j < \text{card } \{j. j < n \wedge j \in J\}$   
**assumes** *infinite I infinite J*  
**shows**  $\text{polyfun } N (\lambda x. (\text{submatrix } (A x) I J) \$\$ (i,j))$   
**proof** –  
**have**  $1: \bigwedge x. (\text{submatrix } (A x) I J) \$\$ (i,j) = (A x) \$\$ (\text{pick } I i, \text{pick } J j)$   
**using** *submatrix-index* **by** (*metis (no-types, lifting) Collect-cong assms(1)*  
*assms(3) assms(4) mat-carrierD(1) mat-carrierD(2)*)

**have** *pick*  $I$   $i < m$  *pick*  $J$   $j < n$  **using** *card-le-pick-inf* [*OF* (*infinite*  $I$ )] *card-le-pick-inf* [*OF* (*infinite*  $J$ )]  
 $\langle i < \text{card } \{i. i < m \wedge i \in I\} \rangle$  [*unfolded set-le-in*]  $\langle j < \text{card } \{j. j < n \wedge j \in J\} \rangle$  [*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*  $j$   $f = \text{submatrix } (A' f) \text{ rows-with-1 rows-with-1}$

**lemma** *polyfun-tensor-deep-model*:  
**assumes**  $is \triangleleft \text{input-sizes } (\text{deep-model-l } rs)$   
**shows** *polyfun*  $\{.. < \text{weight-space-dim}\}$   
 $(\lambda f. \text{Tensor.lookup } (\text{tensors-from-net } (\text{insert-weights } (\text{deep-model-l } rs) f) \$ y) is)$   
**proof** –  
**have**  $1: \lambda f. \text{remove-weights } (\text{insert-weights } (\text{deep-model-l } rs) f) = \text{deep-model-l } rs$   
**using** *remove-insert-weights* **by** *metis*  
**then have**  $y < \text{output-size } (\text{deep-model-l } rs)$  **using** *valid-deep-model y-valid length-output-deep-model* **by force**  
**have**  $0: \{.. < \text{weight-space-dim}\} = \text{set } [0.. < \text{weight-space-dim}]$  **by** *auto*  
**then show** *?thesis unfolding weight-space-dim-def* **using** *polyfun-tensors-from-net assms(1) valid-deep-model*  
 $\langle y < \text{output-size } (\text{deep-model-l } rs) \rangle$  **by** *metis*  
**qed**

**lemma** *input-sizes-deep-model*:  $\text{input-sizes } (\text{deep-model-l } rs) = \text{replicate } (2 * N\text{-half}) (\text{last } rs)$   
**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**  $i < (\text{last } rs) \wedge N\text{-half}$   
**assumes**  $j < (\text{last } rs) \wedge N\text{-half}$   
**shows** *polyfun*  $\{.. < \text{weight-space-dim}\} (\lambda f. A' f \$\$ (i, j))$   
**proof** –  
**have**  $0: y < \text{output-size } (\text{deep-model-l } rs)$  **using** *valid-deep-model y-valid length-output-deep-model* **by force**  
**have**  $1: \lambda f. \text{remove-weights } (\text{insert-weights } (\text{deep-model-l } rs) f) = \text{deep-model-l } rs$   
**using** *remove-insert-weights* **by** *metis*  
**have**  $2: (\lambda f \text{ is. } is \triangleleft \text{replicate } (2 * N\text{-half}) (\text{last } rs) \implies \text{polyfun } \{.. < \text{weight-space-dim}\} (\lambda x. \text{Tensor.lookup } (A x) is))$   
**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} (λf. witness-submatrix y f $$ (i,j))
unfolding witness-submatrix-def
proof (rule polyfun-submatrix)
  have 1: λf. remove-weights (insert-weights (deep-model-l rs) f) = deep-model-l
rs
  using remove-insert-weights by metis
show λf. A' f ∈ carrierm ((last rs) ^ N-half) ((last rs) ^ N-half)
  using 1 dims-A'-pow using weight-space-dim-def by auto
show λf i j. i < last rs ^ N-half ⇒ j < last rs ^ N-half ⇒
polyfun {..<weight-space-dim} (λf. A' f $$ (i, j))
  using polyfun-matrix-deep-model weight-space-dim-def by force
show i < card {i. i < last rs ^ N-half ∧ i ∈ rows-with-1}
  using assms(1) card-rows-with-1 dims-Aw'-pow set-le-in by metis
show j < card {i. i < last rs ^ N-half ∧ i ∈ rows-with-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} (λf. det (witness-submatrix y 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 y f ∈ carrierm (r ^ N-half) (r ^ N-half)
  unfolding witness-submatrix-def apply (rule mat-carrierI) 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 λi j. i < r ^ N-half ⇒ j < r ^ N-half ⇒ polyfun {..<weight-space-dim}
(λf. witness-submatrix y f $$ (i, j))
  using polyfun-submatrix-deep-model by blast
qed

end

```

end

## 34 Alternative Lebesgue Measure Definition

**theory** *Lebesgue-Functional*

**imports** *~/src/HOL/Probability/Lebesgue-Measure ~/src/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*  $\Rightarrow$  (*nat*  $\Rightarrow$  *real*) *measure* **where**

*lborel-f* *n* = ( $\Pi_M$  *b*  $\in$   $\{..<n\}$ ). *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*) = *PiE*  $\{..<n\}$  ( $\lambda$ -. *UNIV*) **unfolding** *lborel-f-def*

**unfolding** *space-PiM space-lborel space-borel* **by** *metis*

**lemma** *space-lborel-f-subset*: *space* (*lborel-f* *n*)  $\subseteq$  *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*  $\in$  *space* (*lborel-f* *n*)

**shows** *f* (*n* := *x*)  $\in$  *space* (*lborel-f* (*Suc* *n*))

**unfolding** *space-lborel-f*

**using** *assms lessThan-Suc space-lborel-f* **by** *auto*

**lemma** *integral-lborel-f*:

**assumes** *f*  $\in$  *borel-measurable* (*lborel-f* (*Suc* *n*))

**shows**  $\text{integral}^N$  (*lborel-f* (*Suc* *n*)) *f* =  $\int^+ y$ .  $\int^+ x$ . *f* (*x* (*n* := *y*))  $\partial$ *lborel-f* *n*  $\partial$ *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**  $A \in \text{sets } (\text{lborel-f } (\text{Suc } n))$

**assumes**  $\bigwedge y. \{x \in \text{space } (\text{lborel-f } n). x(n := y) \in A\} \in \text{sets } (\text{lborel-f } n)$

**shows**  $\text{emeasure } (\text{lborel-f } (\text{Suc } n)) A = \int^+ y. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := y) \in A\} \partial \text{lborel}$

**proof** –

{

**fix**  $x y$  **assume**  $x \in \text{space } (\text{lborel-f } n)$

**then have**  $(\text{indicator } A) (x(n := y)) = (\text{indicator } \{x \in \text{space } (\text{lborel-f } n). x(n := y) \in A\}) x$  **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*:  $(\lambda f. f(n := x)) \in \text{measurable } (\text{lborel-f } n)$   
 $(\text{lborel-f } (\text{Suc } n))$

**proof** –

**have**  $x \in \text{space } \text{lborel}$  **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<sub>M</sub> {..<n}*]  $(\lambda b.$   
 $\text{lborel}), \text{OF } \langle x \in \text{space } \text{lborel} \rangle$ ]

*measurable-add-dim*[*of n {..<n} \lambda b. lborel], unfolded 0*] *lessThan-Suc* **by** *auto*

**qed**

**lemma** *sets-lborel-f-sub-dim*:

**assumes**  $A \in \text{sets } (\text{lborel-f } (\text{Suc } n))$

**shows**  $\{x. x(n := y) \in A\} \cap \text{space } (\text{lborel-f } n) \in \text{sets } (\text{lborel-f } n)$

**proof** –

**have**  $(\lambda f. f(n := y)) -' A \cap \text{space } (\text{lborel-f } n) \in \text{sets } (\text{lborel-f } n)$

**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**  $(\lambda f. \text{restrict } f \{..<m\}) \in \text{measurable } (\text{lborel-f } n) (\text{lborel-f } m)$

**using** *measurable-restrict-subset lborel-f-def assms* **by** *auto*

**lemma** *lborel-measurable-sub-dim*:  $(\lambda f. \text{restrict } f \{..<n\}) \in \text{measurable } (\text{lborel-f } (\text{Suc } n)) (\text{lborel-f } n)$   
**using** *lborel-f-measurable-restrict*[of *n Suc n*] **by** *linarith*

**lemma** *measurable-lborel-component* [*measurable*]:  
**assumes**  $k < n$   
**shows**  $(\lambda x. x \ k) \in \text{borel-measurable } (\text{lborel-f } n)$   
**unfolding** *lborel-f-def* **using** *assms measurable-PiM-component-rev* **by** *simp-all*

**end**

**theory** *PP-Univariate*  
**imports** *PP-MPoly PP-More-MPoly*  $\sim\sim$  */src/HOL/Library/Polynomial*  
**begin**

This file connects univariate MPolys to the theory of univariate polynomials from *HOL/Library/Polynomial*.

**definition** *poly-to-mpoly*:: $\text{nat} \Rightarrow 'a::\text{comm-monoid-add}$  *poly} \Rightarrow 'a *mpoly*  
**where** *poly-to-mpoly*  $v \ p = \text{MPoly } (\text{Abs-poly-mapping } (\lambda m. (\text{coeff } p (\text{PP-Poly-Mapping.lookup } m \ v))) \text{ when } \text{PP-Poly-Mapping.keys } m \subseteq \{v\}))$*

**lemma** *poly-to-mpoly-finite*:  $\text{finite } \{m::\text{nat} \Rightarrow_0 \text{nat}. (\text{coeff } p (\text{PP-Poly-Mapping.lookup } m \ v) \text{ when } \text{PP-Poly-Mapping.keys } m \subseteq \{v\}) \neq 0\}$  **(is finite ?M)**

**proof** –

**have**  $?M \subseteq \text{PP-Poly-Mapping.single } v \ \{x. \text{Polynomial.coeff } p \ x \neq 0\}$

**proof**

**fix**  $m$  **assume**  $m \in ?M$

**then have**  $\bigwedge v'. v' \neq v \implies \text{PP-Poly-Mapping.lookup } m \ v' = 0$  **by** *fastforce*

**then have**  $m = \text{PP-Poly-Mapping.single } v (\text{PP-Poly-Mapping.lookup } m \ v)$

**using** *PP-Poly-Mapping.poly-mapping-eqI* **by** (*metis (full-types) lookup-single-eq lookup-single-not-eq*)

**then show**  $m \in (\text{PP-Poly-Mapping.single } v) \ \{x. \text{Polynomial.coeff } p \ x \neq 0\}$

**using**  $\langle m \in ?M \rangle$  **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*:  $\text{PP-MPoly.coeff } (\text{poly-to-mpoly } v \ p) (\text{PP-Poly-Mapping.single } v \ k) = \text{Polynomial.coeff } p \ k$

**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*:: $\text{nat} \Rightarrow 'a::\text{comm-monoid-add}$  *mpoly} \Rightarrow 'a *poly**

**where** *mpoly-to-poly*  $v \ p = \text{Abs-poly } (\lambda k. \text{PP-MPoly.coeff } p (\text{PP-Poly-Mapping.single } v \ k))$

**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**  $0:PP-Poly-Mapping.single\ v \ ' \ \{x.\ PP-Poly-Mapping.lookup\ (mapping-of\ p)\ (PP-Poly-Mapping.single\ v\ x) \neq 0\}$

$\subseteq \{k.\ PP-Poly-Mapping.lookup\ (mapping-of\ p)\ k \neq 0\}$

**by** *auto*

**have**  $\forall_{\infty} k.\ PP-MPoly.coeff\ p\ (PP-Poly-Mapping.single\ v\ k) = 0$  **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**  $vars\ p \subseteq \{v\}$

**shows**  $poly-to-mpoly\ v\ (mpoly-to-poly\ v\ p) = p$

**proof** –

**def**  $f == (\lambda m.\ Polynomial.coeff\ (mpoly-to-poly\ v\ p)\ (PP-Poly-Mapping.lookup\ m\ v)\ \text{when}\ PP-Poly-Mapping.keys\ m \subseteq \{v\})$

**have**  $finite\ \{m.\ f\ m \neq 0\}$  **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  $\subseteq$  {v}*)

**assume**  $PP-Poly-Mapping.keys\ m \subseteq \{v\}$

**then show** *?thesis* **unfolding** *PP-Poly-Mapping.lookup-Abs-poly-mapping[OF finite {m. f m  $\neq$  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**  $\neg PP-Poly-Mapping.keys\ m \subseteq \{v\}$

**then show** *?thesis* **unfolding** *PP-Poly-Mapping.lookup-Abs-poly-mapping[OF finite {m. f m  $\neq$  0}]* **unfolding** *f-def*

**using** ( $vars\ p \subseteq \{v\}$ ) **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*:  $\text{poly-to-mpoly } v \ 0 = 0$   
**proof** –  
**have**  $\bigwedge m. (\text{Polynomial.coeff } 0 \ (\text{PP-Poly-Mapping.lookup } m \ v) \ \text{when } \text{PP-Poly-Mapping.keys } m \subseteq \{v\}) = 0$  **by** *simp*  
**have**  $\text{Abs-poly-mapping } (\lambda m. \text{Polynomial.coeff } 0 \ (\text{PP-Poly-Mapping.lookup } m \ v))$   
*when*  $\text{PP-Poly-Mapping.keys } m \subseteq \{v\} = 0$   
**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*:  $\text{mpoly-to-poly } v \ (p1 + p2) = \text{mpoly-to-poly } v \ p1 + \text{mpoly-to-poly } v \ p2$   
**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**  $\text{vars } p \subseteq \{v\}$   
**shows**  $\text{poly } (\text{mpoly-to-poly } v \ p) \ x = \text{insertion } (\lambda v. \ x) \ p$   
**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**  $\text{PP-Poly-Mapping.keys } m \subseteq \{v\}$  **using** *monom* **unfolding** *vars-def PP-MPoly.mapping-of-monom keys-single* **by** *simp*  
**then have**  $\bigwedge v'. \ v' \neq v \implies \text{PP-Poly-Mapping.lookup } m \ v' = 0$  **unfolding** *vars-def* **by** *auto*  
**then have**  $m = \text{PP-Poly-Mapping.single } v \ (\text{PP-Poly-Mapping.lookup } m \ v)$   
**by** (*metis* *lookup-single-eq lookup-single-not-eq poly-mapping-eqI*)  
**then have**  $0: \text{insertion } (\lambda v. \ x) \ (\text{PP-MPoly.monom } m \ a) = a * x \wedge (\text{PP-Poly-Mapping.lookup } m \ v)$   
**using** *insertion-single* **by** *metis*  
**have**  $\bigwedge k. \ \text{PP-Poly-Mapping.single } v \ k = m \longleftrightarrow \text{PP-Poly-Mapping.lookup } m \ v = k$   
**using**  $\langle m = \text{PP-Poly-Mapping.single } v \ (\text{PP-Poly-Mapping.lookup } m \ v) \rangle$  **by** *auto*  
**then have**  $\text{monom } a \ (\text{PP-Poly-Mapping.lookup } m \ v) = (\text{Abs-poly } (\lambda k. \ \text{if } \text{PP-Poly-Mapping.single } v \ k = m \ \text{then } a \ \text{else } 0))$   
**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 (λv. x) p1
      poly (mpoly-to-poly v p2) x = insertion (λv. 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 (λv. 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 (λf. 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.premis 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 (λf. insertion f p1) ∈ borel-measurable (lborel-f n)
    (λf. insertion f p2) ∈ borel-measurable (lborel-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 ≠ 0 vars p ⊆ {..<n}
shows {f∈space (lborel-f n). insertion f p = 0} ∈ null-sets (lborel-f n)
using assms proof (induction n arbitrary:p)
  case 0
  then have vars p = {} by simp then have  $\bigwedge f. \text{insertion } f \text{ } p = \text{PP-MPoly.coeff}$ 
p 0
    unfolding insertion-trivial[symmetric] using insertion-irrelevant-vars by
blast
  have  $\bigwedge m. m \neq 0 \implies \text{PP-MPoly.coeff } p \text{ } m = 0$ 
  proof (rule ccontr)
    fix m::nat ⇒0 nat assume m≠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∈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  $PP\text{-MPoly.coeff } p \ 0 \neq 0$  using  $\langle p \neq 0 \rangle$ 
  by  $(metis \text{coeff-all-0})$ 
then have  $\{f. \text{insertion } f \ p = 0\} = \{\}$  using  $\langle \bigwedge f. \text{insertion } f \ p = PP\text{-MPoly.coeff } p \ 0 \rangle$  by auto
then show  $?case$  by auto
next
  case  $(Suc \ n \ p)$ 

    Show that N is finite:

    then have  $\text{extract-var } p \ n \neq 0$  using  $\text{reduce-nested-mpoly-0}$ 
      by  $(metis \text{reduce-nested-mpoly-extract-var})$ 
    let  $?q = \lambda j. PP\text{-MPoly.coeff } (\text{extract-var } p \ n) \ j$ 
    obtain  $j$  where  $?q \ j \neq 0$  using  $\langle \text{extract-var } p \ n \neq 0 \rangle$ 
      by  $(metis \text{coeff-all-0})$ 
    then have  $\text{finite } \{x. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0\}$ 
      using  $\text{univariate-mpoly-roots-finite}[OF \ \text{vars-coeff-extract-var}]$  by metis
    then have  $\text{finite } (\bigcap j. \{x. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0\})$  by auto
    moreover have  $\{x. \forall j. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0\} = (\bigcap j. \{x. \text{insertion } (\lambda v. \ x) \ (?q \ j) = 0\})$  by blast
    ultimately have  $\text{finite } \{x. \forall j. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0\}$  by metis

    def  $p\text{-fix1} == \lambda x1. \text{replace-coeff } (\text{insertion } (\lambda-. \ x1)) \ (\text{extract-var } p \ n)$ 
    def  $N == \{x1. p\text{-fix1 } x1 = 0\}$ 
    have  $N \subseteq \{x. \forall j. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0\}$ 
    proof
      fix  $x$  assume  $x \in N$ 
      then have  $p\text{-fix1 } x = 0$  using  $N\text{-def}$  by auto
      then have  $\bigwedge m. PP\text{-MPoly.coeff } (p\text{-fix1 } x) \ m = 0$  by  $(metis \text{PP-More-MPoly.coeff-monom-monom-zero when-def})$ 
      have  $\bigwedge j. \text{insertion } (\lambda-. \ x) \ (?q \ j) = 0$ 
      using  $\langle \bigwedge m. PP\text{-MPoly.coeff } (p\text{-fix1 } x) \ m = 0 \rangle$   $[\text{unfolded } p\text{-fix1-def coeff-replace-coeff}[of \ \text{insertion } (\lambda-. \ x), \ OF \ \text{insertion-zero}]]$ 
      by metis
      then show  $x \in \{x. \forall j. \text{insertion } (\lambda-. \ x) \ (PP\text{-MPoly.coeff } (\text{extract-var } p \ n) \ j) = 0\}$  by blast
    qed
    then have  $\text{finite } N$  by  $(\text{simp add: } \{ \text{finite } \{x. \forall j. \text{insertion } (\lambda-. \ x) \ (PP\text{-MPoly.coeff } (\text{extract-var } p \ n) \ j) = 0\} \}$   $\text{finite-subset})$ 

    Use the IH:

    def  $A == \{f \in \text{space } (\text{lborel-f } (Suc \ n)). \text{insertion } f \ p = 0\}$ 

    have  $\bigwedge x1. \text{vars } (p\text{-fix1 } x1) \subseteq \{..<n\}$ 
    proof -
      fix  $x1$ 
      have  $\text{vars } (\text{extract-var } p \ n) \subseteq \{..<n\}$ 
      using  $\langle \text{vars } p \subseteq \{..<Suc \ n\} \rangle$   $\text{lessThan-Suc v-not-in-vars-extract-var vars-extract-var-subset}$ 
    by fastforce
    then show  $\text{vars } (p\text{-fix1 } x1) \subseteq \{..<n\}$  unfolding  $p\text{-fix1-def}$ 

```

```

using vars-replace-coeff[of insertion (λ-. x1), OF insertion-zero] by blast
qed
have set-eq:  $\bigwedge x1. \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\}$ 
proof –
  fix x1
  show  $\{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\}$ 
  proof (rule subset-antisym; rule subsetI)
    fix x assume  $x \in \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\}$ 
    then have insertion (x(n := x1)) p = 0  $x \in \text{space } (\text{lborel-f } n)$ 
      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  $x \in \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\}$  using  $\langle x \in \text{space } (\text{lborel-f } n) \rangle$  by blast
  next
    fix f assume  $f \in \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\}$ 
    then have  $f \in \text{space } (\text{lborel-f } n)$  insertion f (p-fix1 x1) = 0 by auto
    have insertion (f(n := x1)) p = 0 using  $\langle \text{insertion } f (p\text{-fix1 } x1) = 0 \rangle$  [unfolded p-fix1-def]
      insertion-replace-coeff insertion-irrelevant-vars replace-coeff-extract-var-cong
      by (metis (no-types, lifting)  $\langle \text{insertion } f (p\text{-fix1 } x1) = 0 \rangle$   $\langle \text{vars } (p\text{-fix1 } x1) \rangle \subseteq \{..<n\}$ )
      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  $\langle f \in \text{space } (\text{lborel-f } n) \rangle$  lborel-f-def mem-Collect-eq by blast
    then show  $f \in \{f \in \text{space } (\text{lborel-f } n). f(n := x1) \in A\}$  using  $\langle f \in \text{space } (\text{lborel-f } n) \rangle$  by auto
  qed
qed

have  $\bigwedge x1. x1 \in N \implies \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n)$ 
  and emeasure-in-N:  $\bigwedge x1. x1 \in N \implies \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = \text{emeasure } (\text{lborel-f } n) (\text{space } (\text{lborel-f } n))$ 
proof –
  fix x1 assume  $x1 \in N$ 
  then have p-fix1 x1 = 0 using N-def by auto
  then have  $\bigwedge f. \text{insertion } f (p\text{-fix1 } x1) = 0$  using insertion-zero by auto
  then have  $\{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = \text{space } (\text{lborel-f } n)$  by simp

```

**show**  $\{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n)$  **unfolding**  
*set-eq*  
**by** (*simp add*:  $\langle \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = \text{space } (\text{lborel-f } n) \rangle$ )  
**show**  $\text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = \text{emeasure } (\text{lborel-f } n) (\text{space } (\text{lborel-f } n))$   
**unfolding** *set-eq*  
**by** (*simp add*:  $\langle \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = \text{space } (\text{lborel-f } n) \rangle$ )  
**qed**

**have**  $\text{emeasure-not-in-}N: \bigwedge x1. x1 \notin N \implies \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0$   
**and**  $\bigwedge x1. x1 \notin N \implies \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n)$

**proof** –  
**fix**  $x1$  **assume**  $x1 \notin N$   
**then have**  $p\text{-fix1 } x1 \neq 0$  **using**  $p\text{-fix1-def } N\text{-def}$  **by** *auto*  
**then have**  $\text{emeasure } (\text{lborel-f } n) \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = 0$   
 $\{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} \in \text{sets } (\text{lborel-f } n)$   
**using**  $\text{Suc.IH}[OF \langle p\text{-fix1 } x1 \neq 0 \rangle] \langle \bigwedge x1. \text{vars } (p\text{-fix1 } x1) \subseteq \{..<n\} \rangle$  **by** *auto*  
**then show**  $\text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0$   
 $\{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} \in \text{sets } (\text{lborel-f } n)$   
**using**  $\langle \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} \in \text{sets } (\text{lborel-f } n) \rangle$   
 $\langle \text{emeasure } (\text{lborel-f } n) \{f \in \text{space } (\text{lborel-f } n). \text{insertion } f (p\text{-fix1 } x1) = 0\} = 0 \rangle$   
**using** *set-eq*  
**by** *auto*  
**qed**

**have**  $N \in \text{null-sets } \text{lborel}$  **using**  $\langle \text{finite } N \rangle$   $\text{finite-imp-null-set-lborel}$  **by** *blast*  
**have**  $\text{ae-zero}: \text{AE } x1 \text{ in } \text{lborel}. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0$   
**apply** (*rule*  $\text{AE-I}'[OF \langle N \in \text{null-sets } \text{lborel} \rangle]$ )  
**using**  $\langle \bigwedge x1. x1 \notin N \implies \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\} = 0 \rangle$   
**by** *force*

**have**  $\text{measurable}: (\lambda x1. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\}) \in \text{borel-measurable } \text{lborel}$   
**proof** (*rule*  $\text{borel-measurableI}$ )  
**let**  $?f = (\lambda x1. \text{emeasure } (\text{lborel-f } n) \{x \in \text{space } (\text{lborel-f } n). x(n := x1) \in A\})$   
**fix**  $S::\text{ennreal set}$  **assume**  $\text{open } S$   
**have**  $0:0 \in S \implies - N \subseteq ?f - ' S$   
**using**  $\text{emeasure-not-in-}N$  **by** *auto*  
**have**  $1:\text{emeasure } (\text{lborel-f } n) (\text{space } (\text{lborel-f } n)) \in S \implies N \subseteq ?f - ' S$   
**using**  $\text{emeasure-in-}N$  **by** *auto*  
**have**  $2:0 \notin S \implies ?f - ' S \subseteq N$  **using**  $\text{emeasure-not-in-}N$  **by** *fastforce*

```

have  $\exists$ :emeasure (lborel-f n) (space (lborel-f n))  $\notin S \implies ?f - ' S \subseteq -N$  using
emeasure-in-N by auto
have  $?f - ' S = \{\}$   $\vee ?f - ' S = N$   $\vee ?f - ' S = UNIV$   $\vee ?f - ' S = - N$ 
apply (cases  $0 \in S$ ; cases emeasure (lborel-f n) (space (lborel-f n))  $\notin S$ )
using 0 1 2 3 by auto
then show  $?f - ' S \cap \text{space lborel} \in \text{sets lborel}$ 
using (finite N) finite-imp-null-set-lborel borel-comp null-setsD2 sets-lborel by
fastforce
qed

have  $A \in \text{sets (lborel-f (Suc n))}$  unfolding A-def
using pred-eq-const1[OF measurable-insertion[OF (vars  $p \subseteq \{..<Suc n\}$ )]]
pred-def by force
then have in-sets:  $\{f \in \text{space (lborel-f (Suc n))}. \text{insertion } f \ p = 0\} \in \text{sets}$ 
(lborel-f (Suc n)) using A-def by metis
have  $\bigwedge x1. \{x \in \text{space (lborel-f n)}. x(n := x1) \in A\} \in \text{sets (lborel-f n)}$ 
using (math  $\bigwedge x1. x1 \in N \implies \{x \in \text{space (lborel-f n)}. x(n := x1) \in A\} \in \text{sets (lborel-f n)}$ )
(math  $\bigwedge x1. x1 \notin N \implies \{x \in \text{space (lborel-f n)}. x(n := x1) \in A\} \in \text{sets (lborel-f n)}$ ) by
auto
have emeasure (lborel-f (Suc n))  $A = \int^+ y. \text{emeasure (lborel-f n)} \{x \in \text{space}$ 
(lborel-f n).  $x(n := y) \in A\} \partial \text{lborel}$ 
using emeasure-lborel-f-Suc[OF (A  $\in \text{sets (lborel-f (Suc n))}$ )]
(math  $\bigwedge x1. \{x \in \text{space (lborel-f n)}. x(n := x1) \in A\} \in \text{sets (lborel-f n)}$ ) 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 Ranks of Submatrices

```

theory DL-Rank-Submatrix
imports DL-Rank DL-Submatrix DL-Missing-Matrix
begin

```

```

definition subvec  $v \ I = \text{vec-of-list (sublist (list-of-vec } v) \ I)$ 

```

```

lemma index-subvec:

```

```

assumes  $i < \text{card } \{i. i < \text{dim}_v \ v \wedge i \in I\}$ 

```

```

shows subvec  $v \ I \ \$ \ i = v \ \$ \ \text{pick } I \ i$ 

```

```

proof -

```

```

have  $i < \text{card } \{i. i < \text{length (list-of-vec } v) \wedge i \in I\}$  using assms by auto

```

```

then show ?thesis using nth-sublist[of  $i$  list-of-vec  $v \ I$ ] by (metis list-of-vec-index
list-vec subvec-def)

```

```

qed

```

**lemma** *dim-subvec*:  $\dim_v (\text{subvec } v \ I) = \text{card } \{i. i < \dim_v v \wedge i \in I\}$   
**unfolding** *subvec-def* **using** *length-sublist* **by** (*metis dim-vec-of-list vec-list*)

**lemma** *subvec-add*:

**assumes**  $\dim_v a = \dim_v b$

**shows**  $\text{subvec } a \ I \oplus_v \text{subvec } b \ I = \text{subvec } (a \oplus_v b) \ I$

**proof**

**show**  $\dim_v (\text{subvec } a \ I \oplus_v \text{subvec } b \ I) = \dim_v (\text{subvec } (a \oplus_v b) \ I)$

**by** (*simp add: dim-subvec*)

**fix**  $i$  **assume**  $i < \dim_v (\text{subvec } (a \oplus_v b) \ I)$

**then have**  $i < \dim_v (\text{subvec } a \ I) \ i < \dim_v (\text{subvec } b \ I)$  **using** *dim-subvec* **by**  
(*metis (full-types) assms vec-index-add(2)*)**+**

**then have** *pick*  $I \ i < \dim_v b$  **using** *dim-subvec pick-le* **by** *metis+*

**have**  $(\text{subvec } a \ I \oplus_v \text{subvec } b \ I) \ \$ \ i = a \ \$ \ \text{pick } I \ i + b \ \$ \ \text{pick } I \ i$

**unfolding** *vec-add-def dim-subvec vec-index-vec*[*OF*  $\langle i < \dim_v (\text{subvec } (a \oplus_v b) \ I) \rangle$ , *unfolded dim-subvec vec-index-add(2)*]

**using** *index-subvec* **by** (*metis (mono-tags, lifting)*  $\langle i < \dim_v (\text{subvec } a \ I) \rangle \langle i < \dim_v (\text{subvec } b \ I) \rangle$  *dim-subvec*)

**also have**  $\dots = (a \oplus_v b) \ \$ \ \text{pick } I \ i$  **by** (*simp add:*  $\langle \text{pick } I \ i < \dim_v b \rangle$ )

**also have**  $\dots = \text{subvec } (a \oplus_v b) \ I \ \$ \ i$  **unfolding** *vec-add-def dim-subvec*

**using** *vec-dim-vec*[*of*  $\dim_v b \ \lambda i. a \ \$ \ i + b \ \$ \ i$ ] *dim-subvec*

*index-subvec*[*of*  $i \ \text{Matrix.vec } (\dim_v b) \ (\lambda i. a \ \$ \ i + b \ \$ \ i) \ I$ ]  $\langle i < \dim_v (\text{subvec } b \ I) \rangle$  **by** *metis*

**finally show**  $(\text{subvec } a \ I \oplus_v \text{subvec } b \ I) \ \$ \ i = \text{subvec } (a \oplus_v b) \ I \ \$ \ i$  **by** *metis*  
**qed**

**lemma** *subvec-0*:  $\text{subvec } (\mathbf{0}_v \ n) \ I = \mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\})$

**proof**

**show**  $\dim_v (\text{subvec } (\mathbf{0}_v \ n) \ I) = \dim_v (\mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\}))$  **by** (*simp add: dim-subvec*)

**fix**  $i$  **assume**  $i < \dim_v (\mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\}))$

**then have**  $0 : i < \text{card } \{i. i < n \wedge i \in I\}$  **by** *simp*

**then have**  $1 : i < \text{card } \{i. i < \dim_v (\mathbf{0}_v \ n) \wedge i \in I\}$  **by** *simp*

**show**  $\text{subvec } (\mathbf{0}_v \ n) \ I \ \$ \ i = \mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\}) \ \$ \ i$  **unfolding**  
*index-subvec*[*OF*  $1$ ]

**using** *pick-le*[*OF*  $0$ ] **using**  $0$  **by** *auto*

**qed**

**lemma** (*in vec-space*) *subvec-finsum*:

**assumes** *finite*  $A$

**assumes**  $\bigwedge a. a \in A \implies a \in \text{carrier}_v \ n$

**shows**  $\text{subvec } (\bigoplus_{a \in A. a}) \ I = (\bigoplus_{\text{module}_v \ \text{TYPE}(a)} (\text{card } \{i. i < n \wedge i \in I\}) \ a)$   
 $a \in A. \text{subvec } a \ I$

**using** *assms* **proof** (*induction*  $A$  *rule:finite-induct*)

**case** *empty*

**have**  $1 : (\bigoplus_{\text{module}_v \ \text{TYPE}(a)} (\text{card } \{i. i < n \wedge i \in I\}) \ a) \in \{\}. \text{subvec } a \ I = \mathbf{0}_v$   
 $(\text{card } \{i. i < n \wedge i \in I\})$

**by** (*metis (no-types, lifting)* *finsum-vec-empty vec-space.finsum-vec*)

**have**  $2:(\bigoplus_{v} a \in \{ \}. a) = \mathbf{0}_v \ n$  **by** (*metis M.finsum-empty*)  
**show** *?case unfolding 1 2 using subvec-0 by auto*  
**next**  
**case** (*insert v A*)  
**then have**  $A:(\bigoplus_{v} a \in (\text{insert } v \ A). a) = v \oplus_v (\bigoplus_{v} a \in A. a)$   
**using** *M.finsum-insert[OF (finite A) (v ∉ A), of λx. x] by (simp add: Pi-I)*  
**have**  $1:(\lambda a. \text{subvec } a \ I) \in A \rightarrow \text{carrier}_v (\text{card } \{i. i < n \wedge i \in I\})$   
**apply** (*rule Pi-I, rule vec-elemsI*) **unfolding** *dim-subvec* **using** *insert by force*  
**have**  $2:\text{subvec } v \ I \in \text{carrier}_v (\text{card } \{i. i < n \wedge i \in I\})$   
**apply** (*rule vec-elemsI*) **unfolding** *dim-subvec* **using** *insert by force*  
**have**  $B:(\bigoplus_{\text{module}_v \ \text{TYPE}('a)} (\text{card } \{i. i < n \wedge i \in I\})^{a \in (\text{insert } v \ A). \text{subvec } a \ I})$   
 $= \text{subvec } v \ I \oplus_v (\bigoplus_{\text{module}_v \ \text{TYPE}('a)} (\text{card } \{i. i < n \wedge i \in I\})^{a \in A. \text{subvec } a \ I})$   
**using** *finsum-vec-insert[OF (finite A) (v ∉ A) 1 2, unfolded vec-space.finsum-vec]*  
**by** *meson*  
**show** *?case unfolding A B using subvec-add insert Pi-I insert-iff vec-elemsD*  
*vec-space.finsum-dim[OF (finite A), of λx. x] by (metis (no-types, lifting))*  
**qed**

**lemma** *subvec-scalar-mult:*

$$a \odot_v \text{subvec } v \ I = \text{subvec } (a \odot_v v) \ I$$

**proof**

**show**  $\text{dim}_v (a \odot_v \text{subvec } v \ I) = \text{dim}_v (\text{subvec } (a \odot_v v) \ I)$  **by** (*simp add: dim-subvec*)

**fix**  $i$  **assume**  $i < \text{dim}_v (\text{subvec } (a \odot_v v) \ I)$

**then have**  $1:i < \text{card } \{i. i < \text{dim}_v (a \odot_v v) \wedge i \in I\}$  **using** *dim-subvec by metis*

**then have**  $2:i < \text{card } \{i. i < \text{dim}_v v \wedge i \in I\}$  **by** *simp*

**then have**  $3:i < \text{dim}_v (\text{subvec } v \ I)$  **unfolding** *dim-subvec* **by** *metis*

**show**  $(a \odot_v \text{subvec } v \ I) \$ i = \text{subvec } (a \odot_v v) \ I \$ i$  **unfolding** *index-subvec[OF 1] vec-index-scalar-mult(1)[OF 3]*

*index-subvec[OF 2] using pick-le[OF 2] pick-le[OF 1] by simp*

**qed**

**lemma** *row-submatrix-UNIV:*

**assumes**  $i < \text{card } \{i. i < \text{dim}_r \ A \wedge i \in I\}$

**shows**  $\text{row } (\text{submatrix } A \ I \ \text{UNIV}) \ i = \text{row } A \ (\text{pick } I \ i)$

**proof** (*rule vec-eqI*)

**show**  $\text{dim-eq}:\text{dim}_v (\text{row } (\text{submatrix } A \ I \ \text{UNIV}) \ i) = \text{dim}_v (\text{row } A \ (\text{pick } I \ i))$

**unfolding** *vec-elemsD[OF row-dim] dim-submatrix* **by** *auto*

**fix**  $j$  **assume**  $j < \text{dim}_v (\text{row } A \ (\text{pick } I \ i))$

**then have**  $j < \text{dim}_c (\text{submatrix } A \ I \ \text{UNIV}) \ j < \text{dim}_c \ A \ j < \text{card } \{j. j < \text{dim}_c \ A \wedge j \in \text{UNIV}\}$  **using** *dim-eq* **by** *auto*

**show**  $\text{row } (\text{submatrix } A \ I \ \text{UNIV}) \ i \$ j = \text{row } A \ (\text{pick } I \ i) \$ j$

**unfolding** *row-def vec-index-vec[OF (j < dim\_c (submatrix A I UNIV))] vec-index-vec[OF (j < dim\_c A)]*

**using** *submatrix-index[OF assms (j < card {j. j < dim\_c 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_c A j < dim_c A (cols A)!i = (cols A)!j i ≠ j
      using distinct-conv-nth cols-length by metis
    have i < dim_c (submatrix A I UNIV) j < dim_c (submatrix A I UNIV)
      unfolding dim-submatrix using ⟨i < dim_c A⟩ ⟨j < dim_c 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 vec-eqI)
        show dim_v (cols (submatrix A I UNIV) ! i) = dim_v (cols (submatrix A I
  UNIV) ! j)
          by (simp add: ⟨i < dim_c (submatrix A I UNIV)⟩ ⟨j < dim_c (submatrix A I
  UNIV)⟩)
        fix k assume k < dim_v (cols (submatrix A I UNIV) ! j)
        then have k < dim_r (submatrix A I UNIV)
          using ⟨j < length (cols (submatrix A I UNIV))⟩ by auto
        then have k < card {j. j < dim_r 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_c (submatrix A I UNIV)⟩] col-def vec-index-vec[OF
  ⟨k < dim_r (submatrix A I UNIV)⟩]
            unfolding submatrix-index[OF ⟨k < card {j. j < dim_r A ∧ j ∈ I}⟩ ⟨i <
  dim_c (submatrix A I UNIV)⟩[unfolded dim-submatrix]]
            unfolding pick-UNIV cols-nth[OF ⟨i < dim_c A⟩] col-def vec-index-vec[OF
  pick-le[OF ⟨k < card {j. j < dim_r 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_c (submatrix A I UNIV)⟩] col-def vec-index-vec[OF
  ⟨k < dim_r (submatrix A I UNIV)⟩]
            unfolding submatrix-index[OF ⟨k < card {j. j < dim_r A ∧ j ∈ I}⟩ ⟨j <
  dim_c (submatrix A I UNIV)⟩[unfolded dim-submatrix]]
            unfolding pick-UNIV cols-nth[OF ⟨j < dim_c A⟩] col-def vec-index-vec[OF
  pick-le[OF ⟨k < card {j. j < dim_r 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*:  $set (cols (submatrix A UNIV J)) \subseteq set (cols A)$   
**proof**  
**fix**  $c$  **assume**  $c \in set (cols (submatrix A UNIV J))$   
**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**  $j < dim_c (submatrix A UNIV J)$  **by** *simp*  
**then have**  $j < card \{j. j < dim_c A \wedge j \in J\}$  **by** (*simp add: dim-submatrix(2)*)  
**have**  $cols (submatrix A UNIV J) ! j = cols A ! (pick J j)$   
**unfolding**  $cols\_nth[OF \langle j < dim_c (submatrix A UNIV J) \rangle] cols\_nth[OF pick\_le[OF \langle j < card \{j. j < dim_c A \wedge j \in J\} \rangle]]$   
**proof** (*rule vec-eqI*)  
**show**  $dim_v (col (submatrix A UNIV J) j) = dim_v (col A (pick J j))$  **unfolding** *dim-col dim-submatrix* **by** *auto*  
**fix**  $i$  **assume**  $i < dim_v (col A (pick J j))$   
**then have**  $i < dim_r A$  **by** *simp*  
**then have**  $i < dim_r (submatrix A UNIV J)$  **using**  $\langle dim_v (col (submatrix A UNIV J) j) = dim_v (col A (pick J j)) \rangle$  **by** *auto*  
**show**  $col (submatrix A UNIV J) j \$ i = col A (pick J j) \$ i$   
**unfolding** *col-def vec-index-vec[OF \langle i < dim\_r (submatrix A UNIV J) \rangle] vec-index-vec[OF \langle i < dim\_r A \rangle]*  
**using** *submatrix-index* **by** (*metis (no-types, lifting) \langle dim\_v (col (submatrix A UNIV J) j) = dim\_v (col A (pick J j)) \rangle \langle i < dim\_v (col A (pick J j)) \rangle \langle j < dim\_c (submatrix A UNIV J) \rangle dim-col dim-submatrix(1) dim-submatrix(2) pick-UNIV*)  
**qed**  
**then show**  $c \in set (cols A)$   
**using**  $\langle cols (submatrix A UNIV J) ! j = c \rangle$   
**using** *pick-le[OF \langle j < card \{j. j < dim\_c A \wedge j \in J\} \rangle]* **by** (*metis cols-length nth-mem*)  
**qed**

**lemma** (*in vec-space*) *lin-dep-submatrix-UNIV*:  
**assumes**  $A \in carrier_m n nc$   
**assumes** *lin-dep* ( $set (cols A)$ )  
**assumes** *distinct* ( $cols (submatrix A I UNIV)$ )  
**shows** *LinearCombinations.module.lin-dep*  $F (module_v TYPE('a) (card \{i. i < n \wedge i \in I\})) (set (cols (submatrix A I UNIV)))$   
 (*is LinearCombinations.module.lin-dep*  $F ?M (set ?S')$ )  
**proof** –  
**obtain**  $v$  **where**  $2:v \in carrier_v nc$  **and**  $3:v \neq \mathbf{0}_v nc$  **and**  $A \otimes_{mv} v = \mathbf{0}_v n$   
**using** *vec-space.lin-depE[OF assms(1) assms(2) distinct-cols-submatrix-UNIV[OF assms(3)]]* **by** *auto*  
**have**  $1: submatrix A I UNIV \in carrier_m (card \{i. i < n \wedge i \in I\}) nc$   
**apply** (*rule mat-carrierI*) **unfolding** *dim-submatrix* **using**  $\langle A \in carrier_m n nc \rangle$  **by** *auto*  
**have**  $4: submatrix A I UNIV \otimes_{mv} v = \mathbf{0}_v (card \{i. i < n \wedge i \in I\})$   
**proof** (*rule vec-eqI*)  
**show**  $dim\_eq: dim_v (submatrix A I UNIV \otimes_{mv} v) = dim_v (\mathbf{0}_v (card \{i. i < n$

$\wedge i \in I$ ) using 1 by auto  
**fix**  $i$  **assume**  $i < \dim_v (\mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\}))$   
**then have**  $i\text{-le}: i < \text{card } \{i. i < n \wedge i \in I\}$  **by auto**  
**have**  $(\text{submatrix } A \ I \ UNIV \ \otimes_{mv} \ v) \ \$ \ i = \text{row } (\text{submatrix } A \ I \ UNIV) \ i \cdot v$   
**using**  $\text{dim-eq } i\text{-le}$  **by auto**  
**also have**  $\dots = \text{row } A \ (\text{pick } I \ i) \cdot v$  **using**  $\text{row-submatrix-UNIV}$   
**by**  $(\text{metis } (\text{no-types, lifting}) \ \text{dim-eq } \text{dim-mat-mult-vec } \text{dim-submatrix}(1) \ \langle i < \dim_v (\mathbf{0}_v (\text{card } \{i. i < n \wedge i \in I\})) \rangle)$   
**also have**  $\dots = 0$   
**using**  $\langle A \ \otimes_{mv} \ v = \mathbf{0}_v \ n \rangle \ i\text{-le}[\text{THEN pick-le}]$  **by**  $(\text{metis } \text{assms}(1) \ \text{index-mat-mult-vec } \text{mat-carrierD}(1) \ \text{vec-zero-index}(1))$   
**also have**  $\dots = \mathbf{0}_v \ (\text{card } \{i. i < n \wedge i \in I\}) \ \$ \ i$  **by**  $(\text{simp add: } i\text{-le})$   
**finally show**  $(\text{submatrix } A \ I \ UNIV \ \otimes_{mv} \ v) \ \$ \ i = \mathbf{0}_v \ (\text{card } \{i. i < n \wedge i \in I\}) \ \$ \ i$  **by metis**  
**qed**  
**show**  $?thesis$  **using**  $\text{vec-space.lin-depI}[OF \ 1 \ 2 \ 3 \ 4]$  **using**  $\text{assms}(3)$  **by auto**  
**qed**

**lemma** (in  $\text{vec-space}$ )  $\text{rank-gt-minor}$ :

**assumes**  $A \in \text{carrier}_m \ n \ nc$

**assumes**  $\det (\text{submatrix } A \ I \ J) \neq 0$

**shows**  $\text{card } \{j. j < nc \wedge j \in J\} \leq \text{rank } A$

**proof** –

**have**  $\text{square: dim}_r (\text{submatrix } A \ I \ J) = \text{dim}_c (\text{submatrix } A \ I \ J)$

**using**  $\text{det-def } \langle \det (\text{submatrix } A \ I \ J) \neq 0 \rangle$  **by metis**

**then have**  $\text{full-rank: vec-space.rank } (\text{dim}_r (\text{submatrix } A \ I \ J)) (\text{submatrix } A \ I \ J) = \text{dim}_r (\text{submatrix } A \ I \ J)$

**using**  $\text{vec-space.low-rank-det-zero } \text{assms}(2) \ \text{mat-carrierI}$  **by auto**

**then have**  $\text{distinct: distinct } (\text{cols } (\text{submatrix } A \ I \ J))$  **using**  $\text{vec-space.non-distinct-low-rank}$

**using**  $\text{square less-irrefl } \text{mat-carrierI}$  **by fastforce**

**then have**  $\text{indpt: LinearCombinations.module.lin-indpt } F \ (\text{module}_v \ \text{TYPE}('a) \ (\text{dim}_r (\text{submatrix } A \ I \ J))) \ (\text{set } (\text{cols } (\text{submatrix } A \ I \ J)))$

**using**  $\text{vec-space.full-rank-lin-indpt}[OF \ - \ \text{full-rank } \text{distinct}] \ \text{square}$  **by fastforce**

**have**  $\text{distinct2: distinct } (\text{cols } (\text{submatrix } (\text{submatrix } A \ UNIV \ J) \ I \ UNIV))$  **using**  $\text{submatrix-split } \text{distinct}$  **by metis**

**have**  $\text{indpt2: LinearCombinations.module.lin-indpt } F \ (\text{module}_v \ \text{TYPE}('a) \ (\text{card } \{i. i < n \wedge i \in I\})) \ (\text{set } (\text{cols } (\text{submatrix } (\text{submatrix } A \ UNIV \ J) \ I \ UNIV)))$

**using**  $\text{submatrix-split } \text{dim-submatrix}(1) \ \text{indpt}$  **by**  $(\text{metis } (\text{full-types}) \ \text{assms}(1) \ \text{mat-carrierD}(1))$

**have**  $\text{submatrix } A \ UNIV \ J \in \text{carrier}_m \ n \ (\text{dim}_c (\text{submatrix } A \ UNIV \ J))$

**apply**  $(\text{rule } \text{mat-carrierI})$  **unfolding**  $\text{dim-submatrix}(1)$  **using**  $\langle A \in \text{carrier}_m \ n \ nc \rangle \ \text{mat-carrierD}$  **by simp-all**

**have**  $\text{lin-indpt } (\text{set } (\text{cols } (\text{submatrix } A \ UNIV \ J)))$

**using**  $\text{indpt2 } \text{vec-space.lin-dep-submatrix-UNIV}[OF \ \langle \text{submatrix } A \ UNIV \ J \in \text{carrier}_m \ n \ (\text{dim}_c (\text{submatrix } A \ UNIV \ J)) \rangle \ - \ \text{distinct2}]$  **by blast**

**have**  $\text{distinct3: distinct } (\text{cols } (\text{submatrix } A \ UNIV \ J))$  **by**  $(\text{metis } \text{distinct } \text{distinct-cols-submatrix-UNIV})$

```

submatrix-split)
  show ?thesis using
    rank-ge-card-indpt[OF (A ∈ carrierm n nc) cols-submatrix-subset (lin-indpt (set
      (cols (submatrix A UNIV J))))],
    unfolded distinct-card[OF distinct3, unfolded cols-length dim-submatrix], un-
      folded mat-carrierD(2)[OF (A ∈ carrierm 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) = repli-
  cate (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.prem(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 < dimv (tensors-from-net (insert-weights (shallow-model' Z M
N) ( $\lambda i. w (i + (count-weights (shallow-model' Z M 0))))$ )))
    using output-size-correct-tensors[of insert-weights (shallow-model' Z M N) ( $\lambda i.$ 
w (i + (count-weights (shallow-model' Z M 0))))),
      unfolded remove-insert-weights, OF valid-shallow-model'
    using Suc.prem(1) output-size-shallow-model' by auto
  have cprank-max1-IH:cprank-max1 (tensors-from-net (insert-weights (shallow-model'
Z M N) ( $\lambda i. w (i + (count-weights (shallow-model' Z M 0))))$ )) $ y)
    using Suc.IH Suc.prem(1) output-size-shallow-model' by auto
  have y-le-0:y < dimv (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:  $\dim_r$  (extract-matrix w Y Z) = Y
    using dim-extract-matrix(1) by force
    have dimc-extract-matrix:  $\dim_c$  (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) ( $\lambda i. w$ 
(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 ( $\lambda j. (extract-matrix w Y Z) \text{ \&\& } (y, j) \cdot (tensors-from-net (insert-weights
(shallow-model' Z M N) ( $\lambda i. w$  (i + Y * Z)))) \text{ \&\& } 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) ( $\lambda i. w$  (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 ( $\lambda j. extract-matrix w Y Z \text{ \&\& } (y, j) \cdot (tensors-from-net
(insert-weights (shallow-model' Z M N) ( $\lambda i. w$  (i + Y * Z)))) \text{ \&\& } j) [0..<Z]$ )

  have  $\bigwedge B. B \in \text{set } Bs \implies \text{cprank-max1 } B \bigwedge B. B \in \text{set } Bs \implies \text{dims } B =$ 
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) ( $\lambda i. w$  (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 $\text{ \&\& } (y, j) \cdot (tensors-from-net
(insert-weights (shallow-model' Z M N) ( $\lambda i. w$  (i + Y * Z)))) \text{ \&\& } j)
      using smult-prod-extract1 cprank-max1-order0[OF 1, of extract-matrix w Y
Z $\text{ \&\& } (y, j) \cdot 1$]
      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 Missing Lemmas of Complete\_Measure

```

theory DL-Missing-Complete-Measure
imports ~/src/HOL/Probability/Complete-Measure
begin

lemma null-sets-completion-subset:
assumes  $A \in \text{null-sets (completion } M)$   $B \subseteq A$ 
shows  $B \in \text{null-sets (completion } M)$ 
proof -
  obtain  $N$  where  $N \in \text{null-sets } M$   $\text{null-part } M \ A \subseteq N$ 
  using null-part[OF null-setsD2[OF assms(1)]] by blast
  then have  $B \subseteq \text{main-part } M \ A \cup N$ 
  using main-part-null-part-Un[OF null-setsD2[OF assms(1)]]  $\langle B \subseteq A \rangle$  by auto
  have  $\text{main-part } M \ A \in \text{null-sets } M$  using emeasure-completion[OF null-setsD2[OF
  assms(1)]]
  by (metis assms(1) main-part-sets null-setsD1 null-setsD2 null-setsI)
  then have  $\text{main-part } M \ A \cup N \in \text{null-sets } M$  using  $\langle N \in \text{null-sets } M \rangle$  by blast
  then show ?thesis using null-sets-completion
  by (metis  $\langle B \subseteq \text{main-part } M \ A \cup N \rangle$  assms(1) assms(2) emeasure-Un-null-set
  le-iff-sup null-setsD1 null-setsI)
  qed
end

```

### 39 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
~/src/HOL/Probability/Complete-Measure DL-Shallow-Model
DL-Missing-Complete-Measure
begin

context deep-model-correct-params-y
begin

theorem fundamental-theorem-network-capacity-polynomial:
obtains  $p$  where  $p \neq 0$  and  $\text{vars } p \subseteq \{..<\text{weight-space-dim}\}$ 
and  $\bigwedge x. \text{insertion } x \ p \neq 0 \implies r \wedge N\text{-half} \leq \text{cprank } (A \ x)$ 

```

**proof** –  
**assume** *assumption*:  $\bigwedge p. p \neq 0 \implies \text{vars } p \subseteq \{..<\text{weight-space-dim}\} \implies$   
 $(\bigwedge x. \text{insertion } x \ p \neq 0 \implies r \wedge N\text{-half} \leq \text{cprank } (A \ x)) \implies \text{thesis}$   
**obtain** *p* **where** *p-def*:  $\bigwedge x. \text{insertion } x \ p = \text{Determinant.det } (\text{witness-submatrix } y \ x)$  **and**  
 $\text{vars-}p: \text{vars } p \subseteq \{..<\text{weight-space-dim}\}$   
**using** *polyfun-det-deep-model*[*unfolded polyfun-def*] **by** *auto*  
**then have** *insertion witness-weights*  $p \neq 0$   
**using** *witness-det*[*unfolded Aw'-def*] *witness-submatrix-def weight-space-dim-def*  
**by** (*metis p-def*)  
**then have**  $p \neq 0$  **by** *auto*  
**have**  $\bigwedge x. \text{insertion } x \ p \neq 0 \implies r \wedge N\text{-half} \leq \text{cprank } (A \ x)$   
**proof** –  
**fix** *x* **assume** *insertion x p*  $\neq 0$   
**then have**  $\text{Determinant.det } (\text{witness-submatrix } y \ x) \neq 0$  **using** *p-def* **by** *auto*  
**have**  $0: \text{weight-space-dim} \leq \text{length } (\text{map } x \ [0..<\text{weight-space-dim}])$  **by** *auto*  
**have**  $r \wedge N\text{-half} \leq \text{mrank } (A' \ x)$   
**using** *vec-space.rank-gt-minor*[*OF mat-carrierI*[*OF dims-A'-pow, unfolded weight-space-dim-def*]  
 $\langle \text{Determinant.det } (\text{witness-submatrix } y \ x) \neq 0 \rangle$ [*unfolded witness-submatrix-def*]]  
*card-rows-with-1*[*unfolded dims-Aw'-pow*] **by** (*metis (no-types, lifting) Collect-cong*  
*dims-A'-pow(1)*)  
**also have**  $\dots \leq \text{cprank } (A \ x)$  **using** *matrix-rank-le-cp-rank A'-def* **by** *auto*  
**finally show**  $r \wedge N\text{-half} \leq \text{cprank } (A \ x)$  .  
**qed**  
**then show** *?thesis* **using** *assumption*  $\langle p \neq 0 \rangle$   $\langle \text{vars } p \subseteq \{..<\text{weight-space-dim}\} \rangle$   
**by** *blast*  
**qed**

**theorem** *fundamental-theorem-network-capacity*:  
 $AE \ x \text{ in } \text{lborel-f weight-space-dim}. r \wedge N\text{-half} \leq \text{cprank } (A \ x)$   
**proof** –  
**obtain** *p* **where**  $p \neq 0$  **and**  $\text{vars } p \subseteq \{..<\text{weight-space-dim}\}$   
**and**  $\bigwedge x. \text{insertion } x \ p \neq 0 \implies r \wedge N\text{-half} \leq \text{cprank } (A \ x)$   
**using** *fundamental-theorem-network-capacity-polynomial* **by** *metis*  
**have**  $\text{null-set}: \{f \in \text{space } (\text{lborel-f weight-space-dim}). \text{insertion } f \ p = 0\} \in \text{null-sets}$   
 $(\text{lborel-f weight-space-dim})$   
**using** *lebesgue-mpoly-zero-set*[*OF*  $\langle p \neq 0 \rangle$   $\langle \text{vars } p \subseteq \{..<\text{weight-space-dim}\} \rangle$ ] **by**  
*simp*  
**have**  $\text{subset}: \{x \in \text{space } (\text{lborel-f weight-space-dim}). r \wedge N\text{-half} > \text{cprank } (A \ x)\}$   
 $\subseteq \{f \in \text{space } (\text{lborel-f weight-space-dim}). \text{insertion } f \ p = 0\}$   
**proof**  
**fix** *x* **assume**  $x \in \{x \in \text{space } (\text{lborel-f weight-space-dim}). r \wedge N\text{-half} > \text{cprank}$   
 $(A \ x)\}$   
**then have**  $x \in \text{space } (\text{lborel-f weight-space-dim}) \ r \wedge N\text{-half} > \text{cprank } (A \ x)$  **by**  
*auto*  
**then have**  $\text{insertion } x \ p = 0$  **using**  $\langle \bigwedge x. \text{insertion } x \ p \neq 0 \implies r \wedge N\text{-half} \leq$   
 $\text{cprank } (A \ x) \rangle$   
*HOL.contra-pos-nn not-le* **by** *metis*

**then show**  $x \in \{f \in \text{space } (\text{lborel-f weight-space-dim}). \text{insertion } f p = 0\}$  **using**  
 $\langle x \in \text{space } (\text{lborel-f weight-space-dim}) \rangle$  **by** *blast*  
**qed**  
**then show** *?thesis* **using** *AE-I'[OF null-set, of  $\lambda x. r \wedge N\text{-half} \leq \text{cprank } (A x)$ ]*  
 $leI$  **by** *blast*  
**qed**

**end**

**context** *deep-model-correct-params*  
**begin**

**theorem** *null-set-tensors-equal:*

**shows** *AE weights-deep in lborel-f weight-space-dim.*

$\neg (\exists \text{weights-shallow } Z. Z < r \wedge N\text{-half} \wedge$   
 $\text{tensors-from-net } (\text{insert-weights } (\text{deep-model-l } rs) \text{ weights-deep})$   
 $= \text{tensors-from-net } (\text{insert-weights } (\text{shallow-model } (rs ! 0) Z (\text{last } rs) (2 * N\text{-half}$   
 $- 1)) \text{ weights-shallow})$   
*(is almost-everywhere ?M ?P)*

**proof** –

**have**  $0 < rs ! 0$  **using** *no-zeros deep* **by** (*metis length-0-conv length-greater-0-conv not-numeral-le-zero nth-mem*)

**have**  $\bigwedge x. \neg ?P x \implies \text{cprank } (\text{deep-model-correct-params-y.A } rs 0 x) < r \wedge N\text{-half}$

**proof** –

**fix**  $x$  **assume**  $\neg ?P x$

**obtain** *weights-shallow*  $Z$  **where**  $Z < r \wedge N\text{-half}$

$\text{tensors-from-net } (\text{insert-weights } (\text{deep-model-l } rs) x) =$

$\text{tensors-from-net } (\text{insert-weights } (\text{shallow-model } (rs ! 0) Z (\text{last } rs) (2 * N\text{-half} - 1)) \text{ weights-shallow})$

**using**  $\langle \neg ?P x \rangle$  **by** *blast*

**then have**  $\text{cprank } (\text{tensors-from-net } (\text{insert-weights } (\text{deep-model-l } rs) x) \$ 0) < r \wedge N\text{-half}$

**using** *cprank-shallow-model  $\langle 0 < rs ! 0 \rangle$  remove-insert-weights* **by** (*metis not-le order-trans*)

**then show**  $\text{cprank } (\text{deep-model-correct-params-y.A } rs 0 x) < r \wedge N\text{-half}$

**by** (*simp add:  $\langle 0 < rs ! 0 \rangle$  deep-model-correct-params-axioms*

*deep-model-correct-params-y.A-def deep-model-correct-params-y.intro deep-model-correct-params-y-axioms.*

**qed**

**then show** *?thesis* **using** *Filter.eventually-mono* **using** *deep-model-correct-params-y.fundamental-theorem-network-capacity-v2: deep-model-correct-params-y.intro,*

*OF deep-model-correct-params-axioms, unfolded deep-model-correct-params-y-axioms-def, OF  $\langle 0 < rs ! 0 \rangle$ ]*

**by** *fastforce*

**qed**

**theorem** *fundamental-theorem-network-capacity-v2:*

**shows** *AE weights-deep in lborel-f weight-space-dim.*

```

  ¬(∃ weights-shallow Z. Z < r ^ N-half ∧ (∀ inputs. map dim_v inputs = input-sizes
(deep-model-l rs) →
evaluate-net (insert-weights (deep-model-l rs) weights-deep) inputs
= evaluate-net (insert-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half-1))
weights-shallow) inputs))
(is almost-everywhere ?M ?P)
proof (rule Filter.eventually-mono)
  show AE weights-deep in lborel-f weight-space-dim.
  ¬(∃ weights-shallow Z. Z < r ^ N-half ∧
tensors-from-net (insert-weights (deep-model-l rs) weights-deep)
= tensors-from-net (insert-weights (shallow-model (rs ! 0) Z (last rs) (2*N-half
-1)) weights-shallow))
(is almost-everywhere ?M ?P')
using null-set-tensors-equal by metis
have ∧x. ¬?P x ⇒ ¬?P' x
proof -
  fix weights-deep assume ¬?P weights-deep
  then obtain weights-shallow Z where 0:Z < r ^ N-half
  and eval-eq:∧inputs. map dim_v inputs = input-sizes (deep-model-l rs) ⇒
evaluate-net (insert-weights (deep-model-l rs) weights-deep) inputs
= evaluate-net (insert-weights (shallow-model (rs ! 0) Z (last rs)
(2*N-half-1)) weights-shallow) inputs
  by blast
  have 2 ≤ length rs using deep by linarith
  have tensors-from-net (insert-weights (deep-model-l rs) weights-deep) =
tensors-from-net (insert-weights (shallow-model (rs ! 0) Z (last rs) (2 *
N-half - 1)) weights-shallow)
  using tensors-from-net-eqI[OF - - - eval-eq, unfolded remove-insert-weights,
OF valid-deep-model valid-shallow-model]
input-sizes-deep-model[OF (2 ≤ length rs)] input-sizes-shallow-model input-sizes-remove-weights
remove-insert-weights
  by (metis (no-types, lifting) N-half-def Suc-diff-1 Suc-diff-Suc Suc-le-lessD deep
numeral-2-eq-2 numeral-3-eq-3 power-Suc power-eq-0-iff zero-less-power zero-power2)
  then show ¬?P' weights-deep
  using 0 by blast
  qed
  then show ∧x. ?P' x ⇒ ?P x by blast
qed
end

thm deep-model-correct-params.fundamental-theorem-network-capacity-v2
end

```

## References

- [1] A. Bentkamp. An Isabelle Formalization of the Expressiveness of Deep Learning. Master's thesis, Universität des Saarlandes, 2016.

- [2] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis.