Matrix

Steven Obua

October 1, 2005

theory *MatrixGeneral* **imports** *Main* **begin**

types *'a infmatrix* = [*nat*, *nat*] ⇒ *'a*

constdefs
 nonzero-positions :: (*'a*::*zero*) *infmatrix* ⇒ (*nat*∗*nat*) *set*
 nonzero-positions A == {*pos*. *A* (*fst pos*) (*snd pos*) ~= *0*}

typedef *'a matrix* = {(*f*::((*'a*::*zero*) *infmatrix*)). *finite* (*nonzero-positions f*)}
apply (*rule-tac x*=(% *j i*. *0*) **in** *exI*)
by (*simp add*: *nonzero-positions-def*)

declare *Rep-matrix-inverse*[*simp*]

lemma *finite-nonzero-positions* : *finite* (*nonzero-positions* (*Rep-matrix A*))
apply (*rule Abs-matrix-induct*)
by (*simp add*: *Abs-matrix-inverse matrix-def*)

constdefs
 nrows :: (*'a*::*zero*) *matrix* ⇒ *nat*
 nrows A == *if nonzero-positions*(*Rep-matrix A*) = {} *then 0 else Suc*(*Max* ((*image fst*) (*nonzero-positions* (*Rep-matrix A*))))
 ncols :: (*'a*::*zero*) *matrix* ⇒ *nat*
 ncols A == *if nonzero-positions*(*Rep-matrix A*) = {} *then 0 else Suc*(*Max* ((*image snd*) (*nonzero-positions* (*Rep-matrix A*))))

lemma *nrows*:
 assumes *hyp*: *nrows A* ≤ *m*
 shows (*Rep-matrix A m n*) = *0* (**is** *?concl*)
proof *cases*
 assume *nonzero-positions*(*Rep-matrix A*) = {}
 then show (*Rep-matrix A m n*) = *0* **by** (*simp add*: *nonzero-positions-def*)
next
 assume *a*: *nonzero-positions*(*Rep-matrix A*) ≠ {}
 let *?S* = *fst*‘(*nonzero-positions*(*Rep-matrix A*))
 from *a* **have** *b*: *?S* ≠ {} **by** (*simp*)

```
  have c: finite (?S) by (simp add: finite-nonzero-positions)
  from hyp have d: Max (?S) < m by (simp add: a nrows-def)
  have m ∉ ?S
    proof −
      have m ∈ ?S ⟹ m <= Max(?S) by (simp add: Max-ge[OF c b])
      moreover from d have ~(m <= Max ?S) by (simp)
      ultimately show m ∉ ?S by (auto)
    qed
  thus Rep-matrix A m n = 0 by (simp add: nonzero-positions-def image-Collect)
qed

constdefs
  transpose-infmatrix :: 'a infmatrix ⇒ 'a infmatrix
  transpose-infmatrix A j i == A i j
  transpose-matrix :: ('a::zero) matrix ⇒ 'a matrix
  transpose-matrix == Abs-matrix o transpose-infmatrix o Rep-matrix

declare transpose-infmatrix-def[simp]

lemma transpose-infmatrix-twice[simp]: transpose-infmatrix (transpose-infmatrix
A) = A
by ((rule ext)+, simp)

lemma transpose-infmatrix: transpose-infmatrix (% j i. P j i) = (% j i. P i j)
  apply (rule ext)+
  by (simp add: transpose-infmatrix-def)

lemma transpose-infmatrix-closed[simp]: Rep-matrix (Abs-matrix (transpose-infmatrix
(Rep-matrix x))) = transpose-infmatrix (Rep-matrix x)
apply (rule Abs-matrix-inverse)
apply (simp add: matrix-def nonzero-positions-def image-def)
proof −
  let ?A = {pos. Rep-matrix x (snd pos) (fst pos) ≠ 0}
  let ?swap = % pos. (snd pos, fst pos)
  let ?B = {pos. Rep-matrix x (fst pos) (snd pos) ≠ 0}
  have swap-image: ?swap`?A = ?B
    apply (simp add: image-def)
    apply (rule set-ext)
    apply (simp)
    proof
      fix y
      assume hyp: ∃ a b. Rep-matrix x b a ≠ 0 ∧ y = (b, a)
      thus Rep-matrix x (fst y) (snd y) ≠ 0
        proof −
         from hyp obtain a b where (Rep-matrix x b a ≠ 0 & y = (b,a)) by blast
          then show Rep-matrix x (fst y) (snd y) ≠ 0 by (simp)
        qed
    next
      fix y
```

assume *hyp*: *Rep-matrix x* (*fst y*) (*snd y*) $\neq$ *0*
show $\exists$ *a b*. (*Rep-matrix x b a* $\neq$ *0* & *y* = (*b*,*a*)) **by** (*rule exI*[*of - snd y*], *rule exI*[*of - fst y*], *simp*)
qed
then have *finite* (*?swap'?A*)
proof –
have *finite* (*nonzero-positions* (*Rep-matrix x*)) **by** (*simp add*: *finite-nonzero-positions*)
then have *finite ?B* **by** (*simp add*: *nonzero-positions-def*)
with *swap-image* **show** *finite* (*?swap'?A*) **by** (*simp*)
qed
moreover
have *inj-on ?swap ?A* **by** (*simp add*: *inj-on-def*)
ultimately show *finite ?A***by** (*rule finite-imageD*[*of ?swap ?A*])
qed

lemma *infmatrixforward*: (*x*::*'a infmatrix*) = *y* $\Longrightarrow$ $\forall$ *a b*. *x a b* = *y a b* **by** *auto*

lemma *transpose-infmatrix-inject*: (*transpose-infmatrix A* = *transpose-infmatrix B*) = (*A* = *B*)
apply (*auto*)
apply (*rule ext*)+
apply (*simp add*: *transpose-infmatrix*)
apply (*drule infmatrixforward*)
apply (*simp*)
done

lemma *transpose-matrix-inject*: (*transpose-matrix A* = *transpose-matrix B*) = (*A* = *B*)
apply (*simp add*: *transpose-matrix-def*)
apply (*subst Rep-matrix-inject*[*THEN sym*])+
apply (*simp only*: *transpose-infmatrix-closed transpose-infmatrix-inject*)
done

lemma *transpose-matrix*[*simp*]: *Rep-matrix*(*transpose-matrix A*) *j i* = *Rep-matrix A i j*
by (*simp add*: *transpose-matrix-def*)

lemma *transpose-transpose-id*[*simp*]: *transpose-matrix* (*transpose-matrix A*) = *A*
by (*simp add*: *transpose-matrix-def*)

lemma *nrows-transpose*[*simp*]: *nrows* (*transpose-matrix A*) = *ncols A*
by (*simp add*: *nrows-def ncols-def nonzero-positions-def transpose-matrix-def image-def*)

lemma *ncols-transpose*[*simp*]: *ncols* (*transpose-matrix A*) = *nrows A*
by (*simp add*: *nrows-def ncols-def nonzero-positions-def transpose-matrix-def image-def*)

lemma *ncols*: *ncols A* <= *n* $\Longrightarrow$ *Rep-matrix A m n* = *0*
proof –
assume *ncols A* <= *n*

then have *nrows* (*transpose-matrix A*) <= *n* **by** (*simp*)
then have *Rep-matrix* (*transpose-matrix A*) *n m* = *0* **by** (*rule nrows*)
thus *Rep-matrix A m n* = *0* **by** (*simp add*: *transpose-matrix-def*)
qed

lemma *ncols-le*: (*ncols A* <= *n*) = (! *j i*. *n* <= *i* ⟶ (*Rep-matrix A j i*) = *0*) (**is** - = *?st*)
apply (*auto*)
apply (*simp add*: *ncols*)
proof (*simp add*: *ncols-def*, *auto*)
let *?P* = *nonzero-positions* (*Rep-matrix A*)
let *?p* = *snd`?P*
have *a*:*finite ?p* **by** (*simp add*: *finite-nonzero-positions*)
let *?m* = *Max ?p*
assume ~(*Suc* (*?m*) <= *n*)
then have *b*:*n* <= *?m* **by** (*simp*)
fix *a b*
assume (*a*,*b*) ∈ *?P*
then have *?p* ≠ {} **by** (*auto*)
with *a* **have** *?m* ∈ *?p* **by** (*simp*)
moreover have !*x*. (*x* ∈ *?p* ⟶ (*? y*. (*Rep-matrix A y x*) ≠ *0*)) **by** (*simp add*: *nonzero-positions-def image-def*)
ultimately have *? y*. (*Rep-matrix A y ?m*) ≠ *0* **by** (*simp*)
moreover assume *?st*
ultimately show *False* **using** *b* **by** (*simp*)
qed

lemma *less-ncols*: (*n* < *ncols A*) = (*? j i*. *n* <= *i* & (*Rep-matrix A j i*) ≠ *0*) (**is** *?concl*)
proof −
have *a*: !! (*a*::*nat*) *b*. (*a* < *b*) = (~(*b* <= *a*)) **by** *arith*
show *?concl* **by** (*simp add*: *a ncols-le*)
qed

lemma *le-ncols*: (*n* <= *ncols A*) = (∀ *m*. (∀ *j i*. *m* <= *i* ⟶ (*Rep-matrix A j i*) = *0*) ⟶ *n* <= *m*) (**is** *?concl*)
apply (*auto*)
apply (*subgoal-tac ncols A* <= *m*)
apply (*simp*)
apply (*simp add*: *ncols-le*)
apply (*drule-tac x*=*ncols A* **in** *spec*)
by (*simp add*: *ncols*)

lemma *nrows-le*: (*nrows A* <= *n*) = (! *j i*. *n* <= *j* ⟶ (*Rep-matrix A j i*) = *0*) (**is** *?s*)
proof −
have (*nrows A* <= *n*) = (*ncols* (*transpose-matrix A*) <= *n*) **by** (*simp*)
also have . . . = (! *j i*. *n* <= *i* ⟶ (*Rep-matrix* (*transpose-matrix A*) *j i* = *0*)) **by** (*rule ncols-le*)

also have ... = (! *j i*. *n* <= *i* $\longrightarrow$ (*Rep-matrix A i j*) = *0*) **by** (*simp*)
finally show (*nrows A* <= *n*) = (! *j i*. *n* <= *j* $\longrightarrow$ (*Rep-matrix A j i*) = *0*) **by** (*auto*)
qed

lemma *less-nrows*: (*m* < *nrows A*) = (*? j i*. *m* <= *j* & (*Rep-matrix A j i*) $\neq$ *0*) (**is** *?concl*)
proof −
have *a*: !! (*a*::*nat*) *b*. (*a* < *b*) = (~(*b* <= *a*)) **by** *arith*
show *?concl* **by** (*simp add*: *a nrows-le*)
qed

lemma *le-nrows*: (*n* <= *nrows A*) = ($\forall$ *m*. ($\forall$ *j i*. *m* <= *j* $\longrightarrow$ (*Rep-matrix A j i*) = *0*) $\longrightarrow$ *n* <= *m*) (**is** *?concl*)
apply (*auto*)
apply (*subgoal-tac nrows A* <= *m*)
apply (*simp*)
apply (*simp add*: *nrows-le*)
apply (*drule-tac x*=*nrows A* **in** *spec*)
by (*simp add*: *nrows*)

lemma *nrows-notzero*: *Rep-matrix A m n* $\neq$ *0* $\Longrightarrow$ *m* < *nrows A*
apply (*case-tac nrows A* <= *m*)
apply (*simp-all add*: *nrows*)
done

lemma *ncols-notzero*: *Rep-matrix A m n* $\neq$ *0* $\Longrightarrow$ *n* < *ncols A*
apply (*case-tac ncols A* <= *n*)
apply (*simp-all add*: *ncols*)
done

lemma *finite-natarray1*: *finite* {*x*. *x* < (*n*::*nat*)}
apply (*induct n*)
apply (*simp*)
proof −
fix *n*
have {*x*. *x* < *Suc n*} = *insert n* {*x*. *x* < *n*} **by** (*rule set-ext*, *simp*, *arith*)
moreover assume *finite* {*x*. *x* < *n*}
ultimately show *finite* {*x*. *x* < *Suc n*} **by** (*simp*)
qed

lemma *finite-natarray2*: *finite* {*pos*. (*fst pos*) < (*m*::*nat*) & (*snd pos*) < (*n*::*nat*)}
apply (*induct m*)
apply (*simp*+)
proof −
fix *m*::*nat*
let *?s0* = {*pos*. *fst pos* < *m* & *snd pos* < *n*}
let *?s1* = {*pos*. *fst pos* < (*Suc m*) & *snd pos* < *n*}
let *?sd* = {*pos*. *fst pos* = *m* & *snd pos* < *n*}

assume *f0*: *finite ?s0*
have *f1*: *finite ?sd*
proof −
let *?f* = % *x*. (*m*, *x*)
have {*pos*. *fst pos* = *m* & *snd pos* < *n*} = *?f* ‘ {*x*. *x* < *n*} **by** (*rule set-ext*, *simp add*: *image-def*, *auto*)
moreover have *finite* {*x*. *x* < *n*} **by** (*simp add*: *finite-natarray1*)
ultimately show *finite* {*pos*. *fst pos* = *m* & *snd pos* < *n*} **by** (*simp*)
qed
have *su*: *?s0* ∪ *?sd* = *?s1* **by** (*rule set-ext*, *simp*, *arith*)
from *f0 f1* **have** *finite* (*?s0* ∪ *?sd*) **by** (*rule finite-UnI*)
with *su* **show** *finite ?s1* **by** (*simp*)
qed

lemma *RepAbs-matrix*:
assumes *aem*: *?* *m*. ! *j i*. *m* <= *j* ⟶ *x j i* = *0* (**is** *?em*) **and** *aen*:*?* *n*. ! *j i*. (*n* <= *i* ⟶ *x j i* = *0*) (**is** *?en*)
shows (*Rep-matrix* (*Abs-matrix x*)) = *x*
apply (*rule Abs-matrix-inverse*)
apply (*simp add*: *matrix-def nonzero-positions-def*)
proof −
from *aem* **obtain** *m* **where** *a*: ! *j i*. *m* <= *j* ⟶ *x j i* = *0* **by** (*blast*)
from *aen* **obtain** *n* **where** *b*: ! *j i*. *n* <= *i* ⟶ *x j i* = *0* **by** (*blast*)
let *?u* = {*pos*. *x* (*fst pos*) (*snd pos*) ≠ *0*}
let *?v* = {*pos*. *fst pos* < *m* & *snd pos* < *n*}
have *c*: !! (*m*::*nat*) *a*. ~(*m* <= *a*) ⟹ *a* < *m* **by** (*arith*)
from *a b* **have** (*?u* ∩ (−*?v*)) = {}
apply (*simp*)
apply (*rule set-ext*)
apply (*simp*)
apply *auto*
by (*rule c*, *auto*)+
then have *d*: *?u* ⊆ *?v* **by** *blast*
moreover have *finite ?v* **by** (*simp add*: *finite-natarray2*)
ultimately show *finite ?u* **by** (*rule finite-subset*)
qed

constdefs
apply-infmatrix :: (′*a* ⇒ ′*b*) ⇒ ′*a infmatrix* ⇒ ′*b infmatrix*
apply-infmatrix f == % *A*. (% *j i*. *f* (*A j i*))
apply-matrix :: (′*a* ⇒ ′*b*) ⇒ (′*a*::*zero*) *matrix* ⇒ (′*b*::*zero*) *matrix*
apply-matrix f == % *A*. *Abs-matrix* (*apply-infmatrix f* (*Rep-matrix A*))
combine-infmatrix :: (′*a* ⇒ ′*b* ⇒ ′*c*) ⇒ ′*a infmatrix* ⇒ ′*b infmatrix* ⇒ ′*c infmatrix*
combine-infmatrix f == % *A B*. (% *j i*. *f* (*A j i*) (*B j i*))
combine-matrix :: (′*a* ⇒ ′*b* ⇒ ′*c*) ⇒ (′*a*::*zero*) *matrix* ⇒ (′*b*::*zero*) *matrix* ⇒ (′*c*::*zero*) *matrix*
combine-matrix f == % *A B*. *Abs-matrix* (*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*))

lemma *expand-apply-infmatrix*[*simp*]: *apply-infmatrix f A j i* = *f* (*A j i*)
by (*simp add*: *apply-infmatrix-def*)

lemma *expand-combine-infmatrix*[*simp*]: *combine-infmatrix f A B j i* = *f* (*A j i*) (*B j i*)
by (*simp add*: *combine-infmatrix-def*)

constdefs
commutative :: ($'a \Rightarrow 'a \Rightarrow 'b$) $\Rightarrow$ *bool*
commutative f == ! *x y*. *f x y* = *f y x*
associative :: ($'a \Rightarrow 'a \Rightarrow 'a$) $\Rightarrow$ *bool*
associative f == ! *x y z*. *f* (*f x y*) *z* = *f x* (*f y z*)

To reason about associativity and commutativity of operations on matrices, let's take a step back and look at the general situtation: Assume that we have sets A and B with $B \subset A$ and an abstraction $u : A \to B$. This abstraction has to fulfill $u(b) = b$ for all $b \in B$, but is arbitrary otherwise. Each function $f : A \times A \to A$ now induces a function $f' : B \times B \to B$ by $f' = u \circ f$. It is obvious that commutativity of f implies commutativity of f': $f'xy = u(fxy) = u(fyx) = f'yx$.

lemma *combine-infmatrix-commute*:
commutative f $\Longrightarrow$ *commutative* (*combine-infmatrix f*)
by (*simp add*: *commutative-def combine-infmatrix-def*)

lemma *combine-matrix-commute*:
commutative f $\Longrightarrow$ *commutative* (*combine-matrix f*)
by (*simp add*: *combine-matrix-def commutative-def combine-infmatrix-def*)

On the contrary, given an associative function f we cannot expect f' to be associative. A counterexample is given by $A = \mathbb{Z}$, $B = \{-1, 0, 1\}$, as f we take addition on $\mathbb{Z}$, which is clearly associative. The abstraction is given by $u(a) = 0$ for $a \notin B$. Then we have

$$f'(f'11) - 1 = u(f(u(f11)) - 1) = u(f(u2) - 1) = u(f0 - 1) = -1,$$

but on the other hand we have

$$f'1(f'1 - 1) = u(f1(u(f1 - 1))) = u(f10) = 1.$$

A way out of this problem is to assume that $f(A \times A) \subset A$ holds, and this is what we are going to do:

lemma *nonzero-positions-combine-infmatrix*[*simp*]: *f 0 0* = *0* $\Longrightarrow$ *nonzero-positions* (*combine-infmatrix f A B*) $\subseteq$ (*nonzero-positions A*) $\cup$ (*nonzero-positions B*)
by (*rule subsetI*, *simp add*: *nonzero-positions-def combine-infmatrix-def*, *auto*)

lemma *finite-nonzero-positions-Rep*[*simp*]: *finite* (*nonzero-positions* (*Rep-matrix A*))
by (*insert Rep-matrix* [*of A*], *simp add*: *matrix-def*)

lemma *combine-infmatrix-closed* [*simp*]:
f 0 0 = *0* $\Longrightarrow$ *Rep-matrix* (*Abs-matrix* (*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*))) = *combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*)
apply (*rule Abs-matrix-inverse*)
apply (*simp add*: *matrix-def*)
apply (*rule finite-subset*[*of* - (*nonzero-positions* (*Rep-matrix A*)) ∪ (*nonzero-positions* (*Rep-matrix B*))])
by (*simp-all*)

We need the next two lemmas only later, but it is analog to the above one, so we prove them now:

lemma *nonzero-positions-apply-infmatrix*[*simp*]: *f 0* = *0* $\Longrightarrow$ *nonzero-positions* (*apply-infmatrix f A*) ⊆ *nonzero-positions A*
by (*rule subsetI*, *simp add*: *nonzero-positions-def apply-infmatrix-def*, *auto*)

lemma *apply-infmatrix-closed* [*simp*]:
f 0 = *0* $\Longrightarrow$ *Rep-matrix* (*Abs-matrix* (*apply-infmatrix f* (*Rep-matrix A*))) = *apply-infmatrix f* (*Rep-matrix A*)
apply (*rule Abs-matrix-inverse*)
apply (*simp add*: *matrix-def*)
apply (*rule finite-subset*[*of* - *nonzero-positions* (*Rep-matrix A*)])
by (*simp-all*)

lemma *combine-infmatrix-assoc*[*simp*]: *f 0 0* = *0* $\Longrightarrow$ *associative f* $\Longrightarrow$ *associative* (*combine-infmatrix f*)
by (*simp add*: *associative-def combine-infmatrix-def*)

lemma *comb*: *f* = *g* $\Longrightarrow$ *x* = *y* $\Longrightarrow$ *f x* = *g y*
by (*auto*)

lemma *combine-matrix-assoc*: *f 0 0* = *0* $\Longrightarrow$ *associative f* $\Longrightarrow$ *associative* (*combine-matrix f*)
apply (*simp*(*no-asm*) *add*: *associative-def combine-matrix-def*, *auto*)
apply (*rule comb* [*of Abs-matrix Abs-matrix*])
by (*auto*, *insert combine-infmatrix-assoc*[*of f*], *simp add*: *associative-def*)

lemma *Rep-apply-matrix*[*simp*]: *f 0* = *0* $\Longrightarrow$ *Rep-matrix* (*apply-matrix f A*) *j i* = *f* (*Rep-matrix A j i*)
by (*simp add*: *apply-matrix-def*)

lemma *Rep-combine-matrix*[*simp*]: *f 0 0* = *0* $\Longrightarrow$ *Rep-matrix* (*combine-matrix f A B*) *j i* = *f* (*Rep-matrix A j i*) (*Rep-matrix B j i*)
by(*simp add*: *combine-matrix-def*)

lemma *combine-nrows*: *f 0 0* = *0* $\Longrightarrow$ *nrows* (*combine-matrix f A B*) <= *max* (*nrows A*) (*nrows B*)
by (*simp add*: *nrows-le*)

lemma *combine-ncols*: *f 0 0* = *0* ⟹ *ncols* (*combine-matrix f A B*) <= *max* (*ncols A*) (*ncols B*)
by (*simp add*: *ncols-le*)

lemma *combine-nrows*: *f 0 0* = *0* ⟹ *nrows A* <= *q* ⟹ *nrows B* <= *q* ⟹ *nrows*(*combine-matrix f A B*) <= *q*
  by (*simp add*: *nrows-le*)

lemma *combine-ncols*: *f 0 0* = *0* ⟹ *ncols A* <= *q* ⟹ *ncols B* <= *q* ⟹ *ncols*(*combine-matrix f A B*) <= *q*
  by (*simp add*: *ncols-le*)

constdefs
  zero-r-neutral :: (′*a* ⇒ ′*b*::*zero* ⇒ ′*a*) ⇒ *bool*
  zero-r-neutral f == ! *a*. *f a 0* = *a*
  zero-l-neutral :: (′*a*::*zero* ⇒ ′*b* ⇒ ′*b*) ⇒ *bool*
  zero-l-neutral f == ! *a*. *f 0 a* = *a*
  zero-closed :: ((′*a*::*zero*) ⇒ (′*b*::*zero*) ⇒ (′*c*::*zero*)) ⇒ *bool*
  zero-closed f == (!*x*. *f x 0* = *0*) & (!*y*. *f 0 y* = *0*)

consts *foldseq* :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (*nat* ⇒ ′*a*) ⇒ *nat* ⇒ ′*a*
primrec
  foldseq f s 0 = *s 0*
  foldseq f s (*Suc n*) = *f* (*s 0*) (*foldseq f* (% *k*. *s*(*Suc k*)) *n*)

consts *foldseq-transposed* :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (*nat* ⇒ ′*a*) ⇒ *nat* ⇒ ′*a*
primrec
  foldseq-transposed f s 0 = *s 0*
  foldseq-transposed f s (*Suc n*) = *f* (*foldseq-transposed f s n*) (*s* (*Suc n*))

lemma *foldseq-assoc* : *associative f* ⟹ *foldseq f* = *foldseq-transposed f*
proof −
  assume *a*:*associative f*
  then have *sublemma*: !! *n*. ! *N s*. *N* <= *n* ⟶ *foldseq f s N* = *foldseq-transposed f s N*
  proof −
    fix *n*
    show !*N s*. *N* <= *n* ⟶ *foldseq f s N* = *foldseq-transposed f s N*
    proof (*induct n*)
      show !*N s*. *N* <= *0* ⟶ *foldseq f s N* = *foldseq-transposed f s N* **by** *simp*
    next
      fix *n*
      assume *b*:! *N s*. *N* <= *n* ⟶ *foldseq f s N* = *foldseq-transposed f s N*
      have *c*:!!*N s*. *N* <= *n* ⟹ *foldseq f s N* = *foldseq-transposed f s N* **by** (*simp add*: *b*)
      show ! *N t*. *N* <= *Suc n* ⟶ *foldseq f t N* = *foldseq-transposed f t N*
      proof (*auto*)
        fix *N t*
        assume *Nsuc*: *N* <= *Suc n*

show *foldseq f t N* = *foldseq-transposed f t N*
proof *cases*
assume $N <= n$
then show *foldseq f t N* = *foldseq-transposed f t N* **by** (*simp add*: *b*)
next
assume ~($N <= n$)
with *Nsuc* **have** *Nsuceq*: $N = Suc\ n$ **by** *simp*
have *neqz*: $n \neq 0 \Longrightarrow ?\ m.\ n = Suc\ m\ \&\ Suc\ m <= n$ **by** *arith*
have *assocf*: !! $x\ y\ z.\ f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z$ **by** (*insert a*, *simp add*: *associative-def*)
show *foldseq f t N* = *foldseq-transposed f t N*
apply (*simp add*: *Nsuceq*)
apply (*subst c*)
apply (*simp*)
apply (*case-tac* $n = 0$)
apply (*simp*)
apply (*drule neqz*)
apply (*erule exE*)
apply (*simp*)
apply (*subst assocf*)
proof −
fix *m*
assume $n = Suc\ m\ \&\ Suc\ m <= n$
then have *mless*: $Suc\ m <= n$ **by** *arith*
then have *step1*: *foldseq-transposed f* (% *k*. *t* (*Suc k*)) *m* = *foldseq f* (% *k*. *t* (*Suc k*)) *m* (**is** *?T1* = *?T2*)
apply (*subst c*)
by *simp*+
have *step2*: *f* (*t 0*) *?T2* = *foldseq f t* (*Suc m*) (**is** - = *?T3*) **by** *simp*
have *step3*: *?T3* = *foldseq-transposed f t* (*Suc m*) (**is** - = *?T4*)
apply (*subst c*)
by (*simp add*: *mless*)+
have *step4*: *?T4* = *f* (*foldseq-transposed f t m*) (*t* (*Suc m*)) (**is** -=*?T5*) **by** *simp*
from *step1 step2 step3 step4* **show** *sowhat*: *f* (*f* (*t 0*) *?T1*) (*t* (*Suc* (*Suc m*))) = *f ?T5* (*t* (*Suc* (*Suc m*))) **by** *simp*
qed
qed
qed
qed
qed
show *foldseq f* = *foldseq-transposed f* **by** ((*rule ext*)+, *insert sublemma*, *auto*)
qed

lemma *foldseq-distr*: ⟦*associative f*; *commutative f*⟧ $\Longrightarrow$ *foldseq f* (% *k*. *f* (*u k*) (*v k*)) *n* = *f* (*foldseq f u n*) (*foldseq f v n*)
proof −
assume *assoc*: *associative f*
assume *comm*: *commutative f*

from *assoc* **have** *a*:!! $x\ y\ z.\ f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$ **by** (*simp add*: *associative-def*)
from *comm* **have** *b*: !! $x\ y.\ f\ x\ y = f\ y\ x$ **by** (*simp add*: *commutative-def*)
from *assoc comm* **have** *c*: !! $x\ y\ z.\ f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$ **by** (*simp add*: *commutative-def associative-def*)
have !! $n.$ (! $u\ v.$ *foldseq* f (%$k.\ f\ (u\ k)\ (v\ k))\ n = f$ (*foldseq* $f\ u\ n$) (*foldseq* $f\ v\ n$))
apply (*induct-tac n*)
apply (*simp+*, *auto*)
by (*simp add*: *a b c*)
then show *foldseq* f (% $k.\ f\ (u\ k)\ (v\ k))\ n = f$ (*foldseq* $f\ u\ n$) (*foldseq* $f\ v\ n$) **by** *simp*
qed

theorem $[\![$*associative* f; *associative* g; $\forall\ a\ b\ c\ d.\ g\ (f\ a\ b)\ (f\ c\ d) = f\ (g\ a\ c)\ (g\ b\ d)$; ? $x\ y.\ (f\ x) \neq (f\ y)$; ? $x\ y.\ (g\ x) \neq (g\ y)$; $f\ x\ x = x$; $g\ x\ x = x]\!] \Longrightarrow f{=}g$ | (! $y.\ f\ y\ x = y$) | (! $y.\ g\ y\ x = y$)
oops

lemma *foldseq-zero*:
assumes *fz*: $f\ 0\ 0 = 0$ **and** *sz*: ! $i.\ i <= n \longrightarrow s\ i = 0$
shows *foldseq* $f\ s\ n = 0$
proof −
have !! $n.$! $s.$ (! $i.\ i <= n \longrightarrow s\ i = 0) \longrightarrow$ *foldseq* $f\ s\ n = 0$
apply (*induct-tac n*)
apply (*simp*)
by (*simp add*: *fz*)
then show *foldseq* $f\ s\ n = 0$ **by** (*simp add*: *sz*)
qed

lemma *foldseq-significant-positions*:
assumes *p*: ! $i.\ i <= N \longrightarrow S\ i = T\ i$
shows *foldseq* $f\ S\ N =$ *foldseq* $f\ T\ N$ (**is** *?concl*)
proof −
have !! m . ! $s\ t.$ (! $i.\ i{<=}m \longrightarrow s\ i = t\ i) \longrightarrow$ *foldseq* $f\ s\ m =$ *foldseq* $f\ t\ m$
apply (*induct-tac m*)
apply (*simp*)
apply (*simp*)
apply (*auto*)
proof −
fix n
fix $s{::}nat{\Rightarrow}'a$
fix $t{::}nat{\Rightarrow}'a$
assume *a*: $\forall\ s\ t.\ (\forall\ i{\leq}n.\ s\ i = t\ i) \longrightarrow$ *foldseq* $f\ s\ n =$ *foldseq* $f\ t\ n$
assume *b*: $\forall\ i{\leq}Suc\ n.\ s\ i = t\ i$
have *c*:!! $a\ b.\ a = b \Longrightarrow f\ (t\ 0)\ a = f\ (t\ 0)\ b$ **by** *blast*
have *d*:!! $s\ t.\ (\forall\ i{\leq}n.\ s\ i = t\ i) \Longrightarrow$ *foldseq* $f\ s\ n =$ *foldseq* $f\ t\ n$ **by** (*simp add*: *a*)
show $f\ (t\ 0)$ (*foldseq* $f\ (\lambda k.\ s\ (Suc\ k))\ n) = f\ (t\ 0)$ (*foldseq* $f\ (\lambda k.\ t\ (Suc$

k)) *n*) **by** (*rule c*, *simp add*: *d b*)
qed
with *p* **show** *?concl* **by** *simp*
qed

lemma *foldseq-tail*: $M <= N \Longrightarrow$ *foldseq f S N* = *foldseq f* (% *k*. (*if k* < *M then* (*S k*) *else* (*foldseq f* (% *k*. *S*(*k*+*M*)) (*N*−*M*)))) *M* (**is** *?p* $\Longrightarrow$ *?concl*)
proof −
have *suc*: !! *a b*. ⟦*a* <= *Suc b*; *a* ≠ *Suc b*⟧ $\Longrightarrow$ *a* <= *b* **by** *arith*
have *a*:!! *a b c* . *a* = *b* $\Longrightarrow$ *f c a* = *f c b* **by** *blast*
have !! *n*. ! *m s*. *m* <= *n* ⟶ *foldseq f s n* = *foldseq f* (% *k*. (*if k* < *m then* (*s k*) *else* (*foldseq f* (% *k*. *s*(*k*+*m*)) (*n*−*m*)))) *m*
apply (*induct-tac n*)
apply (*simp*)
apply (*simp*)
apply (*auto*)
apply (*case-tac m* = *Suc na*)
apply (*simp*)
apply (*rule a*)
apply (*rule foldseq-significant-positions*)
apply (*auto*)
apply (*drule suc*, *simp*+)
proof −
fix *na m s*
assume *suba*:∀ *m*≤*na*. ∀ *s*. *foldseq f s na* = *foldseq f* (λ*k*. *if k* < *m then s k else foldseq f* (λ*k*. *s* (*k* + *m*)) (*na* − *m*))*m*
assume *subb*:*m* <= *na*
from *suba* **have** *subc*:!! *m s*. *m* <= *na* $\Longrightarrow$*foldseq f s na* = *foldseq f* (λ*k*. *if k* < *m then s k else foldseq f* (λ*k*. *s* (*k* + *m*)) (*na* − *m*))*m* **by** *simp*
have *subd*: *foldseq f* (λ*k*. *if k* < *m then s* (*Suc k*) *else foldseq f* (λ*k*. *s* (*Suc* (*k* + *m*))) (*na* − *m*)) *m* =
foldseq f (% *k*. *s*(*Suc k*)) *na*
by (*rule subc*[*of m* % *k*. *s*(*Suc k*), *THEN sym*], *simp add*: *subb*)
from *subb* **have** *sube*: *m* ≠ *0* $\Longrightarrow$? *mm*. *m* = *Suc mm* & *mm* <= *na* **by** *arith*
show *f* (*s 0*) (*foldseq f* (λ*k*. *if k* < *m then s* (*Suc k*) *else foldseq f* (λ*k*. *s* (*Suc* (*k* + *m*))) (*na* − *m*)) *m*) =
foldseq f (λ*k*. *if k* < *m then s k else foldseq f* (λ*k*. *s* (*k* + *m*)) (*Suc na* − *m*)) *m*
apply (*simp add*: *subd*)
apply (*case-tac m*=*0*)
apply (*simp*)
apply (*drule sube*)
apply (*auto*)
apply (*rule a*)
by (*simp add*: *subc if-def*)
qed
then show *?p* $\Longrightarrow$ *?concl* **by** *simp*
qed

lemma *foldseq-zerotail*:
  assumes
  fz: *f 0 0* = *0*
  and *sz*: ! *i*. *n* <= *i* ⟶ *s i* = *0*
  and *nm*: *n* <= *m*
  shows
  foldseq f s n = *foldseq f s m*
proof –
  show *foldseq f s n* = *foldseq f s m*
    apply (*simp add*: *foldseq-tail*[*OF nm*, *of f s*])
    apply (*rule foldseq-significant-positions*)
    apply (*auto*)
    apply (*subst foldseq-zero*)
    by (*simp add*: *fz sz*)+
qed

lemma *foldseq-zerotail2*:
  assumes ! *x*. *f x 0* = *x*
  and ! *i*. *n* < *i* ⟶ *s i* = *0*
  and *nm*: *n* <= *m*
  shows
  foldseq f s n = *foldseq f s m* (**is** *?concl*)
proof –
  have *f 0 0* = *0* **by** (*simp add*: *prems*)
  have *b*:!! *m n*. *n* <= *m* ⟹ *m* ≠ *n* ⟹ *? k*. *m*−*n* = *Suc k* **by** *arith*
  have *c*: *0* <= *m* **by** *simp*
  have *d*: !! *k*. *k* ≠ *0* ⟹ *? l*. *k* = *Suc l* **by** *arith*
  show *?concl*
    apply (*subst foldseq-tail*[*OF nm*])
    apply (*rule foldseq-significant-positions*)
    apply (*auto*)
    apply (*case-tac m*=*n*)
    apply (*simp*+)
    apply (*drule b*[*OF nm*])
    apply (*auto*)
    apply (*case-tac k*=*0*)
    apply (*simp add*: *prems*)
    apply (*drule d*)
    apply (*auto*)
    by (*simp add*: *prems foldseq-zero*)
qed

lemma *foldseq-zerostart*:
  ! *x*. *f 0* (*f 0 x*) = *f 0 x* ⟹ ! *i*. *i* <= *n* ⟶ *s i* = *0* ⟹ *foldseq f s* (*Suc n*) = *f 0* (*s* (*Suc n*))
proof –
  assume *f00x*: ! *x*. *f 0* (*f 0 x*) = *f 0 x*
  have ! *s*. (! *i*. *i*<=*n* ⟶ *s i* = *0*) ⟶ *foldseq f s* (*Suc n*) = *f 0* (*s* (*Suc n*))

apply (*induct n*)
apply (*simp*)
apply (*rule allI*, *rule impI*)
proof −
fix *n*
fix *s*
have *a*:*foldseq f s* (*Suc* (*Suc n*)) = *f* (*s 0*) (*foldseq f* (% *k*. *s*(*Suc k*)) (*Suc n*)) **by** *simp*
assume *b*: ! *s*. ((∀ *i*≤*n*. *s i* = *0*) ⟶ *foldseq f s* (*Suc n*) = *f 0* (*s* (*Suc n*)))
from *b* **have** *c*:!! *s*. (∀ *i*≤*n*. *s i* = *0*) ⟹ *foldseq f s* (*Suc n*) = *f 0* (*s* (*Suc n*)) **by** *simp*
assume *d*: ! *i*. *i* <= *Suc n* ⟶ *s i* = *0*
show *foldseq f s* (*Suc* (*Suc n*)) = *f 0* (*s* (*Suc* (*Suc n*)))
apply (*subst a*)
apply (*subst c*)
by (*simp add*: *d f00x*)+
qed
then show ! *i*. *i* <= *n* ⟶ *s i* = *0* ⟹ *foldseq f s* (*Suc n*) = *f 0* (*s* (*Suc n*)) **by** *simp*
qed

lemma *foldseq-zerostart2*:
! *x*. *f 0 x* = *x* ⟹ ! *i*. *i* < *n* ⟶ *s i* = *0* ⟹ *foldseq f s n* = *s n*
proof −
assume *a*:! *i*. *i*<*n* ⟶ *s i* = *0*
assume *x*:! *x*. *f 0 x* = *x*
from *x* **have** *f00x*: ! *x*. *f 0* (*f 0 x*) = *f 0 x* **by** *blast*
have *b*: !! *i l*. *i* < *Suc l* = (*i* <= *l*) **by** *arith*
have *d*: !! *k*. *k* ≠ *0* ⟹ *?* *l*. *k* = *Suc l* **by** *arith*
show *foldseq f s n* = *s n*
apply (*case-tac n*=*0*)
apply (*simp*)
apply (*insert a*)
apply (*drule d*)
apply (*auto*)
apply (*simp add*: *b*)
apply (*insert f00x*)
apply (*drule foldseq-zerostart*)
by (*simp add*: *x*)+
qed

lemma *foldseq-almostzero*:
assumes *f0x*:! *x*. *f 0 x* = *x* **and** *fx0*: ! *x*. *f x 0* = *x* **and** *s0*:! *i*. *i* ≠ *j* ⟶ *s i* = *0*
shows *foldseq f s n* = (*if* (*j* <= *n*) *then* (*s j*) *else 0*) (**is** *?concl*)
proof −
from *s0* **have** *a*: ! *i*. *i* < *j* ⟶ *s i* = *0* **by** *simp*
from *s0* **have** *b*: ! *i*. *j* < *i* ⟶ *s i* = *0* **by** *simp*
show *?concl*
apply *auto*

apply (*subst foldseq-zerotail2*[*of f*, *OF fx0*, *of j*, *OF b*, *of n*, *THEN sym*])
apply *simp*
apply (*subst foldseq-zerostart2*)
apply (*simp add*: *f0x a*)+
apply (*subst foldseq-zero*)
by (*simp add*: *s0 f0x*)+
qed

lemma *foldseq-distr-unary*:
assumes !! *a b*. *g* (*f a b*) = *f* (*g a*) (*g b*)
shows *g*(*foldseq f s n*) = *foldseq f* (% *x*. *g*(*s x*)) *n* (**is** *?concl*)
proof −
have ! *s*. *g*(*foldseq f s n*) = *foldseq f* (% *x*. *g*(*s x*)) *n*
apply (*induct-tac n*)
apply (*simp*)
apply (*simp*)
apply (*auto*)
apply (*drule-tac x*=% *k*. *s* (*Suc k*) **in** *spec*)
by (*simp add*: *prems*)
then show *?concl* **by** *simp*
qed

constdefs
mult-matrix-n :: *nat* ⇒ ((′*a*::*zero*) ⇒ (′*b*::*zero*) ⇒ (′*c*::*zero*)) ⇒ (′*c* ⇒ ′*c* ⇒ ′*c*) ⇒ ′*a matrix* ⇒ ′*b matrix* ⇒ ′*c matrix*
mult-matrix-n n fmul fadd A B == *Abs-matrix*(% *j i*. *foldseq fadd* (% *k*. *fmul* (*Rep-matrix A j k*) (*Rep-matrix B k i*)) *n*)
mult-matrix :: ((′*a*::*zero*) ⇒ (′*b*::*zero*) ⇒ (′*c*::*zero*)) ⇒ (′*c* ⇒ ′*c* ⇒ ′*c*) ⇒ ′*a matrix* ⇒ ′*b matrix* ⇒ ′*c matrix*
mult-matrix fmul fadd A B == *mult-matrix-n* (*max* (*ncols A*) (*nrows B*)) *fmul fadd A B*

lemma *mult-matrix-n*:
assumes *prems*: *ncols A* ≤ *n* (**is** *?An*) *nrows B* ≤ *n* (**is** *?Bn*) *fadd 0 0* = *0 fmul 0 0* = *0*
shows *c*:*mult-matrix fmul fadd A B* = *mult-matrix-n n fmul fadd A B* (**is** *?concl*)
proof −
show *?concl* **using** *prems*
apply (*simp add*: *mult-matrix-def mult-matrix-n-def*)
apply (*rule comb*[*of Abs-matrix Abs-matrix*], *simp*, (*rule ext*)+)
by (*rule foldseq-zerotail*, *simp-all add*: *nrows-le ncols-le prems*)
qed

lemma *mult-matrix-nm*:
assumes *prems*: *ncols A* <= *n nrows B* <= *n ncols A* <= *m nrows B* <= *m fadd 0 0* = *0 fmul 0 0* = *0*
shows *mult-matrix-n n fmul fadd A B* = *mult-matrix-n m fmul fadd A B*
proof −
from *prems* **have** *mult-matrix-n n fmul fadd A B* = *mult-matrix fmul fadd A B*

by (*simp add*: *mult-matrix-n*)
  also from *prems* **have** ... = *mult-matrix-n m fmul fadd A B* **by** (*simp add*: *mult-matrix-n*[*THEN sym*])
  finally show *mult-matrix-n n fmul fadd A B* = *mult-matrix-n m fmul fadd A B* **by** *simp*
qed

constdefs
  r-distributive :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ ('*b* ⇒ '*b* ⇒ '*b*) ⇒ *bool*
  r-distributive fmul fadd == ! *a u v*. *fmul a* (*fadd u v*) = *fadd* (*fmul a u*) (*fmul a v*)
  l-distributive :: ('*a* ⇒ '*b* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ '*a*) ⇒ *bool*
  l-distributive fmul fadd == ! *a u v*. *fmul* (*fadd u v*) *a* = *fadd* (*fmul u a*) (*fmul v a*)
  distributive :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ '*a*) ⇒ *bool*
  distributive fmul fadd == *l-distributive fmul fadd* & *r-distributive fmul fadd*

lemma *max1*: !! *a x y*. (*a*::*nat*) <= *x* ⟹ *a* <= *max x y* **by** (*arith*)
lemma *max2*: !! *b x y*. (*b*::*nat*) <= *y* ⟹ *b* <= *max x y* **by** (*arith*)

lemma *r-distributive-matrix*:
 assumes *prems*:
  r-distributive fmul fadd
  associative fadd
  commutative fadd
  fadd 0 0 = *0*
  ! *a*. *fmul a 0* = *0*
  ! *a*. *fmul 0 a* = *0*
 shows *r-distributive* (*mult-matrix fmul fadd*) (*combine-matrix fadd*) (**is** *?concl*)
proof −
  from *prems* **show** *?concl*
    apply (*simp add*: *r-distributive-def mult-matrix-def*, *auto*)
    proof −
      fix *a*::'*a matrix*
      fix *u*::'*b matrix*
      fix *v*::'*b matrix*
      let *?mx* = *max* (*ncols a*) (*max* (*nrows u*) (*nrows v*))
     from *prems* **show** *mult-matrix-n* (*max* (*ncols a*) (*nrows* (*combine-matrix fadd u v*))) *fmul fadd a* (*combine-matrix fadd u v*) =
        combine-matrix fadd (*mult-matrix-n* (*max* (*ncols a*) (*nrows u*)) *fmul fadd a u*) (*mult-matrix-n* (*max* (*ncols a*) (*nrows v*)) *fmul fadd a v*)
        apply (*subst mult-matrix-nm*[*of - - - ?mx fadd fmul*])
        apply (*simp add*: *max1 max2 combine-nrows combine-ncols*)+
        apply (*subst mult-matrix-nm*[*of - - v ?mx fadd fmul*])
        apply (*simp add*: *max1 max2 combine-nrows combine-ncols*)+
        apply (*subst mult-matrix-nm*[*of - - u ?mx fadd fmul*])
        apply (*simp add*: *max1 max2 combine-nrows combine-ncols*)+
        apply (*simp add*: *mult-matrix-n-def r-distributive-def foldseq-distr*[*of fadd*])
        apply (*simp add*: *combine-matrix-def combine-infmatrix-def*)

```
      apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
      apply (simplesubst RepAbs-matrix)
      apply (simp, auto)
      apply (rule exI[of - nrows a], simp add: nrows-le foldseq-zero)
      apply (rule exI[of - ncols v], simp add: ncols-le foldseq-zero)
      apply (subst RepAbs-matrix)
      apply (simp, auto)
      apply (rule exI[of - nrows a], simp add: nrows-le foldseq-zero)
      apply (rule exI[of - ncols u], simp add: ncols-le foldseq-zero)
      done
  qed
qed

lemma l-distributive-matrix:
 assumes prems:
  l-distributive fmul fadd
  associative fadd
  commutative fadd
  fadd 0 0 = 0
  ! a. fmul a 0 = 0
  ! a. fmul 0 a = 0
 shows l-distributive (mult-matrix fmul fadd) (combine-matrix fadd) (is ?concl)
proof -
  from prems show ?concl
    apply (simp add: l-distributive-def mult-matrix-def, auto)
    proof -
      fix a::'b matrix
      fix u::'a matrix
      fix v::'a matrix
      let ?mx = max (nrows a) (max (ncols u) (ncols v))
       from prems show mult-matrix-n (max (ncols (combine-matrix fadd u v))
(nrows a)) fmul fadd (combine-matrix fadd u v) a =
              combine-matrix fadd (mult-matrix-n (max (ncols u) (nrows a)) fmul
fadd u a) (mult-matrix-n (max (ncols v) (nrows a)) fmul fadd v a)
      apply (subst mult-matrix-nm[of v - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-nrows combine-ncols)+
      apply (subst mult-matrix-nm[of u - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-nrows combine-ncols)+
      apply (subst mult-matrix-nm[of - - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-nrows combine-ncols)+
      apply (simp add: mult-matrix-n-def l-distributive-def foldseq-distr[of fadd])
      apply (simp add: combine-matrix-def combine-infmatrix-def)
      apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
      apply (simplesubst RepAbs-matrix)
      apply (simp, auto)
      apply (rule exI[of - nrows v], simp add: nrows-le foldseq-zero)
      apply (rule exI[of - ncols a], simp add: ncols-le foldseq-zero)
      apply (subst RepAbs-matrix)
      apply (simp, auto)
```

```
      apply (rule exI[of - nrows u], simp add: nrows-le foldseq-zero)
      apply (rule exI[of - ncols a], simp add: ncols-le foldseq-zero)
      done
    qed
qed

instance matrix :: (zero) zero ..

defs(overloaded)
  zero-matrix-def: (0::('a::zero) matrix) == Abs-matrix(% j i. 0)

lemma Rep-zero-matrix-def[simp]: Rep-matrix 0 j i = 0
  apply (simp add: zero-matrix-def)
  apply (subst RepAbs-matrix)
  by (auto)

lemma zero-matrix-def-nrows[simp]: nrows 0 = 0
proof −
  have a:!! (x::nat). x <= 0 ⟹ x = 0 by (arith)
  show nrows 0 = 0 by (rule a, subst nrows-le, simp)
qed

lemma zero-matrix-def-ncols[simp]: ncols 0 = 0
proof −
  have a:!! (x::nat). x <= 0 ⟹ x = 0 by (arith)
  show ncols 0 = 0 by (rule a, subst ncols-le, simp)
qed

lemma combine-matrix-zero-l-neutral: zero-l-neutral f ⟹ zero-l-neutral (combine-matrix
f)
  by (simp add: zero-l-neutral-def combine-matrix-def combine-infmatrix-def)

lemma combine-matrix-zero-r-neutral: zero-r-neutral f ⟹ zero-r-neutral (combine-matrix
f)
  by (simp add: zero-r-neutral-def combine-matrix-def combine-infmatrix-def)

lemma mult-matrix-zero-closed: ⟦fadd 0 0 = 0; zero-closed fmul⟧ ⟹ zero-closed
(mult-matrix fmul fadd)
  apply (simp add: zero-closed-def mult-matrix-def mult-matrix-n-def)
  apply (auto)
  by (subst foldseq-zero, (simp add: zero-matrix-def)+)+

lemma mult-matrix-n-zero-right[simp]: ⟦fadd 0 0 = 0; !a. fmul a 0 = 0⟧ ⟹
mult-matrix-n n fmul fadd A 0 = 0
  apply (simp add: mult-matrix-n-def)
  apply (subst foldseq-zero)
  by (simp-all add: zero-matrix-def)

lemma mult-matrix-n-zero-left[simp]: ⟦fadd 0 0 = 0; !a. fmul 0 a = 0⟧ ⟹
```

mult-matrix-n n fmul fadd 0 A = 0
  apply (*simp add*: *mult-matrix-n-def*)
  apply (*subst foldseq-zero*)
  by (*simp-all add*: *zero-matrix-def*)

lemma *mult-matrix-zero-left*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul 0 a = 0*⟧ ⟹ *mult-matrix fmul fadd 0 A = 0*
by (*simp add*: *mult-matrix-def*)

lemma *mult-matrix-zero-right*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul a 0 = 0*⟧ ⟹ *mult-matrix fmul fadd A 0 = 0*
by (*simp add*: *mult-matrix-def*)

lemma *apply-matrix-zero*[*simp*]: *f 0 = 0* ⟹ *apply-matrix f 0 = 0*
  apply (*simp add*: *apply-matrix-def apply-infmatrix-def*)
  by (*simp add*: *zero-matrix-def*)

lemma *combine-matrix-zero*: *f 0 0 = 0* ⟹ *combine-matrix f 0 0 = 0*
  apply (*simp add*: *combine-matrix-def combine-infmatrix-def*)
  by (*simp add*: *zero-matrix-def*)

lemma *transpose-matrix-zero*[*simp*]: *transpose-matrix 0 = 0*
apply (*simp add*: *transpose-matrix-def transpose-infmatrix-def zero-matrix-def RepAbs-matrix*)
apply (*subst Rep-matrix-inject*[*symmetric*], (*rule ext*)+)
apply (*simp add*: *RepAbs-matrix*)
done

lemma *apply-zero-matrix-def*[*simp*]: *apply-matrix* (% *x. 0*) *A = 0*
  apply (*simp add*: *apply-matrix-def apply-infmatrix-def*)
  by (*simp add*: *zero-matrix-def*)

constdefs
  singleton-matrix :: *nat* ⇒ *nat* ⇒ ('*a*::*zero*) ⇒ '*a matrix*
  singleton-matrix j i a == *Abs-matrix*(% *m n. if j = m & i = n then a else 0*)
  move-matrix :: ('*a*::*zero*) *matrix* ⇒ *int* ⇒ *int* ⇒ '*a matrix*
  move-matrix A y x == *Abs-matrix*(% *j i. if* (*neg* ((*int j*)−*y*)) | (*neg* ((*int i*)−*x*)) *then 0 else Rep-matrix A* (*nat* ((*int j*)−*y*)) (*nat* ((*int i*)−*x*)))
  take-rows :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
  take-rows A r == *Abs-matrix*(% *j i. if* (*j* < *r*) *then* (*Rep-matrix A j i*) *else 0*)
  take-columns :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
  take-columns A c == *Abs-matrix*(% *j i. if* (*i* < *c*) *then* (*Rep-matrix A j i*) *else 0*)

constdefs
  column-of-matrix :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
  column-of-matrix A n == *take-columns* (*move-matrix A 0* (− *int n*)) *1*
  row-of-matrix :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
  row-of-matrix A m == *take-rows* (*move-matrix A* (− *int m*) *0*) *1*

lemma *Rep-singleton-matrix*[*simp*]: *Rep-matrix* (*singleton-matrix j i e*) *m n* = (*if* *j* = *m* & *i* = *n then e else 0*)
apply (*simp add*: *singleton-matrix-def*)
apply (*auto*)
apply (*subst RepAbs-matrix*)
apply (*rule exI*[*of - Suc m*], *simp*)
apply (*rule exI*[*of - Suc n*], *simp+*)
by (*subst RepAbs-matrix*, *rule exI*[*of - Suc j*], *simp*, *rule exI*[*of - Suc i*], *simp+*)+

lemma *apply-singleton-matrix*[*simp*]: *f 0* = *0* $\Longrightarrow$ *apply-matrix f* (*singleton-matrix* *j i x*) = (*singleton-matrix j i* (*f x*))
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *singleton-matrix-zero*[*simp*]: *singleton-matrix j i 0* = *0*
by (*simp add*: *singleton-matrix-def zero-matrix-def*)

lemma *nrows-singleton*[*simp*]: *nrows*(*singleton-matrix j i e*) = (*if e* = *0 then 0* *else Suc j*)
apply (*auto*)
apply (*rule le-anti-sym*)
apply (*subst nrows-le*)
apply (*simp add*: *singleton-matrix-def*, *auto*)
apply (*subst RepAbs-matrix*)
apply (*simp*, *arith*)
apply (*simp*, *arith*)
apply (*simp*)
apply (*simp add*: *Suc-le-eq*)
apply (*rule not-leE*)
apply (*subst nrows-le*)
by *simp*

lemma *ncols-singleton*[*simp*]: *ncols*(*singleton-matrix j i e*) = (*if e* = *0 then 0 else* *Suc i*)
apply (*auto*)
apply (*rule le-anti-sym*)
apply (*subst ncols-le*)
apply (*simp add*: *singleton-matrix-def*, *auto*)
apply (*subst RepAbs-matrix*)
apply (*simp*, *arith*)
apply (*simp*, *arith*)
apply (*simp*)
apply (*simp add*: *Suc-le-eq*)
apply (*rule not-leE*)
apply (*subst ncols-le*)
by *simp*

lemma *combine-singleton*: *f 0 0* = *0* ⟹ *combine-matrix f* (*singleton-matrix j i a*) (*singleton-matrix j i b*) = *singleton-matrix j i* (*f a b*)
apply (*simp add*: *singleton-matrix-def combine-matrix-def combine-infmatrix-def*)
apply (*subst RepAbs-matrix*)
apply (*rule exI*[*of - Suc j*], *simp*)
apply (*rule exI*[*of - Suc i*], *simp*)
apply (*rule comb*[*of Abs-matrix Abs-matrix*], *simp*, (*rule ext*)+)
apply (*subst RepAbs-matrix*)
apply (*rule exI*[*of - Suc j*], *simp*)
apply (*rule exI*[*of - Suc i*], *simp*)
by *simp*

lemma *transpose-singleton*[*simp*]: *transpose-matrix* (*singleton-matrix j i a*) = *singleton-matrix i j a*
apply (*subst Rep-matrix-inject*[*symmetric*], (*rule ext*)+)
apply (*simp*)
done

lemma *Rep-move-matrix*[*simp*]:
Rep-matrix (*move-matrix A y x*) *j i* =
(*if* (*neg* ((*int j*)−*y*)) | (*neg* ((*int i*)−*x*)) *then 0 else Rep-matrix A* (*nat*((*int j*)−*y*)) (*nat*((*int i*)−*x*)))
apply (*simp add*: *move-matrix-def*)
apply (*auto*)
by (*subst RepAbs-matrix*,
rule exI[*of -* (*nrows A*)+(*nat* (*abs y*))], *auto*, *rule nrows*, *arith*,
rule exI[*of -* (*ncols A*)+(*nat* (*abs x*))], *auto*, *rule ncols*, *arith*)+

lemma *move-matrix-0-0*[*simp*]: *move-matrix A 0 0* = *A*
by (*simp add*: *move-matrix-def*)

lemma *move-matrix-ortho*: *move-matrix A j i* = *move-matrix* (*move-matrix A j 0*) *0 i*
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *transpose-move-matrix*[*simp*]:
transpose-matrix (*move-matrix A x y*) = *move-matrix* (*transpose-matrix A*) *y x*
apply (*subst Rep-matrix-inject*[*symmetric*], (*rule ext*)+)
apply (*simp*)
done

lemma *move-matrix-singleton*[*simp*]: *move-matrix* (*singleton-matrix u v x*) *j i* =
(*if* (*j* + *int u* < *0*) | (*i* + *int v* < *0*) *then 0 else* (*singleton-matrix* (*nat* (*j* + *int u*)) (*nat* (*i* + *int v*)) *x*))
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+

apply (*case-tac* $j + int\ u < 0$)
apply (*simp*, *arith*)
apply (*case-tac* $i + int\ v < 0$)
apply (*simp add*: *neg-def*, *arith*)
apply (*simp add*: *neg-def*)
apply *arith*
done

lemma *Rep-take-columns*[*simp*]:
Rep-matrix (*take-columns A c*) *j i* =
(*if i* < *c then* (*Rep-matrix A j i*) *else 0*)
apply (*simp add*: *take-columns-def*)
apply (*simplesubst RepAbs-matrix*)
apply (*rule exI*[*of - nrows A*], *auto*, *simp add*: *nrows-le*)
apply (*rule exI*[*of - ncols A*], *auto*, *simp add*: *ncols-le*)
done

lemma *Rep-take-rows*[*simp*]:
Rep-matrix (*take-rows A r*) *j i* =
(*if j* < *r then* (*Rep-matrix A j i*) *else 0*)
apply (*simp add*: *take-rows-def*)
apply (*simplesubst RepAbs-matrix*)
apply (*rule exI*[*of - nrows A*], *auto*, *simp add*: *nrows-le*)
apply (*rule exI*[*of - ncols A*], *auto*, *simp add*: *ncols-le*)
done

lemma *Rep-column-of-matrix*[*simp*]:
Rep-matrix (*column-of-matrix A c*) *j i* = (*if i* = *0 then* (*Rep-matrix A j c*) *else 0*)
by (*simp add*: *column-of-matrix-def*)

lemma *Rep-row-of-matrix*[*simp*]:
Rep-matrix (*row-of-matrix A r*) *j i* = (*if j* = *0 then* (*Rep-matrix A r i*) *else 0*)
by (*simp add*: *row-of-matrix-def*)

lemma *column-of-matrix*: *ncols A* <= *n* $\Longrightarrow$ *column-of-matrix A n* = *0*
apply (*subst Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by (*simp add*: *ncols*)

lemma *row-of-matrix*: *nrows A* <= *n* $\Longrightarrow$ *row-of-matrix A n* = *0*
apply (*subst Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by (*simp add*: *nrows*)

lemma *mult-matrix-singleton-right*[*simp*]:
assumes *prems*:
! *x*. *fmul x 0* = *0*
! *x*. *fmul 0 x* = *0*

! x. *fadd 0* $x = x$
! x. *fadd* x *0* $= x$
shows (*mult-matrix fmul fadd A* (*singleton-matrix j i e*)) = *apply-matrix* (% x. *fmul x e*) (*move-matrix* (*column-of-matrix A j*) *0* (*int i*))
apply (*simp add*: *mult-matrix-def*)
apply (*subst mult-matrix-nm*[*of - - - max* (*ncols A*) (*Suc j*)])
apply (*simp add*: *max-def*)+
apply (*auto*)
apply (*simp add*: *prems*)+
apply (*simp add*: *mult-matrix-n-def apply-matrix-def apply-infmatrix-def*)
apply (*rule comb*[*of Abs-matrix Abs-matrix*], *auto*, (*rule ext*)+)
apply (*subst foldseq-almostzero*[*of - j*])
apply (*simp add*: *prems*)+
apply (*auto*)
apply (*insert ncols-le*[*of A j*])
apply (*arith*)
proof −
  fix k
  fix l
  assume *a*:~*neg*(*int l* − *int i*)
  assume *b*:*nat* (*int l* − *int i*) = *0*
  from *a b* **have** *a*: $l = i$ **by**(*insert not-neg-nat*[*of int l* − *int i*], *simp add*: *a b*)
  assume *c*: $i \neq l$
  from *c a* **show** *fmul* (*Rep-matrix A k j*) *e* = *0* **by** *blast*
qed

lemma *mult-matrix-ext*:
assumes
eprem:
? e. (! a b. $a \neq b \longrightarrow$ *fmul a e* $\neq$ *fmul b e*)
and *fprems*:
! a. *fmul 0* a = *0*
! a. *fmul* a *0* = *0*
! a. *fadd* a *0* = a
! a. *fadd 0* a = a
and *contraprems*:
mult-matrix fmul fadd A = *mult-matrix fmul fadd B*
shows
$A = B$
proof(*rule contrapos-np*[*of False*], *simp*)
assume *a*: $A \neq B$
have *b*: !! *f g*. (! *x y*. *f x y* = *g x y*) $\Longrightarrow$ $f = g$ **by** ((*rule ext*)+, *auto*)
have *? j i*. (*Rep-matrix A j i*) $\neq$ (*Rep-matrix B j i*)
  apply (*rule contrapos-np*[*of False*], *simp*+)
  apply (*insert b*[*of Rep-matrix A Rep-matrix B*], *simp*)
  by (*simp add*: *Rep-matrix-inject a*)
then obtain *J I* **where** *c*:(*Rep-matrix A J I*) $\neq$ (*Rep-matrix B J I*) **by** *blast*
from *eprem* **obtain** *e* **where** *eprops*:(! a b. $a \neq b \longrightarrow$ *fmul a e* $\neq$ *fmul b e*) **by** *blast*

let *?S* = *singleton-matrix I 0 e*
let *?comp* = *mult-matrix fmul fadd*
have *d*: !!*x f g*. *f* = *g* $\Longrightarrow$ *f x* = *g x* **by** *blast*
have *e*: (% *x*. *fmul x e*) *0* = *0* **by** (*simp add*: *prems*)
have ~(*?comp A ?S* = *?comp B ?S*)
apply (*rule notI*)
apply (*simp add*: *fprems eprops*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*drule d*[*of - - J*], *drule d*[*of - - 0*])
by (*simp add*: *e c eprops*)
with *contraprems* **show** *False* **by** *simp*
qed

constdefs
foldmatrix :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) $\Rightarrow$ ('*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) $\Rightarrow$ ('*a infmatrix*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ '*a*
foldmatrix f g A m n == *foldseq-transposed g* (% *j*. *foldseq f* (*A j*) *n*) *m*
foldmatrix-transposed :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) $\Rightarrow$ ('*a* $\Rightarrow$ '*a* $\Rightarrow$ '*a*) $\Rightarrow$ ('*a infmatrix*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ '*a*
foldmatrix-transposed f g A m n == *foldseq g* (% *j*. *foldseq-transposed f* (*A j*) *n*) *m*

lemma *foldmatrix-transpose*:
assumes
! *a b c d*. *g*(*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*)
shows
foldmatrix f g A m n = *foldmatrix-transposed g f* (*transpose-infmatrix A*) *n m* (**is** *?concl*)
proof −
have *forall*:!! *P x*. (! *x*. *P x*) $\Longrightarrow$ *P x* **by** *auto*
have *tworows*:! *A*. *foldmatrix f g A 1 n* = *foldmatrix-transposed g f* (*transpose-infmatrix A*) *n 1*
apply (*induct n*)
apply (*simp add*: *foldmatrix-def foldmatrix-transposed-def prems*)+
apply (*auto*)
by (*drule-tac x*=(% *j i*. *A j* (*Suc i*)) **in** *forall*, *simp*)
show *foldmatrix f g A m n* = *foldmatrix-transposed g f* (*transpose-infmatrix A*) *n m*
apply (*simp add*: *foldmatrix-def foldmatrix-transposed-def*)
apply (*induct m*, *simp*)
apply (*simp*)
apply (*insert tworows*)
apply (*drule-tac x*=% *j i*. (*if j* = *0 then* (*foldseq-transposed g* (λ*u*. *A u i*) *m*) *else* (*A* (*Suc m*) *i*)) **in** *spec*)
by (*simp add*: *foldmatrix-def foldmatrix-transposed-def*)
qed

lemma *foldseq-foldseq*:
assumes

associative f
associative g
! *a b c d. g(f a b) (f c d) = f (g a c) (g b d)*
shows
foldseq g (% j. foldseq f (A j) n) m = foldseq f (% j. foldseq g ((transpose-infmatrix A) j) m) n
apply (*insert foldmatrix-transpose[of g f A m n]*)
by (*simp add: foldmatrix-def foldmatrix-transposed-def foldseq-assoc[THEN sym] prems*)

lemma *mult-n-nrows*:
assumes
! *a. fmul 0 a = 0*
! *a. fmul a 0 = 0*
fadd 0 0 = 0
shows *nrows (mult-matrix-n n fmul fadd A B) ≤ nrows A*
apply (*subst nrows-le*)
apply (*simp add: mult-matrix-n-def*)
apply (*subst RepAbs-matrix*)
apply (*rule-tac x=nrows A* **in** *exI*)
apply (*simp add: nrows prems foldseq-zero*)
apply (*rule-tac x=ncols B* **in** *exI*)
apply (*simp add: ncols prems foldseq-zero*)
by (*simp add: nrows prems foldseq-zero*)

lemma *mult-n-ncols*:
assumes
! *a. fmul 0 a = 0*
! *a. fmul a 0 = 0*
fadd 0 0 = 0
shows *ncols (mult-matrix-n n fmul fadd A B) ≤ ncols B*
apply (*subst ncols-le*)
apply (*simp add: mult-matrix-n-def*)
apply (*subst RepAbs-matrix*)
apply (*rule-tac x=nrows A* **in** *exI*)
apply (*simp add: nrows prems foldseq-zero*)
apply (*rule-tac x=ncols B* **in** *exI*)
apply (*simp add: ncols prems foldseq-zero*)
by (*simp add: ncols prems foldseq-zero*)

lemma *mult-nrows*:
assumes
! *a. fmul 0 a = 0*
! *a. fmul a 0 = 0*
fadd 0 0 = 0
shows *nrows (mult-matrix fmul fadd A B) ≤ nrows A*
by (*simp add: mult-matrix-def mult-n-nrows prems*)

lemma *mult-ncols*:

assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows *ncols* (*mult-matrix fmul fadd A B*) ≤ *ncols B*
by (*simp add*: *mult-matrix-def mult-n-ncols prems*)

lemma *nrows-move-matrix-le*: *nrows* (*move-matrix A j i*) <= *nat*((*int* (*nrows A*)) + *j*)
  apply (*auto simp add*: *nrows-le*)
  apply (*rule nrows*)
  apply (*arith*)
  done

lemma *ncols-move-matrix-le*: *ncols* (*move-matrix A j i*) <= *nat*((*int* (*ncols A*)) + *i*)
  apply (*auto simp add*: *ncols-le*)
  apply (*rule ncols*)
  apply (*arith*)
  done

lemma *mult-matrix-assoc*:
  assumes *prems*:
  ! *a*. *fmul1 0 a* = *0*
  ! *a*. *fmul1 a 0* = *0*
  ! *a*. *fmul2 0 a* = *0*
  ! *a*. *fmul2 a 0* = *0*
  fadd1 0 0 = *0*
  fadd2 0 0 = *0*
  ! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
  associative fadd1
  associative fadd2
  ! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
  ! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
  ! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)
  shows *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*) *C* = *mult-matrix fmul1 fadd1 A* (*mult-matrix fmul2 fadd2 B C*) (**is** *?concl*)
proof −
  have *comb-left*: !! *A B x y*. *A* = *B* ⟹ (*Rep-matrix* (*Abs-matrix A*)) *x y* = (*Rep-matrix*(*Abs-matrix B*)) *x y* **by** *blast*
  have *fmul2fadd1fold*: !! *x s n*. *fmul2* (*foldseq fadd1 s n*) *x* = *foldseq fadd1* (% *k*. *fmul2* (*s k*) *x*) *n*
    by (*rule-tac g1* = % *y*. *fmul2 y x* **in** *ssubst* [*OF foldseq-distr-unary*], *simp-all*!)
  have *fmul1fadd2fold*: !! *x s n*. *fmul1 x* (*foldseq fadd2 s n*) = *foldseq fadd2* (% *k*. *fmul1 x* (*s k*)) *n*
    by (*rule-tac g1* = % *y*. *fmul1 x y* **in** *ssubst* [*OF foldseq-distr-unary*], *simp-all*!)
  let *?N* = *max* (*ncols A*) (*max* (*ncols B*) (*max* (*nrows B*) (*nrows C*)))
  show *?concl*
    apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])

apply (*rule ext*)+
apply (*simp add*: *mult-matrix-def*)
apply (*simplesubst mult-matrix-nm*[*of* - *max* (*ncols* (*mult-matrix-n* (*max* (*ncols* *A*) (*nrows* *B*)) *fmul1 fadd1 A B*)) (*nrows* *C*) - *max* (*ncols* *B*) (*nrows* *C*)])
apply (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simplesubst mult-matrix-nm*[*of* - *max* (*ncols* *A*) (*nrows* (*mult-matrix-n* (*max* (*ncols* *B*) (*nrows* *C*)) *fmul2 fadd2 B C*)) - *max* (*ncols* *A*) (*nrows* *B*)]) **apply** (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simplesubst mult-matrix-nm*[*of* - - - *?N*])
apply (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simplesubst mult-matrix-nm*[*of* - - - *?N*])
apply (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simplesubst mult-matrix-nm*[*of* - - - *?N*])
apply (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simplesubst mult-matrix-nm*[*of* - - - *?N*])
apply (*simp add*: *max1 max2 mult-n-ncols mult-n-nrows prems*)+
apply (*simp add*: *mult-matrix-n-def*)
apply (*rule comb-left*)
apply ((*rule ext*)+, *simp*)
apply (*simplesubst RepAbs-matrix*)
apply (*rule exI*[*of* - *nrows B*])
apply (*simp add*: *nrows prems foldseq-zero*)
apply (*rule exI*[*of* - *ncols C*])
apply (*simp add*: *prems ncols foldseq-zero*)
apply (*subst RepAbs-matrix*)
apply (*rule exI*[*of* - *nrows A*])
apply (*simp add*: *nrows prems foldseq-zero*)
apply (*rule exI*[*of* - *ncols B*])
apply (*simp add*: *prems ncols foldseq-zero*)
apply (*simp add*: *fmul2fadd1fold fmul1fadd2fold prems*)
apply (*subst foldseq-foldseq*)
apply (*simp add*: *prems*)+
by (*simp add*: *transpose-infmatrix*)
qed

lemma
assumes *prems*:
! *a*. *fmul1 0 a* = *0*
! *a*. *fmul1 a 0* = *0*
! *a*. *fmul2 0 a* = *0*
! *a*. *fmul2 a 0* = *0*
fadd1 0 0 = *0*
fadd2 0 0 = *0*
! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
associative fadd1
associative fadd2
! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)

shows
(*mult-matrix fmul1 fadd1 A*) *o* (*mult-matrix fmul2 fadd2 B*) = *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*)
apply (*rule ext*)+
apply (*simp add*: *comp-def*)
by (*simp add*: *mult-matrix-assoc prems*)

lemma *mult-matrix-assoc-simple*:
assumes *prems*:
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
associative fadd
commutative fadd
associative fmul
distributive fmul fadd
shows *mult-matrix fmul fadd* (*mult-matrix fmul fadd A B*) *C* = *mult-matrix fmul fadd A* (*mult-matrix fmul fadd B C*) (**is** *?concl*)
proof −
have !! *a b c d*. *fadd* (*fadd a b*) (*fadd c d*) = *fadd* (*fadd a c*) (*fadd b d*)
by (*simp*! *add*: *associative-def commutative-def*)
then show *?concl*
apply (*subst mult-matrix-assoc*)
apply (*simp-all*!)
by (*simp add*: *associative-def distributive-def l-distributive-def r-distributive-def*)+
qed

lemma *transpose-apply-matrix*: *f 0* = *0* ⟹ *transpose-matrix* (*apply-matrix f A*) = *apply-matrix f* (*transpose-matrix A*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by *simp*

lemma *transpose-combine-matrix*: *f 0 0* = *0* ⟹ *transpose-matrix* (*combine-matrix f A B*) = *combine-matrix f* (*transpose-matrix A*) (*transpose-matrix B*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by *simp*

lemma *Rep-mult-matrix*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows
Rep-matrix(*mult-matrix fmul fadd A B*) *j i* =
foldseq fadd (% *k*. *fmul* (*Rep-matrix A j k*) (*Rep-matrix B k i*)) (*max* (*ncols A*) (*nrows B*))
apply (*simp add*: *mult-matrix-def mult-matrix-n-def*)

apply (*subst RepAbs-matrix*)
apply (*rule exI*[*of - nrows A*], *simp*! *add*: *nrows foldseq-zero*)
apply (*rule exI*[*of - ncols B*], *simp*! *add*: *ncols foldseq-zero*)
by *simp*

lemma *transpose-mult-matrix*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
! *x y*. *fmul y x* = *fmul x y*
shows
transpose-matrix (*mult-matrix fmul fadd A B*) = *mult-matrix fmul fadd* (*transpose-matrix B*) (*transpose-matrix A*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by (*simp*! *add*: *Rep-mult-matrix max-ac*)

lemma *column-transpose-matrix*: *column-of-matrix* (*transpose-matrix A*) *n* = *transpose-matrix* (*row-of-matrix A n*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by *simp*

lemma *take-columns-transpose-matrix*: *take-columns* (*transpose-matrix A*) *n* = *transpose-matrix* (*take-rows A n*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
by *simp*

instance *matrix* :: ({*ord*, *zero*}) *ord* **..**

defs (**overloaded**)
le-matrix-def: (*A*::('*a*::{*ord*,*zero*}) *matrix*) <= *B* == ! *j i*. (*Rep-matrix A j i*) <= (*Rep-matrix B j i*)
less-def: (*A*::('*a*::{*ord*,*zero*}) *matrix*) < *B* == (*A* <= *B*) & (*A* ≠ *B*)

instance *matrix* :: ({*order*, *zero*}) *order*
apply *intro-classes*
apply (*simp-all add*: *le-matrix-def less-def*)
apply (*auto*)
apply (*drule-tac x*=*j* **in** *spec*, *drule-tac x*=*j* **in** *spec*)
apply (*drule-tac x*=*i* **in** *spec*, *drule-tac x*=*i* **in** *spec*)
apply (*simp*)
apply (*simp add*: *Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
apply (*drule-tac x*=*xa* **in** *spec*, *drule-tac x*=*xa* **in** *spec*)
apply (*drule-tac x*=*xb* **in** *spec*, *drule-tac x*=*xb* **in** *spec*)
by *simp*

lemma *le-apply-matrix*:
  assumes
  $f\ 0 = 0$
  $!\ x\ y.\ x <= y \longrightarrow f\ x <= f\ y$
  $(a::('a::\{ord,\ zero\})\ matrix) <= b$
  shows
  $apply\text{-}matrix\ f\ a <= apply\text{-}matrix\ f\ b$
  by (*simp*! *add*: *le-matrix-def*)

lemma *le-combine-matrix*:
  assumes
  $f\ 0\ 0 = 0$
  $!\ a\ b\ c\ d.\ a <= b\ \&\ c <= d \longrightarrow f\ a\ c <= f\ b\ d$
  $A <= B$
  $C <= D$
  shows
  $combine\text{-}matrix\ f\ A\ C <= combine\text{-}matrix\ f\ B\ D$
by (*simp*! *add*: *le-matrix-def*)

lemma *le-left-combine-matrix*:
  assumes
  $f\ 0\ 0 = 0$
  $!\ a\ b\ c.\ a <= b \longrightarrow f\ c\ a <= f\ c\ b$
  $A <= B$
  shows
  $combine\text{-}matrix\ f\ C\ A <= combine\text{-}matrix\ f\ C\ B$
  by (*simp*! *add*: *le-matrix-def*)

lemma *le-right-combine-matrix*:
  assumes
  $f\ 0\ 0 = 0$
  $!\ a\ b\ c.\ a <= b \longrightarrow f\ a\ c <= f\ b\ c$
  $A <= B$
  shows
  $combine\text{-}matrix\ f\ A\ C <= combine\text{-}matrix\ f\ B\ C$
  by (*simp*! *add*: *le-matrix-def*)

lemma *le-transpose-matrix*: $(A <= B) = (transpose\text{-}matrix\ A <= transpose\text{-}matrix\ B)$
  by (*simp add*: *le-matrix-def*, *auto*)

lemma *le-foldseq*:
  assumes
  $!\ a\ b\ c\ d\ .\ a <= b\ \&\ c <= d \longrightarrow f\ a\ c <= f\ b\ d$
  $!\ i.\ i <= n \longrightarrow s\ i <= t\ i$
  shows
  $foldseq\ f\ s\ n <= foldseq\ f\ t\ n$
proof −

have ! *s t*. (! *i*. *i*<=*n* ⟶ *s i* <= *t i*) ⟶ *foldseq f s n* <= *foldseq f t n* **by** (*induct-tac n*, *simp-all*!)
then show *foldseq f s n* <= *foldseq f t n* **by** (*simp*!)
qed

lemma *le-left-mult*:
assumes
! *a b c d*. *a* <= *b* & *c* <= *d* ⟶ *fadd a c* <= *fadd b d*
! *c a b*. *0* <= *c* & *a* <= *b* ⟶ *fmul c a* <= *fmul c b*
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
0 <= *C*
A <= *B*
shows
mult-matrix fmul fadd C A <= *mult-matrix fmul fadd C B*
apply (*simp*! *add*: *le-matrix-def Rep-mult-matrix*)
apply (*auto*)
apply (*simplesubst foldseq-zerotail*[*of* - - - *max* (*ncols C*) (*max* (*nrows A*) (*nrows B*))], *simp-all add*: *nrows ncols max1 max2*)+
apply (*rule le-foldseq*)
by (*auto*)

lemma *le-right-mult*:
assumes
! *a b c d*. *a* <= *b* & *c* <= *d* ⟶ *fadd a c* <= *fadd b d*
! *c a b*. *0* <= *c* & *a* <= *b* ⟶ *fmul a c* <= *fmul b c*
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
0 <= *C*
A <= *B*
shows
mult-matrix fmul fadd A C <= *mult-matrix fmul fadd B C*
apply (*simp*! *add*: *le-matrix-def Rep-mult-matrix*)
apply (*auto*)
apply (*simplesubst foldseq-zerotail*[*of* - - - *max* (*nrows C*) (*max* (*ncols A*) (*ncols B*))], *simp-all add*: *nrows ncols max1 max2*)+
apply (*rule le-foldseq*)
by (*auto*)

lemma *spec2*: ! *j i*. *P j i* ⟹ *P j i* **by** *blast*
lemma *neg-imp*: (¬ *Q* ⟶ ¬ *P*) ⟹ *P* ⟶ *Q* **by** *blast*

lemma *singleton-matrix-le*[*simp*]: (*singleton-matrix j i a* <= *singleton-matrix j i b*) = (*a* <= (*b*::-::*order*))
by (*auto simp add*: *le-matrix-def*)

lemma *singleton-le-zero*[*simp*]: (*singleton-matrix j i x* <= *0*) = (*x* <= (*0*::'*a*::{*order*,*zero*}))

apply (*auto*)
apply (*simp add*: *le-matrix-def*)
apply (*drule-tac j=j* **and** *i=i* **in** *spec2*)
apply (*simp*)
apply (*simp add*: *le-matrix-def*)
done

lemma *singleton-ge-zero*[*simp*]: (*0* <= *singleton-matrix j i x*) = ((*0*::'*a*::{*order*,*zero*}) <= *x*)
apply (*auto*)
apply (*simp add*: *le-matrix-def*)
apply (*drule-tac j=j* **and** *i=i* **in** *spec2*)
apply (*simp*)
apply (*simp add*: *le-matrix-def*)
done

lemma *move-matrix-le-zero*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*move-matrix A j i* <= *0*) = (*A* <= (*0*::('*a*::{*order*,*zero*}) *matrix*))
apply (*auto simp add*: *le-matrix-def neg-def*)
apply (*drule-tac j=ja+*(*nat j*) **and** *i=ia+*(*nat i*) **in** *spec2*)
apply (*auto*)
done

lemma *move-matrix-zero-le*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*0* <= *move-matrix A j i*) = ((*0*::('*a*::{*order*,*zero*}) *matrix*) <= *A*)
apply (*auto simp add*: *le-matrix-def neg-def*)
apply (*drule-tac j=ja+*(*nat j*) **and** *i=ia+*(*nat i*) **in** *spec2*)
apply (*auto*)
done

lemma *move-matrix-le-move-matrix-iff*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*move-matrix A j i* <= *move-matrix B j i*) = (*A* <= (*B*::('*a*::{*order*,*zero*}) *matrix*))
apply (*auto simp add*: *le-matrix-def neg-def*)
apply (*drule-tac j=ja+*(*nat j*) **and** *i=ia+*(*nat i*) **in** *spec2*)
apply (*auto*)
done

end

theory *Matrix=MatrixGeneral*:

instance *matrix* :: (*minus*) *minus*
by *intro-classes*

instance *matrix* :: (*plus*) *plus*
by (*intro-classes*)

instance *matrix* :: ({*plus*,*times*}) *times*
by (*intro-classes*)

defs (**overloaded**)
plus-matrix-def: $A + B == $ *combine-matrix* (*op* +) A B
diff-matrix-def: $A - B == $ *combine-matrix* (*op* −) A B
minus-matrix-def: $- A == $ *apply-matrix uminus* A
times-matrix-def: $A * B == $ *mult-matrix* (*op* ∗) (*op* +) A B

lemma *is-meet-combine-matrix-meet*: *is-meet* (*combine-matrix meet*)
by (*simp-all add*: *is-meet-def le-matrix-def meet-left-le meet-right-le meet-imp-le*)

lemma *is-join-combine-matrix-join*: *is-join* (*combine-matrix join*)
by (*simp-all add*: *is-join-def le-matrix-def join-left-le join-right-le join-imp-le*)

instance *matrix* :: (*lordered-ab-group*) *lordered-ab-group-meet*
proof
fix A B C :: ('*a*::*lordered-ab-group*) *matrix*
show $A + B + C = A + (B + C)$
apply (*simp add*: *plus-matrix-def*)
apply (*rule combine-matrix-assoc*[*simplified associative-def*, *THEN spec*, *THEN spec*, *THEN spec*])
apply (*simp-all add*: *add-assoc*)
done
show $A + B = B + A$
apply (*simp add*: *plus-matrix-def*)
apply (*rule combine-matrix-commute*[*simplified commutative-def*, *THEN spec*, *THEN spec*])
apply (*simp-all add*: *add-commute*)
done
show $0 + A = A$
apply (*simp add*: *plus-matrix-def*)
apply (*rule combine-matrix-zero-l-neutral*[*simplified zero-l-neutral-def*, *THEN spec*])
apply (*simp*)
done
show $- A + A = 0$
by (*simp add*: *plus-matrix-def minus-matrix-def Rep-matrix-inject*[*symmetric*] *ext*)
show $A - B = A + - B$
by (*simp add*: *plus-matrix-def diff-matrix-def minus-matrix-def Rep-matrix-inject*[*symmetric*] *ext*)
show ∃ *m*::'*a matrix* ⇒ '*a matrix* ⇒ '*a matrix*. *is-meet m*
by (*auto intro*: *is-meet-combine-matrix-meet*)
assume $A <= B$
then show $C + A <= C + B$
apply (*simp add*: *plus-matrix-def*)
apply (*rule le-left-combine-matrix*)
apply (*simp-all*)

```
  done
qed

defs (overloaded)
  abs-matrix-def: abs (A::('a::lordered-ab-group) matrix) == join A (− A)

instance matrix :: (lordered-ring) lordered-ring
proof
  fix A B C :: ('a :: lordered-ring) matrix
  show A ∗ B ∗ C = A ∗ (B ∗ C)
    apply (simp add: times-matrix-def)
    apply (rule mult-matrix-assoc)
    apply (simp-all add: associative-def ring-eq-simps)
    done
  show (A + B) ∗ C = A ∗ C + B ∗ C
    apply (simp add: times-matrix-def plus-matrix-def)
   apply (rule l-distributive-matrix[simplified l-distributive-def, THEN spec, THEN
spec, THEN spec])
    apply (simp-all add: associative-def commutative-def ring-eq-simps)
    done
  show A ∗ (B + C) = A ∗ B + A ∗ C
    apply (simp add: times-matrix-def plus-matrix-def)
   apply (rule r-distributive-matrix[simplified r-distributive-def, THEN spec, THEN
spec, THEN spec])
    apply (simp-all add: associative-def commutative-def ring-eq-simps)
    done
  show abs A = join A (−A)
    by (simp add: abs-matrix-def)
  assume a: A ≤ B
  assume b: 0 ≤ C
  from a b show C ∗ A ≤ C ∗ B
    apply (simp add: times-matrix-def)
    apply (rule le-left-mult)
    apply (simp-all add: add-mono mult-left-mono)
    done
  from a b show A ∗ C ≤ B ∗ C
    apply (simp add: times-matrix-def)
    apply (rule le-right-mult)
    apply (simp-all add: add-mono mult-right-mono)
    done
qed

lemma Rep-matrix-add[simp]: Rep-matrix ((a::('a::lordered-ab-group)matrix)+b)
j i  = (Rep-matrix a j i) + (Rep-matrix b j i)
by (simp add: plus-matrix-def)

lemma Rep-matrix-mult: Rep-matrix ((a::('a::lordered-ring) matrix) ∗ b) j i =
  foldseq (op +) (% k.  (Rep-matrix a j k) ∗ (Rep-matrix b k i)) (max (ncols a)
(nrows b))
```

apply (*simp add*: *times-matrix-def*)
apply (*simp add*: *Rep-mult-matrix*)
done

lemma *apply-matrix-add*: ! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*) $\Longrightarrow$ *f 0* = (*0*::*'a*) $\Longrightarrow$ *apply-matrix f* ((*a*::(*'a*::*lordered-ab-group*) *matrix*) + *b*) = (*apply-matrix f a*) + (*apply-matrix f b*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *singleton-matrix-add*: *singleton-matrix j i* ((*a*::-::*lordered-ab-group*)+*b*) = (*singleton-matrix j i a*) + (*singleton-matrix j i b*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *nrows-mult*: *nrows* ((*A*::(*'a*::*lordered-ring*) *matrix*) * *B*) <= *nrows A*
by (*simp add*: *times-matrix-def mult-nrows*)

lemma *ncols-mult*: *ncols* ((*A*::(*'a*::*lordered-ring*) *matrix*) * *B*) <= *ncols B*
by (*simp add*: *times-matrix-def mult-ncols*)

constdefs
  one-matrix :: *nat* $\Rightarrow$ (*'a*::{*zero*,*one*}) *matrix*
  one-matrix n == *Abs-matrix* (% *j i*. *if j* = *i* & *j* < *n then 1 else 0*)

lemma *Rep-one-matrix*[*simp*]: *Rep-matrix* (*one-matrix n*) *j i* = (*if* (*j* = *i* & *j* < *n*) *then 1 else 0*)
apply (*simp add*: *one-matrix-def*)
apply (*simplesubst RepAbs-matrix*)
apply (*rule exI*[*of - n*], *simp add*: *split-if*)+
by (*simp add*: *split-if*)

lemma *nrows-one-matrix*[*simp*]: *nrows* ((*one-matrix n*) :: (*'a*::*axclass-0-neq-1*)*matrix*) = *n* (**is** *?r* = -)
proof −
  have *?r* <= *n* **by** (*simp add*: *nrows-le*)
  moreover have *n* <= *?r* **by** (*simp add*:*le-nrows*, *arith*)
  ultimately show *?r* = *n* **by** *simp*
qed

lemma *ncols-one-matrix*[*simp*]: *ncols* ((*one-matrix n*) :: (*'a*::*axclass-0-neq-1*)*matrix*) = *n* (**is** *?r* = -)
proof −
  have *?r* <= *n* **by** (*simp add*: *ncols-le*)

moreover have *n* <= *?r* **by** (*simp add*: *le-ncols*, *arith*)
ultimately show *?r* = *n* **by** *simp*
qed

lemma *one-matrix-mult-right*[*simp*]: *ncols A* <= *n* $\Longrightarrow$ (*A*::('*a*::{*lordered-ring*,*ring-1*}) *matrix*) $*$ (*one-matrix n*) = *A*
apply (*subst Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
apply (*simp add*: *times-matrix-def Rep-mult-matrix*)
apply (*rule-tac j1*=*xa* **in** *ssubst*[*OF foldseq-almostzero*])
apply (*simp-all*)
by (*simp add*: *max-def ncols*)

lemma *one-matrix-mult-left*[*simp*]: *nrows A* <= *n* $\Longrightarrow$ (*one-matrix n*) $*$ *A* = (*A*::('*a*::{*lordered-ring*, *ring-1*}) *matrix*)
apply (*subst Rep-matrix-inject*[*THEN sym*])
apply (*rule ext*)+
apply (*simp add*: *times-matrix-def Rep-mult-matrix*)
apply (*rule-tac j1*=*x* **in** *ssubst*[*OF foldseq-almostzero*])
apply (*simp-all*)
by (*simp add*: *max-def nrows*)

lemma *transpose-matrix-mult*: *transpose-matrix* ((*A*::('*a*::{*lordered-ring*,*comm-ring*}) *matrix*)$*$*B*) = (*transpose-matrix B*) $*$ (*transpose-matrix A*)
apply (*simp add*: *times-matrix-def*)
apply (*subst transpose-mult-matrix*)
apply (*simp-all add*: *mult-commute*)
done

lemma *transpose-matrix-add*: *transpose-matrix* ((*A*::('*a*::*lordered-ab-group*) *matrix*)+*B*) = *transpose-matrix A* + *transpose-matrix B*
by (*simp add*: *plus-matrix-def transpose-combine-matrix*)

lemma *transpose-matrix-diff*: *transpose-matrix* ((*A*::('*a*::*lordered-ab-group*) *matrix*)−*B*) = *transpose-matrix A* − *transpose-matrix B*
by (*simp add*: *diff-matrix-def transpose-combine-matrix*)

lemma *transpose-matrix-minus*: *transpose-matrix* (−(*A*::('*a*::*lordered-ring*) *matrix*)) = − *transpose-matrix* (*A*::('*a*::*lordered-ring*) *matrix*)
by (*simp add*: *minus-matrix-def transpose-apply-matrix*)

constdefs
right-inverse-matrix :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* $\Rightarrow$ '*a matrix* $\Rightarrow$ *bool*
right-inverse-matrix A X == (*A* $*$ *X* = *one-matrix* (*max* (*nrows A*) (*ncols X*))) $\wedge$ *nrows X* $\leq$ *ncols A*
left-inverse-matrix :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* $\Rightarrow$ '*a matrix* $\Rightarrow$ *bool*
left-inverse-matrix A X == (*X* $*$ *A* = *one-matrix* (*max*(*nrows X*) (*ncols A*))) $\wedge$ *ncols X* $\leq$ *nrows A*
inverse-matrix :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* $\Rightarrow$ '*a matrix* $\Rightarrow$ *bool*

inverse-matrix A X == (*right-inverse-matrix A X*) ∧ (*left-inverse-matrix A X*)

lemma *right-inverse-matrix-dim*: *right-inverse-matrix A X* ⟹ *nrows A* = *ncols X*
apply (*insert ncols-mult*[*of A X*], *insert nrows-mult*[*of A X*])
by (*simp add*: *right-inverse-matrix-def*)

lemma *left-inverse-matrix-dim*: *left-inverse-matrix A Y* ⟹ *ncols A* = *nrows Y*
apply (*insert ncols-mult*[*of Y A*], *insert nrows-mult*[*of Y A*])
by (*simp add*: *left-inverse-matrix-def*)

lemma *left-right-inverse-matrix-unique*:
  assumes *left-inverse-matrix A Y right-inverse-matrix A X*
  shows *X* = *Y*
proof −
  have *Y* = *Y* ∗ *one-matrix* (*nrows A*)
    apply (*subst one-matrix-mult-right*)
    apply (*insert prems*)
    by (*simp-all add*: *left-inverse-matrix-def*)
  also have . . . = *Y* ∗ (*A* ∗ *X*)
    apply (*insert prems*)
    apply (*frule right-inverse-matrix-dim*)
    by (*simp add*: *right-inverse-matrix-def*)
  also have . . . = (*Y* ∗ *A*) ∗ *X* **by** (*simp add*: *mult-assoc*)
  also have . . . = *X*
    apply (*insert prems*)
    apply (*frule left-inverse-matrix-dim*)
    apply (*simp-all add*: *left-inverse-matrix-def right-inverse-matrix-def one-matrix-mult-left*)
    done
  ultimately show *X* = *Y* **by** (*simp*)
qed

lemma *inverse-matrix-inject*: ⟦ *inverse-matrix A X*; *inverse-matrix A Y* ⟧ ⟹ *X* = *Y*
  by (*auto simp add*: *inverse-matrix-def left-right-inverse-matrix-unique*)

lemma *one-matrix-inverse*: *inverse-matrix* (*one-matrix n*) (*one-matrix n*)
  by (*simp add*: *inverse-matrix-def left-inverse-matrix-def right-inverse-matrix-def*)

lemma *zero-imp-mult-zero*: (*a*::′*a*::*ring*) = *0* | *b* = *0* ⟹ *a* ∗ *b* = *0*
by *auto*

lemma *Rep-matrix-zero-imp-mult-zero*:
  ! *j i k*. (*Rep-matrix A j k* = *0*) | (*Rep-matrix B k i*) = *0* ⟹ *A* ∗ *B* = (*0*::(′*a*::*lordered-ring*) *matrix*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*auto simp add*: *Rep-matrix-mult foldseq-zero zero-imp-mult-zero*)
done

lemma *add-nrows*: *nrows* (*A*::(*'a*::*comm-monoid-add*) *matrix*) <= *u* $\Longrightarrow$ *nrows B* <= *u* $\Longrightarrow$ *nrows* (*A* + *B*) <= *u*
apply (*simp add*: *plus-matrix-def*)
apply (*rule combine-nrows*)
apply (*simp-all*)
done

lemma *move-matrix-row-mult*: *move-matrix* ((*A*::(*'a*::*lordered-ring*) *matrix*) $*$ *B*) *j 0* = (*move-matrix A j 0*) $*$ *B*
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*auto simp add*: *Rep-matrix-mult foldseq-zero*)
apply (*rule-tac foldseq-zerotail*[*symmetric*])
apply (*auto simp add*: *nrows zero-imp-mult-zero max2*)
apply (*rule order-trans*)
apply (*rule ncols-move-matrix-le*)
apply (*simp add*: *max1*)
done

lemma *move-matrix-col-mult*: *move-matrix* ((*A*::(*'a*::*lordered-ring*) *matrix*) $*$ *B*) *0 i* = *A* $*$ (*move-matrix B 0 i*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*auto simp add*: *Rep-matrix-mult foldseq-zero*)
apply (*rule-tac foldseq-zerotail*[*symmetric*])
apply (*auto simp add*: *ncols zero-imp-mult-zero max1*)
apply (*rule order-trans*)
apply (*rule nrows-move-matrix-le*)
apply (*simp add*: *max2*)
done

lemma *move-matrix-add*: ((*move-matrix* (*A* + *B*) *j i*)::((*'a*::*lordered-ab-group*) *matrix*)) = (*move-matrix A j i*) + (*move-matrix B j i*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *move-matrix-mult*: *move-matrix* ((*A*::(*'a*::*lordered-ring*) *matrix*)$*$*B*) *j i* = (*move-matrix A j 0*) $*$ (*move-matrix B 0 i*)
by (*simp add*: *move-matrix-ortho*[*of A*$*$*B*] *move-matrix-col-mult move-matrix-row-mult*)

constdefs
  scalar-mult :: (*'a*::*lordered-ring*) $\Rightarrow$ *'a matrix* $\Rightarrow$ *'a matrix*
  scalar-mult a m == *apply-matrix* (*op* $*$ *a*) *m*

lemma *scalar-mult-zero*[*simp*]: *scalar-mult y 0* = *0*
  by (*simp add*: *scalar-mult-def*)

lemma *scalar-mult-add*: *scalar-mult y* $(a+b)$ = (*scalar-mult y a*) + (*scalar-mult y b*)
by (*simp add*: *scalar-mult-def apply-matrix-add ring-eq-simps*)

lemma *Rep-scalar-mult*[*simp*]: *Rep-matrix* (*scalar-mult y a*) *j i* = *y* ∗ (*Rep-matrix a j i*)
by (*simp add*: *scalar-mult-def*)

lemma *scalar-mult-singleton*[*simp*]: *scalar-mult y* (*singleton-matrix j i x*) = *singleton-matrix j i* (*y* ∗ *x*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*auto*)
done

lemma *Rep-minus*[*simp*]: *Rep-matrix* (−(*A*::-::*lordered-ab-group*)) *x y* = − (*Rep-matrix A x y*)
by (*simp add*: *minus-matrix-def*)

lemma *join-matrix*: *join* (*A*::(′*a*::*lordered-ring*) *matrix*) *B* = *combine-matrix join A B*
apply (*insert join-unique*[*of* (*combine-matrix join*)::(′*a matrix* ⇒ ′*a matrix* ⇒ ′*a matrix*), *simplified is-join-combine-matrix-join*])
apply (*simp*)
done

lemma *meet-matrix*: *meet* (*A*::(′*a*::*lordered-ring*) *matrix*) *B* = *combine-matrix meet A B*
apply (*insert meet-unique*[*of* (*combine-matrix meet*)::(′*a matrix* ⇒ ′*a matrix* ⇒ ′*a matrix*), *simplified is-meet-combine-matrix-meet*])
apply (*simp*)
done

lemma *Rep-abs*[*simp*]: *Rep-matrix* (*abs* (*A*::-::*lordered-ring*)) *x y* = *abs* (*Rep-matrix A x y*)
by (*simp add*: *abs-lattice join-matrix*)

end

theory *SparseMatrix* **imports** *Matrix* **begin**

types
′*a spvec* = (*nat* ∗ ′*a*) *list*
′*a spmat* = (′*a spvec*) *spvec*

consts

sparse-row-vector :: (*'a::lordered-ring*) *spvec* ⇒ *'a matrix*
sparse-row-matrix :: (*'a::lordered-ring*) *spmat* ⇒ *'a matrix*

defs
sparse-row-vector-def : *sparse-row-vector arr* == *foldl* (% *m x*. *m* + (*singleton-matrix 0* (*fst x*) (*snd x*))) *0 arr*
sparse-row-matrix-def : *sparse-row-matrix arr* == *foldl* (% *m r*. *m* + (*move-matrix* (*sparse-row-vector* (*snd r*)) (*int* (*fst r*)) *0*)) *0 arr*

lemma *sparse-row-vector-empty*[*simp*]: *sparse-row-vector* [] = *0*
  by (*simp add*: *sparse-row-vector-def*)

lemma *sparse-row-matrix-empty*[*simp*]: *sparse-row-matrix* [] = *0*
  by (*simp add*: *sparse-row-matrix-def*)

lemma *foldl-distrstart*[*rule-format*]: ! *a x y*. (*f* (*g x y*) *a* = *g x* (*f y a*)) ⟹ ! *x y*. (*foldl f* (*g x y*) *l* = *g x* (*foldl f y l*))
  by (*induct l*, *auto*)

lemma *sparse-row-vector-cons*[*simp*]: *sparse-row-vector* (*a#arr*) = (*singleton-matrix 0* (*fst a*) (*snd a*)) + (*sparse-row-vector arr*)
  apply (*induct arr*)
  apply (*auto simp add*: *sparse-row-vector-def*)
  apply (*simp add*: *foldl-distrstart*[*of* λ*m x*. *m* + *singleton-matrix 0* (*fst x*) (*snd x*) λ*x m*. *singleton-matrix 0* (*fst x*) (*snd x*) + *m*])
  done

lemma *sparse-row-vector-append*[*simp*]: *sparse-row-vector* (*a* @ *b*) = (*sparse-row-vector a*) + (*sparse-row-vector b*)
  by (*induct a*, *auto*)

lemma *nrows-spvec*[*simp*]: *nrows* (*sparse-row-vector x*) <= (*Suc 0*)
  apply (*induct x*)
  apply (*simp-all add*: *add-nrows*)
  done

lemma *sparse-row-matrix-cons*: *sparse-row-matrix* (*a#arr*) = ((*move-matrix* (*sparse-row-vector* (*snd a*)) (*int* (*fst a*)) *0*)) + *sparse-row-matrix arr*
  apply (*induct arr*)
  apply (*auto simp add*: *sparse-row-matrix-def*)
  apply (*simp add*: *foldl-distrstart*[*of* λ*m x*. *m* + (*move-matrix* (*sparse-row-vector* (*snd x*)) (*int* (*fst x*)) *0*)
    % *a m*. (*move-matrix* (*sparse-row-vector* (*snd a*)) (*int* (*fst a*)) *0*) + *m*])
  done

lemma *sparse-row-matrix-append*: *sparse-row-matrix* (*arr*@*brr*) = (*sparse-row-matrix arr*) + (*sparse-row-matrix brr*)
  apply (*induct arr*)
  apply (*auto simp add*: *sparse-row-matrix-cons*)

done

consts
sorted-spvec :: $'a$ *spvec* ⇒ *bool*
sorted-spmat :: $'a$ *spmat* ⇒ *bool*

primrec
sorted-spmat [] = *True*
sorted-spmat (*a#as*) = ((*sorted-spvec* (*snd a*)) & (*sorted-spmat as*))

primrec
sorted-spvec [] = *True*
sorted-spvec-step: *sorted-spvec* (*a#as*) = (*case as of* [] ⇒ *True* | *b#bs* ⇒ ((*fst a* < *fst b*) & (*sorted-spvec as*)))

declare *sorted-spvec.simps* [*simp del*]

lemma *sorted-spvec-empty*[*simp*]: *sorted-spvec* [] = *True*
by (*simp add*: *sorted-spvec.simps*)

lemma *sorted-spvec-cons1*: *sorted-spvec* (*a#as*) ⟹ *sorted-spvec as*
apply (*induct as*)
apply (*auto simp add*: *sorted-spvec.simps*)
done

lemma *sorted-spvec-cons2*: *sorted-spvec* (*a#b#t*) ⟹ *sorted-spvec* (*a#t*)
apply (*induct t*)
apply (*auto simp add*: *sorted-spvec.simps*)
done

lemma *sorted-spvec-cons3*: *sorted-spvec*(*a#b#t*) ⟹ *fst a* < *fst b*
apply (*auto simp add*: *sorted-spvec.simps*)
done

lemma *sorted-sparse-row-vector-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*) ⟶ *Rep-matrix* (*sparse-row-vector arr*) *j m* = *0*
apply (*induct arr*)
apply (*auto*)
apply (*frule sorted-spvec-cons2*,*simp*)+
apply (*frule sorted-spvec-cons3*, *simp*)
done

lemma *sorted-sparse-row-matrix-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*) ⟶ *Rep-matrix* (*sparse-row-matrix arr*) *m j* = *0*
apply (*induct arr*)
apply (*auto*)
apply (*frule sorted-spvec-cons2*, *simp*)
apply (*frule sorted-spvec-cons3*, *simp*)
apply (*simp add*: *sparse-row-matrix-cons neg-def*)

done

consts
abs-spvec :: ('a::*lordered-ring*) *spvec* ⇒ 'a *spvec*
minus-spvec :: ('a::*lordered-ring*) *spvec* ⇒ 'a *spvec*
smult-spvec :: ('a::*lordered-ring*) ⇒ 'a *spvec* ⇒ 'a *spvec*
addmult-spvec :: ('a::*lordered-ring*) ∗ 'a *spvec* ∗ 'a *spvec* ⇒ 'a *spvec*

primrec
minus-spvec [] = []
minus-spvec (*a*#*as*) = (*fst a*, −(*snd a*))#(*minus-spvec as*)

primrec
abs-spvec [] = []
abs-spvec (*a*#*as*) = (*fst a*, *abs* (*snd a*))#(*abs-spvec as*)

lemma *sparse-row-vector-minus*:
sparse-row-vector (*minus-spvec v*) = − (*sparse-row-vector v*)
apply (*induct v*)
apply (*simp-all add*: *sparse-row-vector-cons*)
apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply *simp*
done

lemma *sparse-row-vector-abs*:
sorted-spvec v ⟹ *sparse-row-vector* (*abs-spvec v*) = *abs* (*sparse-row-vector v*)
apply (*induct v*)
apply (*simp-all add*: *sparse-row-vector-cons*)
apply (*frule-tac sorted-spvec-cons1*, *simp*)
apply (*simp only*: *Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply *auto*
apply (*subgoal-tac Rep-matrix* (*sparse-row-vector v*) *0 a* = *0*)
apply (*simp*)
apply (*rule sorted-sparse-row-vector-zero*)
apply *auto*
done

lemma *sorted-spvec-minus-spvec*:
sorted-spvec v ⟹ *sorted-spvec* (*minus-spvec v*)
apply (*induct v*)
apply (*simp*)
apply (*frule sorted-spvec-cons1*, *simp*)
apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

lemma *sorted-spvec-minus-spvec*:
sorted-spvec v ⟹ *sorted-spvec* (*minus-spvec v*)

apply (*induct v*)
apply (*simp*)
apply (*frule sorted-spvec-cons1*, *simp*)
apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

lemma *sorted-spvec-abs-spvec*:
sorted-spvec v $\Longrightarrow$ *sorted-spvec* (*abs-spvec v*)
apply (*induct v*)
apply (*simp*)
apply (*frule sorted-spvec-cons1*, *simp*)
apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

defs
smult-spvec-def: *smult-spvec y arr* == *map* (% *a*. (*fst a*, *y* * *snd a*)) *arr*

lemma *smult-spvec-empty*[*simp*]: *smult-spvec y* [] = []
by (*simp add*: *smult-spvec-def*)

lemma *smult-spvec-cons*: *smult-spvec y* (*a*#*arr*) = (*fst a*, *y* * (*snd a*)) # (*smult-spvec y arr*)
by (*simp add*: *smult-spvec-def*)

recdef *addmult-spvec measure* (% (*y*, *a*, *b*). *length a* + (*length b*))
addmult-spvec (*y*, *arr*, []) = *arr*
addmult-spvec (*y*, [], *brr*) = *smult-spvec y brr*
addmult-spvec (*y*, *a*#*arr*, *b*#*brr*) = (
if (*fst a*) < (*fst b*) *then* (*a*#(*addmult-spvec* (*y*, *arr*, *b*#*brr*)))
else (*if* (*fst b* < *fst a*) *then* ((*fst b*, *y* * (*snd b*))#(*addmult-spvec* (*y*, *a*#*arr*, *brr*)))
else ((*fst a*, (*snd a*)+ *y**(*snd b*))#(*addmult-spvec* (*y*, *arr*,*brr*)))))

lemma *addmult-spvec-empty1*[*simp*]: *addmult-spvec* (*y*, [], *a*) = *smult-spvec y a*
by (*induct a*, *auto*)

lemma *addmult-spvec-empty2*[*simp*]: *addmult-spvec* (*y*, *a*, []) = *a*
by (*induct a*, *auto*)

lemma *sparse-row-vector-map*: (! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*)) $\Longrightarrow$ (*f*::'*a*⇒('*a*::*lordered-ring*)) *0* = *0* $\Longrightarrow$
sparse-row-vector (*map* (% *x*. (*fst x*, *f* (*snd x*))) *a*) = *apply-matrix f* (*sparse-row-vector a*)
apply (*induct a*)
apply (*simp-all add*: *apply-matrix-add*)
done

lemma *sparse-row-vector-smult*: *sparse-row-vector* (*smult-spvec y a*) = *scalar-mult y* (*sparse-row-vector a*)

apply (*induct a*)
apply (*simp-all add*: *smult-spvec-cons scalar-mult-add*)
done

lemma *sparse-row-vector-addmult-spvec*: *sparse-row-vector* (*addmult-spvec* (y::$'a$::*lordered-ring*, a, b)) =
(*sparse-row-vector a*) + (*scalar-mult y* (*sparse-row-vector b*))
apply (*rule addmult-spvec.induct*[*of - y*])
apply (*simp add*: *scalar-mult-add smult-spvec-cons sparse-row-vector-smult singleton-matrix-add*)+
done

lemma *sorted-smult-spvec*[*rule-format*]: *sorted-spvec a* $\Longrightarrow$ *sorted-spvec* (*smult-spvec y a*)
apply (*auto simp add*: *smult-spvec-def*)
apply (*induct a*)
apply (*auto simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

lemma *sorted-spvec-addmult-spvec-helper*: ⟦*sorted-spvec* (*addmult-spvec* (y, (a, b) # *arr*, *brr*)); $aa < a$; *sorted-spvec* ((a, b) # *arr*);
sorted-spvec ((aa, ba) # *brr*)⟧ $\Longrightarrow$ *sorted-spvec* ((aa, $y * ba$) # *addmult-spvec* (y, (a, b) # *arr*, *brr*))
apply (*induct brr*)
apply (*auto simp add*: *sorted-spvec.simps*)
apply (*simp split*: *list.split*)
apply (*auto*)
apply (*simp split*: *list.split*)
apply (*auto*)
done

lemma *sorted-spvec-addmult-spvec-helper2*:
⟦*sorted-spvec* (*addmult-spvec* (y, *arr*, (aa, ba) # *brr*)); $a < aa$; *sorted-spvec* ((a, b) # *arr*); *sorted-spvec* ((aa, ba) # *brr*)⟧
$\Longrightarrow$ *sorted-spvec* ((a, b) # *addmult-spvec* (y, *arr*, (aa, ba) # *brr*))
apply (*induct arr*)
apply (*auto simp add*: *smult-spvec-def sorted-spvec.simps*)
apply (*simp split*: *list.split*)
apply (*auto*)
done

lemma *sorted-spvec-addmult-spvec-helper3*[*rule-format*]:
sorted-spvec (*addmult-spvec* (y, *arr*, *brr*)) $\longrightarrow$ *sorted-spvec* ((aa, b) # *arr*) $\longrightarrow$ *sorted-spvec* ((aa, ba) # *brr*)
$\longrightarrow$ *sorted-spvec* ((aa, $b + y * ba$) # (*addmult-spvec* (y, *arr*, *brr*)))
apply (*rule addmult-spvec.induct*[*of - y arr brr*])
apply (*simp-all add*: *sorted-spvec.simps smult-spvec-def*)
done

lemma *sorted-addmult-spvec*[*rule-format*]: *sorted-spvec a* $\longrightarrow$ *sorted-spvec b* $\longrightarrow$

sorted-spvec (*addmult-spvec* (*y*, *a*, *b*))
  apply (*rule addmult-spvec.induct*[*of - y a b*])
  apply (*simp-all add*: *sorted-smult-spvec*)
  apply (*rule conjI*, *intro strip*)
  apply (*case-tac* ~(*a* < *aa*))
  apply (*simp-all*)
  apply (*frule-tac as*=*brr* **in** *sorted-spvec-cons1*)
  apply (*simp add*: *sorted-spvec-addmult-spvec-helper*)
  apply (*intro strip* | *rule conjI*)+
  apply (*frule-tac as*=*arr* **in** *sorted-spvec-cons1*)
  apply (*simp add*: *sorted-spvec-addmult-spvec-helper2*)
  apply (*intro strip*)
  apply (*frule-tac as*=*arr* **in** *sorted-spvec-cons1*)
  apply (*frule-tac as*=*brr* **in** *sorted-spvec-cons1*)
  apply (*simp*)
  apply (*simp-all add*: *sorted-spvec-addmult-spvec-helper3*)
  done

consts
  mult-spvec-spmat :: (*'a*::*lordered-ring*) *spvec* ∗ *'a spvec* ∗ *'a spmat* ⇒ *'a spvec*

recdef *mult-spvec-spmat measure* (% (*c*, *arr*, *brr*). (*length arr*) + (*length brr*))
  mult-spvec-spmat (*c*, [], *brr*) = *c*
  mult-spvec-spmat (*c*, *arr*, []) = *c*
  mult-spvec-spmat (*c*, *a*#*arr*, *b*#*brr*) = (
    if ((*fst a*) < (*fst b*)) *then* (*mult-spvec-spmat* (*c*, *arr*, *b*#*brr*))
    else (*if* ((*fst b*) < (*fst a*)) *then* (*mult-spvec-spmat* (*c*, *a*#*arr*, *brr*))
    else (*mult-spvec-spmat* (*addmult-spvec* (*snd a*, *c*, *snd b*), *arr*, *brr*))))

lemma *sparse-row-mult-spvec-spmat*[*rule-format*]: *sorted-spvec* (*a*::(*'a*::*lordered-ring*) *spvec*) ⟶ *sorted-spvec B* ⟶
 sparse-row-vector (*mult-spvec-spmat* (*c*, *a*, *B*)) = (*sparse-row-vector c*) + (*sparse-row-vector a*) ∗ (*sparse-row-matrix B*)
proof −
  have *comp-1*: !! *a b*. *a* < *b* ⟹ *Suc 0* <= *nat* ((*int b*)−(*int a*)) **by** *arith*
  have *not-iff*: !! *a b*. *a* = *b* ⟹ (~ *a*) = (~ *b*) **by** *simp*
  have *max-helper*: !! *a b*. ~ (*a* <= *max* (*Suc a*) *b*) ⟹ *False*
    by *arith*
  {
    fix *a*
    fix *v*
    assume *a*:*a* < *nrows*(*sparse-row-vector v*)
    have *b*:*nrows*(*sparse-row-vector v*) <= *1* **by** *simp*
    note *dummy* = *less-le-trans*[*of a nrows* (*sparse-row-vector v*) *1*, *OF a b*]
    then have *a* = *0* **by** *simp*
  }
  note *nrows-helper* = *this*
  show *?thesis*
    apply (*rule mult-spvec-spmat.induct*)

apply *simp+*
apply (*rule conjI*)
apply (*intro strip*)
apply (*frule-tac as=brr* **in** *sorted-spvec-cons1*)
apply (*simp add*: *ring-eq-simps sparse-row-matrix-cons*)
apply (*simplesubst Rep-matrix-zero-imp-mult-zero*)
apply (*simp*)
apply (*intro strip*)
apply (*rule disjI2*)
apply (*intro strip*)
apply (*subst nrows*)
apply (*rule order-trans*[*of - 1*])
apply (*simp add*: *comp-1*)+
apply (*subst Rep-matrix-zero-imp-mult-zero*)
apply (*intro strip*)
apply (*case-tac k* <= *aa*)
apply (*rule-tac m1* = *k* **and** *n1* = *a* **and** *a1* = *b* **in** *ssubst*[*OF sorted-sparse-row-vector-zero*])
apply (*simp-all*)
apply (*rule impI*)
apply (*rule disjI2*)
apply (*rule nrows*)
apply (*rule order-trans*[*of - 1*])
apply (*simp-all add*: *comp-1*)

apply (*intro strip* | *rule conjI*)+
apply (*frule-tac as=arr* **in** *sorted-spvec-cons1*)
apply (*simp add*: *ring-eq-simps*)
apply (*subst Rep-matrix-zero-imp-mult-zero*)
apply (*simp*)
apply (*rule disjI2*)
apply (*intro strip*)
apply (*simp add*: *sparse-row-matrix-cons neg-def*)
apply (*case-tac a* <= *aa*)
apply (*erule sorted-sparse-row-matrix-zero*)
apply (*simp-all*)
apply (*intro strip*)
apply (*case-tac a=aa*)
apply (*simp-all*)
apply (*frule-tac as=arr* **in** *sorted-spvec-cons1*)
apply (*frule-tac as=brr* **in** *sorted-spvec-cons1*)
apply (*simp add*: *sparse-row-matrix-cons ring-eq-simps sparse-row-vector-addmult-spvec*)
apply (*rule-tac B1* = *sparse-row-matrix brr* **in** *ssubst*[*OF Rep-matrix-zero-imp-mult-zero*])
apply (*auto*)
apply (*rule sorted-sparse-row-matrix-zero*)
apply (*simp-all*)
apply (*rule-tac A1* = *sparse-row-vector arr* **in** *ssubst*[*OF Rep-matrix-zero-imp-mult-zero*])
apply (*auto*)
apply (*rule-tac m=k* **and** *n* = *aa* **and** *a* = *b* **and** *arr=arr* **in** *sorted-sparse-row-vector-zero*)
apply (*simp-all*)

apply (*simp add*: *neg-def*)
apply (*drule nrows-notzero*)
apply (*drule nrows-helper*)
apply (*arith*)

apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
apply (*subst Rep-matrix-mult*)
apply (*rule-tac j1*=*aa* **in** *ssubst*[*OF foldseq-almostzero*])
apply (*simp-all*)
apply (*intro strip*, *rule conjI*)
apply (*intro strip*)
apply (*drule-tac max-helper*)
apply (*simp*)
apply (*auto*)
apply (*rule zero-imp-mult-zero*)
apply (*rule disjI2*)
apply (*rule nrows*)
apply (*rule order-trans*[*of - 1*])
apply (*simp*)
apply (*simp*)
done
qed

lemma *sorted-mult-spvec-spmat*[*rule-format*]:
sorted-spvec (*c*::(*'a*::*lordered-ring*) *spvec*) $\longrightarrow$ *sorted-spmat B* $\longrightarrow$ *sorted-spvec* (*mult-spvec-spmat* (*c*, *a*, *B*))
apply (*rule mult-spvec-spmat.induct*[*of - c a B*])
apply (*simp-all add*: *sorted-addmult-spvec*)
done

consts
mult-spmat :: (*'a*::*lordered-ring*) *spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat*

primrec
mult-spmat [] *A* = []
mult-spmat (*a*#*as*) *A* = (*fst a*, *mult-spvec-spmat* ([], *snd a*, *A*))#(*mult-spmat as A*)

lemma *sparse-row-mult-spmat*[*rule-format*]:
sorted-spmat A $\longrightarrow$ *sorted-spvec B* $\longrightarrow$ *sparse-row-matrix* (*mult-spmat A B*) = (*sparse-row-matrix A*) * (*sparse-row-matrix B*)
apply (*induct A*)
apply (*auto simp add*: *sparse-row-matrix-cons sparse-row-mult-spvec-spmat ring-eq-simps move-matrix-mult*)
done

lemma *sorted-spvec-mult-spmat*[*rule-format*]:

sorted-spvec (*A*::(*'a*::*lordered-ring*) *spmat*) ⟶ *sorted-spvec* (*mult-spmat A B*)
apply (*induct A*)
apply (*auto*)
apply (*drule sorted-spvec-cons1*, *simp*)
apply (*case-tac A*)
apply (*auto simp add*: *sorted-spvec.simps*)
done

lemma *sorted-spmat-mult-spmat*[*rule-format*]:
sorted-spmat (*B*::(*'a*::*lordered-ring*) *spmat*) ⟶ *sorted-spmat* (*mult-spmat A B*)
apply (*induct A*)
apply (*auto simp add*: *sorted-mult-spvec-spmat*)
done

consts
add-spvec :: (*'a*::*lordered-ab-group*) *spvec* ∗ *'a spvec* ⇒ *'a spvec*
add-spmat :: (*'a*::*lordered-ab-group*) *spmat* ∗ *'a spmat* ⇒ *'a spmat*

recdef *add-spvec measure* (% (*a*, *b*). *length a* + (*length b*))
add-spvec (*arr*, []) = *arr*
add-spvec ([], *brr*) = *brr*
add-spvec (*a*#*arr*, *b*#*brr*) = (
if (*fst a*) < (*fst b*) *then* (*a*#(*add-spvec* (*arr*, *b*#*brr*)))
else (*if* (*fst b* < *fst a*) *then* (*b*#(*add-spvec* (*a*#*arr*, *brr*)))
else ((*fst a*, (*snd a*)+(*snd b*))#(*add-spvec* (*arr*,*brr*)))))

lemma *add-spvec-empty1*[*simp*]: *add-spvec* ([], *a*) = *a*
by (*induct a*, *auto*)

lemma *add-spvec-empty2*[*simp*]: *add-spvec* (*a*, []) = *a*
by (*induct a*, *auto*)

lemma *sparse-row-vector-add*: *sparse-row-vector* (*add-spvec* (*a*,*b*)) = (*sparse-row-vector a*) + (*sparse-row-vector b*)
apply (*rule add-spvec.induct*[*of - a b*])
apply (*simp-all add*: *singleton-matrix-add*)
done

recdef *add-spmat measure* (% (*A*,*B*). (*length A*)+(*length B*))
add-spmat ([], *bs*) = *bs*
add-spmat (*as*, []) = *as*
add-spmat (*a*#*as*, *b*#*bs*) = (
if fst a < *fst b then*
(*a*#(*add-spmat* (*as*, *b*#*bs*)))
else (*if fst b* < *fst a then*
(*b*#(*add-spmat* (*a*#*as*, *bs*)))
else
((*fst a*, *add-spvec* (*snd a*, *snd b*))#(*add-spmat* (*as*, *bs*)))))

lemma *sparse-row-add-spmat*: *sparse-row-matrix* (*add-spmat* (*A*, *B*)) = (*sparse-row-matrix* *A*) + (*sparse-row-matrix* *B*)
apply (*rule add-spmat.induct*)
apply (*auto simp add*: *sparse-row-matrix-cons sparse-row-vector-add move-matrix-add*)
done

lemma *sorted-add-spvec-helper1* [*rule-format*]: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
proof −
have (! *x ab a*. *x* = (*a*,*b*)#*arr* ⟶ *add-spvec* (*x*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*ab* = *fst* (*hd brr*))))
by (*rule add-spvec.induct*[*of* - - *brr*], *auto*)
then show *?thesis*
by (*case-tac brr*, *auto*)
qed

lemma *sorted-add-spmat-helper1* [*rule-format*]: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
proof −
have (! *x ab a*. *x* = (*a*,*b*)#*arr* ⟶ *add-spmat* (*x*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*ab* = *fst* (*hd brr*))))
by (*rule add-spmat.induct*[*of* - - *brr*], *auto*)
then show *?thesis*
by (*case-tac brr*, *auto*)
qed

lemma *sorted-add-spvec-helper*[*rule-format*]: *add-spvec* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
apply (*rule add-spvec.induct*[*of* - *arr brr*])
apply (*auto*)
done

lemma *sorted-add-spmat-helper*[*rule-format*]: *add-spmat* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
apply (*rule add-spmat.induct*[*of* - *arr brr*])
apply (*auto*)
done

lemma *add-spvec-commute*: *add-spvec* (*a*, *b*) = *add-spvec* (*b*, *a*)
by (*rule add-spvec.induct*[*of* - *a b*], *auto*)

lemma *add-spmat-commute*: *add-spmat* (*a*, *b*) = *add-spmat* (*b*, *a*)
apply (*rule add-spmat.induct*[*of* - *a b*])
apply (*simp-all add*: *add-spvec-commute*)
done

lemma *sorted-add-spvec-helper2*: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟹ *aa* < *a* ⟹ *sorted-spvec* ((*aa*, *ba*) # *brr*) ⟹ *aa* < *ab*
apply (*drule sorted-add-spvec-helper1*)

apply (*auto*)
apply (*case-tac brr*)
apply (*simp-all*)
apply (*drule-tac sorted-spvec-cons3*)
apply (*simp*)
done

lemma *sorted-add-spmat-helper2*: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* $\Longrightarrow$ *aa* < *a* $\Longrightarrow$ *sorted-spvec* ((*aa*, *ba*) # *brr*) $\Longrightarrow$ *aa* < *ab*
apply (*drule sorted-add-spmat-helper1*)
apply (*auto*)
apply (*case-tac brr*)
apply (*simp-all*)
apply (*drule-tac sorted-spvec-cons3*)
apply (*simp*)
done

lemma *sorted-spvec-add-spvec*[*rule-format*]: *sorted-spvec a* $\longrightarrow$ *sorted-spvec b* $\longrightarrow$ *sorted-spvec* (*add-spvec* (*a*, *b*))
apply (*rule add-spvec.induct*[*of - a b*])
apply (*simp-all*)
apply (*rule conjI*)
apply (*intro strip*)
apply (*simp*)
apply (*frule-tac as*=*brr* **in** *sorted-spvec-cons1*)
apply (*simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)
apply (*simp add*: *sorted-add-spvec-helper2*)
apply (*clarify*)
apply (*rule conjI*)
apply (*case-tac a*=*aa*)
apply (*simp*)
apply (*clarify*)
apply (*frule-tac as*=*arr* **in** *sorted-spvec-cons1*, *simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)
apply (*simp add*: *sorted-add-spvec-helper2 add-spvec-commute*)
apply (*case-tac a*=*aa*)
apply (*simp-all*)
apply (*clarify*)
apply (*frule-tac as*=*arr* **in** *sorted-spvec-cons1*)
apply (*frule-tac as*=*brr* **in** *sorted-spvec-cons1*)
apply (*simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)

apply (*drule-tac sorted-add-spvec-helper*)
apply (*auto*)
apply (*case-tac arr*)
apply (*simp-all*)
apply (*drule sorted-spvec-cons3*)
apply (*simp*)
apply (*case-tac brr*)
apply (*simp-all*)
apply (*drule sorted-spvec-cons3*)
apply (*simp*)
done

lemma *sorted-spvec-add-spmat*[*rule-format*]: *sorted-spvec A* $\longrightarrow$ *sorted-spvec B* $\longrightarrow$ *sorted-spvec* (*add-spmat* (*A*, *B*))
apply (*rule add-spmat.induct*[*of - A B*])
apply (*simp-all*)
apply (*rule conjI*)
apply (*intro strip*)
apply (*simp*)
apply (*frule-tac as=bs* **in** *sorted-spvec-cons1*)
apply (*simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)
apply (*simp add*: *sorted-add-spmat-helper2*)
apply (*clarify*)
apply (*rule conjI*)
apply (*case-tac a=aa*)
apply (*simp*)
apply (*clarify*)
apply (*frule-tac as=as* **in** *sorted-spvec-cons1*, *simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)
apply (*simp add*: *sorted-add-spmat-helper2 add-spmat-commute*)
apply (*case-tac a=aa*)
apply (*simp-all*)
apply (*clarify*)
apply (*frule-tac as=as* **in** *sorted-spvec-cons1*)
apply (*frule-tac as=bs* **in** *sorted-spvec-cons1*)
apply (*simp*)
apply (*subst sorted-spvec-step*)
apply (*simp split*: *list.split*)
apply (*clarify*, *simp*)
apply (*drule-tac sorted-add-spmat-helper*)
apply (*auto*)
apply (*case-tac as*)
apply (*simp-all*)
apply (*drule sorted-spvec-cons3*)

apply (*simp*)
apply (*case-tac bs*)
apply (*simp-all*)
apply (*drule sorted-spvec-cons3*)
apply (*simp*)
done

lemma *sorted-spmat-add-spmat*[*rule-format*]: *sorted-spmat A* ⟶ *sorted-spmat B* ⟶ *sorted-spmat* (*add-spmat* (*A*, *B*))
apply (*rule add-spmat.induct*[*of - A B*])
apply (*simp-all add*: *sorted-spvec-add-spvec*)
done

consts
le-spvec :: (*'a*::*lordered-ab-group*) *spvec* ∗ *'a spvec* ⇒ *bool*
le-spmat :: (*'a*::*lordered-ab-group*) *spmat* ∗ *'a spmat* ⇒ *bool*

recdef *le-spvec measure* (% (*a*,*b*). (*length a*) + (*length b*))
le-spvec ([], []) = *True*
le-spvec (*a*#*as*, []) = ((*snd a* <= *0*) & (*le-spvec* (*as*, [])))
le-spvec ([], *b*#*bs*) = ((*0* <= *snd b*) & (*le-spvec* ([], *bs*)))
le-spvec (*a*#*as*, *b*#*bs*) = (
if (*fst a* < *fst b*) *then*
((*snd a* <= *0*) & (*le-spvec* (*as*, *b*#*bs*)))
else (*if* (*fst b* < *fst a*) *then*
((*0* <= *snd b*) & (*le-spvec* (*a*#*as*, *bs*)))
else
((*snd a* <= *snd b*) & (*le-spvec* (*as*, *bs*)))))

recdef *le-spmat measure* (% (*a*,*b*). (*length a*) + (*length b*))
le-spmat ([], []) = *True*
le-spmat (*a*#*as*, []) = (*le-spvec* (*snd a*, []) & (*le-spmat* (*as*, [])))
le-spmat ([], *b*#*bs*) = (*le-spvec* ([], *snd b*) & (*le-spmat* ([], *bs*)))
le-spmat (*a*#*as*, *b*#*bs*) = (
if fst a < *fst b then*
(*le-spvec*(*snd a*,[]) & *le-spmat*(*as*, *b*#*bs*))
else (*if* (*fst b* < *fst a*) *then*
(*le-spvec*([], *snd b*) & *le-spmat*(*a*#*as*, *bs*))
else
(*le-spvec*(*snd a*, *snd b*) & *le-spmat* (*as*, *bs*))))

constdefs
disj-matrices :: (*'a*::*zero*) *matrix* ⇒ *'a matrix* ⇒ *bool*
disj-matrices A B == (! *j i*. (*Rep-matrix A j i* ≠ *0*) ⟶ (*Rep-matrix B j i* = *0*)) & (! *j i*. (*Rep-matrix B j i* ≠ *0*) ⟶ (*Rep-matrix A j i* = *0*))

ML ⟪ *simp-depth-limit* := *6* ⟫

lemma *disj-matrices-contr1*: *disj-matrices A B* ⟹ *Rep-matrix A j i* ≠ *0* ⟹

Rep-matrix B j i = *0*
  by (*simp add*: *disj-matrices-def*)

lemma *disj-matrices-contr2*: *disj-matrices A B* $\Longrightarrow$ *Rep-matrix B j i* $\neq$ *0* $\Longrightarrow$ *Rep-matrix A j i* = *0*
  by (*simp add*: *disj-matrices-def*)

lemma *disj-matrices-add*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices C D* $\Longrightarrow$ *disj-matrices A D* $\Longrightarrow$ *disj-matrices B C* $\Longrightarrow$
 (*A* + *B* <= *C* + *D*) = (*A* <= *C* & *B* <= (*D*::('*a*::*lordered-ab-group*) *matrix*))
 apply (*auto*)
 apply (*simp* (*no-asm-use*) *only*: *le-matrix-def disj-matrices-def*)
 apply (*intro strip*)
 apply (*erule conjE*)+
 apply (*drule-tac j*=*j* **and** *i*=*i* **in** *spec2*)+
 apply (*case-tac Rep-matrix B j i* = *0*)
 apply (*case-tac Rep-matrix D j i* = *0*)
 apply (*simp-all*)
 apply (*simp* (*no-asm-use*) *only*: *le-matrix-def disj-matrices-def*)
 apply (*intro strip*)
 apply (*erule conjE*)+
 apply (*drule-tac j*=*j* **and** *i*=*i* **in** *spec2*)+
 apply (*case-tac Rep-matrix A j i* = *0*)
 apply (*case-tac Rep-matrix C j i* = *0*)
 apply (*simp-all*)
 apply (*erule add-mono*)
 apply (*assumption*)
 done

lemma *disj-matrices-zero1*[*simp*]: *disj-matrices 0 B*
by (*simp add*: *disj-matrices-def*)

lemma *disj-matrices-zero2*[*simp*]: *disj-matrices A 0*
by (*simp add*: *disj-matrices-def*)

lemma *disj-matrices-commute*: *disj-matrices A B* = *disj-matrices B A*
by (*auto simp add*: *disj-matrices-def*)

lemma *disj-matrices-add-le-zero*: *disj-matrices A B* $\Longrightarrow$
 (*A* + *B* <= *0*) = (*A* <= *0* & (*B*::('*a*::*lordered-ab-group*) *matrix*) <= *0*)
by (*rule disj-matrices-add*[*of A B 0 0*, *simplified*])

lemma *disj-matrices-add-zero-le*: *disj-matrices A B* $\Longrightarrow$
 (*0* <= *A* + *B*) = (*0* <= *A* & *0* <= (*B*::('*a*::*lordered-ab-group*) *matrix*))
by (*rule disj-matrices-add*[*of 0 0 A B*, *simplified*])

lemma *disj-matrices-add-x-le*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices B C* $\Longrightarrow$
 (*A* <= *B* + *C*) = (*A* <= *C* & *0* <= (*B*::('*a*::*lordered-ab-group*) *matrix*))

by (*auto simp add*: *disj-matrices-add*[*of 0 A B C*, *simplified*])

lemma *disj-matrices-add-le-x*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices B C* $\Longrightarrow$
(B + A <= C) = (A <= C & (B::('*a*::*lordered-ab-group*) *matrix*) <= *0*)
by (*auto simp add*: *disj-matrices-add*[*of B A 0 C*,*simplified*] *disj-matrices-commute*)

lemma *disj-sparse-row-singleton*: i <= j $\Longrightarrow$ *sorted-spvec*((j,y)#v) $\Longrightarrow$ *disj-matrices* (*sparse-row-vector v*) (*singleton-matrix 0 i x*)
apply (*simp add*: *disj-matrices-def*)
apply (*rule conjI*)
apply (*rule neg-imp*)
apply (*simp*)
apply (*intro strip*)
apply (*rule sorted-sparse-row-vector-zero*)
apply (*simp-all*)
apply (*intro strip*)
apply (*rule sorted-sparse-row-vector-zero*)
apply (*simp-all*)
done

lemma *disj-matrices-x-add*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices* (A::('*a*::*lordered-ab-group*) *matrix*) (B+C)
apply (*simp add*: *disj-matrices-def*)
apply (*auto*)
apply (*drule-tac j*=*j* **and** *i*=*i* **in** *spec2*)+
apply (*case-tac Rep-matrix B j i* = *0*)
apply (*case-tac Rep-matrix C j i* = *0*)
apply (*simp-all*)
done

lemma *disj-matrices-add-x*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices* (B+C) (A::('*a*::*lordered-ab-group*) *matrix*)
by (*simp add*: *disj-matrices-x-add disj-matrices-commute*)

lemma *disj-singleton-matrices*[*simp*]: *disj-matrices* (*singleton-matrix j i x*) (*singleton-matrix u v y*) = ($j \neq u$ | $i \neq v$ | x = *0* | y = *0*)
by (*auto simp add*: *disj-matrices-def*)

lemma *disj-move-sparse-vec-mat*[*simplified disj-matrices-commute*]:
j <= a $\Longrightarrow$ *sorted-spvec*((a,c)#*as*) $\Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector b*) (*int j*) *i*) (*sparse-row-matrix as*)
apply (*auto simp add*: *neg-def disj-matrices-def*)
apply (*drule nrows-notzero*)
apply (*drule less-le-trans*[*OF - nrows-spvec*])
apply (*subgoal-tac ja* = *j*)
apply (*simp add*: *sorted-sparse-row-matrix-zero*)
apply (*arith*)
apply (*rule nrows*)
apply (*rule order-trans*[*of - 1 -*])

apply (*simp*)
apply (*case-tac nat* (*int ja* − *int j*) = *0*)
apply (*case-tac ja* = *j*)
apply (*simp add*: *sorted-sparse-row-matrix-zero*)
apply *arith*+
done

lemma *disj-move-sparse-row-vector-twice*:
$j \neq u \Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector a*) *j i*) (*move-matrix* (*sparse-row-vector b*) *u v*)
apply (*auto simp add*: *neg-def disj-matrices-def*)
apply (*rule nrows*, *rule order-trans*[*of - 1*], *simp*, *drule nrows-notzero*, *drule less-le-trans*[*OF - nrows-spvec*], *arith*)+
done

lemma *le-spvec-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec a*) ⟶ (*sorted-spvec b*) ⟶ (*le-spvec* (*a*,*b*)) = (*sparse-row-vector a* <= *sparse-row-vector b*)
apply (*rule le-spvec.induct*)
apply (*simp-all add*: *sorted-spvec-cons1 disj-matrices-add-le-zero disj-matrices-add-zero-le*

disj-sparse-row-singleton[*OF order-refl*] *disj-matrices-commute*)
apply (*rule conjI*, *intro strip*)
apply (*simp add*: *sorted-spvec-cons1*)
apply (*subst disj-matrices-add-x-le*)
apply (*simp add*: *disj-sparse-row-singleton*[*OF less-imp-le*] *disj-matrices-x-add disj-matrices-commute*)
apply (*simp add*: *disj-sparse-row-singleton*[*OF order-refl*] *disj-matrices-commute*)
apply (*simp*, *blast*)
apply (*intro strip*, *rule conjI*, *intro strip*)
apply (*simp add*: *sorted-spvec-cons1*)
apply (*subst disj-matrices-add-le-x*)
apply (*simp-all add*: *disj-sparse-row-singleton*[*OF order-refl*] *disj-sparse-row-singleton*[*OF less-imp-le*] *disj-matrices-commute disj-matrices-x-add*)
apply (*blast*)
apply (*intro strip*)
apply (*simp add*: *sorted-spvec-cons1*)
apply (*case-tac a*=*aa*, *simp-all*)
apply (*subst disj-matrices-add*)
apply (*simp-all add*: *disj-sparse-row-singleton*[*OF order-refl*] *disj-matrices-commute*)
done

lemma *le-spvec-empty2-sparse-row*[*rule-format*]: (*sorted-spvec b*) ⟶ (*le-spvec* (*b*,[]) = (*sparse-row-vector b* <= *0*))
apply (*induct b*)
apply (*simp-all add*: *sorted-spvec-cons1*)
apply (*intro strip*)
apply (*subst disj-matrices-add-le-zero*)
apply (*simp add*: *disj-matrices-commute disj-sparse-row-singleton sorted-spvec-cons1*)
apply (*rule-tac y* = *snd a* **in** *disj-sparse-row-singleton*[*OF order-refl*])

apply (*simp-all*)
done

lemma *le-spvec-empty1-sparse-row*[*rule-format*]: (*sorted-spvec b*) ⟶ (*le-spvec* ([],*b*) = (*0* <= *sparse-row-vector b*))
apply (*induct b*)
apply (*simp-all add*: *sorted-spvec-cons1*)
apply (*intro strip*)
apply (*subst disj-matrices-add-zero-le*)
apply (*simp add*: *disj-matrices-commute disj-sparse-row-singleton sorted-spvec-cons1*)
apply (*rule-tac y* = *snd a* **in** *disj-sparse-row-singleton*[*OF order-refl*])
apply (*simp-all*)
done

lemma *le-spmat-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec A*) ⟶ (*sorted-spmat A*) ⟶ (*sorted-spvec B*) ⟶ (*sorted-spmat B*) ⟶
le-spmat(*A*, *B*) = (*sparse-row-matrix A* <= *sparse-row-matrix B*)
apply (*rule le-spmat.induct*)
apply (*simp add*: *sparse-row-matrix-cons disj-matrices-add-le-zero disj-matrices-add-zero-le disj-move-sparse-vec-mat*[*OF order-refl*]
disj-matrices-commute sorted-spvec-cons1 le-spvec-empty2-sparse-row le-spvec-empty1-sparse-row)+

apply (*rule conjI*, *intro strip*)
apply (*simp add*: *sorted-spvec-cons1*)
apply (*subst disj-matrices-add-x-le*)
apply (*rule disj-matrices-add-x*)
apply (*simp add*: *disj-move-sparse-row-vector-twice*)
apply (*simp add*: *disj-move-sparse-vec-mat*[*OF less-imp-le*] *disj-matrices-commute*)
apply (*simp add*: *disj-move-sparse-vec-mat*[*OF order-refl*] *disj-matrices-commute*)
apply (*simp*, *blast*)
apply (*intro strip*, *rule conjI*, *intro strip*)
apply (*simp add*: *sorted-spvec-cons1*)
apply (*subst disj-matrices-add-le-x*)
apply (*simp add*: *disj-move-sparse-vec-mat*[*OF order-refl*])
apply (*rule disj-matrices-x-add*)
apply (*simp add*: *disj-move-sparse-row-vector-twice*)
apply (*simp add*: *disj-move-sparse-vec-mat*[*OF less-imp-le*] *disj-matrices-commute*)
apply (*simp*, *blast*)
apply (*intro strip*)
apply (*case-tac a*=*aa*)
apply (*simp-all*)
apply (*subst disj-matrices-add*)
apply (*simp-all add*: *disj-matrices-commute disj-move-sparse-vec-mat*[*OF order-refl*])
apply (*simp add*: *sorted-spvec-cons1 le-spvec-iff-sparse-row-le*)
done

ML ⟪ *simp-depth-limit* := *999* ⟫

consts

abs-spmat :: (*'a*::*lordered-ring*) *spmat* ⇒ *'a spmat*
minus-spmat :: (*'a*::*lordered-ring*) *spmat* ⇒ *'a spmat*

primrec
abs-spmat [] = []
abs-spmat (*a*#*as*) = (*fst a*, *abs-spvec* (*snd a*))#(*abs-spmat as*)

primrec
minus-spmat [] = []
minus-spmat (*a*#*as*) = (*fst a*, *minus-spvec* (*snd a*))#(*minus-spmat as*)

lemma *sparse-row-matrix-minus*:
sparse-row-matrix (*minus-spmat A*) = − (*sparse-row-matrix A*)
apply (*induct A*)
apply (*simp-all add*: *sparse-row-vector-minus sparse-row-matrix-cons*)
apply (*subst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply *simp*
done

lemma *Rep-sparse-row-vector-zero*: $x \neq 0 \Longrightarrow$ *Rep-matrix* (*sparse-row-vector v*) *x y* = *0*
proof −
assume *x*:$x \neq 0$
have *r*:*nrows* (*sparse-row-vector v*) <= *Suc 0* **by** (*rule nrows-spvec*)
show *?thesis*
apply (*rule nrows*)
apply (*subgoal-tac Suc 0* <= *x*)
apply (*insert r*)
apply (*simp only*:)
apply (*insert x*)
apply *arith*
done
qed

lemma *sparse-row-matrix-abs*:
sorted-spvec A ⟹ *sorted-spmat A* ⟹ *sparse-row-matrix* (*abs-spmat A*) = *abs* (*sparse-row-matrix A*)
apply (*induct A*)
apply (*simp-all add*: *sparse-row-vector-abs sparse-row-matrix-cons*)
apply (*frule-tac sorted-spvec-cons1*, *simp*)
apply (*simplesubst Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply *auto*
apply (*case-tac x*=*a*)
apply (*simp*)
apply (*simplesubst sorted-sparse-row-matrix-zero*)
apply *auto*
apply (*simplesubst Rep-sparse-row-vector-zero*)

apply (*simp-all add*: *neg-def*)
done

lemma *sorted-spvec-minus-spmat*: *sorted-spvec* $A \Longrightarrow$ *sorted-spvec* (*minus-spmat* A)
apply (*induct* A)
apply (*simp*)
apply (*frule sorted-spvec-cons1*, *simp*)
apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

lemma *sorted-spvec-abs-spmat*: *sorted-spvec* $A \Longrightarrow$ *sorted-spvec* (*abs-spmat* A)
apply (*induct* A)
apply (*simp*)
apply (*frule sorted-spvec-cons1*, *simp*)
apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
done

lemma *sorted-spmat-minus-spmat*: *sorted-spmat* $A \Longrightarrow$ *sorted-spmat* (*minus-spmat* A)
apply (*induct* A)
apply (*simp-all add*: *sorted-spvec-minus-spvec*)
done

lemma *sorted-spmat-abs-spmat*: *sorted-spmat* $A \Longrightarrow$ *sorted-spmat* (*abs-spmat* A)
apply (*induct* A)
apply (*simp-all add*: *sorted-spvec-abs-spvec*)
done

constdefs
diff-spmat :: (*'a*::*lordered-ring*) *spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat*
diff-spmat A B == *add-spmat* (A, *minus-spmat* B)

lemma *sorted-spmat-diff-spmat*: *sorted-spmat* $A \Longrightarrow$ *sorted-spmat* $B \Longrightarrow$ *sorted-spmat* (*diff-spmat* A B)
by (*simp add*: *diff-spmat-def sorted-spmat-minus-spmat sorted-spmat-add-spmat*)

lemma *sorted-spvec-diff-spmat*: *sorted-spvec* $A \Longrightarrow$ *sorted-spvec* $B \Longrightarrow$ *sorted-spvec* (*diff-spmat* A B)
by (*simp add*: *diff-spmat-def sorted-spvec-minus-spmat sorted-spvec-add-spmat*)

lemma *sparse-row-diff-spmat*: *sparse-row-matrix* (*diff-spmat* A B) = (*sparse-row-matrix* A) $-$ (*sparse-row-matrix* B)
by (*simp add*: *diff-spmat-def sparse-row-add-spmat sparse-row-matrix-minus*)

constdefs
sorted-sparse-matrix :: *'a spmat* $\Rightarrow$ *bool*
sorted-sparse-matrix A == (*sorted-spvec* A) & (*sorted-spmat* A)

lemma *sorted-sparse-matrix-imp-spvec*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spvec A*
  by (*simp add*: *sorted-sparse-matrix-def*)

lemma *sorted-sparse-matrix-imp-spmat*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spmat A*
  by (*simp add*: *sorted-sparse-matrix-def*)

lemmas *sorted-sp-simps* =
  sorted-spvec.simps
  sorted-spmat.simps
  sorted-sparse-matrix-def

lemma *bool1*: ($\neg$ *True*) = *False* **by** *blast*
lemma *bool2*: ($\neg$ *False*) = *True* **by** *blast*
lemma *bool3*: ((*P*::*bool*) $\wedge$ *True*) = *P* **by** *blast*
lemma *bool4*: (*True* $\wedge$ (*P*::*bool*)) = *P* **by** *blast*
lemma *bool5*: ((*P*::*bool*) $\wedge$ *False*) = *False* **by** *blast*
lemma *bool6*: (*False* $\wedge$ (*P*::*bool*)) = *False* **by** *blast*
lemma *bool7*: ((*P*::*bool*) $\vee$ *True*) = *True* **by** *blast*
lemma *bool8*: (*True* $\vee$ (*P*::*bool*)) = *True* **by** *blast*
lemma *bool9*: ((*P*::*bool*) $\vee$ *False*) = *P* **by** *blast*
lemma *bool10*: (*False* $\vee$ (*P*::*bool*)) = *P* **by** *blast*
lemmas *boolarith* = *bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10*

lemma *if-case-eq*: (*if b then x else y*) = (*case b of True* => *x* | *False* => *y*) **by** *simp*

consts
  pprt-spvec :: ('*a*::{*lordered-ab-group*}) *spvec* $\Rightarrow$ '*a spvec*
  nprt-spvec :: ('*a*::{*lordered-ab-group*}) *spvec* $\Rightarrow$ '*a spvec*
  pprt-spmat :: ('*a*::{*lordered-ab-group*}) *spmat* $\Rightarrow$ '*a spmat*
  nprt-spmat :: ('*a*::{*lordered-ab-group*}) *spmat* $\Rightarrow$ '*a spmat*

primrec
  pprt-spvec [] = []
  pprt-spvec (*a*#*as*) = (*fst a*, *pprt* (*snd a*)) # (*pprt-spvec as*)

primrec
  nprt-spvec [] = []
  nprt-spvec (*a*#*as*) = (*fst a*, *nprt* (*snd a*)) # (*nprt-spvec as*)

primrec
  pprt-spmat [] = []
  pprt-spmat (*a*#*as*) = (*fst a*, *pprt-spvec* (*snd a*))#(*pprt-spmat as*)

primrec
  nprt-spmat [] = []
  nprt-spmat (*a*#*as*) = (*fst a*, *nprt-spvec* (*snd a*))#(*nprt-spmat as*)

lemma *pprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) $\Longrightarrow$ *pprt* (*A*+*B*) = *pprt A* + *pprt B*
  apply (*simp add*: *pprt-def join-matrix*)
  apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
  apply (*rule ext*)+
  apply *simp*
  apply (*case-tac Rep-matrix A x xa* $\neq$ *0*)
  apply (*simp-all add*: *disj-matrices-contr1*)
  done

lemma *nprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) $\Longrightarrow$ *nprt* (*A*+*B*) = *nprt A* + *nprt B*
  apply (*simp add*: *nprt-def meet-matrix*)
  apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
  apply (*rule ext*)+
  apply *simp*
  apply (*case-tac Rep-matrix A x xa* $\neq$ *0*)
  apply (*simp-all add*: *disj-matrices-contr1*)
  done

lemma *pprt-singleton*[*simp*]: *pprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*pprt x*)
  apply (*simp add*: *pprt-def join-matrix*)
  apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
  apply (*rule ext*)+
  apply *simp*
  done

lemma *nprt-singleton*[*simp*]: *nprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*nprt x*)
  apply (*simp add*: *nprt-def meet-matrix*)
  apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
  apply (*rule ext*)+
  apply *simp*
  done

lemma *less-imp-le*: *a* < *b* $\Longrightarrow$ *a* <= (*b*::-::*order*) **by** (*simp add*: *less-def*)

lemma *sparse-row-vector-pprt*: *sorted-spvec v* $\Longrightarrow$ *sparse-row-vector* (*pprt-spvec v*) = *pprt* (*sparse-row-vector v*)
  apply (*induct v*)
  apply (*simp-all*)
  apply (*frule sorted-spvec-cons1*, *auto*)
  apply (*subst pprt-add*)
  apply (*subst disj-matrices-commute*)
  apply (*rule disj-sparse-row-singleton*)

apply *auto*
done

lemma *sparse-row-vector-nprt*: *sorted-spvec v* $\Longrightarrow$ *sparse-row-vector* (*nprt-spvec v*) = *nprt* (*sparse-row-vector v*)
apply (*induct v*)
apply (*simp-all*)
apply (*frule sorted-spvec-cons1*, *auto*)
apply (*subst nprt-add*)
apply (*subst disj-matrices-commute*)
apply (*rule disj-sparse-row-singleton*)
apply *auto*
done

lemma *pprt-move-matrix*: *pprt* (*move-matrix* (*A*::(*'a*::*lordered-ring*) *matrix*) *j i*) = *move-matrix* (*pprt A*) *j i*
apply (*simp add*: *pprt-def*)
apply (*simp add*: *join-matrix*)
apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *nprt-move-matrix*: *nprt* (*move-matrix* (*A*::(*'a*::*lordered-ring*) *matrix*) *j i*) = *move-matrix* (*nprt A*) *j i*
apply (*simp add*: *nprt-def*)
apply (*simp add*: *meet-matrix*)
apply (*simp add*: *Rep-matrix-inject*[*symmetric*])
apply (*rule ext*)+
apply (*simp*)
done

lemma *sparse-row-matrix-pprt*: *sorted-spvec m* $\Longrightarrow$ *sorted-spmat m* $\Longrightarrow$ *sparse-row-matrix* (*pprt-spmat m*) = *pprt* (*sparse-row-matrix m*)
apply (*induct m*)
apply *simp*
apply *simp*
apply (*frule sorted-spvec-cons1*)
apply (*simp add*: *sparse-row-matrix-cons sparse-row-vector-pprt*)
apply (*subst pprt-add*)
apply (*subst disj-matrices-commute*)
apply (*rule disj-move-sparse-vec-mat*)
apply *auto*
apply (*simp add*: *sorted-spvec.simps*)
apply (*simp split*: *list.split*)
apply *auto*
apply (*simp add*: *pprt-move-matrix*)
done

lemma *sparse-row-matrix-nprt*: *sorted-spvec m* $\Longrightarrow$ *sorted-spmat m* $\Longrightarrow$ *sparse-row-matrix* (*nprt-spmat m*) = *nprt* (*sparse-row-matrix m*)
  apply (*induct m*)
  apply *simp*
  apply *simp*
  apply (*frule sorted-spvec-cons1*)
  apply (*simp add*: *sparse-row-matrix-cons sparse-row-vector-nprt*)
  apply (*subst nprt-add*)
  apply (*subst disj-matrices-commute*)
  apply (*rule disj-move-sparse-vec-mat*)
  apply *auto*
  apply (*simp add*: *sorted-spvec.simps*)
  apply (*simp split*: *list.split*)
  apply *auto*
  apply (*simp add*: *nprt-move-matrix*)
  done

lemma *sorted-pprt-spvec*: *sorted-spvec v* $\Longrightarrow$ *sorted-spvec* (*pprt-spvec v*)
  apply (*induct v*)
  apply (*simp*)
  apply (*frule sorted-spvec-cons1*)
  apply *simp*
  apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
  done

lemma *sorted-nprt-spvec*: *sorted-spvec v* $\Longrightarrow$ *sorted-spvec* (*nprt-spvec v*)
  apply (*induct v*)
  apply (*simp*)
  apply (*frule sorted-spvec-cons1*)
  apply *simp*
  apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
  done

lemma *sorted-spvec-pprt-spmat*: *sorted-spvec m* $\Longrightarrow$ *sorted-spvec* (*pprt-spmat m*)
  apply (*induct m*)
  apply (*simp*)
  apply (*frule sorted-spvec-cons1*)
  apply *simp*
  apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
  done

lemma *sorted-spvec-nprt-spmat*: *sorted-spvec m* $\Longrightarrow$ *sorted-spvec* (*nprt-spmat m*)
  apply (*induct m*)
  apply (*simp*)
  apply (*frule sorted-spvec-cons1*)
  apply *simp*
  apply (*simp add*: *sorted-spvec.simps split*:*list.split-asm*)
  done

lemma *sorted-spmat-pprt-spmat*: *sorted-spmat m* $\Longrightarrow$ *sorted-spmat* (*pprt-spmat m*)
apply (*induct m*)
apply (*simp-all add*: *sorted-pprt-spvec*)
done

lemma *sorted-spmat-nprt-spmat*: *sorted-spmat m* $\Longrightarrow$ *sorted-spmat* (*nprt-spmat m*)
apply (*induct m*)
apply (*simp-all add*: *sorted-nprt-spvec*)
done

constdefs
mult-est-spmat :: (*'a*::*lordered-ring*) *spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat*
mult-est-spmat r1 r2 s1 s2 ==
add-spmat (*mult-spmat* (*pprt-spmat s2*) (*pprt-spmat r2*), *add-spmat* (*mult-spmat* (*pprt-spmat s1*) (*nprt-spmat r2*),
add-spmat (*mult-spmat* (*nprt-spmat s2*) (*pprt-spmat r1*), *mult-spmat* (*nprt-spmat s1*) (*nprt-spmat r1*))))

lemmas *sparse-row-matrix-op-simps* =
sorted-sparse-matrix-imp-spmat sorted-sparse-matrix-imp-spvec
sparse-row-add-spmat sorted-spvec-add-spmat sorted-spmat-add-spmat
sparse-row-diff-spmat sorted-spvec-diff-spmat sorted-spmat-diff-spmat
sparse-row-matrix-minus sorted-spvec-minus-spmat sorted-spmat-minus-spmat
sparse-row-mult-spmat sorted-spvec-mult-spmat sorted-spmat-mult-spmat
sparse-row-matrix-abs sorted-spvec-abs-spmat sorted-spmat-abs-spmat
le-spmat-iff-sparse-row-le
sparse-row-matrix-pprt sorted-spvec-pprt-spmat sorted-spmat-pprt-spmat
sparse-row-matrix-nprt sorted-spvec-nprt-spmat sorted-spmat-nprt-spmat

lemma *zero-eq-Numeral0*: (*0*::-::*number-ring*) = *Numeral0* **by** *simp*

lemmas *sparse-row-matrix-arith-simps*[*simplified zero-eq-Numeral0*] =
mult-spmat.simps mult-spvec-spmat.simps
addmult-spvec.simps
smult-spvec-empty smult-spvec-cons
add-spmat.simps add-spvec.simps
minus-spmat.simps minus-spvec.simps
abs-spmat.simps abs-spvec.simps
diff-spmat-def
le-spmat.simps le-spvec.simps
pprt-spmat.simps pprt-spvec.simps
nprt-spmat.simps nprt-spvec.simps
mult-est-spmat-def

lemma *spm-mult-le-dual-prts*:
assumes
sorted-sparse-matrix A1
sorted-sparse-matrix A2
sorted-sparse-matrix c1
sorted-sparse-matrix c2
sorted-sparse-matrix y
sorted-sparse-matrix r1
sorted-sparse-matrix r2
sorted-spvec b
le-spmat ([], *y*)
sparse-row-matrix A1 ≤ *A*
A ≤ *sparse-row-matrix A2*
sparse-row-matrix c1 ≤ *c*
c ≤ *sparse-row-matrix c2*
sparse-row-matrix r1 ≤ *x*
x ≤ *sparse-row-matrix r2*
A ∗ *x* ≤ *sparse-row-matrix* (*b*::(′*a*::*lordered-ring*) *spmat*)
shows
c ∗ *x* ≤ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
(*let s1* = *diff-spmat c1* (*mult-spmat y A2*); *s2* = *diff-spmat c2* (*mult-spmat y A1*) *in*
add-spmat (*mult-spmat* (*pprt-spmat s2*) (*pprt-spmat r2*), *add-spmat* (*mult-spmat* (*pprt-spmat s1*) (*nprt-spmat r2*),
add-spmat (*mult-spmat* (*nprt-spmat s2*) (*pprt-spmat r1*), *mult-spmat* (*nprt-spmat s1*) (*nprt-spmat r1*)))))))
apply (*simp add*: *Let-def*)
apply (*insert prems*)
apply (*simp add*: *sparse-row-matrix-op-simps ring-eq-simps*)
apply (*rule mult-le-dual-prts*[**where** *A*=*A*, *simplified Let-def ring-eq-simps*])
apply (*auto*)
done

lemma *spm-mult-le-dual-prts-no-let*:
assumes
sorted-sparse-matrix A1
sorted-sparse-matrix A2
sorted-sparse-matrix c1
sorted-sparse-matrix c2
sorted-sparse-matrix y
sorted-sparse-matrix r1
sorted-sparse-matrix r2
sorted-spvec b
le-spmat ([], *y*)
sparse-row-matrix A1 ≤ *A*
A ≤ *sparse-row-matrix A2*
sparse-row-matrix c1 ≤ *c*

$c \le$ *sparse-row-matrix c2*
sparse-row-matrix r1 $\le x$
$x \le$ *sparse-row-matrix r2*
$A * x \le$ *sparse-row-matrix* (*b*::(*'a*::*lordered-ring*) *spmat*)
shows
$c * x \le$ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
mult-est-spmat r1 r2 (*diff-spmat c1* (*mult-spmat y A2*)) (*diff-spmat c2* (*mult-spmat y A1*))))
by (*simp add*: *prems mult-est-spmat-def spm-mult-le-dual-prts*[**where** *A*=*A*, *simplified Let-def*])

end

theory *FloatSparseMatrix* **imports** *Float SparseMatrix* **begin**

end

theory *Cplex*
imports *FloatSparseMatrix*
uses *Cplex-tools.ML CplexMatrixConverter.ML FloatSparseMatrixBuilder.ML fspmlp.ML*
begin

end

theory *MatrixLP*
imports *Cplex*
begin

constdefs
list-case-compute :: *'b list* ⇒ *'a* ⇒ (*'b* ⇒ *'b list* ⇒ *'a*) ⇒ *'a*
list-case-compute l a f == *list-case a f l*

lemma *list-case-compute*: *list-case* = (λ (*a*::*'a*) *f* (*l*::*'b list*). *list-case-compute l a f*)
apply (*rule ext*)+
apply (*simp add*: *list-case-compute-def*)
done

lemma *list-case-compute-empty*: *list-case-compute* ([]::*'b list*) = (λ (*a*::*'a*) *f*. *a*)
apply (*rule ext*)+

apply (*simp add*: *list-case-compute-def*)
done

lemma *list-case-compute-cons*: *list-case-compute* $(u\#v) = (\lambda\ (a::'a)\ f.\ (f\ (u::'b)\ v))$
apply (*rule ext*)+
apply (*simp add*: *list-case-compute-def*)
done

lemma *If-True*: (*If True*) = $(\lambda\ x\ y.\ x)$
apply (*rule ext*)+
apply *auto*
done

lemma *If-False*: (*If False*) = $(\lambda\ x\ y.\ y)$
apply (*rule ext*)+
apply *auto*
done

lemma *Let-compute*: *Let* $(x::'a)\ f = ((f\ x)::'b)$
by (*simp add*: *Let-def*)

lemma *fst-compute*: *fst* $(a::'a,\ b::'b) = a$
by *auto*

lemma *snd-compute*: *snd* $(a::'a,\ b::'b) = b$
by *auto*

lemma *bool1*: ($\neg$ *True*) = *False* **by** *blast*
lemma *bool2*: ($\neg$ *False*) = *True* **by** *blast*
lemma *bool3*: ((*P*::*bool*) $\wedge$ *True*) = *P* **by** *blast*
lemma *bool4*: (*True* $\wedge$ (*P*::*bool*)) = *P* **by** *blast*
lemma *bool5*: ((*P*::*bool*) $\wedge$ *False*) = *False* **by** *blast*
lemma *bool6*: (*False* $\wedge$ (*P*::*bool*)) = *False* **by** *blast*
lemma *bool7*: ((*P*::*bool*) $\vee$ *True*) = *True* **by** *blast*
lemma *bool8*: (*True* $\vee$ (*P*::*bool*)) = *True* **by** *blast*
lemma *bool9*: ((*P*::*bool*) $\vee$ *False*) = *P* **by** *blast*
lemma *bool10*: (*False* $\vee$ (*P*::*bool*)) = *P* **by** *blast*
lemmas *boolarith* = *bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10*

lemmas *float-arith* = *Float.arith*
lemmas *sparse-row-matrix-arith-simps* = *SparseMatrix.sparse-row-matrix-arith-simps*
lemmas *sorted-sp-simps* = *SparseMatrix.sorted-sp-simps*
lemmas *fst-snd-conv* = *Product-Type.fst-conv Product-Type.snd-conv*

end