Executable Matrix Operations on Matrices of Arbitrary Dimensions

Christian Sternagel and René Thiemann

February 6, 2026

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] 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 $\forall (x,\ y){\in}A.\ P\ x\ y$
⟨*proof*⟩

lemma *infinite-imp-elem*: $\neg\ \mathit{finite}\ A \Longrightarrow \exists\ x.\ x \in A$
⟨*proof*⟩

lemma *infinite-imp-many-elems*:
$\mathit{infinite}\ A \Longrightarrow \exists\ \mathit{xs}.\ \mathit{set}\ \mathit{xs} \subseteq A \wedge \mathit{length}\ \mathit{xs} = n \wedge \mathit{distinct}\ \mathit{xs}$
⟨*proof*⟩

lemma *inf-pigeonhole-principle*:
assumes $\forall\, k{::}\mathit{nat}.\ \exists\, i{<}n{::}\mathit{nat}.\ f\ k\ i$
shows $\exists\, i{<}n.\ \forall\, k.\ \exists\, k'{\geq}k.\ f\ k'\ i$
⟨*proof*⟩

lemma *map-upt-Suc*: $\mathit{map}\ f\ [0\ ..{<}\ \mathit{Suc}\ n] = f\ 0\ \#\ \mathit{map}\ (\lambda\ i.\ f\ (\mathit{Suc}\ i))\ [0\ ..{<}\ n]$
⟨*proof*⟩

lemma *map-upt-add*: $\mathit{map}\ f\ [0\ ..{<}\ n + m] = \mathit{map}\ f\ [0\ ..{<}\ n]\ @\ \mathit{map}\ (\lambda\ i.\ f\ (i + n))\ [0\ ..{<}\ m]$
⟨*proof*⟩

lemma *map-upt-split*: **assumes** *i*: $i < n$
shows $\mathit{map}\ f\ [0\ ..{<}\ n] = \mathit{map}\ f\ [0\ ..{<}\ i]\ @\ f\ i\ \#\ \mathit{map}\ (\lambda\ j.\ f\ (j + \mathit{Suc}\ i))\ [0\ ..{<}\ n - \mathit{Suc}\ i]$
⟨*proof*⟩

lemma *all-Suc-conv*:
$(\forall\, i{<}\mathit{Suc}\ n.\ P\ i) \longleftrightarrow P\ 0 \wedge (\forall\, i{<}n.\ P\ (\mathit{Suc}\ i))$ (**is** *?l = ?r*)
⟨*proof*⟩

lemma *ex-Suc-conv*:
$(\exists\, i{<}\mathit{Suc}\ n.\ P\ i) \longleftrightarrow P\ 0 \vee (\exists\, i{<}n.\ P\ (\mathit{Suc}\ i))$ (**is** *?l = ?r*)
⟨*proof*⟩

fun *sorted-list-subset* :: *'a :: linorder list ⇒ 'a list ⇒ 'a option* **where**
sorted-list-subset $(a\ \#\ \mathit{as})\ (b\ \#\ \mathit{bs}) =$
$(\mathit{if}\ a = b\ \mathit{then}\ \mathit{sorted\text{-}list\text{-}subset}\ \mathit{as}\ (b\ \#\ \mathit{bs})$

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)*
⟨*proof*⟩

lemma *zip-nth-conv*: *length xs = length ys ⟹ zip xs ys = map (λ i. (xs ! i, ys ! i)) [0 ..< length ys]*
⟨*proof*⟩

lemma *nth-map-conv*:
assumes *length xs = length ys*
and *∀ i<length xs. f (xs ! i) = g (ys ! i)*
shows *map f xs = map g ys*
⟨*proof*⟩

lemma *sum-list-0*: *⟦⋀ x. x ∈ set xs ⟹ x = 0⟧ ⟹ sum-list xs = 0*
⟨*proof*⟩

lemma *foldr-foldr-concat*: *foldr (foldr f) m a = foldr f (concat m) a*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *max-list-mem*: *xs ≠ [] ⟹ max-list xs ∈ set xs*
⟨*proof*⟩

lemma *max-list-set*: *max-list xs = (if set xs = {} then 0 else (THE x. x ∈ set xs ∧ (∀ y ∈ set xs. y ≤ x)))*
⟨*proof*⟩

lemma *max-list-eq-set*: *set xs = set ys ⟹ max-list xs = max-list ys*
⟨*proof*⟩

lemma *all-less-two*: $(\forall\ i < \mathit{Suc}\ (\mathit{Suc}\ 0).\ P\ i) = (P\ 0 \wedge P\ (\mathit{Suc}\ 0))$ (**is** $?l = ?r$)
⟨*proof*⟩

Induction over a finite set of natural numbers.

lemma *bound-nat-induct*[*consumes 1*]:
assumes $n \in \{l..u\}$ **and** $P\ l$ **and** $\bigwedge n.\ [\![P\ n;\ n \in \{l..{<}u\}]\!] \Longrightarrow P\ (\mathit{Suc}\ n)$
shows $P\ n$
⟨*proof*⟩

end

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

record $'a$ *ordered-semiring* $=$ $'a$ *ring* $+$
geq :: $'a \Rightarrow 'a \Rightarrow \mathit{bool}$ (**infix** ‹$\succeq_{\imath}$› *50*)
gt :: $'a \Rightarrow 'a \Rightarrow \mathit{bool}$ (**infix** ‹$\succ_{\imath}$› *50*)
max :: $'a \Rightarrow 'a \Rightarrow 'a$ (‹$\mathit{Max}_{\imath}$›)

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

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

and *wf-max*[*simp*, *intro*]: *x* ∈ *carrier R* ⟹ *y* ∈ *carrier R* ⟹ *Max x y* ∈ *carrier R*
and *one-geq-zero*: **1** ⪰ **0**
begin
lemma *max-ge-right*: **assumes** *x*: *x* ∈ *carrier R* **and** *y*: *y* ∈ *carrier R* **shows** *Max x y* ⪰ *y*
⟨*proof*⟩

lemma *wf-max0*: *x* ∈ *carrier R* ⟹ *Max* **0** *x* ∈ *carrier R* ⟨*proof*⟩

lemma *max0-id-pos*: **assumes** *x*: *x* ⪰ **0** **and** *wf*: *x* ∈ *carrier R*
shows *Max* **0** *x* = *x* ⟨*proof*⟩
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*)
⟨*proof*⟩

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*⟩

lemma (**in** *one-mono-ordered-semiring-1*) *class-ordered-semiring*:
ordered-semiring
(*class-ordered-semiring* (*TYPE*(*'a*)) (≻) *b*)
⟨*proof*⟩

lemma (**in** *both-mono-ordered-semiring-1*) *class-ordered-semiring*:
ordered-semiring

(*class-ordered-semiring* (*TYPE*(*'a*)) (≻) *b*)
⟨*proof*⟩

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 ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a mat
⇒ 'a vec ⇒ '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 ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ nat ⇒ 'a mat
⇒ 'a mat ⇒ 'a mat
where mat-multI ze pl ti nr m1 m2 ≡ map (matT-vec-multI ze pl ti (transpose nr
m1)) m2
```

fun *mat-powI* :: *'a* ⇒ *'a* ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a mat* ⇒ *nat* ⇒ *'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* ⇒ *'a vec* ⇒ *'a vec*
where *sub-vec* = *take*

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

definition *vec-map* :: (*'a* ⇒ *'a*) ⇒ *'a vec* ⇒ *'a vec*
where *vec-map* = *map*

definition *mat-map* :: (*'a* ⇒ *'a*) ⇒ *'a mat* ⇒ *'a mat*
where *mat-map f* = *map* (*vec-map f*)

2.3 algorithms preserve dimensions

lemma *vec0*[*simp*,*intro*]: *vec nr* (*vec0I ze nr*)
⟨*proof*⟩

lemma *replicate-prop*:
assumes *P x*
shows ∀ *y*∈*set* (*replicate n x*). *P y*
⟨*proof*⟩

lemma *mat0*[*simp*,*intro*]: *mat nr nc* (*mat0I ze nr nc*)
⟨*proof*⟩

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

lemma *mat1*[*simp*,*intro*]: *mat nr nr* (*mat1I ze on nr*)
⟨*proof*⟩

lemma *vec-plus*[*simp*,*intro*]: ⟦*vec nr u*; *vec nr v*⟧ ⟹ *vec nr* (*vec-plusI pl u v*)
⟨*proof*⟩

lemma *mat-plus*[*simp*,*intro*]: **assumes** *mat nr nc m1* **and** *mat nr nc m2* **shows** *mat nr nc* (*mat-plusI pl m1 m2*)
⟨*proof*⟩

lemma *vec-map*[*simp*,*intro*]: *vec nr u* ⟹ *vec nr* (*vec-map f u*)
⟨*proof*⟩

lemma *mat-map*[*simp*,*intro*]: *mat nr nc m* ⟹ *mat nr nc* (*mat-map f m*)
⟨*proof*⟩

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* ⟹
  concat m = [*m* ! *i* ! *j*. *i* ← [*0* ..< *nc*], *j* ← [*0* ..< *nr*]]
⟨*proof*⟩

lemma *row*: **assumes** *mat nr nc m*
  and *i* < *nr*
  shows *vec nc* (*row m i*)
  ⟨*proof*⟩

lemma *col*: **assumes** *mat nr nc m*
  and *i* < *nc*
  shows *vec nr* (*col m i*)
  ⟨*proof*⟩

lemma *transpose*[*simp*,*intro*]: **assumes** *mat nr nc m*
  shows *mat nc nr* (*transpose nr m*)
⟨*proof*⟩

lemma *matT-vec-multI*: **assumes** *mat nr nc m*
  shows *vec nc* (*matT-vec-multI ze pl ti m v*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mat-pow*[*simp*,*intro*]: **assumes** *mat n n m*
  shows *mat n n* (*mat-powI ze on pl ti n m i*)
⟨*proof*⟩

lemma *sub-vec*[*simp*,*intro*]: **assumes** *vec nr v* **and** *sd* ≤ *nr*
  shows *vec sd* (*sub-vec sd v*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mat0-row*[*simp*]: **assumes** *i* < *nr*
  shows *row* (*mat0I ze nr nc*) *i* = *vec0I ze nc*
⟨*proof*⟩

lemma *mat0-col*[*simp*]: **assumes** *i* < *nc*
  shows *col* (*mat0I ze nr nc*) *i* = *vec0I ze nr*
⟨*proof*⟩

lemma *vec1-index*: **assumes** *j*: *j* < *n*
  shows *vec1I ze on n i* ! *j* = (*if i* = *j then on else ze*) (**is** - = *?r*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *row-col*: **assumes** *mat nr nc m*
  and *i* < *nr* **and** *j* < *nc*
  shows *row m i* ! *j* = *col m j* ! *i*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *scalar-prod-cons*:
scalar-prodI ze pl ti (*a* # *as*) (*b* # *bs*) = *pl* (*ti a b*) (*scalar-prodI ze pl ti as bs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *col-mat1*: **assumes** *i* < *nr*
shows *col* (*mat1I ze on nr*) *i* = *vec1I ze on nr i*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *transpose-mat1*: *transpose nr* (*mat1I ze on nr*) = (*mat1I ze on nr*) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *row-mat1*: **assumes** *i*: *i* < *nr*
shows *row* (*mat1I ze on nr*) *i* = *vec1I ze on nr i*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩
end

context *ab-semigroup-add*
begin
lemma *vec-plus-comm*: *vec-plus x y* = *vec-plus y x*
⟨*proof*⟩

lemma *mat-plus-comm*: *mat-plus m1 m2* = *mat-plus m2 m1*
⟨*proof*⟩
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*
⟨*proof*⟩

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

lemma *plus-mat0*[*simp*]: **assumes** *wf*: *mat nr nc m* **shows** *mat-plus m* (*mat0 nr nc*) = *m* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *mat0-plus*[*simp*]: **assumes** *wf*: *mat nr nc m* **shows** *mat-plus* (*mat0 nr nc*) *m* = *m* (**is** *?l* = *?r*)
⟨*proof*⟩
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*⟩

lemma *scalar-prod-last*: **assumes** *length v1 = length v2*
shows *scalar-prod (v1 @ [x1]) (v2 @ [x2]) = x1 * x2 + scalar-prod v1 v2*
⟨*proof*⟩

lemma *scalar-product-assoc*:
assumes *wfm*: *mat nr nc m*
and *wfr*: *vec nr r*
and *wfc*: *vec nc c*
shows *scalar-prod (map (λk. scalar-prod r (col m k)) [0..<nc]) c = scalar-prod r (map (λk. scalar-prod (row m k) c) [0..<nr])*
⟨*proof*⟩

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*⟩

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)*
⟨*proof*⟩

lemma *scalar-left-zero*: *scalar-prod (vec0 nn) v = zero*
⟨*proof*⟩

lemma *scalar-right-zero*: *scalar-prod v (vec0 nn) = zero*
⟨*proof*⟩

lemma *mat0-mult-left*: **assumes** *wf*: *mat nc ncc m*
shows *mat-mult nr (mat0 nr nc) m = (mat0 nr ncc)*
⟨*proof*⟩

lemma *mat0-mult-right*: **assumes** *wf*: *mat nr nc m*
shows *mat-mult nr m (mat0 nc ncc) = (mat0 nr ncc)*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩
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*
⟨*proof*⟩

lemma *scalar-right-one*: **assumes** *wf*: *vec nn v*
and *i*: *i* < *nn*
shows *scalar-prod v* (*vec1 nn i*) = *v* ! *i*
⟨*proof*⟩

lemma *mat1-mult-right*: **assumes** *wf*: *mat nr nc m*
  shows *mat-mult nr m* (*mat1 nc*) = *m*
⟨*proof*⟩

lemma *mat1-mult-left*: **assumes** *wf*: *mat nr nc m*
  shows *mat-mult nr* (*mat1 nr*) *m* = *m*
⟨*proof*⟩
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*)
  ⟨*proof*⟩

lemma *mat-group*: *group* (*mat-monoid nr nc b* :: ((′*a* :: *group-add*) *mat*,′*b*)*monoid-scheme*)
(**is** *group ?G*)
⟨*proof*⟩

lemma *mat-comm-monoid*:
  comm-monoid (*mat-monoid nr nc b* :: ((′*a* :: *comm-monoid-add*) *mat*,′*b*)*monoid-scheme*)
(**is** *comm-monoid ?G*)
⟨*proof*⟩

lemma *mat-comm-group*:
  comm-group (*mat-monoid nr nc b* :: ((′*a* :: *ab-group-add*) *mat*,′*b*)*monoid-scheme*)

(**is** *comm-group ?G*)
⟨*proof*⟩

lemma *mat-abelian-monoid*: *abelian-monoid* (*mat-ring n b* :: ((*'a* :: *semiring-1*) *mat*,*'b*)*ring-scheme*)
⟨*proof*⟩

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*⟩

lemma *mat-semiring*: *semiring* (*mat-ring n b* :: ((*'a* :: *semiring-1*) *mat*,*'b*)*ring-scheme*)
(**is** *semiring ?R*)
⟨*proof*⟩

lemma *mat-ring*: *ring* (*mat-ring n b* :: ((*'a* :: *ring-1*) *mat*,*'b*)*ring-scheme*)
(**is** *ring ?R*)
⟨*proof*⟩

lemma *mat-pow-ring-pow*: **assumes** *mat*: *mat n n* (*m* :: (*'a* :: *semiring-1*)*mat*)
shows *mat-pow n m k* = $m\ [\urcorner]_{mat\text{-}ring\ n\ b}\ ^{k}$
(**is** - = $m\ [\urcorner]_{?C}\ k$)
⟨*proof*⟩

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.