

Matrix

Steven Obua

May 23, 2024

```
theory Matrix
imports Main HOL-Library.Lattice-Algebras
begin

type-synonym 'a infmatrix = nat  $\Rightarrow$  nat  $\Rightarrow$  'a

definition nonzero-positions :: (nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero)  $\Rightarrow$  (nat  $\times$  nat) set where
  nonzero-positions A = {pos. A (fst pos) (snd pos)  $\sim$  0}

definition matrix = {(f::(nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero)). finite (nonzero-positions f)}

typedef (overloaded) 'a matrix = matrix :: (nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero) set
  unfolding matrix-def
proof
  show ( $\lambda j i. 0$ )  $\in$  {(f::(nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero)). finite (nonzero-positions f)}
    by (simp add: nonzero-positions-def)
qed

declare Rep-matrix-inverse[simp]

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

definition nrows :: ('a::zero) matrix  $\Rightarrow$  nat where
  nrows A == if nonzero-positions(Rep-matrix A) = {} then 0 else Suc(Max ((image
fst) (nonzero-positions (Rep-matrix A))))

definition ncols :: ('a::zero) matrix  $\Rightarrow$  nat where
  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  $\leq$  m
  shows (Rep-matrix A m n) = 0
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))
have c: finite (?S) by (simp add: finite-nonzero-positions)
from hyp have d: Max (?S) < m by (simp add: a n rows-def)
have m ∉ ?S
  proof -
    have m ∈ ?S ⇒ m ≤ Max (?S) by (simp add: Max-ge [OF c])
    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

```

```

definition transpose-infmatrix :: 'a infmatrix ⇒ 'a infmatrix where
  transpose-infmatrix A j i == A i j

```

```

definition transpose-matrix :: ('a::zero) matrix ⇒ 'a matrix where
  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

```

```

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-eqI)
    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  $y$ )  $\neq 0$  by (simp)
    qed
  next
  fix  $y$ 
  assume hyp: Rep-matrix  $x$  (fst  $y$ ) (snd  $y$ )  $\neq 0$ 
  show  $\exists a b. (\text{Rep-matrix } x \ b \ a \neq 0 \ \& \ y = (b,a))$ 
    by (rule exI[of - snd  $y$ ], rule exI[of - fst  $y$ ]) (simp add: hyp)
  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*: $(x::'a \text{ infmatrix}) = y \implies \forall a b. x \ a \ b = y \ a \ b$ **by** *auto*

```

lemma transpose-infmatrix-inject: (transpose-infmatrix  $A = \text{transpose-infmatrix } B$ ) = ( $A = B$ )
apply (auto)
apply (rule ext)+
apply (simp add: transpose-infmatrix)
apply (drule infmatrixforward)
apply (simp)
done

```

```

lemma transpose-matrix-inject: (transpose-matrix  $A = \text{transpose-matrix } B$ ) = ( $A = B$ )
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  $A$ )  $j \ i = \text{Rep-matrix } A \ i \ j$ 
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*: $\text{ncols } A \leq n \implies \text{Rep-matrix } A \ m \ n = 0$

proof –

assume $\text{ncols } A \leq n$

then have $\text{nrows } (\text{transpose-matrix } A) \leq n$ **by** (*simp*)

then have $\text{Rep-matrix } (\text{transpose-matrix } A) \ n \ m = 0$ **by** (*rule nrows*)

thus $\text{Rep-matrix } A \ m \ n = 0$ **by** (*simp add: transpose-matrix-def*)

qed

lemma *ncols-le*: $(\text{ncols } A \leq n) = (\forall j \ i. \ n \leq i \implies (\text{Rep-matrix } A \ j \ i) = 0)$ (*is - = ?st*)

apply (*auto*)

apply (*simp add: ncols*)

proof (*simp add: ncols-def, auto*)

let $?P = \text{nonzero-positions } (\text{Rep-matrix } A)$

let $?p = \text{snd} \ ?P$

have $a:\text{finite } ?p$ **by** (*simp add: finite-nonzero-positions*)

let $?m = \text{Max } ?p$

assume $\sim(\text{Suc } ?m) \leq n$

then have $b:n \leq ?m$ **by** (*simp*)

fix $a \ b$

assume $(a, b) \in ?P$

then have $?p \neq \{\}$ **by** (*auto*)

with a **have** $?m \in ?p$ **by** (*simp*)

moreover have $\forall x. (x \in ?p \implies (\exists y. (\text{Rep-matrix } A \ y \ x) \neq 0))$ **by** (*simp add: nonzero-positions-def image-def*)

ultimately have $\exists y. (\text{Rep-matrix } A \ y \ ?m) \neq 0$ **by** (*simp*)

moreover assume *?st*

ultimately show *False* **using** b **by** (*simp*)

qed

lemma *less-ncols*: $(n < \text{ncols } A) = (\exists j \ i. \ n \leq i \ \& \ (\text{Rep-matrix } A \ j \ i) \neq 0)$

proof –

have $a:!! (a::\text{nat}) \ b. (a < b) = (\sim(b \leq a))$ **by** *arith*

show *?thesis* **by** (*simp add: a ncols-le*)

qed

lemma *le-ncols*: $(n \leq \text{ncols } A) = (\forall m. (\forall j \ i. \ m \leq i \implies (\text{Rep-matrix } A \ j \ i) = 0) \implies n \leq m)$

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*: $(\text{nrows } A \leq n) = (\forall j \ i. \ n \leq j \implies (\text{Rep-matrix } A \ j \ i) = 0)$

(is ?s)
proof –
 have $(nrows\ A \leq n) = (ncols\ (transpose\text{-}matrix\ A) \leq n)$ **by** (simp)
 also have $\dots = (\forall j\ i.\ n \leq i \longrightarrow (Rep\text{-}matrix\ (transpose\text{-}matrix\ A)\ j\ i = 0))$
by (rule ncols-le)
 also have $\dots = (\forall j\ i.\ n \leq i \longrightarrow (Rep\text{-}matrix\ A\ i\ j) = 0)$ **by** (simp)
 finally show $(nrows\ A \leq n) = (\forall j\ i.\ n \leq j \longrightarrow (Rep\text{-}matrix\ A\ j\ i) = 0)$ **by**
(auto)
qed

lemma less-nrows: $(m < nrows\ A) = (\exists j\ i.\ m \leq j \ \& \ (Rep\text{-}matrix\ A\ j\ i) \neq 0)$
proof –
 have $a:\ !!(a::nat)\ b.\ (a < b) = (\sim(b \leq a))$ **by** arith
 show ?thesis **by** (simp add: a nrows-le)
qed

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

lemma nrows-notzero: $Rep\text{-}matrix\ A\ m\ n \neq 0 \implies m < nrows\ A$
apply (case-tac nrows A $\leq m$)
apply (simp-all add: nrows)
done

lemma ncols-notzero: $Rep\text{-}matrix\ A\ m\ n \neq 0 \implies n < ncols\ A$
apply (case-tac ncols A $\leq n$)
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-eqI, simp, arith)
 moreover assume $finite\ \{x.\ x < n\}$
 ultimately show $finite\ \{x.\ x < Suc\ n\}$ **by** (simp)
qed

lemma finite-natarray2: $finite\ \{(x, y).\ x < (m::nat) \ \& \ y < (n::nat)\}$
by simp

lemma RepAbs-matrix:

assumes $aem: \exists m. \forall j i. m \leq j \longrightarrow x j i = 0$ (**is** $?em$) **and** $aen: \exists n. \forall j i. (n \leq i \longrightarrow x j i = 0)$ (**is** $?en$)
shows $(Rep\text{-matrix} (Abs\text{-matrix } x)) = x$
apply (*rule Abs-matrix-inverse*)
apply (*simp add: matrix-def nonzero-positions-def*)
proof –
from aem **obtain** m **where** $a: \forall j i. m \leq j \longrightarrow x j i = 0$ **by** (*blast*)
from aen **obtain** n **where** $b: \forall j i. n \leq i \longrightarrow x j i = 0$ **by** (*blast*)
let $?u = \{(i, j). x i j \neq 0\}$
let $?v = \{(i, j). i < m \ \& \ j < n\}$
have $c: !! (m::nat) a. \sim(m \leq a) \Longrightarrow a < m$ **by** (*arith*)
from $a b$ **have** $(?u \cap (-?v)) = \{\}$
apply (*simp*)
apply (*rule set-eqI*)
apply (*simp*)
apply *auto*
by (*rule c, auto*)
then have $d: ?u \subseteq ?v$ **by** *blast*
moreover have *finite* $?v$ **by** (*simp add: finite-natarray2*)
moreover have $\{pos. x (fst\ pos) (snd\ pos) \neq 0\} = ?u$ **by** *auto*
ultimately show *finite* $\{pos. x (fst\ pos) (snd\ pos) \neq 0\}$
by (*metis (lifting) finite-subset*)
qed

definition *apply-infmatrix* :: $('a \Rightarrow 'b) \Rightarrow 'a\ infmatrix \Rightarrow 'b\ infmatrix$ **where**
apply-infmatrix $f == \% A. (\% j i. f (A j i))$

definition *apply-matrix* :: $('a \Rightarrow 'b) \Rightarrow ('a::zero)\ matrix \Rightarrow ('b::zero)\ matrix$ **where**
apply-matrix $f == \% A. Abs\text{-matrix} (apply\text{-infmatrix } f (Rep\text{-matrix } A))$

definition *combine-infmatrix* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ infmatrix \Rightarrow 'b\ infmatrix \Rightarrow 'c\ infmatrix$ **where**
combine-infmatrix $f == \% A B. (\% j i. f (A j i) (B j i))$

definition *combine-matrix* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a::zero)\ matrix \Rightarrow ('b::zero)\ matrix \Rightarrow ('c::zero)\ matrix$ **where**
combine-matrix $f == \% A B. Abs\text{-matrix} (combine\text{-infmatrix } f (Rep\text{-matrix } A) (Rep\text{-matrix } B))$

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

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

definition *commutative* :: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow bool$ **where**
commutative $f == \forall x y. f x y = f y x$

definition *associative* :: ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**
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 situation: Assume that we have sets A and B with $B \subset A$ and an abstraction $u : A \rightarrow B$. This abstraction has to fulfill $u(b) = b$ for all $b \in B$, but is arbitrary otherwise. Each function $f : A \times A \rightarrow A$ now induces a function $f' : B \times B \rightarrow B$ 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 ⇒ *commutative* (combine-infmatrix f)
by (*simp* add: *commutative-def* *combine-infmatrix-def*)

lemma *combine-matrix-commute*:
commutative f ⇒ *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*]: $f\ 0\ 0 = 0 \implies \text{nonzero-positions } (\text{combine-infmatrix } f\ A\ B) \subseteq (\text{nonzero-positions } A) \cup (\text{nonzero-positions } B)$
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*]:
 $f\ 0\ 0 = 0 \implies \text{Rep-matrix } (\text{Abs-matrix } (\text{combine-infmatrix } f\ (\text{Rep-matrix } A)\ (\text{Rep-matrix } B))) = \text{combine-infmatrix } f\ (\text{Rep-matrix } A)\ (\text{Rep-matrix } B)$
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]: $f \ 0 = 0 \implies \text{nonzero-positions}(\text{apply-infmatrix } f \ A) \subseteq \text{nonzero-positions } A$
by (rule subsetI, simp add: nonzero-positions-def apply-infmatrix-def, auto)

lemma *apply-infmatrix-closed* [simp]:
 $f \ 0 = 0 \implies \text{Rep-matrix}(\text{Abs-matrix}(\text{apply-infmatrix } f \ (\text{Rep-matrix } A))) = \text{apply-infmatrix } f \ (\text{Rep-matrix } A)$
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]: $f \ 0 \ 0 = 0 \implies \text{associative } f \implies \text{associative}(\text{combine-infmatrix } f)$
by (simp add: associative-def combine-infmatrix-def)

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

lemma *combine-matrix-assoc*: $f \ 0 \ 0 = 0 \implies \text{associative } f \implies \text{associative}(\text{combine-matrix } f)$
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]: $f \ 0 = 0 \implies \text{Rep-matrix}(\text{apply-matrix } f \ A) \ j \ i = f \ (\text{Rep-matrix } A \ j \ i)$
by (simp add: apply-matrix-def)

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

lemma *combine-nrows-max*: $f \ 0 \ 0 = 0 \implies \text{nrows}(\text{combine-matrix } f \ A \ B) \leq \max(\text{nrows } A) \ (\text{nrows } B)$
by (simp add: nrows-le)

lemma *combine-ncols-max*: $f \ 0 \ 0 = 0 \implies \text{ncols}(\text{combine-matrix } f \ A \ B) \leq \max(\text{ncols } A) \ (\text{ncols } B)$
by (simp add: ncols-le)

lemma *combine-nrows*: $f \ 0 \ 0 = 0 \implies \text{nrows } A \leq q \implies \text{nrows } B \leq q \implies \text{nrows}(\text{combine-matrix } f \ A \ B) \leq q$
by (simp add: nrows-le)

lemma *combine-ncols*: $f \ 0 \ 0 = 0 \implies \text{ncols } A \leq q \implies \text{ncols } B \leq q \implies \text{ncols}(\text{combine-matrix } f \ A \ B) \leq q$

by (*simp add: ncols-le*)

definition *zero-r-neutral* :: ('a ⇒ 'b::zero ⇒ 'a) ⇒ bool **where**
zero-r-neutral f == ∀ a. f a 0 = a

definition *zero-l-neutral* :: ('a::zero ⇒ 'b ⇒ 'b) ⇒ bool **where**
zero-l-neutral f == ∀ a. f 0 a = a

definition *zero-closed* :: (('a::zero) ⇒ ('b::zero) ⇒ ('c::zero)) ⇒ bool **where**
zero-closed f == (∀ x. f x 0 = 0) & (∀ y. f 0 y = 0)

primrec *foldseq* :: ('a ⇒ 'a ⇒ 'a) ⇒ (nat ⇒ 'a) ⇒ nat ⇒ 'a
where
foldseq f s 0 = s 0
| *foldseq* f s (Suc n) = f (s 0) (*foldseq* f (% k. s(Suc k)) n)

primrec *foldseq-transposed* :: ('a ⇒ 'a ⇒ 'a) ⇒ (nat ⇒ 'a) ⇒ nat ⇒ 'a
where
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*: $\bigwedge n. \forall N s. N \leq n \longrightarrow \text{foldseq } f s N = \text{foldseq-transposed } f s N$
proof –
fix n
show $\forall N s. N \leq n \longrightarrow \text{foldseq } f s N = \text{foldseq-transposed } f s N$
proof (*induct* n)
show $\forall N s. N \leq 0 \longrightarrow \text{foldseq } f s N = \text{foldseq-transposed } f s N$ **by** *simp*
next
fix n
assume *b*: $\forall N s. N \leq n \longrightarrow \text{foldseq } f s N = \text{foldseq-transposed } f s N$
have *c*: $\bigwedge N s. N \leq n \Longrightarrow \text{foldseq } f s N = \text{foldseq-transposed } f s N$ **by** (*simp add: b*)
show $\forall N t. N \leq \text{Suc } n \longrightarrow \text{foldseq } f t N = \text{foldseq-transposed } f t N$
proof (*auto*)
fix N t
assume *Nsuc*: $N \leq \text{Suc } n$
show $\text{foldseq } f t N = \text{foldseq-transposed } f t N$
proof *cases*
assume $N \leq n$
then show $\text{foldseq } f t N = \text{foldseq-transposed } f t N$ **by** (*simp add: b*)
next
assume $\sim(N \leq n)$
with *Nsuc* **have** *Nsuc*: $N = \text{Suc } n$ **by** *simp*
have *neqz*: $n \neq 0 \Longrightarrow \exists m. n = \text{Suc } m \ \& \ \text{Suc } m \leq n$ **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
  show foldseq f = foldseq-transposed f by ((rule ext)+, insert sublemma, auto)
  qed

lemma foldseq-distr: [[associative f; commutative f]] ==> 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  $\bigwedge n. (\forall u v. \text{foldseq } f \text{ } (\%k. f (u k) (v k)) n = f (\text{foldseq } f \text{ } u \text{ } n) (\text{foldseq } f \text{ } v \text{ } n))$ 
  apply (induct-tac n)
  apply (simp+, auto)
  by (simp add: a b c)

```

then show $\text{foldseq } f (\% k. f (u\ k) (v\ k))\ n = f (\text{foldseq } f\ u\ n) (\text{foldseq } f\ v\ n)$ **by**
simp
qed

theorem $\llbracket \text{associative } f; \text{associative } g; \forall a\ b\ c\ d. g (f\ a\ b) (f\ c\ d) = f (g\ a\ c) (g\ b\ d); \exists x\ y. (f\ x) \neq (f\ y); \exists x\ y. (g\ x) \neq (g\ y); f\ x\ x = x; g\ x\ x = x \rrbracket \implies f=g \mid (\forall y. f\ y\ x = y) \mid (\forall y. g\ y\ x = y)$
oops

lemma *foldseq-zero*:

assumes *fz*: $f\ 0\ 0 = 0$ **and** *sz*: $\forall i. i \leq n \longrightarrow s\ i = 0$

shows $\text{foldseq } f\ s\ n = 0$

proof –

have $\bigwedge n. \forall s. (\forall i. i \leq n \longrightarrow s\ i = 0) \longrightarrow \text{foldseq } f\ s\ n = 0$

apply (*induct-tac n*)

apply (*simp*)

by (*simp add: fz*)

then show $\text{foldseq } f\ s\ n = 0$ **by** (*simp add: sz*)

qed

lemma *foldseq-significant-positions*:

assumes *p*: $\forall i. i \leq N \longrightarrow S\ i = T\ i$

shows $\text{foldseq } f\ S\ N = \text{foldseq } f\ T\ N$

proof –

have $\bigwedge m. \forall s\ t. (\forall i. i \leq m \longrightarrow s\ i = t\ i) \longrightarrow \text{foldseq } f\ s\ m = \text{foldseq } f\ t\ m$

apply (*induct-tac m*)

apply (*simp*)

apply (*simp*)

apply (*auto*)

proof –

fix *n*

fix *s*: $\text{nat} \Rightarrow 'a$

fix *t*: $\text{nat} \Rightarrow 'a$

assume *a*: $\forall s\ t. (\forall i \leq n. s\ i = t\ i) \longrightarrow \text{foldseq } f\ s\ n = \text{foldseq } f\ t\ n$

assume *b*: $\forall i \leq \text{Suc } n. s\ i = t\ i$

have *c*: $a\ b. a = b \implies f (t\ 0)\ a = f (t\ 0)\ b$ **by** *blast*

have *d*: $s\ t. (\forall i \leq n. s\ i = t\ i) \implies \text{foldseq } f\ s\ n = \text{foldseq } f\ t\ n$ **by** (*simp add:*

a)

show $f (t\ 0) (\text{foldseq } f (\lambda k. s (\text{Suc } k))\ n) = f (t\ 0) (\text{foldseq } f (\lambda k. t (\text{Suc } k))$

n) **by** (*rule c, simp add: d b*)

qed

with *p* **show** *?thesis* **by** *simp*

qed

lemma *foldseq-tail*:

assumes $M \leq N$

shows $\text{foldseq } f\ S\ N = \text{foldseq } f (\% k. (\text{if } k < M \text{ then } (S\ k) \text{ else } (\text{foldseq } f (\% k. S(k+M)) (N-M))))\ M$

```

proof –
  have suc:  $\bigwedge a b. \llbracket a \leq \text{Suc } b; a \neq \text{Suc } b \rrbracket \implies a \leq b$  by arith
  have a:  $\bigwedge a b c. a = b \implies f c a = f c b$  by blast
  have  $\bigwedge n. \forall m s. m \leq n \longrightarrow \text{foldseq } f s n = \text{foldseq } f (\% k. (\text{if } k < m \text{ then } (s k) \text{ else } (\text{foldseq } f (\% k. s(k+m)) (n-m)))) m$ 
  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:  $\forall m \leq na. \forall s. \text{foldseq } f s na = \text{foldseq } f (\lambda k. \text{if } k < m \text{ then } s k \text{ else } \text{foldseq } f (\lambda k. s (k + m)) (na - m)) m$ 
    assume subb:  $m \leq na$ 
    from suba have subc:!!  $m s. m \leq na \implies \text{foldseq } f s na = \text{foldseq } f (\lambda k. \text{if } k < m \text{ then } s k \text{ else } \text{foldseq } f (\lambda k. s (k + m)) (na - m)) m$  by simp
    have subd:  $\text{foldseq } f (\lambda k. \text{if } k < m \text{ then } s (\text{Suc } k) \text{ else } \text{foldseq } f (\lambda k. s (\text{Suc } (k + m))) (na - m)) m =$ 
       $\text{foldseq } f (\% k. s(\text{Suc } k)) na$ 
    by (rule subc[of m % k. s(Suc k), THEN sym], simp add: subb)
    from subb have sube:  $m \neq 0 \implies \exists mm. m = \text{Suc } mm \ \& \ mm \leq na$  by arith
    show  $f (s 0) (\text{foldseq } f (\lambda k. \text{if } k < m \text{ then } s (\text{Suc } k) \text{ else } \text{foldseq } f (\lambda k. s (\text{Suc } (k + m))) (na - m)) m) =$ 
       $\text{foldseq } f (\lambda k. \text{if } k < m \text{ then } s k \text{ else } \text{foldseq } f (\lambda k. s (k + m)) (\text{Suc } na - m)) m$ 
    apply (simp add: subd)
    apply (cases m = 0)
    apply simp
    apply (drule sube)
    apply auto
    apply (rule a)
    apply (simp add: subc cong del: if-weak-cong)
    done
  qed
  then show ?thesis using assms by simp
qed

```

lemma *foldseq-zerotail*:

```

assumes
  fz:  $f 0 0 = 0$ 
and sz:  $\forall i. n \leq i \longrightarrow s i = 0$ 
and nm:  $n \leq m$ 

```

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 $\forall x. f\ x\ 0 = x$
and $\forall i. n < i \longrightarrow s\ i = 0$
and $nm: n \leq m$
shows $foldseq\ f\ s\ n = foldseq\ f\ s\ m$
proof –
have $f\ 0\ 0 = 0$ **by** (*simp add: assms*)
have $b: \bigwedge m\ n. n \leq m \implies m \neq n \implies \exists k. m - n = Suc\ k$ **by** *arith*
have $c: 0 \leq m$ **by** *simp*
have $d: \bigwedge k. k \neq 0 \implies \exists l. k = Suc\ l$ **by** *arith*
show *?thesis*
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: assms*)
apply (*drule d*)
apply (*auto*)
apply (*simp add: assms foldseq-zero*)
done
qed

lemma foldseq-zerostart:
 $\forall x. f\ 0\ (f\ 0\ x) = f\ 0\ x \implies \forall i. i \leq n \longrightarrow s\ i = 0 \implies foldseq\ f\ s\ (Suc\ n) = f\ 0\ (s\ (Suc\ n))$
proof –
assume $f00x: \forall x. f\ 0\ (f\ 0\ x) = f\ 0\ x$
have $\forall s. (\forall i. i \leq n \longrightarrow s\ i = 0) \longrightarrow foldseq\ f\ s\ (Suc\ n) = f\ 0\ (s\ (Suc\ n))$
apply (*induct n*)
apply (*simp*)
apply (*rule allI, rule impI*)
proof –
fix n
fix s

have a : $\text{foldseq } f \ s \ (\text{Suc } (\text{Suc } n)) = f \ (s \ 0) \ (\text{foldseq } f \ (\% \ k. \ s(\text{Suc } k)) \ (\text{Suc } n))$
by *simp*
assume b : $\forall s. ((\forall i \leq n. \ s \ i = 0) \longrightarrow \text{foldseq } f \ s \ (\text{Suc } n) = f \ 0 \ (s \ (\text{Suc } n)))$
from b **have** c : $\forall s. (\forall i \leq n. \ s \ i = 0) \Longrightarrow \text{foldseq } f \ s \ (\text{Suc } n) = f \ 0 \ (s \ (\text{Suc } n))$
by *simp*
assume d : $\forall i. \ i \leq \text{Suc } n \longrightarrow s \ i = 0$
show $\text{foldseq } f \ s \ (\text{Suc } (\text{Suc } n)) = f \ 0 \ (s \ (\text{Suc } (\text{Suc } n)))$
apply (*subst* a)
apply (*subst* c)
by (*simp add*: $d \ f00x$)
qed
then show $\forall i. \ i \leq n \longrightarrow s \ i = 0 \Longrightarrow \text{foldseq } f \ s \ (\text{Suc } n) = f \ 0 \ (s \ (\text{Suc } n))$
by *simp*
qed

lemma *foldseq-zerostart2*:

$\forall x. \ f \ 0 \ x = x \Longrightarrow \forall i. \ i < n \longrightarrow s \ i = 0 \Longrightarrow \text{foldseq } f \ s \ n = s \ n$

proof –

assume a : $\forall i. \ i < n \longrightarrow s \ i = 0$
assume x : $\forall x. \ f \ 0 \ x = x$
from x **have** $f00x$: $\forall x. \ f \ 0 \ (f \ 0 \ x) = f \ 0 \ x$ **by** *blast*
have b : $\bigwedge i \ l. \ i < \text{Suc } l = (i \leq l)$ **by** *arith*
have d : $\bigwedge k. \ k \neq 0 \Longrightarrow \exists l. \ k = \text{Suc } l$ **by** *arith*
show $\text{foldseq } f \ s \ n = s \ n$
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 $f0x$: $\forall x. \ f \ 0 \ x = x$ **and** $fx0$: $\forall x. \ f \ x \ 0 = x$ **and** $s0$: $\forall i. \ i \neq j \longrightarrow s \ i = 0$

shows $\text{foldseq } f \ s \ n = (\text{if } (j \leq n) \text{ then } (s \ j) \text{ else } 0)$

proof –

from $s0$ **have** a : $\forall i. \ i < j \longrightarrow s \ i = 0$ **by** *simp*
from $s0$ **have** b : $\forall i. \ j < i \longrightarrow s \ i = 0$ **by** *simp*
show *?thesis*
apply *auto*
apply (*subst* *foldseq-zero**tail2*[*of* f , *OF* $fx0$, *of* j , *OF* b , *of* n , *THEN* *sym*])
apply *simp*
apply (*subst* *foldseq-zero**start2*)
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 $g(\text{foldseq } f \ s \ n) = \text{foldseq } f \ (\% \ x. \ g(s \ x)) \ n$

proof –

have $\forall s. g(\text{foldseq } f \ s \ n) = \text{foldseq } f \ (\% \ x. \ g(s \ x)) \ n$
apply (*induct-tac n*)
apply (*simp*)
apply (*simp*)
apply (*auto*)
apply (*drule-tac x=% k. s (Suc k) in spec*)
by (*simp add: assms*)
then show *?thesis* **by** *simp*

qed

definition *mult-matrix-n* :: $\text{nat} \Rightarrow (('a::\text{zero}) \Rightarrow ('b::\text{zero}) \Rightarrow ('c::\text{zero})) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'a \ \text{matrix} \Rightarrow 'b \ \text{matrix} \Rightarrow 'c \ \text{matrix}$ **where**
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*)

definition *mult-matrix* :: $(('a::\text{zero}) \Rightarrow ('b::\text{zero}) \Rightarrow ('c::\text{zero})) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'a \ \text{matrix} \Rightarrow 'b \ \text{matrix} \Rightarrow 'c \ \text{matrix}$ **where**
mult-matrix fmul fadd A B == mult-matrix-n (*max (ncols A) (nrows B)*) *fmul fadd A B*

lemma *mult-matrix-n*:

assumes $\text{ncols } A \leq n$ (**is** *?An*) $\text{nrows } B \leq n$ (**is** *?Bn*) $\text{fadd } 0 \ 0 = 0$ $\text{fmul } 0 \ 0 = 0$

shows $c:\text{mult-matrix fmul fadd } A \ B = \text{mult-matrix-n } n \ \text{fmul fadd } A \ B$

proof –

show *?thesis* **using** *assms*
apply (*simp add: mult-matrix-def mult-matrix-n-def*)
apply (*rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+*)
apply (*rule foldseq-zerotail, simp-all add: nrows-le ncols-le assms*)
done

qed

lemma *mult-matrix-nm*:

assumes $\text{ncols } A \leq n$ $\text{nrows } B \leq n$ $\text{ncols } A \leq m$ $\text{nrows } B \leq m$ $\text{fadd } 0 \ 0 = 0$ $\text{fmul } 0 \ 0 = 0$

shows $\text{mult-matrix-n } n \ \text{fmul fadd } A \ B = \text{mult-matrix-n } m \ \text{fmul fadd } A \ B$

proof –

from *assms* **have** $\text{mult-matrix-n } n \ \text{fmul fadd } A \ B = \text{mult-matrix-n } m \ \text{fmul fadd } A \ B$
by (*simp add: mult-matrix-n*)
also from *assms* **have** $\dots = \text{mult-matrix-n } m \ \text{fmul fadd } A \ B$
by (*simp add: mult-matrix-n[THEN sym]*)
finally show $\text{mult-matrix-n } n \ \text{fmul fadd } A \ B = \text{mult-matrix-n } m \ \text{fmul fadd } A \ B$

by *simp*
qed

definition *r-distributive* :: ('a ⇒ 'b ⇒ 'b) ⇒ ('b ⇒ 'b ⇒ 'b) ⇒ bool **where**
r-distributive *fmul* *fadd* == ∀ a u v. *fmul* a (*fadd* u v) = *fadd* (*fmul* a u) (*fmul* a v)

definition *l-distributive* :: ('a ⇒ 'b ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**
l-distributive *fmul* *fadd* == ∀ a u v. *fmul* (*fadd* u v) a = *fadd* (*fmul* u a) (*fmul* v a)

definition *distributive* :: ('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ bool **where**
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

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*)

proof –

from *assms* **show** ?thesis

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 *assms* **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
  l-distributive fmul fadd
  associative fadd
  commutative fadd
  fadd 0 0 = 0
   $\forall a. \text{fmul } a \ 0 = 0$ 
   $\forall a. \text{fmul } 0 \ a = 0$ 
shows l-distributive (mult-matrix fmul fadd) (combine-matrix fadd)
proof -
from assms show ?thesis
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 assms 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

instantiation matrix :: (zero) zero
begin

definition zero-matrix-def: 0 = Abs-matrix ( $\lambda j i. 0$ )

instance ..

end

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  $\implies$  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  $\implies$  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  $\implies$  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  $\implies$  zero-r-neutral (combine-matrix
f)
  by (simp add: zero-r-neutral-def combine-matrix-def combine-infmatrix-def)

lemma mult-matrix-zero-closed:  $\llbracket \text{fadd } 0 \ 0 = 0; \text{ zero-closed fmul} \rrbracket \implies \text{zero-closed}$ 
(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]:  $\llbracket \text{fadd } 0 \ 0 = 0; \forall a. \text{fmul } a \ 0 = 0 \rrbracket \implies$ 
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]: $\llbracket \text{fadd } 0 \ 0 = 0; \forall a. \text{fmul } 0 \ a = 0 \rrbracket \implies$
 $\text{mult-matrix-n } n \ \text{fmul } \text{fadd } 0 \ A = 0$

apply (*simp add: mult-matrix-n-def*)
apply (*subst foldseq-zero*)
by (*simp-all add: zero-matrix-def*)

lemma *mult-matrix-zero-left*[simp]: $\llbracket \text{fadd } 0 \ 0 = 0; \forall a. \text{fmul } 0 \ a = 0 \rrbracket \implies$ *mult-matrix*
 $\text{fmul } \text{fadd } 0 \ A = 0$

by (*simp add: mult-matrix-def*)

lemma *mult-matrix-zero-right*[simp]: $\llbracket \text{fadd } 0 \ 0 = 0; \forall a. \text{fmul } a \ 0 = 0 \rrbracket \implies$
 $\text{mult-matrix } \text{fmul } \text{fadd } A \ 0 = 0$

by (*simp add: mult-matrix-def*)

lemma *apply-matrix-zero*[simp]: $f \ 0 = 0 \implies \text{apply-matrix } f \ 0 = 0$

apply (*simp add: apply-matrix-def apply-infmatrix-def*)
by (*simp add: zero-matrix-def*)

lemma *combine-matrix-zero*: $f \ 0 \ 0 = 0 \implies \text{combine-matrix } f \ 0 \ 0 = 0$

apply (*simp add: combine-matrix-def combine-infmatrix-def*)
by (*simp add: zero-matrix-def*)

lemma *transpose-matrix-zero*[simp]: $\text{transpose-matrix } 0 = 0$

apply (*simp add: transpose-matrix-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]: $\text{apply-matrix } (\% \ x. \ 0) \ A = 0$

apply (*simp add: apply-matrix-def apply-infmatrix-def*)
by (*simp add: zero-matrix-def*)

definition *singleton-matrix* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('a::\text{zero}) \Rightarrow 'a \ \text{matrix}$ **where**

singleton-matrix $j \ i \ a == \text{Abs-matrix}(\% \ m \ n. \ \text{if } j = m \ \& \ i = n \ \text{then } a \ \text{else } 0)$

definition *move-matrix* :: $('a::\text{zero}) \ \text{matrix} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow 'a \ \text{matrix}$ **where**

move-matrix $A \ y \ x == \text{Abs-matrix}(\% \ j \ i. \ \text{if } (((\text{int } j) - y) < 0) \mid (((\text{int } i) - x) < 0)) \ \text{then } 0 \ \text{else } \text{Rep-matrix } A \ (\text{nat } ((\text{int } j) - y)) \ (\text{nat } ((\text{int } i) - x)))$

definition *take-rows* :: $('a::\text{zero}) \ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a \ \text{matrix}$ **where**

take-rows $A \ r == \text{Abs-matrix}(\% \ j \ i. \ \text{if } (j < r) \ \text{then } (\text{Rep-matrix } A \ j \ i) \ \text{else } 0)$

definition *take-columns* :: $('a::\text{zero}) \ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a \ \text{matrix}$ **where**

take-columns $A \ c == \text{Abs-matrix}(\% \ j \ i. \ \text{if } (i < c) \ \text{then } (\text{Rep-matrix } A \ j \ i) \ \text{else } 0)$

definition *column-of-matrix* :: $('a::\text{zero}) \ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a \ \text{matrix}$ **where**

column-of-matrix $A \ n == \text{take-columns } (\text{move-matrix } A \ 0 \ (- \ \text{int } n)) \ 1$

definition *row-of-matrix* :: ('a::zero) matrix \Rightarrow nat \Rightarrow 'a matrix **where**
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 \implies 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)
proof –
have th: $\neg (\forall m. m \leq j) \exists n. \neg n \leq i$ **by** arith+
from th **show** ?thesis
apply (auto)
apply (rule le-antisym)
apply (subst nrows-le)
apply (simp add: singleton-matrix-def, auto)
apply (subst RepAbs-matrix)
apply auto
apply (simp add: Suc-le-eq)
apply (rule not-le-imp-less)
apply (subst nrows-le)
by simp
qed

lemma *ncols-singleton[simp]*: ncols(singleton-matrix j i e) = (if e = 0 then 0 else Suc i)
proof –
have th: $\neg (\forall m. m \leq j) \exists n. \neg n \leq i$ **by** arith+
from th **show** ?thesis
apply (auto)
apply (rule le-antisym)
apply (subst ncols-le)
apply (simp add: singleton-matrix-def, auto)
apply (subst RepAbs-matrix)

apply *auto*
apply (*simp add: Suc-le-eq*)
apply (*rule not-le-imp-less*)
apply (*subst ncols-le*)
by *simp*
qed

lemma *combine-singleton*: $f\ 0\ 0 = 0 \implies \text{combine-matrix } f\ (\text{singleton-matrix } j\ i\ a)\ (\text{singleton-matrix } j\ i\ b) = \text{singleton-matrix } j\ i\ (f\ a\ b)$
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]*: $\text{transpose-matrix } (\text{singleton-matrix } j\ i\ a) = \text{singleton-matrix } i\ j\ a$
apply (*subst Rep-matrix-inject[symmetric], (rule ext)+*)
apply (*simp*)
done

lemma *Rep-move-matrix[simp]*:
 $\text{Rep-matrix } (\text{move-matrix } A\ y\ x)\ j\ i =$
(if (((int j)-y) < 0) | (((int i)-x) < 0) 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 |y|)], auto, rule nrows, arith,
rule exI[of - (ncols A)+(nat |x|)], auto, rule ncols, arith)+

lemma *move-matrix-0-0[simp]*: $\text{move-matrix } A\ 0\ 0 = A$
by (*simp add: move-matrix-def*)

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

lemma *transpose-move-matrix[simp]*:
 $\text{transpose-matrix } (\text{move-matrix } A\ x\ y) = \text{move-matrix } (\text{transpose-matrix } A)\ y\ x$
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, arith*)
apply *simp*
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 \implies 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 \implies 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
 $\forall x. \text{fmul } x \ 0 = 0$
 $\forall x. \text{fmul } 0 \ x = 0$
 $\forall x. \text{fadd } 0 \ x = x$
 $\forall x. \text{fadd } x \ 0 = x$
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 (*auto*)
apply (*simp add: assms*)
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: assms*)
apply (*auto*)
done

lemma *mult-matrix-ext*:

assumes
eprem:
 $\exists e. (\forall a \ b. a \neq b \longrightarrow \text{fmul } a \ e \neq \text{fmul } b \ e)$
and *fprems*:
 $\forall a. \text{fmul } 0 \ a = 0$
 $\forall a. \text{fmul } a \ 0 = 0$
 $\forall a. \text{fadd } a \ 0 = a$
 $\forall a. \text{fadd } 0 \ a = a$
and *contraprems*:
 $\text{mult-matrix fmul fadd } A = \text{mult-matrix fmul fadd } B$
shows
 $A = B$
proof(*rule contrapos-np[of False], simp*)
assume *a*: $A \neq B$
have *b*: $\bigwedge f \ g. (\forall x \ y. f \ x \ y = g \ x \ y) \Longrightarrow f = g$ **by** (*(rule ext)+, auto*)
have $\exists j \ i. (\text{Rep-matrix } A \ j \ i) \neq (\text{Rep-matrix } B \ j \ i)$
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*: $(\text{Rep-matrix } A \ J \ I) \neq (\text{Rep-matrix } B \ J \ I)$ **by** *blast*
from *eprem* **obtain** *e* **where** *eprops*: $(\forall a \ b. a \neq b \longrightarrow \text{fmul } a \ e \neq \text{fmul } b \ e)$ **by** *blast*
let *?S* = *singleton-matrix I 0 e*
let *?comp* = *mult-matrix fmul fadd*
have *d*: $\forall x \ f \ g. f = g \Longrightarrow f \ x = g \ x$ **by** *blast*
have *e*: $(\% x. \text{fmul } x \ e) \ 0 = 0$ **by** (*simp add: assms*)

```

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 contrapremis show False by simp
qed

```

```

definition foldmatrix :: ('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ ('a infmatrix) ⇒
nat ⇒ nat ⇒ 'a where
  foldmatrix f g A m n == foldseq-transposed g (% j. foldseq f (A j) n) m

```

```

definition foldmatrix-transposed :: ('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ ('a
infmatrix) ⇒ nat ⇒ nat ⇒ 'a where
  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
proof -
  have forall: ∧ P x. (∀ x. P x) ⇒ 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 assms)+
    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 (λu. 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*]
assms)

lemma *mult-n-nrows:*

assumes

$\forall a. \text{fmul } 0 \ a = 0$

$\forall a. \text{fmul } a \ 0 = 0$

$\text{fadd } 0 \ 0 = 0$

shows $\text{nrows } (\text{mult-matrix-n } n \ \text{fmul } \text{fadd } A \ B) \leq \text{nrows } A$

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 assms foldseq-zero*)

apply (*rule-tac x=ncols B in exI*)

apply (*simp add: ncols assms foldseq-zero*)

apply (*simp add: nrows assms foldseq-zero*)

done

lemma *mult-n-ncols:*

assumes

$\forall a. \text{fmul } 0 \ a = 0$

$\forall a. \text{fmul } a \ 0 = 0$

$\text{fadd } 0 \ 0 = 0$

shows $\text{ncols } (\text{mult-matrix-n } n \ \text{fmul } \text{fadd } A \ B) \leq \text{ncols } B$

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 assms foldseq-zero*)

apply (*rule-tac x=ncols B in exI*)

apply (*simp add: ncols assms foldseq-zero*)

apply (*simp add: ncols assms foldseq-zero*)

done

lemma *mult-nrows:*

assumes

$\forall a. \text{fmul } 0 \ a = 0$

$\forall a. \text{fmul } a \ 0 = 0$

$\text{fadd } 0 \ 0 = 0$

shows $\text{nrows } (\text{mult-matrix } \text{fmul } \text{fadd } A \ B) \leq \text{nrows } A$

by (*simp add: mult-matrix-def mult-n-nrows assms*)

lemma *mult-ncols:*

assumes

$\forall a. \text{fmul } 0 \ a = 0$

$\forall a. \text{fmul } a \ 0 = 0$

```

fadd 0 0 = 0
shows ncols (mult-matrix fmul fadd A B) ≤ ncols B
by (simp add: mult-matrix-def mult-n-ncols assms)

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
    ∀ 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)
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], insert
assms, simp-all)
  have fmul1fadd2fold: !! x s n. fmul1 x (foldseq fadd2 s n) = foldseq fadd2 (% k.
fmul1 x (s k)) n
  using assms 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 ?thesis
    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 assms)+
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 assms)+
apply (simplesubst mult-matrix-nm[of - - ?N])
apply (simp add: max1 max2 mult-n-ncols mult-n-nrows assms)+
apply (simplesubst mult-matrix-nm[of - - ?N])
apply (simp add: max1 max2 mult-n-ncols mult-n-nrows assms)+
apply (simplesubst mult-matrix-nm[of - - ?N])
apply (simp add: max1 max2 mult-n-ncols mult-n-nrows assms)+
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 assms foldseq-zero)
apply (rule exI[of - ncols C])
apply (simp add: assms ncols foldseq-zero)
apply (subst RepAbs-matrix)
apply (rule exI[of - nrows A])
apply (simp add: nrows assms foldseq-zero)
apply (rule exI[of - ncols B])
apply (simp add: assms ncols foldseq-zero)
apply (simp add: fmul2fadd1fold fmul1fadd2fold assms)
apply (subst foldseq-foldseq)
apply (simp add: assms)+
apply (simp add: transpose-infmatrix)
done

```

qed

lemma

assumes

$\forall a. \text{fmul1 } 0 \ a = 0$

$\forall a. \text{fmul1 } a \ 0 = 0$

$\forall a. \text{fmul2 } 0 \ a = 0$

$\forall a. \text{fmul2 } a \ 0 = 0$

$\text{fadd1 } 0 \ 0 = 0$

$\text{fadd2 } 0 \ 0 = 0$

$\forall a \ b \ c \ d. \text{fadd2 } (\text{fadd1 } a \ b) (\text{fadd1 } c \ d) = \text{fadd1 } (\text{fadd2 } a \ c) (\text{fadd2 } b \ d)$

associative fadd1

associative fadd2

$\forall a \ b \ c. \text{fmul2 } (\text{fmul1 } a \ b) \ c = \text{fmul1 } a \ (\text{fmul2 } b \ c)$

$\forall a \ b \ c. \text{fmul2 } (\text{fadd1 } a \ b) \ c = \text{fadd1 } (\text{fmul2 } a \ c) (\text{fmul2 } b \ c)$

$\forall a \ b \ c. \text{fmul1 } c \ (\text{fadd2 } a \ b) = \text{fadd2 } (\text{fmul1 } c \ a) (\text{fmul1 } c \ b)$

```

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 )
apply (simp add: mult-matrix-assoc assms)
done

```

lemma *mult-matrix-assoc-simple*:

```

assumes
   $\forall a. \text{fmul } 0 \ a = 0$ 
   $\forall a. \text{fmul } a \ 0 = 0$ 
   $\text{fadd } 0 \ 0 = 0$ 
  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)
proof -
  have !! a b c d. fadd (fadd a b) (fadd c d) = fadd (fadd a c) (fadd b d)
    using assms by (simp add: associative-def commutative-def)
  then show ?thesis
    apply (subst mult-matrix-assoc)
    using assms
    apply simp-all
    apply (simp-all add: associative-def distributive-def l-distributive-def r-distributive-def)
  done
qed

```

```

lemma transpose-apply-matrix:  $f \ 0 = 0 \implies \text{transpose-matrix } (\text{apply-matrix } f \ A)$ 
  =  $\text{apply-matrix } f \ (\text{transpose-matrix } A)$ 
apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
by simp

```

```

lemma transpose-combine-matrix:  $f \ 0 \ 0 = 0 \implies \text{transpose-matrix } (\text{combine-matrix}$ 
   $f \ A \ B) = \text{combine-matrix } f \ (\text{transpose-matrix } A) \ (\text{transpose-matrix } B)$ 
apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
by simp

```

lemma *Rep-mult-matrix*:

```

assumes
   $\forall a. \text{fmul } 0 \ a = 0$ 
   $\forall a. \text{fmul } a \ 0 = 0$ 
   $\text{fadd } 0 \ 0 = 0$ 
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], insert assms, simp add: nrows foldseq-zero)
apply (rule exI[of - ncols B], insert assms, simp add: ncols foldseq-zero)
apply simp
done

```

lemma *transpose-mult-matrix*:

```

assumes
   $\forall a. \text{fmul } 0 \ a = 0$ 
   $\forall a. \text{fmul } a \ 0 = 0$ 
   $\text{fadd } 0 \ 0 = 0$ 
   $\forall x \ y. \text{fmul } y \ x = \text{fmul } x \ y$ 
shows
   $\text{transpose-matrix } (\text{mult-matrix fmul fadd } A \ B) = \text{mult-matrix fmul fadd } (\text{transpose-matrix } B) \ (\text{transpose-matrix } A)$ 
apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
using assms
apply (simp add: Rep-mult-matrix ac-simps)
done

```

lemma *column-transpose-matrix*: $\text{column-of-matrix } (\text{transpose-matrix } A) \ n = \text{transpose-matrix } (\text{row-of-matrix } A \ n)$

```

apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
by simp

```

lemma *take-columns-transpose-matrix*: $\text{take-columns } (\text{transpose-matrix } A) \ n = \text{transpose-matrix } (\text{take-rows } A \ n)$

```

apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
by simp

```

instantiation *matrix* :: ($\{\text{zero}, \text{ord}\}$) *ord*
begin

definition

le-matrix-def: $A \leq B \iff (\forall j \ i. \text{Rep-matrix } A \ j \ i \leq \text{Rep-matrix } B \ j \ i)$

definition

less-def: $A < (B::'a \text{ matrix}) \iff A \leq B \wedge \neg B \leq A$

instance ..

end

```

instance matrix :: ({zero, order}) 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)
apply simp
done

```

```

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

```

```

lemma le-combine-matrix:
  assumes
    f 0 0 = 0
     $\forall a b c d. a \leq b \ \& \ c \leq d \longrightarrow f a c \leq f b d$ 
    A <= B
    C <= D
  shows
    combine-matrix f A C <= combine-matrix f B D
  using assms by (simp add: le-matrix-def)

```

```

lemma le-left-combine-matrix:
  assumes
    f 0 0 = 0
     $\forall a b c. a \leq b \longrightarrow f c a \leq f c b$ 
    A <= B
  shows
    combine-matrix f C A <= combine-matrix f C B
  using assms by (simp add: le-matrix-def)

```

```

lemma le-right-combine-matrix:
  assumes
    f 0 0 = 0
     $\forall a b c. a \leq b \longrightarrow f a c \leq f b c$ 
    A <= B
  shows
    combine-matrix f A C <= combine-matrix f B C

```

using *assms* **by** (*simp add: le-matrix-def*)

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

lemma *le-foldseq*:

assumes

$\forall a b c d. a \leq b \ \& \ c \leq d \longrightarrow f a c \leq f b d$

$\forall i. i \leq n \longrightarrow s i \leq t i$

shows

$\text{foldseq } f s n \leq \text{foldseq } f t n$

proof -

have $\forall s t. (\forall i. i \leq n \longrightarrow s i \leq t i) \longrightarrow \text{foldseq } f s n \leq \text{foldseq } f t n$

by (*induct n*) (*simp-all add: assms*)

then show $\text{foldseq } f s n \leq \text{foldseq } f t n$ **using** *assms* **by** *simp*

qed

lemma *le-left-mult*:

assumes

$\forall a b c d. a \leq b \ \& \ c \leq d \longrightarrow \text{fadd } a c \leq \text{fadd } b d$

$\forall c a b. 0 \leq c \ \& \ a \leq b \longrightarrow \text{fmul } c a \leq \text{fmul } c b$

$\forall a. \text{fmul } 0 a = 0$

$\forall a. \text{fmul } a 0 = 0$

$\text{fadd } 0 0 = 0$

$0 \leq C$

$A \leq B$

shows

$\text{mult-matrix } \text{fmul } \text{fadd } C A \leq \text{mult-matrix } \text{fmul } \text{fadd } C B$

using *assms*

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*)

apply (*auto*)

done

lemma *le-right-mult*:

assumes

$\forall a b c d. a \leq b \ \& \ c \leq d \longrightarrow \text{fadd } a c \leq \text{fadd } b d$

$\forall c a b. 0 \leq c \ \& \ a \leq b \longrightarrow \text{fmul } a c \leq \text{fmul } b c$

$\forall a. \text{fmul } 0 a = 0$

$\forall a. \text{fmul } a 0 = 0$

$\text{fadd } 0 0 = 0$

$0 \leq C$

$A \leq B$

shows

$\text{mult-matrix } \text{fmul } \text{fadd } A C \leq \text{mult-matrix } \text{fmul } \text{fadd } B C$

```

using assms
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)
apply (auto)
done

```

```

lemma spec2:  $\forall j i. P j i \implies P j i$  by blast
lemma neg-imp:  $(\neg Q \longrightarrow \neg P) \implies P \longrightarrow Q$  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]:  $0 <= j \implies 0 <= i \implies (\text{move-matrix } A j i <= 0) = (A <= (0::('a::{order,zero}) \text{ matrix}))$ 
apply (auto simp add: le-matrix-def)
apply (drule-tac j=ja+(nat j) and i=ia+(nat i) in spec2)
apply (auto)
done

```

```

lemma move-matrix-zero-le[simp]:  $0 <= j \implies 0 <= i \implies (0 <= \text{move-matrix } A j i) = ((0::('a::{order,zero}) \text{ matrix}) <= A)$ 
apply (auto simp add: le-matrix-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]:  $0 <= j \implies 0 <= i \implies (\text{move-matrix } A j i <= \text{move-matrix } B j i) = (A <= (B::('a::{order,zero}) \text{ matrix}))$ 

```



```

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

instantiation matrix :: ({lattice, zero}) lattice
begin

definition inf = combine-matrix inf

definition sup = combine-matrix sup

instance
  by standard (auto simp add: le-infI le-matrix-def inf-matrix-def sup-matrix-def)

end

instantiation matrix :: ({plus, zero}) plus
begin

definition
  plus-matrix-def:  $A + B = \text{combine-matrix } (+) A B$ 

instance ..

end

instantiation matrix :: ({uminus, zero}) uminus
begin

definition
  minus-matrix-def:  $- A = \text{apply-matrix } \text{uminus } A$ 

instance ..

end

instantiation matrix :: ({minus, zero}) minus
begin

definition
  diff-matrix-def:  $A - B = \text{combine-matrix } (-) A B$ 

instance ..

end

instantiation matrix :: ({plus, times, zero}) times
begin

```

definition

times-matrix-def: $A * B = \text{mult-matrix } ((*) \text{ } (+) \text{ } A \text{ } B$

instance ..

end

instantiation *matrix* :: (*lattice*, *uminus*, *zero*) *abs*

begin

definition

abs-matrix-def: $|A :: 'a \text{ matrix}| = \text{sup } A \text{ } (- \text{ } A)$

instance ..

end

instance *matrix* :: (*monoid-add*) *monoid-add*

proof

fix *A B C* :: *'a 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 $0 + A = A$

apply (*simp add: plus-matrix-def*)

apply (*rule combine-matrix-zero-l-neutral*[*simplified zero-l-neutral-def*, *THEN spec*])

apply (*simp*)

done

show $A + 0 = A$

apply (*simp add: plus-matrix-def*)

apply (*rule combine-matrix-zero-r-neutral* [*simplified zero-r-neutral-def*, *THEN spec*])

apply (*simp*)

done

qed

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

proof

fix *A B* :: *'a matrix*

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
qed

```

```

instance matrix :: (group-add) group-add
proof
  fix A B :: 'a matrix
  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)
qed

```

```

instance matrix :: (ab-group-add) ab-group-add
proof
  fix A B :: 'a matrix
  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)
qed

```

```

instance matrix :: (ordered-ab-group-add) ordered-ab-group-add
proof
  fix A B C :: 'a matrix
  assume  $A \leq B$ 
  then show  $C + A \leq C + B$ 
    apply (simp add: plus-matrix-def)
    apply (rule le-left-combine-matrix)
    apply (simp-all)
  done
qed

```

```

instance matrix :: (lattice-ab-group-add) semilattice-inf-ab-group-add ..
instance matrix :: (lattice-ab-group-add) semilattice-sup-ab-group-add ..

```

```

instance matrix :: (semiring-0) semiring-0
proof
  fix A B C :: 'a 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 algebra-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 algebra-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 algebra-simps)
done
show  $0 * A = 0$  by (simp add: times-matrix-def)
show  $A * 0 = 0$  by (simp add: times-matrix-def)
qed

```

```

instance matrix :: (ring) ring ..

```

```

instance matrix :: (ordered-ring) ordered-ring
proof
fix  $A B C :: 'a$  matrix
assume  $a: A \leq B$ 
assume  $b: 0 \leq C$ 
from  $a b$  show  $C * A \leq C * B$ 
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  $A * C \leq B * C$ 
apply (simp add: times-matrix-def)
apply (rule le-right-mult)
apply (simp-all add: add-mono mult-right-mono)
done
qed

```

```

instance matrix :: (lattice-ring) lattice-ring
proof
fix  $A B C :: ('a :: lattice-ring)$  matrix
show  $|A| = \sup A (-A)$ 
by (simp add: abs-matrix-def)
qed

```

```

lemma Rep-matrix-add[simp]:
  Rep-matrix ((a::('a::monoid-add)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::semiring-0) matrix) * *b*) *j i* =
foldseq (+) (% *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*: $\forall x y. f (x+y) = (f x) + (f y) \implies f 0 = (0::'a)$
 $\implies \text{apply-matrix } f ((a::('a::monoid-add) \text{ matrix}) + b) = (\text{apply-matrix } f a) +$
 $(\text{apply-matrix } f b)$
apply (subst *Rep-matrix-inject*[symmetric])
apply (rule ext)+
apply (simp)
done

lemma *singleton-matrix-add*: *singleton-matrix* *j i* ((*a*::monoid-add)+*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::semiring-0) matrix) * *B*) <= *nrows A*
by (simp add: times-matrix-def mult-nrows)

lemma *ncols-mult*: *ncols* ((*A*::('a::semiring-0) matrix) * *B*) <= *ncols B*
by (simp add: times-matrix-def mult-ncols)

definition
one-matrix :: nat \Rightarrow ('a::{zero,one}) matrix **where**
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 (simpsubst *RepAbs-matrix*)
apply (rule exI[of - *n*], simp add: if-split)+
by (simp add: if-split)

lemma *nrows-one-matrix*[simp]: *nrows* ((*one-matrix* *n*) :: ('a::zero-neq-one) 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]: $ncols ((one\text{-}matrix\ n) :: ('a::zero\text{-}neq\text{-}one)matrix) = n$ (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]: $ncols\ A <= n \implies (A::('a::\{semiring-1\})\ matrix) * (one\text{-}matrix\ n) = A$

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: ncols)

lemma *one-matrix-mult-left*[simp]: $nrows\ A <= n \implies (one\text{-}matrix\ n) * A = (A::('a::semiring-1)\ matrix)$

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: nrows)

lemma *transpose-matrix-mult*: $transpose\text{-}matrix\ ((A::('a::comm\text{-}ring)\ matrix)*B) = (transpose\text{-}matrix\ B) * (transpose\text{-}matrix\ A)$

apply (simp add: times-matrix-def)
apply (subst transpose-mult-matrix)
apply (simp-all add: mult.commute)
done

lemma *transpose-matrix-add*: $transpose\text{-}matrix\ ((A::('a::monoid\text{-}add)\ matrix)+B) = transpose\text{-}matrix\ A + transpose\text{-}matrix\ B$

by (simp add: plus-matrix-def transpose-combine-matrix)

lemma *transpose-matrix-diff*: $transpose\text{-}matrix\ ((A::('a::group\text{-}add)\ matrix)-B) = transpose\text{-}matrix\ A - transpose\text{-}matrix\ B$

by (simp add: diff-matrix-def transpose-combine-matrix)

lemma *transpose-matrix-minus*: $transpose\text{-}matrix\ (-(A::('a::group\text{-}add)\ matrix)) = -\ transpose\text{-}matrix\ (A::'a\ matrix)$

by (simp add: minus-matrix-def transpose-apply-matrix)

definition *right-inverse-matrix* :: $('a::\{ring-1\})\ matrix \Rightarrow 'a\ matrix \Rightarrow bool$ **where**
 right-inverse-matrix A X == $(A * X = one\text{-}matrix\ (max\ (nrows\ A)\ (ncols\ X)))$
 $\wedge\ nrows\ X \leq\ ncols\ A$

definition *left-inverse-matrix* :: ('a::{ring-1}) matrix \Rightarrow 'a matrix \Rightarrow bool **where**
left-inverse-matrix A X == (X * A = one-matrix (max(nrows X) (ncols A))) \wedge
ncols X \leq nrows A

definition *inverse-matrix* :: ('a::{ring-1}) matrix \Rightarrow 'a matrix \Rightarrow bool **where**
inverse-matrix A X == (right-inverse-matrix A X) \wedge (left-inverse-matrix A X)

lemma *right-inverse-matrix-dim*: right-inverse-matrix A X \implies 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 \implies 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)
using assms
apply (simp-all add: left-inverse-matrix-def)
done

also have ... = Y * (A * X)
apply (insert assms)
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 assms)
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*: \llbracket inverse-matrix A X; inverse-matrix A Y $\rrbracket \implies$ 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::semiring-0) = 0 \mid b = 0 \implies a * b = 0

by auto

lemma *Rep-matrix-zero-imp-mult-zero*:

$\forall j \ i \ k. (\text{Rep-matrix } A \ j \ k = 0) \mid (\text{Rep-matrix } B \ k \ i) = 0 \implies A * B =$
 $(0::('a::\text{lattice-ring}) \text{ matrix})$
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::monoid-add) matrix) <= u \implies nrows B <= u*
 $\implies \text{nrows } (A + B) <= u$
apply (*simp add: plus-matrix-def*)
apply (*rule combine-nrows*)
apply (*simp-all*)
done

lemma *move-matrix-row-mult: move-matrix ((A::('a::semiring-0) matrix) * B) j*
 $0 = (\text{move-matrix } A \ j \ 0) * B$
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::semiring-0) matrix) * B) 0 i*
 $= A * (\text{move-matrix } B \ 0 \ i)$
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::monoid-add) matrix))*
 $= (\text{move-matrix } A \ j \ i) + (\text{move-matrix } B \ j \ i)$
apply (*subst Rep-matrix-inject[symmetric]*)
apply (*rule ext*)
apply (*simp*)
done

lemma *move-matrix-mult: move-matrix ((A::('a::semiring-0) matrix)*B) j i =*
 $(\text{move-matrix } A \ j \ 0) * (\text{move-matrix } B \ 0 \ i)$
by (*simp add: move-matrix-ortho[of A*B] move-matrix-col-mult move-matrix-row-mult*)

definition *scalar-mult* :: ('a::ring) ⇒ 'a matrix ⇒ 'a matrix **where**
scalar-mult a m == apply-matrix ((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 algebra-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:::group-add)) x y = -(Rep-matrix A x y)*
by (*simp add: minus-matrix-def*)

lemma *Rep-abs[simp]*: *Rep-matrix |A:::lattice-ab-group-add| x y = |Rep-matrix A x y|*
by (*simp add: abs-lattice sup-matrix-def*)

end

theory *SparseMatrix*
imports *Matrix*
begin

type-synonym 'a *svec* = (nat * 'a) list
type-synonym 'a *spmat* = 'a *svec svec*

definition *sparse-row-vector* :: ('a::ab-group-add) *svec* ⇒ 'a matrix
where *sparse-row-vector arr = foldl (% m x. m + (singleton-matrix 0 (fst x) (snd x))) 0 arr*

definition *sparse-row-matrix* :: ('a::ab-group-add) *spmat* ⇒ 'a matrix
where *sparse-row-matrix arr = foldl (% m r. m + (move-matrix (sparse-row-vector (snd r)) (int (fst r)) 0)) 0 arr*

code-datatype *sparse-row-vector sparse-row-matrix*

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*)

lemmas [code] = *sparse-row-vector-empty* [symmetric]

lemma *foldl-distrstart*: $\forall a x y. (f (g x y) a = g x (f y a)) \implies (foldl f (g x y) l = g x (foldl f y l))$
by (*induct l arbitrary: x y, 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 $\lambda m x. m + \text{singleton-matrix } 0 \text{ (fst } x \text{) (snd } x \text{) } \lambda x m. \text{singleton-matrix } 0 \text{ (fst } x \text{) (snd } x \text{) } + m]$)*)
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 $\lambda m x. m + (\text{move-matrix } (\text{sparse-row-vector } (\text{snd } x)) (\text{int } (\text{fst } x)) 0)$)*)
 $\% a m. (\text{move-matrix } (\text{sparse-row-vector } (\text{snd } a)) (\text{int } (\text{fst } a)) 0) + m]$
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

primrec *sorted-spvec* :: 'a spvec \Rightarrow bool

where

sorted-spvec [] = True

| *sorted-spvec-step*: *sorted-spvec* (a#as) = (case as of [] \Rightarrow True | b#bs \Rightarrow ((fst a

$< \text{fst } b) \ \& \ (\text{sorted-spvec } as))$

primrec *sorted-spmat* :: 'a *spmat* \Rightarrow *bool*

where

sorted-spmat [] = *True*

| *sorted-spmat* (a#as) = ((*sorted-spvec* (snd a)) & (*sorted-spmat* 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) \Longrightarrow *sorted-spvec* as

apply (*induct as*)

apply (*auto simp add: sorted-spvec.simps*)

done

lemma *sorted-spvec-cons2*: *sorted-spvec* (a#b#t) \Longrightarrow *sorted-spvec* (a#t)

apply (*induct t*)

apply (*auto simp add: sorted-spvec.simps*)

done

lemma *sorted-spvec-cons3*: *sorted-spvec*(a#b#t) \Longrightarrow *fst a* < *fst b*

apply (*auto simp add: sorted-spvec.simps*)

done

lemma *sorted-sparse-row-vector-zero*[*rule-format*]: $m \leq n \Longrightarrow$ *sorted-spvec* ((n,a)#arr)

\longrightarrow *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*]: $m \leq n \Longrightarrow$ *sorted-spvec* ((n,a)#arr)

\longrightarrow *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*)

done

primrec *minus-spvec* :: ('a::*ab-group-add*) *spvec* \Rightarrow 'a *spvec*

where

minus-spvec [] = []

| *minus-spvec* (a#as) = (*fst a*, -(*snd a*))#(*minus-spvec* as)

primrec *abs-spvec* :: ('a::*lattice-ab-group-add-abs*) *spvec* \Rightarrow 'a *spvec*

where

$abs\text{-}spvec \ [] = []$
 $| abs\text{-}spvec (a\#as) = (fst\ a, |snd\ a|)\#(abs\text{-}spvec\ as)$

lemma *sparse-row-vector-minus*:

$sparse\text{-}row\text{-}vector\ (minus\text{-}spvec\ v) = -\ (sparse\text{-}row\text{-}vector\ v)$
apply (*induct v*)
apply (*simp-all add: sparse-row-vector-cons*)
apply (*simp add: Rep-matrix-inject[symmetric]*)
apply (*rule ext*)
apply *simp*
done

instance *matrix* :: (*lattice-ab-group-add-abs*) *lattice-ab-group-add-abs*

apply *standard*
unfolding *abs-matrix-def*
apply *rule*
done

lemma *sparse-row-vector-abs*:

$sorted\text{-}spvec\ (v :: 'a::lattice\text{-}ring\ spvec) \implies sparse\text{-}row\text{-}vector\ (abs\text{-}spvec\ v) =$
 $|sparse\text{-}row\text{-}vector\ v|$
apply (*induct v*)
apply *simp-all*
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*:

$sorted\text{-}spvec\ v \implies sorted\text{-}spvec\ (minus\text{-}spvec\ v)$
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*:

$sorted\text{-}spvec\ v \implies sorted\text{-}spvec\ (abs\text{-}spvec\ v)$
apply (*induct v*)
apply (*simp*)
apply (*frule sorted-spvec-cons1, simp*)
apply (*simp add: sorted-spvec.simps split:list.split-asm*)

done

definition *smult-spvec* $y = \text{map } (\% a. (\text{fst } a, y * \text{snd } a))$

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

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

fun *addmult-spvec* :: ('a::ring) \Rightarrow 'a *spvec* \Rightarrow 'a *spvec* \Rightarrow 'a *spvec*
where
addmult-spvec $y \ \text{arr} \ [] = \text{arr}$
| *addmult-spvec* $y \ [] \ \text{brr} = \text{smult-spvec } y \ \text{brr}$
| *addmult-spvec* $y \ ((i,a)\#\text{arr}) \ ((j,b)\#\text{brr}) =$
 if $i < j$ *then* $((i,a)\#(\text{addmult-spvec } y \ \text{arr} \ ((j,b)\#\text{brr})))$
 else (*if* $(j < i)$ *then* $((j, y * b)\#(\text{addmult-spvec } y \ ((i,a)\#\text{arr}) \ \text{brr}))$
 else $((i, a + y*b)\#(\text{addmult-spvec } y \ \text{arr} \ \text{brr})))$

lemma *addmult-spvec-empty1[simp]*: *addmult-spvec* $y \ [] \ a = \text{smult-spvec } y \ a$
by (*induct a*) *auto*

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

lemma *sparse-row-vector-map*: $(\forall x y. f (x+y) = (f x) + (f y)) \Longrightarrow (f :: 'a \Rightarrow ('a::\text{lattice-ring}))$
 $0 = 0 \Longrightarrow$
sparse-row-vector $(\text{map } (\% x. (\text{fst } x, f (\text{snd } x))) \ a) = \text{apply-matrix } f \ (\text{sparse-row-vector } a)$
apply (*induct a*)
apply (*simp-all add: apply-matrix-add*)
done

lemma *sparse-row-vector-smult*: *sparse-row-vector* $(\text{smult-spvec } y \ a) = \text{scalar-mult } y \ (\text{sparse-row-vector } a)$
apply (*induct a*)
apply (*simp-all add: smult-spvec-cons scalar-mult-add*)
done

lemma *sparse-row-vector-addmult-spvec*: *sparse-row-vector* $(\text{addmult-spvec } (y :: 'a::\text{lattice-ring}) \ a \ b) =$
 $(\text{sparse-row-vector } a) + (\text{scalar-mult } y \ (\text{sparse-row-vector } b))$
apply (*induct y a b rule: addmult-spvec.induct*)
apply (*simp add: scalar-mult-add smult-spvec-cons sparse-row-vector-smult singleton-matrix-add*)
done

```

lemma sorted-smult-spvec: sorted-spvec a  $\implies$  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-admult-spvec-helper:  $\llbracket$ sorted-spvec (admult-spvec y ((a, b) # arr) brr); aa < a; sorted-spvec ((a, b) # arr); sorted-spvec ((aa, ba) # brr) $\rrbracket \implies$  sorted-spvec ((aa, y * ba) # admult-spvec y ((a, b) # arr) brr)
  apply (induct brr)
  apply (auto simp add: sorted-spvec.simps)
  done

lemma sorted-spvec-admult-spvec-helper2:
   $\llbracket$ sorted-spvec (admult-spvec y arr ((aa, ba) # brr)); a < aa; sorted-spvec ((a, b) # arr); sorted-spvec ((aa, ba) # brr) $\rrbracket$ 
   $\implies$  sorted-spvec ((a, b) # admult-spvec y arr ((aa, ba) # brr))
  apply (induct arr)
  apply (auto simp add: smult-spvec-def sorted-spvec.simps)
  done

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

lemma sorted-admult-spvec: sorted-spvec a  $\implies$  sorted-spvec b  $\implies$  sorted-spvec (admult-spvec y a b)
  apply (induct y a b rule: admult-spvec.induct)
  apply (simp-all add: sorted-smult-spvec)
  apply (rule conjI, intro strip)
  apply (case-tac  $\sim(i < j)$ )
  apply (simp-all)
  apply (frule-tac as=brr in sorted-spvec-cons1)
  apply (simp add: sorted-spvec-admult-spvec-helper)
  apply (intro strip | rule conjI)+
  apply (frule-tac as=arr in sorted-spvec-cons1)
  apply (simp add: sorted-spvec-admult-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-admult-spvec-helper3)
  done

```

```

fun mult-spvec-spmat :: ('a::lattice-ring) spvec ⇒ 'a spvec ⇒ 'a spmat ⇒ 'a spvec
where
  mult-spvec-spmat c [] brr = c
| mult-spvec-spmat c arr [] = c
| mult-spvec-spmat c ((i,a)#arr) ((j,b)#brr) = (
  if (i < j) then mult-spvec-spmat c arr ((j,b)#brr)
  else if (j < i) then mult-spvec-spmat c ((i,a)#arr) brr
  else mult-spvec-spmat (addmult-spvec a c b) arr brr)

lemma sparse-row-mult-spvec-spmat[rule-format]: sorted-spvec (a:('a::lattice-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 (induct c a B 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: algebra-simps sparse-row-matrix-cons)
  apply (simpsubst Rep-matrix-zero-imp-mult-zero)
  apply (simp)
  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 <= j)
  apply (rule-tac m1 = k and n1 = i and a1 = a in ssubst[OF sorted-sparse-row-vector-zero])
  apply (simp-all)
  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: algebra-simps)
apply (subst Rep-matrix-zero-imp-mult-zero)
apply (simp)
apply (rule disjI2)
apply (intro strip)
apply (simp add: sparse-row-matrix-cons)
apply (case-tac i <= j)
apply (erule sorted-sparse-row-matrix-zero)
apply (simp-all)
apply (intro strip)
apply (case-tac i=j)
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 algebra-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 = j and a = a and arr=arr in sorted-sparse-row-vector-zero)
apply (simp-all)
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=j 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::lattice-ring) *spvec*) \longrightarrow *sorted-spmat* *B* \longrightarrow *sorted-spvec* (*mult-spvec-spmat*
 c a B)
 apply (*induct c a B rule: mult-spvec-spmat.induct*)
 apply (*simp-all add: sorted-addmult-spvec*)
 done

primrec *mult-spmat* :: ('a::lattice-ring) *spmat* \Rightarrow 'a *spmat* \Rightarrow 'a *spmat*
where
 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*:
 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 alge-*
 bra-simps move-matrix-mult)
 done

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

fun *add-spvec* :: ('a::lattice-ab-group-add) *spvec* \Rightarrow 'a *spvec* \Rightarrow 'a *spvec*
where

add-spvec *arr* [] = *arr*
| *add-spvec* [] *brr* = *brr*
| *add-spvec* ((*i,a*)#*arr*) ((*j,b*)#*brr*) = (
 if i < j then (i,a)#(add-spvec arr ((j,b)#brr))
 else if (j < i) then (j,b) # add-spvec ((i,a)#arr) brr
 else (i, a+b) # add-spvec arr brr)

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

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

fun *add-spmat* :: ('*a*::*lattice-ab-group-add*) *spmat* \Rightarrow '*a* *spmat* \Rightarrow '*a* *spmat*
where

```

  add-spmat [] bs = bs
| add-spmat as [] = as
| add-spmat ((i,a)#as) ((j,b)#bs) = (
  if i < j then
    (i,a) # add-spmat as ((j,b)#bs)
  else if j < i then
    (j,b) # add-spmat ((i,a)#as) bs
  else
    (i, add-spvec a b) # add-spmat as bs)

```

lemma *add-spmat-Nil2*[simp]: *add-spmat as* [] = *as*
by(*cases as*) *auto*

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

lemmas [*code*] = *sparse-row-add-spmat* [*symmetric*]
lemmas [*code*] = *sparse-row-vector-add* [*symmetric*]

lemma *sorted-add-spvec-helper1*[*rule-format*]: *add-spvec* ((*a,b*)#*arr*) *brr* = (*ab*, *bb*) # *list* \longrightarrow (*ab* = *a* | (*brr* \neq [] & *ab* = *fst* (*hd brr*)))
proof –
have ($\forall x$ *ab a. x* = (*a,b*)#*arr* \longrightarrow *add-spvec x brr* = (*ab*, *bb*) # *list* \longrightarrow (*ab* = *a* | (*ab* = *fst* (*hd brr*))))
by (*induct brr rule: add-spvec.induct*) (*auto split:if-splits*)
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* \longrightarrow (*ab* = *a* | (*brr* \neq [] & *ab* = *fst* (*hd brr*)))
proof –
have ($\forall x$ *ab a. x* = (*a,b*)#*arr* \longrightarrow *add-spmat x brr* = (*ab*, *bb*) # *list* \longrightarrow (*ab*

```

= a | (ab = fst (hd brr)))
  by (rule add-spmat.induct) (auto split:if-splits)
  then show ?thesis
  by (case-tac brr, auto)
qed

```

```

lemma sorted-add-spvec-helper: add-spvec arr brr = (ab, bb) # list  $\implies$  ((arr  $\neq$ 
[] & ab = fst (hd arr)) | (brr  $\neq$  [] & ab = fst (hd brr)))
  apply (induct arr brr rule: add-spvec.induct)
  apply (auto split:if-splits)
  done

```

```

lemma sorted-add-spmat-helper: add-spmat arr brr = (ab, bb) # list  $\implies$  ((arr  $\neq$ 
[] & ab = fst (hd arr)) | (brr  $\neq$  [] & ab = fst (hd brr)))
  apply (induct arr brr rule: add-spmat.induct)
  apply (auto split:if-splits)
  done

```

```

lemma add-spvec-commute: add-spvec a b = add-spvec b a
  by (induct a b rule: add-spvec.induct) auto

```

```

lemma add-spmat-commute: add-spmat a b = add-spmat b a
  apply (induct a b rule: add-spmat.induct)
  apply (simp-all add: add-spvec-commute)
  done

```

```

lemma sorted-add-spvec-helper2: add-spvec ((a,b)#arr) brr = (ab, bb) # list  $\implies$ 
aa < a  $\implies$  sorted-spvec ((aa, ba) # brr)  $\implies$  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  $\implies$ 
aa < a  $\implies$  sorted-spvec ((aa, ba) # brr)  $\implies$  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 (induct a b rule: add-spvec.induct)

```

```

apply (simp-all)
apply (rule conjI)
apply (clarsimp)
apply (frule-tac as=brr in sorted-svvec-cons1)
apply (simp)
apply (subst sorted-svvec-step)
apply (clarsimp simp: sorted-add-svvec-helper2 split: list.split)
apply (clarify)
apply (rule conjI)
apply (clarify)
apply (frule-tac as=arr in sorted-svvec-cons1, simp)
apply (subst sorted-svvec-step)
apply (clarsimp simp: sorted-add-svvec-helper2 add-svvec-commute split: list.split)
apply (clarify)
apply (frule-tac as=arr in sorted-svvec-cons1)
apply (frule-tac as=brr in sorted-svvec-cons1)
apply (simp)
apply (subst sorted-svvec-step)
apply (simp split: list.split)
apply (clarsimp)
apply (drule-tac sorted-add-svvec-helper)
apply (auto simp: neq-Nil-conv)
apply (drule sorted-svvec-cons3)
apply (simp)
apply (drule sorted-svvec-cons3)
apply (simp)
done

```

```

lemma sorted-svvec-add-spmat[rule-format]: sorted-svvec A  $\longrightarrow$  sorted-svvec B
 $\longrightarrow$  sorted-svvec (add-spmat A B)
apply (induct A B rule: add-spmat.induct)
apply (simp-all)
apply (rule conjI)
apply (intro strip)
apply (simp)
apply (frule-tac as=bs in sorted-svvec-cons1)
apply (simp)
apply (subst sorted-svvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (simp add: sorted-add-spmat-helper2)
apply (clarify)
apply (rule conjI)
apply (clarify)
apply (frule-tac as=as in sorted-svvec-cons1, simp)
apply (subst sorted-svvec-step)
apply (clarsimp simp: sorted-add-spmat-helper2 add-spmat-commute split: list.split)
apply (clarsimp)
apply (frule-tac as=as in sorted-svvec-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 simp: neq-Nil-conv)
apply (drule sorted-spvec-cons3)
apply (simp)
apply (drule sorted-spvec-cons3)
apply (simp)
done

```

```

lemma sorted-spmat-add-spmat[rule-format]: sorted-spmat A  $\implies$  sorted-spmat B
 $\implies$  sorted-spmat (add-spmat A B)
apply (induct A B rule: add-spmat.induct)
apply (simp-all add: sorted-spvec-add-spvec)
done

```

```

fun le-spvec :: ('a::lattice-ab-group-add) spvec  $\Rightarrow$  'a spvec  $\Rightarrow$  bool
where

```

```

  le-spvec [] [] = True
| le-spvec ((-,a)#as) [] = (a <= 0 & le-spvec as [])
| le-spvec [] ((-,b)#bs) = (0 <= b & le-spvec [] bs)
| le-spvec ((i,a)#as) ((j,b)#bs) = (
  if (i < j) then a <= 0 & le-spvec as ((j,b)#bs)
  else if (j < i) then 0 <= b & le-spvec ((i,a)#as) bs
  else a <= b & le-spvec as bs)

```

```

fun le-spmat :: ('a::lattice-ab-group-add) spmat  $\Rightarrow$  'a spmat  $\Rightarrow$  bool
where

```

```

  le-spmat [] [] = True
| le-spmat ((i,a)#as) [] = (le-spvec a [] & le-spmat as [])
| le-spmat [] ((j,b)#bs) = (le-spvec [] b & le-spmat [] bs)
| le-spmat ((i,a)#as) ((j,b)#bs) = (
  if i < j then (le-spvec a [] & le-spmat as ((j,b)#bs))
  else if j < i then (le-spvec [] b & le-spmat ((i,a)#as) bs)
  else (le-spvec a b & le-spmat as bs))

```

```

definition disj-matrices :: ('a::zero) matrix  $\Rightarrow$  'a matrix  $\Rightarrow$  bool where
  disj-matrices A B  $\longleftrightarrow$ 

```

```

  ( $\forall j i. (\text{Rep-matrix } A \ j \ i \neq 0) \longrightarrow (\text{Rep-matrix } B \ j \ i = 0)$ ) & ( $\forall j i. (\text{Rep-matrix } B \ j \ i \neq 0) \longrightarrow (\text{Rep-matrix } A \ j \ i = 0)$ )

```

```

declare [[simp-depth-limit = 6]]

```

```

lemma disj-matrices-contr1: disj-matrices A B  $\implies$  Rep-matrix A j i  $\neq$  0  $\implies$ 

```

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

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

lemma *disj-matrices-add*: *disj-matrices* A $B \implies$ *disj-matrices* C $D \implies$ *disj-matrices*
 A $D \implies$ *disj-matrices* B $C \implies$
 $(A + B \leq C + D) = (A \leq C \ \& \ B \leq (D::('a::lattice-ab-group-add) \text{matrix}))$
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 $B \implies$
 $(A + B \leq 0) = (A \leq 0 \ \& \ (B::('a::lattice-ab-group-add) \text{matrix}) \leq 0)$
by (*rule disj-matrices-add[of A B 0 0, simplified]*)

lemma *disj-matrices-add-zero-le*: *disj-matrices* A $B \implies$
 $(0 \leq A + B) = (0 \leq A \ \& \ 0 \leq (B::('a::lattice-ab-group-add) \text{matrix}))$
by (*rule disj-matrices-add[of 0 0 A B, simplified]*)

lemma *disj-matrices-add-x-le*: *disj-matrices* A $B \implies$ *disj-matrices* B $C \implies$
 $(A \leq B + C) = (A \leq C \ \& \ 0 \leq (B::('a::lattice-ab-group-add) \text{matrix}))$

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

lemma *disj-matrices-add-le-x*: $disj\text{-matrices } A B \implies disj\text{-matrices } B C \implies$
 $(B + A \leq C) = (A \leq C \ \& \ (B::('a::lattice\text{-ab}\text{-group}\text{-add})\text{ matrix}) \leq 0)$
by (*auto simp add: disj-matrices-add*[of $B A 0 C$, *simplified*] *disj-matrices-commute*)

lemma *disj-sparse-row-singleton*: $i \leq j \implies sorted\text{-spvec}((j,y)\#v) \implies disj\text{-matrices}$
 $(sparse\text{-row}\text{-vector } v) (singleton\text{-matrix } 0 i x)$
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*: $disj\text{-matrices } A B \implies disj\text{-matrices } A C \implies disj\text{-matrices}$
 $(A::('a::lattice\text{-ab}\text{-group}\text{-add})\text{ matrix}) (B+C)$
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*: $disj\text{-matrices } A B \implies disj\text{-matrices } A C \implies disj\text{-matrices}$
 $(B+C) (A::('a::lattice\text{-ab}\text{-group}\text{-add})\text{ matrix})$
by (*simp add: disj-matrices-x-add disj-matrices-commute*)

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

lemma *disj-move-sparse-vec-mat*[*simplified disj-matrices-commute*]:
 $j \leq a \implies sorted\text{-spvec}((a,c)\#as) \implies disj\text{-matrices } (move\text{-matrix } (sparse\text{-row}\text{-vector}$
 $b) (int j) i) (sparse\text{-row}\text{-matrix } as)$
apply (*auto simp add: 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:
   $j \neq u \implies \text{disj-matrices } (\text{move-matrix } (\text{sparse-row-vector } a) j i) (\text{move-matrix } (\text{sparse-row-vector } b) u v)$ 
apply (auto simp add: 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)  $\longrightarrow$  (sorted-spvec b)  $\longrightarrow$  (le-spvec a b) = (sparse-row-vector a <= sparse-row-vector b)
apply (induct a b 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=b, 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  $\longrightarrow$  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 (auto simp add: disj-matrices-commute disj-sparse-row-singleton[OF order-refl] sorted-spvec-cons1)

```



```

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 (auto simp add: disj-matrices-commute disj-sparse-row-singleton[OF or-
der-refl] sorted-spvec-cons1)
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 (induct A B 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 i=j)
  apply (simp-all)
  apply (subst disj-matrices-add)
  apply (simp-all add: disj-matrices-commute disj-move-sparse-vec-mat[OF or-
der-refl])
  apply (simp add: sorted-spvec-cons1 le-spvec-iff-sparse-row-le)
done

declare [[simp-depth-limit = 999]]

primrec abs-spmat :: ('a::lattice-ring) spmat ⇒ 'a spmat
where

```

$abs\text{-spmat } [] = []$
 $| abs\text{-spmat } (a\#as) = (fst\ a, abs\text{-spvec } (snd\ a))\#(abs\text{-spmat } as)$

primrec $minus\text{-spmat} :: ('a::lattice\text{-ring})\ spmat \Rightarrow 'a\ spmat$
where

$minus\text{-spmat } [] = []$
 $| minus\text{-spmat } (a\#as) = (fst\ a, minus\text{-spvec } (snd\ a))\#(minus\text{-spmat } as)$

lemma $sparse\text{-row-matrix-minus}$:

$sparse\text{-row-matrix } (minus\text{-spmat } A) = - (sparse\text{-row-matrix } A)$
apply $(induct\ A)$
apply $(simp\text{-all } add:\ sparse\text{-row-vector-minus } sparse\text{-row-matrix-cons})$
apply $(subst\ Rep\text{-matrix-inject}[symmetric])$
apply $(rule\ ext)+$
apply $simp$
done

lemma $Rep\text{-sparse-row-vector-zero}$: $x \neq 0 \implies Rep\text{-matrix } (sparse\text{-row-vector } v) x = 0$

proof $-$

assume $x:x \neq 0$
have $r:nrows (sparse\text{-row-vector } v) \leq Suc\ 0$ **by** $(rule\ nrows\text{-spvec})$
show $?thesis$
apply $(rule\ nrows)$
apply $(subgoal\ tac\ Suc\ 0 \leq x)$
apply $(insert\ r)$
apply $(simp\ only:)$
apply $(insert\ x)$
apply $arith$
done

qed

lemma $sparse\text{-row-matrix-abs}$:

$sorted\text{-spvec } A \implies sorted\text{-spmat } A \implies sparse\text{-row-matrix } (abs\text{-spmat } A) = |sparse\text{-row-matrix } A|$

apply $(induct\ A)$
apply $(simp\text{-all } add:\ sparse\text{-row-vector-abs } sparse\text{-row-matrix-cons})$
apply $(frule\ tac\ sorted\text{-spvec-cons1}, simp)$
apply $(simplesubst\ Rep\text{-matrix-inject}[symmetric])$
apply $(rule\ ext)+$
apply $auto$
apply $(case\ tac\ x=a)$
apply $(simp)$
apply $(simplesubst\ sorted\text{-sparse-row-matrix-zero})$
apply $auto$
apply $(simplesubst\ Rep\text{-sparse-row-vector-zero})$
apply $simp\text{-all}$
done

lemma *sorted-spvec-minus-spmat*: $\text{sorted-spvec } A \implies \text{sorted-spvec } (\text{minus-spmat } A)$

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*: $\text{sorted-spvec } A \implies \text{sorted-spvec } (\text{abs-spmat } A)$

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*: $\text{sorted-spmat } A \implies \text{sorted-spmat } (\text{minus-spmat } A)$

apply (*induct* *A*)
apply (*simp-all* *add*: *sorted-spvec-minus-spvec*)
done

lemma *sorted-spmat-abs-spmat*: $\text{sorted-spmat } A \implies \text{sorted-spmat } (\text{abs-spmat } A)$

apply (*induct* *A*)
apply (*simp-all* *add*: *sorted-spvec-abs-spvec*)
done

definition *diff-spmat* :: (*'a*::*lattice-ring*) *spmat* \Rightarrow *'a* *spmat* \Rightarrow *'a* *spmat*
where *diff-spmat* *A B* = *add-spmat* *A* (*minus-spmat* *B*)

lemma *sorted-spmat-diff-spmat*: $\text{sorted-spmat } A \implies \text{sorted-spmat } B \implies \text{sorted-spmat } (\text{diff-spmat } A B)$

by (*simp* *add*: *diff-spmat-def sorted-spmat-minus-spmat sorted-spmat-add-spmat*)

lemma *sorted-spvec-diff-spmat*: $\text{sorted-spvec } A \implies \text{sorted-spvec } B \implies \text{sorted-spvec } (\text{diff-spmat } A B)$

by (*simp* *add*: *diff-spmat-def sorted-spvec-minus-spmat sorted-spvec-add-spmat*)

lemma *sparse-row-diff-spmat*: $\text{sparse-row-matrix } (\text{diff-spmat } A B) = (\text{sparse-row-matrix } A) - (\text{sparse-row-matrix } B)$

by (*simp* *add*: *diff-spmat-def sparse-row-add-spmat sparse-row-matrix-minus*)

definition *sorted-sparse-matrix* :: *'a* *spmat* \Rightarrow *bool*

where *sorted-sparse-matrix* *A* \longleftrightarrow *sorted-spvec* *A* & *sorted-spmat* *A*

lemma *sorted-sparse-matrix-imp-spvec*: $\text{sorted-sparse-matrix } A \implies \text{sorted-spvec } A$

by (*simp* *add*: *sorted-sparse-matrix-def*)

lemma *sorted-sparse-matrix-imp-spmat*: $\text{sorted-sparse-matrix } A \implies \text{sorted-spmat } A$

```

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

primrec pprt-spvec :: ('a::{lattice-ab-group-add}) spvec  $\Rightarrow$  'a spvec
where
  pprt-spvec [] = []
| pprt-spvec (a#as) = (fst a, pprt (snd a)) # (pprt-spvec as)

primrec nprt-spvec :: ('a::{lattice-ab-group-add}) spvec  $\Rightarrow$  'a spvec
where
  nprt-spvec [] = []
| nprt-spvec (a#as) = (fst a, nprt (snd a)) # (nprt-spvec as)

primrec pprt-spmat :: ('a::{lattice-ab-group-add}) spmat  $\Rightarrow$  'a spmat
where
  pprt-spmat [] = []
| pprt-spmat (a#as) = (fst a, pprt-spvec (snd a)) # (pprt-spmat as)

primrec nprt-spmat :: ('a::{lattice-ab-group-add}) spmat  $\Rightarrow$  'a spmat
where
  nprt-spmat [] = []
| nprt-spmat (a#as) = (fst a, nprt-spvec (snd a)) # (nprt-spmat as)

lemma pprt-add: disj-matrices A (B::(-::lattice-ring) matrix)  $\implies$  pprt (A+B) =
  pprt A + pprt B
  apply (simp add: pprt-def sup-matrix-def)
  apply (simp add: Rep-matrix-inject[symmetric])
  apply (rule ext)+
  apply simp

```

```

apply (case-tac Rep-matrix A x xa ≠ 0)
apply (simp-all add: disj-matrices-contr1)
done

```

```

lemma nprt-add: disj-matrices A (B::(-::lattice-ring) matrix)  $\implies$  nprt (A+B) =
nprt A + nprt B
apply (simp add: nprt-def inf-matrix-def)
apply (simp add: Rep-matrix-inject[symmetric])
apply (rule ext)+
apply simp
apply (case-tac Rep-matrix A x xa ≠ 0)
apply (simp-all add: disj-matrices-contr1)
done

```

```

lemma ppert-singleton[simp]: ppert (singleton-matrix j i (x::(-::lattice-ring))) = sin-
gleton-matrix j i (ppert x)
apply (simp add: ppert-def sup-matrix-def)
apply (simp add: Rep-matrix-inject[symmetric])
apply (rule ext)+
apply simp
done

```

```

lemma npert-singleton[simp]: npert (singleton-matrix j i (x::(-::lattice-ring))) = sin-
gleton-matrix j i (npert x)
apply (simp add: npert-def inf-matrix-def)
apply (simp add: Rep-matrix-inject[symmetric])
apply (rule ext)+
apply simp
done

```

```

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

```

```

lemma sparse-row-vector-ppert: sorted-spvec (v :: 'a::lattice-ring spvec)  $\implies$  sparse-row-vector
(ppert-spvec v) = ppert (sparse-row-vector v)
apply (induct v)
apply (simp-all)
apply (frule sorted-spvec-cons1, auto)
apply (subst ppert-add)
apply (subst disj-matrices-commute)
apply (rule disj-sparse-row-singleton)
apply auto
done

```

```

lemma sparse-row-vector-npert: sorted-spvec (v :: 'a::lattice-ring spvec)  $\implies$  sparse-row-vector
(npert-spvec v) = npert (sparse-row-vector v)
apply (induct v)
apply (simp-all)
apply (frule sorted-spvec-cons1, auto)
apply (subst npert-add)

```

```

apply (subst disj-matrices-commute)
apply (rule disj-sparse-row-singleton)
apply auto
done

```

```

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

```

```

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

```

```

lemma sparse-row-matrix-pprt: sorted-spvec (m :: 'a::lattice-ring spmat)  $\implies$  sorted-spmat
m  $\implies$  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 :: 'a::lattice-ring spmat)  $\implies$  sorted-spmat
m  $\implies$  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  $\implies$  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  $\implies$  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  $\implies$  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  $\implies$  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  $\implies$  sorted-spmat (pprt-spmat m)
apply (induct m)
apply (simp-all add: sorted-pprt-spvec)
done

lemma sorted-spmat-nprt-spmat: sorted-spmat m  $\implies$  sorted-spmat (nprt-spmat m)

```

```

apply (induct m)
apply (simp-all add: sorted-nprt-spvec)
done

```

```

definition mult-est-spmat :: ('a::lattice-ring) spmat  $\Rightarrow$  'a spmat  $\Rightarrow$  'a spmat  $\Rightarrow$  'a
spmat  $\Rightarrow$  'a spmat where
  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

```

```

lemmas sparse-row-matrix-arith-simps =
  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

```

end

```

theory LP
imports Main HOL-Library.Lattice-Algebras
begin

```

```

lemma le-add-right-mono:
  assumes
     $a \leq b + (c::'a::ordered-ab-group-add)$ 

```



```

c <= d
shows a <= b + d
apply (rule-tac order-trans[where y = b+c])
apply (simp-all add: assms)
done

```

lemma *linprog-dual-estimate*:

```

assumes
  A * x ≤ (b::'a::lattice-ring)
  0 ≤ y
  |A - A'| ≤ δ-A
  b ≤ b'
  |c - c'| ≤ δ-c
  |x| ≤ r
shows
  c * x ≤ y * b' + (y * δ-A + |y * A' - c'| + δ-c) * r
proof -
  from assms have 1: y * b <= y * b' by (simp add: mult-left-mono)
  from assms have 2: y * (A * x) <= y * b by (simp add: mult-left-mono)
  have 3: y * (A * x) = c * x + (y * (A - A') + (y * A' - c') + (c' - c)) * x by
    (simp add: algebra-simps)
  from 1 2 3 have 4: c * x + (y * (A - A') + (y * A' - c') + (c' - c)) * x <=
    y * b' by simp
  have 5: c * x <= y * b' + |(y * (A - A') + (y * A' - c') + (c' - c)) * x|
    by (simp only: 4 estimate-by-abs)
  have 6: |(y * (A - A') + (y * A' - c') + (c' - c)) * x| <= |y * (A - A') + (y
    * A' - c') + (c' - c)| * |x|
    by (simp add: abs-le-mult)
  have 7: (|y * (A - A') + (y * A' - c') + (c' - c)|) * |x| <= (|y * (A - A') +
    (y * A' - c')| + |c' - c|) * |x|
    by (rule abs-triangle-ineq [THEN mult-right-mono]) simp
  have 8: (|y * (A - A') + (y * A' - c')| + |c' - c|) * |x| <= (|y * (A - A')| +
    |y * A' - c'| + |c' - c|) * |x|
    by (simp add: abs-triangle-ineq mult-right-mono)
  have 9: (|y * (A - A')| + |y * A' - c'| + |c' - c|) * |x| <= (|y| * |A - A'| + |y * A' - c'|
    + |c' - c|) * |x|
    by (simp add: abs-le-mult mult-right-mono)
  have 10: c' - c = -(c - c') by (simp add: algebra-simps)
  have 11: |c' - c| = |c - c'|
    by (subst 10, subst abs-minus-cancel, simp)
  have 12: (|y| * |A - A'| + |y * A' - c'| + |c' - c|) * |x| <= (|y| * |A - A'| + |y * A' - c'|
    + δ-c) * |x|
    by (simp add: 11 assms mult-right-mono)
  have 13: (|y| * |A - A'| + |y * A' - c'| + δ-c) * |x| <= (|y| * δ-A + |y * A' - c'| +
    δ-c) * |x|
    by (simp add: assms mult-right-mono mult-left-mono)
  have r: (|y| * δ-A + |y * A' - c'| + δ-c) * |x| <= (|y| * δ-A + |y * A' - c'| + δ-c) *
    r
    apply (rule mult-left-mono)

```

```

apply (simp add: assms)
apply (rule-tac add-mono[ $of\ 0::'a - 0$ , simplified]) +
apply (rule mult-left-mono[ $of\ 0\ \delta-A$ , simplified])
apply (simp-all)
apply (rule order-trans[where  $y=|A-A'|$ ], simp-all add: assms)
apply (rule order-trans[where  $y=|c-c'|$ ], simp-all add: assms)
done
from 6 7 8 9 12 13 r have 14:  $|(y * (A - A') + (y * A' - c') + (c' - c)) * x|$ 
 $\leq (|y| * \delta - A + |y * A' - c'| + \delta - c) * r$ 
by (simp)
show ?thesis
apply (rule le-add-right-mono[ $of\ - - |(y * (A - A') + (y * A' - c') + (c' - c))$ 
 $* x|]$ )
apply (simp-all only: 5 14[simplified abs-of-nonneg[ $of\ y$ , simplified assms]])
done
qed

```

```

lemma le-ge-imp-abs-diff-1:
assumes
 $A1 \leq (A::'a::lattice-ring)$ 
 $A \leq A2$ 
shows  $|A - A1| \leq A2 - A1$ 
proof -
have  $0 \leq A - A1$ 
proof -
from assms add-right-mono [of A1 A - A1] show ?thesis by simp
qed
then have  $|A - A1| = A - A1$  by (rule abs-of-nonneg)
with assms show  $|A - A1| \leq (A2 - A1)$  by simp
qed

```

```

lemma mult-le-prts:
assumes
 $a1 \leq (a::'a::lattice-ring)$ 
 $a \leq a2$ 
 $b1 \leq b$ 
 $b \leq b2$ 
shows
 $a * b \leq pp\text{rt } a2 * pp\text{rt } b2 + pp\text{rt } a1 * n\text{prt } b2 + n\text{prt } a2 * pp\text{rt } b1 + n\text{prt } a1$ 
 $* n\text{prt } b1$ 
proof -
have  $a * b = (pp\text{rt } a + n\text{prt } a) * (pp\text{rt } b + n\text{prt } b)$ 
apply (subst prts[symmetric]) +
apply simp
done
then have  $a * b = pp\text{rt } a * pp\text{rt } b + pp\text{rt } a * n\text{prt } b + n\text{prt } a * pp\text{rt } b + n\text{prt}$ 
 $a * n\text{prt } b$ 
by (simp add: algebra-simps)
moreover have  $pp\text{rt } a * pp\text{rt } b \leq pp\text{rt } a2 * pp\text{rt } b2$ 

```

```

    by (simp-all add: assms mult-mono)
  moreover have pprr a * npr b <= pprr a1 * npr b2
proof -
  have pprr a * npr b <= pprr a * npr b2
    by (simp add: mult-left-mono assms)
  moreover have pprr a * npr b2 <= pprr a1 * npr b2
    by (simp add: mult-right-mono-neg assms)
  ultimately show ?thesis
    by simp
qed
moreover have npr a * pprr b <= npr a2 * pprr b1
proof -
  have npr a * pprr b <= npr a2 * pprr b
    by (simp add: mult-right-mono assms)
  moreover have npr a2 * pprr b <= npr a2 * pprr b1
    by (simp add: mult-left-mono-neg assms)
  ultimately show ?thesis
    by simp
qed
moreover have npr a * npr b <= npr a1 * npr b1
proof -
  have npr a * npr b <= npr a * npr b1
    by (simp add: mult-left-mono-neg assms)
  moreover have npr a * npr b1 <= npr a1 * npr b1
    by (simp add: mult-right-mono-neg assms)
  ultimately show ?thesis
    by simp
qed
ultimately show ?thesis
  by - (rule add-mono | simp)+
qed

lemma mult-le-dual-prts:
  assumes
    A * x ≤ (b::'a::lattice-ring)
    0 ≤ y
    A1 ≤ A
    A ≤ A2
    c1 ≤ c
    c ≤ c2
    r1 ≤ x
    x ≤ r2
  shows
    c * x ≤ y * b + (let s1 = c1 - y * A2; s2 = c2 - y * A1 in pprr s2 * pprr r2
+ pprr s1 * npr r2 + npr s2 * pprr r1 + npr s1 * npr r1)
    (is - <= - + ?C)
proof -
  from assms have y * (A * x) <= y * b by (simp add: mult-left-mono)
  moreover have y * (A * x) = c * x + (y * A - c) * x by (simp add: alge-

```

```

bra-simps)
ultimately have  $c * x + (y * A - c) * x \leq y * b$  by simp
then have  $c * x \leq y * b - (y * A - c) * x$  by (simp add: le-diff-eq)
then have  $cx: c * x \leq y * b + (c - y * A) * x$  by (simp add: algebra-simps)
have  $s2: c - y * A \leq c2 - y * A1$ 
  by (simp add: assms add-mono mult-left-mono algebra-simps)
have  $s1: c1 - y * A2 \leq c - y * A$ 
  by (simp add: assms add-mono mult-left-mono algebra-simps)
have prts:  $(c - y * A) * x \leq ?C$ 
  apply (simp add: Let-def)
  apply (rule mult-le-prts)
  apply (simp-all add: assms s1 s2)
done
then have  $y * b + (c - y * A) * x \leq y * b + ?C$ 
  by simp
with cx show ?thesis
  by(simp only:)
qed

end

```

1 Floating Point Representation of the Reals

```

theory ComputeFloat
imports Complex-Main HOL-Library.Lattice-Algebras
begin

```

```

ML-file <~~/src/Tools/float.ML>

```

```

definition int-of-real :: real  $\Rightarrow$  int
  where int-of-real  $x = (SOME y. real-of-int y = x)$ 

```

```

definition real-is-int :: real  $\Rightarrow$  bool
  where real-is-int  $x = (\exists (u::int). x = real-of-int u)$ 

```

```

lemma real-is-int-def2: real-is-int  $x = (x = real-of-int (int-of-real x))$ 
  by (auto simp add: real-is-int-def int-of-real-def)

```

```

lemma real-is-int-real[simp]: real-is-int (real-of-int (x::int))
  by (auto simp add: real-is-int-def int-of-real-def)

```

```

lemma int-of-real-real[simp]: int-of-real (real-of-int  $x$ ) =  $x$ 
  by (simp add: int-of-real-def)

```

```

lemma real-int-of-real[simp]: real-is-int  $x \implies real-of-int (int-of-real x) = x$ 
  by (auto simp add: int-of-real-def real-is-int-def)

```

lemma *real-is-int-add-int-of-real*: $real-is-int\ a \implies real-is-int\ b \implies (int-of-real\ (a+b)) = (int-of-real\ a) + (int-of-real\ b)$
by (*auto simp add: int-of-real-def real-is-int-def*)

lemma *real-is-int-add[simp]*: $real-is-int\ a \implies real-is-int\ b \implies real-is-int\ (a+b)$
apply (*subst real-is-int-def2*)
apply (*simp add: real-is-int-add-int-of-real real-int-of-real*)
done

lemma *int-of-real-sub*: $real-is-int\ a \implies real-is-int\ b \implies (int-of-real\ (a-b)) = (int-of-real\ a) - (int-of-real\ b)$
by (*auto simp add: int-of-real-def real-is-int-def*)

lemma *real-is-int-sub[simp]*: $real-is-int\ a \implies real-is-int\ b \implies real-is-int\ (a-b)$
apply (*subst real-is-int-def2*)
apply (*simp add: int-of-real-sub real-int-of-real*)
done

lemma *real-is-int-rep*: $real-is-int\ x \implies \exists!(a::int). real-of-int\ a = x$
by (*auto simp add: real-is-int-def*)

lemma *int-of-real-mult*:
assumes $real-is-int\ a\ real-is-int\ b$
shows $(int-of-real\ (a*b)) = (int-of-real\ a) * (int-of-real\ b)$
using *assms*
by (*auto simp add: real-is-int-def of-int-mult[symmetric] simp del: of-int-mult*)

lemma *real-is-int-mult[simp]*: $real-is-int\ a \implies real-is-int\ b \implies real-is-int\ (a*b)$
apply (*subst real-is-int-def2*)
apply (*simp add: int-of-real-mult*)
done

lemma *real-is-int-0[simp]*: $real-is-int\ (0::real)$
by (*simp add: real-is-int-def int-of-real-def*)

lemma *real-is-int-1[simp]*: $real-is-int\ (1::real)$
proof –
have $real-is-int\ (1::real) = real-is-int(real-of-int\ (1::int))$ **by** *auto*
also have $\dots = True$ **by** (*simp only: real-is-int-real*)
ultimately show *?thesis* **by** *auto*
qed

lemma *real-is-int-n1*: $real-is-int\ (-1::real)$
proof –
have $real-is-int\ (-1::real) = real-is-int(real-of-int\ (-1::int))$ **by** *auto*
also have $\dots = True$ **by** (*simp only: real-is-int-real*)
ultimately show *?thesis* **by** *auto*
qed

lemma *real-is-int-numeral*[simp]: *real-is-int* (numeral x)
by (*auto simp: real-is-int-def intro!: exI[of - numeral x]*)

lemma *real-is-int-neg-numeral*[simp]: *real-is-int* ($-$ numeral x)
by (*auto simp: real-is-int-def intro!: exI[of - $-$ numeral x]*)

lemma *int-of-real-0*[simp]: *int-of-real* ($0::\text{real}$) = ($0::\text{int}$)
by (*simp add: int-of-real-def*)

lemma *int-of-real-1*[simp]: *int-of-real* ($1::\text{real}$) = ($1::\text{int}$)
proof $-$
have $1: (1::\text{real}) = \text{real-of-int } (1::\text{int})$ **by** *auto*
show *?thesis* **by** (*simp only: 1 int-of-real-real*)
qed

lemma *int-of-real-numeral*[simp]: *int-of-real* (numeral b) = numeral b
unfolding *int-of-real-def* **by** *simp*

lemma *int-of-real-neg-numeral*[simp]: *int-of-real* ($-$ numeral b) = $-$ numeral b
unfolding *int-of-real-def*
by (*metis int-of-real-def int-of-real-real of-int-minus of-int-of-nat-eq of-nat-numeral*)

lemma *int-div-zdiv*: *int* ($a \text{ div } b$) = (*int* a) *div* (*int* b)
by (*rule zdiv-int*)

lemma *int-mod-zmod*: *int* ($a \text{ mod } b$) = (*int* a) *mod* (*int* b)
by (*rule zmod-int*)

lemma *abs-div-2-less*: $a \neq 0 \implies a \neq -1 \implies |(a::\text{int}) \text{ div } 2| < |a|$
by *arith*

lemma *norm-0-1*: ($1:::\text{numeral}$) = *Numeral1*
by *auto*

lemma *add-left-zero*: $0 + a = (a::'a::\text{comm-monoid-add})$
by *simp*

lemma *add-right-zero*: $a + 0 = (a::'a::\text{comm-monoid-add})$
by *simp*

lemma *mult-left-one*: $1 * a = (a::'a::\text{semiring-1})$
by *simp*

lemma *mult-right-one*: $a * 1 = (a::'a::\text{semiring-1})$
by *simp*

lemma *int-pow-0*: $(a::\text{int})^{\wedge}0 = 1$

by *simp*

lemma *int-pow-1*: $(a::int) \wedge (\text{Numeral1}) = a$
by *simp*

lemma *one-eq-Numeral1-nring*: $(1::'a::numeral) = \text{Numeral1}$
by *simp*

lemma *one-eq-Numeral1-nat*: $(1::nat) = \text{Numeral1}$
by *simp*

lemma *zpower-Pls*: $(z::int) \wedge 0 = \text{Numeral1}$
by *simp*

lemma *fst-cong*: $a=a' \implies \text{fst } (a,b) = \text{fst } (a',b)$
by *simp*

lemma *snd-cong*: $b=b' \implies \text{snd } (a,b) = \text{snd } (a,b')$
by *simp*

lemma *lift-bool*: $x \implies x = \text{True}$
by *simp*

lemma *nlift-bool*: $\sim x \implies x = \text{False}$
by *simp*

lemma *not-false-eq-true*: $(\sim \text{False}) = \text{True}$ **by** *simp*

lemma *not-true-eq-false*: $(\sim \text{True}) = \text{False}$ **by** *simp*

lemmas *powerarith = nat-numeral power-numeral-even power-numeral-odd zpower-Pls*

definition *float* :: $(int \times int) \Rightarrow real$ **where**
float = $(\lambda(a, b). \text{real-of-int } a * 2^{\text{powr } \text{real-of-int } b})$

lemma *float-add-l0*: $\text{float } (0, e) + x = x$
by (*simp add: float-def*)

lemma *float-add-r0*: $x + \text{float } (0, e) = x$
by (*simp add: float-def*)

lemma *float-add*:
 $\text{float } (a1, e1) + \text{float } (a2, e2) =$
(if $e1 \leq e2$ *then* $\text{float } (a1 + a2 * 2^{\text{nat}(e2 - e1)}, e1)$ *else* $\text{float } (a1 * 2^{\text{nat}(e1 - e2)} + a2,$
 $e2)$ *)*
by (*simp add: float-def algebra-simps powr-realpow[symmetric] powr-diff*)

lemma *float-mult-l0*: $\text{float } (0, e) * x = \text{float } (0, 0)$

by (*simp add: float-def*)

lemma float-mult-r0: $x * \text{float } (0, e) = \text{float } (0, 0)$
by (*simp add: float-def*)

lemma float-mult:
 $\text{float } (a1, e1) * \text{float } (a2, e2) = (\text{float } (a1 * a2, e1 + e2))$
by (*simp add: float-def powr-add*)

lemma float-minus:
 $-(\text{float } (a, b)) = \text{float } (-a, b)$
by (*simp add: float-def*)

lemma zero-le-float:
 $(0 \leq \text{float } (a, b)) = (0 \leq a)$
by (*simp add: float-def zero-le-mult-iff*)

lemma float-le-zero:
 $(\text{float } (a, b) \leq 0) = (a \leq 0)$
by (*simp add: float-def mult-le-0-iff*)

lemma float-abs:
 $|\text{float } (a, b)| = (\text{if } 0 \leq a \text{ then } (\text{float } (a, b)) \text{ else } (\text{float } (-a, b)))$
by (*simp add: float-def abs-if mult-less-0-iff not-less*)

lemma float-zero:
 $\text{float } (0, b) = 0$
by (*simp add: float-def*)

lemma float-pprt:
 $\text{pprt } (\text{float } (a, b)) = (\text{if } 0 \leq a \text{ then } (\text{float } (a, b)) \text{ else } (\text{float } (0, b)))$
by (*auto simp add: zero-le-float float-le-zero float-zero*)

lemma float-nprt:
 $\text{nprt } (\text{float } (a, b)) = (\text{if } 0 \leq a \text{ then } (\text{float } (0, b)) \text{ else } (\text{float } (a, b)))$
by (*auto simp add: zero-le-float float-le-zero float-zero*)

definition lbound :: $\text{real} \Rightarrow \text{real}$
where $\text{lbound } x = \min 0 x$

definition ubound :: $\text{real} \Rightarrow \text{real}$
where $\text{ubound } x = \max 0 x$

lemma lbound: $\text{lbound } x \leq x$
by (*simp add: lbound-def*)

lemma ubound: $x \leq \text{ubound } x$
by (*simp add: ubound-def*)


```

lemma ppert-lbound: ppert (lbound x) = float (0, 0)
  by (auto simp: float-def lbound-def)

lemma npert-ubound: npert (ubound x) = float (0, 0)
  by (auto simp: float-def ubound-def)

lemmas floatarith[simplified norm-0-1] = float-add float-add-l0 float-add-r0 float-mult
float-mult-l0 float-mult-r0
float-minus float-abs zero-le-float float-ppert float-npert ppert-lbound npert-ubound

lemmas arith = arith-simps rel-simps diff-nat-numeral nat-0
nat-neg-numeral powerarith floatarith not-false-eq-true not-true-eq-false

ML-file <float-arith.ML>

end

theory Compute-Oracle imports HOL.HOL
begin

ML-file <am.ML>
ML-file <am-compiler.ML>
ML-file <am-interpreter.ML>
ML-file <am-ghc.ML>
ML-file <am-sml.ML>
ML-file <report.ML>
ML-file <compute.ML>
ML-file <linker.ML>

end
theory ComputeHOL
imports Complex-Main Compute-Oracle/Compute-Oracle
begin

lemma Trueprop-eq-eq: Trueprop X == (X == True) by (simp add: atomize-eq)
lemma meta-eq-trivial: x == y ==> x == y by simp
lemma meta-eq-imp-eq: x == y ==> x = y by auto
lemma eq-trivial: x = y ==> x = y by auto
lemma bool-to-true: x :: bool ==> x == True by simp
lemma transmeta-1: x = y ==> y == z ==> x = z by simp
lemma transmeta-2: x == y ==> y = z ==> x = z by simp
lemma transmeta-3: x == y ==> y == z ==> x = z by simp

lemma If-True: If True = (λ x y. x) by ((rule ext)+, auto)

```

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

lemmas *compute-if* = *If-True If-False*

lemma *bool1*: $(\neg True) = False$ **by** *blast*
lemma *bool2*: $(\neg False) = True$ **by** *blast*
lemma *bool3*: $(P \wedge True) = P$ **by** *blast*
lemma *bool4*: $(True \wedge P) = P$ **by** *blast*
lemma *bool5*: $(P \wedge False) = False$ **by** *blast*
lemma *bool6*: $(False \wedge P) = False$ **by** *blast*
lemma *bool7*: $(P \vee True) = True$ **by** *blast*
lemma *bool8*: $(True \vee P) = True$ **by** *blast*
lemma *bool9*: $(P \vee False) = P$ **by** *blast*
lemma *bool10*: $(False \vee P) = P$ **by** *blast*
lemma *bool11*: $(True \longrightarrow P) = P$ **by** *blast*
lemma *bool12*: $(P \longrightarrow True) = True$ **by** *blast*
lemma *bool13*: $(True \longrightarrow P) = P$ **by** *blast*
lemma *bool14*: $(P \longrightarrow False) = (\neg P)$ **by** *blast*
lemma *bool15*: $(False \longrightarrow P) = True$ **by** *blast*
lemma *bool16*: $(False = False) = True$ **by** *blast*
lemma *bool17*: $(True = True) = True$ **by** *blast*
lemma *bool18*: $(False = True) = False$ **by** *blast*
lemma *bool19*: $(True = False) = False$ **by** *blast*

lemmas *compute-bool* = *bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10 bool11 bool12 bool13 bool14 bool15 bool16 bool17 bool18 bool19*

lemma *compute-fst*: $fst\ (x,y) = x$ **by** *simp*
lemma *compute-snd*: $snd\ (x,y) = y$ **by** *simp*
lemma *compute-pair-eq*: $((a, b) = (c, d)) = (a = c \wedge b = d)$ **by** *auto*

lemma *case-prod-simp*: $case\ prod\ f\ (x,y) = f\ x\ y$ **by** *simp*

lemmas *compute-pair* = *compute-fst compute-snd compute-pair-eq case-prod-simp*

lemma *compute-the*: $the\ (Some\ x) = x$ **by** *simp*
lemma *compute-None-Some-eq*: $(None = Some\ x) = False$ **by** *auto*
lemma *compute-Some-None-eq*: $(Some\ x = None) = False$ **by** *auto*
lemma *compute-None-None-eq*: $(None = None) = True$ **by** *auto*
lemma *compute-Some-Some-eq*: $(Some\ x = Some\ y) = (x = y)$ **by** *auto*

definition *case-option-compute* :: $'b\ option \Rightarrow 'a \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$

where *case-option-compute* $opt\ a\ f = case-option\ a\ f\ opt$

lemma *case-option-compute*: $case-option = (\lambda\ a\ f\ opt.\ case-option-compute\ opt\ a\ f)$
by (*simp add: case-option-compute-def*)

lemma *case-option-compute-None*: $case-option-compute\ None = (\lambda\ a\ f.\ a)$
apply (*rule ext*)
apply (*simp add: case-option-compute-def*)
done

lemma *case-option-compute-Some*: $case-option-compute\ (Some\ x) = (\lambda\ a\ f.\ f\ x)$
apply (*rule ext*)
apply (*simp add: case-option-compute-def*)
done

lemmas *compute-case-option* = *case-option-compute case-option-compute-None case-option-compute-Some*

lemmas *compute-option* = *compute-the compute-None-Some-eq compute-Some-None-eq compute-None-None-eq compute-Some-Some-eq compute-case-option*

lemma *length-cons*: $length\ (x\#\ xs) = 1 + (length\ xs)$
by *simp*

lemma *length-nil*: $length\ [] = 0$
by *simp*

lemmas *compute-list-length* = *length-nil length-cons*

definition *case-list-compute* :: $'b\ list \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b\ list \Rightarrow 'a) \Rightarrow 'a$
where *case-list-compute* $l\ a\ f = case-list\ a\ f\ l$

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

lemma *case-list-compute-empty*: $case-list-compute\ ([]::'b\ list) = (\lambda\ (a::'a)\ f.\ a)$
apply (*rule ext*)
apply (*simp add: case-list-compute-def*)
done

lemma *case-list-compute-cons*: $case-list-compute\ (u\#\ v) = (\lambda\ (a::'a)\ f.\ (f\ (u::'b)\ v))$

```

apply (rule ext)+
apply (simp add: case-list-compute-def)
done

```

lemmas *compute-case-list = case-list-compute case-list-compute-empty case-list-compute-cons*

```

lemma compute-list-nth: ((x#xs) ! n) = (if n = 0 then x else (xs ! (n - 1)))
  by (cases n, auto)

```

lemmas *compute-list = compute-case-list compute-list-length compute-list-nth*

lemmas *compute-let = Let-def*

lemmas *compute-hol = compute-if compute-bool compute-pair compute-option compute-list compute-let*

```

ML <
signature ComputeHOL =
sig
  val prep-thms : thm list -> thm list
  val to-meta-eq : thm -> thm
  val to-hol-eq : thm -> thm
  val symmetric : thm -> thm
  val trans : thm -> thm -> thm
end

structure ComputeHOL : ComputeHOL =
struct

  local
  fun lhs-of eq = fst (Thm.dest-equals (Thm.cprop-of eq));
  in
  fun rewrite-conv [] ct = raise CTERM (rewrite-conv, [ct])
    | rewrite-conv (eq :: eqs) ct =
      Thm.instantiate (Thm.match (lhs-of eq, ct)) eq
        handle Pattern.MATCH => rewrite-conv eqs ct;
  end

```

```
val convert-conditions = Conv.fconv-rule (Conv.premis-conv ~ 1 (Conv.try-conv (rewrite-conv
[@{thm Trueprop-eq-eq}])))
```

```
val eq-th = @{thm HOL.eq-reflection}
val meta-eq-trivial = @{thm ComputeHOL.meta-eq-trivial}
val bool-to-true = @{thm ComputeHOL.bool-to-true}
```

```
fun to-meta-eq th = eq-th OF [th] handle THM - => meta-eq-trivial OF [th] handle
THM - => bool-to-true OF [th]
```

```
fun to-hol-eq th = @{thm meta-eq-imp-eq} OF [th] handle THM - => @{thm
eq-trivial} OF [th]
```

```
fun prep-thms ths = map (convert-conditions o to-meta-eq) ths
```

```
fun symmetric th = @{thm HOL.sym} OF [th] handle THM - => @{thm Pure.symmetric}
OF [th]
```

```
local
```

```
  val trans-HOL = @{thm HOL.trans}
  val trans-HOL-1 = @{thm ComputeHOL.transmeta-1}
  val trans-HOL-2 = @{thm ComputeHOL.transmeta-2}
  val trans-HOL-3 = @{thm ComputeHOL.transmeta-3}
  fun tr [] th1 th2 = trans-HOL OF [th1, th2]
    | tr (t::ts) th1 th2 = (t OF [th1, th2] handle THM - => tr ts th1 th2)
```

```
in
```

```
  fun trans th1 th2 = tr [trans-HOL, trans-HOL-1, trans-HOL-2, trans-HOL-3]
  th1 th2
end
```

```
end
```

```
>
```

```
end
```

```
theory ComputeNumeral
```

```
imports ComputeHOL ComputeFloat
```

```
begin
```

```
lemmas biteq = eq-num-simps
```

```
lemmas bitless = less-num-simps
```

```
lemmas bitle = le-num-simps
```

```
lemmas bitadd = add-num-simps
```

lemmas *bitmul = mult-num-simps*

lemmas *bitarith = arith-simps*

lemmas *natnorm = one-eq-Numeral1-nat*

fun *natfac :: nat ⇒ nat*
where *natfac n = (if n = 0 then 1 else n * (natfac (n - 1)))*

lemmas *compute-natarith =*
arith-simps rel-simps
diff-nat-numeral nat-numeral nat-0 nat-neg-numeral
numeral-One [symmetric]
numeral-1-eq-Suc-0 [symmetric]
Suc-numeral natfac.simps

lemmas *number-norm = numeral-One[symmetric]*

lemmas *compute-numberarith =*
arith-simps rel-simps number-norm

lemmas *compute-num-conversions =*
of-nat-numeral of-nat-0
nat-numeral nat-0 nat-neg-numeral
of-int-numeral of-int-neg-numeral of-int-0

lemmas *zpowerarith = power-numeral-even power-numeral-odd zpower-Pls int-pow-1*

lemmas *compute-div-mod = div-0 mod-0 div-by-0 mod-by-0 div-by-1 mod-by-1*
one-div-numeral one-mod-numeral minus-one-div-numeral minus-one-mod-numeral
one-div-minus-numeral one-mod-minus-numeral
numeral-div-numeral numeral-mod-numeral minus-numeral-div-numeral minus-numeral-mod-numeral
numeral-div-minus-numeral numeral-mod-minus-numeral
div-minus-minus mod-minus-minus Parity.adjust-div-eq of-bool-eq one-neq-zero
numeral-neq-zero neg-equal-0-iff-equal arith-simps arith-special divmod-trivial
divmod-steps divmod-cancel divmod-step-def fst-conv snd-conv numeral-One
case-prod-beta rel-simps Parity.adjust-mod-def div-minus1-right mod-minus1-right
minus-minus numeral-times-numeral mult-zero-right mult-1-right

lemma *even-0-int: even (0::int) = True*
by *simp*

```

lemma even-One-int: even (numeral Num.One :: int) = False
  by simp

lemma even-Bit0-int: even (numeral (Num.Bit0 x) :: int) = True
  by (simp only: even-numeral)

lemma even-Bit1-int: even (numeral (Num.Bit1 x) :: int) = False
  by (simp only: odd-numeral)

lemmas compute-even = even-0-int even-One-int even-Bit0-int even-Bit1-int

lemmas compute-numeral = compute-if compute-let compute-pair compute-bool
  compute-natarith compute-numberarith max-def min-def
compute-num-conversions zpowerarith compute-div-mod compute-even

end

theory Cplex
imports SparseMatrix LP ComputeFloat ComputeNumeral
begin

ML-file ⟨Cplex-tools.ML⟩
ML-file ⟨CplexMatrixConverter.ML⟩
ML-file ⟨FloatSparseMatrixBuilder.ML⟩
ML-file ⟨fspmlp.ML⟩

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::lattice-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) (nprrt-spmat r2))
  (add-spmat (mult-spmat (nprrt-spmat s2) (pprt-spmat r1)) (mult-spmat (nprrt-spmat
s1) (nprrt-spmat r1))))))
  apply (simp add: Let-def)
  apply (insert assms)
  apply (simp add: sparse-row-matrix-op-simps algebra-simps)
  apply (rule mult-le-dual-prts[where A=A, simplified Let-def algebra-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-spmat 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::lattice-ring) spmat)
shows
  c * x ≤ 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: assms mult-est-spmat-def spm-mult-le-dual-prts[where A=A, sim-
plified Let-def])

```

ML-file *<matrixlp.ML>*

end