# Matrix

Steven Obua

June 8, 2008

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

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

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

**typedef** *'a matrix* = {(*f*::((*'a*::*zero*) *infmatrix*)). *finite* (*nonzero-positions f*)}
⟨*proof*⟩

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

**lemma** *finite-nonzero-positions* : *finite* (*nonzero-positions* (*Rep-matrix A*))
⟨*proof*⟩

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

**lemma** *nrows*:
**assumes** *hyp*: *nrows A* ≤ *m*
**shows** (*Rep-matrix A m n*) = *0* (**is** *?concl*)
⟨*proof*⟩

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

**declare** *transpose-infmatrix-def*[*simp*]

**lemma** *transpose-infmatrix-twice*[*simp*]: *transpose-infmatrix* (*transpose-infmatrix* *A*) = *A*
⟨*proof*⟩

**lemma** *transpose-infmatrix*: *transpose-infmatrix* (% *j i*. *P j i*) = (% *j i*. *P i j*)
⟨*proof*⟩

**lemma** *transpose-infmatrix-closed*[*simp*]: *Rep-matrix* (*Abs-matrix* (*transpose-infmatrix* (*Rep-matrix* *x*))) = *transpose-infmatrix* (*Rep-matrix* *x*)
⟨*proof*⟩

**lemma** *infmatrixforward*: (*x*::'*a* *infmatrix*) = *y* ⟹ ∀ *a b*. *x a b* = *y a b* ⟨*proof*⟩

**lemma** *transpose-infmatrix-inject*: (*transpose-infmatrix* *A* = *transpose-infmatrix* *B*) = (*A* = *B*)
⟨*proof*⟩

**lemma** *transpose-matrix-inject*: (*transpose-matrix* *A* = *transpose-matrix* *B*) = (*A* = *B*)
⟨*proof*⟩

**lemma** *transpose-matrix*[*simp*]: *Rep-matrix*(*transpose-matrix* *A*) *j i* = *Rep-matrix* *A i j*
⟨*proof*⟩

**lemma** *transpose-transpose-id*[*simp*]: *transpose-matrix* (*transpose-matrix* *A*) = *A*
⟨*proof*⟩

**lemma** *nrows-transpose*[*simp*]: *nrows* (*transpose-matrix* *A*) = *ncols* *A*
⟨*proof*⟩

**lemma** *ncols-transpose*[*simp*]: *ncols* (*transpose-matrix* *A*) = *nrows* *A*
⟨*proof*⟩

**lemma** *ncols*: *ncols* *A* <= *n* ⟹ *Rep-matrix* *A m n* = *0*
⟨*proof*⟩

**lemma** *ncols-le*: (*ncols* *A* <= *n*) = (! *j i*. *n* <= *i* ⟶ (*Rep-matrix* *A j i*) = *0*) (**is** - = *?st*)
⟨*proof*⟩

**lemma** *less-ncols*: (*n* < *ncols* *A*) = (*? j i*. *n* <= *i* & (*Rep-matrix* *A j i*) ≠ *0*) (**is** *?concl*)
⟨*proof*⟩

**lemma** *le-ncols*: (*n* <= *ncols* *A*) = (∀ *m*. (∀ *j i*. *m* <= *i* ⟶ (*Rep-matrix* *A j i*)

$= 0) \longrightarrow n <= m)$ (**is** *?concl*)
⟨*proof*⟩

**lemma** *nrows-le*: $(nrows\ A <= n) = (!\ j\ i.\ n <= j \longrightarrow (Rep\text{-}matrix\ A\ j\ i) = 0)$ (**is** *?s*)
⟨*proof*⟩

**lemma** *less-nrows*: $(m < nrows\ A) = (?\ j\ i.\ m <= j\ \&\ (Rep\text{-}matrix\ A\ j\ i) \neq 0)$ (**is** *?concl*)
⟨*proof*⟩

**lemma** *le-nrows*: $(n <= nrows\ A) = (\forall\ m.\ (\forall\ j\ i.\ m <= j \longrightarrow (Rep\text{-}matrix\ A\ j\ i) = 0) \longrightarrow n <= m)$ (**is** *?concl*)
⟨*proof*⟩

**lemma** *nrows-notzero*: $Rep\text{-}matrix\ A\ m\ n \neq 0 \Longrightarrow m < nrows\ A$
⟨*proof*⟩

**lemma** *ncols-notzero*: $Rep\text{-}matrix\ A\ m\ n \neq 0 \Longrightarrow n < ncols\ A$
⟨*proof*⟩

**lemma** *finite-natarray1*: *finite* $\{x.\ x < (n::nat)\}$
⟨*proof*⟩

**lemma** *finite-natarray2*: *finite* $\{pos.\ (fst\ pos) < (m::nat)\ \&\ (snd\ pos) < (n::nat)\}$
⟨*proof*⟩

**lemma** *RepAbs-matrix*:
**assumes** *aem*: $?\ m.\ !\ j\ i.\ m <= j \longrightarrow x\ j\ i = 0$ (**is** *?em*) **and** *aen*: $?\ n.\ !\ j\ i.\ (n <= i \longrightarrow x\ j\ i = 0)$ (**is** *?en*)
**shows** $(Rep\text{-}matrix\ (Abs\text{-}matrix\ x)) = x$
⟨*proof*⟩

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

**lemma** *expand-apply-infmatrix*[*simp*]: $apply\text{-}infmatrix\ f\ A\ j\ i = f\ (A\ j\ i)$
⟨*proof*⟩

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

$(B\ j\ i)$
⟨*proof*⟩

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

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

**lemma** *combine-infmatrix-commute*:
$commutative\ f \Longrightarrow commutative\ (combine\text{-}infmatrix\ f)$
⟨*proof*⟩

**lemma** *combine-matrix-commute*:
$commutative\ f \Longrightarrow commutative\ (combine\text{-}matrix\ f)$
⟨*proof*⟩

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

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

but on the other hand we have

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

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

**lemma** *nonzero-positions-combine-infmatrix*[*simp*]: $f\ 0\ 0 = 0 \Longrightarrow nonzero\text{-}positions\ (combine\text{-}infmatrix\ f\ A\ B) \subseteq (nonzero\text{-}positions\ A) \cup (nonzero\text{-}positions\ B)$
⟨*proof*⟩

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

**lemma** *combine-infmatrix-closed* [*simp*]:
$f\ 0\ 0 = 0 \Longrightarrow Rep\text{-}matrix\ (Abs\text{-}matrix\ (combine\text{-}infmatrix\ f\ (Rep\text{-}matrix\ A)\ (Rep\text{-}matrix\ B))) = combine\text{-}infmatrix\ f\ (Rep\text{-}matrix\ A)\ (Rep\text{-}matrix\ B)$

⟨*proof*⟩

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

**lemma** *nonzero-positions-apply-infmatrix*[*simp*]: $f\ 0 = 0 \Longrightarrow$ *nonzero-positions* (*apply-infmatrix f A*) $\subseteq$ *nonzero-positions A*
⟨*proof*⟩

**lemma** *apply-infmatrix-closed* [*simp*]:
$f\ 0 = 0 \Longrightarrow$ *Rep-matrix* (*Abs-matrix* (*apply-infmatrix f* (*Rep-matrix A*))) = *apply-infmatrix f* (*Rep-matrix A*)
⟨*proof*⟩

**lemma** *combine-infmatrix-assoc*[*simp*]: *f 0 0 = 0* $\Longrightarrow$ *associative f* $\Longrightarrow$ *associative* (*combine-infmatrix f*)
⟨*proof*⟩

**lemma** *comb*: $f = g \Longrightarrow x = y \Longrightarrow f\ x = g\ y$
⟨*proof*⟩

**lemma** *combine-matrix-assoc*: *f 0 0 = 0* $\Longrightarrow$ *associative f* $\Longrightarrow$ *associative* (*combine-matrix f*)
⟨*proof*⟩

**lemma** *Rep-apply-matrix*[*simp*]: $f\ 0 = 0 \Longrightarrow$ *Rep-matrix* (*apply-matrix f A*) *j i* = *f* (*Rep-matrix A j i*)
⟨*proof*⟩

**lemma** *Rep-combine-matrix*[*simp*]: $f\ 0\ 0 = 0 \Longrightarrow$ *Rep-matrix* (*combine-matrix f A B*) *j i* = *f* (*Rep-matrix A j i*) (*Rep-matrix B j i*)
⟨*proof*⟩

**lemma** *combine-nrows-max*: $f\ 0\ 0 = 0 \Longrightarrow$ *nrows* (*combine-matrix f A B*) <= *max* (*nrows A*) (*nrows B*)
⟨*proof*⟩

**lemma** *combine-ncols-max*: *f 0 0 = 0* $\Longrightarrow$ *ncols* (*combine-matrix f A B*) <= *max* (*ncols A*) (*ncols B*)
⟨*proof*⟩

**lemma** *combine-nrows*: $f\ 0\ 0 = 0 \Longrightarrow$ *nrows A* <= *q* $\Longrightarrow$ *nrows B* <= *q* $\Longrightarrow$ *nrows*(*combine-matrix f A B*) <= *q*
⟨*proof*⟩

**lemma** *combine-ncols*: $f\ 0\ 0 = 0 \Longrightarrow$ *ncols A* <= *q* $\Longrightarrow$ *ncols B* <= *q* $\Longrightarrow$ *ncols*(*combine-matrix f A B*) <= *q*
⟨*proof*⟩

**constdefs**

*zero-r-neutral* :: $('a \Rightarrow 'b{::}zero \Rightarrow 'a) \Rightarrow bool$
*zero-r-neutral* $f == !\ a.\ f\ a\ 0 = a$
*zero-l-neutral* :: $('a{::}zero \Rightarrow 'b \Rightarrow 'b) \Rightarrow bool$
*zero-l-neutral* $f == !\ a.\ f\ 0\ a = a$
*zero-closed* :: $(('a{::}zero) \Rightarrow ('b{::}zero) \Rightarrow ('c{::}zero)) \Rightarrow bool$
*zero-closed* $f == (!x.\ f\ x\ 0 = 0)\ \&\ (!y.\ f\ 0\ y = 0)$

**consts** *foldseq* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$
**primrec**
*foldseq* $f\ s\ 0 = s\ 0$
*foldseq* $f\ s\ (Suc\ n) = f\ (s\ 0)\ (foldseq\ f\ (\%\ k.\ s(Suc\ k))\ n)$

**consts** *foldseq-transposed* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$
**primrec**
*foldseq-transposed* $f\ s\ 0 = s\ 0$
*foldseq-transposed* $f\ s\ (Suc\ n) = f\ (foldseq\text{-}transposed\ f\ s\ n)\ (s\ (Suc\ n))$

**lemma** *foldseq-assoc* : $associative\ f \Longrightarrow foldseq\ f = foldseq\text{-}transposed\ f$
⟨*proof*⟩

**lemma** *foldseq-distr*: $[\![associative\ f;\ commutative\ f]\!] \Longrightarrow foldseq\ f\ (\%\ k.\ f\ (u\ k)\ (v\ k))\ n = f\ (foldseq\ f\ u\ n)\ (foldseq\ f\ v\ n)$
⟨*proof*⟩

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

**lemma** *foldseq-zero*:
**assumes** *fz*: $f\ 0\ 0 = 0$ **and** *sz*: $!\ i.\ i <= n \longrightarrow s\ i = 0$
**shows** $foldseq\ f\ s\ n = 0$
⟨*proof*⟩

**lemma** *foldseq-significant-positions*:
**assumes** *p*: $!\ i.\ i <= N \longrightarrow S\ i = T\ i$
**shows** $foldseq\ f\ S\ N = foldseq\ f\ T\ N$ (**is** *?concl*)
⟨*proof*⟩

**lemma** *foldseq-tail*: $M <= N \Longrightarrow foldseq\ f\ S\ N = foldseq\ f\ (\%\ k.\ (if\ k < M\ then\ (S\ k)\ else\ (foldseq\ f\ (\%\ k.\ S(k{+}M))\ (N{-}M))))\ M$ (**is** *?p* $\Longrightarrow$ *?concl*)
⟨*proof*⟩

**lemma** *foldseq-zerotail*:
**assumes**
*fz*: $f\ 0\ 0 = 0$
**and** *sz*: $!\ i.\ \ n <= i \longrightarrow s\ i = 0$
**and** *nm*: $n <= m$

**shows**
*foldseq f s n = foldseq f s m*
⟨*proof*⟩

**lemma** *foldseq-zerotail2*:
**assumes** ! *x. f x 0 = x*
**and** ! *i. n < i ⟶ s i = 0*
**and** *nm*: *n <= m*
**shows**
*foldseq f s n = foldseq f s m* (**is** *?concl*)
⟨*proof*⟩

**lemma** *foldseq-zerostart*:
! *x. f 0 (f 0 x) = f 0 x ⟹* ! *i. i <= n ⟶ s i = 0 ⟹ foldseq f s (Suc n) = f 0 (s (Suc n))*
⟨*proof*⟩

**lemma** *foldseq-zerostart2*:
! *x. f 0 x = x ⟹* ! *i. i < n ⟶ s i = 0 ⟹ foldseq f s n = s n*
⟨*proof*⟩

**lemma** *foldseq-almostzero*:
**assumes** *f0x*:! *x. f 0 x = x* **and** *fx0*: ! *x. f x 0 = x* **and** *s0*:! *i. i ≠ j ⟶ s i = 0*
**shows** *foldseq f s n = (if (j <= n) then (s j) else 0)* (**is** *?concl*)
⟨*proof*⟩

**lemma** *foldseq-distr-unary*:
**assumes** !! *a b. g (f a b) = f (g a) (g b)*
**shows** *g(foldseq f s n) = foldseq f (% x. g(s x)) n* (**is** *?concl*)
⟨*proof*⟩

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

**lemma** *mult-matrix-n*:
**assumes** *prems*: *ncols A ≤ n* (**is** *?An*) *nrows B ≤ n* (**is** *?Bn*) *fadd 0 0 = 0 fmul 0 0 = 0*
**shows** *c:mult-matrix fmul fadd A B = mult-matrix-n n fmul fadd A B* (**is** *?concl*)
⟨*proof*⟩

**lemma** *mult-matrix-nm*:
**assumes** *prems*: *ncols A <= n nrows B <= n ncols A <= m nrows B <= m*

*fadd 0 0 = 0 fmul 0 0 = 0*
**shows** *mult-matrix-n n fmul fadd A B = mult-matrix-n m fmul fadd A B*
⟨*proof*⟩

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

**lemma** *max1*: !! *a x y*. (*a*::*nat*) <= *x* ⟹ *a* <= *max x y* ⟨*proof*⟩
**lemma** *max2*: !! *b x y*. (*b*::*nat*) <= *y* ⟹ *b* <= *max x y* ⟨*proof*⟩

**lemma** *r-distributive-matrix*:
**assumes** *prems*:
*r-distributive fmul fadd*
*associative fadd*
*commutative fadd*
*fadd 0 0 = 0*
! *a*. *fmul a 0 = 0*
! *a*. *fmul 0 a = 0*
**shows** *r-distributive* (*mult-matrix fmul fadd*) (*combine-matrix fadd*) (**is** *?concl*)
⟨*proof*⟩

**lemma** *l-distributive-matrix*:
**assumes** *prems*:
*l-distributive fmul fadd*
*associative fadd*
*commutative fadd*
*fadd 0 0 = 0*
! *a*. *fmul a 0 = 0*
! *a*. *fmul 0 a = 0*
**shows** *l-distributive* (*mult-matrix fmul fadd*) (*combine-matrix fadd*) (**is** *?concl*)
⟨*proof*⟩

**instance** *matrix* :: (*zero*) *zero* ⟨*proof*⟩

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

**lemma** *Rep-zero-matrix-def*[*simp*]: *Rep-matrix 0 j i = 0*
⟨*proof*⟩

**lemma** *zero-matrix-def-nrows*[*simp*]: *nrows 0 = 0*
⟨*proof*⟩

**lemma** *zero-matrix-def-ncols*[*simp*]: *ncols 0 = 0*
⟨*proof*⟩

**lemma** *combine-matrix-zero-l-neutral*: *zero-l-neutral f* ⟹ *zero-l-neutral* (*combine-matrix f*)
⟨*proof*⟩

**lemma** *combine-matrix-zero-r-neutral*: *zero-r-neutral f* ⟹ *zero-r-neutral* (*combine-matrix f*)
⟨*proof*⟩

**lemma** *mult-matrix-zero-closed*: ⟦*fadd 0 0 = 0*; *zero-closed fmul*⟧ ⟹ *zero-closed* (*mult-matrix fmul fadd*)
⟨*proof*⟩

**lemma** *mult-matrix-n-zero-right*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul a 0 = 0*⟧ ⟹ *mult-matrix-n n fmul fadd A 0 = 0*
⟨*proof*⟩

**lemma** *mult-matrix-n-zero-left*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul 0 a = 0*⟧ ⟹ *mult-matrix-n n fmul fadd 0 A = 0*
⟨*proof*⟩

**lemma** *mult-matrix-zero-left*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul 0 a = 0*⟧ ⟹ *mult-matrix fmul fadd 0 A = 0*
⟨*proof*⟩

**lemma** *mult-matrix-zero-right*[*simp*]: ⟦*fadd 0 0 = 0*; !*a. fmul a 0 = 0*⟧ ⟹ *mult-matrix fmul fadd A 0 = 0*
⟨*proof*⟩

**lemma** *apply-matrix-zero*[*simp*]: *f 0 = 0* ⟹ *apply-matrix f 0 = 0*
⟨*proof*⟩

**lemma** *combine-matrix-zero*: *f 0 0 = 0* ⟹ *combine-matrix f 0 0 = 0*
⟨*proof*⟩

**lemma** *transpose-matrix-zero*[*simp*]: *transpose-matrix 0 = 0*
⟨*proof*⟩

**lemma** *apply-zero-matrix-def*[*simp*]: *apply-matrix* (% *x. 0*) *A = 0*
⟨*proof*⟩

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

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

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

**lemma** *Rep-singleton-matrix*[*simp*]: *Rep-matrix (singleton-matrix j i e) m n = (if j = m & i = n then e else 0)*
⟨*proof*⟩

**lemma** *apply-singleton-matrix*[*simp*]: *f 0 = 0 ⟹ apply-matrix f (singleton-matrix j i x) = (singleton-matrix j i (f x))*
⟨*proof*⟩

**lemma** *singleton-matrix-zero*[*simp*]: *singleton-matrix j i 0 = 0*
⟨*proof*⟩

**lemma** *nrows-singleton*[*simp*]: *nrows(singleton-matrix j i e) = (if e = 0 then 0 else Suc j)*
⟨*proof*⟩

**lemma** *ncols-singleton*[*simp*]: *ncols(singleton-matrix j i e) = (if e = 0 then 0 else Suc i)*
⟨*proof*⟩

**lemma** *combine-singleton*: *f 0 0 = 0 ⟹ combine-matrix f (singleton-matrix j i a) (singleton-matrix j i b) = singleton-matrix j i (f a b)*
⟨*proof*⟩

**lemma** *transpose-singleton*[*simp*]: *transpose-matrix (singleton-matrix j i a) = singleton-matrix i j a*
⟨*proof*⟩

**lemma** *Rep-move-matrix*[*simp*]:
*Rep-matrix (move-matrix A y x) j i =*
*(if (neg ((int j)−y)) | (neg ((int i)−x)) then 0 else Rep-matrix A (nat((int j)−y)) (nat((int i)−x)))*
⟨*proof*⟩

**lemma** *move-matrix-0-0*[*simp*]: *move-matrix A 0 0 = A*
⟨*proof*⟩

**lemma** *move-matrix-ortho*: *move-matrix A j i = move-matrix (move-matrix A j 0) 0 i*
⟨*proof*⟩

**lemma** *transpose-move-matrix*[*simp*]:
*transpose-matrix (move-matrix A x y) = move-matrix (transpose-matrix A) y x*
⟨*proof*⟩

**lemma** *move-matrix-singleton*[*simp*]: *move-matrix (singleton-matrix u v x) j i =*
*(if (j + int u < 0) | (i + int v < 0) then 0 else (singleton-matrix (nat (j + int u)) (nat (i + int v)) x))*
⟨*proof*⟩

**lemma** *Rep-take-columns*[*simp*]:
*Rep-matrix (take-columns A c) j i =*
*(if i < c then (Rep-matrix A j i) else 0)*
⟨*proof*⟩

**lemma** *Rep-take-rows*[*simp*]:
*Rep-matrix (take-rows A r) j i =*
*(if j < r then (Rep-matrix A j i) else 0)*
⟨*proof*⟩

**lemma** *Rep-column-of-matrix*[*simp*]:
*Rep-matrix (column-of-matrix A c) j i = (if i = 0 then (Rep-matrix A j c) else 0)*
⟨*proof*⟩

**lemma** *Rep-row-of-matrix*[*simp*]:
*Rep-matrix (row-of-matrix A r) j i = (if j = 0 then (Rep-matrix A r i) else 0)*
⟨*proof*⟩

**lemma** *column-of-matrix*: *ncols A <= n ⟹ column-of-matrix A n = 0*
⟨*proof*⟩

**lemma** *row-of-matrix*: *nrows A <= n ⟹ row-of-matrix A n = 0*
⟨*proof*⟩

**lemma** *mult-matrix-singleton-right*[*simp*]:
**assumes** *prems*:
! *x*. *fmul x 0 = 0*
! *x*. *fmul 0 x = 0*
! *x*. *fadd 0 x = x*
! *x*. *fadd x 0 = x*
**shows** *(mult-matrix fmul fadd A (singleton-matrix j i e)) = apply-matrix (% x. fmul x e) (move-matrix (column-of-matrix A j) 0 (int i))*
⟨*proof*⟩

**lemma** *mult-matrix-ext*:

**assumes**
*eprem*:
*? e*. (! *a b*. *a* ≠ *b* ⟶ *fmul a e* ≠ *fmul b e*)
**and** *fprems*:
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
! *a*. *fadd a 0* = *a*
! *a*. *fadd 0 a* = *a*
**and** *contraprems*:
*mult-matrix fmul fadd A* = *mult-matrix fmul fadd B*
**shows**
*A* = *B*
⟨*proof*⟩

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

**lemma** *foldmatrix-transpose*:
**assumes**
! *a b c d*. *g*(*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*)
**shows**
*foldmatrix f g A m n* = *foldmatrix-transposed g f* (*transpose-infmatrix A*) *n m*
(**is** *?concl*)
⟨*proof*⟩

**lemma** *foldseq-foldseq*:
**assumes**
*associative f*
*associative g*
! *a b c d*. *g*(*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*)
**shows**
*foldseq g* (% *j*. *foldseq f* (*A j*) *n*) *m* = *foldseq f* (% *j*. *foldseq g* ((*transpose-infmatrix A*) *j*) *m*) *n*
⟨*proof*⟩

**lemma** *mult-n-nrows*:
**assumes**
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
*fadd 0 0* = *0*
**shows** *nrows* (*mult-matrix-n n fmul fadd A B*) ≤ *nrows A*
⟨*proof*⟩

**lemma** *mult-n-ncols*:
**assumes**
! *a*. *fmul 0 a = 0*
! *a*. *fmul a 0 = 0*
*fadd 0 0 = 0*
**shows** *ncols* (*mult-matrix-n n fmul fadd A B*) ≤ *ncols B*
⟨*proof*⟩

**lemma** *mult-nrows*:
**assumes**
! *a*. *fmul 0 a = 0*
! *a*. *fmul a 0 = 0*
*fadd 0 0 = 0*
**shows** *nrows* (*mult-matrix fmul fadd A B*) ≤ *nrows A*
⟨*proof*⟩

**lemma** *mult-ncols*:
**assumes**
! *a*. *fmul 0 a = 0*
! *a*. *fmul a 0 = 0*
*fadd 0 0 = 0*
**shows** *ncols* (*mult-matrix fmul fadd A B*) ≤ *ncols B*
⟨*proof*⟩

**lemma** *nrows-move-matrix-le*: *nrows* (*move-matrix A j i*) <= *nat*((*int* (*nrows A*)) + *j*)
⟨*proof*⟩

**lemma** *ncols-move-matrix-le*: *ncols* (*move-matrix A j i*) <= *nat*((*int* (*ncols A*)) + *i*)
⟨*proof*⟩

**lemma** *mult-matrix-assoc*:
**assumes** *prems*:
! *a*. *fmul1 0 a = 0*
! *a*. *fmul1 a 0 = 0*
! *a*. *fmul2 0 a = 0*
! *a*. *fmul2 a 0 = 0*
*fadd1 0 0 = 0*
*fadd2 0 0 = 0*
! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
*associative fadd1*
*associative fadd2*
! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)
**shows** *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*) *C* = *mult-matrix fmul1 fadd1 A* (*mult-matrix fmul2 fadd2 B C*) (**is** *?concl*)
⟨*proof*⟩

**lemma**
**assumes** *prems*:
! *a*. *fmul1 0 a = 0*
! *a*. *fmul1 a 0 = 0*
! *a*. *fmul2 0 a = 0*
! *a*. *fmul2 a 0 = 0*
*fadd1 0 0 = 0*
*fadd2 0 0 = 0*
! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
*associative fadd1*
*associative fadd2*
! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)
**shows**
(*mult-matrix fmul1 fadd1 A*) *o* (*mult-matrix fmul2 fadd2 B*) = *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*)
⟨*proof*⟩

**lemma** *mult-matrix-assoc-simple*:
**assumes** *prems*:
! *a*. *fmul 0 a = 0*
! *a*. *fmul a 0 = 0*
*fadd 0 0 = 0*
*associative fadd*
*commutative fadd*
*associative fmul*
*distributive fmul fadd*
**shows** *mult-matrix fmul fadd* (*mult-matrix fmul fadd A B*) *C* = *mult-matrix fmul fadd A* (*mult-matrix fmul fadd B C*) (**is** *?concl*)
⟨*proof*⟩

**lemma** *transpose-apply-matrix*: *f 0 = 0* ⟹ *transpose-matrix* (*apply-matrix f A*) = *apply-matrix f* (*transpose-matrix A*)
⟨*proof*⟩

**lemma** *transpose-combine-matrix*: *f 0 0 = 0* ⟹ *transpose-matrix* (*combine-matrix f A B*) = *combine-matrix f* (*transpose-matrix A*) (*transpose-matrix B*)
⟨*proof*⟩

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

(*nrows B*))
⟨*proof*⟩

**lemma** *transpose-mult-matrix*:
**assumes**
! *a*. *fmul 0 a = 0*
! *a*. *fmul a 0 = 0*
*fadd 0 0 = 0*
! *x y*. *fmul y x = fmul x y*
**shows**
*transpose-matrix* (*mult-matrix fmul fadd A B*) = *mult-matrix fmul fadd* (*transpose-matrix B*) (*transpose-matrix A*)
⟨*proof*⟩

**lemma** *column-transpose-matrix*: *column-of-matrix* (*transpose-matrix A*) *n* = *transpose-matrix* (*row-of-matrix A n*)
⟨*proof*⟩

**lemma** *take-columns-transpose-matrix*: *take-columns* (*transpose-matrix A*) *n* = *transpose-matrix* (*take-rows A n*)
⟨*proof*⟩

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

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

**definition**
*less-def*: $A < (B::'a$ *matrix*$) \longleftrightarrow A \leq B \wedge A \neq B$

**instance** ⟨*proof*⟩

**end**

**instance** *matrix* :: ({*order*, *zero*}) *order*
⟨*proof*⟩

**lemma** *le-apply-matrix*:
**assumes**
*f 0 = 0*
! *x y*. *x <= y* ⟶ *f x <= f y*
(*a*::('*a*::{*ord*, *zero*}) *matrix*) <= *b*
**shows**
*apply-matrix f a <= apply-matrix f b*
⟨*proof*⟩

**lemma** *le-combine-matrix*:
**assumes**

*f 0 0 = 0*
*! a b c d. a <= b & c <= d ⟶ f a c <= f b d*
*A <= B*
*C <= D*
**shows**
*combine-matrix f A C <= combine-matrix f B D*
⟨*proof*⟩

**lemma** *le-left-combine-matrix*:
**assumes**
*f 0 0 = 0*
*! a b c. a <= b ⟶ f c a <= f c b*
*A <= B*
**shows**
*combine-matrix f C A <= combine-matrix f C B*
⟨*proof*⟩

**lemma** *le-right-combine-matrix*:
**assumes**
*f 0 0 = 0*
*! a b c. a <= b ⟶ f a c <= f b c*
*A <= B*
**shows**
*combine-matrix f A C <= combine-matrix f B C*
⟨*proof*⟩

**lemma** *le-transpose-matrix*: (*A <= B*) = (*transpose-matrix A <= transpose-matrix B*)
⟨*proof*⟩

**lemma** *le-foldseq*:
**assumes**
*! a b c d . a <= b & c <= d ⟶ f a c <= f b d*
*! i. i <= n ⟶ s i <= t i*
**shows**
*foldseq f s n <= foldseq f t n*
⟨*proof*⟩

**lemma** *le-left-mult*:
**assumes**
*! a b c d. a <= b & c <= d ⟶ fadd a c <= fadd b d*
*! c a b. 0 <= c & a <= b ⟶ fmul c a <= fmul c b*
*! a. fmul 0 a = 0*
*! a. fmul a 0 = 0*
*fadd 0 0 = 0*
*0 <= C*
*A <= B*
**shows**
*mult-matrix fmul fadd C A <= mult-matrix fmul fadd C B*

⟨*proof*⟩

**lemma** *le-right-mult*:
**assumes**
! *a b c d*. *a* <= *b* & *c* <= *d* ⟶ *fadd a c* <= *fadd b d*
! *c a b*. *0* <= *c* & *a* <= *b* ⟶ *fmul a c* <= *fmul b c*
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
*fadd 0 0* = *0*
*0* <= *C*
*A* <= *B*
**shows**
*mult-matrix fmul fadd A C* <= *mult-matrix fmul fadd B C*
⟨*proof*⟩

**lemma** *spec2*: ! *j i*. *P j i* ⟹ *P j i* ⟨*proof*⟩
**lemma** *neg-imp*: (¬ *Q* ⟶ ¬ *P*) ⟹ *P* ⟶ *Q* ⟨*proof*⟩

**lemma** *singleton-matrix-le*[*simp*]: (*singleton-matrix j i a* <= *singleton-matrix j i b*) = (*a* <= (*b*::-::*order*))
⟨*proof*⟩

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

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

**lemma** *move-matrix-le-zero*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*move-matrix A j i* <= *0*) = (*A* <= (*0*::('*a*::{*order*,*zero*}) *matrix*))
⟨*proof*⟩

**lemma** *move-matrix-zero-le*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*0* <= *move-matrix A j i*) = ((*0*::('*a*::{*order*,*zero*}) *matrix*) <= *A*)
⟨*proof*⟩

**lemma** *move-matrix-le-move-matrix-iff*[*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*move-matrix A j i* <= *move-matrix B j i*) = (*A* <= (*B*::('*a*::{*order*,*zero*}) *matrix*))
⟨*proof*⟩

**end**

**theory** *Matrix*
**imports** *MatrixGeneral*
**begin**

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

**definition**
*inf* = *combine-matrix inf*

**definition**
*sup* = *combine-matrix sup*

**instance**
⟨*proof*⟩

**end**

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

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

**instance** ⟨*proof*⟩

**end**

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

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

**instance** ⟨*proof*⟩

**end**

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

**definition**
*diff-matrix-def*: $A - B$ = *combine-matrix* $(op -)$ $A$ $B$

**instance** ⟨*proof*⟩

**end**

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

**definition**
*times-matrix-def*: $A * B$ = *mult-matrix* $(op *)$ $(op +)$ $A$ $B$

**instance** ⟨*proof*⟩

**end**

**instantiation** *matrix* :: (*lordered-ab-group-add*) *abs*
**begin**

**definition**
*abs-matrix-def*: *abs* (*A* :: *'a matrix*) = *sup A* (− *A*)

**instance** ⟨*proof*⟩

**end**

**instance** *matrix* :: (*lordered-ab-group-add*) *lordered-ab-group-add-meet*
⟨*proof*⟩

**instance** *matrix* :: (*lordered-ring*) *lordered-ring*
⟨*proof*⟩

**lemma** *Rep-matrix-add*[*simp*]:
*Rep-matrix* ((*a*::(*'a*::*lordered-ab-group-add*)*matrix*)+*b*) *j i* = (*Rep-matrix a j i*)
+ (*Rep-matrix b j i*)
⟨*proof*⟩

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

**lemma** *apply-matrix-add*: ! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*) ⟹ *f 0* = (*0*::*'a*) ⟹
*apply-matrix f* ((*a*::(*'a*::*lordered-ab-group-add*) *matrix*) + *b*) = (*apply-matrix f a*)
+ (*apply-matrix f b*)
⟨*proof*⟩

**lemma** *singleton-matrix-add*: *singleton-matrix j i* ((*a*::-::*lordered-ab-group-add*)+*b*)
= (*singleton-matrix j i a*) + (*singleton-matrix j i b*)
⟨*proof*⟩

**lemma** *nrows-mult*: *nrows* ((*A*::(*'a*::*lordered-ring*) *matrix*) ∗ *B*) <= *nrows A*
⟨*proof*⟩

**lemma** *ncols-mult*: *ncols* ((*A*::(*'a*::*lordered-ring*) *matrix*) ∗ *B*) <= *ncols B*
⟨*proof*⟩

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

**lemma** *Rep-one-matrix*[*simp*]: *Rep-matrix* (*one-matrix n*) *j i* = (*if* (*j* = *i* & *j* < *n*) *then 1 else 0*)
⟨*proof*⟩

**lemma** *nrows-one-matrix*[*simp*]: *nrows* ((*one-matrix n*) :: ('*a*::*zero-neq-one*)*matrix*) = *n* (**is** *?r* = -)
⟨*proof*⟩

**lemma** *ncols-one-matrix*[*simp*]: *ncols* ((*one-matrix n*) :: ('*a*::*zero-neq-one*)*matrix*) = *n* (**is** *?r* = -)
⟨*proof*⟩

**lemma** *one-matrix-mult-right*[*simp*]: *ncols A* <= *n* ⟹ (*A*::('*a*::{*lordered-ring*,*ring-1*}) *matrix*) ∗ (*one-matrix n*) = *A*
⟨*proof*⟩

**lemma** *one-matrix-mult-left*[*simp*]: *nrows A* <= *n* ⟹ (*one-matrix n*) ∗ *A* = (*A*::('*a*::{*lordered-ring*, *ring-1*}) *matrix*)
⟨*proof*⟩

**lemma** *transpose-matrix-mult*: *transpose-matrix* ((*A*::('*a*::{*lordered-ring*,*comm-ring*}) *matrix*)∗*B*) = (*transpose-matrix B*) ∗ (*transpose-matrix A*)
⟨*proof*⟩

**lemma** *transpose-matrix-add*: *transpose-matrix* ((*A*::('*a*::*lordered-ab-group-add*) *matrix*)+*B*) = *transpose-matrix A* + *transpose-matrix B*
⟨*proof*⟩

**lemma** *transpose-matrix-diff*: *transpose-matrix* ((*A*::('*a*::*lordered-ab-group-add*) *matrix*)−*B*) = *transpose-matrix A* − *transpose-matrix B*
⟨*proof*⟩

**lemma** *transpose-matrix-minus*: *transpose-matrix* (−(*A*::('*a*::*lordered-ring*) *matrix*)) = − *transpose-matrix* (*A*::('*a*::*lordered-ring*) *matrix*)
⟨*proof*⟩

**constdefs**
*right-inverse-matrix* :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* ⇒ '*a matrix* ⇒ *bool*
*right-inverse-matrix A X* == (*A* ∗ *X* = *one-matrix* (*max* (*nrows A*) (*ncols X*))) ∧ *nrows X* ≤ *ncols A*
*left-inverse-matrix* :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* ⇒ '*a matrix* ⇒ *bool*
*left-inverse-matrix A X* == (*X* ∗ *A* = *one-matrix* (*max*(*nrows X*) (*ncols A*))) ∧ *ncols X* ≤ *nrows A*
*inverse-matrix* :: ('*a*::{*lordered-ring*, *ring-1*}) *matrix* ⇒ '*a matrix* ⇒ *bool*
*inverse-matrix A X* == (*right-inverse-matrix A X*) ∧ (*left-inverse-matrix A X*)

**lemma** *right-inverse-matrix-dim*: *right-inverse-matrix A X* ⟹ *nrows A* = *ncols X*

⟨*proof*⟩

**lemma** *left-inverse-matrix-dim*: *left-inverse-matrix A Y* ⟹ *ncols A* = *nrows Y*
⟨*proof*⟩

**lemma** *left-right-inverse-matrix-unique*:
**assumes** *left-inverse-matrix A Y right-inverse-matrix A X*
**shows** *X* = *Y*
⟨*proof*⟩

**lemma** *inverse-matrix-inject*: ⟦ *inverse-matrix A X*; *inverse-matrix A Y* ⟧ ⟹ *X* = *Y*
⟨*proof*⟩

**lemma** *one-matrix-inverse*: *inverse-matrix* (*one-matrix n*) (*one-matrix n*)
⟨*proof*⟩

**lemma** *zero-imp-mult-zero*: (*a*::'*a*::*ring*) = *0* | *b* = *0* ⟹ *a* * *b* = *0*
⟨*proof*⟩

**lemma** *Rep-matrix-zero-imp-mult-zero*:
! *j i k*. (*Rep-matrix A j k* = *0*) | (*Rep-matrix B k i*) = *0* ⟹ *A* * *B* = (*0*::('*a*::*lordered-ring*) *matrix*)
⟨*proof*⟩

**lemma** *add-nrows*: *nrows* (*A*::('*a*::*comm-monoid-add*) *matrix*) <= *u* ⟹ *nrows B* <= *u* ⟹ *nrows* (*A* + *B*) <= *u*
⟨*proof*⟩

**lemma** *move-matrix-row-mult*: *move-matrix* ((*A*::('*a*::*lordered-ring*) *matrix*) * *B*) *j 0* = (*move-matrix A j 0*) * *B*
⟨*proof*⟩

**lemma** *move-matrix-col-mult*: *move-matrix* ((*A*::('*a*::*lordered-ring*) *matrix*) * *B*) *0 i* = *A* * (*move-matrix B 0 i*)
⟨*proof*⟩

**lemma** *move-matrix-add*: ((*move-matrix* (*A* + *B*) *j i*)::(('*a*::*lordered-ab-group-add*) *matrix*)) = (*move-matrix A j i*) + (*move-matrix B j i*)
⟨*proof*⟩

**lemma** *move-matrix-mult*: *move-matrix* ((*A*::('*a*::*lordered-ring*) *matrix*)*B*) *j i* = (*move-matrix A j 0*) * (*move-matrix B 0 i*)
⟨*proof*⟩

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

**lemma** *scalar-mult-zero*[*simp*]: *scalar-mult y 0 = 0*
⟨*proof*⟩

**lemma** *scalar-mult-add*: *scalar-mult y (a+b) = (scalar-mult y a) + (scalar-mult y b)*
⟨*proof*⟩

**lemma** *Rep-scalar-mult*[*simp*]: *Rep-matrix (scalar-mult y a) j i = y * (Rep-matrix a j i)*
⟨*proof*⟩

**lemma** *scalar-mult-singleton*[*simp*]: *scalar-mult y (singleton-matrix j i x) = singleton-matrix j i (y * x)*
⟨*proof*⟩

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

**lemma** *Rep-abs*[*simp*]: *Rep-matrix (abs (A::-::lordered-ring)) x y = abs (Rep-matrix A x y)*
⟨*proof*⟩

**end**

**theory** *LP*
**imports** *Main*
**begin**

**lemma** *linprog-dual-estimate*:
**assumes**
*A * x ≤ (b::'a::lordered-ring)*
*0 ≤ y*
*abs (A − A') ≤ δA*
*b ≤ b'*
*abs (c − c') ≤ δc*
*abs x ≤ r*
**shows**
*c * x ≤ y * b' + (y * δA + abs (y * A' − c') + δc) * r*
⟨*proof*⟩

**lemma** *le-ge-imp-abs-diff-1*:
**assumes**
*A1 <= (A::'a::lordered-ring)*
*A <= A2*
**shows** *abs (A−A1) <= A2−A1*
⟨*proof*⟩

**lemma** *mult-le-prts*:
**assumes**
*a1* <= (*a*::'*a*::*lordered-ring*)
*a* <= *a2*
*b1* <= *b*
*b* <= *b2*
**shows**
*a* * *b* <= *pprt a2* * *pprt b2* + *pprt a1* * *nprt b2* + *nprt a2* * *pprt b1* + *nprt a1* * *nprt b1*
⟨*proof*⟩

**lemma** *mult-le-dual-prts*:
**assumes**
*A* * *x* ≤ (*b*::'*a*::*lordered-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 pprt s2* * *pprt r2* + *pprt s1* * *nprt r2* + *nprt s2* * *pprt r1* + *nprt s1* * *nprt r1*)
(**is** - <= - + *?C*)
⟨*proof*⟩

**end**

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

**types**
'*a spvec* = (*nat* * '*a*) *list*
'*a spmat* = ('*a spvec*) *spvec*

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

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

**lemma** *sparse-row-vector-empty*[*simp*]: *sparse-row-vector* [] = *0*
⟨*proof*⟩

**lemma** *sparse-row-matrix-empty*[*simp*]: *sparse-row-matrix* [] = *0*
⟨*proof*⟩

**lemma** *foldl-distrstart*[*rule-format*]: ! *a x y*. (*f* (*g x y*) *a* = *g x* (*f y a*)) ⟹ ! *x y*. (*foldl f* (*g x y*) *l* = *g x* (*foldl f y l*))
⟨*proof*⟩

**lemma** *sparse-row-vector-cons*[*simp*]: *sparse-row-vector* (*a*#*arr*) = (*singleton-matrix 0* (*fst a*) (*snd a*)) + (*sparse-row-vector arr*)
⟨*proof*⟩

**lemma** *sparse-row-vector-append*[*simp*]: *sparse-row-vector* (*a* @ *b*) = (*sparse-row-vector a*) + (*sparse-row-vector b*)
⟨*proof*⟩

**lemma** *nrows-spvec*[*simp*]: *nrows* (*sparse-row-vector x*) <= (*Suc 0*)
⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *sparse-row-matrix-append*: *sparse-row-matrix* (*arr*@*brr*) = (*sparse-row-matrix arr*) + (*sparse-row-matrix brr*)
⟨*proof*⟩

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

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

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

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

**lemma** *sorted-spvec-empty*[*simp*]: *sorted-spvec* [] = *True*
⟨*proof*⟩

**lemma** *sorted-spvec-cons1*: *sorted-spvec* (*a*#*as*) ⟹ *sorted-spvec as*
⟨*proof*⟩

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

⟨*proof*⟩

**lemma** *sorted-spvec-cons3*: *sorted-spvec*(*a*#*b*#*t*) ⟹ *fst a* < *fst b*
⟨*proof*⟩

**lemma** *sorted-sparse-row-vector-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*)
⟶ *Rep-matrix* (*sparse-row-vector arr*) *j m* = *0*
⟨*proof*⟩

**lemma** *sorted-sparse-row-matrix-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*)
⟶ *Rep-matrix* (*sparse-row-matrix arr*) *m j* = *0*
⟨*proof*⟩

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

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

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

**lemma** *sparse-row-vector-minus*:
*sparse-row-vector* (*minus-spvec v*) = − (*sparse-row-vector v*)
⟨*proof*⟩

**lemma** *sparse-row-vector-abs*:
*sorted-spvec v* ⟹ *sparse-row-vector* (*abs-spvec v*) = *abs* (*sparse-row-vector v*)
⟨*proof*⟩

**lemma** *sorted-spvec-minus-spvec*:
*sorted-spvec v* ⟹ *sorted-spvec* (*minus-spvec v*)
⟨*proof*⟩

**lemma** *sorted-spvec-abs-spvec*:
*sorted-spvec v* ⟹ *sorted-spvec* (*abs-spvec v*)
⟨*proof*⟩

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

**lemma** *smult-spvec-empty*[*simp*]: *smult-spvec y* [] = []
⟨*proof*⟩

**lemma** *smult-spvec-cons*: *smult-spvec y (a#arr) = (fst a, y * (snd a)) # (smult-spvec y arr)*
⟨*proof*⟩

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

**lemma** *addmult-spvec-empty1*[*simp*]: *addmult-spvec* (*y*, [], *a*) = *smult-spvec y a*
⟨*proof*⟩

**lemma** *addmult-spvec-empty2*[*simp*]: *addmult-spvec* (*y*, *a*, []) = *a*
⟨*proof*⟩

**lemma** *sparse-row-vector-map*: (! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*)) ⟹ (*f*::*'a*⇒(*'a*::*lordered-ring*)) *0* = *0* ⟹
*sparse-row-vector* (*map* (% *x*. (*fst x*, *f* (*snd x*))) *a*) = *apply-matrix f* (*sparse-row-vector a*)
⟨*proof*⟩

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

**lemma** *sparse-row-vector-addmult-spvec*: *sparse-row-vector* (*addmult-spvec* (*y*::*'a*::*lordered-ring*, *a*, *b*)) =
(*sparse-row-vector a*) + (*scalar-mult y* (*sparse-row-vector b*))
⟨*proof*⟩

**lemma** *sorted-smult-spvec*[*rule-format*]: *sorted-spvec a* ⟹ *sorted-spvec* (*smult-spvec y a*)
⟨*proof*⟩

**lemma** *sorted-spvec-addmult-spvec-helper*: ⟦*sorted-spvec* (*addmult-spvec* (*y*, (*a*, *b*) # *arr*, *brr*)); *aa* < *a*; *sorted-spvec* ((*a*, *b*) # *arr*);
*sorted-spvec* ((*aa*, *ba*) # *brr*)⟧ ⟹ *sorted-spvec* ((*aa*, *y* * *ba*) # *addmult-spvec* (*y*, (*a*, *b*) # *arr*, *brr*))
⟨*proof*⟩

**lemma** *sorted-spvec-addmult-spvec-helper2*:
⟦*sorted-spvec* (*addmult-spvec* (*y*, *arr*, (*aa*, *ba*) # *brr*)); *a* < *aa*; *sorted-spvec* ((*a*, *b*) # *arr*); *sorted-spvec* ((*aa*, *ba*) # *brr*)⟧
⟹ *sorted-spvec* ((*a*, *b*) # *addmult-spvec* (*y*, *arr*, (*aa*, *ba*) # *brr*))
⟨*proof*⟩

**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*)))
⟨*proof*⟩

**lemma** *sorted-addmult-spvec*[*rule-format*]: *sorted-spvec a* ⟶ *sorted-spvec b* ⟶ *sorted-spvec* (*addmult-spvec* (*y*, *a*, *b*))
⟨*proof*⟩

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

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

**lemma** *sparse-row-mult-spvec-spmat*[*rule-format*]: *sorted-spvec* (*a*::('*a*::*lordered-ring*) *spvec*) ⟶ *sorted-spvec B* ⟶
*sparse-row-vector* (*mult-spvec-spmat* (*c*, *a*, *B*)) = (*sparse-row-vector c*) + (*sparse-row-vector a*) ∗ (*sparse-row-matrix B*)
⟨*proof*⟩

**lemma** *sorted-mult-spvec-spmat*[*rule-format*]:
*sorted-spvec* (*c*::('*a*::*lordered-ring*) *spvec*) ⟶ *sorted-spmat B* ⟶ *sorted-spvec* (*mult-spvec-spmat* (*c*, *a*, *B*))
⟨*proof*⟩

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

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

**lemma** *sparse-row-mult-spmat*[*rule-format*]:
*sorted-spmat A* ⟶ *sorted-spvec B* ⟶ *sparse-row-matrix* (*mult-spmat A B*) = (*sparse-row-matrix A*) ∗ (*sparse-row-matrix B*)
⟨*proof*⟩

**lemma** *sorted-spvec-mult-spmat*[*rule-format*]:
*sorted-spvec* (*A*::('*a*::*lordered-ring*) *spmat*) ⟶ *sorted-spvec* (*mult-spmat A B*)
⟨*proof*⟩

**lemma** *sorted-spmat-mult-spmat*[*rule-format*]:
*sorted-spmat* (*B*::('*a*::*lordered-ring*) *spmat*) ⟶ *sorted-spmat* (*mult-spmat A B*)
⟨*proof*⟩

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

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

**lemma** *add-spvec-empty1*[*simp*]: *add-spvec* ([], *a*) = *a*
⟨*proof*⟩

**lemma** *add-spvec-empty2*[*simp*]: *add-spvec* (*a*, []) = *a*
⟨*proof*⟩

**lemma** *sparse-row-vector-add*: *sparse-row-vector* (*add-spvec* (*a*,*b*)) = (*sparse-row-vector a*) + (*sparse-row-vector b*)
⟨*proof*⟩

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

**lemma** *sparse-row-add-spmat*: *sparse-row-matrix* (*add-spmat* (*A*, *B*)) = (*sparse-row-matrix A*) + (*sparse-row-matrix B*)
⟨*proof*⟩

**lemma** *sorted-add-spvec-helper1*[*rule-format*]: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

**lemma** *sorted-add-spmat-helper1*[*rule-format*]: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

**lemma** *sorted-add-spvec-helper*[*rule-format*]: *add-spvec* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

**lemma** *sorted-add-spmat-helper*[*rule-format*]: *add-spmat* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

**lemma** *add-spvec-commute*: *add-spvec* (*a*, *b*) = *add-spvec* (*b*, *a*)
⟨*proof*⟩

**lemma** *add-spmat-commute*: *add-spmat* (*a*, *b*) = *add-spmat* (*b*, *a*)
⟨*proof*⟩

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

**lemma** *sorted-add-spmat-helper2*: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟹ *aa* < *a* ⟹ *sorted-spvec* ((*aa*, *ba*) # *brr*) ⟹ *aa* < *ab*
⟨*proof*⟩

**lemma** *sorted-spvec-add-spvec*[*rule-format*]: *sorted-spvec a* ⟶ *sorted-spvec b* ⟶ *sorted-spvec* (*add-spvec* (*a*, *b*))
⟨*proof*⟩

**lemma** *sorted-spvec-add-spmat*[*rule-format*]: *sorted-spvec A* ⟶ *sorted-spvec B* ⟶ *sorted-spvec* (*add-spmat* (*A*, *B*))
⟨*proof*⟩

**lemma** *sorted-spmat-add-spmat*[*rule-format*]: *sorted-spmat A* ⟶ *sorted-spmat B* ⟶ *sorted-spmat* (*add-spmat* (*A*, *B*))
⟨*proof*⟩

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

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

((*snd a* <= *snd b*) & (*le-spvec* (*as*, *bs*)))))

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

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

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

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

**lemma** *disj-matrices-contr2*: *disj-matrices A B* ⟹ *Rep-matrix B j i* ≠ *0* ⟹ *Rep-matrix A j i* = *0*
⟨*proof*⟩

**lemma** *disj-matrices-add*: *disj-matrices A B* ⟹ *disj-matrices C D* ⟹ *disj-matrices A D* ⟹ *disj-matrices B C* ⟹
(*A* + *B* <= *C* + *D*) = (*A* <= *C* & *B* <= (*D*::('*a*::*lordered-ab-group-add*) *matrix*))
⟨*proof*⟩

**lemma** *disj-matrices-zero1*[*simp*]: *disj-matrices 0 B*
⟨*proof*⟩

**lemma** *disj-matrices-zero2*[*simp*]: *disj-matrices A 0*
⟨*proof*⟩

**lemma** *disj-matrices-commute*: *disj-matrices A B* = *disj-matrices B A*
⟨*proof*⟩

**lemma** *disj-matrices-add-le-zero*: *disj-matrices A B* ⟹
(*A* + *B* <= *0*) = (*A* <= *0* & (*B*::('*a*::*lordered-ab-group-add*) *matrix*) <= *0*)
⟨*proof*⟩

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

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

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

**lemma** *disj-sparse-row-singleton*: *i* <= *j* $\Longrightarrow$ *sorted-spvec*((*j*,*y*)#*v*) $\Longrightarrow$ *disj-matrices*
(*sparse-row-vector v*) (*singleton-matrix 0 i x*)
⟨*proof*⟩

**lemma** *disj-matrices-x-add*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices*
(*A*::('*a*::*lordered-ab-group-add*) *matrix*) (*B*+*C*)
⟨*proof*⟩

**lemma** *disj-matrices-add-x*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices*
(*B*+*C*) (*A*::('*a*::*lordered-ab-group-add*) *matrix*)
⟨*proof*⟩

**lemma** *disj-singleton-matrices*[*simp*]: *disj-matrices* (*singleton-matrix j i x*) (*singleton-matrix*
*u v y*) = (*j* $\neq$ *u* | *i* $\neq$ *v* | *x* = *0* | *y* = *0*)
⟨*proof*⟩

**lemma** *disj-move-sparse-vec-mat*[*simplified disj-matrices-commute*]:
*j* <= *a* $\Longrightarrow$ *sorted-spvec*((*a*,*c*)#*as*) $\Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector*
*b*) (*int j*) *i*) (*sparse-row-matrix as*)
⟨*proof*⟩

**lemma** *disj-move-sparse-row-vector-twice*:
*j* $\neq$ *u* $\Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector a*) *j i*) (*move-matrix*
(*sparse-row-vector b*) *u v*)
⟨*proof*⟩

**lemma** *le-spvec-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec a*) $\longrightarrow$ (*sorted-spvec*
*b*) $\longrightarrow$ (*le-spvec* (*a*,*b*)) = (*sparse-row-vector a* <= *sparse-row-vector b*)
⟨*proof*⟩

**lemma** *le-spvec-empty2-sparse-row*[*rule-format*]: (*sorted-spvec b*) $\longrightarrow$ (*le-spvec* (*b*,[])
= (*sparse-row-vector b* <= *0*))
⟨*proof*⟩

**lemma** *le-spvec-empty1-sparse-row*[*rule-format*]: (*sorted-spvec b*) $\longrightarrow$ (*le-spvec* ([],*b*)
= (*0* <= *sparse-row-vector b*))
⟨*proof*⟩

**lemma** *le-spmat-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec A*) ⟶ (*sorted-spmat A*) ⟶ (*sorted-spvec B*) ⟶ (*sorted-spmat B*) ⟶
*le-spmat*(*A*, *B*) = (*sparse-row-matrix A* <= *sparse-row-matrix B*)
⟨*proof*⟩

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

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

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

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

**lemma** *sparse-row-matrix-minus*:
*sparse-row-matrix* (*minus-spmat A*) = − (*sparse-row-matrix A*)
⟨*proof*⟩

**lemma** *Rep-sparse-row-vector-zero*: *x* ≠ *0* ⟹ *Rep-matrix* (*sparse-row-vector v*) *x y* = *0*
⟨*proof*⟩

**lemma** *sparse-row-matrix-abs*:
*sorted-spvec A* ⟹ *sorted-spmat A* ⟹ *sparse-row-matrix* (*abs-spmat A*) = *abs* (*sparse-row-matrix A*)
⟨*proof*⟩

**lemma** *sorted-spvec-minus-spmat*: *sorted-spvec A* ⟹ *sorted-spvec* (*minus-spmat A*)
⟨*proof*⟩

**lemma** *sorted-spvec-abs-spmat*: *sorted-spvec A* ⟹ *sorted-spvec* (*abs-spmat A*)
⟨*proof*⟩

**lemma** *sorted-spmat-minus-spmat*: *sorted-spmat A* ⟹ *sorted-spmat* (*minus-spmat A*)
⟨*proof*⟩

**lemma** *sorted-spmat-abs-spmat*: *sorted-spmat A* ⟹ *sorted-spmat* (*abs-spmat A*)
⟨*proof*⟩

**constdefs**
*diff-spmat* :: ('*a*::*lordered-ring*) *spmat* ⇒ '*a spmat* ⇒ '*a spmat*

*diff-spmat A B == add-spmat (A, minus-spmat B)*

**lemma** *sorted-spmat-diff-spmat*: *sorted-spmat A ⟹ sorted-spmat B ⟹ sorted-spmat (diff-spmat A B)*
⟨*proof*⟩

**lemma** *sorted-spvec-diff-spmat*: *sorted-spvec A ⟹ sorted-spvec B ⟹ sorted-spvec (diff-spmat A B)*
⟨*proof*⟩

**lemma** *sparse-row-diff-spmat*: *sparse-row-matrix (diff-spmat A B ) = (sparse-row-matrix A) − (sparse-row-matrix B)*
⟨*proof*⟩

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

**lemma** *sorted-sparse-matrix-imp-spvec*: *sorted-sparse-matrix A ⟹ sorted-spvec A*
⟨*proof*⟩

**lemma** *sorted-sparse-matrix-imp-spmat*: *sorted-sparse-matrix A ⟹ sorted-spmat A*
⟨*proof*⟩

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

**lemma** *bool1*: *(¬ True) = False* ⟨*proof*⟩
**lemma** *bool2*: *(¬ False) = True* ⟨*proof*⟩
**lemma** *bool3*: *((P::bool) ∧ True) = P* ⟨*proof*⟩
**lemma** *bool4*: *(True ∧ (P::bool)) = P* ⟨*proof*⟩
**lemma** *bool5*: *((P::bool) ∧ False) = False* ⟨*proof*⟩
**lemma** *bool6*: *(False ∧ (P::bool)) = False* ⟨*proof*⟩
**lemma** *bool7*: *((P::bool) ∨ True) = True* ⟨*proof*⟩
**lemma** *bool8*: *(True ∨ (P::bool)) = True* ⟨*proof*⟩
**lemma** *bool9*: *((P::bool) ∨ False) = P* ⟨*proof*⟩
**lemma** *bool10*: *(False ∨ (P::bool)) = P* ⟨*proof*⟩
**lemmas** *boolarith = bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10*

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

**consts**
*pprt-spvec :: ('a::{lordered-ab-group-add}) spvec ⇒ 'a spvec*
*nprt-spvec :: ('a::{lordered-ab-group-add}) spvec ⇒ 'a spvec*
*pprt-spmat :: ('a::{lordered-ab-group-add}) spmat ⇒ 'a spmat*

*nprt-spmat* :: ('*a*::{*lordered-ab-group-add*}) *spmat* ⇒ '*a spmat*

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

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

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

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

**lemma** *pprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) ⟹ *pprt* (*A*+*B*) = *pprt A* + *pprt B*
⟨*proof*⟩

**lemma** *nprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) ⟹ *nprt* (*A*+*B*) = *nprt A* + *nprt B*
⟨*proof*⟩

**lemma** *pprt-singleton*[*simp*]: *pprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*pprt x*)
⟨*proof*⟩

**lemma** *nprt-singleton*[*simp*]: *nprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*nprt x*)
⟨*proof*⟩

**lemma** *less-imp-le*: *a* < *b* ⟹ *a* <= (*b*::-::*order*) ⟨*proof*⟩

**lemma** *sparse-row-vector-pprt*: *sorted-spvec v* ⟹ *sparse-row-vector* (*pprt-spvec v*) = *pprt* (*sparse-row-vector v*)
⟨*proof*⟩

**lemma** *sparse-row-vector-nprt*: *sorted-spvec v* ⟹ *sparse-row-vector* (*nprt-spvec v*) = *nprt* (*sparse-row-vector v*)
⟨*proof*⟩

**lemma** *pprt-move-matrix*: *pprt* (*move-matrix* (*A*::('*a*::*lordered-ring*) *matrix*) *j i*)

$= \mathit{move\text{-}matrix}\ (\mathit{pprt}\ A)\ j\ i$
⟨*proof*⟩

**lemma** *nprt-move-matrix*: $\mathit{nprt}\ (\mathit{move\text{-}matrix}\ (A{::}('a{::}\mathit{lordered\text{-}ring})\ \mathit{matrix})\ j\ i)$
$= \mathit{move\text{-}matrix}\ (\mathit{nprt}\ A)\ j\ i$
⟨*proof*⟩

**lemma** *sparse-row-matrix-pprt*: $\mathit{sorted\text{-}spvec}\ m \Longrightarrow \mathit{sorted\text{-}spmat}\ m \Longrightarrow \mathit{sparse\text{-}row\text{-}matrix}$
$(\mathit{pprt\text{-}spmat}\ m) = \mathit{pprt}\ (\mathit{sparse\text{-}row\text{-}matrix}\ m)$
⟨*proof*⟩

**lemma** *sparse-row-matrix-nprt*: $\mathit{sorted\text{-}spvec}\ m \Longrightarrow \mathit{sorted\text{-}spmat}\ m \Longrightarrow \mathit{sparse\text{-}row\text{-}matrix}$
$(\mathit{nprt\text{-}spmat}\ m) = \mathit{nprt}\ (\mathit{sparse\text{-}row\text{-}matrix}\ m)$
⟨*proof*⟩

**lemma** *sorted-pprt-spvec*: $\mathit{sorted\text{-}spvec}\ v \Longrightarrow \mathit{sorted\text{-}spvec}\ (\mathit{pprt\text{-}spvec}\ v)$
⟨*proof*⟩

**lemma** *sorted-nprt-spvec*: $\mathit{sorted\text{-}spvec}\ v \Longrightarrow \mathit{sorted\text{-}spvec}\ (\mathit{nprt\text{-}spvec}\ v)$
⟨*proof*⟩

**lemma** *sorted-spvec-pprt-spmat*: $\mathit{sorted\text{-}spvec}\ m \Longrightarrow \mathit{sorted\text{-}spvec}\ (\mathit{pprt\text{-}spmat}\ m)$
⟨*proof*⟩

**lemma** *sorted-spvec-nprt-spmat*: $\mathit{sorted\text{-}spvec}\ m \Longrightarrow \mathit{sorted\text{-}spvec}\ (\mathit{nprt\text{-}spmat}\ m)$
⟨*proof*⟩

**lemma** *sorted-spmat-pprt-spmat*: $\mathit{sorted\text{-}spmat}\ m \Longrightarrow \mathit{sorted\text{-}spmat}\ (\mathit{pprt\text{-}spmat}$
$m)$
⟨*proof*⟩

**lemma** *sorted-spmat-nprt-spmat*: $\mathit{sorted\text{-}spmat}\ m \Longrightarrow \mathit{sorted\text{-}spmat}\ (\mathit{nprt\text{-}spmat}$
$m)$
⟨*proof*⟩

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

**lemmas** *sparse-row-matrix-op-simps* =
*sorted-sparse-matrix-imp-spmat sorted-sparse-matrix-imp-spvec*
*sparse-row-add-spmat sorted-spvec-add-spmat sorted-spmat-add-spmat*
*sparse-row-diff-spmat sorted-spvec-diff-spmat sorted-spmat-diff-spmat*
*sparse-row-matrix-minus sorted-spvec-minus-spmat sorted-spmat-minus-spmat*

*sparse-row-mult-spmat sorted-spvec-mult-spmat sorted-spmat-mult-spmat*
*sparse-row-matrix-abs sorted-spvec-abs-spmat sorted-spmat-abs-spmat*
*le-spmat-iff-sparse-row-le*
*sparse-row-matrix-pprt sorted-spvec-pprt-spmat sorted-spmat-pprt-spmat*
*sparse-row-matrix-nprt sorted-spvec-nprt-spmat sorted-spmat-nprt-spmat*

**lemma** *zero-eq-Numeral0*: $(0::-::number\text{-}ring) = Numeral0\ \langle proof \rangle$

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

**lemma** *spm-mult-le-dual-prts*:
**assumes**
*sorted-sparse-matrix A1*
*sorted-sparse-matrix A2*
*sorted-sparse-matrix c1*
*sorted-sparse-matrix c2*
*sorted-sparse-matrix y*
*sorted-sparse-matrix r1*
*sorted-sparse-matrix r2*
*sorted-spvec b*
*le-spmat* ([], *y*)
*sparse-row-matrix A1* $\leq$ *A*
*A* $\leq$ *sparse-row-matrix A2*
*sparse-row-matrix c1* $\leq$ *c*
*c* $\leq$ *sparse-row-matrix c2*
*sparse-row-matrix r1* $\leq$ *x*
*x* $\leq$ *sparse-row-matrix r2*
*A* $*$ *x* $\leq$ *sparse-row-matrix* (*b*::('*a*::*lordered-ring*) *spmat*)
**shows**
*c* $*$ *x* $\leq$ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
(*let s1* = *diff-spmat c1* (*mult-spmat y A2*); *s2* = *diff-spmat c2* (*mult-spmat y A1*) *in*
*add-spmat* (*mult-spmat* (*pprt-spmat s2*) (*pprt-spmat r2*), *add-spmat* (*mult-spmat* (*pprt-spmat s1*) (*nprt-spmat r2*),
*add-spmat* (*mult-spmat* (*nprt-spmat s2*) (*pprt-spmat r1*), *mult-spmat* (*nprt-spmat*

*s1*) (*nprt-spmat r1*))))))))
⟨*proof*⟩

**lemma** *spm-mult-le-dual-prts-no-let*:
**assumes**
*sorted-sparse-matrix A1*
*sorted-sparse-matrix A2*
*sorted-sparse-matrix c1*
*sorted-sparse-matrix c2*
*sorted-sparse-matrix y*
*sorted-sparse-matrix r1*
*sorted-sparse-matrix r2*
*sorted-spvec b*
*le-spmat* ([], *y*)
*sparse-row-matrix A1* ≤ *A*
*A* ≤ *sparse-row-matrix A2*
*sparse-row-matrix c1* ≤ *c*
*c* ≤ *sparse-row-matrix c2*
*sparse-row-matrix r1* ≤ *x*
*x* ≤ *sparse-row-matrix r2*
*A* ∗ *x* ≤ *sparse-row-matrix* (*b*::('*a*::*lordered-ring*) *spmat*)
**shows**
*c* ∗ *x* ≤ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
*mult-est-spmat r1 r2* (*diff-spmat c1* (*mult-spmat y A2*)) (*diff-spmat c2* (*mult-spmat y A1*))))
⟨*proof*⟩

**end**

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

**end**

**theory** *Compute-Oracle* **imports** *Pure*
**uses** *am.ML am-compiler.ML am-interpreter.ML am-ghc.ML am-sml.ML report.ML compute.ML linker.ML*
**begin**

⟨*ML*⟩

**end**
**theory** *ComputeHOL*
**imports** *Main* ~~/*src*/*Tools*/*Compute-Oracle*/*Compute-Oracle*
**begin**

**lemma** *Trueprop-eq-eq*: *Trueprop X* == (*X* == *True*) ⟨*proof*⟩
**lemma** *meta-eq-trivial*: *x* == *y* ⟹ *x* == *y* ⟨*proof*⟩
**lemma** *meta-eq-imp-eq*: *x* == *y* ⟹ *x* = *y* ⟨*proof*⟩
**lemma** *eq-trivial*: *x* = *y* ⟹ *x* = *y* ⟨*proof*⟩
**lemma** *bool-to-true*: *x* :: *bool* ⟹ *x* == *True* ⟨*proof*⟩
**lemma** *transmeta-1*: *x* = *y* ⟹ *y* == *z* ⟹ *x* = *z* ⟨*proof*⟩
**lemma** *transmeta-2*: *x* == *y* ⟹ *y* = *z* ⟹ *x* = *z* ⟨*proof*⟩
**lemma** *transmeta-3*: *x* == *y* ⟹ *y* == *z* ⟹ *x* = *z* ⟨*proof*⟩

**lemma** *If-True*: *If True* = (λ *x y*. *x*) ⟨*proof*⟩
**lemma** *If-False*: *If False* = (λ *x y*. *y*) ⟨*proof*⟩

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

**lemma** *bool1*: (¬ *True*) = *False* ⟨*proof*⟩
**lemma** *bool2*: (¬ *False*) = *True* ⟨*proof*⟩
**lemma** *bool3*: (*P* ∧ *True*) = *P* ⟨*proof*⟩
**lemma** *bool4*: (*True* ∧ *P*) = *P* ⟨*proof*⟩
**lemma** *bool5*: (*P* ∧ *False*) = *False* ⟨*proof*⟩
**lemma** *bool6*: (*False* ∧ *P*) = *False* ⟨*proof*⟩
**lemma** *bool7*: (*P* ∨ *True*) = *True* ⟨*proof*⟩
**lemma** *bool8*: (*True* ∨ *P*) = *True* ⟨*proof*⟩
**lemma** *bool9*: (*P* ∨ *False*) = *P* ⟨*proof*⟩
**lemma** *bool10*: (*False* ∨ *P*) = *P* ⟨*proof*⟩
**lemma** *bool11*: (*True* ⟶ *P*) = *P* ⟨*proof*⟩
**lemma** *bool12*: (*P* ⟶ *True*) = *True* ⟨*proof*⟩
**lemma** *bool13*: (*True* ⟶ *P*) = *P* ⟨*proof*⟩
**lemma** *bool14*: (*P* ⟶ *False*) = (¬ *P*) ⟨*proof*⟩
**lemma** *bool15*: (*False* ⟶ *P*) = *True* ⟨*proof*⟩
**lemma** *bool16*: (*False* = *False*) = *True* ⟨*proof*⟩
**lemma** *bool17*: (*True* = *True*) = *True* ⟨*proof*⟩
**lemma** *bool18*: (*False* = *True*) = *False* ⟨*proof*⟩
**lemma** *bool19*: (*True* = *False*) = *False* ⟨*proof*⟩

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

**lemma** *compute-fst*: *fst* (*x*,*y*) = *x* ⟨*proof*⟩
**lemma** *compute-snd*: *snd* (*x*,*y*) = *y* ⟨*proof*⟩
**lemma** *compute-pair-eq*: ((*a*, *b*) = (*c*, *d*)) = (*a* = *c* ∧ *b* = *d*) ⟨*proof*⟩

**lemma** *prod-case-simp*: *prod-case f (x,y) = f x y ⟨proof⟩*

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

**lemma** *compute-the*: *the (Some x) = x ⟨proof⟩*
**lemma** *compute-None-Some-eq*: *(None = Some x) = False ⟨proof⟩*
**lemma** *compute-Some-None-eq*: *(Some x = None) = False ⟨proof⟩*
**lemma** *compute-None-None-eq*: *(None = None) = True ⟨proof⟩*
**lemma** *compute-Some-Some-eq*: *(Some x = Some y) = (x = y) ⟨proof⟩*

**definition**
*option-case-compute* :: *'b option ⇒ 'a ⇒ ('b ⇒ 'a) ⇒ 'a*
**where**
*option-case-compute opt a f = option-case a f opt*

**lemma** *option-case-compute*: *option-case = (λ a f opt. option-case-compute opt a f)*
*⟨proof⟩*

**lemma** *option-case-compute-None*: *option-case-compute None = (λ a f. a)*
*⟨proof⟩*

**lemma** *option-case-compute-Some*: *option-case-compute (Some x) = (λ a f. f x)*
*⟨proof⟩*

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

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

**lemma** *length-cons*:*length (x#xs) = 1 + (length xs)*
*⟨proof⟩*

**lemma** *length-nil*: *length [] = 0*
*⟨proof⟩*

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

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

**lemma** *list-case-compute*: *list-case* = (λ (*a*::'*a*) *f* (*l*::'*b list*). *list-case-compute l a f*)
⟨*proof*⟩

**lemma** *list-case-compute-empty*: *list-case-compute* ([]::'*b list*) = (λ (*a*::'*a*) *f*. *a*)
⟨*proof*⟩

**lemma** *list-case-compute-cons*: *list-case-compute* (*u*#*v*) = (λ (*a*::'*a*) *f*. (*f* (*u*::'*b*) *v*))
⟨*proof*⟩

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

**lemma** *compute-list-nth*: ((*x*#*xs*) ! *n*) = (*if n* = *0 then x else* (*xs* ! (*n* − *1*)))
⟨*proof*⟩

**lemmas** *compute-list* = *compute-list-case 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*⟩

**end**

**theory** *ComputeNumeral*
**imports** *ComputeHOL* ~~/*src*/*HOL*/*Real*/*Float*
**begin**

**lemmas** *bitnorm* = *normalize-bin-simps*

**lemma** *neg1*: *neg Int.Pls* = *False* ⟨*proof*⟩
**lemma** *neg2*: *neg Int.Min* = *True* ⟨*proof*⟩
**lemma** *neg3*: *neg* (*Int.Bit0 x*) = *neg x* ⟨*proof*⟩

**lemma** *neg4*: *neg* (*Int.Bit1 x*) = *neg x* ⟨*proof*⟩
**lemmas** *bitneg* = *neg1 neg2 neg3 neg4*

**lemma** *iszero1*: *iszero Int.Pls* = *True* ⟨*proof*⟩
**lemma** *iszero2*: *iszero Int.Min* = *False* ⟨*proof*⟩
**lemma** *iszero3*: *iszero* (*Int.Bit0 x*) = *iszero x* ⟨*proof*⟩
**lemma** *iszero4*: *iszero* (*Int.Bit1 x*) = *False* ⟨*proof*⟩
**lemmas** *bitiszero* = *iszero1 iszero2 iszero3 iszero4*

**constdefs**
*lezero x* == (*x* ≤ *0*)
**lemma** *lezero1*: *lezero Int.Pls* = *True* ⟨*proof*⟩
**lemma** *lezero2*: *lezero Int.Min* = *True* ⟨*proof*⟩
**lemma** *lezero3*: *lezero* (*Int.Bit0 x*) = *lezero x* ⟨*proof*⟩
**lemma** *lezero4*: *lezero* (*Int.Bit1 x*) = *neg x* ⟨*proof*⟩
**lemmas** *bitlezero* = *lezero1 lezero2 lezero3 lezero4*

**lemma** *biteq1*: (*Int.Pls* = *Int.Pls*) = *True* ⟨*proof*⟩
**lemma** *biteq2*: (*Int.Min* = *Int.Min*) = *True* ⟨*proof*⟩
**lemma** *biteq3*: (*Int.Pls* = *Int.Min*) = *False* ⟨*proof*⟩
**lemma** *biteq4*: (*Int.Min* = *Int.Pls*) = *False* ⟨*proof*⟩
**lemma** *biteq5*: (*Int.Bit0 x* = *Int.Bit0 y*) = (*x* = *y*) ⟨*proof*⟩
**lemma** *biteq6*: (*Int.Bit1 x* = *Int.Bit1 y*) = (*x* = *y*) ⟨*proof*⟩
**lemma** *biteq7*: (*Int.Bit0 x* = *Int.Bit1 y*) = *False* ⟨*proof*⟩
**lemma** *biteq8*: (*Int.Bit1 x* = *Int.Bit0 y*) = *False* ⟨*proof*⟩
**lemma** *biteq9*: (*Int.Pls* = *Int.Bit0 x*) = (*Int.Pls* = *x*) ⟨*proof*⟩
**lemma** *biteq10*: (*Int.Pls* = *Int.Bit1 x*) = *False* ⟨*proof*⟩
**lemma** *biteq11*: (*Int.Min* = *Int.Bit0 x*) = *False* ⟨*proof*⟩
**lemma** *biteq12*: (*Int.Min* = *Int.Bit1 x*) = (*Int.Min* = *x*) ⟨*proof*⟩
**lemma** *biteq13*: (*Int.Bit0 x* = *Int.Pls*) = (*x* = *Int.Pls*) ⟨*proof*⟩
**lemma** *biteq14*: (*Int.Bit1 x* = *Int.Pls*) = *False* ⟨*proof*⟩
**lemma** *biteq15*: (*Int.Bit0 x* = *Int.Min*) = *False* ⟨*proof*⟩
**lemma** *biteq16*: (*Int.Bit1 x* = *Int.Min*) = (*x* = *Int.Min*) ⟨*proof*⟩
**lemmas** *biteq* = *biteq1 biteq2 biteq3 biteq4 biteq5 biteq6 biteq7 biteq8 biteq9 biteq10 biteq11 biteq12 biteq13 biteq14 biteq15 biteq16*

**lemma** *bitless1*: (*Int.Pls* < *Int.Min*) = *False* ⟨*proof*⟩
**lemma** *bitless2*: (*Int.Pls* < *Int.Pls*) = *False* ⟨*proof*⟩
**lemma** *bitless3*: (*Int.Min* < *Int.Pls*) = *True* ⟨*proof*⟩
**lemma** *bitless4*: (*Int.Min* < *Int.Min*) = *False* ⟨*proof*⟩
**lemma** *bitless5*: (*Int.Bit0 x* < *Int.Bit0 y*) = (*x* < *y*) ⟨*proof*⟩
**lemma** *bitless6*: (*Int.Bit1 x* < *Int.Bit1 y*) = (*x* < *y*) ⟨*proof*⟩
**lemma** *bitless7*: (*Int.Bit0 x* < *Int.Bit1 y*) = (*x* ≤ *y*) ⟨*proof*⟩
**lemma** *bitless8*: (*Int.Bit1 x* < *Int.Bit0 y*) = (*x* < *y*) ⟨*proof*⟩
**lemma** *bitless9*: (*Int.Pls* < *Int.Bit0 x*) = (*Int.Pls* < *x*) ⟨*proof*⟩

**lemma** *bitless10*: $(Int.Pls < Int.Bit1\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
**lemma** *bitless11*: $(Int.Min < Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
**lemma** *bitless12*: $(Int.Min < Int.Bit1\ x) = (Int.Min < x)$ ⟨*proof*⟩
**lemma** *bitless13*: $(Int.Bit0\ x < Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
**lemma** *bitless14*: $(Int.Bit1\ x < Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
**lemma** *bitless15*: $(Int.Bit0\ x < Int.Min) = (x < Int.Pls)$ ⟨*proof*⟩
**lemma** *bitless16*: $(Int.Bit1\ x < Int.Min) = (x < Int.Min)$ ⟨*proof*⟩
**lemmas** *bitless = bitless1 bitless2 bitless3 bitless4 bitless5 bitless6 bitless7 bitless8 bitless9 bitless10 bitless11 bitless12 bitless13 bitless14 bitless15 bitless16*

**lemma** *bitle1*: $(Int.Pls \leq Int.Min) = False$ ⟨*proof*⟩
**lemma** *bitle2*: $(Int.Pls \leq Int.Pls) = True$ ⟨*proof*⟩
**lemma** *bitle3*: $(Int.Min \leq Int.Pls) = True$ ⟨*proof*⟩
**lemma** *bitle4*: $(Int.Min \leq Int.Min) = True$ ⟨*proof*⟩
**lemma** *bitle5*: $(Int.Bit0\ x \leq Int.Bit0\ y) = (x \leq y)$ ⟨*proof*⟩
**lemma** *bitle6*: $(Int.Bit1\ x \leq Int.Bit1\ y) = (x \leq y)$ ⟨*proof*⟩
**lemma** *bitle7*: $(Int.Bit0\ x \leq Int.Bit1\ y) = (x \leq y)$ ⟨*proof*⟩
**lemma** *bitle8*: $(Int.Bit1\ x \leq Int.Bit0\ y) = (x < y)$ ⟨*proof*⟩
**lemma** *bitle9*: $(Int.Pls \leq Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
**lemma** *bitle10*: $(Int.Pls \leq Int.Bit1\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
**lemma** *bitle11*: $(Int.Min \leq Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
**lemma** *bitle12*: $(Int.Min \leq Int.Bit1\ x) = (Int.Min \leq x)$ ⟨*proof*⟩
**lemma** *bitle13*: $(Int.Bit0\ x \leq Int.Pls) = (x \leq Int.Pls)$ ⟨*proof*⟩
**lemma** *bitle14*: $(Int.Bit1\ x \leq Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
**lemma** *bitle15*: $(Int.Bit0\ x \leq Int.Min) = (x < Int.Pls)$ ⟨*proof*⟩
**lemma** *bitle16*: $(Int.Bit1\ x \leq Int.Min) = (x \leq Int.Min)$ ⟨*proof*⟩
**lemmas** *bitle = bitle1 bitle2 bitle3 bitle4 bitle5 bitle6 bitle7 bitle8 bitle9 bitle10 bitle11 bitle12 bitle13 bitle14 bitle15 bitle16*

**lemmas** *bitsucc = succ-bin-simps*

**lemmas** *bitpred = pred-bin-simps*

**lemmas** *bituminus = minus-bin-simps*

**lemmas** *bitadd = add-bin-simps*

**lemma** *mult-Pls-right*: $x * Int.Pls = Int.Pls$ ⟨*proof*⟩
**lemma** *mult-Min-right*: $x * Int.Min = -\ x$ ⟨*proof*⟩
**lemma** *multb0x*: $(Int.Bit0\ x) * y = Int.Bit0\ (x * y)$ ⟨*proof*⟩
**lemma** *multxb0*: $x * (Int.Bit0\ y) = Int.Bit0\ (x * y)$ ⟨*proof*⟩
**lemma** *multb1*: $(Int.Bit1\ x) * (Int.Bit1\ y) = Int.Bit1\ (Int.Bit0\ (x * y) + x + y)$ ⟨*proof*⟩

**lemmas** *bitmul = mult-Pls mult-Min mult-Pls-right mult-Min-right multb0x multxb0 multb1*

**lemmas** *bitarith = bitnorm bitiszero bitneg bitlezero biteq bitless bitle bitsucc bitpred bituminus bitadd bitmul*

**constdefs**
*nat-norm-number-of* (*x*::*nat*) == *x*

**lemma** *nat-norm-number-of*: *nat-norm-number-of* (*number-of w*) = (*if lezero w then 0 else number-of w*)
⟨*proof*⟩

**lemma** *natnorm0*: (*0*::*nat*) = *number-of* (*Int.Pls*) ⟨*proof*⟩
**lemma** *natnorm1*: (*1* :: *nat*) = *number-of* (*Int.Bit1 Int.Pls*) ⟨*proof*⟩
**lemmas** *natnorm = natnorm0 natnorm1 nat-norm-number-of*

**lemma** *natsuc*: *Suc* (*number-of x*) = (*if neg x then 1 else number-of* (*Int.succ x*)) ⟨*proof*⟩

**lemma** *natadd*: *number-of x* + ((*number-of y*)::*nat*) = (*if neg x then* (*number-of y*) *else* (*if neg y then number-of x else* (*number-of* (*x* + *y*))))
⟨*proof*⟩

**lemma** *natsub*: (*number-of x*) − ((*number-of y*)::*nat*) =
(*if neg x then 0 else* (*if neg y then number-of x else* (*nat-norm-number-of* (*number-of* (*x* + (− *y*)))))))
⟨*proof*⟩

**lemma** *natmul*: (*number-of x*) ∗ ((*number-of y*)::*nat*) =
(*if neg x then 0 else* (*if neg y then 0 else number-of* (*x* ∗ *y*)))
⟨*proof*⟩

**lemma** *nateq*: (((*number-of x*)::*nat*) = (*number-of y*)) = ((*lezero x* ∧ *lezero y*) ∨ (*x* = *y*))
⟨*proof*⟩

**lemma** *natless*: (((*number-of x*)::*nat*) < (*number-of y*)) = ((*x* < *y*) ∧ (¬ (*lezero y*)))
⟨*proof*⟩

**lemma** *natle*: (((*number-of x*)::*nat*) ≤ (*number-of y*)) = (*y* < *x* ⟶ *lezero x*)
⟨*proof*⟩

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

**lemmas** *compute-natarith* = *bitarith natnorm natsuc natadd natsub natmul nateq natless natle natfac.simps*

**lemma** *number-eq*: (((*number-of x*)::′*a*::{*number-ring*, *ordered-idom*}) = (*number-of y*)) = (*x* = *y*)
  ⟨*proof*⟩

**lemma** *number-le*: (((*number-of x*)::′*a*::{*number-ring*, *ordered-idom*}) ≤ (*number-of y*)) = (*x* ≤ *y*)
  ⟨*proof*⟩

**lemma** *number-less*: (((*number-of x*)::′*a*::{*number-ring*, *ordered-idom*}) < (*number-of y*)) = (*x* < *y*)
  ⟨*proof*⟩

**lemma** *number-diff*: ((*number-of x*)::′*a*::{*number-ring*, *ordered-idom*}) − *number-of y* = *number-of* (*x* + (− *y*))
  ⟨*proof*⟩

**lemmas** *number-norm* = *number-of-Pls*[*symmetric*] *numeral-1-eq-1*[*symmetric*]

**lemmas** *compute-numberarith* = *number-of-minus*[*symmetric*] *number-of-add*[*symmetric*] *number-diff number-of-mult*[*symmetric*] *number-norm number-eq number-le number-less*

**lemma** *compute-real-of-nat-number-of*: *real* ((*number-of v*)::*nat*) = (*if neg v then 0 else number-of v*)
  ⟨*proof*⟩

**lemma** *compute-nat-of-int-number-of*: *nat* ((*number-of v*)::*int*) = (*number-of v*)
  ⟨*proof*⟩

**lemmas** *compute-num-conversions* = *compute-real-of-nat-number-of compute-nat-of-int-number-of real-number-of*

**lemmas** *zpowerarith* = *zpower-number-of-even*
  *zpower-number-of-odd*[*simplified zero-eq-Numeral0-nring one-eq-Numeral1-nring*]
  *zpower-Pls zpower-Min*

**lemma** *adjust*: *adjust b* (*q*, *r*) = (*if* 0 ≤ *r* − *b then* (2 ∗ *q* + 1, *r* − *b*) *else* (2 ∗ *q*, *r*))
  ⟨*proof*⟩

**lemma** *negateSnd*: *negateSnd* (*q*, *r*) = (*q*, −*r*)

⟨*proof*⟩

**lemma** *divAlg*: *divAlg* (*a*, *b*) = (*if* 0≤*a then*
*if* 0≤*b then posDivAlg a b*
*else if a*=0 *then* (0, 0)
*else negateSnd* (*negDivAlg* (−*a*) (−*b*))
*else*
*if* 0<*b then negDivAlg a b*
*else negateSnd* (*posDivAlg* (−*a*) (−*b*)))
⟨*proof*⟩

**lemmas** *compute-div-mod* = *div-def mod-def divAlg adjust negateSnd posDivAlg.simps negDivAlg.simps*

**lemma** *even-Pls*: *even* (*Int.Pls*) = *True*
⟨*proof*⟩

**lemma** *even-Min*: *even* (*Int.Min*) = *False*
⟨*proof*⟩

**lemma** *even-B0*: *even* (*Int.Bit0 x*) = *True*
⟨*proof*⟩

**lemma** *even-B1*: *even* (*Int.Bit1 x*) = *False*
⟨*proof*⟩

**lemma** *even-number-of*: *even* ((*number-of w*)::*int*) = *even w*
⟨*proof*⟩

**lemmas** *compute-even* = *even-Pls even-Min even-B0 even-B1 even-number-of*

**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** *FloatSparseMatrix* ~~/*src*/*HOL*/*Tools*/*ComputeNumeral*
**uses** *Cplex-tools.ML CplexMatrixConverter.ML FloatSparseMatrixBuilder.ML fspmlp.ML*

**begin**

**end**

**theory** *MatrixLP*
**imports** *Cplex*
**uses** *matrixlp.ML*
**begin**
**end**