

Executable Matrix Operations on Matrices of Arbitrary Dimensions

Christian Sternagel and René Thiemann

April 20, 2020

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  $\bigwedge x y. (x, y) \in A \implies P x y$ 
  shows  $\forall (x, y) \in A. P x y$ 
  using assms by auto
```

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

```
lemma infinite-imp-many-elems:
  infinite  $A \implies \exists xs. \text{set } xs \subseteq A \wedge \text{length } xs = n \wedge \text{distinct } xs$ 
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  $\text{set } xs \subseteq A - \{x\}$  and  $\text{length } xs = n$ 
  and  $\text{distinct } xs$  by auto
  with  $x$  show ?case by (intro exI[of -  $x \# xs$ ], auto)
qed auto
```

```
lemma inf-pigeonhole-principle:
  assumes  $\forall k::\text{nat}. \exists i < n::\text{nat}. f k i$ 
  shows  $\exists i < n. \forall k. \exists k' \geq k. f k' i$ 
proof -
  have  $n\text{fin}: \sim \text{finite } (\text{UNIV} :: \text{nat set})$  by auto
  have  $\text{fin}: \text{finite } (\{i. i < n\})$  by auto
  from pigeonhole-infinite-rel[OF  $n\text{fin } \text{fin}$ ] assms
  obtain  $i$  where  $i: i < n$  and  $n\text{fin}: \neg \text{finite } \{a. f a i\}$  by auto
  show ?thesis
  proof (intro exI conjI, rule  $i$ , intro allI)
    fix  $k$ 
    have  $\text{finite } \{a. f a i \wedge a < k\}$  by auto
    with  $n\text{fin}$  have  $\neg \text{finite } (\{a. f a i\} - \{a. f a i \wedge a < k\})$  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:  $\text{map } f [0 ..< \text{Suc } n] = f 0 \# \text{map } (\lambda i. f (\text{Suc } i)) [0 ..< n]$ 
  by (induct  $n$  arbitrary:  $f$ , auto)
```

lemma *map-upt-add*: $\text{map } f [0 \dots n + m] = \text{map } f [0 \dots n] @ \text{map } (\lambda i. f (i + n)) [0 \dots m]$
proof (*induct n arbitrary: f*)
 case (*Suc n f*)
 have $\text{map } f [0 \dots \text{Suc } n + m] = \text{map } f [0 \dots \text{Suc } (n+m)]$ **by** *simp*
 also have $\dots = f 0 \# \text{map } (\lambda i. f (\text{Suc } i)) [0 \dots n + m]$ **unfolding** *map-upt-Suc*
 ..
 finally show *?case unfolding Suc map-upt-Suc by simp*
qed *simp*

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

lemma *all-Suc-conv*:
 $(\forall i < \text{Suc } n. P i) \longleftrightarrow P 0 \wedge (\forall i < n. P (\text{Suc } i))$ (**is** $?l = ?r$)
proof
 assume $?l$ **thus** $?r$ **by** *auto*
next
 assume $?r$ **show** $?l$
 proof (*intro allI impI*)
 fix i
 assume $i < \text{Suc } n$
 with $\langle ?r \rangle$ **show** $P i$ **by** (*cases i, auto*)
 qed
qed

lemma *ex-Suc-conv*:
 $(\exists i < \text{Suc } n. P i) \longleftrightarrow P 0 \vee (\exists i < n. P (\text{Suc } i))$ (**is** $?l = ?r$)
 using *all-Suc-conv* [*of n λi. ¬ P i*] **by** *blast*

fun *sorted-list-subset* :: $'a :: \text{linorder list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ option}$ **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 $bs = \text{None}$) = (*set* $as \subseteq \text{set } bs$)

```

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:  $\bigwedge a'. a' \in \text{set } as \implies a \leq a'$  by (auto)
  from 1(4) have sbs: sorted bs and b:  $\bigwedge b'. b' \in \text{set } bs \implies b \leq b'$  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  $\implies$  zip xs ys = map ( $\lambda i. (xs ! i, ys ! i)$ ) [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  $\forall i < \text{length } xs. f (xs ! i) = g (ys ! i)$ 
  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  $\forall i < \text{length } xs. f (xs ! i) = g (ys ! i)$ 
    proof (intro allI impI)
      fix i assume  $i < \text{length } xs$  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:  $\llbracket \bigwedge x. x \in \text{set } xs \implies x = 0 \rrbracket \implies \text{sum-list } xs = 0$ 
  by (induct xs, auto)

```

```

lemma foldr-foldr-concat:  $\text{foldr } (f \text{ foldr } f) m a = \text{foldr } f (\text{concat } m) a$ 
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  $f :: 'b \Rightarrow 'c \Rightarrow 'a :: \text{comm-monoid-add}$  and  $g \text{ as } bs$ 
  shows  $\text{sum-list } (\text{concat } (\text{map } (\lambda i. \text{map } (\lambda j. f i j + g i j) \text{ as}) bs))$ 
    =  $\text{sum-list } (\text{concat } (\text{map } (\lambda i. \text{map } (\lambda j. f i j) \text{ as}) bs)) +$ 
       $\text{sum-list } (\text{concat } (\text{map } (\lambda i. \text{map } (\lambda j. g i j) \text{ as}) bs))$ 
proof (induct bs)
  case Nil thus ?case by simp
next
  case (Cons b bs)
  have  $\text{id}: (\sum j \leftarrow \text{as}. f b j + g b j) = \text{sum-list } (\text{map } (f b) \text{ as}) + \text{sum-list } (\text{map } (g$ 
 $b) \text{ as})$ 
  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 ::  $\text{nat list} \Rightarrow \text{nat}$  where
  max-list [] = 0
| max-list (x # xs) = max x (max-list xs)

```

```

lemma max-list:  $x \in \text{set } xs \implies x \leq \text{max-list } xs$ 
  by (induct xs) auto

```

```

lemma max-list-mem:  $xs \neq [] \implies \text{max-list } xs \in \text{set } xs$ 

```

```

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  $\bigwedge n. \llbracket P\ n; n \in \{l..<u\} \rrbracket \implies P\ (Suc\ n)$ 
  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 (Max1)

```

```

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

```

```

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

```

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

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

lemma *max0-id-pos*: **assumes** $x: x \succeq 0$ **and** $wf: x \in \text{carrier } R$
shows $\text{Max } 0 \ x = x$ **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* :: $'a \text{ itself} \Rightarrow 'b \Rightarrow ('a :: \{\text{plus, times, one, zero}\}, 'b) \text{ring-scheme}$
where
class-semiring - b $\equiv (\mid \text{carrier} = \text{UNIV}, \text{mult} = (*), \text{one} = 1, \text{zero} = 0, \text{add} = (+), \dots = b)$

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* :: $'a \text{ itself} \Rightarrow ('a :: \text{ordered-semiring-1} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow ('a, 'b) \text{ordered-semiring-scheme}$ **where**
class-ordered-semiring a gt b $\equiv \text{class-semiring } a \ (\mid$
ordered-semiring.geq = (\geq),
gt = *gt*,
max = *max*,
 $\dots = b)$

lemma *class-ordered-semiring: assumes order-pair* (*gt* :: ($'a :: \text{ordered-semiring-1} \Rightarrow 'a \Rightarrow \text{bool}$)) *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)) (\succ) 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)) (\succ) 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* \Rightarrow 'x *vec* \Rightarrow *bool*

where *vec* n x = (*length* x = n)

definition *mat* :: *nat* \Rightarrow *nat* \Rightarrow 'a *mat* \Rightarrow *bool* **where**

mat nr nc m = (*length* m = nc \wedge *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 $vec0I :: 'a \Rightarrow nat \Rightarrow 'a\ vec$ **where**
 $vec0I\ ze\ n = replicate\ n\ ze$

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

definition $vec1I :: 'a \Rightarrow 'a \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ vec$
where $vec1I\ ze\ on\ n\ i \equiv replicate\ i\ ze\ @\ on\ \# replicate\ (n - 1 - i)\ ze$

definition $mat1I :: 'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'a\ mat$
where $mat1I\ ze\ on\ n \equiv map\ (vec1I\ ze\ on\ n)\ [0 ..< n]$

definition $vec-plusI :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ vec \Rightarrow 'a\ vec \Rightarrow 'a\ vec$ **where**
 $vec-plusI\ pl\ v\ w = map\ (\lambda\ xy.\ pl\ (fst\ xy)\ (snd\ xy))\ (zip\ v\ w)$

definition $mat-plusI :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ mat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$
where $mat-plusI\ pl\ m1\ m2 = map\ (\lambda\ uv.\ vec-plusI\ pl\ (fst\ uv)\ (snd\ uv))\ (zip\ m1\ m2)$

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

definition $row :: 'a\ mat \Rightarrow nat \Rightarrow 'a\ vec$
where $row\ m\ i \equiv map\ (\lambda\ w.\ w\ !\ i)\ m$

definition $col :: 'a\ mat \Rightarrow nat \Rightarrow 'a\ vec$
where $col\ m\ i \equiv m\ !\ i$

fun $transpose :: nat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$
where $transpose\ nr\ [] = replicate\ nr\ []$
 $| transpose\ nr\ (v\ \# m) = map\ (\lambda\ (vi,mi).\ (vi\ \# mi))\ (zip\ v\ (transpose\ nr\ m))$

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

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

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

definition $sub\text{-vec} :: nat \Rightarrow 'a\ vec \Rightarrow 'a\ vec$
where $sub\text{-vec} = take$

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

definition $vec\text{-map} :: ('a \Rightarrow 'a) \Rightarrow 'a\ vec \Rightarrow 'a\ vec$
where $vec\text{-map} = map$

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

2.3 algorithms preserve dimensions

lemma $vec0[simp,intro]: vec\ nr\ (vec0I\ ze\ nr)$
by $(simp\ add: vec\text{-def}\ vec0I\text{-def})$

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

lemma $mat0[simp,intro]: mat\ nr\ nc\ (mat0I\ ze\ nr\ nc)$
unfolding $mat\text{-def}\ mat0I\text{-def}$
using $replicate\text{-prop}[of\ vec\ nr\ vec0I\ ze\ nr\ nc]$ **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]*: $\llbracket \text{vec } nr \ u; \text{vec } nr \ v \rrbracket \implies \text{vec } nr \ (\text{vec-plusI } pl \ u \ v)$
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* \implies *vec nr (vec-map f u)*
unfolding *vec-map-def* *vec-def*
by *auto*

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

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

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

```

lemma concat-mat: mat nr nc m  $\implies$ 
  concat m = [ m ! i ! j. i  $\leftarrow$  [0 ..< nc], j  $\leftarrow$  [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 ( $\lambda$  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  $\langle$ mat nr nc (v # m) $\rangle$  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 $mat\ sr\ sc\ (sub\text{-}mat\ sr\ sc\ m)$
using $assms\ in\text{-}set\text{-}takeD[of\text{-}\ sc\ m]\ sub\text{-}vec[OF\text{-}\ sr]$ **unfolding** $mat\text{-}def\ sub\text{-}mat\text{-}def$
by $auto$

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

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

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

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

lemma $vec1\text{-}index$: **assumes** $j: j < n$
shows $vec1I\ ze\ on\ n\ i\ !\ j = (if\ i = j\ then\ on\ else\ ze)\ (is\ - =\ ?r)$
unfolding $vec1I\text{-}def$

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** ($simp\ add: nth\text{-}append$)
next
case $False$
show $?thesis$
proof ($cases\ j < i$)
case $True$
thus $?thesis$ **by** ($simp\ add: nth\text{-}append$)
next
case $False$
with $\langle j \neq i \rangle$ **have** gt : $j > i$ **by** $auto$
from $this$ **have** $\exists k. j = i + Suc\ k$ **by** $arith$
from $this$ **obtain** k **where** $k: j = i + Suc\ k$ **by** $auto$
with j **show** $?thesis$ **by** ($simp\ add: nth\text{-}append$)
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  $(m1 = m2) = (\forall i < nc. \forall j < nr. m1 ! i ! j = m2 ! i ! j)$  (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  $\langle ?r \rangle i$  show  $\bigwedge j. j < length (m1 ! i) \implies m1 ! i ! j = m2 ! i ! j$ 
      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:  $\bigwedge i j. i < nc \implies j < nr \implies m1 ! i ! j = m2 ! i ! j$ 
  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  $(v1 = v2) = (\forall i < n. v1 ! i = v2 ! i)$  (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  $\langle ?r \rangle wf1$  show  $\bigwedge i. i < length v1 \implies v1 ! i = v2 ! i$  unfolding vec-def
  by simp
  qed
qed
qed

lemma vec-eqI:

```

```

assumes wf1: vec n v1
and wf2: vec n v2
and id:  $\bigwedge i. i < n \implies v1 ! i = v2 ! i$ 
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 ( $\lambda j. m ! i ! j$ ) [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 ( $\lambda j. m ! j ! i$ ) [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 ( $\lambda j. row m i ! j$ ) [0 ..< nc] unfolding id by simp
also have ... = map ( $\lambda j. m ! j ! i$ ) [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) = ( $\forall i < nr. row m1 i = row m2 i$ ) (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  $\text{length } (m1 ! i) = \text{length } (m2 ! i)$  using wf1 wf2 i unfolding mat-def
by (auto simp: vec-def)
  next
    show  $m1 ! i ! j = m2 ! i ! j$  if  $j: j < \text{length } (m1 ! i)$  for  $j$ 
    proof –
      from  $i j wf1$  have  $i1: i < nc$  and  $j1: j < nr$  unfolding mat-def by
      (auto simp: vec-def)
      from  $\langle ?r \rangle j1$  have  $\text{col } m1 i ! j = \text{col } m2 i ! j$ 
      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  $wf1: \text{mat } nr \ nc \ m1$ 
  and  $wf2: \text{mat } nr \ nc \ m2$ 
  and  $id: \bigwedge i. i < nr \implies \text{row } m1 \ i = \text{row } m2 \ i$ 
  shows  $m1 = m2$ 
  unfolding mat-row-eq[OF wf1 wf2] using id by auto

```

```

lemma row-transpose-is-col[simp]: assumes  $wf: \text{mat } nr \ nc \ m$ 
  and  $i: i < nc$ 
  shows  $\text{row } (\text{transpose } nr \ m) \ i = \text{col } m \ i$ 
proof –
  have  $len: \text{length } (\text{row } (\text{transpose } nr \ m) \ i) = \text{length } (\text{col } m \ i)$ 
  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  $j < \text{length } (\text{row } (\text{transpose } nr \ m) \ i)$ 
    hence  $j: j < nr$  using transpose[OF wf] wf i unfolding row-def col-def mat-def
by (auto simp: vec-def)
    show  $\text{row } (\text{transpose } nr \ m) \ i ! j = \text{col } m \ i ! j$ 
    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 $matT\text{-}vec\text{-}multI\ ze\ pl\ ti\ m\ v\ !\ i = scalar\text{-}prodI\ ze\ pl\ ti\ (col\ m\ i)\ v$
unfolding $matT\text{-}vec\text{-}multI\text{-}def$ **using** $assms$ **unfolding** $mat\text{-}def\ col\text{-}def$ **by** $(auto\ simp:\ vec\text{-}def)$

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

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

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

lemma $row\text{-}mat\text{-}mult\text{-}index$:
assumes $wf1:\ mat\ nr\ n\ m1$

and $wf2: mat\ n\ nc\ m2$
and $i: i < nr$
shows $row\ (mat-multI\ ze\ pl\ ti\ nr\ m1\ m2)\ i = map\ (\lambda\ j.\ scalar-prodI\ ze\ pl\ ti\ (row\ m1\ i)\ (col\ m2\ j))\ [0\ ..<\ nc]\ (is\ row\ ?l\ i = ?r)$
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** $\dots = map\ f\ (m\ !\ i)\ !\ j$ **using** i **by** $auto$
also **have** $\dots = f\ (m\ !\ i\ !\ j)$ **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 ≤ nr
and sc: sc ≤ 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 *

```

$x2 + \text{scalar-prod } w1 \ w2$.
have $\text{scalar-prod } ((y1 \ \# \ w1) \ @ \ [x1]) \ (v2 \ @ \ [x2]) =$
 $(y1 * y2 + x1 * x2) + \text{scalar-prod } w1 \ w2$ **by** (*simp add: scalar-prod-cons v2*
rec add.assoc)
also have $\dots = (x1 * x2 + y1 * y2) + \text{scalar-prod } w1 \ w2$ **using** *add.commute*[of
 $x1 * x2$] **by** *simp*
also have $\dots = x1 * x2 + (\text{scalar-prod } (y1 \ \# \ w1) \ v2)$ **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 $\text{scalar-prod } (\text{map } (\lambda k. \text{scalar-prod } r \ (\text{col } m \ k)) \ [0..\<nc]) \ c = \text{scalar-prod}$
 $r \ (\text{map } (\lambda k. \text{scalar-prod } (\text{row } m \ k) \ c) \ [0..\<nr])$
using *wfm wfc*
unfolding *col-def*
proof (*induct m arbitrary: nc c*)
case *Nil*
hence $nc = 0$ **unfolding** *mat-def* **by** (*auto*)
from *wfr* **have** $nr = \text{length } r$ **unfolding** *vec-def* **by** *auto*
let *?term = $\lambda r :: 'a \ \text{vec}. \ \text{zip } r \ (\text{map } (\lambda k. \ \text{zero}) \ [0..\<\text{length } r])$*
let *?fun = $\lambda (x,y). \ \text{plus } (\text{times } x \ y)$*
have $\text{foldr } ?fun \ (?term \ r) \ \text{zero} = \text{zero}$
proof (*induct r, simp*)
case (*Cons d r*)
have $\text{foldr } ?fun \ (?term \ (d \ \# \ r)) \ \text{zero} = \text{foldr } ?fun \ ((d, \ \text{zero}) \ \# \ ?term \ r) \ \text{zero}$
by (*simp only: map-replicate-trivial, simp*)
also have $\dots = \text{zero}$ **using** *Cons* **by** *simp*
finally show *?case* .
qed
hence $\text{zero} = \text{foldr } ?fun \ (\text{zip } r \ (\text{map } (\lambda k. \ \text{zero}) \ [0..\<nr])) \ \text{zero}$ **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** $nc = \text{Suc } ncc$ **and** *wf: mat nr ncc m* **unfolding**
mat-def **by** (*auto simp: vec-def*)
from *nc* (*vec nc c*) **obtain** *a cc* **where** $c = a \ \# \ cc$ **and** *wfc: vec ncc cc*
unfolding *vec-def* **by** (*cases c, auto*)
have *rec: scalar-prod* ($\text{map } (\lambda k. \ \text{scalar-prod } r \ (m \ ! \ k)) \ [0..\<ncc]) \ cc = \text{scalar-prod}$
 $r \ (\text{map } (\lambda k. \ \text{scalar-prod } (\text{row } m \ k) \ cc) \ [0..\<nr])$
by (*rule Cons, rule wf, rule wfc*)
have *id: map* ($\lambda k. \ \text{scalar-prod } r \ ((v \ \# \ m) \ ! \ k)) \ [0..\<\text{Suc } ncc] = \text{scalar-prod } r \ v$
 $\ \# \ \text{map } (\lambda k. \ \text{scalar-prod } r \ (m \ ! \ k)) \ [0..\<ncc]$ **by** (*induct ncc, auto*)
from *wfr* **have** $nr = \text{length } r$ **unfolding** *vec-def* **by** *auto*

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

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 ?recl (times b ?uc))* **by** (*simp only: rec*)
also have ... = *plus ?bca (plus (times b ?uc) ?recl)* **by** (*simp add: add commute*)
also have ... = *plus (times b (plus (times c a) ?uc)) ?recl* **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)) ?recl* .
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])

$row\text{-}mat\text{-}mult\text{-}index[OF\ wf1\ wf2\ j]$
 $col\text{-}mat\text{-}mult\text{-}index[OF\ wf2\ wf3\ i]$
 $scalar\text{-}product\text{-}assoc[OF\ wf2\ row[OF\ wf1\ j]\ col[OF\ wf3\ i]],\ simp)$

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 $mat\text{-}mult\ n\ (mat\text{-}mult\ n\ m1\ m2)\ m3 = mat\text{-}mult\ n\ m1\ (mat\text{-}mult\ n\ m2\ m3)$

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 $mat\text{-}mult\ nr\ (mat0\ nr\ nc)\ m = (mat0\ nr\ ncc)$

proof (*rule mat-eqI*)

fix *i j*

assume *i: i < ncc and j: j < nr*

show $mat\text{-}mult\ nr\ (mat0\ nr\ nc)\ m\ !\ i\ !\ j = mat0\ nr\ ncc\ !\ i\ !\ j$

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 $mat\text{-}mult\ nr\ m\ (mat0\ nc\ ncc) = (mat0\ nr\ ncc)$

proof (*rule mat-eqI*)

fix *i j*

assume *i: i < ncc and j: j < nr*

show $mat\text{-}mult\ nr\ m\ (mat0\ nc\ ncc)\ !\ i\ !\ j = mat0\ nr\ ncc\ !\ i\ !\ j$

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 $\text{scalar-prod } u \ (\text{vec-plus } v \ w) = \text{plus } (\text{scalar-prod } u \ v) \ (\text{scalar-prod } u \ w)$
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** $lu: \text{length } uu = n$ **and** $lv: \text{length } vv = n$ **and** $lw: \text{length } ww = n$ **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 $\text{scalar-prod } (\text{vec-plus } u \ v) \ w = \text{plus } (\text{scalar-prod } u \ w) \ (\text{scalar-prod } v \ w)$
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** $lu: \text{length } uu = n$ **and** $lv: \text{length } vv = n$ **and** $lw: \text{length } ww = n$ **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 $\text{mat-mult } nr \ m1 \ (\text{mat-plus } m2 \ m3) = \text{mat-plus } (\text{mat-mult } nr \ m1 \ m2)$
 $(\text{mat-mult } nr \ m1 \ m3) \ (\text{is } \text{mat-mult } nr \ m1 \ ?m23 = \text{mat-plus } ?m12 \ ?m13)$
proof –
note $wf = wf1 \ wf2 \ wf3$
let $?m1-23 = \text{mat-mult } nr \ m1 \ ?m23$
let $?m12-13 = \text{mat-plus } ?m12 \ ?m13$

```

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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow 'b \Rightarrow ((\text{'a} :: \{\text{plus, zero}\}) \text{ mat}, 'b) \text{ monoid-scheme}$
where

```

mat-monoid nr nc b  $\equiv$  (
  carrier = Collect (mat nr nc),
  mult = mat-plus,
  one = mat0 nr nc,
  ... = b)

```

definition *mat-ring* :: $\text{nat} \Rightarrow 'b \Rightarrow ((\text{'a} :: \text{semiring-1}) \text{ mat}, 'b) \text{ ring-scheme}$ **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-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.