Expressiveness of Deep Learning

Alexander Bentkamp

October 11, 2017

Abstract

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

Contents

1 Tensor **3**

2 Subtensors **7**

3 Tensor Addition **8**

4 Tensor Scalar Multiplication **11**

5 Tensor Product **12**

6 Unit Vectors as Tensors **15**

7 Tensor CP-Rank **15**

8 Missing Lemmas of Vector_Space **17**

9 Missing Lemmas of VS_Connect **18**

10 Missing Lemmas of List **18**

11 Matrix Rank **19**

12 Subadditivity of rank **22**

13 Missing Lemmas of Sublist **23**

14 Pick **24**

15 Sublist **25**

16 weave **26**

17 Tensor Matricization **27**

18 Submatrices **29**

19 Submatrix **29**

20 CP-Rank and Matrix Rank **30**

21 Missing Lemmas of Matrix **31**

22 Matrix to Vector Conversion **32**

23 Deep Learning Networks **33**

24 Concrete Matrices **36**

25 Missing Lemmas of Finite_Set **38**

26 Deep Network Model **38**

27 Less common functions on lists **45**

28 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view **49**
28.1 Preliminary: auxiliary operations for 'almost everywhere zero' 50
28.2 Type definition 53
28.3 Additive structure 54
28.4 Multiplicative structure 56
28.5 Single-point mappings 58
28.6 Integral domains 59
28.7 Mapping order 60
28.8 Fundamental mapping notions 60
28.9 Degree 62
28.10Inductive structure 63
28.11Quasi-functorial structure 63
28.12Canonical dense representation of $nat \Rightarrow_0 'a$ 65
28.13Canonical sparse representation of $'a \Rightarrow_0 'b$ 66

28.14Size estimation 67
28.15Further mapping operations and properties 68

29 An abstract type for multivariate polynomials 69
29.1 Abstract type definition 69
29.2 Additive structure 69
29.3 Multiplication by a coefficient 70
29.4 Multiplicative structure 70
29.5 Monomials 71
29.6 Integral domains 73
29.7 Monom coefficient lookup 73
29.8 Insertion morphism 73
29.9 Degree 74
29.10Pseudo-division of polynomials 75
29.11Primitive poly, etc 76

30 MPpoly Mapping extenion 77

31 MPoly extension 78

32 Nested MPoly 80

33 Polynomials representing the Deep Network Model 83

34 Alternative Lebesgue Measure Definition 86

35 Lebesgue Measure of Polynomial Zero Sets 89

36 Rank and Submatrices 89

37 Shallow Network Model 90

38 Fundamental Theorem of Network Capacity 91

1 Tensor

theory *Tensor*
imports *Main*
begin

typedef *'a tensor* = {*t*::*nat list* × *'a list*. *length* (*snd t*) = *prod-list* (*fst t*)}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *tensor-from-vec-simp*[*simp*]: *tensor-from-vec* (*dims A*) (*vec A*) = *A*
⟨*proof*⟩

lemma *length-vec*: *length* (*vec A*) = *prod-list* (*dims A*)
⟨*proof*⟩

lemma *tensor-eqI*[*intro*]:
assumes *dims A* = *dims B* **and** *vec A* = *vec B*
shows *A*=*B*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *valid-index-lt*: *is* ◁ *ds* ⟹ *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
⟨*proof*⟩

lemma *valid-indexI*:
assumes *length is* = *length ds* **and** ⋀*m*. *m*<*length ds* ⟹ *is*!*m* < *ds*!*m*
shows *is* ◁ *ds*
⟨*proof*⟩

lemma *valid-index-append*:
assumes *is1-valid*:*is1* ◁ *ds1* **and** *is2-valid*:*is2* ◁ *ds2*
shows *is1* @ *is2* ◁ *ds1* @ *ds2*
⟨*proof*⟩

lemma *valid-index-list-all2-iff*: $is \lhd ds \longleftrightarrow$ *list-all2* $(op <)$ *is ds*
⟨*proof*⟩

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

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

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

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

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

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

lemma *concat-parts-eq*:
assumes *a* ∗ *d* = *length v*
shows *concat* (*map* (*fixed-length-sublist v d*) [*0*..<*a*]) = *v*
⟨*proof*⟩

lemma *tensor-lookup-base*:
assumes *length v* = *prod-list ds*
and ⋀*is*. *is* ◁ *ds* ⟹ *lookup-base ds v is* = *e is*
shows *tensor-vec-from-lookup ds e* = *v*
⟨*proof*⟩

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

lemma *concat-equal-length*:
assumes ⋀*xs*. *xs*∈*set xss* ⟹ *length xs* = *l*
shows *length* (*concat xss*) = *length xss*∗*l*
⟨*proof*⟩

lemma *concat-equal-length-map*:
assumes $\bigwedge i.\ i{<}a \Longrightarrow \mathit{length}\ (f\ i) = d$
shows $\mathit{length}\ (\mathit{concat}\ (\mathit{map}\ (\lambda i.\ f\ i)\ [0..{<}a])) = a{*}d$
⟨*proof*⟩

lemma *concat-parts*:
assumes $\bigwedge xs.\ xs{\in}\mathit{set}\ \mathit{xss} \Longrightarrow \mathit{length}\ \mathit{xs} = d$ **and** $i{<}\mathit{length}\ \mathit{xss}$
shows $\mathit{fixed\text{-}length\text{-}sublist}\ (\mathit{concat}\ \mathit{xss})\ d\ i = \mathit{xss}\ !\ i$
⟨*proof*⟩

lemma *concat-parts'*:
assumes $\bigwedge i.\ i{<}a \Longrightarrow \mathit{length}\ (f\ i) = d$
and $i{<}a$
shows $\mathit{fixed\text{-}length\text{-}sublist}\ (\mathit{concat}\ (\mathit{map}\ (\lambda i.\ f\ i)\ [0..{<}a]))\ d\ i = f\ i$
⟨*proof*⟩

lemma *length-tensor-vec-from-lookup*:
$\mathit{length}\ (\mathit{tensor\text{-}vec\text{-}from\text{-}lookup}\ \mathit{ds}\ e) = \mathit{prod\text{-}list}\ \mathit{ds}$
⟨*proof*⟩

lemma *lookup-tensor-vec*:
assumes $\mathit{is}{\lhd}\mathit{ds}$
shows $\mathit{lookup\text{-}base}\ \mathit{ds}\ (\mathit{tensor\text{-}vec\text{-}from\text{-}lookup}\ \mathit{ds}\ e)\ \mathit{is} = e\ \mathit{is}$
⟨*proof*⟩

lemma *lookup-tensor-from-lookup*:
assumes $\mathit{is}{\lhd}\mathit{ds}$
shows $\mathit{lookup}\ (\mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e)\ \mathit{is} = e\ \mathit{is}$
⟨*proof*⟩

lemma *dims-tensor-from-lookup*: $\mathit{dims}\ (\mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e) = \mathit{ds}$
⟨*proof*⟩

lemma *tensor-lookup-cong*:
assumes $\mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e_1 = \mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e_2$
and $\mathit{is}{\lhd}\mathit{ds}$
shows $e_1\ \mathit{is} = e_2\ \mathit{is}$ ⟨*proof*⟩

lemma *tensor-from-lookup-eqI*:
assumes $\bigwedge \mathit{is}.\ \mathit{is}{\lhd}\mathit{ds} \Longrightarrow e_1\ \mathit{is} = e_2\ \mathit{is}$
shows $\mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e_1 = \mathit{tensor\text{-}from\text{-}lookup}\ \mathit{ds}\ e_2$
⟨*proof*⟩

lemma *tensor-lookup-eqI*:
assumes $\mathit{dims}\ A = \mathit{dims}\ B$ **and** $\bigwedge \mathit{is}.\ \mathit{is}{\lhd}(\mathit{dims}\ A) \Longrightarrow \mathit{lookup}\ A\ \mathit{is} = \mathit{lookup}\ B\ \mathit{is}$
shows $A = B$ ⟨*proof*⟩

end

2 Subtensors

theory *Tensor-Subtensor*
imports *Tensor*
begin

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

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

lemma *length-fixed-length-sublist*[*simp*]:
assumes (*Suc i*)∗*l* ≤ *length xs*
shows *length* (*fixed-length-sublist xs l i*) = *l*
⟨*proof*⟩

lemma *vec-subtensor*[*simp*]:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *vec* (*subtensor A i*) = *fixed-length-sublist* (*vec A*) (*prod-list* (*tl* (*dims A*))) *i*
⟨*proof*⟩

lemma *dims-subtensor*[*simp*]:
assumes *dims A* ≠ [] **and** *i* < *hd* (*dims A*)
shows *dims* (*subtensor A i*) = *tl* (*dims A*)
⟨*proof*⟩

lemma *subtensor-combine-subtensor*[*simp*]:
assumes *dims A* ≠ []
shows *subtensor-combine* (*tl* (*dims A*)) (*map* (*subtensor A*) [*0*..<*hd* (*dims A*)]) = *A*
⟨*proof*⟩

lemma
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *subtensor-combine-dims*[*simp*]: *dims* (*subtensor-combine ds As*) = *length As* # *ds* (**is** *?D*)
and *subtensor-combine-vec*[*simp*]: *vec* (*subtensor-combine ds As*) = *concat* (*map vec As*) (**is** *?V*)
⟨*proof*⟩

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

lemma *subtensor-induct*[*case-names order-0 order-step*]:

assumes *order-0*: $\bigwedge A.\ dims\ A = [] \Longrightarrow P\ A$
and *order-step*: $\bigwedge A.\ dims\ A \neq [] \Longrightarrow (\bigwedge i.\ i < hd\ (dims\ A) \Longrightarrow P\ (subtensor\ A\ i)) \Longrightarrow P\ A$
shows *P B*
⟨*proof*⟩

lemma *subtensor-combine-induct*[*case-names order-0 order-step*]:
assumes *order-0*:$\bigwedge A.\ dims\ A = [] \Longrightarrow P\ A$
and *order-step*:$\bigwedge As\ ds.\ (\bigwedge A.\ A{\in}set\ As \Longrightarrow P\ A) \Longrightarrow (\bigwedge A.\ A{\in}set\ As \Longrightarrow dims\ A = ds) \Longrightarrow P\ (subtensor\text{-}combine\ ds\ As)$
shows *P A*
⟨*proof*⟩

lemma *lookup-subtensor1*[*simp*]:
assumes $i\ \#\ is \lhd dims\ A$
shows *lookup* (*subtensor A i*) *is* = *lookup A* (*i # is*)
⟨*proof*⟩

lemma *lookup-subtensor*:
assumes $is \lhd dims\ A$
shows *lookup A is* = *hd* (*vec* (*fold* (λ*i A. subtensor A i*) *is A*))
⟨*proof*⟩

lemma *subtensor-eqI*:
assumes $dims\ A \neq []$
and *dims-eq*:*dims A* = *dims B*
and $\bigwedge i.\ i < hd\ (dims\ A) \Longrightarrow subtensor\ A\ i = subtensor\ B\ i$
shows *A*=*B*
⟨*proof*⟩

end

3 Tensor Addition

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

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

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

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

else undefined)
instance ⟨*proof*⟩
end

lemma *plus-dim1*[*simp*]: *dims A* = *dims B* ⟹ *dims* (*A* + *B*) = *dims A* ⟨*proof*⟩
lemma *plus-dim2*[*simp*]: *dims A* = *dims B* ⟹ *dims* (*A* + *B*) = *dims B* ⟨*proof*⟩
lemma *plus-base*: *dims A* = *dims B* ⟹ *A* + *B* = *plus-base A B* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *vec-plus*[*simp*]:
assumes *dims A* = *dims B*
shows *vec* (*A*+*B*) = *vec-plus* (*vec A*) (*vec B*)
⟨*proof*⟩

lemma *subtensor-plus*:
fixes *A*::′*a*::*semigroup-add tensor* **and** *B*::′*a*::*semigroup-add tensor*
assumes *i* < *hd* (*dims A*)
and *dims A* = *dims B*
and *dims A* ≠ []
shows *subtensor* (*A* + *B*) *i* = *subtensor A i* + *subtensor B i*
⟨*proof*⟩

lemma *lookup-plus*[*simp*]:
assumes *dims A* = *dims B*
and *is* ◁ *dims A*
shows *lookup* (*A* + *B*) *is* = *lookup A is* + *lookup B is*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *tensor-comm*[*simp*]:
fixes *A*::′*a*::*ab-semigroup-add tensor*
shows *A* + *B* = *B* + *A*
⟨*proof*⟩

definition *vec0 n* = *replicate n 0*

definition *tensor0*::*nat list* ⇒ ′*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*)
⟨*proof*⟩

lemma *lookup-is-in-vec*: *is* ◁ (*dims A*) ⟹ *lookup A is* ∈ *set* (*vec A*)
⟨*proof*⟩

lemma *lookup-tensor0*:
assumes *is* ◁ *ds*
shows *lookup* (*tensor0 ds*) *is* = *0*
⟨*proof*⟩

lemma
fixes *A*::'*a*::*monoid-add tensor*
shows *tensor-add-0-right*[*simp*]: *A* + *tensor0* (*dims A*) = *A*
⟨*proof*⟩

lemma
fixes *A*::'*a*::*monoid-add tensor*
shows *tensor-add-0-left*[*simp*]: *tensor0* (*dims A*) + *A* = *A*
⟨*proof*⟩

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

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

lemma *listsum-Nil*: *listsum ds* [] = *tensor0 ds* ⟨*proof*⟩

lemma *listsum-one*: *listsum* (*dims A*) [*A*] = *A* ⟨*proof*⟩

lemma *listsum-Cons*: *listsum ds* (*A* # *As*) = *A* + *listsum ds As*
⟨*proof*⟩

lemma *listsum-dims*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
shows *dims* (*listsum ds As*) = *ds*
⟨*proof*⟩

lemma *subtensor0*:
assumes *ds* ≠ [] **and** *i*<*hd ds*
shows *subtensor* (*tensor0 ds*) *i* = *tensor0* (*tl ds*)
⟨*proof*⟩

lemma *subtensor-listsum*:
assumes ⋀*A*. *A*∈*set As* ⟹ *dims A* = *ds*
and *ds* ≠ [] **and** *i*<*hd ds*
shows *subtensor* (*listsum ds As*) *i* = *listsum* (*tl ds*) (*map* (λ*A*. *subtensor A i*) *As*)

⟨*proof*⟩

lemma *listsum0*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow A = tensor0\ ds$
shows $listsum\ ds\ As = tensor0\ ds$
⟨*proof*⟩

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

lemma *lookup-listsum*:
assumes $is \lhd ds$
and $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows $lookup\ (listsum\ ds\ As)\ is = (\sum A \leftarrow As.\ lookup\ A\ is)$
⟨*proof*⟩

end

4 Tensor Scalar Multiplication

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

definition $vec\text{-}smult :: 'a :: ring \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$vec\text{-}smult\ \alpha\ \beta = map\ (op * \alpha)\ \beta$

lemma *vec-smult0*: $vec\text{-}smult\ 0\ as = vec0\ (length\ as)$
⟨*proof*⟩

lemma *vec-smult-distr-right*:
shows $vec\text{-}smult\ (\alpha + \beta)\ as = vec\text{-}plus\ (vec\text{-}smult\ \alpha\ as)\ (vec\text{-}smult\ \beta\ as)$
⟨*proof*⟩

lemma *vec-smult-Cons*:
shows $vec\text{-}smult\ \alpha\ (a \#\ as) = (\alpha * a) \#\ vec\text{-}smult\ \alpha\ as$ ⟨*proof*⟩

lemma *vec-plus-Cons*:
shows $vec\text{-}plus\ (a \#\ as)\ (b \#\ bs) = (a+b) \#\ vec\text{-}plus\ as\ bs$ ⟨*proof*⟩

lemma *vec-smult-distr-left*:
assumes $length\ as = length\ bs$
shows $vec\text{-}smult\ \alpha\ (vec\text{-}plus\ as\ bs) = vec\text{-}plus\ (vec\text{-}smult\ \alpha\ as)\ (vec\text{-}smult\ \alpha\ bs)$

⟨*proof*⟩

lemma *length-vec-smult*: *length* (*vec-smult* α *v*) = *length v* ⟨*proof*⟩

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

lemma *tensor-smult0*: **fixes** *A*::'*a*::*ring tensor*
shows *0* · *A* = *tensor0* (*dims A*)
⟨*proof*⟩

lemma *dims-smult*[*simp*]:*dims* (α · *A*) = *dims A*
and *vec-smult*[*simp*]: *vec* (α · *A*) = *map* (*op* ∗ α) (*vec A*)
⟨*proof*⟩

lemma *tensor-smult-distr-right*: (α + β) · *A* = α · *A* + β · *A*
⟨*proof*⟩

lemma *tensor-smult-distr-left*: *dims A* = *dims B* ⟹ α · (*A* + *B*) = α · *A* + α · *B*
⟨*proof*⟩

lemma *smult-fixed-length-sublist*:
assumes *length xs* = *l* ∗ *c i*<*c*
shows *fixed-length-sublist* (*vec-smult* α *xs*) *l i* = *vec-smult* α (*fixed-length-sublist xs l i*)
⟨*proof*⟩

lemma *smult-subtensor*:
assumes *dims A* ≠ [] *i* < *hd* (*dims A*)
shows α · *subtensor A i* = *subtensor* (α · *A*) *i*
⟨*proof*⟩

lemma *lookup-smult*:
assumes *is* ◁ *dims A*
shows *lookup* (α · *A*) *is* = α ∗ *lookup A is*
⟨*proof*⟩

lemma *tensor-smult-assoc*:
fixes *A*::'*a*::*ring tensor*
shows α · (β · *A*) = (α ∗ β) · *A*
⟨*proof*⟩

end

5 Tensor Product

theory *Tensor-Product*

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

instantiation *tensor*:: (*ring*) *semigroup-mult*
begin
definition *tensor-prod-def*:$A * B = tensor\text{-}from\text{-}vec\ (dims\ A\ @\ dims\ B)\ (concat\ (map\ (\lambda a.\ vec\text{-}smult\ a\ (vec\ B))\ (vec\ A)))$

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*]: $vec\ (A \otimes B) = concat\ (map\ (\lambda a.\ vec\text{-}smult\ a\ (vec\ B))\ (vec\ A))$ (**is** *?V*)
and *dims-tensor-prod*[*simp*]: $dims\ (A \otimes B) = dims\ A\ @\ dims\ B$ (**is** *?D*)
⟨*proof*⟩

lemma *tensorprod-subtensor-base*:
shows $concat\ (map\ f\ (concat\ xss)) = concat\ (map\ (\lambda xs.\ concat\ (map\ f\ xs))\ xss)$
⟨*proof*⟩

lemma *subtensor-combine-tensor-prod*:
assumes $\bigwedge A.\ A \in set\ As \Longrightarrow dims\ A = ds$
shows $subtensor\text{-}combine\ ds\ As \otimes B = subtensor\text{-}combine\ (ds\ @\ dims\ B)\ (map\ (\lambda A.\ A \otimes B)\ As)$
⟨*proof*⟩

lemma *subtensor-tensor-prod*:
assumes $dims\ A \neq []$ **and** $i < hd\ (dims\ A)$
shows $subtensor\ (A \otimes B)\ i = subtensor\ A\ i \otimes B$
⟨*proof*⟩

lemma *lookup-tensor-prod*[*simp*]:
assumes *is1-valid*:$is1 \lhd dims\ A$ **and** *is2-valid*:$is2 \lhd dims\ B$
shows $lookup\ (A \otimes B)\ (is1\ @\ is2) = lookup\ A\ is1 * lookup\ B\ is2$
⟨*proof*⟩

lemma *valid-index-split*:
assumes $is \lhd ds1\ @\ ds2$
obtains *is1 is2* **where** $is1\ @\ is2 = is\ is1 \lhd ds1\ is2 \lhd ds2$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *tensor-prod-distr-left*:

assumes *dims A = dims B*
shows $(A + B) \otimes C = (A \otimes C) + (B \otimes C)$
⟨*proof*⟩

lemma *tensor-prod-distr-right*:
assumes *dims A = dims B*
shows $C \otimes (A + B) = (C \otimes A) + (C \otimes B)$
⟨*proof*⟩

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*)
    ⟨*proof*⟩

  instance ⟨*proof*⟩
end

lemma *order-tensor-one*: *order 1 = 0* ⟨*proof*⟩

lemma *smult-prod-extract1*:
fixes *a*::'*a*::*comm-ring-1*
shows $a \cdot (A \otimes B) = (a \cdot A) \otimes B$
⟨*proof*⟩

lemma *smult-prod-extract2*:
fixes *a*::'*a*::*comm-ring-1*
shows $a \cdot (A \otimes B) = A \otimes (a \cdot B)$
⟨*proof*⟩

lemma *order-0-multiple-of-one*:
assumes *order A = 0*
obtains *a* **where** $A = a \cdot 1$
⟨*proof*⟩

lemma *smult-1*:
fixes *A*::'*a*::*ring-1 tensor*
shows $A = 1 \cdot A$ ⟨*proof*⟩

lemma *tensor0-prod-right*[*simp*]: $A \otimes$ *tensor0 ds = tensor0* (*dims A* @ *ds*)
⟨*proof*⟩

lemma *tensor0-prod-left*[*simp*]: *tensor0 ds* $\otimes A =$ *tensor0* (*ds* @ *dims A*)
⟨*proof*⟩

lemma *subtensor-prod-with-vec*:

assumes *order A = 1 i < hd (dims A)*
shows *subtensor (A ⊗ B) i = lookup A [i] · B*
⟨*proof*⟩

end

6 Unit Vectors as Tensors

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

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

lemma *dims-unit-vec*: *dims (unit-vec n i) = [n]* ⟨*proof*⟩

lemma *lookup-unit-vec*:
assumes *j<n*
shows *lookup (unit-vec n i) [j] = (if i=j then 1 else 0)*
⟨*proof*⟩

lemma *subtensor-prod-with-unit-vec*:
fixes *A*::*'a*::*ring-1 tensor*
assumes *j<n*
shows *subtensor (unit-vec n i ⊗ A) j = (if i=j then A else (tensor0 (dims A)))*
⟨*proof*⟩

lemma *subtensor-decomposition*:
assumes *dims A ≠ []*
shows *listsum (dims A) (map (λi. unit-vec (hd (dims A)) i ⊗ subtensor A i) [0..<hd (dims A)]) = A* (**is** *?LS = A*)
⟨*proof*⟩

end

7 Tensor CP-Rank

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

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

lemma *cprank-max1-order0*: *cprank-max1 B ⟹ order A = 0 ⟹ cprank-max1 (A ⊗ B)*

⟨*proof*⟩

lemma *cprank-max1-order-le1*: *order A* ≤ *0* ⟹ *cprank-max1 B* ⟹ *cprank-max1* (*A* ⊗ *B*)
⟨*proof*⟩

lemma *cprank-max1-prod*: *cprank-max1 A* ⟹ *cprank-max1 B* ⟹ *cprank-max1* (*A* ⊗ *B*)
⟨*proof*⟩

lemma *cprank-max1-prod-list*:
assumes ⋀*B*. *B*∈*set Bs* ⟹ *cprank-max1 B*
shows *cprank-max1* (*prod-list Bs*)
⟨*proof*⟩

lemma *cprank-max1-prod-listE*:
fixes *A*::'*a*::*comm-ring-1 tensor*
assumes *cprank-max1 A*
obtains *Bs a* **where** ⋀*B*. *B*∈*set Bs* ⟹ *order B* = *1 a* · *prod-list Bs* = *A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cprank-max-plus*: *cprank-max i A* ⟹ *cprank-max j B* ⟹ *dims A* = *dims B* ⟹ *cprank-max* (*i*+*j*) (*A*+*B*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cprank-maxI*:
assumes ⋀*B*. *B*∈*set BS* ⟹ *cprank-max1 B*
and ⋀*B*. *B*∈*set BS* ⟹ *dims B* = *ds*
shows *cprank-max* (*length BS*) (*listsum ds BS*)

⟨*proof*⟩

lemma *cprank-max-0E*: *cprank-max 0 A* ⟹ *A* = *tensor0* (*dims A*) ⟨*proof*⟩

lemma *listsum-prod-distr-right*:
assumes (⋀*C*. *C*∈*set CS* ⟹ *dims C* = *ds*)
shows *A* ⊗ *listsum ds CS* = *listsum* (*dims A* @ *ds*) (*map* (λ*C*. *A* ⊗ *C*) *CS*)
⟨*proof*⟩

lemma *cprank-max-prod-order1*:
assumes *order A* = *1*
and *cprank-max n B*
shows *cprank-max n* (*A*⊗*B*)
⟨*proof*⟩

lemma *cprank-max-upper-bound*:
shows *cprank-max* (*prod-list* (*dims A*)) *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cprank-max-cprank*: *cprank-max* (*cprank A*) *A*
⟨*proof*⟩

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*⟩

lemma (**in** *vectorspace*) *dim0I*:
assumes *gen-set* {**0**$_V$}
shows *dim* = *0*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-le1I*:

assumes *gen-set* $\{v\}$
assumes $v \in$ *carrier* V
shows *dim* $\leq$ *1*
⟨*proof*⟩

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* $\subseteq$ *carrier V*
shows *vectorspace.fin-dim class-ring* (*vs* (*span A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *fin-dim-span-cols*:
assumes $A \in$ *carrier-mat n nc*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*cols A*))))
⟨*proof*⟩

end

10 Missing Lemmas of List

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

lemma *nth-map-zip*:
assumes $i <$ *length xs*
assumes $i <$ *length ys*
shows *map f* (*zip xs ys*) ! $i =$ *f* (*xs* ! i, *ys* ! i)
⟨*proof*⟩

lemma *nth-map-zip2*:
assumes $i <$ *length* (*map f* (*zip xs ys*))
shows *map f* (*zip xs ys*) ! $i =$ *f* (*xs* ! i, *ys* ! i)
⟨*proof*⟩

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

lemma *find-first-le*:
assumes $a \in set\ xs$
shows *find-first a xs* $<$ *length xs*
⟨*proof*⟩

lemma *nth-find-first*:
assumes $a \in set\ xs$
shows $xs\ !\ (find\text{-}first\ a\ xs) = a$
⟨*proof*⟩

lemma *find-first-unique*:
assumes *distinct xs*
and $i <$ *length xs*
shows *find-first* $(xs\ !\ i)\ xs = i$
⟨*proof*⟩

end

11 Matrix Rank

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

lemma (**in** *vectorspace*) *full-dim-span*:
assumes $S \subseteq carrier\ V$
and *finite S*
and *vectorspace.dim K* $(span\text{-}vs\ S) = card\ S$
shows *lin-indpt S*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-span*:
assumes $S \subseteq carrier\ V$
and *finite S*
and *maximal U* $(\lambda T.\ T \subseteq S \wedge lin\text{-}indpt\ T)$
shows *vectorspace.dim K* $(span\text{-}vs\ S) = card\ U$
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *maximal-exists-superset*:
  assumes *finite S*
  assumes *maxc*: $\bigwedge A.\ P\ A \Longrightarrow A \subseteq S$ **and** $P\ B$
  shows $\exists A.$ *finite* $A \wedge$ *maximal* $A\ P \wedge B \subseteq A$
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-ge-card-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes $U \subseteq$ *set* (*cols A*)
assumes *lin-indpt U*
shows *rank* $A \geq$ *card U*
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-indpt-full-rank*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
assumes *lin-indpt* (*set* (*cols A*))
shows *rank* $A = nc$
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-le-nc*:
assumes $A \in$ *carrier-mat n nc*
shows *rank* $A \leq nc$
⟨*proof*⟩

lemma (**in** *vec-space*) *full-rank-lin-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes *rank* $A = nc$
assumes *distinct* (*cols A*)
shows *lin-indpt* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *mat-mult-eq-lincomb*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
shows $A *_v$ (*vec nc* ($\lambda i.$ *a* (*col A i*))) = *lincomb a* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *lincomb-eq-mat-mult*:
assumes $A \in$ *carrier-mat n nc*
assumes $v \in$ *carrier-vec nc*
assumes *distinct* (*cols A*)
shows *lincomb* ($\lambda a.$ *v* \$ *find-first a* (*cols A*)) (*set* (*cols A*)) = ($A *_v$ *v*)
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-depI*:

assumes $A \in$ *carrier-mat n nc*
assumes $v \in$ *carrier-vec nc* $v \neq 0_v$ *nc* $A *_v v = 0_v$ *n*
assumes *distinct* (*cols A*)
shows *lin-dep* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-depE*:
assumes $A \in$ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols A*)
obtains v **where** $v \in$ *carrier-vec nc* $v \neq 0_v$ *nc* $A *_v v = 0_v$ *n*
⟨*proof*⟩

lemma (**in** *vec-space*) *non-distinct-low-rank*:
assumes $A \in$ *carrier-mat n n*
and ¬ *distinct* (*cols A*)
shows *rank A* < *n*
⟨*proof*⟩

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

lemma (**in** *vec-space*) *det-zero-low-rank*:
assumes $A \in$ *carrier-mat n n*
and *det A* = *0*
shows *rank A* < *n*
⟨*proof*⟩

lemma *det-identical-cols*:
assumes *A*: $A \in$ *carrier-mat n n*
and *ij*: $i \neq j$
and *i*: $i <$ *n* **and** *j*: $j <$ *n*
and *r*: *col A i* = *col A j*
shows *det A* = *0*
⟨*proof*⟩

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

lemma (**in** *vec-space*) *det-rank-iff*:
assumes $A \in$ *carrier-mat n n*
shows *det A* ≠ *0* ⟷ *rank A* = *n*
⟨*proof*⟩

12 Subadditivity of rank

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

lemma (**in** *module*) *lincomb-add*:
assumes *finite* (*b1* ∪ *b2*)
assumes *b1* ∪ *b2* ⊆ *carrier M*
assumes *x1* = *lincomb a1 b1 a1*∈ (*b1*→*carrier R*)
assumes *x2* = *lincomb a2 b2 a2*∈ (*b2*→*carrier R*)
assumes *x* = *x1* $\oplus_M$ *x2*
shows *lincomb* (λ*v*. (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) *v* ⊕ (λ*v*. *if v* ∈ *b2 then a2 v else* **0**) *v*) (*b1* ∪ *b2*) = *x*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-subadditive*:
assumes *subspace K W1 V*
and *vectorspace.fin-dim K* (*vs W1*)
assumes *subspace K W2 V*
and *vectorspace.fin-dim K* (*vs W2*)
shows *vectorspace.dim K* (*vs* (*subspace-sum W1 W2*)) ≤ *vectorspace.dim K* (*vs W1*) + *vectorspace.dim K* (*vs W2*)
⟨*proof*⟩

lemma (**in** *module*) *nested-submodules*:
assumes *submodule R W M*
assumes *submodule R X M*
assumes *X* ⊆ *W*
shows *submodule R X* (*md W*)
⟨*proof*⟩

lemma (**in** *vectorspace*) *nested-subspaces*:
assumes *subspace K W V*
assumes *subspace K X V*
assumes *X* ⊆ *W*
shows *subspace K X* (*vs W*)
⟨*proof*⟩

lemma (**in** *vectorspace*) *subspace-dim*:
assumes *subspace K X V fin-dim vectorspace.fin-dim K* (*vs X*)
shows *vectorspace.dim K* (*vs X*) ≤ *dim*
⟨*proof*⟩

lemma (**in** *vectorspace*) *fin-dim-subspace-sum*:
assumes *subspace K W1 V*
assumes *subspace K W2 V*
assumes *vectorspace.fin-dim K* (*vs W1*) *vectorspace.fin-dim K* (*vs W2*)
shows *vectorspace.fin-dim K* (*vs* (*subspace-sum W1 W2*))
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-subadditive*:
assumes $A \in$ *carrier-mat n nc*
assumes $B \in$ *carrier-mat n nc*
shows *rank* $(A + B) \leq$ *rank A* + *rank B*
⟨*proof*⟩

lemma (**in** *vec-space*) *span-zero*: *span* {*zero V*} = {*zero V*}
⟨*proof*⟩

lemma (**in** *vec-space*) *dim-zero-vs*: *vectorspace.dim class-ring* (*span-vs* {}) = *0*
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-0I*: *rank* $(0_m\ n\ nc) = 0$
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-le-1-product-entries*:
fixes *f g*::*nat* ⇒ *'a*
assumes $A \in$ *carrier-mat n nc*
assumes $\bigwedge r\ c.\ r<$*dim-row A* $\Longrightarrow$ $c<$*dim-col A* $\Longrightarrow$ *A* $$ $(r,c) = f\ r * g\ c$
shows *rank A* ≤ *1*
⟨*proof*⟩

end

13 Missing Lemmas of Sublist

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

lemma *nths-only-one*:
assumes $\{i.\ i < length\ xs \wedge i \in I\} = \{j\}$
shows *nths xs I* = [*xs*!*j*]
⟨*proof*⟩

lemma *nths-replicate*:
nths (*replicate n x*) *A* = (*replicate* (*card* $\{i.\ i < n \wedge i \in A\}$) *x*)
⟨*proof*⟩

lemma *length-nths-even*:
assumes *even* (*length xs*)
shows *length* (*nths xs* (*Collect even*)) = *length* (*nths xs* (*Collect odd*))
⟨*proof*⟩

lemma *nths-map*:
nths (*map f xs*) *A* = *map f* (*nths xs A*)
⟨*proof*⟩

14 Pick

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

lemma *pick-in-set-inf*:
assumes *infinite S*
shows *pick S n* ∈ *S*
⟨*proof*⟩

lemma *pick-mono-inf*:
assumes *infinite S*
shows *m* < *n* ⟹ *pick S m* < *pick S n*
⟨*proof*⟩

lemma *pick-eq-iff-inf*:
assumes *infinite S*
shows *x* = *y* ⟷ *pick S x* = *pick S y*
⟨*proof*⟩

lemma *card-le-pick-inf*:
assumes *infinite S*
and *pick S n* ≥ *i*
shows *card* {*a*∈*S*. *a* < *i*} ≤ *n*
⟨*proof*⟩

lemma *card-pick-inf*:
assumes *infinite S*
shows *card* {*a*∈*S*. *a* < *pick S n*} = *n*
⟨*proof*⟩

lemma
assumes *n* < *card S*
shows
pick-in-set-le:*pick S n* ∈ *S* **and**
card-pick-le: *card* {*a*∈*S*. *a* < *pick S n*} = *n* **and**
pick-mono-le: *m* < *n* ⟹ *pick S m* < *pick S n*
⟨*proof*⟩

lemma *card-le-pick-le*:
assumes *n* < *card S*
and *pick S n* ≥ *i*
shows *card* {*a*∈*S*. *a* < *i*} ≤ *n*
⟨*proof*⟩

lemma
assumes *n* < *card S* ∨ *infinite S*
shows

pick-in-set:*pick S n* $\in$ *S* **and**
card-le-pick: $i \leq$ *pick S n* ==> *card* $\{a{\in}S.\ a < i\} \leq n$ **and**
card-pick: *card* $\{a{\in}S.\ a <$ *pick S n*$\} = n$ **and**
pick-mono: $m < n \Longrightarrow$ *pick S m* < *pick S n*
⟨*proof*⟩

lemma *pick-card*:
pick I (*card* $\{a{\in}I.\ a < i\}$) = (*LEAST a*. $a{\in}I \wedge a \geq i$)
⟨*proof*⟩

lemma *pick-card-in-set*: $i{\in}I \Longrightarrow$ *pick I* (*card* $\{a{\in}I.\ a < i\}$) = *i*
⟨*proof*⟩

15 Sublist

lemma *nth-nths-card*:
assumes *j*<*length xs*
and $j{\in}J$
shows *nths xs J* ! *card* $\{j0.\ j0 < j \wedge j0 \in J\}$ = *xs*!*j*
⟨*proof*⟩

lemma *pick-reduce-set*:
assumes *i*<*card* $\{a.\ a{<}m \wedge a{\in}I\}$
shows *pick I i* = *pick* $\{a.\ a < m \wedge a \in I\}$ *i*
⟨*proof*⟩

lemma *nth-nths*:
assumes *i*<*card* $\{i.\ i{<}$*length xs* $\wedge\ i{\in}I\}$
shows *nths xs I* ! *i* = *xs* ! *pick I i*
⟨*proof*⟩

lemma *pick-UNIV*: *pick UNIV j* = *j*
⟨*proof*⟩

lemma *pick-le*:
assumes *n* < *card* $\{a.\ a < i \wedge a \in S\}$
shows *pick S n* < *i*
⟨*proof*⟩

lemma *prod-list-complementary-nthss*:
fixes *f* ::$'a \Rightarrow\ 'b$::*comm-monoid-mult*
shows *prod-list* (*map f xs*) = *prod-list* (*map f* (*nths xs A*)) $*$ *prod-list* (*map f* (*nths xs* (−*A*)))
⟨*proof*⟩

lemma *nths-zip*: *nths* (*zip xs ys*) *I* = *zip* (*nths xs I*) (*nths ys I*)
⟨*proof*⟩

16 weave

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

lemma *length-weave*:
shows *length* (*weave A xs ys*) = *length xs* + *length ys*
⟨*proof*⟩

lemma *nth-weave*:
assumes *i* < *length* (*weave A xs ys*)
shows *weave A xs ys* ! *i* = (*if i*∈*A then xs*!(*card* {*a*∈*A*. *a*<*i*}) *else ys*!(*card* {*a*∈−*A*. *a*<*i*}))
⟨*proof*⟩

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*⟩

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

lemma *nths-nth*:
assumes *n*∈*A n*<*length xs*
shows *nths xs A* ! (*card* {*i*. *i*<*n* ∧ *i*∈*A*}) = *xs* ! *n*
⟨*proof*⟩

lemma *list-all2-nths*:
assumes *list-all2 P* (*nths xs A*) (*nths ys A*)
and *list-all2 P* (*nths xs* (−*A*)) (*nths ys* (−*A*))
shows *list-all2 P xs ys*
⟨*proof*⟩

lemma *nths-weave*:
assumes *length xs* = *card* {*a*∈*A*. *a* < *length xs* + *length ys*}
assumes *length ys* = *card* {*a*∈(−*A*). *a* < *length xs* + *length ys*}
shows *nths* (*weave A xs ys*) *A* = *xs* ∧ *nths* (*weave A xs ys*) (−*A*) = *ys*
⟨*proof*⟩

lemma *set-weave*:
assumes *length xs* = *card* {*a*∈*A*. *a* < *length xs* + *length ys*}
assumes *length ys* = *card* {*a*∈−*A*. *a* < *length xs* + *length ys*}
shows *set* (*weave A xs ys*) = *set xs* ∪ *set ys*

⟨*proof*⟩

lemma *weave-complementary-nthss*[*simp*]:
weave A (*nths xs A*) (*nths xs* (−*A*)) = *xs*
⟨*proof*⟩

lemma *length-nths'*:
length (*nths xs I*) = *card* {*i*∈*I*. *i* < *length xs*}
⟨*proof*⟩

end

17 Tensor Matricization

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

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

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

lemma *digit-encode-decode*[*simp*]:
assumes *is* ◁ *ds*
shows *digit-encode ds* (*digit-decode ds is*) = *is*
⟨*proof*⟩

lemma *digit-decode-encode*[*simp*]:
shows *digit-decode ds* (*digit-encode ds a*) = *a mod* (*prod-list ds*)
⟨*proof*⟩

lemma *digit-decode-encode-lt*[*simp*]:
assumes *a* < *prod-list ds*
shows *digit-decode ds* (*digit-encode ds a*) = *a*
⟨*proof*⟩

lemma *digit-decode-lt*:
assumes *is* ◁ *ds*
shows *digit-decode ds is* < *prod-list ds*
⟨*proof*⟩

lemma *digit-encode-valid-index*:
assumes $a < prod\text{-}list\ ds$
shows $digit\text{-}encode\ ds\ a \lhd ds$
⟨*proof*⟩

lemma *length-digit-encode*:
shows $length\ (digit\text{-}encode\ ds\ a) = length\ ds$
⟨*proof*⟩

lemma *digit-encode-0*:
$prod\text{-}list\ ds\ dvd\ a \Longrightarrow digit\text{-}encode\ ds\ a = replicate\ (length\ ds)\ 0$
⟨*proof*⟩

lemma *valid-index-weave*:
assumes $is1 \lhd (nths\ ds\ A)$
and $is2 \lhd (nths\ ds\ (-A))$
shows $weave\ A\ is1\ is2 \lhd ds$
and $nths\ (weave\ A\ is1\ is2)\ A = is1$
and $nths\ (weave\ A\ is1\ is2)\ (-A) = is2$
⟨*proof*⟩

definition *matricize* :: $nat\ set \Rightarrow 'a\ tensor \Rightarrow 'a\ mat$ **where**
$matricize\ rmodes\ T = mat$
$(prod\text{-}list\ (nths\ (Tensor.dims\ T)\ rmodes))$
$(prod\text{-}list\ (nths\ (Tensor.dims\ T)\ (-rmodes)))$
$(\lambda(r,\ c).\ Tensor.lookup\ T\ (weave\ rmodes$
$(digit\text{-}encode\ (nths\ (Tensor.dims\ T)\ rmodes)\ r)$
$(digit\text{-}encode\ (nths\ (Tensor.dims\ T)\ (-rmodes))\ c)$
))

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

lemma *dims-matricize*:
$dim\text{-}row\ (matricize\ rmodes\ T) = prod\text{-}list\ (nths\ (Tensor.dims\ T)\ rmodes)$
$dim\text{-}col\ (matricize\ rmodes\ T) = prod\text{-}list\ (nths\ (Tensor.dims\ T)\ (-rmodes))$
⟨*proof*⟩

lemma *dims-dematricize*: $Tensor.dims\ (dematricize\ rmodes\ A\ ds) = ds$
⟨*proof*⟩

lemma *valid-index-nths*:
assumes $is \lhd ds$
shows $nths\ is\ A \lhd nths\ ds\ A$

⟨*proof*⟩

lemma *dematricize-matricize*:
shows *dematricize rmodes* (*matricize rmodes T*) (*Tensor.dims T*) = *T*
⟨*proof*⟩

lemma *matricize-dematricize*:
assumes *dim-row A* = *prod-list* (*nths ds rmodes*)
and *dim-col A* = *prod-list* (*nths ds* (−*rmodes*))
shows *matricize rmodes* (*dematricize rmodes A ds*) = *A*
⟨*proof*⟩

lemma *matricize-add*:
assumes *dims A* = *dims B*
shows *matricize I A* + *matricize I B* = *matricize I* (*A*+*B*)
⟨*proof*⟩

lemma *matricize-0*:
shows *matricize I* (*tensor0 ds*) = 0_m (*dim-row* (*matricize I* (*tensor0 ds*))) (*dim-col* (*matricize I* (*tensor0 ds*)))
⟨*proof*⟩

end

18 Submatrices

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

19 Submatrix

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

lemma *dim-submatrix*: *dim-row* (*submatrix A I J*) = *card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
dim-col (*submatrix A I J*) = *card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
⟨*proof*⟩

lemma *submatrix-index*:
assumes *i*<*card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
assumes *j*<*card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
shows *submatrix A I J* $$ (*i*,*j*) = *A* $$ (*pick I i*, *pick J j*)
⟨*proof*⟩

lemma *set-le-in*:{*a*. *a* < *n* ∧ *a* ∈ *I*} = {*a* ∈ *I*. *a* < *n*} ⟨*proof*⟩

lemma *submatrix-index-card*:
assumes $i<$*dim-row* A $j<$*dim-col* 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*⟩

lemma *submatrix-split*: *submatrix* A I J $=$ *submatrix* (*submatrix* A *UNIV* J) I *UNIV*
⟨*proof*⟩

end

20 CP-Rank and Matrix Rank

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

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

no-notation *normal-rel* (**infixl** $\lhd$ *60*)

lemma *lookup-order1-prod*:
assumes $\bigwedge B.\ B\in$*set* $Bs \Longrightarrow$ *Tensor.order* $B = 1$
assumes $is \lhd$ *dims* (*prod-list* Bs)
shows *lookup* (*prod-list* Bs) $is =$ *prod-list* (*map* ($\lambda(i,B).$ *lookup* B $[i]$) (*zip* is Bs))
⟨*proof*⟩

lemma *matricize-cprank-max1*:
fixes A::$'a$::*field tensor*
assumes *cprank-max1* A
shows *mrank* (*matricize* I A) $\leq$ *1*
⟨*proof*⟩

lemma *matrix-rank-le-cprank-max*:
fixes A :: ($'a$::*field*) *tensor*
assumes *cprank-max* r A
shows *mrank* (*matricize* I A) $\leq r$
⟨*proof*⟩

lemma *matrix-rank-le-cp-rank*:
fixes A :: ($'a$::*field*) *tensor*
shows *mrank* (*matricize* I A) $\leq$ *cprank* A
⟨*proof*⟩

end

21 Missing Lemmas of Matrix

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

lemma *dim-vec-of-list*[*simp*] :*dim-vec* (*vec-of-list as*) = *length as* ⟨*proof*⟩

lemma *list-vec*: *list-of-vec* (*vec-of-list xs*) = *xs*
⟨*proof*⟩

lemma *vec-list*: *vec-of-list* (*list-of-vec v*) = *v*
⟨*proof*⟩

lemma *index-vec-of-list*: *i*<*length xs* ⟹ (*vec-of-list xs*) $ *i* = *xs* ! *i*
⟨*proof*⟩

lemma *nth-list-of-vec*: *i*<*dim-vec v* ⟹ (*list-of-vec v*) ! *i* = *v* $ *i*
⟨*proof*⟩

lemma *vec-of-list-index*: *vec-of-list xs* $ *j* = *xs* ! *j*
⟨*proof*⟩

lemma *list-of-vec-index*: *list-of-vec v* ! *j* = *v* $ *j*
⟨*proof*⟩

lemma *list-of-vec-map*: *list-of-vec xs* = *map* (*op* $ *xs*) [*0*..<*dim-vec xs*] ⟨*proof*⟩

definition *component-mult v w* = *vec* (*min* (*dim-vec v*) (*dim-vec w*)) (λ*i*. *v* $ *i* ∗ *w* $ *i*)
definition *vec-set*::′*a vec* ⇒ ′*a set* (*set*$_v$)
where *vec-set v* = *vec-index v* ‘ {..<*dim-vec v*}

lemma *index-component-mult*:
assumes *i* < *dim-vec v i* < *dim-vec w*
shows *component-mult v w* $ *i* = *v* $ *i* ∗ *w* $ *i*
⟨*proof*⟩

lemma *dim-component-mult*:
dim-vec (*component-mult v w*) = *min* (*dim-vec v*) (*dim-vec w*)
⟨*proof*⟩

lemma *vec-setE*:
assumes *a* ∈ *set*$_v$ *v*
obtains *i* **where** *v*$*i* = *a i*<*dim-vec v* ⟨*proof*⟩

lemma *vec-setI*:
assumes *v*$*i* = *a i*<*dim-vec v*
shows *a* ∈ *set*$_v$ *v* ⟨*proof*⟩

lemma *set-list-of-vec*:
set (*list-of-vec v*) = set_v *v* ⟨*proof*⟩

lemma *length-list-of-vec*[*simp*] :*length* (*list-of-vec v*) = *dim-vec v* ⟨*proof*⟩

end

22 Matrix to Vector Conversion

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

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

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

lemma *two-digit-le*:
fixes *i j* :: *nat*
assumes *i* < *m j* < *n*
shows *i*∗*n* + *j* < *m*∗*n*
⟨*proof*⟩

lemma *extract-matrix-cong*:
assumes ⋀*i*. *i* < *m* ∗ *n* ⟹ *a i* = *b i*
shows *extract-matrix a m n* = *extract-matrix b m n*
⟨*proof*⟩

lemma *extract-matrix-flatten-matrix*:
extract-matrix (*flatten-matrix A*) (*dim-row A*) (*dim-col A*) = *A*
⟨*proof*⟩

lemma *flatten-matrix-extract-matrix*:
shows ⋀*k*. *k*<*m*∗*n* ⟹ *flatten-matrix* (*extract-matrix a m n*) *k* = *a k*
⟨*proof*⟩

lemma *index-extract-matrix*:
assumes *i*<*m j*<*n*
shows *extract-matrix a m n* $$ (*i*,*j*) = *a* (*i*∗*n* + *j*)
⟨*proof*⟩

lemma *dim-extract-matrix*:
shows *dim-row* (*extract-matrix as m n*) = *m*
and *dim-col* (*extract-matrix as m n*) = *n*
⟨*proof*⟩

end

23 Deep Learning Networks

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

This symbol is used for the Tensor product:

no-notation *Group.monoid.mult* (**infixl** $\otimes_1$ *70*)

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

fun *input-sizes* :: *'a convnet* ⇒ *nat list* **where**
input-sizes (Input M) = [M] |
input-sizes (Conv A m) = input-sizes m |
input-sizes (Pool m1 m2) = input-sizes m1 @ input-sizes m2

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

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

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

fun *insert-weights* :: *(nat × nat) convnet* ⇒ *(nat* ⇒ *real)* ⇒ *real mat convnet* **where**
insert-weights (Input M) w = Input M |
insert-weights (Conv (r0,r1) m) w = Conv
(extract-matrix w r0 r1)
*(insert-weights m (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-row A*, *dim-col A*) (*remove-weights m*)
|
remove-weights (*Pool m1 m2*) = *Pool* (*remove-weights m1*) (*remove-weights m2*)

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

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

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

lemma *insert-weights-cong*:
assumes (⋀*i*. *i*<*count-weights m* ⟹ *w1 i* = *w2 i*)
shows *insert-weights m w1* = *insert-weights m w2*
⟨*proof*⟩

lemma *dims-mat-tensorlist-mult*:
assumes *T*∈*set*$_v$ (*mat-tensorlist-mult A Ts ds*)
shows *Tensor.dims T* = *ds*
⟨*proof*⟩

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

lemma *output-size-correct-tensors*:
assumes *valid-net' m*
shows *output-size' m* = *dim-vec* (*tensors-from-net m*)
⟨*proof*⟩

lemma *output-size-correct*:
assumes *valid-net' m*
and *map dim-vec inputs* = *input-sizes m*
shows *output-size' m* = *dim-vec* (*evaluate-net m inputs*)
⟨*proof*⟩

lemma *input-sizes-remove-weights*: *input-sizes m* = *input-sizes* (*remove-weights m*)
⟨*proof*⟩

lemma *dims-tensors-from-net*:
assumes $T \in set_v$ (*tensors-from-net m*)
shows *Tensor.dims T* = *input-sizes m*
⟨*proof*⟩

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

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

lemma *base-input-length*:
assumes *is* ◁ *input-sizes m*
shows *input-sizes m* = *map dim-vec* (*base-input m is*)
⟨*proof*⟩

lemma *nth-mat-tensorlist-mult*:
assumes ⋀*A*. *A*∈set_v *Ts* ⟹ *dims A* = *ds*
assumes *i* < *dim-row A*
assumes *dim-vec Ts* = *dim-col A*
shows *mat-tensorlist-mult A Ts ds* $ *i* = *listsum ds* (*map* (λ*j*. (*A* $$ (*i*,*j*)) · *Ts* $ *j*) [*0*..<*dim-vec Ts*])
(**is** - = *listsum ds ?Ts'*)
⟨*proof*⟩

lemma *lookup-tensors-from-net*:
assumes *valid-net' m*
and *is* ◁ *input-sizes m*
and *j* < *output-size' m*
shows *Tensor.lookup* (*tensors-from-net m* $ *j*) *is* = *evaluate-net m* (*base-input m is*) $ *j*
⟨*proof*⟩

lemma *insert-remove-weights*:
obtains *w* **where** *m* = *insert-weights* (*remove-weights m*) *w*
⟨*proof*⟩

lemma *remove-insert-weights*:
shows *remove-weights* (*insert-weights m w*) = *m*
⟨*proof*⟩

lemma *finite-valid-index*: *finite* {*is*. *is* ◁ *ds*}
⟨*proof*⟩

lemma *setsum-valid-index-split*:
(∑ *is* | *is* ◁ *ds1* @ *ds2*. *f is*) = (∑ *is1* | *is1* ◁ *ds1*. (∑ *is2* | *is2* ◁ *ds2*. *f* (*is1* @

is2)))
⟨*proof*⟩

lemma *prod-lessThan-split*:
fixes *g* :: *nat* ⇒ *real* **shows** *prod g* {..<*n*+*m*} = *prod g* {..<*n*} ∗ *prod* (λ*x*. *g* (*x*+*n*)) {..<*m*}
⟨*proof*⟩

lemma *evaluate-net-from-tensors*:
assumes *valid-net*′ *m*
and *map dim-vec inputs* = *input-sizes m*
and *j* < *output-size*′ *m*
shows *evaluate-net m inputs* $ *j*
= (∑ *is*∈{*is*. *is* ◁ *input-sizes m*}. (∏ *k*<*length inputs*. *inputs* ! *k* $ (*is*!*k*)) ∗ *Tensor.lookup* (*tensors-from-net m* $ *j*) *is*)
⟨*proof*⟩

lemma *tensors-from-net-eqI*:
assumes *valid-net*′ *m1 valid-net*′ *m2 input-sizes m1* = *input-sizes m2*
assumes ⋀*inputs*. *input-sizes m1* = *map dim-vec inputs* ⟹ *evaluate-net m1 inputs* = *evaluate-net m2 inputs*
shows *tensors-from-net m1* = *tensors-from-net m2*
⟨*proof*⟩

end

24 Concrete Matrices

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

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

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

lemma *eye-matrix-dim*: *dim-row* (*eye-matrix nr nc*) = *nr dim-col* (*eye-matrix nr nc*) = *nc* ⟨*proof*⟩

lemma *row-eye-matrix*:
assumes *i* < *nr*
shows *row* (*eye-matrix nr nc*) *i* = *unit-vec nc i*
⟨*proof*⟩

lemma *unit-eq-0*[*simp*]:
assumes *i*: *i* ≥ *n*
shows *unit-vec n i* = 0$_v$ *n*

⟨*proof*⟩

lemma *mult-eye-matrix*:
assumes $i < nr$
shows (*eye-matrix nr* (*dim-vec v*) $*_v$ *v*) $ *i* = (*if i<dim-vec v then v* $ *i else 0*)
(**is** *?a* $ *i* = *?b*)
⟨*proof*⟩

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

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

lemma *all1-matrix-dim*: *dim-row* (*all1-matrix nr nc*) = *nr dim-col* (*all1-matrix nr nc*) = *nc*
⟨*proof*⟩

lemma *row-all1-matrix*:
assumes $i < nr$
shows *row* (*all1-matrix nr nc*) *i* = *all1-vec nc*
⟨*proof*⟩

lemma *all1-vec-scalar-prod*:
shows *all1-vec* (*length xs*) · (*vec-of-list xs*) = *sum-list xs*
⟨*proof*⟩

lemma *mult-all1-matrix*:
assumes $i < nr$
shows ((*all1-matrix nr* (*dim-vec v*)) $*_v$ *v*) $ *i* = *sum-list* (*list-of-vec v*) (**is** *?a* $ *i* = *sum-list* (*list-of-vec v*))
⟨*proof*⟩

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

lemma *copy-first-matrix-dim*: *dim-row* (*copy-first-matrix nr nc*) = *nr dim-col* (*copy-first-matrix nr nc*) = *nc*
⟨*proof*⟩

lemma *row-copy-first-matrix*:
assumes $i < nr$
shows *row* (*copy-first-matrix nr nc*) *i* = *unit-vec nc 0*
⟨*proof*⟩

lemma *mult-copy-first-matrix*:

assumes $i < nr$ **and** *dim-vec* $v > 0$
shows (*copy-first-matrix* nr (*dim-vec* v) $*_v$ v) $ $ $i = v $ $ 0$ (**is** *?a* $ $ $i = v $ $ 0$)
⟨*proof*⟩

end

25 Missing Lemmas of Finite_Set

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

lemma *card-even*[*simp*]: *card* $\{a \in$ *Collect even*. $a < 2 * n\} = n$
⟨*proof*⟩

lemma *card-odd*[*simp*]: *card* $\{a \in$ *Collect odd*. $a < 2 * n\} = n$
⟨*proof*⟩

end

26 Deep Network Model

theory *DL-Deep-Model*
imports *DL-Network Tensor-Matricization DL-Submatrix DL-Concrete-Matrices DL-Missing-Finite-Set DL-Missing-Sublist Jordan-Normal-Form.Determinant*
begin

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

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

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

lemma *valid-deep-model*: *valid-net* (*deep-model Y r rs*)
⟨*proof*⟩

lemma *valid-deep-model′*: *valid-net* (*deep-model′ r rs*)
⟨*proof*⟩

lemma *input-sizes-deep-model′*:
assumes *length rs* ≥ *1*
shows *input-sizes* (*deep-model′-l rs*) = *replicate* (*2^*(*length rs* − *1*)) (*last rs*)
⟨*proof*⟩

lemma *input-sizes-deep-model*:
assumes *length rs* $\geq$ *2*
shows *input-sizes* (*deep-model-l rs*) = *replicate* (*2^*(*length rs* − *2*)) (*last rs*)
⟨*proof*⟩

lemma *evaluate-net-Conv-id*:
assumes *valid-net' m*
and *input-sizes m* = *map dim-vec input*
and *j*<*nr*
shows *evaluate-net* (*Conv* (*eye-matrix nr* (*output-size' m*)) *m*) *input* $ *j*
= (*if j*<*output-size' m then evaluate-net m input* $ *j else 0*)
⟨*proof*⟩

lemma *tensors-from-net-Conv-id*:
assumes *valid-net' m*
and *i*<*nr*
shows *tensors-from-net* (*Conv* (*eye-matrix nr* (*output-size' m*)) *m*) $ *i*
= (*if i*<*output-size' m then tensors-from-net m* $ *i else tensor0* (*input-sizes m*))
(**is** *?a* $ *i* = *?b*)
⟨*proof*⟩

lemma *evaluate-net-Conv-copy-first*:
assumes *valid-net' m*
and *input-sizes m* = *map dim-vec input*
and *j*<*nr*
and *output-size' m* > *0*
shows *evaluate-net* (*Conv* (*copy-first-matrix nr* (*output-size' m*)) *m*) *input* $ *j*
= *evaluate-net m input* $ *0*
⟨*proof*⟩

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*⟩

lemma *evaluate-net-Conv-all1*:
assumes *valid-net' m*
and *input-sizes m* = *map dim-vec input*
and *i*<*nr*
shows *evaluate-net* (*Conv* (*all1-matrix nr* (*output-size' m*)) *m*) *input* $ *i*
= *Groups-List.sum-list* (*list-of-vec* (*evaluate-net m input*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *witness'-is-deep-model*: *remove-weights* (*witness'* Y rs) $=$ *deep-model'* Y rs
⟨*proof*⟩

lemma *witness-valid*: *valid-net'* (*witness* Y r rs)
⟨*proof*⟩

lemma *witness'-valid*: *valid-net'* (*witness'* Y rs)
⟨*proof*⟩

lemma *witness-l0'*: *witness'* Y [M] $=$
(*Pool*
(*Conv* (*eye-matrix* Y M) (*Input* M))
(*Conv* (*eye-matrix* Y M) (*Input* M))
)
⟨*proof*⟩

lemma *witness-l1*: *witness* Y *r0* [M] $=$
Conv (*all1-matrix* Y *r0*) (*witness'* *r0* [M])
⟨*proof*⟩

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])
⟨*proof*⟩

lemma *tensor-prod-unit-vec*:
unit-vec M j $\otimes$ *unit-vec* M j $=$ *tensor-from-lookup* [M,M] (λ*is*. *if is*=[j,j] *then 1 else 0*) (**is** *?A=?B*)

⟨*proof*⟩

lemma *tensors-ht-l0'*:
assumes *j*<*r0*
shows *tensors-from-net* (*witness' r0* [*M*]) $ *j*
= (*if j*<*M then unit-vec M j* ⊗ *unit-vec M j else tensor0* [*M*,*M*]) (**is** - = *?b*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *lookup-tensors-ht-l1*:
assumes *j* < *r1*
and *is* ◁ [*M*,*M*]
shows *Tensor.lookup* (*tensors-from-net* (*witness r1 r0* [*M*]) $ *j*) *is*
= (*if is*!*0* = *is*!*1* ∧ *is*!*0*<*r0 then 1 else 0*)
⟨*proof*⟩

lemma *length-output-deep-model*:
assumes *remove-weights m* = *deep-model-l rs*
shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-deep-model'*:
assumes *remove-weights m* = *deep-model'-l rs*
shows *dim-vec* (*tensors-from-net m*) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-witness*:
dim-vec (*tensors-from-net* (*witness-l rs*)) = *rs* ! *0*
⟨*proof*⟩

lemma *length-output-witness'*:
dim-vec (*tensors-from-net* (*witness'-l rs*)) = *rs* ! *0*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *dims-output-witness*:
assumes *length rs* $\geq$ *2*
and $\bigwedge$*r*. *r*$\in$*set rs* $\Longrightarrow$ *r* > *0*
and *j* < *rs*!*0*
shows *Tensor.dims* (*tensors-from-net* (*witness-l rs*) $ *j*) = *replicate* (*2*^(*length rs* − *2*)) (*last rs*)
⟨*proof*⟩

lemma *dims-output-deep-model′*:
assumes *length rs* $\geq$ *1*
and $\bigwedge$*r*. *r*$\in$*set rs* $\Longrightarrow$ *r* > *0*
and *j* < *rs*!*0*
and *remove-weights m* = *deep-model′-l rs*
shows *Tensor.dims* (*tensors-from-net m* $ *j*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
⟨*proof*⟩

lemma *dims-output-witness′*:
assumes *length rs* $\geq$ *1*
and $\bigwedge$*r*. *r*$\in$*set rs* $\Longrightarrow$ *r* > *0*
and *j* < *rs*!*0*
shows *Tensor.dims* (*tensors-from-net* (*witness′-l rs*) $ *j*) = *replicate* (*2*^(*length rs* − *1*)) (*last rs*)
⟨*proof*⟩

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* $\geq$ *3*
and *no-zeros*:$\bigwedge$*r*. *r*$\in$*set rs* $\Longrightarrow$ *0* < *r*
begin

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

end

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

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

lemma *dims-tensor-deep-model*:
assumes *remove-weights m = deep-model-l rs*
shows *dims (tensors-from-net m $ y) = replicate (2 * N-half) (last rs)*
⟨*proof*⟩

lemma *order-tensor-deep-model*:
assumes *remove-weights m = deep-model-l rs*
shows *order (tensors-from-net m $ y) = 2 * N-half*
⟨*proof*⟩

lemma *dims-A*:
shows *Tensor.dims (A ws) = replicate (2 * N-half) (last rs)*
⟨*proof*⟩

lemma *order-A*:
shows *order (A ws) = 2 * N-half* ⟨*proof*⟩

lemma *dims-A'*:
shows *dim-row (A' ws) = prod-list (nths (Tensor.dims (A ws)) {n. even n})*
and *dim-col (A' ws) = prod-list (nths (Tensor.dims (A ws)) {n. odd n})*
⟨*proof*⟩

lemma *dims-A'-pow*:
shows *dim-row (A' ws) = (last rs) ^ N-half dim-col (A' ws) = (last rs) ^ N-half*
⟨*proof*⟩

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

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

lemma *witness-weights*:*witness-l rs = insert-weights (deep-model-l rs) witness-weights*
⟨*proof*⟩

lemma *Aw-def'*: *Aw = A witness-weights* ⟨*proof*⟩

lemma *Aw'-def'*: *Aw' = A' witness-weights* ⟨*proof*⟩

lemma *dims-Aw*: *Tensor.dims Aw = replicate (2 * N-half) (last rs)*
⟨*proof*⟩

lemma *order-Aw*: *order Aw = 2 * N-half*
⟨*proof*⟩

lemma *dims-Aw'*:
dim-row Aw' = prod-list (nths (Tensor.dims Aw) {n. even n})
dim-col Aw' = prod-list (nths (Tensor.dims Aw) {n. odd n})
⟨*proof*⟩

lemma *dims-Aw'-pow*: *dim-row Aw' = (last rs) ^ N-half dim-col Aw' = (last rs) ^ N-half*
⟨*proof*⟩

lemma *witness-tensor*:
assumes *is ◁ Tensor.dims Aw*
shows *Tensor.lookup Aw is*
= (if nths is {n. even n} = nths is {n. odd n} ∧ (∀ i∈set is. i < last (butlast rs)) then 1 else 0)
⟨*proof*⟩

lemma *witness-matricization*:
assumes *i < dim-row Aw'* **and** *j < dim-col Aw'*
shows *Aw' $$ (i, j)*
= (if i=j ∧ (∀ i0∈set (digit-encode (nths (Tensor.dims Aw) {n. even n}) i). i0 < last (butlast rs)) then 1 else 0)
⟨*proof*⟩

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

lemma *card-low-digits*:
assumes *m>0 ⋀d. d∈set ds ⟹ m ≤ d*
shows *card {i. i<prod-list ds ∧ (∀ i0∈set (digit-encode ds i). i0 < m)} = m ^ (length ds)*
⟨*proof*⟩

lemma *card-rows-with-1*: *card {i∈rows-with-1. i<dim-row Aw'} = r ^ N-half*
⟨*proof*⟩

lemma *infinite-rows-with-1*: *infinite rows-with-1*
⟨*proof*⟩

lemma *witness-submatrix*: *submatrix Aw' rows-with-1 rows-with-1 = 1_m (r^N-half)*
⟨*proof*⟩

lemma *witness-det*: *det (submatrix Aw' rows-with-1 rows-with-1) ≠ 0* ⟨*proof*⟩

end

end

27 Less common functions on lists

theory *PP-More-List2*
imports
Main
begin

definition *strip-while* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
$strip\text{-}while\ P = rev \circ dropWhile\ P \circ rev$

lemma *strip-while-rev* [*simp*]:
$strip\text{-}while\ P\ (rev\ xs) = rev\ (dropWhile\ P\ xs)$
⟨*proof*⟩

lemma *strip-while-Nil* [*simp*]:
$strip\text{-}while\ P\ [] = []$
⟨*proof*⟩

lemma *strip-while-append* [*simp*]:
$\neg\ P\ x \Longrightarrow strip\text{-}while\ P\ (xs\ @\ [x]) = xs\ @\ [x]$
⟨*proof*⟩

lemma *strip-while-append-rec* [*simp*]:
$P\ x \Longrightarrow strip\text{-}while\ P\ (xs\ @\ [x]) = strip\text{-}while\ P\ xs$
⟨*proof*⟩

lemma *strip-while-Cons* [*simp*]:
$\neg\ P\ x \Longrightarrow strip\text{-}while\ P\ (x\ \#\ xs) = x\ \#\ strip\text{-}while\ P\ xs$
⟨*proof*⟩

lemma *strip-while-eq-Nil* [*simp*]:
$strip\text{-}while\ P\ xs = [] \longleftrightarrow (\forall\, x{\in}set\ xs.\ P\ x)$
⟨*proof*⟩

lemma *strip-while-eq-Cons-rec*:
$strip\text{-}while\ P\ (x\ \#\ xs) = x\ \#\ strip\text{-}while\ P\ xs \longleftrightarrow \neg\ (P\ x \wedge (\forall\, x{\in}set\ xs.\ P\ x))$
⟨*proof*⟩

lemma *strip-while-not-last* [*simp*]:
$\neg\ P\ (last\ xs) \Longrightarrow strip\text{-}while\ P\ xs = xs$
⟨*proof*⟩

lemma *split-strip-while-append*:
fixes *xs* :: *'a list*
obtains *ys zs* :: *'a list*
where $strip\text{-}while\ P\ xs = ys$ **and** $\forall\, x{\in}set\ zs.\ P\ x$ **and** $xs = ys\ @\ zs$
⟨*proof*⟩

lemma *strip-while-snoc* [*simp*]:
strip-while P (xs @ [x]) = (if P x then strip-while P xs else xs @ [x])
⟨*proof*⟩

lemma *strip-while-map*:
strip-while P (map f xs) = map f (strip-while (P ∘ f) xs)
⟨*proof*⟩

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

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
⟨*proof*⟩

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading P (x # xs) ⟷ ¬ P x
⟨*proof*⟩

lemma *no-leading-append* [*simp*]:
no-leading P (xs @ ys) ⟷ no-leading P xs ∧ (xs = [] ⟶ no-leading P ys)
⟨*proof*⟩

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (dropWhile P xs)
⟨*proof*⟩

lemma *dropWhile-eq-obtain-leading*:
assumes *dropWhile P xs = ys*
obtains *zs* **where** *xs = zs @ ys* **and** *⋀z. z ∈ set zs ⟹ P z* **and** *no-leading P ys*
⟨*proof*⟩

lemma *dropWhile-idem-iff*:
dropWhile P xs = xs ⟷ no-leading P xs
⟨*proof*⟩

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

lemma *no-trailing-unfold*:
no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))
⟨*proof*⟩

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

no-trailing P []
⟨*proof*⟩

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

lemma *no-trailing-append-Cons* [*simp*]:
no-trailing P (xs @ y # ys) ⟷ no-trailing P (y # ys)
⟨*proof*⟩

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (strip-while P xs)
⟨*proof*⟩

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs = ys*
obtains *zs* **where** *xs = ys @ zs* **and** ⋀*z*. *z ∈ set zs ⟹ P z* **and** *no-trailing P ys*
⟨*proof*⟩

lemma *strip-while-idem-iff*:
strip-while P xs = xs ⟷ no-trailing P xs
⟨*proof*⟩

lemma *no-trailing-map*:
no-trailing P (map f xs) = no-trailing (P ∘ f) xs
⟨*proof*⟩

lemma *no-trailing-upt* [*simp*]:
no-trailing P [n..<m] ⟷ (n < m ⟶ ¬ P (m − 1))
⟨*proof*⟩

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

lemma *nth-default-nth*:
n < length xs ⟹ nth-default dflt xs n = xs ! n
⟨*proof*⟩

lemma *nth-default-beyond*:
length xs ≤ n ⟹ nth-default dflt xs n = dflt
⟨*proof*⟩

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] n = dflt
⟨*proof*⟩

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n′* ⇒ *nth-default dflt xs n′*)
⟨*proof*⟩

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
⟨*proof*⟩

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
⟨*proof*⟩

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = *dflt* ⟷ (*k* < *length xs* ⟶ *xs* ! *k* = *dflt*)
⟨*proof*⟩

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ *dflt* ⟹ (*n*, *x*) ∈ *set* (*enumerate 0 xs*) ⟷ *nth-default dflt xs n* = *x*
⟨*proof*⟩

lemma *last-conv-nth-default*:
assumes *xs* ≠ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
⟨*proof*⟩

lemma *nth-default-map-eq*:
f dflt′ = *dflt* ⟹ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt′ xs n*)
⟨*proof*⟩

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* ≠ *dflt*}

⟨*proof*⟩

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k* ≠ *dflt*} = *map fst* (*filter* (λ(-, *x*). *x* ≠ *dflt*) (*enumerate 0 xs*))
⟨*proof*⟩

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
⟨*proof*⟩

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
⟨*proof*⟩

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
⟨*proof*⟩

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
⟨*proof*⟩

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-eq-iff*:
nth-default dflt xs = *nth-default dflt ys*
⟷ *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

end

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

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

28.1 Preliminary: auxiliary operations for 'almost everywhere zero'

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

lemma *finite-mult-not-eq-zero-leftI*:
  fixes $f :: 'b \Rightarrow 'a :: mult\text{-}zero$
  assumes *finite* $\{a.\ f\ a \neq 0\}$
  shows *finite* $\{a.\ g\ a * f\ a \neq 0\}$
⟨*proof*⟩

lemma *finite-mult-not-eq-zero-rightI*:
  fixes $f :: 'b \Rightarrow 'a :: mult\text{-}zero$
  assumes *finite* $\{a.\ f\ a \neq 0\}$
  shows *finite* $\{a.\ f\ a * g\ a \neq 0\}$
⟨*proof*⟩

lemma *finite-mult-not-eq-zero-prodI*:
  fixes $f\ g :: 'a \Rightarrow 'b :: semiring\text{-}0$
  assumes *finite* $\{a.\ f\ a \neq 0\}$ (**is** *finite ?F*)
  assumes *finite* $\{b.\ g\ b \neq 0\}$ (**is** *finite ?G*)
  shows *finite* $\{(a, b).\ f\ a * g\ b \neq 0\}$
⟨*proof*⟩

lemma *finite-not-eq-zero-sumI*:
  fixes $f\ g :: 'a :: monoid\text{-}add \Rightarrow 'b :: semiring\text{-}0$
  assumes *finite* $\{a.\ f\ a \neq 0\}$ (**is** *finite ?F*)
  assumes *finite* $\{b.\ g\ b \neq 0\}$ (**is** *finite ?G*)
  shows *finite* $\{a + b \mid a\ b.\ f\ a \neq 0 \wedge g\ b \neq 0\}$ (**is** *finite ?FG*)
⟨*proof*⟩

lemma *finite-mult-not-eq-zero-sumI*:
  fixes $f\ g :: 'a :: monoid\text{-}add \Rightarrow 'b :: semiring\text{-}0$
  assumes *finite* $\{a.\ f\ a \neq 0\}$
  assumes *finite* $\{b.\ g\ b \neq 0\}$
  shows *finite* $\{a + b \mid a\ b.\ f\ a * g\ b \neq 0\}$
⟨*proof*⟩

lemma *finite-Sum-any-not-eq-zero-weakenI*:
  assumes *finite* $\{a.\ \exists\, b.\ f\ a\ b \neq 0\}$
  shows *finite* $\{a.\ Sum\text{-}any\ (f\ a) \neq 0\}$
⟨*proof*⟩

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*]:
$P \Longrightarrow (a\ when\ P) = a$
$\neg\ P \Longrightarrow (a\ when\ P) = 0$
⟨*proof*⟩

lemma *when-simps* [*simp*]:
$(a\ when\ True) = a$
$(a\ when\ False) = 0$
⟨*proof*⟩

lemma *when-cong*:
assumes $P \longleftrightarrow Q$
and $Q \Longrightarrow a = b$
shows $(a\ when\ P) = (b\ when\ Q)$
⟨*proof*⟩

lemma *zero-when* [*simp*]:
$(0\ when\ P) = 0$
⟨*proof*⟩

lemma *when-when*:
$(a\ when\ P\ when\ Q) = (a\ when\ P \wedge Q)$
⟨*proof*⟩

lemma *when-commute*:
$(a\ when\ Q\ when\ P) = (a\ when\ P\ when\ Q)$
⟨*proof*⟩

lemma *when-neq-zero* [*simp*]:
$(a\ when\ P) \neq 0 \longleftrightarrow P \wedge a \neq 0$
⟨*proof*⟩

end

context *monoid-add*
begin

lemma *when-add-distrib*:
$(a + b\ when\ P) = (a\ when\ P) + (b\ when\ P)$
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *zero-power-eq*:
$0\ \hat{}\ n = (1\ when\ n = 0)$
⟨*proof*⟩

end

context *comm-monoid-add*
begin

lemma *Sum-any-when-equal* [*simp*]:
$(\sum a.\ (f\ a\ when\ a = b)) = f\ b$
⟨*proof*⟩

lemma *Sum-any-when-equal'* [*simp*]:
$(\sum a.\ (f\ a\ when\ b = a)) = f\ b$
⟨*proof*⟩

lemma *Sum-any-when-independent*:
$(\sum a.\ g\ a\ when\ P) = ((\sum a.\ g\ a)\ when\ P)$
⟨*proof*⟩

lemma *Sum-any-when-dependent-prod-right*:
$(\sum (a,\ b).\ g\ a\ when\ b = h\ a) = (\sum a.\ g\ a)$
⟨*proof*⟩

lemma *Sum-any-when-dependent-prod-left*:
$(\sum (a,\ b).\ g\ b\ when\ a = h\ b) = (\sum b.\ g\ b)$
⟨*proof*⟩

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
$(a - b\ when\ P) = (a\ when\ P) - (b\ when\ P)$
⟨*proof*⟩

end

context *group-add*
begin

lemma *when-uminus-distrib*:
$(-\ a\ when\ P) = -\ (a\ when\ P)$
⟨*proof*⟩

end

context *mult-zero*
begin

lemma *mult-when*:
$a * (b \ when \ P) = (a * b \ when \ P)$
⟨*proof*⟩

lemma *when-mult*:
$(a \ when \ P) * b = (a * b \ when \ P)$
⟨*proof*⟩

end

28.2 Type definition

The following type is of central importance:

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

lemma *lookup-Abs-poly-mapping*:
$finite \ \{x. \ f \ x \neq 0\} \Longrightarrow lookup \ (Abs\text{-}poly\text{-}mapping \ f) = f$
⟨*proof*⟩

lemma *finite-lookup* [*simp*]:
$finite \ \{k. \ lookup \ f \ k \neq 0\}$
⟨*proof*⟩

lemma *finite-lookup-nat* [*simp*]:
fixes $f :: 'a \Rightarrow_0 nat$
shows $finite \ \{k. \ 0 < lookup \ f \ k\}$
⟨*proof*⟩

lemma *poly-mapping-eqI*:
assumes $\bigwedge k. \ lookup \ f \ k = lookup \ g \ k$
shows $f = g$
⟨*proof*⟩

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

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

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

satisfying

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

finite {*x*. *?f x* ≠ (*0*::*?'b*)} ⟹ *lookup* (*Abs-poly-mapping* *?f*) = *?f*

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

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

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

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

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

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

28.3 Additive structure

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

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

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

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

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

lift-definition *zero-poly-mapping* :: $'a \Rightarrow_0 'b$

is $\lambda k.\ 0$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *lookup-zero* [*simp*]:
lookup 0 k = 0
⟨*proof*⟩

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*⟩

instance
⟨*proof*⟩

end

lemma *lookup-add*:
lookup $(f + g)\ k =$ *lookup* $f\ k +$ *lookup* $g\ k$
⟨*proof*⟩

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

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*⟩

instance
⟨*proof*⟩

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*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-uminus* [*simp*]:
lookup (− *f*) *k* = − *lookup f k*
⟨*proof*⟩

lemma *lookup-minus*:
lookup (*f* − *g*) *k* = *lookup f k* − *lookup g k*
⟨*proof*⟩

28.4 Multiplicative structure

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

lift-definition *one-poly-mapping* :: *'a* ⇒$_0$ *'b*
is λ*k*. *1 when k* = *0*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-one*:
lookup 1 k = (*1 when k* = *0*)
⟨*proof*⟩

lemma *lookup-one-zero* [*simp*]:
lookup 1 0 = *1*
⟨*proof*⟩

definition *prod-fun* :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a*::*monoid-add* ⇒ *'b*::*semiring-0*
where
prod-fun f1 f2 k = (∑ *l*. *f1 l* ∗ (∑ *q*. (*f2 q when k* = *l* + *q*)))

lemma *prod-fun-unfold-prod*:
fixes *f g* :: *'a* :: *monoid-add* ⇒ *'b*::*semiring-0*
assumes *fin-f*: *finite* {*a*. *f a* ≠ *0*}
assumes *fin-g*: *finite* {*b*. *g b* ≠ *0*}
shows *prod-fun f g k* = (∑ (*a*, *b*). *f a* ∗ *g b when k* = *a* + *b*)
⟨*proof*⟩

lemma *finite-prod-fun*:
fixes *f1 f2* :: *'a* :: *monoid-add* ⇒ *'b* :: *semiring-0*
assumes *fin1*: *finite* {*l*. *f1 l* ≠ *0*}
and *fin2*: *finite* {*q*. *f2 q* ≠ *0*}
shows *finite* {*k*. *prod-fun f1 f2 k* ≠ *0*}
⟨*proof*⟩

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

lift-definition *times-poly-mapping* :: (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *'a* ⇒$_0$ *'b*
is *prod-fun*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *lookup-mult*:
lookup (*f* ∗ *g*) *k* = (∑ *l*. *lookup f l* ∗ (∑ *q*. *lookup g q when k* = *l* + *q*))
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0*) *comm-semiring-0*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-1*) *comm-semiring-1*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-1-cancel*) *semiring-1-cancel*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring*) *ring*
⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring*) *comm-ring*
⟨*proof*⟩

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

⟨*proof*⟩

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*
⟨*proof*⟩

28.5 Single-point mappings

lift-definition *single* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_0 'b$::*zero*
is $\lambda k\ v\ k'.\ (v\ when\ k = k')$
⟨*proof*⟩

lemma *inj-single* [*iff*]:
inj (*single k*)
⟨*proof*⟩

lemma *lookup-single*:
lookup (*single k v*) $k' = (v\ when\ k = k')$
⟨*proof*⟩

lemma *lookup-single-eq* [*simp*]:
lookup (*single k v*) *k* = *v*
⟨*proof*⟩

lemma *lookup-single-not-eq*:
$k \neq k' \Longrightarrow$ *lookup* (*single k v*) $k' = 0$
⟨*proof*⟩

lemma *single-zero* [*simp*]:
single k 0 = *0*
⟨*proof*⟩

lemma *single-one* [*simp*]:
single 0 1 = *1*
⟨*proof*⟩

lemma *single-add*:
single k (*a* + *b*) = *single k a* + *single k b*
⟨*proof*⟩

lemma *single-uminus*:
single k (− *a*) = − *single k a*
⟨*proof*⟩

lemma *single-diff*:
single k (*a* − *b*) = *single k a* − *single k b*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *lookup-numeral*:
lookup (*numeral n*) *k* = (*numeral n when k* = *0*)
⟨*proof*⟩

lemma *single-of-nat* [*simp*]:
single 0 (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *lookup-of-nat*:
lookup (*of-nat n*) *k* = (*of-nat n when k* = *0*)
⟨*proof*⟩

lemma *of-nat-single*:
of-nat = *single 0* ∘ *of-nat*
⟨*proof*⟩

lemma *mult-single*:
single k a ∗ *single l b* = *single* (*k* + *l*) (*a* ∗ *b*)
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*
⟨*proof*⟩

lemma *single-of-int* [*simp*]:
single 0 (*of-int k*) = *of-int k*
⟨*proof*⟩

lemma *lookup-of-int*:
lookup (*of-int l*) *k* = (*of-int l when k* = *0*)
⟨*proof*⟩

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*⟩

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

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

⟨*proof*⟩

28.7 Mapping order

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

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

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

instance ⟨*proof*⟩

end

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

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

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

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

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
⟨*proof*⟩

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* ⇒$_0$ '*b*::*zero*) ⇒ '*a set*
is λ*f*. {*k*. *f k* ≠ *0*} ⟨*proof*⟩

lift-definition *range* :: ('*a* ⇒$_0$ '*b*::*zero*) ⇒ '*b set*
is λ*f* :: '*a* ⇒ '*b*. *Set.range f* − {*0*} ⟨*proof*⟩

lemma *finite-keys* [*simp*]:
finite (*keys f*)
⟨*proof*⟩

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

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

lemma *lookup-eq-zero-in-keys-contradict* [*dest*]:
$lookup\ f\ k = 0 \Longrightarrow \neg\ k \in keys\ f$
⟨*proof*⟩

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

lemma *in-keys-lookup-in-range* [*simp*]:
$k \in keys\ f \Longrightarrow lookup\ f\ k \in range\ f$
⟨*proof*⟩

lemma *keys-zero* [*simp*]:
$keys\ 0 = \{\}$
⟨*proof*⟩

lemma *range-zero* [*simp*]:
$range\ 0 = \{\}$
⟨*proof*⟩

lemma *keys-add-subset*:
$keys\ (f + g) \subseteq keys\ f \cup keys\ g$
⟨*proof*⟩

lemma *keys-one* [*simp*]:
$keys\ 1 = \{0\}$
⟨*proof*⟩

lemma *range-one* [*simp*]:
$range\ 1 = \{1\}$
⟨*proof*⟩

lemma *keys-single* [*simp*]:
$keys\ (single\ k\ v) = (if\ v = 0\ then\ \{\}\ else\ \{k\})$
⟨*proof*⟩

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

range (*single k v*) = (*if v = 0 then* {} *else* {*v*})
⟨*proof*⟩

lemma *keys-mult*:
keys (*f* ∗ *g*) ⊆ {*a* + *b* | *a b*. *a* ∈ *keys f* ∧ *b* ∈ *keys g*}
⟨*proof*⟩

lemma *setsum-keys-plus-distrib*:
assumes *hom-0*: ⋀*k*. *f k 0* = *0*
and *hom-plus*: ⋀*k*. *k* ∈ *PP-Poly-Mapping.keys p* ∪ *PP-Poly-Mapping.keys q* ⟹ *f k* (*PP-Poly-Mapping.lookup p k* + *PP-Poly-Mapping.lookup q k*) = *f k* (*PP-Poly-Mapping.lookup p k*) + *f k* (*PP-Poly-Mapping.lookup q k*)
shows
(∑ *k*∈*PP-Poly-Mapping.keys* (*p* + *q*). *f k* (*PP-Poly-Mapping.lookup* (*p* + *q*) *k*))
=
(∑ *k*∈*PP-Poly-Mapping.keys p*. *f k* (*PP-Poly-Mapping.lookup p k*)) +
(∑ *k*∈*PP-Poly-Mapping.keys q*. *f k* (*PP-Poly-Mapping.lookup q k*))
(**is** *?lhs* = *?p* + *?q*)
⟨*proof*⟩

28.9 Degree

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

lemma *degree-zero* [*simp*]:
degree 0 = *0*
⟨*proof*⟩

lemma *degree-one* [*simp*]:
degree 1 = *1*
⟨*proof*⟩

lemma *degree-single-zero* [*simp*]:
degree (*single k 0*) = *0*
⟨*proof*⟩

lemma *degree-single-not-zero* [*simp*]:
v ≠ *0* ⟹ *degree* (*single k v*) = *Suc k*
⟨*proof*⟩

lemma *degree-zero-iff* [*simp*]:
degree f = *0* ⟷ *f* = *0*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *in-keys-less-degree*:
  $n \in keys\ f \Longrightarrow n < degree\ f$
⟨*proof*⟩

lemma *beyond-degree-lookup-zero*:
  $degree\ f \leq n \Longrightarrow lookup\ f\ n = 0$
  ⟨*proof*⟩

lemma *degree-add*:
  $degree\ (f + g) \leq max\ (degree\ f)\ (PP\text{-}Poly\text{-}Mapping.degree\ g)$
⟨*proof*⟩

lemma *sorted-list-of-set-keys*:
  $sorted\text{-}list\text{-}of\text{-}set\ (keys\ f) = filter\ (\lambda k.\ k \in keys\ f)\ [0..<degree\ f]$ (**is** - = *?r*)
⟨*proof*⟩

28.10 Inductive structure

lift-definition *update* :: $'a \Rightarrow 'b \Rightarrow ('a \Rightarrow_0 'b::zero) \Rightarrow 'a \Rightarrow_0 'b$
  is $\lambda k\ v\ f.\ f(k := v)$
⟨*proof*⟩

lemma *update-induct* [*case-names const update*]:
  assumes *const'*: $P\ 0$
  assumes *update'*: $\bigwedge f\ a\ b.\ a \notin keys\ f \Longrightarrow b \neq 0 \Longrightarrow P\ f \Longrightarrow P\ (update\ a\ b\ f)$
  shows $P\ f$
⟨*proof*⟩

lemma *lookup-update*:
  $lookup\ (update\ k\ v\ f)\ k' = (if\ k = k'\ then\ v\ else\ lookup\ f\ k')$
  ⟨*proof*⟩

lemma *keys-update*:
  $keys\ (update\ k\ v\ f) = (if\ v = 0\ then\ keys\ f - \{k\}\ else\ insert\ k\ (keys\ f))$
  ⟨*proof*⟩

28.11 Quasi-functorial structure

lift-definition *map* :: $('b::zero \Rightarrow 'c::zero)$
  $\Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c::zero)$
  is $\lambda g\ f\ k.\ g\ (f\ k)\ when\ f\ k \neq 0$
  ⟨*proof*⟩

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*⟩

end

lemma *map-key-compose*:
  assumes [*transfer-rule*]: *inj f inj g*
  shows *map-key f* (*map-key g p*) = *map-key* (*g* ∘ *f*) *p*
⟨*proof*⟩

lemma *map-key-id*:
  map-key (λ*x*. *x*) *p* = *p*
⟨*proof*⟩

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

lemma *map-key-map*:
  map-key f (*map g p*) = *map g* (*map-key f p*)
  ⟨*proof*⟩

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

lemma *keys-map-key*:
  keys (*map-key f p*) = *f* −‘ *keys p*
  ⟨*proof*⟩

lemma *map-key-zero* [*simp*]:
  map-key f 0 = *0*
  ⟨*proof*⟩

lemma *map-key-single* [*simp*]:
  map-key f (*single* (*f k*) *v*) = *single k v*
  ⟨*proof*⟩

end

lemma *mult-map-scale-conv-mult*: *map* (*op* ∗ *s*) *p* = *single 0 s* ∗ *p*
⟨*proof*⟩

lemma *map-single* [*simp*]:
  (*c* = *0* ⟹ *f 0* = *0*) ⟹ *map f* (*single x c*) = *single x* (*f c*)
⟨*proof*⟩

lemma *map-eq-zero-iff*: *map f p = 0 ⟷ (∀ k ∈ keys p. f (lookup p k) = 0)*
⟨*proof*⟩

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

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

lift-definition *nth* :: *'a list ⇒ (nat ⇒$_0$ 'a::zero)*
is *nth-default 0*
⟨*proof*⟩

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

lemma *nth-Nil* [*simp*]:
nth [] = 0
⟨*proof*⟩

lemma *nth-singleton* [*simp*]:
nth [v] = single 0 v
⟨*proof*⟩

lemma *nth-replicate* [*simp*]:
nth (replicate n 0 @ [v]) = single n v
⟨*proof*⟩

lemma *nth-strip-while* [*simp*]:
nth (strip-while (op = 0) xs) = nth xs
⟨*proof*⟩

lemma *nth-strip-while'* [*simp*]:
nth (strip-while (λk. k = 0) xs) = nth xs
⟨*proof*⟩

lemma *nth-eq-iff*:
nth xs = nth ys ⟷ strip-while (HOL.eq 0) xs = strip-while (HOL.eq 0) ys
⟨*proof*⟩

lemma *lookup-nth* [*simp*]:
lookup (nth xs) = nth-default 0 xs
⟨*proof*⟩

lemma *keys-nth* [*simp*]:
keys (nth xs) = fst ‘ {(n, v) ∈ set (enumerate 0 xs). v ≠ 0}
⟨*proof*⟩

lemma *range-nth* [*simp*]:
range (nth xs) = set xs − {0}
⟨*proof*⟩

lemma *degree-nth*:
no-trailing-zeros xs $\Longrightarrow$ *degree* (*nth xs*) = *length xs*
⟨*proof*⟩

lemma *nth-trailing-zeros* [*simp*]:
nth (*xs* @ *replicate n 0*) = *nth xs*
⟨*proof*⟩

lemma *nth-idem*:
nth (*List.map* (*lookup f*) [*0*..<*degree f*]) = *f*
⟨*proof*⟩

lemma *nth-idem-bound*:
assumes *degree f* ≤ *n*
shows *nth* (*List.map* (*lookup f*) [*0*..<*n*]) = *f*
⟨*proof*⟩

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

lift-definition *the-value* :: (*'a* × *'b*) *list* ⇒ *'a* ⇒$_0$ *'b*::*zero*
is λ*xs k*. *case map-of xs k of None* ⇒ *0* | *Some v* ⇒ *v*
⟨*proof*⟩

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

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *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*
⟨*proof*⟩

lemma *lookup-the-value*:
lookup (*the-value xs*) *k* = (*case map-of xs k of None* ⇒ *0* | *Some v* ⇒ *v*)
⟨*proof*⟩

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

lemma *the-value-Nil* [*simp*]:
the-value [] = *0*
⟨*proof*⟩

lemma *the-value-Cons* [*simp*]:
the-value (*x* # *xs*) = *update* (*fst x*) (*snd x*) (*the-value xs*)
⟨*proof*⟩

lemma *items-zero* [*simp*]:
items 0 = []
⟨*proof*⟩

lemma *items-one* [*simp*]:
items 1 = [(*0*, *1*)]
⟨*proof*⟩

lemma *items-single* [*simp*]:
items (*single k v*) = (*if v* = *0 then* [] *else* [(*k*, *v*)])
⟨*proof*⟩

lemma *in-set-items-iff* [*simp*]:
(*k*, *v*) ∈ *set* (*items f*) ⟷ *k* ∈ *keys f* ∧ *lookup f k* = *v*
⟨*proof*⟩

28.14 Size estimation

context
fixes *f* :: *'a* ⇒ *nat*
and *g* :: *'b* :: *zero* ⇒ *nat*
begin

definition *poly-mapping-size* :: (*'a* ⇒$_0$ *'b*) ⇒ *nat*
where
poly-mapping-size m = *g 0* + (∑ *k* ∈ *keys m*. *Suc* (*f k* + *g* (*lookup m k*)))

lemma *poly-mapping-size-0* [*simp*]:
poly-mapping-size 0 = *g 0*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *keys-less-poly-mapping-size*:
k ∈ *keys m* ⟹ *f k* + *g* (*lookup m k*) < *poly-mapping-size m*
⟨*proof*⟩

lemma *lookup-le-poly-mapping-size*:
g (*lookup m k*) ≤ *poly-mapping-size m*
⟨*proof*⟩

lemma *poly-mapping-size-estimation*:
k ∈ *keys m* ⟹ *y* ≤ *f k* + *g* (*lookup m k*) ⟹ *y* < *poly-mapping-size m*

⟨*proof*⟩

lemma *poly-mapping-size-estimation2*:
assumes $v \in range\ m$ **and** $y \leq g\ v$
shows $y < poly\text{-}mapping\text{-}size\ m$
⟨*proof*⟩

end

lemma *poly-mapping-size-one* [*simp*]:
$poly\text{-}mapping\text{-}size\ f\ g\ 1 = g\ 0 + f\ 0 + g\ 1 + 1$
⟨*proof*⟩

lemma *poly-mapping-size-cong* [*fundef-cong*]:
$m = m' \Longrightarrow g\ 0 = g'\ 0 \Longrightarrow (\bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k = f'\ k)$
$\Longrightarrow (\bigwedge v.\ v \in range\ m' \Longrightarrow g\ v = g'\ v)$
$\Longrightarrow poly\text{-}mapping\text{-}size\ f\ g\ m = poly\text{-}mapping\text{-}size\ f'\ g'\ m'$
⟨*proof*⟩

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

definition $size = poly\text{-}mapping\text{-}size\ (\lambda\text{-}.\ 0)\ (\lambda\text{-}.\ 0)$

instance ⟨*proof*⟩

end

28.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::
$('a \Rightarrow 'b :: zero \Rightarrow 'c :: zero) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c)$
is $\lambda f\ p\ k.\ (if\ k \in keys\ p\ then\ f\ k\ (lookup\ p\ k)\ else\ 0)$
⟨*proof*⟩

lemma *mapp-cong* [*fundef-cong*]:
$[\![\ m = m';\ \bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k\ (lookup\ m'\ k) = f'\ k\ (lookup\ m'\ k)\ ]\!]$
$\Longrightarrow mapp\ f\ m = mapp\ f'\ m'$
⟨*proof*⟩

lemma *lookup-mapp*:
$lookup\ (mapp\ f\ p)\ k = (f\ k\ (lookup\ p\ k)\ when\ k \in keys\ p)$
⟨*proof*⟩

lemma *keys-mapp-subset*: $keys\ (mapp\ f\ p) \subseteq keys\ p$
⟨*proof*⟩

hide-const (**open**) *lookup single update keys range map map-key degree nth the-value*

items foldr mapp

end

29 An abstract type for multivariate polynomials

theory *PP-MPoly*
imports *PP-Poly-Mapping*
begin

29.1 Abstract type definition

typedef (**overloaded**) *'a mpoly* =
  UNIV :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*zero*) *set*
  morphisms *mapping-of MPoly*
⟨*proof*⟩

setup-lifting *type-definition-mpoly*

thm *mapping-of-inverse* **thm** *MPoly-inverse*
thm *mapping-of-inject* **thm** *MPoly-inject*
thm *mapping-of-induct* **thm** *MPoly-induct*
thm *mapping-of-cases* **thm** *MPoly-cases*

29.2 Additive structure

instantiation *mpoly* :: (*zero*) *zero*
begin

lift-definition *zero-mpoly* :: *'a mpoly*
  is *0* :: (*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a* ⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *mpoly* :: (*monoid-add*) *monoid-add*
begin

lift-definition *plus-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
  is *Groups.plus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

instance *mpoly* :: (*comm-monoid-add*) *comm-monoid-add*

⟨*proof*⟩

instantiation *mpoly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
is *Groups.minus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *mpoly* :: (*ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-mpoly* :: *'a mpoly* ⇒ *'a mpoly*
is *Groups.uminus* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

29.3 Multiplication by a coefficient

lift-definition *smult* :: *'a*::{*times*,*zero*} ⇒ *'a mpoly* ⇒ *'a mpoly*
is *λa. PP-Poly-Mapping.map* (*Groups.times a*) :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

29.4 Multiplicative structure

instantiation *mpoly* :: (*zero-neq-one*) *zero-neq-one*
begin

lift-definition *one-mpoly* :: *'a mpoly*
is *1* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *mpoly* :: (*semiring-0*) *semiring-0*
begin

lift-definition *times-mpoly* :: *'a mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly*
is *Groups.times* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *mpoly* :: (*comm-semiring-0*) *comm-semiring-0*
⟨*proof*⟩

instance *mpoly* :: (*semiring-0-cancel*) *semiring-0-cancel*
⟨*proof*⟩

instance *mpoly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel*
⟨*proof*⟩

instance *mpoly* :: (*semiring-1*) *semiring-1*
⟨*proof*⟩

instance *mpoly* :: (*comm-semiring-1*) *comm-semiring-1*
⟨*proof*⟩

instance *mpoly* :: (*semiring-1-cancel*) *semiring-1-cancel*
⟨*proof*⟩

instance *mpoly* :: (*ring*) *ring*
⟨*proof*⟩

instance *mpoly* :: (*comm-ring*) *comm-ring*
⟨*proof*⟩

instance *mpoly* :: (*ring-1*) *ring-1*
⟨*proof*⟩

instance *mpoly* :: (*comm-ring-1*) *comm-ring-1*
⟨*proof*⟩

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* ⇒$_0$ *nat*) ⇒ *'a*::*zero* ⇒ *'a mpoly*
is *PP-Poly-Mapping.single* :: (*nat* ⇒$_0$ *nat*) ⇒ - ⟨*proof*⟩

lemma *mapping-of-monom* [*simp*]:
mapping-of (*monom m a*) = *PP-Poly-Mapping.single m a*
⟨*proof*⟩

lemma *monom-zero* [*simp*]:
monom 0 0 = 0
⟨*proof*⟩

lemma *monom-one* [*simp*]:
monom 0 1 = 1
⟨*proof*⟩

lemma *monom-add*:
monom m (a + b) = monom m a + monom m b
⟨*proof*⟩

lemma *monom-uminus*:
monom m (− a) = − monom m a
⟨*proof*⟩

lemma *monom-diff*:
monom m (a − b) = monom m a − monom m b
⟨*proof*⟩

lemma *monom-numeral* [*simp*]:
monom 0 (numeral n) = numeral n
⟨*proof*⟩

lemma *monom-of-nat* [*simp*]:
monom 0 (of-nat n) = of-nat n
⟨*proof*⟩

lemma *of-nat-monom*:
of-nat = monom 0 ∘ of-nat
⟨*proof*⟩

lemma *inj-monom* [*iff*]:
inj (monom m)
⟨*proof*⟩

lemma *mult-monom*: *monom x a ∗ monom y b = monom (x + y) (a ∗ b)*
⟨*proof*⟩

instance *mpoly* :: (*semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *mpoly* :: (*ring-char-0*) *ring-char-0*
⟨*proof*⟩

lemma *monom-of-int* [*simp*]:
monom 0 (of-int k) = of-int k
⟨*proof*⟩

29.6 Integral domains

instance *mpoly* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
⟨*proof*⟩

instance *mpoly* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors*
⟨*proof*⟩

instance *mpoly* :: (*idom*) *idom*
⟨*proof*⟩

29.7 Monom coefficient lookup

definition *coeff* :: *'a*::*zero mpoly* ⇒ (*nat* ⇒$_0$ *nat*) ⇒ *'a*
where
coeff p = *PP-Poly-Mapping.lookup* (*mapping-of p*)

29.8 Insertion morphism

definition *insertion-fun-natural* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒ *nat*) ⇒ *'a*) ⇒ *'a*::*comm-semiring-1*
where
insertion-fun-natural f p = (∑ *m*. *p m* ∗ (∏ *v*. *f v* ^ *m v*))

definition *insertion-fun* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒$_0$ *nat*) ⇒ *'a*) ⇒ *'a*::*comm-semiring-1*
where
insertion-fun f p = (∑ *m*. *p m* ∗ (∏ *v*. *f v* ^ *PP-Poly-Mapping.lookup m v*))

N.b. have been unable to relate this to *insertion-fun-natural* using lifting!

lift-definition *insertion-aux* :: (*nat* ⇒ *'a*) ⇒ ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ *'a*::*comm-semiring-1*
is *insertion-fun* ⟨*proof*⟩

lift-definition *insertion* :: (*nat* ⇒ *'a*) ⇒ *'a mpoly* ⇒ *'a*::*comm-semiring-1*
is *insertion-aux* ⟨*proof*⟩

lemma *aux*:
PP-Poly-Mapping.lookup f = (λ-. *0*) ⟷ *f* = *0*
⟨*proof*⟩

lemma *insertion-trivial* [*simp*]:
insertion (λ-. *0*) *p* = *coeff p 0*
⟨*proof*⟩

lemma *insertion-zero* [*simp*]:
insertion f 0 = *0*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *insertion-add*:
insertion f (p + q) = insertion f p + insertion f q
⟨*proof*⟩

lemma *insertion-one* [*simp*]:
insertion f 1 = 1
⟨*proof*⟩

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*⟩

lemma *insertion-mult*:
*insertion f (p * q) = insertion f p * insertion f q*
⟨*proof*⟩

29.9 Degree

lift-definition *degree* :: *'a::zero mpoly ⇒ nat ⇒ nat*
is *λp v. Max (insert 0 ((λm. PP-Poly-Mapping.lookup m v) ‘ PP-Poly-Mapping.keys p))* ⟨*proof*⟩

lift-definition *total-degree* :: *'a::zero mpoly ⇒ nat*
is *λp. Max (insert 0 ((λm. sum (PP-Poly-Mapping.lookup m) (PP-Poly-Mapping.keys m)) ‘ PP-Poly-Mapping.keys p))* ⟨*proof*⟩

lemma *degree-zero* [*simp*]:
degree 0 v = 0
⟨*proof*⟩

lemma *total-degree-zero* [*simp*]:
total-degree 0 = 0
⟨*proof*⟩

lemma *degree-one* [*simp*]:
degree 1 v = 0
⟨*proof*⟩

lemma *total-degree-one* [*simp*]:
total-degree 1 = 0
⟨*proof*⟩

29.10 Pseudo-division of polynomials

lemma *smult-conv-mult*: *smult s p* = *monom 0 s* ∗ *p*
⟨*proof*⟩

lemma *smult-monom* [*simp*]:
fixes *c* :: - :: *mult-zero*
shows *smult c* (*monom x c′*) = *monom x* (*c* ∗ *c′*)
⟨*proof*⟩

lemma *smult-0* [*simp*]:
fixes *p* :: - :: *mult-zero mpoly*
shows *smult 0 p* = *0*
⟨*proof*⟩

lemma *mult-smult-left*: *smult s p* ∗ *q* = *smult s* (*p* ∗ *q*)
⟨*proof*⟩

lift-definition *sdiv* :: *′a*::*ring-div* ⇒ *′a mpoly* ⇒ *′a mpoly*
is λ*a*. *PP-Poly-Mapping.map* (λ*b*. *b div a*) :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *′a*) ⇒ -
⟨*proof*⟩

'Polynomial division' is only possible on univariate polynomials $K[x]$ over a field K, all other kinds of polynomials only allow pseudo-division [1]p.40/41":

∀ *x y* :: *′a mpoly*. *y* ≠ *0* ⇒ ∃ *a q r*. *smult a x* = *q* ∗ *y* + *r*

The introduction of pseudo-division below generalises `~~/src/HOL/Computational_Algebra/Polynomial.thy`. [1] Winkler, Polynomial Algorithms, 1996. The generalisation raises issues addressed by Wenda Li and commented below. Florian replied to the issues conjecturing, that the abstract mpoly needs not be aware of the issues, in case these are only concerned with executability.

definition *pseudo-divmod-rel*
:: *′a*::*ring-div* => *′a mpoly* => *′a mpoly* => *′a mpoly* => *′a mpoly* => *bool*
where
pseudo-divmod-rel a x y q r ⟷
smult a x = *q* ∗ *y* + *r* ∧ (*if y* = *0 then q* = *0 else r* = *0* ∨ *degree r* < *degree y*)

definition *pdiv* :: *′a*::*ring-div mpoly* ⇒ *′a mpoly* ⇒ (*′a* × *′a mpoly*) (**infixl** *pdiv 70*)
where
x pdiv y = (*THE* (*a*, *q*). ∃ *r*. *pseudo-divmod-rel a x y q r*)

definition *pmod* :: *′a*::*ring-div mpoly* ⇒ *′a mpoly* ⇒ *′a mpoly* (**infixl** *pmod 70*)
where
x pmod y = (*THE r*. ∃ *a q*. *pseudo-divmod-rel a x y q r*)

definition *pdivmod* :: *′a*::*ring-div mpoly* ⇒ *′a mpoly* ⇒ (*′a* × *′a mpoly*) × *′a mpoly*

where
pdivmod p q = (p pdiv q, p pmod q)

lemma *pdiv-code*:
p pdiv q = fst (pdivmod p q)
⟨*proof*⟩

lemma *pmod-code*:
p pmod q = snd (pdivmod p q)
⟨*proof*⟩

definition *div* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *div 70*)
where
x div y = (*THE q'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*q'* = *smult* (*inverse a*) *q*))

definition *mod* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* (**infixl** *mod 70*)
where
x mod y = (*THE r'*. ∃ *a q r*. (*pseudo-divmod-rel a x y q r*) ∧ (*r'* = *smult* (*inverse a*) *r*))

definition *divmod* :: *'a*::{*ring-div*,*field*} *mpoly* ⇒ *'a mpoly* ⇒ *'a mpoly* × *'a mpoly*
where
divmod p q = (*p div q*, *p mod q*)

lemma *div-poly-code*:
p div q = fst (divmod p q)
⟨*proof*⟩

lemma *mod-poly-code*:
p mod q = snd (divmod p q)
⟨*proof*⟩

29.11 Primitive poly, etc

lift-definition *coeffs* :: *'a* :: *zero mpoly* ⇒ *'a set*
is *PP-Poly-Mapping.range* :: ((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*) ⇒ - ⟨*proof*⟩

lemma *finite-coeffs* [*simp*]: *finite* (*coeffs p*)
⟨*proof*⟩

[1]p.82 A "primitive"' polynomial has coefficients with GCD equal to 1. A polynomial is factored into "content" and "primitive part" for many different purposes.

definition *primitive* :: *'a*::{*ring-div*,*semiring-Gcd*} *mpoly* ⇒ *bool*
where

primitive p $\longleftrightarrow$ *Gcd* (*coeffs p*) = *1*

definition *content-primitive* :: *'a*::{*ring-div*,*GCD.Gcd*} *mpoly* ⇒ *'a* × *'a mpoly*
where
content-primitive p = (
let d = *Gcd* (*coeffs p*)
in (*d*, *sdiv d p*))

value *let p* = *M* [*1*,*2*,*3*] (*4*::*int*) + *M* [*2*,*0*,*4*] *6* + *M* [*2*,*0*,*5*] *8*
in content-primitive p

end

theory *PP-More-MPoly*
imports *PP-MPoly*
begin

abbreviation *lookup* == *PP-Poly-Mapping.lookup*
abbreviation *keys* == *PP-Poly-Mapping.keys*

30 MPpoly Mapping extenion

lemma *lookup-Abs-poly-mapping-when-finite*:
assumes *finite S*
shows *lookup* (*Abs-poly-mapping* (λ*x*. *f x when x*∈*S*)) = (λ*x*. *f x when x*∈*S*)
⟨*proof*⟩

definition *remove-key*::*'a* ⇒ (*'a* ⇒$_0$ *'b*::*monoid-add*) ⇒ (*'a* ⇒$_0$ *'b*) **where**
remove-key k0 f = *Abs-poly-mapping* (λ*k*. *lookup f k when k* ≠ *k0*)

lemma *remove-key-lookup*:
lookup (*remove-key k0 f*) *k* = (*lookup f k when k* ≠ *k0*)
⟨*proof*⟩

lemma *remove-key-keys*: *keys f* − {*k*} = *keys* (*remove-key k f*) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *remove-key-sum*: *remove-key k f* + *PP-Poly-Mapping.single k* (*lookup f k*)
= *f*
⟨*proof*⟩

lemma *remove-key-single*[*simp*]: *remove-key v* (*PP-Poly-Mapping.single v n*) = *0*
⟨*proof*⟩

lemma *remove-key-add*: *remove-key v m* + *remove-key v m'* = *remove-key v* (*m*
+ *m'*)

⟨*proof*⟩

lemma *poly-mapping-induct* [*case-names single sum*]:
fixes *P*::('*a*, '*b*::*monoid-add*) *poly-mapping* ⇒ *bool*
assumes *single*:⋀*k v*. *P* (*PP-Poly-Mapping.single k v*)
and *sum*:(⋀*f g k v*. *P f* ⟹ *P g* ⟹ *g* = (*PP-Poly-Mapping.single k v*) ⟹ *k* ∉ *keys f* ⟹ *P* (*f*+*g*))
shows *P f* ⟨*proof*⟩

lemma *map-lookup*:
assumes *g 0* = *0*
shows *lookup* (*PP-Poly-Mapping.map g f*) *x* = *g* ((*lookup f*) *x*)
⟨*proof*⟩

lemma *keys-add*:
assumes *keys f* ∩ *keys g* = {}
shows *keys f* ∪ *keys g* = *keys* (*f*+*g*)
⟨*proof*⟩

lemma *fun-when*:
f 0 = *0* ⟹ *f* (*a when P*) = (*f a when P*) ⟨*proof*⟩

31 MPoly extension

lemma *coeff-all-0*:(⋀*m*. *coeff p m* = *0*) ⟹ *p*=*0*
⟨*proof*⟩

definition *vars*::'*a*::*zero mpoly* ⇒ *nat set* **where**
vars p = *UNION* (*keys* (*mapping-of p*)) (λ*m*. *keys m*)

lemma *vars-finite*: *finite* (*vars p*) ⟨*proof*⟩

lemma *vars-monom-single*: *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*) ⊆ {*v*}
⟨*proof*⟩

lemma *vars-monom-keys*:
assumes *a*≠*0*
shows *vars* (*monom m a*) = *keys m*
⟨*proof*⟩

lemma *vars-monom-subset*:
shows *vars* (*monom m a*) ⊆ *keys m*
⟨*proof*⟩

lemma *vars-monom-single-cases*: *vars* (*monom* (*PP-Poly-Mapping.single v k*) *a*)
= (*if k*=*0* ∨ *a*=*0 then* {} *else* {*v*})
⟨*proof*⟩

lemma *vars-monom*:
assumes *a≠0*
shows *vars* (*monom m* (*1*::*'a*::*zero-neq-one*)) = *vars* (*monom m* (*a*::*'a*))
⟨*proof*⟩

lemma *vars-add*: *vars* (*p1* + *p2*) ⊆ *vars p1* ∪ *vars p2*
⟨*proof*⟩

lemma *vars-mult*: *vars* (*p*∗*q*) ⊆ *vars p* ∪ *vars q*
⟨*proof*⟩

lemma *vars-add-monom*:
assumes *p2* = *monom m a m* ∉ *keys* (*mapping-of p1*)
shows *vars* (*p1* + *p2*) = *vars p1* ∪ *vars p2*
⟨*proof*⟩

lemma *vars-setsum*: *finite S* ⟹ *vars* (∑ *m*∈*S*. *f m*) ⊆ (⋃ *m*∈*S*. *vars* (*f m*))
⟨*proof*⟩

lemma *coeff-monom*: *coeff* (*monom m a*) *m'* = (*a when m'*=*m*)
⟨*proof*⟩

lemma *coeff-add*: *coeff p m* + *coeff q m* = *coeff* (*p*+*q*) *m*
⟨*proof*⟩

lemma *coeff-eq*: *coeff p* = *coeff q* ⟷ *p*=*q* ⟨*proof*⟩

lemma *coeff-monom-mult*: *coeff* ((*monom m' a*) ∗ *q*) (*m'* + *m*) = *a* ∗ *coeff q m*
⟨*proof*⟩

lemma *one-term-is-monomial*:
assumes *card* (*keys* (*mapping-of p*)) ≤ *1*
obtains *m* **where** *p* = *monom m* (*coeff p m*)
⟨*proof*⟩

definition *remove-term*::(*nat* ⇒$_0$ *nat*) ⇒ *'a*::*zero mpoly* ⇒ *'a mpoly* **where**
remove-term m0 p = *MPoly* (*Abs-poly-mapping* (λ*m*. *coeff p m when m* ≠ *m0*))

lemma *remove-term-coeff*: *coeff* (*remove-term m0 p*) *m* = (*coeff p m when m* ≠ *m0*)
⟨*proof*⟩

lemma *coeff-keys*: *m* ∈ *keys* (*mapping-of p*) ⟷ *coeff p m* ≠ *0* ⟨*proof*⟩

lemma *remove-term-keys*:
shows *keys* (*mapping-of p*) − {*m*} = *keys* (*mapping-of* (*remove-term m p*)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *remove-term-sum*: *remove-term m p + monom m (coeff p m) = p*
⟨*proof*⟩

lemma *mpoly-induct* [*case-names monom sum*]:
assumes *monom*:⋀*m a. P* (*monom m a*)
and *sum*:(⋀*p1 p2 m a. P p1* ⟹ *P p2* ⟹ *p2 = (monom m a)* ⟹ *m* ∉ *keys* (*mapping-of p1*) ⟹ *P* (*p1+p2*))
shows *P p* ⟨*proof*⟩

lemma *monom-pow*:*monom* (*PP-Poly-Mapping.single v n0*) *a* ^ *n* = *monom* (*PP-Poly-Mapping.single v* (*n0*∗*n*)) (*a* ^ *n*)
⟨*proof*⟩

lemma *insertion-fun-single*: *insertion-fun f* (λ*m*. (*a when* (*PP-Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) = *m*)) = *a* ∗ *f v* ^ *n* (**is** *?i* = -)
⟨*proof*⟩

lemma *insertion-single*[*simp*]: *insertion f* (*monom* (*PP-Poly-Mapping.single* (*v*::*nat*) (*n*::*nat*)) *a*) = *a* ∗ *f v* ^ *n*
⟨*proof*⟩

lemma *insertion-fun-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒ *'a*::*comm-ring-1*)
assumes ⋀*m v. p m* ≠ *0* ⟹ *lookup m v* ≠ *0* ⟹ *f v = g v*
shows *insertion-fun f p = insertion-fun g p*
⟨*proof*⟩

lemma *insertion-aux-irrelevant-vars*:
fixes *p*::((*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*::*comm-ring-1*)
assumes ⋀*m v. lookup p m* ≠ *0* ⟹ *lookup m v* ≠ *0* ⟹ *f v = g v*
shows *insertion-aux f p = insertion-aux g p*
⟨*proof*⟩

lemma *insertion-irrelevant-vars*:
fixes *p*::*'a*::*comm-ring-1 mpoly*
assumes ⋀*v. v*∈*vars p* ⟹ *f v = g v*
shows *insertion f p = insertion g p*
⟨*proof*⟩

32 Nested MPoly

definition *reduce-nested-mpoly*::*'a*::*comm-ring-1 mpoly mpoly* ⇒ *'a mpoly* **where**
reduce-nested-mpoly pp = insertion (λ*v. monom* (*PP-Poly-Mapping.single v 1*) *1*) *pp*

lemma *reduce-nested-mpoly-sum*:
fixes *p1*::*'a*::*comm-ring-1 mpoly mpoly*

shows *reduce-nested-mpoly* (*p1* + *p2*) = *reduce-nested-mpoly p1* + *reduce-nested-mpoly p2*
⟨*proof*⟩

lemma *reduce-nested-mpoly-prod*:
fixes *p1*::'*a*::*comm-ring-1 mpoly mpoly*
shows *reduce-nested-mpoly* (*p1* ∗ *p2*) = *reduce-nested-mpoly p1* ∗ *reduce-nested-mpoly p2*
⟨*proof*⟩

lemma *reduce-nested-mpoly-0*:
shows *reduce-nested-mpoly 0* = *0* ⟨*proof*⟩

lemma *insertion-nested-poly*:
fixes *pp*::'*a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*insertion* (λ*v*. *monom 0* (*f v*)) *pp*) = *insertion f* (*reduce-nested-mpoly pp*)
⟨*proof*⟩

definition *extract-var*::'*a*::*comm-ring-1 mpoly* ⇒ *nat* ⇒ '*a*::*comm-ring-1 mpoly mpoly* **where**
extract-var p v = (∑ *m*. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))

lemma *extract-var-finite-set*:
assumes {*m'*. *coeff p m'* ≠ *0*} ⊆ *S*
assumes *finite S*
shows *extract-var p v* = (∑ *m*∈*S*. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
⟨*proof*⟩

lemma *extract-var-non-zero-coeff*: *extract-var p v* = (∑ *m*∈{*m'*. *coeff p m'* ≠ *0*}. *monom* (*remove-key v m*) (*monom* (*PP-Poly-Mapping.single v* (*lookup m v*)) (*coeff p m*)))
⟨*proof*⟩

lemma *extract-var-sum*: *extract-var* (*p*+*p'*) *v* = *extract-var p v* + *extract-var p' v*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *extract-var-single'*:
assumes $v \neq v'$
shows *extract-var* (*monom* (*PP-Poly-Mapping.single v n*) *a*) v' = *monom* (*PP-Poly-Mapping.single v n*) (*monom 0 a*)
⟨*proof*⟩

lemma *reduce-nested-mpoly-extract-var*:
fixes *p*::'*a*::*comm-ring-1 mpoly*
shows *reduce-nested-mpoly* (*extract-var p v*) = *p*
⟨*proof*⟩

lemma *vars-extract-var-subset*: *vars* (*extract-var p v*) ⊆ *vars p*
⟨*proof*⟩

lemma *v-not-in-vars-extract-var*: *v* ∉ *vars* (*extract-var p v*)
⟨*proof*⟩

lemma *vars-coeff-extract-var*: *vars* (*coeff* (*extract-var p v*) *j*) ⊆ {*v*}
⟨*proof*⟩

definition *replace-coeff*
where *replace-coeff f p* = *MPoly* (*Abs-poly-mapping* (λ*m*. *f* (*lookup* (*mapping-of p*) *m*)))

lemma *coeff-replace-coeff*:
assumes *f 0* = *0*
shows *coeff* (*replace-coeff f p*) *m* = *f* (*coeff p m*)
⟨*proof*⟩

lemma *replace-coeff-monom*:
assumes *f 0* = *0*
shows *replace-coeff f* (*monom m a*) = *monom m* (*f a*)
⟨*proof*⟩

lemma *replace-coeff-add*:
assumes *f 0* = *0*
assumes ⋀*a b*. *f* (*a*+*b*) = *f a* + *f b*
shows *replace-coeff f* (*p1* + *p2*) = *replace-coeff f p1* + *replace-coeff f p2*
⟨*proof*⟩

lemma *insertion-replace-coeff*:

fixes *pp*::*'a*::*comm-ring-1 mpoly mpoly*
shows *insertion f* (*replace-coeff* (*insertion f*) *pp*) = *insertion f* (*reduce-nested-mpoly pp*)
⟨*proof*⟩

lemma *replace-coeff-extract-var-cong*:
assumes *f v* = *g v*
shows *replace-coeff* (*insertion f*) (*extract-var p v*) = *replace-coeff* (*insertion g*) (*extract-var p v*)
⟨*proof*⟩

lemma *vars-replace-coeff*:
assumes *f 0* = *0*
shows *vars* (*replace-coeff f p*) ⊆ *vars p*
⟨*proof*⟩

end

33 Polynomials representing the Deep Network Model

theory *DL-Deep-Model-Poly*
imports *DL-Deep-Model PP-More-MPoly Jordan-Normal-Form.Determinant*
begin

definition *polyfun N f* = (∃ *p*. *vars p* ⊆ *N* ∧ (∀ *x*. *insertion x p* = *f x*))

lemma *polyfunI*: (⋀*P*. (⋀*p*. *vars p* ⊆ *N* ⟹ (⋀*x*. *insertion x p* = *f x*) ⟹ *P*) ⟹ *P*) ⟹ *polyfun N f*
⟨*proof*⟩

lemma *polyfun-subset*: *N*⊆*N'* ⟹ *polyfun N f* ⟹ *polyfun N' f*
⟨*proof*⟩

lemma *polyfun-const*: *polyfun N* (λ-. *c*)
⟨*proof*⟩

lemma *polyfun-add*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* + *g x*)
⟨*proof*⟩

lemma *polyfun-mult*:
assumes *polyfun N f polyfun N g*
shows *polyfun N* (λ*x*. *f x* ∗ *g x*)
⟨*proof*⟩

lemma *polyfun-Sum*:
assumes *finite I*
assumes ⋀*i*. *i*∈*I* ⟹ *polyfun N* (*f i*)

shows *polyfun N* ($\lambda x.\ \sum i{\in}I.\ f\ i\ x$)
⟨*proof*⟩

lemma *polyfun-Prod*:
assumes *finite I*
assumes $\bigwedge i.\ i{\in}I \Longrightarrow$ *polyfun N* (*f i*)
shows *polyfun N* ($\lambda x.\ \prod i{\in}I.\ f\ i\ x$)
⟨*proof*⟩

lemma *polyfun-single*:
assumes $i{\in}N$
shows *polyfun N* ($\lambda x.\ x\ i$)
⟨*proof*⟩

lemma *polyfun-det*:
assumes $\bigwedge x.\ (A\ x) \in$ *carrier-mat n n*
assumes $\bigwedge x\ i\ j.\ i{<}n \Longrightarrow j{<}n \Longrightarrow$ *polyfun N* ($\lambda x.\ (A\ x)$ \$\$ (i,j))
shows *polyfun N* ($\lambda x.\ det\ (A\ x)$)
⟨*proof*⟩

lemma *polyfun-extract-matrix*:
assumes $i{<}m\ j{<}n$
shows *polyfun* $\{..{<}a + (m * n + c)\}$ ($\lambda f.$ *extract-matrix* ($\lambda i.\ f\ (i + a)$) *m n* \$\$ (i,j))
⟨*proof*⟩

lemma *polyfun-mult-mat-vec*:
assumes $\bigwedge x.\ v\ x \in$ *carrier-vec n*
assumes $\bigwedge j.\ j{<}n \Longrightarrow$ *polyfun N* ($\lambda x.\ v\ x$ \$ j)
assumes $\bigwedge x.\ A\ x \in$ *carrier-mat m n*
assumes $\bigwedge i\ j.\ i{<}m \Longrightarrow j{<}n \Longrightarrow$ *polyfun N* ($\lambda x.\ A\ x$ \$\$ (i,j))
assumes $j < m$
shows *polyfun N* ($\lambda x.\ ((A\ x) *_v (v\ x))$ \$ j)
⟨*proof*⟩

lemma *polyfun-evaluate-net-plus-a*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes $j <$ *output-size m*
shows *polyfun* $\{..{<}a +$ *count-weights m*$\}$ ($\lambda f.$ *evaluate-net* (*insert-weights m* ($\lambda i.$ *f* $(i + a)$)) *inputs* \$ j)
⟨*proof*⟩

lemma *polyfun-evaluate-net*:
assumes *map dim-vec inputs* = *input-sizes m*
assumes *valid-net m*
assumes $j <$ *output-size m*
shows *polyfun* $\{..{<}$*count-weights m*$\}$ ($\lambda f.$ *evaluate-net* (*insert-weights m f*) *inputs*

$ *j*)
⟨*proof*⟩

lemma *polyfun-tensors-from-net*:
assumes *valid-net m*
assumes *is* ◁ *input-sizes m*
assumes *j* < *output-size m*
shows *polyfun* {..<*count-weights m*} (λ*f*. *Tensor.lookup* (*tensors-from-net* (*insert-weights m f*) $ *j*) *is*)
⟨*proof*⟩

lemma *polyfun-matricize*:
assumes ⋀*x*. *dims* (*T x*) = *ds*
assumes ⋀*is*. *is* ◁ *ds* ⟹ *polyfun N* (λ*x*. *Tensor.lookup* (*T x*) *is*)
assumes ⋀*x*. *dim-row* (*matricize I* (*T x*)) = *nr*
assumes ⋀*x*. *dim-col* (*matricize I* (*T x*)) = *nc*
assumes *i* < *nr*
assumes *j* < *nc*
shows *polyfun N* (λ*x*. *matricize I* (*T x*) $$ (*i*,*j*))
⟨*proof*⟩

lemma (¬ (*a*::*nat*) < *b*) = (*a* ≥ *b*)
⟨*proof*⟩

lemma *polyfun-submatrix*:
assumes ⋀*x*. (*A x*) ∈ *carrier-mat m n*
assumes ⋀*x i j*. *i*<*m* ⟹ *j*<*n* ⟹ *polyfun N* (λ*x*. (*A x*) $$ (*i*,*j*))
assumes *i* < *card* {*i*. *i* < *m* ∧ *i* ∈ *I*}
assumes *j* < *card* {*j*. *j* < *n* ∧ *j* ∈ *J*}
assumes *infinite I infinite J*
shows *polyfun N* (λ*x*. (*submatrix* (*A x*) *I J*) $$ (*i*,*j*))
⟨*proof*⟩

context *deep-model-correct-params-y*
begin

definition *witness-submatrix* **where**
witness-submatrix f = *submatrix* (*A*′ *f*) *rows-with-1 rows-with-1*

lemma *polyfun-tensor-deep-model*:
assumes *is* ◁ *input-sizes* (*deep-model-l rs*)
shows *polyfun* {..<*weight-space-dim*}
(λ*f*. *Tensor.lookup* (*tensors-from-net* (*insert-weights* (*deep-model-l rs*) *f*) $ *y*) *is*)
⟨*proof*⟩

lemma *input-sizes-deep-model*: *input-sizes* (*deep-model-l rs*) = *replicate* (2 ∗ *N-half*) (*last rs*)
⟨*proof*⟩

lemma *polyfun-matrix-deep-model*:
assumes $i<(last\ rs)\ \hat{}\ N\text{-}half$
assumes $j<(last\ rs)\ \hat{}\ N\text{-}half$
shows *polyfun* $\{..<weight\text{-}space\text{-}dim\}\ (\lambda f.\ A'\ f\ \$\$\ (i,j))$
⟨*proof*⟩

lemma *polyfun-submatrix-deep-model*:
assumes $i < r\ \hat{}\ N\text{-}half$
assumes $j < r\ \hat{}\ N\text{-}half$
shows *polyfun* $\{..<weight\text{-}space\text{-}dim\}\ (\lambda f.\ witness\text{-}submatrix\ f\ \$\$\ (i,j))$
⟨*proof*⟩

lemma *polyfun-det-deep-model*:
shows *polyfun* $\{..<weight\text{-}space\text{-}dim\}\ (\lambda f.\ det\ (witness\text{-}submatrix\ f))$
⟨*proof*⟩

end

end

34 Alternative Lebesgue Measure Definition

theory *Lebesgue-Functional*
imports *HOL−Analysis.Lebesgue-Measure HOL.Topological-Spaces*
begin

Lebesgue_Measure.lborel is defined on the typeclass euclidean_space, which does not allow the space dimension to be dependent on a variable. As the Lebesgue measure of higher dimensions is the product measure of the one dimensional Lebesgue measure, we can easily define a more flexible version of the Lebesgue measure as follows. This version of the Lebesgue measure measures sets of functions from nat to real whose values are undefined for arguments higher than n. These "Extensional Function Spaces" are defined in HOL/Library/FuncSet.

definition *lborel-f* :: $nat \Rightarrow (nat \Rightarrow real)\ measure$ **where**
$lborel\text{-}f\ n = (\Pi_M\ b{\in}\{..<n\}.\ lborel)$

lemma *product-sigma-finite-interval*: *product-sigma-finite* $(\lambda b.\ interval\text{-}measure\ (\lambda x.\ x))$
⟨*proof*⟩

lemma *l-borel-f-1*: $distr\ (lborel\text{-}f\ 1)\ lborel\ (\lambda x.\ x\ 0) = lborel$
⟨*proof*⟩

lemma *space-lborel-f*: $space\ (lborel\text{-}f\ n) = Pi_E\ \{..<n\}\ (\lambda\text{-}.\ UNIV)$ ⟨*proof*⟩

lemma *space-lborel-f-subset*: $space\ (lborel\text{-}f\ n) \subseteq space\ (lborel\text{-}f\ (Suc\ n))$

⟨*proof*⟩

lemma *space-lborel-add-dim*:
assumes $f \in$ *space* (*lborel-f* n)
shows $f(n{:=}x) \in$ *space* (*lborel-f* (*Suc* n))
⟨*proof*⟩

lemma *integral-lborel-f*:
assumes $f \in$ *borel-measurable* (*lborel-f* (*Suc* n))
shows *integral*^{N} (*lborel-f* (*Suc* n)) $f = \int^+ y.\ \int^+ x.\ f\ (x(n := y))\ \partial$*lborel-f* n ∂*lborel*
⟨*proof*⟩

lemma *emeasure-lborel-f-Suc*:
assumes $A \in$ *sets* (*lborel-f* (*Suc* n))
assumes $\bigwedge y.\ \{x{\in}$*space* (*lborel-f* n). $x(n := y) \in A\} \in$ *sets* (*lborel-f* n)
shows *emeasure* (*lborel-f* (*Suc* n)) $A = \int^+ y.$ *emeasure* (*lborel-f* n) $\{x{\in}$*space* (*lborel-f* n). $x(n := y) \in A\}$ ∂*lborel*
⟨*proof*⟩

lemma *lborel-f-measurable-add-dim*: $(\lambda f.\ f(n := x)) \in$ *measurable* (*lborel-f* n) (*lborel-f* (*Suc* n))
⟨*proof*⟩

lemma *sets-lborel-f-sub-dim*:
assumes $A \in$ *sets* (*lborel-f* (*Suc* n))
shows $\{x.\ x(n := y) \in A\} \cap$ *space* (*lborel-f* n) $\in$ *sets* (*lborel-f* n)
⟨*proof*⟩

lemma *lborel-f-measurable-restrict*:
assumes $m \leq n$
shows $(\lambda f.$ *restrict* f $\{..{<}m\}) \in$ *measurable* (*lborel-f* n) (*lborel-f* m)
⟨*proof*⟩

lemma *lborel-measurable-sub-dim*: $(\lambda f.$ *restrict* f $\{..{<}n\}) \in$ *measurable* (*lborel-f* (*Suc* n)) (*lborel-f* n)
⟨*proof*⟩

lemma *measurable-lborel-component* [*measurable*]:
assumes $k{<}n$
shows $(\lambda x.\ x\ k) \in$ *borel-measurable* (*lborel-f* n)
⟨*proof*⟩

end

theory *PP-Univariate*
imports *PP-MPoly PP-More-MPoly HOL−Computational-Algebra.Polynomial*
begin

This file connects univariate MPolys to the theory of univariate polyno-

mials from `~~/src/HOL/Computational_Algebra/Polynomial.thy`.

definition *poly-to-mpoly*::*nat* ⇒ *'a*::*comm-monoid-add poly* ⇒ *'a mpoly*
where *poly-to-mpoly v p* = *MPoly* (*Abs-poly-mapping* (λ*m*. (*coeff p* (*PP-Poly-Mapping.lookup m v*)) *when PP-Poly-Mapping.keys m* ⊆ {*v*}))

lemma *poly-to-mpoly-finite*: *finite* {*m*::*nat* ⇒$_0$ *nat*. (*coeff p* (*PP-Poly-Mapping.lookup m v*) *when PP-Poly-Mapping.keys m* ⊆ {*v*}) ≠ *0*} (**is** *finite ?M*)
⟨*proof*⟩

lemma *coeff-poly-to-mpoly*: *PP-MPoly.coeff* (*poly-to-mpoly v p*) (*PP-Poly-Mapping.single v k*) = *Polynomial.coeff p k*
⟨*proof*⟩

definition *mpoly-to-poly*::*nat* ⇒ *'a*::*comm-monoid-add mpoly* ⇒ *'a poly*
where *mpoly-to-poly v p* = *Abs-poly* (λ*k*. *PP-MPoly.coeff p* (*PP-Poly-Mapping.single v k*))

lemma *coeff-mpoly-to-poly*[*simp*]: *Polynomial.coeff* (*mpoly-to-poly v p*) *k* = *PP-MPoly.coeff p* (*PP-Poly-Mapping.single v k*)
⟨*proof*⟩

lemma *mpoly-to-poly-inverse*:
assumes *vars p* ⊆ {*v*}
shows *poly-to-mpoly v* (*mpoly-to-poly v p*) = *p*
⟨*proof*⟩

lemma *poly-to-mpoly-inverse*: *mpoly-to-poly v* (*poly-to-mpoly v p*) = *p*
⟨*proof*⟩

lemma *poly-to-mpoly0*: *poly-to-mpoly v 0* = *0*
⟨*proof*⟩

lemma *mpoly-to-poly-add*: *mpoly-to-poly v* (*p1* + *p2*) = *mpoly-to-poly v p1* + *mpoly-to-poly v p2*
⟨*proof*⟩

lemma *poly-eq-insertion*:
assumes *vars p* ⊆ {*v*}
shows *poly* (*mpoly-to-poly v p*) *x* = *insertion* (λ*v*. *x*) *p*
⟨*proof*⟩

Using the new connection between MPoly and univariate polynomials, we can transfer:

lemma *univariate-mpoly-roots-finite*:
fixes *p*::*'a*::*idom mpoly*
assumes *vars p* ⊆ {*v*} *p* ≠ *0*
shows *finite* {*x*. *insertion* (λ*v*. *x*) *p* = *0*}
⟨*proof*⟩

end

35 Lebesgue Measure of Polynomial Zero Sets

theory *Lebesgue-Zero-Set*
imports *PP-MPoly PP-More-MPoly Lebesgue-Functional PP-Univariate*
begin

lemma *measurable-insertion* [*measurable*]:
assumes *vars p* ⊆ {..<*n*}
shows (λ*f*. *insertion f p*) ∈ *borel-measurable* (*lborel-f n*)
⟨*proof*⟩

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*)
⟨*proof*⟩

end

36 Rank and Submatrices

theory *DL-Rank-Submatrix*
imports *DL-Rank DL-Submatrix DL-Missing-Matrix*
begin

lemma *row-submatrix-UNIV*:
assumes *i* < *card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*}
shows *row* (*submatrix A I UNIV*) *i* = *row A* (*pick I i*)
⟨*proof*⟩

lemma *distinct-cols-submatrix-UNIV*:
assumes *distinct* (*cols* (*submatrix A I UNIV*))
shows *distinct* (*cols A*)
⟨*proof*⟩

lemma *cols-submatrix-subset*: *set* (*cols* (*submatrix A UNIV J*)) ⊆ *set* (*cols A*)
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-dep-submatrix-UNIV*:
assumes *A* ∈ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols* (*submatrix A I UNIV*))

shows *LinearCombinations.module.lin-dep class-ring* (*module-vec TYPE*('*a*) (*card* {*i*. *i* < *n* ∧ *i* ∈ *I*})) (*set* (*cols* (*submatrix A I UNIV*)))
(**is** *LinearCombinations.module.lin-dep class-ring ?M* (*set ?S'*))
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-gt-minor*:
assumes *A* ∈ *carrier-mat n nc*
assumes *det* (*submatrix A I J*) ≠ *0*
shows *card* {*j*. *j* < *nc* ∧ *j* ∈ *J*} ≤ *rank A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *output-size-shallow-model'*: *output-size* (*shallow-model' Z M N*) = *Z*
⟨*proof*⟩

lemma *valid-shallow-model*: *valid-net* (*shallow-model Y Z M N*)
⟨*proof*⟩

lemma *output-size-shallow-model*: *output-size* (*shallow-model Y Z M N*) = *Y*
⟨*proof*⟩

lemma *input-sizes-shallow-model*: *input-sizes* (*shallow-model Y Z M N*) = *replicate* (*Suc N*) *M*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

end

38 Fundamental Theorem of Network Capacity

theory *DL-Fundamental-Theorem-Network-Capacity*
imports *DL-Rank-CP-Rank DL-Deep-Model-Poly Lebesgue-Zero-Set DL-Rank-Submatrix HOL−Analysis.Complete-Measure DL-Shallow-Model*
begin

context *deep-model-correct-params-y*
begin

definition *polynomial-f w = det (submatrix (matricize {n. even n} (A w)) rows-with-1 rows-with-1)*

lemma *polyfun-polynomial*:
shows *polyfun {..<weight-space-dim} polynomial-f*
⟨*proof*⟩

definition *polynomial-p = (SOME p. vars p ⊆ {..<weight-space-dim} ∧ (∀ x. insertion x p = polynomial-f x))*

lemma
polynomial-p-not-0: *polynomial-p≠0* **and**
vars-polynomial-p: *vars polynomial-p ⊆ {..<weight-space-dim}* **and**
polynomial-pf: ⋀*w. insertion w polynomial-p = polynomial-f w*
⟨*proof*⟩

lemma *if-polynomial-0-rank*:
assumes *polynomial-f w ≠ 0*
shows *r ^ N-half ≤ cprank (A w)*
⟨*proof*⟩

lemma *if-polynomial-0-evaluate*:
assumes *polynomial-f wd ≠ 0*
assumes ∀ *inputs. input-sizes (deep-model-l rs) = map dim-vec inputs ⟶ evaluate-net (insert-weights (deep-model-l rs) wd) inputs*
= evaluate-net (insert-weights (shallow-model (rs ! 0) Z (last rs) (2∗N-half−1)) ws) inputs
shows *Z ≥ r ^ N-half*

⟨*proof*⟩

lemma *if-polynomial-0-evaluate-notex*:
assumes *polynomial-f wd* ≠ *0*
shows ¬(∃ *weights-shallow Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
evaluate-net (*insert-weights* (*deep-model-l rs*) *wd*) *inputs*
= *evaluate-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*N-half*−*1*)) *ws*) *inputs*))
⟨*proof*⟩

theorem *fundamental-theorem-network-capacity*:
AE x in lborel-f weight-space-dim. *r* ^ *N-half* ≤ *cprank* (*A x*)
⟨*proof*⟩

theorem *fundamental-theorem-network-capacity-v2*:
shows *AE wd in lborel-f weight-space-dim*.
¬(∃ *ws Z*. *Z* < *r* ^ *N-half* ∧ (∀ *inputs*. *input-sizes* (*deep-model-l rs*) = *map dim-vec inputs* ⟶
evaluate-net (*insert-weights* (*deep-model-l rs*) *wd*) *inputs*
= *evaluate-net* (*insert-weights* (*shallow-model* (*rs* ! *0*) *Z* (*last rs*) (*2*N-half*−*1*)) *ws*) *inputs*))
⟨*proof*⟩

end
end

References

[1] A. Bentkamp. An Isabelle Formalization of the Expressiveness of Deep Learning. Master's thesis, Universität des Saarlandes, 2016. http://matryoshka.gforge.inria.fr/bentkamp_msc_thesis.pdf.

[2] N. Cohen, O. Sharir, and A. Shashua. On the expressive power of deep learning: A tensor analysis. In V. Feldman, A. Rakhlin, and O. Shamir, editors, *Conference on Learning Theory (COLT 2016)*, volume 49 of *JMLR Workshop and Conference Proceedings*, pages 698–728. JMLR.org, 2016.