

Matrix

Steven Obua

December 17, 2025

```
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 matrix-eqI:
  fixes A B :: 'a::zero matrix
  assumes  $\bigwedge m n. \text{Rep-matrix } A \ m \ n = \text{Rep-matrix } B \ m \ n$ 
  shows A=B
  using Rep-matrix-inject assms by blast

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

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

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

```

lemma nrows:
  assumes hyp:  $\text{nrows } A \leq m$ 
  shows  $(\text{Rep-matrix } A \ m \ n) = 0$ 
proof cases
  assume  $\text{nonzero-positions}(\text{Rep-matrix } A) = \{\}$ 
  then show  $(\text{Rep-matrix } A \ m \ n) = 0$  by (simp add: nonzero-positions-def)
next
  assume  $a: \text{nonzero-positions}(\text{Rep-matrix } A) \neq \{\}$ 
  let  $?S = \text{fst}(\text{nonzero-positions}(\text{Rep-matrix } A))$ 
  have  $c: \text{finite } (?S)$  by (simp add: finite-nonzero-positions)
  from hyp have  $d: \text{Max } (?S) < m$  by (simp add: a nrows-def)
  have  $m \notin ?S$ 
  proof  $-$ 
    have  $m \in ?S \implies m \leq \text{Max } (?S)$  by (simp add: Max-ge [OF c])
    moreover from d have  $\sim(m \leq \text{Max } ?S)$  by (simp)
    ultimately show  $m \notin ?S$  by (auto)
  qed
  thus  $\text{Rep-matrix } A \ m \ n = 0$  by (simp add: nonzero-positions-def image-Collect)
qed

definition transpose-infmatrix ::  $'a \ \text{infmatrix} \Rightarrow 'a \ \text{infmatrix}$  where
   $\text{transpose-infmatrix } A \ j \ i == A \ i \ j$ 

definition transpose-matrix ::  $('a::\text{zero}) \ \text{matrix} \Rightarrow 'a \ \text{matrix}$  where
   $\text{transpose-matrix} == \text{Abs-matrix } o \ \text{transpose-infmatrix } o \ \text{Rep-matrix}$ 

declare transpose-infmatrix-def[simp]

lemma transpose-infmatrix-twice[simp]:  $\text{transpose-infmatrix } (\text{transpose-infmatrix } A) = A$ 
by ((rule ext)+, simp)

lemma transpose-infmatrix:  $\text{transpose-infmatrix } (\lambda j \ i. P \ j \ i) = (\lambda j \ i. P \ i \ j)$ 
by force

lemma transpose-infmatrix-closed[simp]:  $\text{Rep-matrix } (\text{Abs-matrix } (\text{transpose-infmatrix } (\text{Rep-matrix } x))) = \text{transpose-infmatrix } (\text{Rep-matrix } x)$ 
proof  $-$ 
  let  $?A = \{\text{pos. } \text{Rep-matrix } x \ (\text{snd pos}) \ (\text{fst pos}) \neq 0\}$ 
  let  $?B = \{\text{pos. } \text{Rep-matrix } x \ (\text{fst pos}) \ (\text{snd pos}) \neq 0\}$ 
  let  $?swap = \lambda \text{pos. } (\text{snd pos}, \text{fst pos})$ 
  have finite  $?A$ 
  proof  $-$ 
    have swap-image:  $?swap' ?A = ?B$ 
    by (force simp add: image-def)
    then have finite  $(?swap' ?A)$ 
    by (metis (full-types) finite-nonzero-positions nonzero-positions-def)
    moreover

```

```

    have inj-on ?swap ?A by (simp add: inj-on-def)
    ultimately show finite ?A
      using finite-imageD by blast
  qed
  then show ?thesis
    by (simp add: Abs-matrix-inverse matrix-def nonzero-positions-def)
qed

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

lemma transpose-infmatrix-inject: (transpose-infmatrix A = transpose-infmatrix B) = (A = B)
  by (metis transpose-infmatrix-twice)

lemma transpose-matrix-inject: (transpose-matrix A = transpose-matrix B) = (A = B)
  unfolding transpose-matrix-def o-def
  by (metis Rep-matrix-inject transpose-infmatrix-closed transpose-infmatrix-inject)

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

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

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

lemma ncols-transpose[simp]: ncols (transpose-matrix A) = nrows A
  by (metis nrows-transpose transpose-transpose-id)

lemma ncols: ncols A  $\leq$  n  $\implies$  Rep-matrix A m n = 0
  by (metis nrows nrows-transpose transpose-matrix)

lemma ncols-le: (ncols A  $\leq$  n)  $\longleftrightarrow$  ( $\forall j i. n \leq i \longrightarrow$  (Rep-matrix A j i) = 0) (is
- = ?st)
proof -
  have Rep-matrix A j i = 0
    if ncols A  $\leq$  n n  $\leq$  i for j i
    by (meson that le-trans ncols)
  moreover have ncols A  $\leq$  n
    if  $\forall j i. n \leq i \longrightarrow$  Rep-matrix A j i = 0
    unfolding ncols-def
  proof (clarsimp split: if-split-asm)
    assume  $\S$ : nonzero-positions (Rep-matrix A)  $\neq$  {}
    let ?P = nonzero-positions (Rep-matrix A)

```

```

let ?p = snd' ?P
have a:finite ?p by (simp add: finite-nonzero-positions)
let ?m = Max ?p
show Suc (Max (snd ' nonzero-positions (Rep-matrix A))) ≤ n
  using § that obtains-MAX [OF finite-nonzero-positions]
  by (metis (mono-tags, lifting) mem-Collect-eq nonzero-positions-def not-less-eq-eq)
qed
ultimately show ?thesis
  by auto
qed

```

lemma less-ncols: $(n < \text{ncols } A) = (\exists j \ i. n \leq i \wedge (\text{Rep-matrix } A \ j \ i) \neq 0)$
 by (meson linorder-not-le ncols-le)

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

lemma nrows-le: $(\text{nrows } A \leq n) = (\forall j \ i. n \leq j \longrightarrow (\text{Rep-matrix } A \ j \ i) = 0)$ (is ?s)
 by (metis ncols-le ncols-transpose transpose-matrix)

lemma less-nrows: $(m < \text{nrows } A) = (\exists j \ i. m \leq j \wedge (\text{Rep-matrix } A \ j \ i) \neq 0)$
 by (meson linorder-not-le nrows-le)

lemma le-nrows: $(n \leq \text{nrows } A) = (\forall m. (\forall j \ i. m \leq j \longrightarrow (\text{Rep-matrix } A \ j \ i) = 0) \longrightarrow n \leq m)$
 by (meson order.trans nrows nrows-le)

lemma nrows-notzero: $\text{Rep-matrix } A \ m \ n \neq 0 \implies m < \text{nrows } A$
 by (meson leI nrows)

lemma ncols-notzero: $\text{Rep-matrix } A \ m \ n \neq 0 \implies n < \text{ncols } A$
 by (meson leI ncols)

lemma finite-natarray1: $\text{finite } \{x. x < (n::\text{nat})\}$
 by simp

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

lemma RepAbs-matrix:
 assumes $\exists m. \forall j \ i. m \leq j \longrightarrow x \ j \ i = 0$
 and $\exists n. \forall j \ i. (n \leq i \longrightarrow x \ j \ i = 0)$
 shows $(\text{Rep-matrix } (\text{Abs-matrix } x)) = x$
proof –
 have finite $\{\text{pos. } x \ (\text{fst pos}) \ (\text{snd pos}) \neq 0\}$
proof –
 from assms obtain $m \ n$ where $a: \forall j \ i. m \leq j \longrightarrow x \ j \ i = 0$

```

    and 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 =  $\{(i, j). i < m \wedge j < n\}$ 
  have c:  $\bigwedge(m::nat)\ a. \sim(m \leq a) \implies a < m$  by (arith)
  with a b 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
then show ?thesis
  by (simp add: Abs-matrix-inverse matrix-def nonzero-positions-def)
qed

```

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

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

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

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

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

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

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

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

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*:
 $\text{commutative } f \implies \text{commutative } (\text{combine-infmatrix } f)$
by (*simp add: commutative-def combine-infmatrix-def*)

lemma *combine-matrix-commute*:
 $\text{commutative } f \implies \text{commutative } (\text{combine-matrix } f)$
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 (*smt (verit) UnCI expand-combine-infmatrix mem-Collect-eq nonzero-positions-def subsetI*)

lemma *finite-nonzero-positions-Rep[simp]*: $\text{finite } (\text{nonzero-positions } (\text{Rep-matrix } A))$
by (*simp add: finite-nonzero-positions*)

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*)
by (*meson finite-Un finite-nonzero-positions-Rep finite-subset nonzero-positions-combine-infmatrix*)

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*)

by (*meson finite-nonzero-positions-Rep finite-subset nonzero-positions-apply-infmatrix*)

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

lemma *combine-matrix-assoc*: $f\ 0\ 0 = 0 \implies \text{associative } f \implies \text{associative}$ (*combine-matrix f*)
by (*smt (verit) associative-def combine-infmatrix-assoc combine-infmatrix-closed combine-matrix-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* :: $('a \Rightarrow 'b :: \text{zero} \Rightarrow 'a) \Rightarrow \text{bool}$ **where**
zero-r-neutral f == $\forall a. f\ a\ 0 = a$

definition *zero-l-neutral* :: $('a :: \text{zero} \Rightarrow 'b \Rightarrow 'b) \Rightarrow \text{bool}$ **where**
zero-l-neutral f == $\forall a. f\ 0\ a = a$

definition *zero-closed* :: $(('a :: \text{zero}) \Rightarrow ('b :: \text{zero}) \Rightarrow ('c :: \text{zero})) \Rightarrow \text{bool}$ **where**
zero-closed f == $(\forall x. f\ x\ 0 = 0) \wedge (\forall y. f\ 0\ y = 0)$

primrec *foldseq* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a$
where
foldseq f s 0 = s 0
 $| \text{foldseq } f\ s\ (\text{Suc } n) = f\ (s\ 0)\ (\text{foldseq } f\ (\lambda k. s(\text{Suc } k))\ n)$

```

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:
  assumes a:associative f
  shows associative f ⇒ foldseq f = foldseq-transposed f
proof -
  have N ≤ n ⇒ foldseq f s N = foldseq-transposed f s N for N s n
  proof (induct n arbitrary: N s)
    case 0
    then show ?case
    by auto
  next
    case (Suc n)
    show ?case
    proof cases
      assume N ≤ n
      then show ?thesis
      by (simp add: Suc.hyps)
    next
      assume ~ (N ≤ n)
      then have Nsucceq: N = Suc n
      using Suc.premis by linarith
      have neqz: n ≠ 0 ⇒ ∃ m. n = Suc m ∧ Suc m ≤ n
      by arith
      have assocf: !! x y z. f x (f y z) = f (f x y) z
      by (metis a associative-def)
      have f (f (s 0) (foldseq-transposed f (λk. s (Suc k)) m)) (s (Suc (Suc m))) =
        f (f (foldseq-transposed f s m) (s (Suc m))) (s (Suc (Suc m)))
      if n = Suc m for m
      proof -
        have §: foldseq-transposed f (λk. s (Suc k)) m = foldseq f (λk. s (Suc k))
m (is ?T1 = ?T2)
        by (simp add: Suc.hyps that)
        have f (s 0) ?T2 = foldseq f s (Suc m) by simp
        also have ... = foldseq-transposed f s (Suc m)
        using Suc.hyps that by blast
        also have ... = f (foldseq-transposed f s m) (s (Suc m))
        by simp
        finally show ?thesis
        by (simp add: §)
      qed
      then show foldseq f s N = foldseq-transposed f s N
      unfolding Nsucceq using assocf Suc.hyps neqz by force
    qed
  qed
then show ?thesis

```


by *blast*
qed

lemma *foldseq-distr*:

assumes *assoc*: *associative f* and *comm*: *commutative f*
shows $\text{foldseq } f \ (\lambda k. f \ (u \ k) \ (v \ k)) \ n = f \ (\text{foldseq } f \ u \ n) \ (\text{foldseq } f \ v \ n)$
proof –
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 $(\forall u \ v. \text{foldseq } f \ (\lambda k. f \ (u \ k) \ (v \ k)) \ n = f \ (\text{foldseq } f \ u \ n) \ (\text{foldseq } f \ v \ n))$ for n
by (*induct n (simp-all add: assoc b c foldseq-assoc)*)
then show $\text{foldseq } f \ (\lambda k. f \ (u \ k) \ (v \ k)) \ n = f \ (\text{foldseq } f \ u \ n) \ (\text{foldseq } f \ v \ n)$ 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 $\forall s. (\forall i. i \leq n \longrightarrow s \ i = 0) \longrightarrow \text{foldseq } f \ s \ n = 0$ for n
by (*induct n (simp-all add: fz)*)
then show *?thesis*
by (*simp add: sz*)
qed

lemma *foldseq-significant-positions*:

assumes *p*: $\forall i. i \leq N \longrightarrow S \ i = T \ i$
shows $\text{foldseq } f \ S \ N = \text{foldseq } f \ T \ N$
using *assms*
proof (*induction N arbitrary: S T*)
case 0
then show *?case* by *simp*
next
case (*Suc N*)
then show *?case*
unfolding *foldseq.simps* by (*metis not-less-eq-eq le0*)
qed

lemma *foldseq-tail*:

assumes $M \leq N$

```

shows foldseq f S N = foldseq f (λk. (if k < M then (S k) else (foldseq f (λk.
S(k+M)) (N-M)))) M
using assms
proof (induction N arbitrary: M S)
  case 0
  then show ?case by auto
next
  case (Suc N)
  show ?case
  proof (cases M = Suc N)
    case True
    then show ?thesis
    by (auto intro!: arg-cong [of concl: f (S 0)] foldseq-significant-positions)
  next
  case False
  then have M ≤ N
  using Suc.prem by force
  show ?thesis
  proof (cases M = 0)
    case True
    then show ?thesis
    by auto
  next
  case False
  then obtain M' where M': M = Suc M' M' ≤ N
  by (metis Suc-leD ⟨M ≤ N⟩ nat.nchotomy)
  then show ?thesis
  apply (simp add: Suc.IH [OF ⟨M' ≤ N⟩])
  using add-Suc-right diff-Suc-Suc by presburger
  qed
qed
qed

```

lemma foldseq-zerotail:

```

assumes fz: f 0 0 = 0 and sz: ∀ i. n ≤ i ⟶ s i = 0 and nm: n ≤ m
shows foldseq f s n = foldseq f s m
unfolding foldseq-tail[OF nm]
by (metis (no-types, lifting) foldseq-zero fz le-add2 linorder-not-le sz)

```

lemma foldseq-zerotail2:

```

assumes ∀ x. f x 0 = x
and ∀ i. n < i ⟶ s i = 0
and nm: n ≤ m
shows foldseq f s n = foldseq f s m
proof –
  have s i = (if i < n then s i else foldseq f (λk. s (k + n)) (m - n))
  if i ≤ n for i
  proof (cases m=n)
    case True

```

```

    then show ?thesis
      using that by auto
  next
    case False
    then obtain k where m - n = Suc k
      by (metis Suc-diff-Suc le-neq-implies-less nm)
    then show ?thesis
      apply simp
      by (simp add: assms(1,2) foldseq-zero nat-less-le that)
  qed
  then show ?thesis
    unfolding foldseq-tail[OF nm]
    by (auto intro: foldseq-significant-positions)
qed

lemma foldseq-zerostart:
  assumes f00x:  $\forall x. f\ 0\ (f\ 0\ x) = f\ 0\ x$  and 0:  $\forall i. i \leq n \longrightarrow s\ i = 0$ 
  shows foldseq f s (Suc n) = f 0 (s (Suc n))
  using 0
proof (induction n arbitrary: s)
  case 0
  then show ?case by auto
next
  case (Suc n s)
  then show ?case
    apply (simp add: le-Suc-eq)
    by (smt (verit, ccfv-threshold) Suc.premis Suc-le-mono f00x foldseq-significant-positions le0)
qed

lemma foldseq-zerostart2:
  assumes x:  $\forall x. f\ 0\ x = x$  and 0:  $\forall i. i < n \longrightarrow s\ i = 0$ 
  shows foldseq f s n = s n
proof -
  show foldseq f s n = s n
  proof (cases n)
    case 0
    then show ?thesis
      by auto
  next
    case (Suc n')
    then show ?thesis
      by (metis 0 foldseq-zerostart le-imp-less-Suc x)
  qed
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 $\text{foldseq } f \ s \ n = (\text{if } (j \leq n) \text{ then } (s \ j) \text{ else } 0)$
by (*smt* (*verit*, *ccfv-SIG*) *f0x foldseq-zerostart2 foldseq-zerotail2 fx0 le-refl nat-less-le s0*)

lemma *foldseq-distr-unary*:

assumes $\bigwedge a \ b. \ g \ (f \ a \ b) = f \ (g \ a) \ (g \ b)$
shows $g(\text{foldseq } f \ s \ n) = \text{foldseq } f \ (\lambda x. \ g(s \ x)) \ n$
proof (*induction n arbitrary: s*)
case 0
then show ?*case*
by *auto*
next
case (*Suc n*)
then show ?*case*
using *assms* **by** *fastforce*
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**
 $\text{mult-matrix-n } n \ \text{fmul} \ \text{fadd} \ A \ B == \text{Abs-matrix}(\lambda j \ i. \ \text{foldseq} \ \text{fadd} \ (\lambda k. \ \text{fmul} \ (\text{Rep-matrix } A \ j \ k) \ (\text{Rep-matrix } B \ k \ i))) \ n$

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**
 $\text{mult-matrix} \ \text{fmul} \ \text{fadd} \ A \ B == \text{mult-matrix-n} \ (\max \ (\text{ncols } A) \ (\text{nrows } B)) \ \text{fmul} \ \text{fadd} \ A \ B$

lemma *mult-matrix-n*:

assumes $\text{ncols } A \leq n \ \text{nrows } B \leq n \ \text{fadd } 0 \ 0 = 0 \ \text{fmul } 0 \ 0 = 0$
shows $\text{mult-matrix} \ \text{fmul} \ \text{fadd} \ A \ B = \text{mult-matrix-n } n \ \text{fmul} \ \text{fadd} \ A \ B$
proof –
have $\text{foldseq} \ \text{fadd} \ (\lambda k. \ \text{fmul} \ (\text{Rep-matrix } A \ j \ k) \ (\text{Rep-matrix } B \ k \ i))$
 $(\max \ (\text{ncols } A) \ (\text{nrows } B)) =$
 $\text{foldseq} \ \text{fadd} \ (\lambda k. \ \text{fmul} \ (\text{Rep-matrix } A \ j \ k) \ (\text{Rep-matrix } B \ k \ i)) \ n$ **for** $i \ j$
using *assms* **by** (*simp add: foldseq-zerotail nrows-le ncols-le*)
then show ?*thesis*
by (*simp add: mult-matrix-def mult-matrix-n-def*)
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 $\text{mult-matrix-n } n \ \text{fmul} \ \text{fadd} \ A \ B = \text{mult-matrix-n } m \ \text{fmul} \ \text{fadd} \ A \ B$
proof –
from *assms* **have** $\text{mult-matrix-n } n \ \text{fmul} \ \text{fadd} \ A \ B = \text{mult-matrix} \ \text{fmul} \ \text{fadd} \ A \ B$
by (*simp add: mult-matrix-n*)
also from *assms* **have** $\dots = \text{mult-matrix-n } m \ \text{fmul} \ \text{fadd} \ A \ B$
by (*simp add: mult-matrix-n[THEN sym]*)
finally show $\text{mult-matrix-n } n \ \text{fmul} \ \text{fadd} \ A \ B = \text{mult-matrix-n } m \ \text{fmul} \ \text{fadd} \ A \ B$

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 (*intro* *ext* *arg-cong*[of *concl*: *Abs-matrix*])

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 (intro ext arg-cong[of concl: Abs-matrix])
      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
  end

```

qed
qed

instantiation *matrix* :: (zero) zero
begin

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

instance ..

end

lemma *Rep-zero-matrix-def[simp]*: $\text{Rep-matrix } 0 \ j \ i = 0$
by (*simp add: RepAbs-matrix zero-matrix-def*)

lemma *zero-matrix-def-nrows[simp]*: $\text{nrows } 0 = 0$
using *nrows-le* **by** *force*

lemma *zero-matrix-def-ncols[simp]*: $\text{ncols } 0 = 0$
using *ncols-le* **by** *fastforce*

lemma *combine-matrix-zero-l-neutral*: $\text{zero-l-neutral } f \implies \text{zero-l-neutral } (\text{combine-matrix } f)$
by (*simp add: zero-l-neutral-def combine-matrix-def combine-infmatrix-def*)

lemma *combine-matrix-zero-r-neutral*: $\text{zero-r-neutral } f \implies \text{zero-r-neutral } (\text{combine-matrix } f)$
by (*simp add: zero-r-neutral-def combine-matrix-def combine-infmatrix-def*)

lemma *mult-matrix-zero-closed*: $\llbracket \text{fadd } 0 \ 0 = 0; \text{zero-closed } \text{fmul} \rrbracket \implies \text{zero-closed}$
(*mult-matrix fmul fadd*)
apply (*simp add: zero-closed-def mult-matrix-def mult-matrix-n-def*)
by (*simp add: foldseq-zero 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
by (*simp add: RepAbs-matrix foldseq-zero matrix-eqI mult-matrix-n-def*)

lemma *mult-matrix-n-zero-left[simp]*: $\llbracket \text{fadd } 0 \ 0 = 0; \forall a. \text{fmul } 0 \ a = 0 \rrbracket \implies$
mult-matrix-n n fmul fadd 0 A = 0
by (*simp add: RepAbs-matrix foldseq-zero matrix-eqI mult-matrix-n-def*)

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

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

lemma *apply-matrix-zero*[simp]: $f\ 0 = 0 \implies \text{apply-matrix}\ f\ 0 = 0$
by (*simp add: matrix-eqI*)

lemma *combine-matrix-zero*: $f\ 0\ 0 = 0 \implies \text{combine-matrix}\ f\ 0\ 0 = 0$
by (*simp add: matrix-eqI*)

lemma *transpose-matrix-zero*[simp]: $\text{transpose-matrix}\ 0 = 0$
by (*simp add: matrix-eqI*)

lemma *apply-zero-matrix-def*[simp]: $\text{apply-matrix}\ (\lambda x. 0)\ A = 0$
by (*simp add: matrix-eqI*)

definition *singleton-matrix* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('a::\text{zero}) \Rightarrow 'a\ \text{matrix}$ **where**
singleton-matrix $j\ i\ a == \text{Abs-matrix}(\lambda m\ n. \text{if } j = m \wedge i = n \text{ then } a \text{ else } 0)$

definition *move-matrix* :: $('a::\text{zero})\ \text{matrix} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow 'a\ \text{matrix}$ **where**
move-matrix $A\ y\ x == \text{Abs-matrix}(\lambda j\ i. \text{if } (((\text{int } j) - y) < 0) \mid (((\text{int } i) - x) < 0) \text{ then } 0 \text{ else } \text{Rep-matrix}\ A\ (\text{nat } ((\text{int } j) - y))\ (\text{nat } ((\text{int } i) - x)))$

definition *take-rows* :: $('a::\text{zero})\ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a\ \text{matrix}$ **where**
take-rows $A\ r == \text{Abs-matrix}(\lambda j\ i. \text{if } (j < r) \text{ then } (\text{Rep-matrix}\ A\ j\ i) \text{ else } 0)$

definition *take-columns* :: $('a::\text{zero})\ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a\ \text{matrix}$ **where**
take-columns $A\ c == \text{Abs-matrix}(\lambda j\ i. \text{if } (i < c) \text{ then } (\text{Rep-matrix}\ A\ j\ i) \text{ else } 0)$

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* :: $('a::\text{zero})\ \text{matrix} \Rightarrow \text{nat} \Rightarrow 'a\ \text{matrix}$ **where**
row-of-matrix $A\ m == \text{take-rows}\ (\text{move-matrix}\ A\ (-\ \text{int } m)\ 0)\ 1$

lemma *Rep-singleton-matrix*[simp]: $\text{Rep-matrix}\ (\text{singleton-matrix}\ j\ i\ e)\ m\ n = (\text{if } j = m \wedge i = n \text{ then } e \text{ else } 0)$
unfolding *singleton-matrix-def*
by (*smt (verit, del-insts) RepAbs-matrix Suc-n-not-le-n*)

lemma *apply-singleton-matrix*[simp]: $f\ 0 = 0 \implies \text{apply-matrix}\ f\ (\text{singleton-matrix}\ j\ i\ x) = (\text{singleton-matrix}\ j\ i\ (f\ x))$
by (*simp add: matrix-eqI*)

lemma *singleton-matrix-zero*[simp]: $\text{singleton-matrix}\ j\ i\ 0 = 0$
by (*simp add: singleton-matrix-def zero-matrix-def*)

lemma *nrows-singleton*[simp]: $\text{nrows}(\text{singleton-matrix}\ j\ i\ e) = (\text{if } e = 0 \text{ then } 0 \text{ else } \text{Suc } j)$

proof –

have $e \neq 0 \implies \text{Suc } j \leq \text{nrows}(\text{singleton-matrix}\ j\ i\ e)$
by (*metis Rep-singleton-matrix not-less-eq-eq nrows*)

then show *?thesis*
by (*simp add: le-antisym nrows-le*)
qed

lemma *ncols-singleton[simp]*: $\text{ncols}(\text{singleton-matrix } j \ i \ e) = (\text{if } e = 0 \text{ then } 0 \text{ else } \text{Suc } i)$
by (*simp add: Suc-leI le-antisym ncols-le ncols-notzero*)

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 (*intro ext arg-cong[of concl: Abs-matrix]*)
by (*metis Rep-singleton-matrix singleton-matrix-def*)

lemma *transpose-singleton[simp]*: $\text{transpose-matrix} \ (\text{singleton-matrix } j \ i \ a) = \text{singleton-matrix } i \ j \ a$
by (*simp add: matrix-eqI*)

lemma *Rep-move-matrix[simp]*:
 $\text{Rep-matrix} \ (\text{move-matrix } A \ y \ x) \ j \ i =$
 $(\text{if } (((\text{int } j) - y) < 0) \mid (((\text{int } i) - x) < 0) \text{ then } 0 \text{ else } \text{Rep-matrix } A \ (\text{nat}((\text{int } j) - y)) \ (\text{nat}((\text{int } i) - x)))$
apply (*simp add: move-matrix-def*)
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*: $\text{move-matrix } A \ j \ i = \text{move-matrix} \ (\text{move-matrix } A \ j \ 0) \ 0 \ i$
by (*simp add: matrix-eqI*)

lemma *transpose-move-matrix[simp]*:
 $\text{transpose-matrix} \ (\text{move-matrix } A \ x \ y) = \text{move-matrix} \ (\text{transpose-matrix } A) \ y \ x$
by (*simp add: matrix-eqI*)

lemma *move-matrix-singleton[simp]*: $\text{move-matrix} \ (\text{singleton-matrix } u \ v \ x) \ j \ i =$
 $(\text{if } (j + \text{int } u < 0) \mid (i + \text{int } v < 0) \text{ then } 0 \text{ else } (\text{singleton-matrix } (\text{nat } (j + \text{int } u)) \ (\text{nat } (i + \text{int } v)) \ x))$
by (*auto intro!: matrix-eqI split: if-split-asm*)

lemma *Rep-take-columns[simp]*:
 $\text{Rep-matrix} \ (\text{take-columns } A \ c) \ j \ i = (\text{if } i < c \text{ then } (\text{Rep-matrix } A \ j \ i) \text{ else } 0)$
unfolding *take-columns-def*
by (*smt (verit, best) RepAbs-matrix leD nrows*)

lemma *Rep-take-rows[simp]*:

$Rep\text{-}matrix\ (take\text{-}rows\ A\ r)\ j\ i = (if\ j < r\ then\ (Rep\text{-}matrix\ A\ j\ i)\ else\ 0)$
unfolding *take-rows-def*
by (*smt* (*verit*, *best*) *RepAbs-matrix leD ncols*)

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

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

lemma *column-of-matrix: ncols A ≤ n ⇒ column-of-matrix A n = 0*
by (*simp add: matrix-eqI ncols*)

lemma *row-of-matrix: nrow A ≤ n ⇒ row-of-matrix A n = 0*
by (*simp add: matrix-eqI nrow*)

lemma *mult-matrix-singleton-right[simp]*:
assumes $\forall x. fmul\ x\ 0 = 0\ \forall x. fmul\ 0\ x = 0\ \forall x. fadd\ 0\ x = x\ \forall x. fadd\ x\ 0 = x$
shows $(mult\text{-}matrix\ fmul\ fadd\ A\ (singleton\text{-}matrix\ j\ i\ e)) = apply\text{-}matrix\ (\lambda x. fmul\ x\ e)\ (move\text{-}matrix\ (column\text{-}of\text{-}matrix\ A\ j)\ 0\ (int\ i))$
using *assms*
unfolding *mult-matrix-def*
apply (*subst mult-matrix-nm[of - - max (ncols A) (Suc j)]*;
simp add: mult-matrix-n-def apply-matrix-def apply-infmatrix-def)
apply (*intro ext arg-cong[of concl: Abs-matrix]*)
by (*simp add: max-def assms foldseq-almostzero[of - j]*)

lemma *mult-matrix-ext*:
assumes
eprem:
 $\exists e. (\forall a\ b. a \neq b \longrightarrow fmul\ a\ e \neq fmul\ b\ e)$
and *fpreds*:
 $\forall a. fmul\ 0\ a = 0$
 $\forall a. fmul\ a\ 0 = 0$
 $\forall a. fadd\ a\ 0 = a$
 $\forall a. fadd\ 0\ a = a$
and *contrapreds*: $mult\text{-}matrix\ fmul\ fadd\ A = mult\text{-}matrix\ fmul\ fadd\ B$
shows $A = B$
proof(*rule ccontr*)
assume $A \neq B$
then obtain $J\ I$ **where** *ne*: $(Rep\text{-}matrix\ A\ J\ I) \neq (Rep\text{-}matrix\ B\ J\ I)$
by (*meson matrix-eqI*)
from *eprem* **obtain** e **where** *eprops*: $(\forall a\ b. a \neq b \longrightarrow fmul\ a\ e \neq fmul\ b\ e)$ **by**
blast
let $?S = singleton\text{-}matrix\ I\ 0\ e$
let $?comp = mult\text{-}matrix\ fmul\ fadd$

```

have d: !!x f g. f = g  $\implies$  f x = g x by blast
have e: ( $\lambda x$ . fmul x e) 0 = 0 by (simp add: assms)
have Rep-matrix (apply-matrix ( $\lambda x$ . fmul x e) (column-of-matrix A I))  $\neq$ 
  Rep-matrix (apply-matrix ( $\lambda x$ . fmul x e) (column-of-matrix B I))
using fprems
by (metis Rep-apply-matrix Rep-column-of-matrix eprops ne)
then have ?comp A ?S  $\neq$  ?comp B ?S
by (simp add: fprems eprops Rep-matrix-inject)
with contraprems show False by simp
qed

definition foldmatrix :: ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  ('a infmatrix)  $\Rightarrow$ 
  nat  $\Rightarrow$  nat  $\Rightarrow$  'a where
  foldmatrix f g A m n == foldseq-transposed g ( $\lambda j$ . foldseq f (A j) n) m

definition foldmatrix-transposed :: ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  ('a  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  ('a
  infmatrix)  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a where
  foldmatrix-transposed f g A m n == foldseq g ( $\lambda j$ . foldseq-transposed f (A j) n)
  m

lemma foldmatrix-transpose:
  assumes  $\forall a b c d. g(f a b) (f c d) = f (g a c) (g b d)$ 
  shows foldmatrix f g A m n = foldmatrix-transposed g f (transpose-infmatrix A)
  n m
proof –
  have forall: $\bigwedge P x. (\forall x. P x) \implies P x$  by auto
  have tworows: $\forall A. \text{foldmatrix } f g A 1 n = \text{foldmatrix-transposed } g f (\text{transpose-infmatrix } A) n 1$ 
  proof (induct n)
  case 0
  then show ?case
  by (simp add: foldmatrix-def foldmatrix-transposed-def)
  next
  case (Suc n)
  then show ?case
  apply (clarsimp simp: foldmatrix-def foldmatrix-transposed-def assms)
  apply (rule arg-cong [of concl: f -])
  by meson
qed
have foldseq-transposed g ( $\lambda j$ . foldseq f (A j) n) m =
  foldseq f ( $\lambda j$ . foldseq-transposed g (transpose-infmatrix A j) m) n
proof (induct m)
  case 0
  then show ?case by auto
  next
  case (Suc m)
  then show ?case
  using tworows
  apply (drule-tac x= $\lambda j$  i. (if j = 0 then (foldseq-transposed g ( $\lambda u$ . A u i) m)

```

```

else (A (Suc m) i) in spec)
  by (simp add: Suc foldmatrix-def foldmatrix-transposed-def)
qed
then show foldmatrix f g A m n = foldmatrix-transposed g f (transpose-infmatrix
A) n m
  by (simp add: foldmatrix-def foldmatrix-transposed-def)
qed

```

lemma *foldseq-foldseq*:

assumes *associative f associative g* $\forall a\ b\ c\ d. g(f\ a\ b)\ (f\ c\ d) = f\ (g\ a\ c)\ (g\ b\ d)$

shows

$foldseq\ g\ (\lambda j. foldseq\ f\ (A\ j)\ n)\ m = foldseq\ f\ (\lambda j. foldseq\ g\ ((transpose-infmatrix\ A)\ j)\ m)\ n$

using *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 $\forall a. fmul\ 0\ a = 0 \ \forall a. fmul\ a\ 0 = 0\ fadd\ 0\ 0 = 0$

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

unfolding *nrows-le 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 $\forall a. fmul\ 0\ a = 0 \ \forall a. fmul\ a\ 0 = 0\ fadd\ 0\ 0 = 0$

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

unfolding *ncols-le 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

$\forall a. fmul\ 0\ a = 0$

$\forall a. fmul\ a\ 0 = 0$

$fadd\ 0\ 0 = 0$

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

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

lemma *mult-ncols*:

assumes

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

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

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

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

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

lemma *nrows-move-matrix-le*: $\text{nrows } (\text{move-matrix } A \ j \ i) \leq \text{nat}((\text{int } (\text{nrows } A)) + j)$

by (*smt (verit) Rep-move-matrix int-nat-eq nrows nrows-le of-nat-le-iff*)

lemma *ncols-move-matrix-le*: $\text{ncols } (\text{move-matrix } A \ j \ i) \leq \text{nat}((\text{int } (\text{ncols } A)) + i)$

by (*metis nrows-move-matrix-le nrows-transpose transpose-move-matrix*)

lemma *mult-matrix-assoc*:

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 $\text{mult-matrix } \text{fmul2 } \text{fadd2 } (\text{mult-matrix } \text{fmul1 } \text{fadd1 } A \ B) \ C = \text{mult-matrix } \text{fmul1 } \text{fadd1 } A \ (\text{mult-matrix } \text{fmul2 } \text{fadd2 } B \ C)$

proof –

have *comb-left*: $!! \ A \ B \ x \ y. \ A = B \implies (\text{Rep-matrix } (\text{Abs-matrix } A)) \ x \ y = (\text{Rep-matrix } (\text{Abs-matrix } B)) \ x \ y$ **by** *blast*

have *fmul2fadd1fold*: $!! \ x \ s \ n. \ \text{fmul2 } (\text{foldseq } \text{fadd1 } s \ n) \ x = \text{foldseq } \text{fadd1 } (\lambda k. \text{fmul2 } (s \ k) \ x) \ n$

by (*rule-tac g1 = $\lambda y. \text{fmul2 } y \ x$ in ssust [OF foldseq-distr-unary], insert assms, simp-all*)

have *fmul1fadd2fold*: $!! \ x \ s \ n. \ \text{fmul1 } x \ (\text{foldseq } \text{fadd2 } s \ n) = \text{foldseq } \text{fadd2 } (\lambda k. \text{fmul1 } x \ (s \ k)) \ n$

using *assms* **by** (*rule-tac g1 = $\lambda y. \text{fmul1 } x \ y$ in ssust [OF foldseq-distr-unary], simp-all*)

let $?N = \max (\text{ncols } A) (\max (\text{ncols } B) (\max (\text{nrows } B) (\text{nrows } C)))$

show *?thesis*

apply (*intro matrix-eqI*)

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-cols mult-n-rows 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-cols mult-n-rows assms)+
    apply (simplesubst mult-matrix-nm[of - - - ?N])
    apply (simp add: max1 max2 mult-n-cols mult-n-rows assms)+
    apply (simplesubst mult-matrix-nm[of - - - ?N])
    apply (simp add: max1 max2 mult-n-cols mult-n-rows assms)+
    apply (simplesubst mult-matrix-nm[of - - - ?N])
    apply (simp add: max1 max2 mult-n-cols mult-n-rows assms)+
    apply (simplesubst mult-matrix-nm[of - - - ?N])
    apply (simp add: max1 max2 mult-n-cols mult-n-rows 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 *mult-matrix-assoc-simple*:

```

  assumes
     $\forall a. \text{fmul } 0 \ a = 0$ 
     $\forall a. \text{fmul } a \ 0 = 0$ 
    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)
  by (smt (verit) assms associative-def commutative-def distributive-def l-distributive-def
mult-matrix-assoc r-distributive-def)

```

lemma *transpose-apply-matrix*: $f \ 0 = 0 \implies \text{transpose-matrix } (\text{apply-matrix } f \ A)$
 $= \text{apply-matrix } f \ (\text{transpose-matrix } A)$
 by (simp add: matrix-eqI)

lemma *transpose-combine-matrix*: $f\ 0\ 0 = 0 \implies \text{transpose-matrix } (\text{combine-matrix } f\ A\ B) = \text{combine-matrix } f\ (\text{transpose-matrix } A)\ (\text{transpose-matrix } B)$
by (*simp add: matrix-eqI*)

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
 $\text{Rep-matrix}(\text{mult-matrix } \text{fmul } \text{fadd } A\ B)\ j\ i =$
 $\text{foldseq } \text{fadd } (\lambda k. \text{fmul } (\text{Rep-matrix } A\ j\ k)\ (\text{Rep-matrix } B\ k\ i))\ (\text{max } (\text{ncols } A)$
 $(\text{nrows } B))$
using *assms*
apply (*simp add: mult-matrix-def mult-matrix-n-def*)
apply (*subst RepAbs-matrix*)
apply (*rule exI[of - nrows A], simp add: nrows foldseq-zero*)
apply (*rule exI[of - ncols B], simp add: ncols foldseq-zero*)
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
 $\text{transpose-matrix } (\text{mult-matrix } \text{fmul } \text{fadd } A\ B) = \text{mult-matrix } \text{fmul } \text{fadd } (\text{transpose-matrix } B)\ (\text{transpose-matrix } A)$
using *assms*
by (*simp add: matrix-eqI Rep-mult-matrix ac-simps*)

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

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

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

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

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

instance ..

end

instance *matrix* :: (*{zero, order}*) *order*

proof

fix *x y z* :: '*a matrix*

assume $x \leq y \leq z$

show $x \leq z$

by (*meson* $\langle x \leq y \rangle \langle y \leq z \rangle$ *le-matrix-def order-trans*)

next

fix *x y* :: '*a matrix*

assume $x \leq y \leq x$

show $x = y$

by (*meson* $\langle x \leq y \rangle \langle y \leq x \rangle$ *le-matrix-def matrix-eqI order-antisym*)

qed (*auto simp: less-def le-matrix-def*)

lemma *le-apply-matrix*:

assumes

$f \ 0 = 0$

$\forall x y. x \leq y \longrightarrow f x \leq f y$

$(a :: ('a :: \{ord, zero\}) \text{ matrix}) \leq b$

shows $\text{apply-matrix } f \ a \leq \text{apply-matrix } f \ b$

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

lemma *le-combine-matrix*:

assumes

$f \ 0 \ 0 = 0$

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

$A \leq B$

$C \leq D$

shows $\text{combine-matrix } f \ A \ C \leq \text{combine-matrix } f \ B \ D$

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

lemma *le-left-combine-matrix*:

assumes

$f \ 0 \ 0 = 0$

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

$A \leq B$

shows $\text{combine-matrix } f \ C \ A \leq \text{combine-matrix } f \ C \ B$

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

lemma *le-right-combine-matrix*:

assumes

$f \ 0 \ 0 = 0$

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

$A \leq B$

shows $\text{combine-matrix } f \ A \ C \leq \text{combine-matrix } f \ B \ C$

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

lemma *le-transpose-matrix*: $(A \leq B) = (\text{transpose-matrix } A \leq \text{transpose-matrix } B)$

B)
by (*simp add: le-matrix-def, auto*)

lemma *le-foldseq*:

assumes
 $\forall a\ b\ c\ d. a \leq b \wedge c \leq d \longrightarrow f\ a\ c \leq f\ b\ d$
 $\forall i. i \leq n \longrightarrow s\ i \leq t\ i$
shows $foldseq\ f\ s\ n \leq foldseq\ f\ t\ n$
proof -
have $\forall s\ t. (\forall i. i \leq n \longrightarrow s\ i \leq t\ i) \longrightarrow foldseq\ f\ s\ n \leq foldseq\ f\ t\ n$
by (*induct n*) (*simp-all add: assms*)
then show $foldseq\ f\ s\ n \leq foldseq\ f\ t\ n$ **using** *assms* **by** *simp*
qed

lemma *le-left-mult*:

assumes
 $\forall a\ b\ c\ d. a \leq b \wedge c \leq d \longrightarrow fadd\ a\ c \leq fadd\ b\ d$
 $\forall c\ a\ b. 0 \leq c \wedge a \leq b \longrightarrow fmul\ c\ a \leq fmul\ c\ b$
 $\forall a. fmul\ 0\ a = 0$
 $\forall a. fmul\ a\ 0 = 0$
 $fadd\ 0\ 0 = 0$
 $0 \leq C$
 $A \leq B$
shows $mult-matrix\ fmul\ fadd\ C\ A \leq mult-matrix\ fmul\ fadd\ C\ B$
using *assms*
apply (*auto simp: le-matrix-def Rep-mult-matrix*)
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
 $\forall a\ b\ c\ d. a \leq b \wedge c \leq d \longrightarrow fadd\ a\ c \leq fadd\ b\ d$
 $\forall c\ a\ b. 0 \leq c \wedge a \leq b \longrightarrow fmul\ a\ c \leq fmul\ b\ c$
 $\forall a. fmul\ 0\ a = 0$
 $\forall a. fmul\ a\ 0 = 0$
 $fadd\ 0\ 0 = 0$
 $0 \leq C$
 $A \leq B$
shows $mult-matrix\ fmul\ fadd\ A\ C \leq mult-matrix\ fmul\ fadd\ B\ C$
using *assms*
apply (*auto simp: le-matrix-def Rep-mult-matrix*)
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 *singleton-matrix-le[simp]*: $(\text{singleton-matrix } j\ i\ a \leq \text{singleton-matrix } j\ i\ b)$
 $= (a \leq (b:::\text{order}))$
by (*auto simp: le-matrix-def*)

lemma *singleton-le-zero[simp]*: $(\text{singleton-matrix } j\ i\ x \leq 0) = (x \leq (0::'\text{a}::\{\text{order}, \text{zero}\}))$
by (*metis singleton-matrix-le singleton-matrix-zero*)

lemma *singleton-ge-zero[simp]*: $(0 \leq \text{singleton-matrix } j\ i\ x) = ((0::'\text{a}::\{\text{order}, \text{zero}\}) \leq x)$
by (*metis singleton-matrix-le singleton-matrix-zero*)

lemma *move-matrix-le-zero[simp]*:
fixes $A::'\text{a}::\{\text{order}, \text{zero}\}$ *matrix*
assumes $0 \leq j\ 0 \leq i$
shows $(\text{move-matrix } A\ j\ i \leq 0) = (A \leq 0)$
proof –
have $\text{Rep-matrix } A\ j'\ i' \leq 0$
if $\forall n\ m. \neg \text{int } n < j \wedge \neg \text{int } m < i \longrightarrow \text{Rep-matrix } A\ (\text{nat } (\text{int } n - j))\ (\text{nat } (\text{int } m - i)) \leq 0$
for $j'\ i'$
using *that[rule-format, of j' + nat j i' + nat i]* **by** (*simp add: assms*)
then show *?thesis*
by (*auto simp: le-matrix-def*)
qed

lemma *move-matrix-zero-le[simp]*:
fixes $A::'\text{a}::\{\text{order}, \text{zero}\}$ *matrix*
assumes $0 \leq j\ 0 \leq i$
shows $(0 \leq \text{move-matrix } A\ j\ i) = (0 \leq A)$
proof –
have $0 \leq \text{Rep-matrix } A\ j'\ i'$
if $\forall n\ m. \neg \text{int } n < j \wedge \neg \text{int } m < i \longrightarrow 0 \leq \text{Rep-matrix } A\ (\text{nat } (\text{int } n - j))\ (\text{nat } (\text{int } m - i))$
for $j'\ i'$
using *that[rule-format, of j' + nat j i' + nat i]* **by** (*simp add: assms*)
then show *?thesis*
by (*auto simp: le-matrix-def*)
qed

lemma *move-matrix-le-move-matrix-iff[simp]*:
fixes $A::'\text{a}::\{\text{order}, \text{zero}\}$ *matrix*
assumes $0 \leq j\ 0 \leq i$
shows $(\text{move-matrix } A\ j\ i \leq \text{move-matrix } B\ j\ i) = (A \leq B)$
proof –
have $\text{Rep-matrix } A\ j'\ i' \leq \text{Rep-matrix } B\ j'\ i'$
if $\forall n\ m. \neg \text{int } n < j \wedge \neg \text{int } m < i \longrightarrow \text{Rep-matrix } A\ (\text{nat } (\text{int } n - j))\ (\text{nat } (\text{int } m - i)) \leq \text{Rep-matrix } B\ (\text{nat } (\text{int } n - j))\ (\text{nat } (\text{int } m - i))$
for $j'\ i'$
using *that[rule-format, of j' + nat j i' + nat i]* **by** (*simp add: assms*)
then show *?thesis*
by (*auto simp: le-matrix-def*)
qed

```

(int m - i)) ≤ Rep-matrix B (nat (int n - j)) (nat (int m - i))
  for j' i'
    using that[rule-format, of j' + nat j i' + nat i] by (simp add: assms)
  then show ?thesis
    by (auto simp: le-matrix-def)
qed

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

definition inf = combine-matrix inf

definition sup = combine-matrix sup

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

end

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

definition
  plus-matrix-def: A + B = combine-matrix (+) A B

instance ..

end

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

definition
  minus-matrix-def: - A = apply-matrix uminus A

instance ..

end

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

definition
  diff-matrix-def: A - B = 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 $A + B + C = A + (B + C)$
 by (*simp add: add.assoc matrix-eqI plus-matrix-def*)
show $0 + A = A$
 by (*simp add: matrix-eqI plus-matrix-def*)
show $A + 0 = A$
 by (*simp add: matrix-eqI plus-matrix-def*)

qed

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

proof

fix *A B* :: *'a matrix*
show $A + B = B + A$
 by (*simp add: add.commute matrix-eqI plus-matrix-def*)
show $0 + A = A$
 by (*simp add: plus-matrix-def matrix-eqI*)

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 matrix-eqI*)
show $A + - B = A - B$
 by (*simp add: plus-matrix-def diff-matrix-def minus-matrix-def matrix-eqI*)

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 matrix-eqI)
  show  $A - B = A + - B$ 
    by (simp add: plus-matrix-def diff-matrix-def minus-matrix-def matrix-eqI)
qed

instance matrix :: (ordered-ab-group-add) ordered-ab-group-add
proof
  fix A B C :: 'a matrix
  assume  $A \leq B$ 
  then show  $C + A \leq C + B$ 
    by (simp add: le-matrix-def plus-matrix-def)
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  $A * B * C = A * (B * C)$ 
    unfolding times-matrix-def
    by (smt (verit, best) add.assoc associative-def distrib-left distrib-right group-cancel.add2
mult.assoc mult-matrix-assoc mult-not-zero)
  show  $(A + B) * C = A * C + B * C$ 
    unfolding times-matrix-def plus-matrix-def
    using l-distributive-matrix
    by (metis (full-types) add.assoc add.commute associative-def commutative-def
distrib-right l-distributive-def mult-not-zero)
  show  $A * (B + C) = A * B + A * C$ 
    unfolding times-matrix-def plus-matrix-def
    using r-distributive-matrix
    by (metis (no-types, lifting) add.assoc add.commute associative-def commuta-
tive-def distrib-left mult-zero-left mult-zero-right r-distributive-def)
qed (auto simp: times-matrix-def)

instance matrix :: (ring) ring ..

instance matrix :: (ordered-ring) ordered-ring
proof
  fix A B C :: 'a matrix
  assume  $\S: A \leq B \ 0 \leq C$ 
  from  $\S$  show  $C * A \leq C * B$ 
    by (simp add: times-matrix-def add-mono le-left-mult mult-left-mono)
  from  $\S$  show  $A * C \leq B * C$ 
    by (simp add: times-matrix-def add-mono le-right-mult mult-right-mono)

```

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

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

proof

show $\bigwedge a:: 'a \text{ matrix. } |a| = \sup a \ (-a)$

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 (+) ($\lambda k. \text{ (Rep-matrix } a \text{ } j \text{ } k) * \text{ (Rep-matrix } b \text{ } k \text{ } i) \text{ (max (ncols } a) \text{ (nrows } b))}$)

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

lemma *apply-matrix-add*: $\forall x y. f \ (x+y) = (f \ x) + (f \ y) \implies f \ 0 = (0::'a)$

$\implies \text{apply-matrix } f \ ((a::('a::\text{monoid-add}) \text{ matrix}) + b) = (\text{apply-matrix } f \ a) + (\text{apply-matrix } f \ b)$

by (*simp add: matrix-eqI*)

lemma *singleton-matrix-add*: *singleton-matrix* *j i* ((*a*::(*monoid-add*)+*b*) = (*singleton-matrix* *j i a*) + (*singleton-matrix* *j i b*)

by (*simp add: matrix-eqI*)

lemma *nrows-mult*: *nrows* ((*A*::('*a*::*semiring-0*) *matrix*) * *B*) \leq *nrows* *A*

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

lemma *ncols-mult*: *ncols* ((*A*::('*a*::*semiring-0*) *matrix*) * *B*) \leq *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* ($\lambda j \ i. \text{ if } j = i \wedge j < n \text{ then } 1 \text{ else } 0$)

lemma *Rep-one-matrix[simp]*: *Rep-matrix* (*one-matrix* *n*) *j i* = (if (*j* = *i* \wedge *j* < *n*) then 1 else 0)

unfolding *one-matrix-def*

by (*smt (verit, del-insts) RepAbs-matrix not-le*)

```

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-matrix n) :: ('a::zero-neq-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]:
  fixes A :: ('a::semiring-1) matrix
  shows ncols A ≤ n ⇒ A * (one-matrix n) = A
  apply (intro matrix-eqI)
  apply (simp add: times-matrix-def Rep-mult-matrix)
  apply (subst foldseq-almostzero, auto simp: ncols)
  done

lemma one-matrix-mult-left[simp]:
  fixes A :: ('a::semiring-1) matrix
  shows nrows A ≤ n ⇒ (one-matrix n) * A = A
  apply (intro matrix-eqI)
  apply (simp add: times-matrix-def Rep-mult-matrix)
  apply (subst foldseq-almostzero, auto simp: nrows)
  done

lemma transpose-matrix-mult:
  fixes A :: ('a::comm-ring) matrix
  shows transpose-matrix (A*B) = (transpose-matrix B) * (transpose-matrix A)
  by (simp add: times-matrix-def transpose-mult-matrix mult.commute)

lemma transpose-matrix-add:
  fixes A :: ('a::monoid-add) matrix
  shows transpose-matrix (A+B) = transpose-matrix A + transpose-matrix B
  by (simp add: plus-matrix-def transpose-combine-matrix)

lemma transpose-matrix-diff:
  fixes A :: ('a::group-add) matrix
  shows transpose-matrix (A-B) = transpose-matrix A - transpose-matrix B
  by (simp add: diff-matrix-def transpose-combine-matrix)

lemma transpose-matrix-minus:
  fixes A :: ('a::group-add) matrix

```

shows $\text{transpose-matrix } (-A) = - \text{transpose-matrix } (A :: 'a \text{ matrix})$
by (*simp add: minus-matrix-def transpose-apply-matrix*)

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

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

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

lemma $\text{right-inverse-matrix-dim}$: $\text{right-inverse-matrix } A \ X \implies \text{nrows } A = \text{ncols } X$
using $\text{ncols-mult[of } A \ X] \ \text{nrows-mult[of } A \ X]$
by (*simp add: right-inverse-matrix-def*)

lemma $\text{left-inverse-matrix-dim}$: $\text{left-inverse-matrix } A \ Y \implies \text{ncols } A = \text{nrows } Y$
using $\text{ncols-mult[of } Y \ A] \ \text{nrows-mult[of } Y \ A]$
by (*simp add: left-inverse-matrix-def*)

lemma $\text{left-right-inverse-matrix-unique}$:
assumes $\text{left-inverse-matrix } A \ Y \ \text{right-inverse-matrix } A \ X$
shows $X = Y$
proof –
have $Y = Y * \text{one-matrix } (\text{nrows } A)$
by (*metis assms(1) left-inverse-matrix-def one-matrix-mult-right*)
also have $\dots = Y * (A * X)$
by (*metis assms(2) max.idem right-inverse-matrix-def right-inverse-matrix-dim*)

also have $\dots = (Y * A) * X$ **by** (*simp add: mult.assoc*)
also have $\dots = X$
using $\text{assms left-inverse-matrix-def right-inverse-matrix-def}$
by (*metis left-inverse-matrix-dim max.idem one-matrix-mult-left*)
ultimately show $X = Y$ **by** (*simp*)
qed

lemma $\text{inverse-matrix-inject}$: $\llbracket \text{inverse-matrix } A \ X; \text{inverse-matrix } A \ Y \rrbracket \implies X = Y$
by (*auto simp: inverse-matrix-def left-right-inverse-matrix-unique*)

lemma $\text{one-matrix-inverse}$: $\text{inverse-matrix } (\text{one-matrix } n) (\text{one-matrix } n)$
by (*simp add: inverse-matrix-def left-inverse-matrix-def right-inverse-matrix-def*)

lemma $\text{zero-imp-mult-zero}$: $(a :: 'a :: \text{semiring-0}) = 0 \mid b = 0 \implies a * b = 0$
by *auto*

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

$\forall j \ i \ k. (Rep\text{-}matrix \ A \ j \ k = 0) \mid (Rep\text{-}matrix \ B \ k \ i) = 0 \implies A * B =$
 $(0 :: ('a :: lattice\text{-}ring) \ matrix)$
by (*simp add: matrix-eqI Rep-matrix-mult foldseq-zero zero-imp-mult-zero*)

lemma *add-nrows*: $nrows \ (A :: ('a :: monoid\text{-}add) \ matrix) \leq u \implies nrows \ B \leq u \implies$
 $nrows \ (A + B) \leq u$
by (*simp add: nrows-le*)

lemma *move-matrix-row-mult*:

fixes $A :: ('a :: semiring\text{-}0) \ matrix$
shows $move\text{-}matrix \ (A * B) \ j \ 0 = (move\text{-}matrix \ A \ j \ 0) * B$
proof –
have $\bigwedge m. \neg int \ m < j \implies ncols \ (move\text{-}matrix \ A \ j \ 0) \leq max \ (ncols \ A) \ (nrows \ B)$
by (*smt (verit, best) max1 nat-int ncols-move-matrix-le*)
then show *?thesis*
apply (*intro matrix-eqI*)
apply (*auto simp: Rep-matrix-mult foldseq-zero*)
apply (*rule-tac foldseq-zerotail[symmetric]*)
apply (*auto simp: nrows zero-imp-mult-zero max2*)
done
qed

lemma *move-matrix-col-mult*:

fixes $A :: ('a :: semiring\text{-}0) \ matrix$
shows $move\text{-}matrix \ (A * B) \ 0 \ i = A * (move\text{-}matrix \ B \ 0 \ i)$
proof –
have $\bigwedge n. \neg int \ n < i \implies nrows \ (move\text{-}matrix \ B \ 0 \ i) \leq max \ (ncols \ A) \ (nrows \ B)$
by (*smt (verit, del-insts) max2 nat-int nrows-move-matrix-le*)
then show *?thesis*
apply (*intro matrix-eqI*)
apply (*auto simp: Rep-matrix-mult foldseq-zero*)
apply (*rule-tac foldseq-zerotail[symmetric]*)
apply (*auto simp: ncols zero-imp-mult-zero max1*)
done
qed

lemma *move-matrix-add*: $((move\text{-}matrix \ (A + B) \ j \ i) :: ('a :: monoid\text{-}add) \ matrix))$
 $= (move\text{-}matrix \ A \ j \ i) + (move\text{-}matrix \ B \ j \ i)$
by (*simp add: matrix-eqI*)

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

definition *scalar-mult* :: $('a :: ring) \Rightarrow 'a \ matrix \Rightarrow 'a \ matrix$ **where**
 $scalar\text{-}mult \ a \ m == apply\text{-}matrix \ ((* \ a) \ m)$

```

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)
  by (simp add: scalar-mult-def)

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 smat = 'a svec svec

definition sparse-row-vector :: ('a::ab-group-add) svec  $\Rightarrow$  '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) smat  $\Rightarrow$  '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)

```

```

lemma [code]:
  ⟨0 = sparse-row-vector []⟩
  by simp

lemma foldl-distrstart: ∀ a x y. (f (g x y) a = g x (f y a)) ⇒ (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)
  by (induct arr) (auto simp: foldl-distrstart sparse-row-vector-def)

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)
  by (induct x) (auto simp: add-nrows)

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
  by (induct arr) (auto simp: foldl-distrstart sparse-row-matrix-def)

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

fun sorted-spvec :: 'a spvec ⇒ bool
where
  sorted-spvec [] = True
| sorted-spvec-step1: sorted-spvec [a] = True
| sorted-spvec-step: sorted-spvec ((m,x)#(n,y)#bs) = ((m < n) ∧ (sorted-spvec
((n,y)#bs)))

primrec sorted-spmat :: 'a spat ⇒ 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) ⇒ sorted-spvec as
  using sorted-spvec.elims(2) sorted-spvec-empty by blast

```

lemma *sorted-spvec-cons2*: $\text{sorted-spvec } (a\#b\#t) \implies \text{sorted-spvec } (a\#t)$
by (*smt* (*verit*, *del-insts*) *sorted-spvec-step order.strict-trans list.inject sorted-spvec.elims*(3) *surj-pair*)

lemma *sorted-spvec-cons3*: $\text{sorted-spvec}(a\#b\#t) \implies \text{fst } a < \text{fst } b$
by (*metis sorted-spvec-step prod.collapse*)

lemma *sorted-sparse-row-vector-zero*:
assumes $m \leq n$
shows $\text{sorted-spvec } ((n,a)\#\text{arr}) \implies \text{Rep-matrix } (\text{sparse-row-vector } \text{arr}) \ j \ m = 0$
proof (*induct arr*)
case *Nil*
then show ?*case* **by** *auto*
next
case (*Cons a arr*)
with *assms* **show** ?*case*
by (*auto dest: sorted-spvec-cons2 sorted-spvec-cons3*)
qed

lemma *sorted-sparse-row-matrix-zero*[*rule-format*]:
assumes $m \leq n$
shows $\text{sorted-spvec } ((n,a)\#\text{arr}) \implies \text{Rep-matrix } (\text{sparse-row-matrix } \text{arr}) \ m \ j = 0$
proof (*induct arr*)
case *Nil*
then show ?*case* **by** *auto*
next
case (*Cons a arr*)
with *assms* **show** ?*case*
unfolding *sparse-row-matrix-cons*
by (*auto dest: sorted-spvec-cons2 sorted-spvec-cons3*)
qed

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

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

lemma *sparse-row-vector-minus*:
 $\text{sparse-row-vector } (\text{minus-spvec } v) = - (\text{sparse-row-vector } v)$
proof (*induct v*)
case *Nil*
then show ?*case*

```

    by auto
next
  case (Cons a v)
  then have singleton-matrix 0 (fst a) (- snd a) = - singleton-matrix 0 (fst a)
    (snd a)
    by (simp add: Rep-matrix-inject minus-matrix-def)
  then show ?case
    by (simp add: local.Cons)
qed

```

lemma *sparse-row-vector-abs*:

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

proof (induct v)

case Nil

then show ?case

by simp

next

case (Cons ab v)

then have v: sorted-spvec v

using sorted-spvec-cons1 by blast

show ?case

proof (cases ab)

case (Pair a b)

then have 0: Rep-matrix (sparse-row-vector v) 0 a = 0

using Cons.premis sorted-sparse-row-vector-zero by blast

with v Cons show ?thesis

by (fastforce simp: Pair simp flip: Rep-matrix-inject)

qed

qed

lemma *sorted-spvec-minus-spvec*:

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

by (induct v rule: sorted-spvec.induct) (auto simp: sorted-spvec-step1 sorted-spvec-step)

lemma *sorted-spvec-abs-spvec*:

$\text{sorted-spvec } v \implies \text{sorted-spvec } (\text{abs-spvec } v)$

by (induct v rule: sorted-spvec.induct) (auto simp: sorted-spvec-step1 sorted-spvec-step)

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

lemma *smult-spvec-empty*[simp]: $\text{smult-spvec } y [] = []$

by (simp add: smult-spvec-def)

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

by (simp add: smult-spvec-def)

fun *addmult-spvec* :: $('a::\text{ring}) \Rightarrow 'a \text{ spvec} \Rightarrow 'a \text{ spvec} \Rightarrow 'a \text{ spvec}$

where

```

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

```

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

lemma *addmult-spvec-empty2*[simp]: *addmult-spvec y a [] = a*
by *simp*

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

lemma *sparse-row-vector-smult*: *sparse-row-vector (smult-spvec y a) = scalar-mult y (sparse-row-vector a)*
by (*induct a*) (*simp-all add: smult-spvec-cons scalar-mult-add*)

lemma *sparse-row-vector-addmult-spvec*: *sparse-row-vector (addmult-spvec (y::'a::lattice-ring) a b) =*
(sparse-row-vector a) + (scalar-mult y (sparse-row-vector b))
by (*induct y a b rule: addmult-spvec.induct*)
(simp-all add: scalar-mult-add smult-spvec-cons sparse-row-vector-smult singleton-matrix-add)

lemma *sorted-smult-spvec*: *sorted-spvec a \implies sorted-spvec (smult-spvec y a)*
by (*induct a rule: sorted-spvec.induct*) (*auto simp: smult-spvec-def sorted-spvec-step1 sorted-spvec-step*)

lemma *sorted-spvec-addmult-spvec-helper*: $\llbracket \text{sorted-spvec } (\text{addmult-spvec } y ((a, b) \# \text{arr}) \text{ brr}); aa < a; \text{sorted-spvec } ((a, b) \# \text{arr}); \text{sorted-spvec } ((aa, ba) \# \text{brr}) \rrbracket \implies \text{sorted-spvec } ((aa, y * ba) \# \text{addmult-spvec } y ((a, b) \# \text{arr}) \text{ brr})$
by (*induct brr*) (*auto simp: sorted-spvec.simps*)

lemma *sorted-spvec-addmult-spvec-helper2*:
 $\llbracket \text{sorted-spvec } (\text{addmult-spvec } y \text{ arr } ((aa, ba) \# \text{brr})); a < aa; \text{sorted-spvec } ((a, b) \# \text{arr}); \text{sorted-spvec } ((aa, ba) \# \text{brr}) \rrbracket$
 $\implies \text{sorted-spvec } ((a, b) \# \text{addmult-spvec } y \text{ arr } ((aa, ba) \# \text{brr}))$
by (*induct arr*) (*auto simp: smult-spvec-def sorted-spvec.simps*)

lemma *sorted-spvec-addmult-spvec-helper3*[*rule-format*]:

```

sorted-spvec (addmult-spvec y arr brr) ==>
  sorted-spvec ((aa, b) # arr) ==>
  sorted-spvec ((aa, ba) # brr) ==>
  sorted-spvec ((aa, b + y * ba) # (addmult-spvec y arr brr))
by (smt (verit, ccfu-threshold) sorted-spvec-step addmult-spvec.simps(1) list.distinct(1)
list.sel(3) sorted-spvec.elims(1) sorted-spvec-addmult-spvec-helper2)

```

lemma sorted-addmult-spvec: sorted-spvec a ==> sorted-spvec b ==> sorted-spvec (addmult-spvec y a b)

proof (induct y a b rule: addmult-spvec.induct)

case (1 y arr)

then show ?case

by simp

next

case (2 y v va)

then show ?case

by (simp add: sorted-smult-spvec)

next

case (3 y i a arr j b brr)

show ?case

proof (cases i j rule: linorder-cases)

case less

with 3 show ?thesis

by (simp add: sorted-spvec-addmult-spvec-helper2 sorted-spvec-cons1)

next

case equal

with 3 show ?thesis

by (simp add: sorted-spvec-addmult-spvec-helper3 sorted-spvec-cons1)

next

case greater

with 3 show ?thesis

by (simp add: sorted-spvec-addmult-spvec-helper sorted-spvec-cons1)

qed

qed

fun mult-spvec-spmat :: ('a::lattice-ring) spvec => 'a spvec => 'a smat => '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:

assumes sorted-spvec (a::('a::lattice-ring) spvec) sorted-spvec B

shows 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
{
  fix a
  fix v :: (nat × 'a) list
  assume a: a < nrows(sparse-row-vector v)
  have nrows(sparse-row-vector v) ≤ 1 by simp
  then have a = 0
    using a dual-order.strict-trans1 by blast
}
note nrows-helper = this
show ?thesis
  using assms
proof (induct c a B rule: mult-spvec-spmat.induct)
  case (1 c brr)
  then show ?case
    by simp
next
  case (2 c v va)
  then show ?case
    by simp
next
  case (3 c i a arr j b brr)
  then have abrr: sorted-spvec arr sorted-spvec brr
    using sorted-spvec-cons1 by blast+
  have ∧m n. [a ≠ 0; 0 < m]
    ==> a * Rep-matrix (sparse-row-vector b) m n = 0
    by (metis mult-zero-right neq0-conv nrows-helper nrows-notzero)
  then have †: scalar-mult a (sparse-row-vector b) =
    singleton-matrix 0 j a * move-matrix (sparse-row-vector b) (int j) 0
    apply (intro matrix-eqI)
    apply (simp)
    apply (subst Rep-matrix-mult)
    apply (subst foldseq-almostzero, auto)
    done
  show ?case
proof (cases i j rule: linorder-cases)
  case less
  with 3 abrr † show ?thesis
    apply (simp add: algebra-simps sparse-row-matrix-cons Rep-matrix-zero-imp-mult-zero)
    by (metis Rep-matrix-zero-imp-mult-zero Rep-singleton-matrix less-imp-le-nat
      sorted-sparse-row-matrix-zero)
next
  case equal
  with 3 abrr † show ?thesis
    apply (simp add: sparse-row-matrix-cons algebra-simps sparse-row-vector-addmult-spvec)
    apply (subst Rep-matrix-zero-imp-mult-zero)
    using sorted-sparse-row-matrix-zero apply fastforce
    apply (subst Rep-matrix-zero-imp-mult-zero)

```



```

apply (metis Rep-move-matrix comp-1 nrows-le nrows-spvec sorted-sparse-row-vector-zero
verit-comp-simplify1 3))
apply simp
done
next
case greater
have Rep-matrix (sparse-row-vector arr) j' k = 0  $\vee$ 
Rep-matrix (move-matrix (sparse-row-vector b) (int j) 0) k
i' = 0
if sorted-spvec ((i, a) # arr) for j' i' k
proof (cases k  $\leq$  j)
case True
with greater that show ?thesis
by (meson order.trans nat-less-le sorted-sparse-row-vector-zero)
qed (use nrows-helper nrows-notzero in force)
then have sparse-row-vector arr * move-matrix (sparse-row-vector b) (int j)
0 = 0
using greater 3
by (simp add: Rep-matrix-zero-imp-mult-zero)
with greater 3 abrr show ?thesis
apply (simp add: algebra-simps sparse-row-matrix-cons)
by (metis Rep-matrix-zero-imp-mult-zero Rep-move-matrix Rep-singleton-matrix
comp-1 nrows-le nrows-spvec)
qed
qed
qed

```

lemma sorted-mult-spvec-spmat:
sorted-spvec (c::('a::lattice-ring) spvec) \implies sorted-spmat B \implies sorted-spvec (mult-spvec-spmat
c a B)
by (induct c a B rule: mult-spvec-spmat.induct) (simp-all add: sorted-addmult-spvec)

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 \implies sorted-spvec B \implies
sparse-row-matrix (mult-spmat A B) = (sparse-row-matrix A) * (sparse-row-matrix
B)
by (induct A) (auto simp: sparse-row-matrix-cons sparse-row-mult-spvec-spmat
algebra-simps move-matrix-mult)

lemma sorted-spvec-mult-spmat:
fixes A :: ('a::lattice-ring) spmat
shows sorted-spvec A \implies sorted-spvec (mult-spmat A B)
by (induct A rule: sorted-spvec.induct) (auto simp: sorted-spvec.simps)

lemma *sorted-spmat-mult-spmat*:
sorted-spmat (*B*::('a::lattice-ring) *spmat*) \implies *sorted-spmat* (*mult-spmat* *A B*)
by (*induct A*) (*auto simp: sorted-mult-spvec-spmat*)

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*)
by (*induct a b rule: add-spvec.induct*) (*simp-all add: singleton-matrix-add*)

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*)
by (*induct A B rule: add-spmat.induct*) (*auto simp: sparse-row-matrix-cons sparse-row-vector-add move-matrix-add*)

lemma [*code*]:
 $\langle \text{sparse-row-matrix } A + \text{sparse-row-matrix } B = \text{sparse-row-matrix } (\text{add-spmat } A \ B) \rangle$
 $\langle \text{sparse-row-vector } a + \text{sparse-row-vector } b = \text{sparse-row-vector } (\text{add-spvec } a \ b) \rangle$
by (*simp-all add: sparse-row-add-spmat sparse-row-vector-add*)

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 ($\forall x \text{ } ab \text{ } a. x = (a,b)\#arr \longrightarrow add\text{-}spvec \text{ } x \text{ } brr = (ab, bb) \# list \longrightarrow (ab = a \mid (ab = fst (hd \text{ } brr)))$)

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* \implies (*ab* = *a* | (*brr* \neq [] & *ab* = *fst* (*hd brr*)))

by (*smt (verit) add-spmat.elims fst-conv list.distinct(1) list.sel(1)*)

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*)))

by (*induct arr brr rule: add-spvec.induct*) (*auto split:if-splits*)

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*)))

by (*induct arr brr rule: add-spmat.induct*) (*auto split:if-splits*)

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*

by (*induct a b rule: add-spmat.induct*) (*simp-all add: add-spvec-commute*)

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*

by (*smt (verit, best) add-spvec.elims fst-conv list.sel(1) sorted-spvec-cons3*)

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*

by (*metis (no-types, opaque-lifting) add-spmat.simps(1) list.sel(1) neq-Nil-conv sorted-add-spmat-helper sorted-spvec-cons3*)

lemma *sorted-spvec-add-spvec*: *sorted-spvec a* \implies *sorted-spvec b* \implies *sorted-spvec* (*add-spvec a b*)

proof (*induct a b rule: add-spvec.induct*)

case ($\exists i \text{ } a \text{ } arr \text{ } j \text{ } b \text{ } brr$)

then have *sorted-spvec arr sorted-spvec brr*

using *sorted-spvec-cons1* **by** *blast+*

with \exists **show** ?case

apply *simp*

by (*smt (verit, ccfv-SIG) add-spvec.simps(2) list.sel(3) sorted-add-spvec-helper sorted-spvec.elims(1)*)

qed *auto*

lemma *sorted-spvec-add-spmat*:

sorted-spvec A \implies sorted-spvec B \implies sorted-spvec (add-spmat A B)

proof (*induct A B rule: add-spmat.induct*)

case (*1 bs*)

then show *?case* **by** *auto*

next

case (*2 v va*)

then show *?case* **by** *auto*

next

case (*3 i a as j b bs*)

then have *sorted-spvec as sorted-spvec bs*

using *sorted-spvec-cons1* **by** *blast+*

with *3* **show** *?case*

apply *simp*

by (*smt (verit) Pair-inject add-spmat.elims list.discI list.inject sorted-spvec.elims(1)*)

qed

lemma *sorted-spmat-add-spmat[rule-format]: sorted-spmat A \implies sorted-spmat B*

\implies sorted-spmat (add-spmat A B)

by (*induct A B rule: add-spmat.induct*) (*simp-all add: sorted-spvec-add-spmat*)

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

where

le-spvec [] [] = *True*

| *le-spvec* ((-,a)#as) [] = (*a* \leq 0 & *le-spvec as* [])

| *le-spvec* [] ((-,b)#bs) = (0 \leq *b* & *le-spvec* [] *bs*)

| *le-spvec* ((i,a)#as) ((j,b)#bs) = (

if (*i* < *j*) *then* *a* \leq 0 & *le-spvec as* ((j,b)#bs)

else if (*j* < *i*) *then* 0 \leq *b* & *le-spvec* ((i,a)#as) *bs*

else *a* \leq *b* & *le-spvec as* *bs*)

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

where

le-spmat [] [] = *True*

| *le-spmat* ((i,a)#as) [] = (*le-spmat a* [] & *le-spmat as* [])

| *le-spmat* [] ((j,b)#bs) = (*le-spmat* [] *b* & *le-spmat* [] *bs*)

| *le-spmat* ((i,a)#as) ((j,b)#bs) = (

if *i* < *j* *then* (*le-spmat a* [] & *le-spmat as* ((j,b)#bs))

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

else (*le-spmat a* *b* & *le-spmat as* *bs*))

definition *disj-matrices* :: (*'a::zero*) *matrix* \Rightarrow *'a matrix* \Rightarrow *bool* **where**

disj-matrices A B \longleftrightarrow

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

lemma *disj-matrices-contr1*: $\text{disj-matrices } A \ B \implies \text{Rep-matrix } A \ j \ i \neq 0 \implies \text{Rep-matrix } B \ j \ i = 0$
by (*simp add: disj-matrices-def*)

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

lemma *disj-matrices-add*:
fixes $A :: ('a::\text{lattice-ab-group-add}) \text{ matrix}$
shows $\text{disj-matrices } A \ B \implies \text{disj-matrices } C \ D \implies \text{disj-matrices } A \ D$
 $\implies \text{disj-matrices } B \ C \implies (A + B \leq C + D) = (A \leq C \wedge B \leq D)$
apply (*intro iffI conjI*)
unfolding *le-matrix-def disj-matrices-def*
apply (*metis Rep-matrix-add group-cancel.rule0 order-refl*)
apply (*metis (no-types, lifting) Rep-matrix-add add-cancel-right-left dual-order.refl*)
by (*meson add-mono le-matrix-def*)

lemma *disj-matrices-zero1*[*simp*]: $\text{disj-matrices } 0 \ B$
by (*simp add: disj-matrices-def*)

lemma *disj-matrices-zero2*[*simp*]: $\text{disj-matrices } A \ 0$
by (*simp add: disj-matrices-def*)

lemma *disj-matrices-commute*: $\text{disj-matrices } A \ B = \text{disj-matrices } B \ A$
by (*auto simp: disj-matrices-def*)

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

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

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

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

lemma *disj-sparse-row-singleton*: $i \leq j \implies \text{sorted-spvec}((j,y)\#v) \implies \text{disj-matrices} (\text{sparse-row-vector } v) (\text{singleton-matrix } 0 \ i \ x)$
apply (*simp add: disj-matrices-def*)
using *sorted-sparse-row-vector-zero* **by** *blast*

lemma *disj-matrices-x-add*: $\text{disj-matrices } A \ B \implies \text{disj-matrices } A \ C \implies \text{disj-matrices } (A :: ('a :: \text{lattice-ab-group-add}) \text{ matrix}) \ (B + C)$
by (*smt* (*verit*, *ccfv-SIG*) *Rep-matrix-add add-0 disj-matrices-def*)

lemma *disj-matrices-add-x*: $\text{disj-matrices } A \ B \implies \text{disj-matrices } A \ C \implies \text{disj-matrices } (B + C) \ (A :: ('a :: \text{lattice-ab-group-add}) \text{ matrix})$
by (*simp add: disj-matrices-x-add disj-matrices-commute*)

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

lemma *disj-move-sparse-vec-mat*:
assumes $j \leq a$ **and** *sorted-spvec* $((a, c) \# as)$
shows $\text{disj-matrices } (\text{sparse-row-matrix } as) \ (\text{move-matrix } (\text{sparse-row-vector } b) \ (int \ j) \ i)$
proof –
have $\text{Rep-matrix } (\text{sparse-row-vector } b) \ (n - j) \ (nat \ (int \ m - i)) = 0$
if $\neg n < j$ **and** *nz*: $\text{Rep-matrix } (\text{sparse-row-matrix } as) \ n \ m \neq 0$
for $n \ m$
proof –
have $n \neq j$
using *assms sorted-sparse-row-matrix-zero nz* **by** *blast*
with *that* **have** $j < n$ **by** *auto*
then show *?thesis*
by (*metis One-nat-def Suc-diff-Suc nrows nrows-spvec plus-1-eq-Suc trans-le-add1*)
qed
then show *?thesis*
by (*auto simp: disj-matrices-def nat-minus-as-int*)
qed

lemma *disj-move-sparse-row-vector-twice*:
 $j \neq u \implies \text{disj-matrices } (\text{move-matrix } (\text{sparse-row-vector } a) \ j \ i) \ (\text{move-matrix } (\text{sparse-row-vector } b) \ u \ v)$
unfolding *disj-matrices-def*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Rep-move-matrix of-nat-1 le-nat-iff nrows nrows-spvec of-nat-le-iff*)

lemma *le-spvec-iff-sparse-row-le*:
 $\text{sorted-spvec } a \implies \text{sorted-spvec } b \implies (\text{le-spvec } a \ b) \longleftrightarrow (\text{sparse-row-vector } a \leq \text{sparse-row-vector } b)$
proof (*induct a b rule: le-spvec.induct*)
case 1
then show *?case*
by *auto*
next
case (2 *uu a as*)
then have *sorted-spvec as*

```

    by (metis sorted-spvec-cons1)
  with 2 show ?case
    apply (simp add: add commute)
    by (metis disj-matrices-add-le-zero disj-sparse-row-singleton le-refl singleton-le-zero)
next
  case (3 uv b bs)
  then have sorted-spvec bs
    by (metis sorted-spvec-cons1)
  with 3 show ?case
    apply (simp add: add commute)
    by (metis disj-matrices-add-zero-le disj-sparse-row-singleton le-refl singleton-ge-zero)
next
  case (4 i a as j b bs)
  then obtain §: sorted-spvec as sorted-spvec bs
    by (metis sorted-spvec-cons1)
  show ?case
  proof (cases i j rule: linorder-cases)
    case less
    with 4 § show ?thesis
    apply (simp add: )
    by (metis disj-matrices-add-le-x disj-matrices-add-x disj-matrices-commute
disj-singleton-matrices disj-sparse-row-singleton less-imp-le-nat singleton-le-zero not-le)
  next
    case equal
    with 4 § show ?thesis
    apply (simp add: )
    by (metis disj-matrices-add disj-matrices-commute disj-sparse-row-singleton
order-refl singleton-matrix-le)
  next
    case greater
    with 4 § show ?thesis
    apply (simp add: )
    by (metis disj-matrices-add-x disj-matrices-add-x-le disj-matrices-commute
disj-singleton-matrices disj-sparse-row-singleton le-refl order-less-le singleton-ge-zero)
  qed
qed

```

lemma *le-spvec-empty2-sparse-row*:
 $\text{sorted-spvec } b \implies \text{le-spvec } b \ \square = (\text{sparse-row-vector } b \leq 0)$
by (simp add: le-spvec-iff-sparse-row-le)

lemma *le-spvec-empty1-sparse-row*:
 $(\text{sorted-spvec } b) \implies (\text{le-spvec } \square \ b = (0 \leq \text{sparse-row-vector } b))$
by (simp add: le-spvec-iff-sparse-row-le)

lemma *le-spmat-iff-sparse-row-le*:
 $\llbracket \text{sorted-spvec } A; \text{sorted-spmat } A; \text{sorted-spvec } B; \text{sorted-spmat } B \rrbracket \implies$
 $\text{le-spmat } A \ B = (\text{sparse-row-matrix } A \leq \text{sparse-row-matrix } B)$
proof (induct A B rule: le-spmat.induct)

```

case (4 i a as j b bs)
then obtain §: sorted-spvec as sorted-spvec bs
  by (metis sorted-spvec-cons1)
show ?case
proof (cases i j rule: linorder-cases)
  case less
  with 4 § show ?thesis
    apply (simp add: sparse-row-matrix-cons le-spvec-empty2-sparse-row)
    by (metis disj-matrices-add-le-x disj-matrices-add-x disj-matrices-commute
disj-move-sparse-row-vector-twice disj-move-sparse-vec-mat int-eq-iff less-not-refl move-matrix-le-zero
order-le-less)
  next
  case equal
  with 4 § show ?thesis
    by (simp add: sparse-row-matrix-cons le-spvec-iff-sparse-row-le disj-matrices-commute
disj-move-sparse-vec-mat[OF order-refl] disj-matrices-add)
  next
  case greater
  with 4 § show ?thesis
    apply (simp add: sparse-row-matrix-cons le-spvec-empty1-sparse-row)
    by (metis disj-matrices-add-x disj-matrices-add-x-le disj-matrices-commute
disj-move-sparse-row-vector-twice disj-move-sparse-vec-mat move-matrix-zero-le nat-int
nat-less-le of-nat-0-le-iff order-refl)
  qed
qed (auto 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)

```

```

primrec abs-spmat :: ('a::lattice-ring) spmat  $\Rightarrow$  'a spmat
where
  abs-spmat [] = []
  | abs-spmat (a#as) = (fst a, abs-spvec (snd a))#(abs-spmat as)

```

```

primrec minus-spmat :: ('a::lattice-ring) spmat  $\Rightarrow$  'a spmat
where
  minus-spmat [] = []
  | minus-spmat (a#as) = (fst a, minus-spvec (snd a))#(minus-spmat as)

```

```

lemma sparse-row-matrix-minus:
  sparse-row-matrix (minus-spmat A) = - (sparse-row-matrix A)
proof (induct A)
  case Nil
  then show ?case by auto
next
  case (Cons a A)
  then show ?case
    by (simp add: sparse-row-vector-minus sparse-row-matrix-cons matrix-eqI)
  qed

```


lemma *Rep-sparse-row-vector-zero*:

assumes $x \neq 0$

shows *Rep-matrix* (*sparse-row-vector* v) $x \ y = 0$

by (*metis Suc-leI assms le0 le-eq-less-or-eq nrows-le nrows-spvec*)

lemma *sparse-row-matrix-abs*:

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

proof (*induct A*)

case *Nil*

then show *?case* **by** *auto*

next

case (*Cons ab A*)

then have A : *sorted-spvec* A

using *sorted-spvec-cons1* **by** *blast*

show *?case*

proof (*cases ab*)

case (*Pair a b*)

show *?thesis*

unfolding *Pair*

proof (*intro matrix-eqI*)

fix $m \ n$

show *Rep-matrix* (*sparse-row-matrix* (*abs-spmat* ((a, b) $\#$ A))) $m \ n$
 $=$ *Rep-matrix* $|\text{sparse-row-matrix } ((a, b) \# A)| \ m \ n$

using *Cons Pair A*

apply (*simp add: sparse-row-vector-abs sparse-row-matrix-cons*)

apply (*cases m=a*)

using *sorted-sparse-row-matrix-zero* **apply** *fastforce*

by (*simp add: Rep-sparse-row-vector-zero*)

qed

qed

qed

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

by (*induct A rule: sorted-spvec.induct*) (*auto simp: sorted-spvec.simps*)

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

by (*induct A rule: sorted-spvec.induct*) (*auto simp: sorted-spvec.simps*)

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

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

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

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

definition *diff-spmat* :: ($'a::\text{lattice-ring}$) *spmat* $\Rightarrow 'a \text{ spat} \Rightarrow 'a \text{ spat}$

where *diff-spmat* $A\ B = \text{add-spmat } A\ (\text{minus-spmat } B)$

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* $A \longleftrightarrow \text{sorted-spvec } A \ \& \ \text{sorted-spmat } A$

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

lemma *if-case-eq*: $(\text{if } b \text{ then } x \text{ else } y) = (\text{case } b \text{ of } \text{True} \Rightarrow x \mid \text{False} \Rightarrow y)$ **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 *nprr-spvec* :: ('a::lattice-ab-group-add) spvec \Rightarrow 'a spvec
where

nprr-spvec [] = []
| *nprr-spvec* (a#as) = (fst a, nprr (snd a)) # (*nprr-spvec* as)

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

pprr-spmat [] = []
| *pprr-spmat* (a#as) = (fst a, ppr-def sup-matrix-def) # (pprr-spmat as)

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

nprr-spmat [] = []
| *nprr-spmat* (a#as) = (fst a, nprr-spvec (snd a)) # (nprr-spmat as)

lemma *pprr-add: disj-matrices A (B::(-::lattice-ring) matrix) \Longrightarrow ppr (A+B) = ppr A + ppr B*

apply (*simp add: ppr-def sup-matrix-def*)
apply (*intro matrix-eqI*)
by (*smt (verit, del-insts) Rep-combine-matrix Rep-zero-matrix-def add.commute comm-monoid-add-class.add-0 disj-matrices-def plus-matrix-def sup.idem*)

lemma *nprr-add: disj-matrices A (B::(-::lattice-ring) matrix) \Longrightarrow nprr (A+B) = nprr A + nprr B*

unfolding *nprr-def inf-matrix-def*
apply (*intro matrix-eqI*)
by (*smt (verit, ccfv-threshold) Rep-combine-matrix Rep-matrix-add add.commute add-cancel-right-right add-eq-inf-sup disj-matrices-contr2 sup.idem*)

lemma *pprr-singleton[simp]:*

fixes *x:: -::lattice-ring*
shows *pprr (singleton-matrix j i x) = singleton-matrix j i (pprr x)*
unfolding *pprr-def sup-matrix-def*
by (*simp add: matrix-eqI*)

lemma *nprr-singleton[simp]:*

fixes *x:: -::lattice-ring*
shows *nprr (singleton-matrix j i x) = singleton-matrix j i (nprr x)*
by (*metis add-left-imp-eq ppr-singleton prts singleton-matrix-add*)

lemma *sparse-row-vector-ppr:*

fixes *v:: -::lattice-ring spvec*
shows *sorted-spvec v \Longrightarrow sparse-row-vector (ppr-spvec v) = ppr (sparse-row-vector v)*

proof (*induct v rule: sorted-spvec.induct*)

case (*3 m x n y bs*)

then show ?case

apply *simp*

```

    apply (subst pprt-add)
    apply (metis disj-matrices-commute disj-sparse-row-singleton order.refl fst-conv
prod.sel(2) sparse-row-vector-cons)
    by (metis pprt-singleton sorted-spvec-cons1)
qed auto

```

```

lemma sparse-row-vector-nprt:
  fixes v:: 'a::lattice-ring spvec
  shows sorted-spvec v  $\implies$  sparse-row-vector (nprt-spvec v) = nprt (sparse-row-vector
v)
proof (induct v rule: sorted-spvec.induct)
  case (3 m x n y bs)
  then show ?case
    apply simp
    apply (subst nprt-add)
    apply (metis disj-matrices-commute disj-sparse-row-singleton dual-order.refl
fst-conv prod.sel(2) sparse-row-vector-cons)
    using sorted-spvec-cons1 by force
qed auto

```

```

lemma pprt-move-matrix: pprt (move-matrix (A::('a::lattice-ring) matrix) j i) =
move-matrix (pprt A) j i
  by (simp add: pprt-def sup-matrix-def matrix-eqI)

```

```

lemma nprr-move-matrix: nprr (move-matrix (A::('a::lattice-ring) matrix) j i) =
move-matrix (nprr A) j i
  by (simp add: nprr-def inf-matrix-def matrix-eqI)

```

```

lemma sparse-row-matrix-pprt:
  fixes m:: 'a::lattice-ring spmat
  shows sorted-spvec m  $\implies$  sorted-spmat m  $\implies$  sparse-row-matrix (pprt-spmat
m) = pprr (sparse-row-matrix m)
proof (induct m rule: sorted-spvec.induct)
  case (2 a)
  then show ?case
    by (simp add: pprr-move-matrix sparse-row-matrix-cons sparse-row-vector-pprr)
next
  case (3 m x n y bs)
  then show ?case
    apply (simp add: sparse-row-matrix-cons sparse-row-vector-pprr)
    apply (subst pprr-add)
    apply (subst disj-matrices-commute)
    apply (metis disj-move-sparse-vec-mat eq-imp-le fst-conv prod.sel(2) sparse-row-matrix-cons)
    apply (simp add: sorted-spvec.simps pprr-move-matrix)
    done
qed auto

```

```

lemma sparse-row-matrix-nprr:

```

```

fixes m:: 'a::lattice-ring spmat
shows sorted-spvec m  $\implies$  sorted-spmat m  $\implies$  sorted-spmat m  $\implies$  sparse-row-matrix
(npert-spmat m) = npert (sparse-row-matrix m)
proof (induct m rule: sorted-spvec.induct)
  case (2 a)
  then show ?case
    by (simp add: npert-move-matrix sparse-row-matrix-cons sparse-row-vector-npert)
next
  case (3 m x n y bs)
  then show ?case
    apply (simp add: sparse-row-matrix-cons sparse-row-vector-npert)
    apply (subst npert-add)
    apply (subst disj-matrices-commute)
    apply (metis disj-move-sparse-vec-mat fst-conv nle-le prod.sel(2) sparse-row-matrix-cons)
    apply (simp add: sorted-spvec.simps npert-move-matrix)
    done
qed auto

lemma sorted-pprt-spvec: sorted-spvec v  $\implies$  sorted-spvec (pprt-spvec v)
proof (induct v rule: sorted-spvec.induct)
  case 1
  then show ?case by auto
next
  case (2 a)
  then show ?case
    by (simp add: sorted-spvec-step1)
next
  case (3 m x n y bs)
  then show ?case
    by (simp add: sorted-spvec-step)
qed

lemma sorted-npert-spvec: sorted-spvec v  $\implies$  sorted-spvec (npert-spvec v)
by (induct v rule: sorted-spvec.induct) (simp-all add: sorted-spvec.simps split:list.split-asm)

lemma sorted-spvec-pprt-spmat: sorted-spvec m  $\implies$  sorted-spvec (pprt-spmat m)
by (induct m rule: sorted-spvec.induct) (simp-all add: sorted-spvec.simps split:list.split-asm)

lemma sorted-spvec-npert-spmat: sorted-spvec m  $\implies$  sorted-spvec (npert-spmat m)
by (induct m rule: sorted-spvec.induct) (simp-all add: sorted-spvec.simps split:list.split-asm)

lemma sorted-spmat-pprt-spmat: sorted-spmat m  $\implies$  sorted-spmat (pprt-spmat m)
by (induct m) (simp-all add: sorted-pprt-spvec)

lemma sorted-spmat-npert-spmat: sorted-spmat m  $\implies$  sorted-spmat (npert-spmat m)
by (induct m) (simp-all add: sorted-npert-spvec)

```

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) (*nprrt-spmat* r2))
(*add-spmat* (*mult-spmat* (*nprrt-spmat* s2) (*pprt-spmat* r1)) (*mult-spmat* (*nprrt-spmat* s1) (*nprrt-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 pprrt-spvec.simps
nprrt-spmat.simps nprrt-spvec.simps
mult-est-spmat-def

end

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

lemma *le-add-right-mono*:
assumes
a <= *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

$A * x \leq (b::'a::lattice-ring)$

$0 \leq y$

$|A - A'| \leq \delta \cdot A$

$b \leq b'$

$|c - c'| \leq \delta \cdot c$

$|x| \leq r$

shows

$c * x \leq y * b' + (y * \delta \cdot A + |y * A' - c'| + \delta \cdot c) * r$

proof -

from *assms* **have** 1: $y * b \leq y * b'$ **by** (*simp add: mult-left-mono*)

from *assms* **have** 2: $y * (A * x) \leq y * b$ **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: $c * x + (y * (A - A') + (y * A' - c') + (c' - c)) * x \leq y * b'$ **by** *simp*

have 5: $c * x \leq y * b' + |(y * (A - A') + (y * A' - c') + (c' - c)) * x|$

by (*simp only: 4 estimate-by-abs*)

have 6: $|(y * (A - A') + (y * A' - c') + (c' - c)) * x| \leq |y * (A - A') + (y * A' - c') + (c' - c)| * |x|$

by (*simp add: abs-le-mult*)

have 7: $(|y * (A - A') + (y * A' - c') + (c' - c)|) * |x| \leq (|y * (A - A') + (y * A' - c')| + |c' - c|) * |x|$

by (*rule abs-triangle-ineq [THEN mult-right-mono] simp*)

have 8: $(|y * (A - A') + (y * A' - c')| + |c' - c|) * |x| \leq (|y * (A - A')| + |y * A' - c'| + |c' - c|) * |x|$

by (*simp add: abs-triangle-ineq mult-right-mono*)

have 9: $(|y * (A - A')| + |y * A' - c'| + |c' - c|) * |x| \leq (|y| * |A - A'| + |y * A' - c'| + |c' - c|) * |x|$

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: $(|y| * |A - A'| + |y * A' - c'| + |c' - c|) * |x| \leq (|y| * |A - A'| + |y * A' - c'| + \delta \cdot c) * |x|$

by (*simp add: 11 assms mult-right-mono*)

have 13: $(|y| * |A - A'| + |y * A' - c'| + \delta \cdot c) * |x| \leq (|y| * \delta \cdot A + |y * A' - c'| + \delta \cdot c) * |x|$

by (*simp add: assms mult-right-mono mult-left-mono*)

have *r*: $(|y| * \delta \cdot A + |y * A' - c'| + \delta \cdot c) * |x| \leq (|y| * \delta \cdot A + |y * A' - c'| + \delta \cdot c) * r$

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 $\delta \cdot A$, 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|
  <= (|y| * δ-A + |y*A'-c'| + δ-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 <= (A::'a::lattice-ring)
    A <= A2
  shows |A-A1| <= A2-A1
proof -
  have 0 <= 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| <= (A2-A1) by simp
qed

```

```

lemma mult-le-prts:
  assumes
    a1 <= (a::'a::lattice-ring)
    a <= a2
    b1 <= b
    b <= b2
  shows
    a * b <= ppert a2 * ppert b2 + ppert a1 * npert b2 + npert a2 * ppert b1 + npert a1
  * npert b1
proof -
  have a * b = (ppert a + npert a) * (ppert b + npert b)
    apply (subst prts[symmetric])
    apply simp
  done
  then have a * b = ppert a * ppert b + ppert a * npert b + npert a * ppert b + npert
  a * npert b
    by (simp add: algebra-simps)
  moreover have ppert a * ppert b <= ppert a2 * ppert b2
    by (simp-all add: assms mult-mono)
  moreover have ppert a * npert b <= ppert a1 * npert b2
proof -
  have ppert a * npert b <= ppert a * npert b2

```



```

    by (simp add: mult-left-mono assms)
  moreover have pprr a * nprr b2 <= pprr a1 * nprr b2
    by (simp add: mult-right-mono-neg assms)
  ultimately show ?thesis
    by simp
qed
moreover have nprr a * pprr b <= nprr a2 * pprr b1
proof -
  have nprr a * pprr b <= nprr a2 * pprr b
    by (simp add: mult-right-mono assms)
  moreover have nprr a2 * pprr b <= nprr a2 * pprr b1
    by (simp add: mult-left-mono-neg assms)
  ultimately show ?thesis
    by simp
qed
moreover have nprr a * nprr b <= nprr a1 * nprr b1
proof -
  have nprr a * nprr b <= nprr a * nprr b1
    by (simp add: mult-left-mono-neg assms)
  moreover have nprr a * nprr b1 <= nprr a1 * nprr 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 * nprr r2 + nprr s2 * pprr r1 + nprr s1 * nprr 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: algebra-simps)
  ultimately have c * x + (y * A - c) * x <= y * b by simp
  then have c * x <= y * b - (y * A - c) * x by (simp add: le-diff-eq)
  then have cx: c * x <= 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 $\langle \sim \sim /src/Tools/float.ML \rangle$

```

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

lemma *real-is-int-sub*[simp]: *real-is-int* $a \implies \text{real-is-int } b \implies \text{real-is-int } (a-b)$
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* $x \implies \exists!(a::\text{int}). \text{real-of-int } a = x$
by (*auto simp add: real-is-int-def*)

lemma *int-of-real-mult*:
assumes *real-is-int* a *real-is-int* b
shows $(\text{int-of-real } (a*b)) = (\text{int-of-real } a) * (\text{int-of-real } b)$
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* $a \implies \text{real-is-int } b \implies \text{real-is-int } (a*b)$
apply (*subst* *real-is-int-def2*)
apply (*simp* *add: int-of-real-mult*)
done

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

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

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

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

lemma int-of-real-1[simp]: int-of-real ( $1::real$ ) = ( $1::int$ )
proof -
  have  $1: (1::real) = real-of-int (1::int)$  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:  $a \neq 0 \implies a \neq -1 \implies |(a::int) \text{ div } 2| < |a|$ 
by arith

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

lemma add-left-zero:  $0 + a = (a::'a::comm-monoid-add)$ 
  by simp

lemma add-right-zero:  $a + 0 = (a::'a::comm-monoid-add)$ 
  by simp

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

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

lemma int-pow-0:  $(a::int)^0 = 1$ 
  by simp

lemma int-pow-1:  $(a::int)^(Numerals1) = a$ 
  by simp

```

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

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

lemma *zpower-Pls*: $(z::\text{int})^0 = \text{Numeral1}$
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* :: $(\text{int} \times \text{int}) \Rightarrow \text{real}$ **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*:
 $\text{float } (a1, e1) + \text{float } (a2, e2) =$
 $(\text{if } e1 \leq e2 \text{ then } \text{float } (a1 + a2 * 2^{\text{nat}(e2-e1)}, e1) \text{ else } \text{float } (a1 * 2^{\text{nat}(e1-e2)} + a2, e2))$
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 pprrt-lbound: $\text{pprrt } (\text{lbound } x) = \text{float } (0, 0)$
by (*auto simp: float-def lbound-def*)

lemma nprrt-ubound: $\text{nprrt } (\text{ubound } x) = \text{float } (0, 0)$

```

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-pprt float-nprt pprrt-lbound nprrt-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:  $x == y \implies x == y$  by simp
lemma meta-eq-imp-eq:  $x == y \implies x = y$  by auto
lemma eq-trivial:  $x = y \implies x = y$  by auto
lemma bool-to-true:  $x :: \text{bool} \implies x == True$  by simp
lemma transmeta-1:  $x = y \implies y == z \implies x = z$  by simp
lemma transmeta-2:  $x == y \implies y = z \implies x = z$  by simp
lemma transmeta-3:  $x == y \implies y == z \implies x = z$  by simp

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

lemmas compute-if = If-True If-False

```

lemma *bool1*: $(\neg \text{True}) = \text{False}$ **by** *blast*
lemma *bool2*: $(\neg \text{False}) = \text{True}$ **by** *blast*
lemma *bool3*: $(P \wedge \text{True}) = P$ **by** *blast*
lemma *bool4*: $(\text{True} \wedge P) = P$ **by** *blast*
lemma *bool5*: $(P \wedge \text{False}) = \text{False}$ **by** *blast*
lemma *bool6*: $(\text{False} \wedge P) = \text{False}$ **by** *blast*
lemma *bool7*: $(P \vee \text{True}) = \text{True}$ **by** *blast*
lemma *bool8*: $(\text{True} \vee P) = \text{True}$ **by** *blast*
lemma *bool9*: $(P \vee \text{False}) = P$ **by** *blast*
lemma *bool10*: $(\text{False} \vee P) = P$ **by** *blast*
lemma *bool11*: $(\text{True} \longrightarrow P) = P$ **by** *blast*
lemma *bool12*: $(P \longrightarrow \text{True}) = \text{True}$ **by** *blast*
lemma *bool13*: $(\text{True} \longrightarrow P) = P$ **by** *blast*
lemma *bool14*: $(P \longrightarrow \text{False}) = (\neg P)$ **by** *blast*
lemma *bool15*: $(\text{False} \longrightarrow P) = \text{True}$ **by** *blast*
lemma *bool16*: $(\text{False} = \text{False}) = \text{True}$ **by** *blast*
lemma *bool17*: $(\text{True} = \text{True}) = \text{True}$ **by** *blast*
lemma *bool18*: $(\text{False} = \text{True}) = \text{False}$ **by** *blast*
lemma *bool19*: $(\text{True} = \text{False}) = \text{False}$ **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*: $\text{fst } (x, y) = x$ **by** *simp*
lemma *compute-snd*: $\text{snd } (x, y) = y$ **by** *simp*
lemma *compute-pair-eq*: $((a, b) = (c, d)) = (a = c \wedge b = d)$ **by** *auto*

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

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

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

definition *case-option-compute* :: $'b \text{ option} \Rightarrow 'a \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$
where *case-option-compute* *opt a f* = *case-option a f opt*

lemma *case-option-compute*: $\text{case-option} = (\lambda \ a \ f \ \text{opt}. \ \text{case-option-compute } \text{opt } a \ f)$


```

by (simp add: case-option-compute-def)

lemma case-option-compute-None: case-option-compute None = (λ a f. a)
  apply (rule ext)+
  apply (simp add: case-option-compute-def)
  done

lemma case-option-compute-Some: case-option-compute (Some x) = (λ 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 ⇒ 'a ⇒ ('b ⇒ 'b list ⇒ 'a) ⇒ 'a
  where case-list-compute l a f = case-list a f l

lemma case-list-compute: case-list = (λ (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) = (λ (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) = (λ (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* \rightarrow *thm list*

val to-meta-eq : *thm* \rightarrow *thm*

val to-hol-eq : *thm* \rightarrow *thm*

val symmetric : *thm* \rightarrow *thm*

val trans : *thm* \rightarrow *thm* \rightarrow *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* \Rightarrow *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-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)
      (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) (npmt-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