Executable Matrix Operations on Matrices of Arbitrary Dimensions

Christian Sternagel and René Thiemann

December 14, 2021

Abstract

We provide the operations of matrix addition, multiplication, transposition, and matrix comparisons as executable functions over ordered semirings. Moreover, it is proven that strongly normalizing (monotone) orders can be lifted to strongly normalizing (monotone) orders over matrices.

We further show that the standard semirings over the naturals, integers, and rationals, as well as the arctic semirings satisfy the axioms that are required by our matrix theory.

Our formalization was performed as part of the IsaFoR/CeTA-system [3][1] which contains several termination techniques. The provided theories have been essential to formalize matrix-interpretations [1] and arctic interpretations [2]. A short description of this formalization can be found in [4].

Contents

1 Utility Functions and Lemmas **2**
- 1.1 Miscellaneous . . . 2
- 1.2 A connection between class based semirings and set based semirings . . . 8

2 Basic Operations on Matrices **9**
- 2.1 types and well-formedness of vectors / matrices . . . 9
- 2.2 definitions / algorithms . . . 10
- 2.3 algorithms preserve dimensions . . . 11
- 2.4 properties of algorithms which do not depend on properties of type of matrix . . . 15
- 2.5 lemmas requiring properties of plus, times, ... 24
- 2.6 Connection to HOL-Algebra . . . 34

[1] http://cl-informatik.uibk.ac.at/software/ceta

1 Utility Functions and Lemmas

theory *Utility*
imports *Main*
begin

1.1 Miscellaneous

lemma *ballI2*[*Pure.intro*]:
  assumes $\bigwedge x\ y.\ (x,\ y) \in A \Longrightarrow P\ x\ y$
  shows $\forall\,(x,\ y){\in}A.\ P\ x\ y$
  using *assms* **by** *auto*

lemma *infinite-imp-elem*: $\neg$ *finite* $A \Longrightarrow \exists\ x.\ x \in A$
  by (*cases* $A = \{\}$, *auto*)

lemma *infinite-imp-many-elems*:
  infinite $A \Longrightarrow \exists\ xs.\ set\ xs \subseteq A \wedge length\ xs = n \wedge distinct\ xs$
proof (*induct n arbitrary*: A)
  case (*Suc n*)
  from *infinite-imp-elem*[*OF Suc*(*2*)] **obtain** x **where** x: $x \in A$ **by** *auto*
  from *Suc*(*2*) **have** *infinite* $(A - \{x\})$ **by** *auto*
  from *Suc*(*1*)[*OF this*] **obtain** *xs* **where** *set xs* $\subseteq A - \{x\}$ **and** *length xs* $= n$
and *distinct xs* **by** *auto*
  with x **show** *?case* **by** (*intro exI*[*of - x # xs*], *auto*)
qed *auto*

lemma *inf-pigeonhole-principle*:
  assumes $\forall\, k{::}nat.\ \exists\, i{<}n{::}nat.\ f\ k\ i$
  shows $\exists\, i{<}n.\ \forall\, k.\ \exists\, k'{\geq}k.\ f\ k'\ i$
proof −
  have *nfin*: ~ *finite* (*UNIV* :: *nat set*) **by** *auto*
  have *fin*: *finite* $(\{i.\ i < n\})$ **by** *auto*
  from *pigeonhole-infinite-rel*[*OF nfin fin*] *assms*
  obtain i **where** i: $i < n$ **and** *nfin*: $\neg$ *finite* $\{a.\ f\ a\ i\}$ **by** *auto*
  show *?thesis*
  proof (*intro exI conjI*, *rule i*, *intro allI*)
    fix k
    have *finite* $\{a.\ f\ a\ i \wedge a < k\}$ **by** *auto*
    with *nfin* **have** $\neg$ *finite* $(\{a.\ f\ a\ i\} - \{a.\ f\ a\ i \wedge a < k\})$ **by** *auto*
    from *infinite-imp-elem*[*OF this*]
    obtain a **where** $f\ a\ i$ **and** $a \geq k$ **by** *auto*
    thus $\exists\ k' \geq k.\ f\ k'\ i$ **by** *force*
  qed
qed

lemma *map-upt-Suc*: *map* f $[0\ ..{<}\ Suc\ n] = f\ 0\ \#\ map\ (\lambda\ i.\ f\ (Suc\ i))\ [0\ ..{<}\ n]$
  by (*induct n arbitrary*: f, *auto*)

lemma *map-upt-add*: *map f [0 ..< n + m] = map f [0 ..< n] @ map (λ i. f (i + n)) [0 ..< m]*
proof (*induct n arbitrary*: *f*)
  case (*Suc n f*)
  have *map f [0 ..< Suc n + m] = map f [0 ..< Suc (n+m)]* **by** *simp*
  also have *... = f 0 # map (λ i. f (Suc i)) [0 ..< n + m]* **unfolding** *map-upt-Suc*
..
  finally show *?case* **unfolding** *Suc map-upt-Suc* **by** *simp*
qed *simp*

lemma *map-upt-split*: **assumes** *i*: *i < n*
  shows *map f [0 ..< n] = map f [0 ..< i] @ f i # map (λ j. f (j + Suc i)) [0 ..< n − Suc i]*
proof −
  from *i* **have** *n = i + Suc 0 + (n − Suc i)* **by** *arith*
  hence *id*: *[0 ..< n] = [0 ..< i + Suc 0 + (n − Suc i)]* **by** *simp*
  show *?thesis* **unfolding** *id*
    unfolding *map-upt-add* **by** *auto*
qed

lemma *all-Suc-conv*:
  (∀ i<Suc n. P i) ⟷ P 0 ∧ (∀ i<n. P (Suc i)) (**is** *?l = ?r*)
proof
  assume *?l* **thus** *?r* **by** *auto*
next
  assume *?r* **show** *?l*
  proof (*intro allI impI*)
    fix *i*
    assume *i < Suc n*
    with ‹*?r*› **show** *P i* **by** (*cases i*, *auto*)
  qed
qed

lemma *ex-Suc-conv*:
  (∃ i<Suc n. P i) ⟷ P 0 ∨ (∃ i<n. P (Suc i)) (**is** *?l = ?r*)
  using *all-Suc-conv*[*of n λi. ¬ P i*] **by** *blast*

fun *sorted-list-subset* :: *'a :: linorder list ⇒ 'a list ⇒ 'a option* **where**
  sorted-list-subset (a # as) (b # bs) =
    (*if a = b then sorted-list-subset as (b # bs)*
     else if a > b then sorted-list-subset (a # as) bs
     else Some a)
| *sorted-list-subset [] - = None*
| *sorted-list-subset (a # -) [] = Some a*

lemma *sorted-list-subset*:
  assumes *sorted as* **and** *sorted bs*
  shows (*sorted-list-subset as bs = None*) = (*set as ⊆ set bs*)

using *assms*
proof (*induct rule*: *sorted-list-subset.induct*)
  case (*2 bs*)
  thus *?case* **by** *auto*
next
  case (*3 a as*)
  thus *?case* **by** *auto*
next
  case (*1 a as b bs*)
  from *1*(*3*) **have** *sas*: *sorted as* **and** *a*: $\bigwedge a'.\ a' \in set\ as \Longrightarrow a \leq a'$ **by** (*auto*)
  from *1*(*4*) **have** *sbs*: *sorted bs* **and** *b*: $\bigwedge b'.\ b' \in set\ bs \Longrightarrow b \leq b'$ **by** (*auto*)
  show *?case*
  proof (*cases a* = *b*)
    case *True*
    from *1*(*1*)[*OF this sas 1*(*4*)] *True* **show** *?thesis* **by** *auto*
  next
    case *False* **note** *oFalse* = *this*
    show *?thesis*
    proof (*cases a* > *b*)
      case *True*
      with *a b* **have** $b \notin set\ as$ **by** *force*
      with *1*(*2*)[*OF False True 1*(*3*) *sbs*] *False True* **show** *?thesis* **by** *auto*
    next
      case *False*
      with *oFalse* **have** *a* < *b* **by** *auto*
      with *a b* **have** $a \notin set\ bs$ **by** *force*
      with *oFalse False* **show** *?thesis* **by** *auto*
    qed
  qed
qed

lemma *zip-nth-conv*: *length xs* = *length ys* $\Longrightarrow$ *zip xs ys* = *map* (λ *i*. (*xs* ! *i*, *ys* ! *i*)) [*0* ..< *length ys*]
proof (*induct xs arbitrary*: *ys*, *simp*)
  case (*Cons x xs*)
  then obtain *y yys* **where** *ys*: *ys* = *y* # *yys* **by** (*cases ys*, *auto*)
  with *Cons* **have** *len*: *length xs* = *length yys* **by** *simp*
  show *?case* **unfolding** *ys*
    by (*simp del*: *upt-Suc add*: *map-upt-Suc*, *unfold Cons*(*1*)[*OF len*], *simp*)
qed

lemma *nth-map-conv*:
  assumes *length xs* = *length ys*
    and $\forall i{<}length\ xs.\ f\ (xs\ !\ i) = g\ (ys\ !\ i)$
  shows *map f xs* = *map g ys*
using *assms*
proof (*induct xs arbitrary*: *ys*)
  case (*Cons x xs*) **thus** *?case*
  proof (*induct ys*)

```
    case (Cons y ys)
    have ∀ i<length xs. f (xs ! i) = g (ys ! i)
    proof (intro allI impI)
      fix i assume i < length xs thus f (xs ! i) = g (ys ! i) using Cons(4) by
force
    qed
    with Cons show ?case by auto
  qed simp
qed simp

lemma sum-list-0: ⟦⋀ x. x ∈ set xs ⟹ x = 0⟧ ⟹ sum-list xs = 0
  by (induct xs, auto)

lemma foldr-foldr-concat: foldr (foldr f) m a = foldr f (concat m) a
proof (induct m arbitrary: a)
  case Nil show ?case by simp
next
  case (Cons v m a)
  show ?case
    unfolding concat.simps foldr-Cons o-def Cons
    unfolding foldr-append by simp
qed

lemma sum-list-double-concat:
  fixes f :: 'b ⇒ 'c ⇒ 'a :: comm-monoid-add and g as bs
  shows sum-list (concat (map (λ i. map (λ j. f i j + g i j) as) bs))
      = sum-list (concat (map (λ i. map (λ j. f i j) as) bs)) +
        sum-list (concat (map (λ i. map (λ j. g i j) as) bs))
proof (induct bs)
  case Nil thus ?case by simp
next
  case (Cons b bs)
  have id: (∑j←as. f b j + g b j) = sum-list (map (f b) as) + sum-list (map (g
b) as)
    by (induct as, auto simp: ac-simps)
  show ?case unfolding list.map concat.simps sum-list-append
    unfolding Cons
    unfolding id
    by (simp add: ac-simps)
qed

fun max-list :: nat list ⇒ nat where
  max-list [] = 0
| max-list (x # xs) = max x (max-list xs)

lemma max-list: x ∈ set xs ⟹ x ≤ max-list xs
  by (induct xs) auto

lemma max-list-mem: xs ≠ [] ⟹ max-list xs ∈ set xs
```

```
proof (induct xs)
  case (Cons x xs)
  show ?case
  proof (cases x ≥ max-list xs)
    case True
    thus ?thesis by auto
  next
    case False
    hence max: max-list xs > x by auto
    hence nil: xs ≠ [] by (cases xs, auto)
    from max have max: max x (max-list xs) = max-list xs by auto
    from Cons(1)[OF nil] max show ?thesis by auto
  qed
qed simp

lemma max-list-set: max-list xs = (if set xs = {} then 0 else (THE x. x ∈ set xs
∧ (∀ y ∈ set xs. y ≤ x)))
proof (cases xs = [])
  case True thus ?thesis by simp
next
  case False
  note p = max-list-mem[OF this] max-list[of - xs]
  from False have id: (set xs = {}) = False by simp
  show ?thesis unfolding id if-False
  proof (rule the-equality[symmetric], intro conjI ballI, rule p, rule p)
    fix x
    assume x ∈ set xs ∧ (∀ y ∈ set xs. y ≤ x)
    hence mem: x ∈ set xs and le: ⋀ y. y ∈ set xs ⟹ y ≤ x by auto
    from max-list[OF mem] le[OF max-list-mem[OF False]]
    show x = max-list xs by simp
  qed
qed

lemma max-list-eq-set: set xs = set ys ⟹ max-list xs = max-list ys
  unfolding max-list-set by simp

lemma all-less-two: (∀ i < Suc (Suc 0). P i) = (P 0 ∧ P (Suc 0)) (is ?l = ?r)
proof
  assume ?r
  show ?l
  proof(intro allI impI)
    fix i
    assume i < Suc (Suc 0)
    hence i = 0 ∨ i = Suc 0 by auto
    with ‹?r› show P i by auto
  qed
qed auto
```

Induction over a finite set of natural numbers.

lemma *bound-nat-induct*[*consumes 1*]:
assumes $n \in \{l..u\}$ **and** $P\ l$ **and** $\bigwedge n.\ [\![P\ n;\ n \in \{l..<u\}]\!] \Longrightarrow P\ (Suc\ n)$
shows $P\ n$
using *assms*
proof (*induct n*)
case (*Suc n*) **thus** *?case* **by** (*cases Suc n = l*) *auto*
qed *simp*

end

theory *Ordered-Semiring*
imports
HOL−Algebra.Ring
Abstract−Rewriting.SN-Orders
begin

record $'a$ *ordered-semiring* = $'a$ *ring* +
geq :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\succeq_1$ *50*)
gt :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\succ_1$ *50*)
max :: $'a \Rightarrow 'a \Rightarrow 'a$ (Max_1)

lemmas *ordered-semiring-record-simps* = *ring-record-simps ordered-semiring.simps*

locale *ordered-semiring* = *semiring* +
assumes *compat*: $[\![s \succeq (t :: 'a);\ t \succ u;\ s \in carrier\ R;\ t \in carrier\ R;\ u \in carrier\ R]\!] \Longrightarrow s \succ u$
and *compat2*: $[\![s \succ (t :: 'a);\ t \succeq u;\ s \in carrier\ R\ ;\ t \in carrier\ R;\ u \in carrier\ R]\!] \Longrightarrow s \succ u$
and *plus-left-mono*: $[\![x \succeq y;\ x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \oplus z \succeq y \oplus z$
and *times-left-mono*: $[\![z \succeq \mathbf{0};\ x \succeq y;\ x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \otimes z \succeq y \otimes z$
and *times-right-mono*: $[\![x \succeq \mathbf{0};\ y \succeq z;\ x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \otimes y \succeq x \otimes z$
and *geq-refl*: $x \in carrier\ R \Longrightarrow x \succeq x$
and *geq-trans*[*trans*]: $[\![x \succeq y;\ y \succeq z;\ x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \succeq z$
and *gt-trans*[*trans*]: $[\![x \succ y;\ y \succ z;\ x \in carrier\ R;\ y \in carrier\ R;\ z \in carrier\ R]\!] \Longrightarrow x \succ z$
and *gt-imp-ge*: $x \succ y \Longrightarrow x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow x \succeq y$
and *max-comm*: $x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow Max\ x\ y = Max\ y\ x$
and *max-ge*: $x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow Max\ x\ y \succeq x$
and *max-id*: $x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow x \succeq y \Longrightarrow Max\ x\ y = x$
and *max-mono*: $x \succeq y \Longrightarrow x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow z \in carrier\ R \Longrightarrow Max\ z\ x \succeq Max\ z\ y$
and *wf-max*[*simp*, *intro*]: $x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow Max\ x\ y \in carrier\ R$

and *one-geq-zero*: $\mathbf{1} \succeq \mathbf{0}$
begin
lemma *max-ge-right*: **assumes** *x*: $x \in$ *carrier R* **and** *y*: $y \in$ *carrier R* **shows** *Max x y* $\succeq y$
by (*unfold max-comm*[*OF x y*], *rule max-ge*[*OF y x*])

lemma *wf-max0*: $x \in$ *carrier R* $\Longrightarrow$ *Max* $\mathbf{0}$ $x \in$ *carrier R* **using** *wf-max*[*of* $\mathbf{0}$ *x*]
by *auto*

lemma *max0-id-pos*: **assumes** *x*: $x \succeq \mathbf{0}$ **and** *wf*: $x \in$ *carrier R*
shows *Max* $\mathbf{0}$ $x = x$ **unfolding** *max-comm*[*OF zero-closed wf*] **by** (*rule max-id*[*OF wf zero-closed x*])
end
hide-const (**open**) *gt geq max*

1.2 A connection between class based semirings and set based semirings

definition *class-semiring* :: *'a itself* ⇒ *'b* ⇒ (*'a* :: {*plus*,*times*,*one*,*zero*},*'b*)*ring-scheme*
where
class-semiring - b ≡ (| *carrier* = *UNIV*, *mult* = (∗), *one* = *1*, *zero* = *0*, *add* = (+), ... = *b*|)

lemma *class-semiring*: *semiring* (*class-semiring* (*TYPE*(*'a* :: *ordered-semiring-1*)) *b*)
unfolding *class-semiring-def*
by (*unfold-locales*, *auto simp*: *field-simps*)

definition *class-ordered-semiring* :: *'a itself* ⇒ (*'a* :: *ordered-semiring-1* ⇒ *'a* ⇒ *bool*) ⇒ *'b* ⇒ (*'a*,*'b*) *ordered-semiring-scheme* **where**
class-ordered-semiring a gt b ≡ *class-semiring a* (|
ordered-semiring.geq = (≥),
gt = *gt*,
max = *max*,
... = *b*|)

lemma *class-ordered-semiring*: **assumes** *order-pair* (*gt* :: (*'a* :: *ordered-semiring-1* ⇒ *'a* ⇒ *bool*)) *d*
shows *ordered-semiring*
(*class-ordered-semiring* (*TYPE*(*'a*)) *gt b*)
(**is** *ordered-semiring ?R*)
proof −
interpret *order-pair gt d* **by** *fact*
interpret *semiring ?R* **unfolding** *class-ordered-semiring-def* **by** (*rule class-semiring*)
show *?thesis*
by (*unfold-locales*, *unfold class-ordered-semiring-def class-semiring-def*, *auto*
intro: *compat compat2 gt-imp-ge ge-trans max-comm max-id max-mono ge-refl one-ge-zero*
times-left-mono times-right-mono plus-left-mono)

qed

lemma (**in** *one-mono-ordered-semiring-1*) *class-ordered-semiring*:
ordered-semiring
(*class-ordered-semiring* (*TYPE*('*a*)) (≻) *b*)
by (*rule class-ordered-semiring*[*of - default*], *unfold-locales*)

lemma (**in** *both-mono-ordered-semiring-1*) *class-ordered-semiring*:
ordered-semiring
(*class-ordered-semiring* (*TYPE*('*a*)) (≻) *b*)
by (*rule class-ordered-semiring*[*of - default*], *unfold-locales*)

end

2 Basic Operations on Matrices

theory *Matrix-Legacy*
imports
Utility
Ordered-Semiring
begin

This theory is marked as legacy, since there is a better implementation of matrices available in `../Jordan_Normal_Form/Matrix.thy`. That formalization is more abstract, more complete in terms of operations, and it still provides an efficient implementation.

This theory provides the operations of matrix addition, multiplication, and transposition as executable functions. Most properties are proven via pointwise equality of matrices.

2.1 types and well-formedness of vectors / matrices

type-synonym '*a vec* = '*a list*
type-synonym '*a mat* = '*a vec list*

definition *vec* :: *nat* ⇒ '*x vec* ⇒ *bool*
where *vec n x* = (*length x* = *n*)

definition *mat* :: *nat* ⇒ *nat* ⇒ '*a mat* ⇒ *bool* **where**
mat nr nc m = (*length m* = *nc* ∧ *Ball* (*set m*) (*vec nr*))

2.2 definitions / algorithms

note that these algorithms are generic in all basic definitions / operations like 0 (ze) 1 (on) addition (pl) multiplication (ti) and in the dimension(s) of the matrix/vector. Hence, many of these algorithms require these definitions/operations/sizes as arguments. All indices start from 0.

definition *vec0I* :: *'a* ⇒ *nat* ⇒ *'a vec* **where**
vec0I ze n = *replicate n ze*

definition *mat0I* :: *'a* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* **where**
mat0I ze nr nc = *replicate nc* (*vec0I ze nr*)

definition *vec1I* :: *'a* ⇒ *'a* ⇒ *nat* ⇒ *nat* ⇒ *'a vec*
where *vec1I ze on n i* ≡ *replicate i ze* @ *on* # *replicate* (*n* − *1* − *i*) *ze*

definition *mat1I* :: *'a* ⇒ *'a* ⇒ *nat* ⇒ *'a mat*
where *mat1I ze on n* ≡ *map* (*vec1I ze on n*) [*0* ..< *n*]

definition *vec-plusI* :: (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a vec* ⇒ *'a vec* ⇒ *'a vec* **where**
vec-plusI pl v w = *map* (λ *xy*. *pl* (*fst xy*) (*snd xy*)) (*zip v w*)

definition *mat-plusI* :: (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a mat* ⇒ *'a mat* ⇒ *'a mat*
where *mat-plusI pl m1 m2* = *map* (λ *uv*. *vec-plusI pl* (*fst uv*) (*snd uv*)) (*zip m1 m2*)

definition *scalar-prodI* :: *'a* ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a vec* ⇒ *'a vec* ⇒ *'a* **where**
scalar-prodI ze pl ti v w = *foldr* (λ (*x*,*y*) *s*. *pl* (*ti x y*) *s*) (*zip v w*) *ze*

definition *row* :: *'a mat* ⇒ *nat* ⇒ *'a vec*
where *row m i* ≡ *map* (λ *w*. *w* ! *i*) *m*

definition *col* :: *'a mat* ⇒ *nat* ⇒ *'a vec*
where *col m i* ≡ *m* ! *i*

fun *transpose* :: *nat* ⇒ *'a mat* ⇒ *'a mat*
where *transpose nr* [] = *replicate nr* []
| *transpose nr* (*v* # *m*) = *map* (λ (*vi*,*mi*). (*vi* # *mi*)) (*zip v* (*transpose nr m*))

definition *matT-vec-multI* :: $'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ mat \Rightarrow 'a\ vec \Rightarrow 'a\ vec$
where *matT-vec-multI ze pl ti m v* = *map* (λ *w. scalar-prodI ze pl ti w v*) *m*

definition *mat-multI* :: $'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$
where *mat-multI ze pl ti nr m1 m2* $\equiv$ *map* (*matT-vec-multI ze pl ti* (*transpose nr m1*)) *m2*

fun *mat-powI* :: $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow nat \Rightarrow 'a\ mat$
where *mat-powI ze on pl ti n m 0* = *mat1I ze on n*
| *mat-powI ze on pl ti n m* (*Suc i*) = *mat-multI ze pl ti n* (*mat-powI ze on pl ti n m i*) *m*

definition *sub-vec* :: $nat \Rightarrow 'a\ vec \Rightarrow 'a\ vec$
where *sub-vec* = *take*

definition *sub-mat* :: $nat \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$
where *sub-mat nr nc m* = *map* (*sub-vec nr*) (*take nc m*)

definition *vec-map* :: $('a \Rightarrow 'a) \Rightarrow 'a\ vec \Rightarrow 'a\ vec$
where *vec-map* = *map*

definition *mat-map* :: $('a \Rightarrow 'a) \Rightarrow 'a\ mat \Rightarrow 'a\ mat$
where *mat-map f* = *map* (*vec-map f*)

2.3 algorithms preserve dimensions

lemma *vec0*[*simp*,*intro*]: *vec nr* (*vec0I ze nr*)
by (*simp add*: *vec-def vec0I-def*)

lemma *replicate-prop*:
assumes *P x*
shows $\forall\, y \in set$ (*replicate n x*). *P y*
using *assms* **by** (*induct n*) *simp-all*

lemma *mat0*[*simp*,*intro*]: *mat nr nc* (*mat0I ze nr nc*)
unfolding *mat-def mat0I-def*
using *replicate-prop*[*of vec nr vec0I ze nr nc*] **by** *simp*

lemma *vec1*[*simp*,*intro*]: **assumes** $i < nr$ **shows** *vec nr* (*vec1I ze on nr i*)

unfolding *vec-def vec1I-def* **using** *assms* **by** *auto*

lemma *mat1*[*simp*,*intro*]: *mat nr nr* (*mat1I ze on nr*)
unfolding *mat-def mat1I-def* **using** *vec1* **by** *auto*

lemma *vec-plus*[*simp*,*intro*]: ⟦*vec nr u*; *vec nr v*⟧ ⟹ *vec nr* (*vec-plusI pl u v*)
unfolding *vec-plusI-def vec-def*
by *auto*

lemma *mat-plus*[*simp*,*intro*]: **assumes** *mat nr nc m1* **and** *mat nr nc m2* **shows** *mat nr nc* (*mat-plusI pl m1 m2*)
using *assms*
unfolding *mat-def mat-plusI-def*
proof (*simp*, *induct nc arbitrary*: *m1 m2*, *simp*)
  case (*Suc nn*)
  show *?case*
  proof (*cases m1*)
    case *Nil* **with** *Suc* **show** *?thesis* **by** *auto*
  next
    case (*Cons v1 mm1*) **note** *oCons* = *this*
    with *Suc* **have** *l1*: *length mm1* = *nn* **by** *auto*
    show *?thesis*
    proof (*cases m2*)
      case *Nil* **with** *Suc* **show** *?thesis* **by** *auto*
    next
      case (*Cons v2 mm2*)
      with *Suc* **have** *l2*: *length mm2* = *nn* **by** *auto*
     show *?thesis* **by** (*simp add*: *Cons oCons*, *intro conjI*[*OF vec-plus*], (*simp add*: *Cons oCons Suc*)+, *rule Suc*, *auto simp*: *Cons oCons Suc l1 l2*)
    qed
  qed
qed

lemma *vec-map*[*simp*,*intro*]: *vec nr u* ⟹ *vec nr* (*vec-map f u*)
unfolding *vec-map-def vec-def*
by *auto*

lemma *mat-map*[*simp*,*intro*]: *mat nr nc m* ⟹ *mat nr nc* (*mat-map f m*)
using *vec-map*
unfolding *mat-map-def mat-def*
by *auto*

fun *vec-fold* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a vec* ⇒ '*b* ⇒ '*b*
  where [*code-unfold*]: *vec-fold f* = *foldr f*

fun *mat-fold* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a mat* ⇒ '*b* ⇒ '*b*
  where [*code-unfold*]: *mat-fold f* = *foldr* (*vec-fold f*)

lemma *concat-mat*: *mat nr nc m* $\Longrightarrow$
concat m = [*m* ! *i* ! *j*. *i* ← [*0* ..< *nc*], *j* ← [*0* ..< *nr*]]
proof (*induct m arbitrary*: *nc*)
case *Nil*
thus *?case* **unfolding** *mat-def* **by** *auto*
next
case (*Cons v m snc*)
from *Cons*(*2*) **obtain** *nc* **where** *snc*: *snc* = *Suc nc* **and** *mat*: *mat nr nc m* **and** *v*: *vec nr v*
unfolding *mat-def* **by** (*cases snc*, *auto*)
from *v* **have** *nr*: *nr* = *length v* **unfolding** *vec-def* **by** *auto*
have *v*: *map* (λ *i*. *v* ! *i*) [*0* ..< *nr*] = *v* **unfolding** *nr map-nth* **by** *simp*
note *IH* = *Cons*(*1*)[*OF mat*]
show *?case*
unfolding *snc*
unfolding *map-upt-Suc*
unfolding *nth.simps nat.simps concat.simps*
unfolding *IH v* **..**
qed

lemma *row*: **assumes** *mat nr nc m*
and *i* < *nr*
shows *vec nc* (*row m i*)
using *assms*
unfolding *vec-def row-def mat-def*
by (*auto simp*: *vec-def*)

lemma *col*: **assumes** *mat nr nc m*
and *i* < *nc*
shows *vec nr* (*col m i*)
using *assms*
unfolding *vec-def col-def mat-def*
by (*auto simp*: *vec-def*)

lemma *transpose*[*simp*,*intro*]: **assumes** *mat nr nc m*
shows *mat nc nr* (*transpose nr m*)
using *assms*
proof (*induct m arbitrary*: *nc*)
case (*Cons v m*)
from ‹*mat nr nc* (*v* # *m*)› **obtain** *ncc* **where** *nc*: *nc* = *Suc ncc* **by** (*cases nc*, *auto simp*: *mat-def*)
with *Cons* **have** *wfRec*: *mat ncc nr* (*transpose nr m*) **unfolding** *mat-def* **by** *auto*
have *min nr* (*length* (*transpose nr m*)) = *nr* **using** *wfRec* **unfolding** *mat-def* **by** *auto*
moreover have *Ball* (*set* (*transpose nr* (*v* # *m*))) (*vec nc*)
proof −
{

fix *a b*
assume *mem*: (*a*,*b*) ∈ *set* (*zip v* (*transpose nr m*))
from *mem* **have** *b* ∈ *set* (*transpose nr m*) **by** (*rule set-zip-rightD*)
with *wfRec* **have** *length b* = *ncc* **unfolding** *mat-def* **using** *vec-def*[*of ncc*]
by *auto*
hence *length* (*case-prod* (#) (*a*,*b*)) = *Suc ncc* **by** *auto*
}
thus *?thesis*
by (*auto simp*: *vec-def nc*)
qed
moreover from ‹*mat nr nc* (*v* # *m*)› **have** *wfV*: *length v* = *nr* **unfolding** *mat-def* **by** (*simp add*: *vec-def*)
ultimately
show *?case* **unfolding** *mat-def*
by (*intro conjI*, *auto simp*: *wfV wfRec mat-def vec-def*)
qed (*simp add*: *mat-def vec-def set-replicate-conv-if*)

lemma *matT-vec-multI*: **assumes** *mat nr nc m*
shows *vec nc* (*matT-vec-multI ze pl ti m v*)
unfolding *matT-vec-multI-def*
using *assms*
unfolding *mat-def*
by (*simp add*: *vec-def*)

lemma *mat-mult*[*simp*,*intro*]: **assumes** *wf1*: *mat nr n m1*
and *wf2*: *mat n nc m2*
shows *mat nr nc* (*mat-multI ze pl ti nr m1 m2*)
using *assms*
unfolding *mat-def mat-multI-def* **by** (*auto simp*: *matT-vec-multI*[*OF transpose*[*OF wf1*]])

lemma *mat-pow*[*simp*,*intro*]: **assumes** *mat n n m*
shows *mat n n* (*mat-powI ze on pl ti n m i*)
proof (*induct i*)
case *0*
show *?case* **unfolding** *mat-powI.simps* **by** (*rule mat1*)
next
case (*Suc i*)
show *?case* **unfolding** *mat-powI.simps*
by (*rule mat-mult*[*OF Suc assms*])
qed

lemma *sub-vec*[*simp*,*intro*]: **assumes** *vec nr v* **and** *sd* ≤ *nr*
shows *vec sd* (*sub-vec sd v*)
using *assms* **unfolding** *vec-def sub-vec-def* **by** *auto*

lemma *sub-mat*[*simp*,*intro*]: **assumes** *wf*: *mat nr nc m* **and** *sr*: *sr* ≤ *nr* **and** *sc*: *sc* ≤ *nc*

shows *mat sr sc* (*sub-mat sr sc m*)
using *assms in-set-takeD*[*of - sc m*] *sub-vec*[*OF - sr*] **unfolding** *mat-def sub-mat-def*
by *auto*

2.4 properties of algorithms which do not depend on properties of type of matrix

lemma *mat0-index*[*simp*]: **assumes** $i < nc$ **and** $j < nr$
shows *mat0I ze nr nc* ! *i* ! *j* = *ze*
unfolding *mat0I-def vec0I-def* **using** *assms* **by** *auto*

lemma *mat0-row*[*simp*]: **assumes** $i < nr$
shows *row* (*mat0I ze nr nc*) *i* = *vec0I ze nc*
unfolding *row-def mat0I-def vec0I-def*
using *assms* **by** *auto*

lemma *mat0-col*[*simp*]: **assumes** $i < nc$
shows *col* (*mat0I ze nr nc*) *i* = *vec0I ze nr*
unfolding *mat0I-def col-def*
using *assms* **by** *auto*

lemma *vec1-index*: **assumes** *j*: $j < n$
shows *vec1I ze on n i* ! *j* = (*if i* = *j then on else ze*) (**is** - = *?r*)
unfolding *vec1I-def*
proof −
let *?l* = *replicate i ze* @ *on* # *replicate* (*n* − *1* − *i*) *ze*
have *len*: *length ?l* > *i* **by** *auto*
have *len2*: *length* (*replicate i ze* @ *on* # []) > *i* **by** *auto*
show *?l* ! *j* = *?r*
proof (*cases j* = *i*)
case *True*
thus *?thesis* **by** (*simp add*: *nth-append*)
next
case *False*
show *?thesis*
proof (*cases j* < *i*)
case *True*
thus *?thesis* **by** (*simp add*: *nth-append*)
next
case *False*
with ‹*j* ≠ *i*› **have** *gt*: *j* > *i* **by** *auto*
from *this* **have** ∃ *k*. *j* = *i* + *Suc k* **by** *arith*
from *this* **obtain** *k* **where** *k*: *j* = *i* + *Suc k* **by** *auto*
with *j* **show** *?thesis* **by** (*simp add*: *nth-append*)
qed
qed
qed

lemma *col-transpose-is-row*[*simp*]:
assumes *wf*: *mat nr nc m*
and *i*: *i* < *nr*
shows *col* (*transpose nr m*) *i* = *row m i*
using *wf*
proof (*induct m arbitrary*: *nc*)
case (*Cons v m*)
from ‹*mat nr nc* (*v* # *m*)› **obtain** *ncc* **where** *nc*: *nc* = *Suc ncc* **and** *wf*: *mat nr ncc m* **by** (*cases nc*, *auto simp*: *mat-def*)
from ‹*mat nr nc* (*v* # *m*)› *nc* **have** *lengths*: (∀ *w* ∈ *set m*. *length w* = *nr*) ∧ *length v* = *nr* ∧ *length m* = *ncc* **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
from *wf Cons* **have** *colRec*: *col* (*transpose nr m*) *i* = *row m i* **by** *auto*
hence *simpme*: *transpose nr m* ! *i* = *row m i* **unfolding** *col-def* **by** *auto*
from *wf* **have** *trans*: *mat ncc nr* (*transpose nr m*) **by** (*rule transpose*)
hence *lengths2*: (∀ *w* ∈ *set* (*transpose nr m*). *length w* = *ncc*) ∧ *length* (*transpose nr m*) = *nr* **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
{
fix *j*
assume *j* < *length* (*col* (*transpose nr* (*v* # *m*)) *i*)
hence *j* < *Suc ncc* **by** (*simp add*: *col-def lengths2 lengths i*)
hence *col* (*transpose nr* (*v* # *m*)) *i* ! *j* = *row* (*v* # *m*) *i* ! *j*
by (*cases j*, *simp add*: *row-def col-def i lengths lengths2*, *simp add*: *row-def col-def i lengths lengths2 simpme*)
} **note** *simpme* = *this*
show *?case* **by** (*rule nth-equalityI*, *simp add*: *col-def row-def lengths lengths2 i*, *rule simpme*)
qed (*simp add*: *col-def row-def mat-def i*)

lemma *mat-col-eq*:
assumes *wf1*: *mat nr nc m1*
and *wf2*: *mat nr nc m2*
shows (*m1* = *m2*) = (∀ *i* < *nc*. *col m1 i* = *col m2 i*) (**is** *?l* = *?r*)
proof
assume *?l* **thus** *?r* **by** *auto*
next
assume *?r* **show** *?l*
proof (*rule nth-equalityI*)
show *length m1* = *length m2* **using** *wf1 wf2* **unfolding** *mat-def* **by** *auto*
next
from ‹*?r*› **show** ⋀*i*. *i* < *length m1* ⟹ *m1* ! *i* = *m2* ! *i* **using** *wf1* **unfolding** *col-def mat-def* **by** *auto*
qed
qed

lemma *mat-col-eqI*:
assumes *wf1*: *mat nr nc m1*
and *wf2*: *mat nr nc m2*
and *id*: ⋀ *i*. *i* < *nc* ⟹ *col m1 i* = *col m2 i*

```
  shows m1 = m2
  unfolding mat-col-eq[OF wf1 wf2] using id by auto

lemma mat-eq:
  assumes wf1: mat nr nc m1
  and wf2: mat nr nc m2
  shows (m1 = m2) = (∀ i < nc. ∀ j < nr. m1 ! i ! j = m2 ! i ! j) (is ?l = ?r)
proof
  assume ?l thus ?r by auto
next
  assume ?r show ?l
  proof (rule mat-col-eqI[OF wf1 wf2], unfold col-def)
    fix i
    assume i: i < nc
    show m1 ! i = m2 ! i
    proof (rule nth-equalityI)
      show length (m1 ! i) = length (m2 ! i) using wf1 wf2 i unfolding mat-def
by (auto simp: vec-def)
    next
      from ‹?r› i show ⋀j. j < length (m1 ! i) ⟹ m1 ! i ! j = m2 ! i ! j
        using wf1 wf2 unfolding mat-def by (auto simp: vec-def)
    qed
  qed
qed

lemma mat-eqI:
  assumes wf1: mat nr nc m1
  and wf2: mat nr nc m2
  and id: ⋀ i j. i < nc ⟹ j < nr ⟹ m1 ! i ! j = m2 ! i ! j
  shows m1 = m2
  unfolding mat-eq[OF wf1 wf2] using id by auto

lemma vec-eq:
  assumes wf1: vec n v1
  and wf2: vec n v2
  shows (v1 = v2) = (∀ i < n. v1 ! i = v2 ! i) (is ?l = ?r)
proof
  assume ?l thus ?r by auto
next
  assume ?r show ?l
  proof (rule nth-equalityI)
    from wf1 wf2 show length v1 = length v2 unfolding vec-def by simp
  next
    from ‹?r› wf1 show ⋀i. i < length v1 ⟹ v1 ! i = v2 ! i unfolding vec-def
by simp
  qed
qed

lemma vec-eqI:
```

```
  assumes wf1: vec n v1
  and wf2: vec n v2
  and id: ⋀ i. i < n ⟹ v1 ! i = v2 ! i
  shows v1 = v2
  unfolding vec-eq[OF wf1 wf2] using id by auto


lemma row-col: assumes mat nr nc m
  and i < nr and j < nc
  shows row m i ! j = col m j ! i
using assms unfolding mat-def row-def col-def
  by auto

lemma col-index: assumes m: mat nr nc m
  and i: i < nc
  shows col m i = map (λ j. m ! i ! j) [0 ..< nr]
proof −
  from m[unfolded mat-def] i
  have nr: nr = length (m ! i) by (auto simp: vec-def)
  show ?thesis unfolding nr col-def
    by (rule map-nth[symmetric])
qed

lemma row-index: assumes m: mat nr nc m
  and i: i < nr
  shows row m i = map (λ j. m ! j ! i) [0 ..< nc]
proof −
  note rc = row-col[OF m i]
  from row[OF m i] have id: length (row m i) = nc unfolding vec-def by simp
  from map-nth[of row m i]
  have row m i = map (λ j. row m i ! j) [0 ..< nc] unfolding id by simp
  also have ... = map (λ j. m ! j ! i) [0 ..< nc] using rc[unfolded col-def] by auto
  finally show ?thesis .
qed


lemma mat-row-eq:
  assumes wf1: mat nr nc m1
  and wf2: mat nr nc m2
  shows (m1 = m2) = (∀ i < nr. row m1 i = row m2 i) (is ?l = ?r)
proof
  assume ?l thus ?r by auto
next
  assume ?r show ?l
  proof (rule nth-equalityI)
    show length m1 = length m2 using wf1 wf2 unfolding mat-def by auto
  next
    show m1 ! i = m2 ! i if i: i < length m1 for i
    proof −
```

```
      show m1 ! i = m2 ! i
      proof (rule nth-equalityI)
       show length (m1 ! i) = length (m2 ! i) using wf1 wf2 i unfolding mat-def
by (auto simp: vec-def)
      next
       show m1 ! i ! j = m2 ! i ! j if j: j < length (m1 ! i) for j
       proof −
        from i j wf1 have i1: i < nc and j1: j < nr unfolding mat-def by (auto
simp: vec-def)
         from ‹?r› j1 have col m1 i ! j = col m2 i ! j
           by (simp add: row-col[OF wf1 j1 i1, symmetric] row-col[OF wf2 j1 i1,
symmetric])
         thus m1 ! i ! j = m2 ! i ! j unfolding col-def .
       qed
      qed
    qed
  qed
qed

lemma mat-row-eqI:
  assumes wf1: mat nr nc m1
  and wf2: mat nr nc m2
  and id: ⋀ i. i < nr ⟹ row m1 i = row m2 i
  shows m1 = m2
  unfolding mat-row-eq[OF wf1 wf2] using id by auto

lemma row-transpose-is-col[simp]:  assumes wf: mat nr nc m
  and i: i < nc
  shows row (transpose nr m) i = col m i
proof −
  have len: length (row (transpose nr m) i) = length (col m i)
     using transpose[OF wf]  wf i  unfolding row-def col-def mat-def by (auto
simp: vec-def)
  show ?thesis
  proof (rule nth-equalityI[OF len])
    fix j
    assume j < length (row (transpose nr m) i)
   hence j: j < nr using transpose[OF wf] wf i unfolding row-def col-def mat-def
by (auto simp: vec-def)
    show row (transpose nr m) i ! j = col m i ! j
      by (simp only: row-col[OF transpose[OF wf] i j],
        simp only: col-transpose-is-row[OF wf j],
        simp only: row-col[OF wf j i])
  qed
qed


lemma matT-vec-mult-to-scalar:
  assumes mat nr nc m
```

and *vec nr v*
and $i < nc$
shows *matT-vec-multI ze pl ti m v ! i = scalar-prodI ze pl ti (col m i) v*
unfolding *matT-vec-multI-def* **using** *assms* **unfolding** *mat-def col-def* **by** (*auto simp*: *vec-def*)

lemma *mat-vec-mult-index*:
assumes *wf*: *mat nr nc m*
and *wfV*: *vec nc v*
and *i*: $i < nr$
shows *matT-vec-multI ze pl ti (transpose nr m) v ! i = scalar-prodI ze pl ti (row m i) v*
by (*simp only*:*matT-vec-mult-to-scalar*[*OF transpose*[*OF wf*] *wfV i*],
simp only: *col-transpose-is-row*[*OF wf i*])

lemma *mat-mult-index*[*simp*] :
assumes *wf1*: *mat nr n m1*
and *wf2*: *mat n nc m2*
and *i*: $i < nr$
and *j*: $j < nc$
shows *mat-multI ze pl ti nr m1 m2 ! j ! i = scalar-prodI ze pl ti (row m1 i) (col m2 j)*
proof −
have *jlen*: $j <$ *length m2* **using** *wf2 j* **unfolding** *mat-def* **by** *auto*
have *wfj*: *vec n (m2 ! j)* **using** *jlen j wf2* **unfolding** *mat-def* **by** *auto*
show *?thesis*
unfolding *mat-multI-def*
by (*simp add*: *jlen*, *simp only*: *mat-vec-mult-index*[*OF wf1 wfj i*], *unfold col-def*, *simp*)
qed

lemma *col-mat-mult-index* :
assumes *wf1*: *mat nr n m1*
and *wf2*: *mat n nc m2*
and *j*: $j < nc$
shows *col (mat-multI ze pl ti nr m1 m2) j = map* (λ *i. scalar-prodI ze pl ti (row m1 i) (col m2 j))* [*0* ..< *nr*] (**is** *col ?l j = ?r*)
proof −
have *wf12*: *mat nr nc ?l* **by** (*rule mat-mult*[*OF wf1 wf2*])
have *len*: *length (col ?l j) = length ?r* **and** *nr*: *length (col ?l j) = nr* **using** *wf1 wf2 wf12 j* **unfolding** *mat-def col-def* **by** (*auto simp*: *vec-def*)
show *?thesis* **by** (*rule nth-equalityI*[*OF len*], *simp add*: *j nr*, *unfold col-def*, *simp only*:
mat-mult-index[*OF wf1 wf2 - j*], *simp add*: *col-def*)
qed

lemma *row-mat-mult-index* :
assumes *wf1*: *mat nr n m1*
and *wf2*: *mat n nc m2*

```
  and i: i < nr
  shows row (mat-multI ze pl ti nr m1 m2) i = map (λ j. scalar-prodI ze pl ti (row
m1 i) (col m2 j)) [0 ..< nc] (is row ?l i = ?r)
proof −
  have wf12: mat nr nc ?l by (rule mat-mult[OF wf1 wf2])
  hence lenL: length ?l = nc unfolding mat-def by simp
  have len: length (row ?l i) = length ?r and nc: length (row ?l i) = nc using wf1
wf2 wf12 i unfolding mat-def row-def by (auto simp: vec-def)
  show ?thesis by (rule nth-equalityI[OF len], simp add: i nc, unfold row-def, simp
add: lenL, simp only:
    mat-mult-index[OF wf1 wf2 i], simp add: row-def)
qed

lemma scalar-prod-cons:
  scalar-prodI ze pl ti (a # as) (b # bs) = pl (ti a b) (scalar-prodI ze pl ti as bs)
unfolding scalar-prodI-def by auto


lemma vec-plus-index[simp]:
  assumes wf1: vec nr v1
  and wf2: vec nr v2
  and i: i < nr
  shows vec-plusI pl v1 v2 ! i = pl (v1 ! i)  (v2 ! i)
using wf1 wf2 i
unfolding vec-def vec-plusI-def
proof (induct v1 arbitrary: i v2 nr, simp)
  case (Cons a v11)
  from Cons obtain b v22 where v2: v2 = b # v22 by (cases v2, auto)
  from v2 Cons obtain nrr where nr: nr = Suc nrr by (force)
  from Cons show ?case
    by (cases i, simp add: v2, auto simp: v2 nr)
qed

lemma mat-map-index[simp]: assumes wf: mat nr nc m and i: i < nc and j: j
< nr
  shows mat-map f m ! i ! j = f (m ! i ! j)
proof −
  from wf i have i: i < length m unfolding mat-def by auto
  with wf j have j: j < length (m ! i) unfolding mat-def by (auto simp: vec-def)
   have mat-map f m ! i ! j = map (map f) m ! i ! j unfolding mat-map-def
vec-map-def by auto
  also have ... = map f (m ! i) ! j using i by auto
  also have ... = f (m ! i ! j) using j by auto
  finally show ?thesis .
qed


lemma mat-plus-index[simp]:
  assumes wf1: mat nr nc m1
```

and *wf2*: *mat nr nc m2*
and *i*: $i < nc$
and *j*: $j < nr$
shows *mat-plusI pl m1 m2* ! *i* ! *j* = *pl* (*m1* ! *i* ! *j*) (*m2* ! *i* ! *j*)
using *wf1 wf2 i*
unfolding *mat-plusI-def mat-def*
proof (*simp*, *induct m1 arbitrary*: *m2 i nc*, *simp*)
case (*Cons v1 m11*)
from *Cons* **obtain** *v2 m22* **where** *m2*: *m2* = *v2* # *m22* **by** (*cases m2*, *auto*)
from *m2 Cons* **obtain** *ncc* **where** *nc*: *nc* = *Suc ncc* **by** *force*
show *?case*
proof (*cases i*, *simp add*: *m2*, *rule vec-plus-index*[**where** *nr* = *nr*], (*auto simp*: *Cons j m2*)[*3*])
case (*Suc ii*)
with *Cons* **show** *?thesis* **using** *m2 nc* **by** *auto*
qed
qed

lemma *col-mat-plus*: **assumes** *wf1*: *mat nr nc m1*
and *wf2*: *mat nr nc m2*
and *i*: $i < nc$
shows *col* (*mat-plusI pl m1 m2*) *i* = *vec-plusI pl* (*col m1 i*) (*col m2 i*)
using *assms*
unfolding *mat-plusI-def col-def mat-def*
proof (*induct m1 arbitrary*: *m2 nc i*, *simp*)
case (*Cons v m1*)
from *Cons* **obtain** *v2 m22* **where** *m2*: *m2* = *v2* # *m22* **by** (*cases m2*, *auto*)
from *m2 Cons* **obtain** *ncc* **where** *nc*: *nc* = *Suc ncc* **by** *force*
show *?case*
proof (*cases i*, *simp add*: *m2*)
case (*Suc ii*)
with *Cons* **show** *?thesis* **using** *m2 nc* **by** *auto*
qed
qed

lemma *transpose-index*[*simp*]: **assumes** *wf*: *mat nr nc m*
and *i*: $i < nr$
and *j*: $j < nc$
shows *transpose nr m* ! *i* ! *j* = *m* ! *j* ! *i*
proof −
have *transpose nr m* ! *i* ! *j* = *col* (*transpose nr m*) *i* ! *j* **unfolding** *col-def* **by** *simp*
also have ... = *row m i* ! *j* **using** *col-transpose-is-row*[*OF wf i*] **by** *simp*
also have ... = *m* ! *j* ! *i* **unfolding** *row-def* **using** *wf j* **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
finally show *?thesis* **.**
qed

lemma *transpose-mat-plus*: **assumes** *wf*: *mat nr nc m1 mat nr nc m2*

shows *transpose nr* (*mat-plusI pl m1 m2*) = *mat-plusI pl* (*transpose nr m1*) (*transpose nr m2*) (**is** *?l* = *?r*)
proof (*rule mat-eqI*)
fix *i j*
assume *i*: *i* < *nr* **and** *j*: *j* < *nc*
note [*simp*] = *transpose-index*[*OF - this*] *mat-plus-index*[*OF - - j i*] *mat-plus-index*[*OF - - this*]
show *?l* ! *i* ! *j* = *?r* ! *i* ! *j* **using** *wf* **by** *simp*
qed (*auto intro*: *wf*)

lemma *row-mat-plus*: **assumes** *wf1*: *mat nr nc m1*
and *wf2*: *mat nr nc m2*
and *i*: *i* < *nr*
shows *row* (*mat-plusI pl m1 m2*) *i* = *vec-plusI pl* (*row m1 i*) (*row m2 i*)
by (
simp only: *col-transpose-is-row*[*OF mat-plus*[*OF wf1 wf2*] *i*, *symmetric*],
simp only: *transpose-mat-plus*[*OF wf1 wf2*],
simp only: *col-mat-plus*[*OF transpose*[*OF wf1*] *transpose*[*OF wf2*] *i*],
simp only: *col-transpose-is-row*[*OF wf1 i*],
simp only: *col-transpose-is-row*[*OF wf2 i*])

lemma *col-mat1*: **assumes** *i* < *nr*
shows *col* (*mat1I ze on nr*) *i* = *vec1I ze on nr i*
unfolding *mat1I-def col-def* **using** *assms* **by** *auto*

lemma *mat1-index*: **assumes** *i*: *i* < *n* **and** *j*: *j* < *n*
shows *mat1I ze on n* ! *i* ! *j* = (*if i* = *j then on else ze*)
by (*simp add*: *col-mat1*[*OF i*, *simplified col-def*] *vec1-index*[*OF j*])

lemma *transpose-mat1*: *transpose nr* (*mat1I ze on nr*) = (*mat1I ze on nr*) (**is** *?l* = *?r*)
proof (*rule mat-eqI*)
fix *i j*
assume *i*:*i* < *nr* **and** *j*: *j* < *nr*
note [*simp*] = *transpose-index*[*OF - this*] *mat1-index*[*OF this*] *mat1-index*[*OF j i*]
show *?l* ! *i* ! *j* = *?r* ! *i* ! *j* **by** *auto*
qed *auto*

lemma *row-mat1*: **assumes** *i*: *i* < *nr*
shows *row* (*mat1I ze on nr*) *i* = *vec1I ze on nr i*
by (*simp only*: *col-transpose-is-row*[*OF mat1 i*, *symmetric*],
simp only: *transpose-mat1*,
simp only: *col-mat1*[*OF i*])

lemma *sub-mat-index*:
assumes *wf*: *mat nr nc m*

and *sr*: $sr \leq nr$
and *sc*: $sc \leq nc$
and *j*: $j < sr$
and *i*: $i < sc$
shows *sub-mat sr sc m* ! *i* ! *j* = *m* ! *i* ! *j*
proof −
from *assms* **have** *im*: $i <$ *length m* **unfolding** *mat-def* **by** *auto*
from *assms* **have** *jm*: $j <$ *length* (*m* ! *i*) **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
have *sub-mat sr sc m* ! *i* ! *j* = *map* (*take sr*) (*take sc m*) ! *i* ! *j*
unfolding *sub-mat-def sub-vec-def* **by** *auto*
also have ... = *take sr* (*m* ! *i*) ! *j* **using** *i im* **by** *auto*
also have ... = *m* ! *i* ! *j* **using** *j jm* **by** *auto*
finally show *?thesis* **.**
qed

2.5 lemmas requiring properties of plus, times, ...

context *plus*
begin

abbreviation *vec-plus* :: *'a vec* ⇒ *'a vec* ⇒ *'a vec*
where *vec-plus* ≡ *vec-plusI plus*

abbreviation *mat-plus* :: *'a mat* ⇒ *'a mat* ⇒ *'a mat*
where *mat-plus* ≡ *mat-plusI plus*
end

context *semigroup-add*
begin
lemma *vec-plus-assoc*: **assumes** *vec*: *vec nr u vec nr v vec nr w*
shows *vec-plus u* (*vec-plus v w*) = *vec-plus* (*vec-plus u v*) *w*
proof (*rule vec-eqI*)
fix *i*
assume *i*: $i < nr$
note [*simp*] = *vec-plus-index*[*OF - - i*]
from *vec*
show *vec-plus u* (*vec-plus v w*) ! *i* = *vec-plus* (*vec-plus u v*) *w* ! *i*
by (*auto simp*: *add.assoc*)
qed (*auto intro*: *vec*)

lemma *mat-plus-assoc*: **assumes** *wf*: *mat nr nc m1 mat nr nc m2 mat nr nc m3*
shows *mat-plus m1* (*mat-plus m2 m3*) = *mat-plus* (*mat-plus m1 m2*) *m3* (**is** *?l* = *?r*)
proof (*rule mat-eqI*)
fix *i j*
assume $i < nc$ $j < nr$
note [*simp*] = *mat-plus-index*[*OF - - this*]
show *?l* ! *i* ! *j* = *?r* ! *i* ! *j* **using** *wf* **by** (*simp add*: *add.assoc*)

qed (*auto simp*: *wf*)
end

context *ab-semigroup-add*
begin
lemma *vec-plus-comm*: *vec-plus x y* = *vec-plus y x*
unfolding *vec-plusI-def*
proof (*induct x arbitrary*: *y*)
case (*Cons a x*)
thus *?case*
by (*cases y*, *auto simp*: *add.commute*)
qed *simp*

lemma *mat-plus-comm*: *mat-plus m1 m2* = *mat-plus m2 m1*
unfolding *mat-plusI-def*
proof (*induct m1 arbitrary*: *m2*)
case (*Cons v m1*) **note** *oCons* = *this*
thus *?case*
proof (*cases m2*)
case (*Cons w m2a*)
hence *mat-plus* (*v* # *m1*) *m2* = *vec-plus v w* # *mat-plus m1 m2a* **by** (*auto simp*: *mat-plusI-def*)
also have ... = *vec-plus w v* # *mat-plus m1 m2a* **using** *vec-plus-comm* **by** *auto*
finally show *?thesis* **using** *Cons oCons* **by** (*auto simp*: *mat-plusI-def*)
qed *simp*
qed *simp*
end

context *zero*
begin
abbreviation *vec0* :: *nat* ⇒ *'a vec*
where *vec0* ≡ *vec0I zero*

abbreviation *mat0* :: *nat* ⇒ *nat* ⇒ *'a mat*
where *mat0* ≡ *mat0I zero*
end

context *monoid-add*
begin
lemma *vec0-plus*[*simp*]: **assumes** *vec nr u* **shows** *vec-plus* (*vec0 nr*) *u* = *u*
using *assms*
unfolding *vec-def vec-plusI-def vec0I-def*
proof (*induct nr arbitrary*: *u*)
case (*Suc nn*) **thus** *?case* **by** (*cases u*, *auto*)
qed *simp*

lemma *plus-vec0*[*simp*]: **assumes** *vec nr u* **shows** *vec-plus u* (*vec0 nr*) = *u*

using *assms*
unfolding *vec-def vec-plusI-def vec0I-def*
proof (*induct nr arbitrary*: *u*)
case (*Suc nn*) **thus** *?case* **by** (*cases u*, *auto*)
qed *simp*

lemma *plus-mat0*[*simp*]: **assumes** *wf*: *mat nr nc m* **shows** *mat-plus m* (*mat0 nr nc*) = *m* (**is** *?l* = *?r*)
proof (*rule mat-eqI*)
fix *i j*
assume *i* < *nc j* < *nr*
note [*simp*] = *mat-plus-index*[*OF - - this*] *mat0-index*[*OF this*]
show *?l* ! *i* ! *j* = *?r* ! *i* ! *j* **using** *wf* **by** *simp*
qed (*insert wf*, *auto*)

lemma *mat0-plus*[*simp*]: **assumes** *wf*: *mat nr nc m* **shows** *mat-plus* (*mat0 nr nc*) *m* = *m* (**is** *?l* = *?r*)
proof (*rule mat-eqI*)
fix *i j*
assume *i* < *nc j* < *nr*
note [*simp*] = *mat-plus-index*[*OF - - this*] *mat0-index*[*OF this*]
show *?l* ! *i* ! *j* = *?r* ! *i* ! *j* **using** *wf* **by** *simp*
qed (*insert wf*, *auto*)
end

context *semiring-0*
begin
abbreviation *scalar-prod* :: *'a vec* ⇒ *'a vec* ⇒ *'a*
where *scalar-prod* ≡ *scalar-prodI zero plus times*

abbreviation *mat-mult* :: *nat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *'a mat*
where *mat-mult* ≡ *mat-multI zero plus times*

lemma *scalar-prod*: *scalar-prod v1 v2* = *sum-list* (*map* (λ(*x*,*y*). *x* ∗ *y*) (*zip v1 v2*))
proof −
obtain *z* **where** *z*: *zip v1 v2* = *z* **by** *auto*
show *?thesis* **unfolding** *scalar-prodI-def z*
by (*induct z*, *auto*)
qed

lemma *scalar-prod-last*: **assumes** *length v1* = *length v2*
shows *scalar-prod* (*v1* @ [*x1*]) (*v2* @ [*x2*]) = *x1* ∗ *x2* + *scalar-prod v1 v2*
using *assms*
proof (*induct v1 arbitrary*: *v2*)
case (*Cons y1 w1*)
from *Cons*(*2*) **obtain** *y2 w2* **where** *v2*: *v2* = *Cons y2 w2* **and** *len*: *length w1* = *length w2* **by** (*cases v2*, *auto*)
from *Cons*(*1*)[*OF len*] **have** *rec*: *scalar-prod* (*w1* @ [*x1*]) (*w2* @ [*x2*]) = *x1* ∗ *x2* + *scalar-prod w1 w2* **.**

have *scalar-prod* ((*y1* # *w1*) @ [*x1*]) (*v2* @ [*x2*]) =
(*y1* ∗ *y2* + *x1* ∗ *x2*) + *scalar-prod w1 w2* **by** (*simp add*: *scalar-prod-cons v2 rec add.assoc*)
also have ... = (*x1* ∗ *x2* + *y1* ∗ *y2*) + *scalar-prod w1 w2* **using** *add.commute*[*of x1* ∗ *x2*] **by** *simp*
also have ... = *x1* ∗ *x2* + (*scalar-prod* (*y1* # *w1*) *v2*) **by** (*simp add*: *add.assoc scalar-prod-cons v2*)
finally show *?case* **.**
qed (*simp add*: *scalar-prodI-def*)

lemma *scalar-product-assoc*:
assumes *wfm*: *mat nr nc m*
and *wfr*: *vec nr r*
and *wfc*: *vec nc c*
shows *scalar-prod* (*map* (λ*k*. *scalar-prod r* (*col m k*)) [*0*..<*nc*]) *c* = *scalar-prod r* (*map* (λ*k*. *scalar-prod* (*row m k*) *c*) [*0*..<*nr*])
using *wfm wfc*
unfolding *col-def*
proof (*induct m arbitrary*: *nc c*)
case *Nil*
hence *nc*: *nc* = *0* **unfolding** *mat-def* **by** (*auto*)
from *wfr* **have** *nr*: *nr* = *length r* **unfolding** *vec-def* **by** *auto*
let *?term* = λ *r* :: *'a vec*. *zip r* (*map* (λ *k*. *zero*) [*0*..<*length r*])
let *?fun* = λ (*x*,*y*). *plus* (*times x y*)
have *foldr ?fun* (*?term r*) *zero* = *zero*
proof (*induct r*, *simp*)
case (*Cons d r*)
have *foldr ?fun* (*?term* (*d* # *r*)) *zero* = *foldr ?fun* ((*d*,*zero*) # *?term r*) *zero* **by** (*simp only*: *map-replicate-trivial*, *simp*)
also have ... = *zero* **using** *Cons* **by** *simp*
finally show *?case* **.**
qed
hence *zero* = *foldr ?fun* (*zip r* (*map* (λ *k*. *zero*) [*0*..<*nr*])) *zero* **by** (*simp add*: *nr*)
with *Nil nc* **show** *?case*
by (*simp add*: *scalar-prodI-def row-def*)
next
case (*Cons v m*)
from *this* **obtain** *ncc* **where** *nc*: *nc* = *Suc ncc* **and** *wf*: *mat nr ncc m* **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
from *nc* ‹*vec nc c*› **obtain** *a cc* **where** *c*: *c* = *a* # *cc* **and** *wfc*: *vec ncc cc* **unfolding** *vec-def* **by** (*cases c*, *auto*)
have *rec*: *scalar-prod* (*map* (λ *k*. *scalar-prod r* (*m* ! *k*)) [*0*..<*ncc*]) *cc* = *scalar-prod r* (*map* (λ *k*. *scalar-prod* (*row m k*) *cc*) [*0*..<*nr*])
by (*rule Cons*, *rule wf*, *rule wfc*)
have *id*: *map* (λ*k*. *scalar-prod r* ((*v* # *m*) ! *k*)) [*0*..<*Suc ncc*] = *scalar-prod r v* # *map* (λ *k*. *scalar-prod r* (*m* ! *k*)) [*0*..<*ncc*] **by** (*induct ncc*, *auto*)
from *wfr* **have** *nr*: *nr* = *length r* **unfolding** *vec-def* **by** *auto*
with *Cons* **have** *v*: *length v* = *length r* **unfolding** *mat-def* **by** (*auto simp*:

vec-def)
have $\forall\ i < nr.\ vec\ ncc\ (row\ m\ i)$ **by** (*intro allI impI*, *rule row*[*OF wf*], *simp*)
obtain *tm* **where** *tm*: $tm = transpose\ nr\ m$ **by** *auto*
hence *idk*: $\forall\ k < length\ r.\ row\ m\ k = tm\ !\ k$ **using** *col-transpose-is-row*[*OF wf*] **unfolding** *col-def* **by** (*auto simp*: *nr*)
hence *idtm1*: $map\ (\lambda k.\ scalar\text{-}prod\ (row\ m\ k)\ cc)\ [0..<length\ r] = map\ (\lambda k.\ scalar\text{-}prod\ (tm\ !\ k)\ cc)\ [0..<length\ r]$
and *idtm2*: $map\ (\lambda k.\ plus\ (times\ (v\ !\ k)\ a)\ (scalar\text{-}prod\ (row\ m\ k)\ cc))\ [0..<length\ r] = map\ (\lambda k.\ plus\ (times\ (v\ !\ k)\ a)\ (scalar\text{-}prod\ (tm\ !\ k)\ cc))\ [0..<length\ r]$ **by** *auto*
from *tm transpose*[*OF wf*] **have** *mat ncc nr tm* **by** *simp*
with *nr* **have** $length\ tm = length\ r$ **and** $(\forall\ i < length\ r.\ length\ (tm\ !\ i) = ncc)$ **unfolding** *mat-def* **by** (*auto simp*: *vec-def*)
with *v* **have** *main*: $plus\ (times\ (scalar\text{-}prod\ r\ v)\ a)\ (scalar\text{-}prod\ r\ (map\ (\lambda k.\ scalar\text{-}prod\ (tm\ !\ k)\ cc)\ [0..<length\ r])) =$
$scalar\text{-}prod\ r\ (map\ (\lambda k.\ plus\ (times\ (v\ !\ k)\ a)\ (scalar\text{-}prod\ (tm\ !\ k)\ cc))\ [0..<length\ r])$
proof (*induct r arbitrary*: *v tm*)
case *Nil*
thus *?case* **by** (*auto simp*: *scalar-prodI-def row-def*)
next
case (*Cons b r*)
from *this* **obtain** *c vv* **where** *v*: $v = c\ \#\ vv$ **and** *vvlen*: $length\ vv = length\ r$ **by** (*cases v*, *auto*)
from *Cons* **obtain** *u mm* **where** *tm*: $tm = u\ \#\ mm$ **and** *mmlen*: $length\ mm = length\ r$ **by** (*cases tm*, *auto*)
from *Cons tm* **have** *argLen*: $\forall\ i < length\ r.\ length\ (mm\ !\ i) = ncc$ **by** *auto*
have *rec*: $plus\ (times\ (scalar\text{-}prod\ r\ vv)\ a)\ (scalar\text{-}prod\ r\ (map\ (\lambda k.\ scalar\text{-}prod\ (mm\ !\ k)\ cc)\ [0..<length\ r])) =$
$scalar\text{-}prod\ r\ (map\ (\lambda k.\ plus\ (times\ (vv\ !\ k)\ a)\ (scalar\text{-}prod\ (mm\ !\ k)\ cc))\ [0..<length\ r])$
(**is** $plus\ (times\ ?rv\ a)\ ?recl = ?recr$)
by (*rule Cons*, *auto simp*: *vvlen mmlen argLen*)
have *id*: $map\ (\lambda k.\ scalar\text{-}prod\ ((u\ \#\ mm)\ !\ k)\ cc)\ [0..<length\ (b\ \#\ r)] = scalar\text{-}prod\ u\ cc\ \#\ map\ (\lambda k.\ scalar\text{-}prod\ (mm\ !\ k)\ cc)\ [0..<length\ r]$
by (*simp*, *induct r*, *auto*)
have *id2*: $map\ (\lambda k.\ plus\ (times\ ((c\ \#\ vv)\ !\ k)\ a)\ (scalar\text{-}prod\ ((u\ \#\ mm)\ !\ k)\ cc))\ [0..<length\ (b\ \#\ r)] =$
$(plus\ (times\ c\ a)\ (scalar\text{-}prod\ u\ cc))\ \#$
$map\ (\lambda k.\ plus\ (times\ (vv\ !\ k)\ a)\ (scalar\text{-}prod\ (mm\ !\ k)\ cc))\ [0..<length\ r]$
by (*simp*, *induct r*, *auto*)
show *?case* **proof** (*simp only*: *v tm*, *simp only*: *id*, *simp only*: *id2*, *simp only*: *scalar-prod-cons*)
let *?uc* = *scalar-prod u cc*
let *?bca* = *times* (*times b c*) *a*
have $plus\ (times\ (plus\ (times\ b\ c)\ ?rv)\ a)\ (plus\ (times\ b\ ?uc)\ ?recl) = plus\ (plus\ ?bca\ (times\ ?rv\ a))\ (plus\ (times\ b\ ?uc)\ ?recl)$
by (*simp add*: *distrib-right*)

```
    also have ... = plus (plus ?bca (times ?rv a)) (plus ?recl (times b ?uc)) by
(simp add: add.commute)
    also have ... = plus ?bca (plus (plus (times ?rv a) ?recl) (times b ?uc)) by
(simp add: add.assoc)
    also have ... = plus ?bca (plus ?recr (times b ?uc)) by (simp only: rec)
   also have ... = plus ?bca (plus (times b ?uc) ?recr) by (simp add: add.commute)
    also have ... = plus (times b (plus (times c a) ?uc)) ?recr by (simp add:
distrib-left mult.assoc add.assoc)
    finally show plus (times (plus (times b c) ?rv) a) (plus (times b ?uc) ?recl)
= plus (times b (plus (times c a) ?uc)) ?recr .
   qed
  qed
  show ?case
    by (simp only: c scalar-prod-cons, simp only: nc, simp only: id, simp only:
scalar-prod-cons, simp only: rec, simp only: nr, simp only: idtm1 idtm2, simp only:
main, simp only: idtm2[symmetric], simp add: row-def scalar-prod-cons)
qed


lemma mat-mult-assoc:
  assumes wf1: mat nr n1 m1
  and wf2: mat n1 n2 m2
  and wf3: mat n2 nc m3
  shows mat-mult nr (mat-mult nr m1 m2) m3 = mat-mult nr m1 (mat-mult n1
m2 m3) (is ?m12-3 = ?m1-23)
proof −
  note wf = wf1 wf2 wf3
  let ?m12 = mat-mult nr m1 m2
  let ?m23 = mat-mult n1 m2 m3
  from wf have
    wf12: mat nr n2 ?m12 and
    wf23: mat n1 nc ?m23 and
    wf1-23: mat nr nc ?m1-23 and
    wf12-3: mat nr nc ?m12-3 by auto
  show ?thesis
  proof (rule mat-col-eqI, unfold col-def)
    fix i
    assume i: i < nc
    with wf1-23 wf12-3 wf3 have len: length (?m12-3 ! i) = length (?m1-23 ! i)
and ilen: i < length m3 unfolding mat-def by (auto simp: vec-def)
    show ?m12-3 ! i = ?m1-23 ! i
    proof (rule nth-equalityI[OF len])
      fix j
      assume jlen: j < length (?m12-3 ! i)
      with wf12-3 i have j: j < nr unfolding mat-def by (auto simp: vec-def)
      show ?m12-3 ! i ! j = ?m1-23 ! i ! j
        by (unfold mat-mult-index[OF wf12 wf3 j i]
              mat-mult-index[OF wf1 wf23 j i]
              row-mat-mult-index[OF wf1 wf2 j]
```

```
        col-mat-mult-index[OF wf2 wf3 i]
        scalar-product-assoc[OF wf2 row[OF wf1 j] col[OF wf3 i]], simp)
   qed
  qed (insert wf, auto)
qed

lemma mat-mult-assoc-n:
  assumes wf1: mat n n m1
  and wf2: mat n n m2
  and wf3: mat n n m3
  shows mat-mult n (mat-mult n m1 m2) m3 = mat-mult n m1 (mat-mult n m2
m3)
using assms
 by (rule mat-mult-assoc)


lemma scalar-left-zero: scalar-prod (vec0 nn) v = zero
  unfolding vec0I-def scalar-prodI-def
proof (induct nn arbitrary: v)
  case (Suc m)
  thus ?case by (cases v, auto)
qed simp

lemma scalar-right-zero: scalar-prod v (vec0 nn) = zero
  unfolding vec0I-def scalar-prodI-def
proof (induct v arbitrary: nn)
  case (Cons a vv)
  thus ?case by (cases nn, auto)
qed simp

lemma mat0-mult-left: assumes wf: mat nc ncc m
  shows mat-mult nr (mat0 nr nc) m = (mat0 nr ncc)
proof (rule mat-eqI)
  fix i j
  assume i: i < ncc and j: j < nr
  show mat-mult nr (mat0 nr nc) m ! i ! j = mat0 nr ncc ! i ! j
    by (unfold mat-mult-index[OF mat0 wf j i] mat0-index[OF i j] mat0-row[OF j]
scalar-left-zero, simp)
qed (auto simp: wf)


lemma mat0-mult-right: assumes wf: mat nr nc m
  shows mat-mult nr m (mat0 nc ncc) = (mat0 nr ncc)
proof (rule mat-eqI)
  fix i j
  assume i: i < ncc and j: j < nr
  show mat-mult nr m (mat0 nc ncc) ! i ! j = mat0 nr ncc ! i ! j
    by (unfold mat-mult-index[OF wf mat0 j i] mat0-index[OF i j] mat0-col[OF i]
scalar-right-zero, simp)
```

qed (*insert wf*, *auto*)

lemma *scalar-vec-plus-distrib-right*:
assumes *wf1*: *vec nr u*
assumes *wf2*: *vec nr v*
assumes *wf3*: *vec nr w*
shows *scalar-prod u* (*vec-plus v w*) = *plus* (*scalar-prod u v*) (*scalar-prod u w*)
using *assms*
unfolding *vec-def scalar-prodI-def vec-plusI-def*
proof (*induct nr arbitrary*: *u v w*)
case (*Suc n*)
from *Suc* **obtain** *a uu* **where** *u*: *u* = *a* # *uu* **by** (*cases u*, *auto*)
from *Suc* **obtain** *b vv* **where** *v*: *v* = *b* # *vv* **by** (*cases v*, *auto*)
from *Suc* **obtain** *c ww* **where** *w*: *w* = *c* # *ww* **by** (*cases w*, *auto*)
from *Suc u v w* **have** *lu*: *length uu* = *n* **and** *lv*: *length vv* = *n* **and** *lw*: *length ww* = *n* **by** *auto*
show *?case* **by** (*simp only*: *u v w*, *simp*, *simp only*: *Suc*(*1*)[*OF lu lv lw*], *simp add*: *add.commute*[*of - times a c*] *distrib-left add.assoc*[*symmetric*])
qed *simp*

lemma *scalar-vec-plus-distrib-left*:
assumes *wf1*: *vec nr u*
assumes *wf2*: *vec nr v*
assumes *wf3*: *vec nr w*
shows *scalar-prod* (*vec-plus u v*) *w* = *plus* (*scalar-prod u w*) (*scalar-prod v w*)
using *assms*
unfolding *vec-def scalar-prodI-def vec-plusI-def*
proof (*induct nr arbitrary*: *u v w*)
case (*Suc n*)
from *Suc* **obtain** *a uu* **where** *u*: *u* = *a* # *uu* **by** (*cases u*, *auto*)
from *Suc* **obtain** *b vv* **where** *v*: *v* = *b* # *vv* **by** (*cases v*, *auto*)
from *Suc* **obtain** *c ww* **where** *w*: *w* = *c* # *ww* **by** (*cases w*, *auto*)
from *Suc u v w* **have** *lu*: *length uu* = *n* **and** *lv*: *length vv* = *n* **and** *lw*: *length ww* = *n* **by** *auto*
show *?case* **by** (*simp only*: *u v w*, *simp*, *simp only*: *Suc*(*1*)[*OF lu lv lw*], *simp add*: *add.commute*[*of - times b c*] *distrib-right add.assoc*[*symmetric*])
qed *simp*

lemma *mat-mult-plus-distrib-right*:
assumes *wf1*: *mat nr nc m1*
and *wf2*: *mat nc ncc m2*
and *wf3*: *mat nc ncc m3*
shows *mat-mult nr m1* (*mat-plus m2 m3*) = *mat-plus* (*mat-mult nr m1 m2*) (*mat-mult nr m1 m3*) (**is** *mat-mult nr m1 ?m23* = *mat-plus ?m12 ?m13*)
proof −
note *wf* = *wf1 wf2 wf3*
let *?m1-23* = *mat-mult nr m1 ?m23*
let *?m12-13* = *mat-plus ?m12 ?m13*
from *wf* **have**

```
    wf23: mat nc ncc ?m23 and
    wf12: mat nr ncc ?m12 and
    wf13: mat nr ncc ?m13 and
    wf1-23: mat nr ncc ?m1-23 and
    wf12-13: mat nr ncc ?m12-13 by auto
  show ?thesis
  proof (rule mat-eqI)
    fix i j
    assume i: i < ncc and j: j < nr
    show ?m1-23 ! i ! j = ?m12-13 ! i ! j
      by (unfold mat-mult-index[OF wf1 wf23 j i]
          mat-plus-index[OF wf12 wf13 i j]
          mat-mult-index[OF wf1 wf2 j i]
          mat-mult-index[OF wf1 wf3 j i]
          col-mat-plus[OF wf2 wf3 i],
        rule scalar-vec-plus-distrib-right[OF row[OF wf1 j] col[OF wf2 i] col[OF wf3
i]])
  qed (insert wf, auto)
qed

lemma mat-mult-plus-distrib-left:
  assumes wf1: mat nr nc m1
  and wf2: mat nr nc m2
  and wf3: mat nc ncc m3
  shows mat-mult nr (mat-plus m1 m2) m3 = mat-plus (mat-mult nr m1 m3)
(mat-mult nr m2 m3) (is mat-mult nr ?m12 - = mat-plus ?m13 ?m23)
proof −
  note wf = wf1 wf2 wf3
  let ?m12-3 = mat-mult nr ?m12 m3
  let ?m13-23 = mat-plus ?m13 ?m23
  from wf have
    wf12: mat nr nc ?m12 and
    wf13: mat nr ncc ?m13 and
    wf23: mat nr ncc ?m23 and
    wf12-3: mat nr ncc ?m12-3 and
    wf13-23: mat nr ncc ?m13-23 by auto
  show ?thesis
  proof (rule mat-eqI)
    fix i j
    assume i: i < ncc and j: j < nr
    show ?m12-3 ! i ! j = ?m13-23 ! i ! j
      by (unfold mat-mult-index[OF wf12 wf3 j i]
          mat-plus-index[OF wf13 wf23 i j]
          mat-mult-index[OF wf1 wf3 j i]
          mat-mult-index[OF wf2 wf3 j i]
          row-mat-plus[OF wf1 wf2 j],
           rule scalar-vec-plus-distrib-left[OF row[OF wf1 j] row[OF wf2 j] col[OF
wf3 i]])
  qed (insert wf, auto)
```

qed
end

context *semiring-1*
begin
abbreviation *vec1* :: *nat* ⇒ *nat* ⇒ *'a vec*
where *vec1* ≡ *vec1I zero one*

abbreviation *mat1* :: *nat* ⇒ *'a mat*
where *mat1* ≡ *mat1I zero one*

abbreviation *mat-pow* **where** *mat-pow* ≡ *mat-powI* (*0* :: *'a*) *1* (+) (∗)

lemma *scalar-left-one*: **assumes** *wf*: *vec nn v*
and *i*: *i* < *nn*
shows *scalar-prod* (*vec1 nn i*) *v* = *v* ! *i*
using *assms*
unfolding *vec1I-def vec-def*
proof (*induct nn arbitrary*: *v i*)
case (*Suc n*) **note** *oSuc* = *this*
from *this* **obtain** *a vv* **where** *v*: *v* = *a* # *vv* **and** *lvv*: *length vv* = *n* **by** (*cases v*, *auto*)
show *?case*
proof (*cases i*)
case *0*
thus *?thesis* **using** *scalar-left-zero* **unfolding** *vec0I-def* **by** (*simp add*: *v scalar-prod-cons add.commute*)
next
case (*Suc ii*)
thus *?thesis* **using** *oSuc lvv v* **by** (*auto simp*: *scalar-prod-cons*)
qed
qed *blast*

lemma *scalar-right-one*: **assumes** *wf*: *vec nn v*
and *i*: *i* < *nn*
shows *scalar-prod v* (*vec1 nn i*) = *v* ! *i*
using *assms*
unfolding *vec1I-def vec-def*
proof (*induct nn arbitrary*: *v i*)
case (*Suc n*) **note** *oSuc* = *this*
from *this* **obtain** *a vv* **where** *v*: *v* = *a* # *vv* **and** *lvv*: *length vv* = *n* **by** (*cases v*, *auto*)
show *?case*
proof (*cases i*)
case *0*
thus *?thesis* **using** *scalar-right-zero* **unfolding** *vec0I-def* **by** (*simp add*: *v scalar-prod-cons add.commute*)

next
case (*Suc ii*)
thus *?thesis* **using** *oSuc lvv v* **by** (*auto simp*: *scalar-prod-cons*)
qed
qed *blast*

lemma *mat1-mult-right*: **assumes** *wf*: *mat nr nc m*
shows *mat-mult nr m* (*mat1 nc*) = *m*
proof (*rule mat-eqI*)
fix *i j*
assume *i*: *i* < *nc* **and** *j*: *j* < *nr*
show *mat-mult nr m* (*mat1 nc*) ! *i* ! *j* = *m* ! *i* ! *j*
by (*unfold mat-mult-index*[*OF wf mat1 j i*]
col-mat1[*OF i*]
scalar-right-one[*OF row*[*OF wf j*] *i*]
row-col[*OF wf j i*],
unfold col-def, *simp*)
qed (*insert wf*, *auto*)

lemma *mat1-mult-left*: **assumes** *wf*: *mat nr nc m*
shows *mat-mult nr* (*mat1 nr*) *m* = *m*
proof (*rule mat-eqI*)
fix *i j*
assume *i*: *i* < *nc* **and** *j*: *j* < *nr*
show *mat-mult nr* (*mat1 nr*) *m* ! *i* ! *j* = *m* ! *i* ! *j*
by (*unfold mat-mult-index*[*OF mat1 wf j i*]
row-mat1[*OF j*]
scalar-left-one[*OF col*[*OF wf i*] *j*], *unfold col-def*, *simp*)
qed (*insert wf*, *auto*)
end

declare *vec0*[*simp del*] *mat0*[*simp del*] *vec0-plus*[*simp del*] *plus-vec0*[*simp del*] *plus-mat0*[*simp del*]

2.6 Connection to HOL-Algebra

definition *mat-monoid* :: *nat* ⇒ *nat* ⇒ *'b* ⇒ ((*'a* :: {*plus*,*zero*}) *mat*,*'b*) *monoid-scheme*
where
mat-monoid nr nc b ≡ (|
carrier = *Collect* (*mat nr nc*),
mult = *mat-plus*,
one = *mat0 nr nc*,
... = *b*|)

definition *mat-ring* :: *nat* ⇒ *'b* ⇒ ((*'a* :: *semiring-1*) *mat*,*'b*) *ring-scheme* **where**
mat-ring n b ≡ (|

```
    carrier = Collect (mat n n),
    mult = mat-mult n,
    one = mat1 n,
    zero = mat0 n n,
    add = mat-plus,
    ... = b⦈

lemma mat-monoid: monoid (mat-monoid nr nc b :: (('a :: monoid-add) mat,'b)monoid-scheme)
  by (unfold-locales, auto simp: mat-plus-assoc mat-monoid-def plus-mat0)

lemma mat-group: group (mat-monoid nr nc b :: (('a :: group-add) mat,'b)monoid-scheme)
(is group ?G)
proof −
  interpret monoid ?G by (rule mat-monoid)
  {
    fix m :: 'a mat
    assume wf: mat nr nc m
    let ?m' = mat-map uminus m
    have ∃ m'. mat nr nc m' ∧ mat-plus m' m = mat0 nr nc ∧ mat-plus m m' =
mat0 nr nc
    proof (rule exI[of - ?m'], intro conjI mat-eqI)
      fix i j
      assume i < nc j < nr
    note [simp] = mat-plus-index[OF - - this] mat-map-index[OF - this] mat0-index[OF
this]
      show mat-plus ?m' m ! i ! j = mat0 nr nc ! i ! j using wf by simp
      show mat-plus m ?m' ! i ! j = mat0 nr nc ! i ! j using wf by simp
    qed (auto intro: wf)
  } note Units = this
  show ?thesis
    by (unfold-locales, auto simp: mat-monoid-def Units-def Units)
qed

lemma mat-comm-monoid:
  comm-monoid (mat-monoid nr nc b :: (('a :: comm-monoid-add) mat,'b)monoid-scheme)
(is comm-monoid ?G)
proof −
  interpret monoid ?G by (rule mat-monoid)
  show ?thesis
    by (unfold-locales, insert mat-plus-comm, auto simp: mat-monoid-def)
qed

lemma mat-comm-group:
  comm-group (mat-monoid nr nc b :: (('a :: ab-group-add) mat,'b)monoid-scheme)
(is comm-group ?G)
proof −
  interpret group ?G by (rule mat-group)
  interpret comm-monoid ?G by (rule mat-comm-monoid)
  show ?thesis ..
```

qed

lemma *mat-abelian-monoid*: *abelian-monoid* (*mat-ring n b* :: ((*'a* :: *semiring-1*) *mat*,*'b*)*ring-scheme*)
unfolding *mat-ring-def*
unfolding *abelian-monoid-def* **using** *mat-comm-monoid*[*of n n*, *unfolded mat-monoid-def mat-ring-def*]
by *simp*

lemma *mat-abelian-group*: *abelian-group* (*mat-ring n b* :: ((*'a* :: {*ab-group-add*,*semiring-1*}) *mat*,*'b*)*ring-scheme*)
(**is** *abelian-group ?R*)
proof −
interpret *abelian-monoid ?R* **by** (*rule mat-abelian-monoid*)
show *?thesis*
apply *unfold-locales*
apply (*rule group.Units*)
by (*metis mat-group mat-monoid-def mat-ring-def partial-object.simps*(*1*) *ring.simps*(*1*) *ring.simps*(*2*))
qed

lemma *mat-semiring*: *semiring* (*mat-ring n b* :: ((*'a* :: *semiring-1*) *mat*,*'b*)*ring-scheme*)
(**is** *semiring ?R*)
proof −
interpret *abelian-monoid ?R* **by** (*rule mat-abelian-monoid*)
show *?thesis*
by (*unfold-locales*, *unfold mat-ring-def*, *insert*
mat-mult-assoc mat0-mult-left mat0-mult-right mat1-mult-left mat1-mult-right
mat-mult-plus-distrib-left mat-mult-plus-distrib-right, *auto*)
qed

lemma *mat-ring*: *ring* (*mat-ring n b* :: ((*'a* :: *ring-1*) *mat*,*'b*)*ring-scheme*)
(**is** *ring ?R*)
proof −
interpret *abelian-group ?R* **by** (*rule mat-abelian-group*)
show *?thesis*
by (*unfold-locales*, *unfold mat-ring-def*, *insert*
mat-mult-assoc mat1-mult-left mat1-mult-right mat-mult-plus-distrib-left
mat-mult-plus-distrib-right, *auto*)
qed

lemma *mat-pow-ring-pow*: **assumes** *mat*: *mat n n* (*m* :: (*'a* :: *semiring-1*)*mat*)
shows *mat-pow n m k* = *m* [^]$_{mat\text{-}ring\ n\ b}$ *k*
(**is** - = *m* [^]$_{?C}$ *k*)
proof −
interpret *semiring ?C* **by** (*rule mat-semiring*)
show *?thesis*
by (*induct k*, *auto*, *auto simp*: *mat-ring-def*)
qed

end

References

[1] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[2] A. Koprowski and J. Waldmann. Arctic termination ... below zero. In *Proc. RTA'08*, LNCS 5117, pages 202–216, 2008.

[3] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, LNCS 5674, pages 452–468, 2009.

[4] R. Thiemann and C. Sternagel. Certified polynomial interpretations over matrices and over domains. In *Proc. WST '10*, 2010. To appear.