Matrix

Steven Obua

April 19, 2009

theory *Matrix*
imports *Main*
begin

types *'a infmatrix* = *nat* ⇒ *nat* ⇒ *'a*

definition *nonzero-positions* :: (*nat* ⇒ *nat* ⇒ *'a*::*zero*) ⇒ *nat* × *nat* ⇒ *bool*
where
nonzero-positions A = {*pos*. *A* (*fst pos*) (*snd pos*) ~= *0*}

typedef *'a matrix* = {(*f*::(*nat* ⇒ *nat* ⇒ *'a*::*zero*)). *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*⟩

instantiation *matrix* :: (*zero*) *zero*
begin

definition *zero-matrix-def* [*code del*]: *0* = *Abs-matrix* (λ*j i*. *0*)

instance ⟨*proof*⟩

end

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

definition
le-matrix-def: *A* ≤ *B* ⟷ (∀ *j i*. *Rep-matrix A j i* ≤ *Rep-matrix B j i*)

definition
less-def: *A* < (*B*::'*a matrix*) ⟷ *A* ≤ *B* ∧ ¬ *B* ≤ *A*

instance ⟨*proof*⟩

end

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

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

definition [*code del*]: *inf = combine-matrix inf*

definition [*code del*]: *sup* = *combine-matrix sup*

instance
⟨*proof*⟩

end

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

definition
plus-matrix-def [*code del*]: *A* + *B* = *combine-matrix* (*op* +) *A B*

instance ⟨*proof*⟩

end

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

definition
minus-matrix-def [*code del*]: − *A* = *apply-matrix uminus A*

instance ⟨*proof*⟩

end

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

definition
diff-matrix-def [*code del*]: *A* − *B* = *combine-matrix* (*op* −) *A B*

instance ⟨*proof*⟩

end

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

definition
times-matrix-def [*code del*]: *A* ∗ *B* = *mult-matrix* (*op* ∗) (*op* +) *A B*

instance ⟨*proof*⟩

end

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

begin

definition
abs-matrix-def [*code del*]: *abs* (*A* :: *'a matrix*) = *sup A* (− *A*)

instance ⟨*proof*⟩

end

instance *matrix* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *matrix* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

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

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

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

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

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

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

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

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*)
⟨*proof*⟩

lemma *Rep-matrix-mult*: *Rep-matrix* ((*a*::(*'a*::*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*::*monoid-add*) *matrix*) + *b*) = (*apply-matrix f a*) + (*apply-matrix f b*)

⟨*proof*⟩

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

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

lemma *ncols-mult*: *ncols* ((*A*::('*a*::*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*::{*ring-1*}) *matrix*) ∗ (*one-matrix n*) = *A*
⟨*proof*⟩

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

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

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

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

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

$= -$ *transpose-matrix* $(A::'a\ matrix)$
⟨*proof*⟩

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

lemma *right-inverse-matrix-dim*: *right-inverse-matrix* $A\ X \Longrightarrow nrows\ A = ncols\ X$
⟨*proof*⟩

lemma *left-inverse-matrix-dim*: *left-inverse-matrix* $A\ Y \Longrightarrow ncols\ A = nrows\ Y$
⟨*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*: $[\![\ inverse\text{-}matrix\ A\ X;\ inverse\text{-}matrix\ A\ Y\ ]\!] \Longrightarrow X = Y$
⟨*proof*⟩

lemma *one-matrix-inverse*: *inverse-matrix* $(one\text{-}matrix\ n)\ (one\text{-}matrix\ n)$
⟨*proof*⟩

lemma *zero-imp-mult-zero*: $(a::'a::ring) = 0 \mid b = 0 \Longrightarrow a * b = 0$
⟨*proof*⟩

lemma *Rep-matrix-zero-imp-mult-zero*:
$!\ j\ i\ k.\ (Rep\text{-}matrix\ A\ j\ k = 0) \mid (Rep\text{-}matrix\ B\ k\ i) = 0 \Longrightarrow A * B = (0::('a::lordered\text{-}ring)\ matrix)$
⟨*proof*⟩

lemma *add-nrows*: *nrows* $(A::('a::monoid\text{-}add)\ matrix) <= u \Longrightarrow nrows\ B <= u \Longrightarrow nrows\ (A + B) <= u$
⟨*proof*⟩

lemma *move-matrix-row-mult*: *move-matrix* $((A::('a::ring)\ matrix) * B)\ j\ 0 = (move\text{-}matrix\ A\ j\ 0) * B$
⟨*proof*⟩

lemma *move-matrix-col-mult*: *move-matrix* $((A::('a::ring)\ matrix) * B)\ 0\ i = A$

$*$ (*move-matrix B 0 i*)
⟨*proof*⟩

lemma *move-matrix-add*: ((*move-matrix* (*A* + *B*) *j i*)::((′*a*::*monoid-add*) *matrix*))
= (*move-matrix A j i*) + (*move-matrix B j i*)
⟨*proof*⟩

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

constdefs
scalar-mult :: (′*a*::*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*::-::*group-add*)) *x y* = − (*Rep-matrix*
A x y)
⟨*proof*⟩

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

end

theory *SparseMatrix*
imports *Matrix*
begin

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

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

definition *sparse-row-matrix* :: (*'a*::*ab-group-add*) *spmat* ⇒ *'a matrix* **where**
sparse-row-matrix-def: *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
⟨*proof*⟩

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

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

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*⟩

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

primrec *sorted-spmat* :: *'a spmat* ⇒ *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*
⟨*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*⟩

primrec *minus-spvec* :: ('*a*::*ab-group-add*) *spvec* ⇒ '*a spvec* **where**
minus-spvec [] = []
| *minus-spvec* (*a*#*as*) = (*fst a*, −(*snd a*))#(*minus-spvec as*)

primrec *abs-spvec* :: ('*a*::*lordered-ab-group-add-abs*) *spvec* ⇒ '*a spvec* **where**
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*⟩

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

lemma *sparse-row-vector-abs*:
sorted-spvec (*v* :: '*a*::*lordered-ring spvec*) ⟹ *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 $\Longrightarrow$ *sorted-spvec* (*abs-spvec v*)
⟨*proof*⟩

definition
smult-spvec y = *map* (% *a*. (*fst a*, *y* ∗ *snd a*))

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*⟩

consts *addmult-spvec* :: ('*a*::*ring*) ∗ '*a spvec* ∗ '*a spvec* ⇒ '*a spvec*
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*)) $\Longrightarrow$ (*f*::'*a*⇒('*a*::*lordered-ring*)) *0* = *0* $\Longrightarrow$
sparse-row-vector (*map* (% *x*. (*fst x*, *f* (*snd x*))) *a*) = *apply-matrix f* (*sparse-row-vector a*)
⟨*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* $\Longrightarrow$ *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*⟩

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

lemma *sorted-add-spvec-helper1*[*rule-format*]: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*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 ⟹*
(0 <= A + B) = (0 <= A & 0 <= (B::('a::lordered-ab-group-add) matrix))
⟨*proof*⟩

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

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

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

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

lemma *disj-matrices-add-x*: *disj-matrices A B ⟹ disj-matrices A C ⟹ 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 ≠ u | i ≠ v | x = 0 | y = 0)*
⟨*proof*⟩

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

lemma *disj-move-sparse-row-vector-twice*:
j ≠ u ⟹ 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*) ⟶ (*sorted-spvec b*) ⟶ (*le-spvec* (*a*,*b*)) = (*sparse-row-vector a* <= *sparse-row-vector b*)
⟨*proof*⟩

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

lemma *le-spvec-empty1-sparse-row*[*rule-format*]: (*sorted-spvec b*) ⟶ (*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* $\Longrightarrow$ *sorted-spvec* (*abs-spmat A*)
⟨*proof*⟩

lemma *sorted-spmat-minus-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat* (*minus-spmat A*)
⟨*proof*⟩

lemma *sorted-spmat-abs-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat* (*abs-spmat A*)
⟨*proof*⟩

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

lemma *sorted-spmat-diff-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat B* $\Longrightarrow$ *sorted-spmat* (*diff-spmat A B*)
⟨*proof*⟩

lemma *sorted-spvec-diff-spmat*: *sorted-spvec A* $\Longrightarrow$ *sorted-spvec B* $\Longrightarrow$ *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* $\Rightarrow$ *bool*
sorted-sparse-matrix A == (*sorted-spvec A*) & (*sorted-spmat A*)

lemma *sorted-sparse-matrix-imp-spvec*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spvec A*
⟨*proof*⟩

lemma *sorted-sparse-matrix-imp-spmat*: *sorted-sparse-matrix A* $\Longrightarrow$ *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*: $((P::bool) \vee True) = True$ ⟨*proof*⟩
lemma *bool8*: $(True \vee (P::bool)) = True$ ⟨*proof*⟩
lemma *bool9*: $((P::bool) \vee False) = P$ ⟨*proof*⟩
lemma *bool10*: $(False \vee (P::bool)) = P$ ⟨*proof*⟩
lemmas *boolarith = bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10*

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

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

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

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

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

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

lemma *pprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) ⟹ *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*: $a < b \Longrightarrow a <= (b::-::order)$ ⟨*proof*⟩

lemma *sparse-row-vector-pprt*: *sorted-spvec* $(v :: 'a::lordered\text{-}ring\ spvec) \Longrightarrow$ *sparse-row-vector* (*pprt-spvec* v) = *pprt* (*sparse-row-vector* v)
⟨*proof*⟩

lemma *sparse-row-vector-nprt*: *sorted-spvec* $(v :: 'a::lordered\text{-}ring\ spvec) \Longrightarrow$ *sparse-row-vector* (*nprt-spvec* v) = *nprt* (*sparse-row-vector* v)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

lemma *zero-eq-Numeral0*: (*0*::-::*number-ring*) = *Numeral0* ⟨*proof*⟩

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

end

theory *LP*
imports *Main*
begin

lemma *linprog-dual-estimate*:
assumes

$A * x \leq (b::'a::lordered\text{-}ring)$
$0 \leq y$
$abs\ (A - A') \leq \delta A$
$b \leq b'$
$abs\ (c - c') \leq \delta c$
$abs\ x \leq r$
shows
$c * x \leq y * b' + (y * \delta A + abs\ (y * A' - c') + \delta c) * r$
⟨*proof*⟩

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

lemma *mult-le-prts*:
assumes
$a1 <= (a::'a::lordered\text{-}ring)$
$a <= a2$
$b1 <= b$
$b <= b2$
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
$A * x \leq (b::'a::lordered\text{-}ring)$
$0 \leq y$
$A1 \leq A$
$A \leq A2$
$c1 \leq c$
$c \leq c2$
$r1 \leq x$
$x \leq r2$
shows
$c * x \leq y * b + (let\ s1 = c1 - y * A2;\ s2 = c2 - y * A1\ in\ pprt\ s2 * pprt\ r2 + pprt\ s1 * nprt\ r2 + nprt\ s2 * pprt\ r1 + nprt\ s1 * nprt\ r1)$
(**is** - <= - + ?C)
⟨*proof*⟩

end

1 Floating Point Representation of the Reals

theory *ComputeFloat*

imports *Complex-Main*
uses ~~/*src*/*Tools*/*float.ML* (~~/*src*/*HOL*/*Tools*/*float-arith.ML*)
begin

definition
pow2 :: *int* ⇒ *real* **where**
pow2 a = (*if* (*0* <= *a*) *then* (*2*^(*nat a*)) *else* (*inverse* (*2*^(*nat* (−*a*)))))

definition
float :: *int* ∗ *int* ⇒ *real* **where**
float x = *real* (*fst x*) ∗ *pow2* (*snd x*)

lemma *pow2-0*[*simp*]: *pow2 0* = *1*
⟨*proof*⟩

lemma *pow2-1*[*simp*]: *pow2 1* = *2*
⟨*proof*⟩

lemma *pow2-neg*: *pow2 x* = *inverse* (*pow2* (−*x*))
⟨*proof*⟩

lemma *pow2-add1*: *pow2* (*1* + *a*) = *2* ∗ (*pow2 a*)
⟨*proof*⟩

lemma *pow2-add*: *pow2* (*a*+*b*) = (*pow2 a*) ∗ (*pow2 b*)
⟨*proof*⟩

lemma *float* (*a*, *e*) + *float* (*b*, *e*) = *float* (*a* + *b*, *e*)
⟨*proof*⟩

definition
int-of-real :: *real* ⇒ *int* **where**
int-of-real x = (*SOME y*. *real y* = *x*)

definition
real-is-int :: *real* ⇒ *bool* **where**
real-is-int x = (*EX* (*u*::*int*). *x* = *real u*)

lemma *real-is-int-def2*: *real-is-int x* = (*x* = *real* (*int-of-real x*))
⟨*proof*⟩

lemma *float-transfer*: *real-is-int* ((*real a*)∗(*pow2 c*)) ⟹ *float* (*a*, *b*) = *float* (*int-of-real* ((*real a*)∗(*pow2 c*)), *b* − *c*)
⟨*proof*⟩

lemma *pow2-int*: *pow2* (*int c*) = *2*^*c*
⟨*proof*⟩

lemma *float-transfer-nat*: *float* (*a*, *b*) = *float* (*a* ∗ *2*^*c*, *b* − *int c*)

⟨*proof*⟩

lemma *real-is-int-real*[*simp*]: *real-is-int* (*real* (*x*::*int*))
⟨*proof*⟩

lemma *int-of-real-real*[*simp*]: *int-of-real* (*real x*) = *x*
⟨*proof*⟩

lemma *real-int-of-real*[*simp*]: *real-is-int x* ⟹ *real* (*int-of-real x*) = *x*
⟨*proof*⟩

lemma *real-is-int-add-int-of-real*: *real-is-int a* ⟹ *real-is-int b* ⟹ (*int-of-real* (*a*+*b*)) = (*int-of-real a*) + (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-add*[*simp*]: *real-is-int a* ⟹ *real-is-int b* ⟹ *real-is-int* (*a*+*b*)
⟨*proof*⟩

lemma *int-of-real-sub*: *real-is-int a* ⟹ *real-is-int b* ⟹ (*int-of-real* (*a*−*b*)) = (*int-of-real a*) − (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-sub*[*simp*]: *real-is-int a* ⟹ *real-is-int b* ⟹ *real-is-int* (*a*−*b*)
⟨*proof*⟩

lemma *real-is-int-rep*: *real-is-int x* ⟹ *?*! (*a*::*int*). *real a* = *x*
⟨*proof*⟩

lemma *int-of-real-mult*:
assumes *real-is-int a real-is-int b*
shows (*int-of-real* (*a*∗*b*)) = (*int-of-real a*) ∗ (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-mult*[*simp*]: *real-is-int a* ⟹ *real-is-int b* ⟹ *real-is-int* (*a*∗*b*)
⟨*proof*⟩

lemma *real-is-int-0*[*simp*]: *real-is-int* (*0*::*real*)
⟨*proof*⟩

lemma *real-is-int-1*[*simp*]: *real-is-int* (*1*::*real*)
⟨*proof*⟩

lemma *real-is-int-n1*: *real-is-int* (−*1*::*real*)
⟨*proof*⟩

lemma *real-is-int-number-of*[*simp*]: *real-is-int* ((*number-of* :: *int* ⇒ *real*) *x*)
⟨*proof*⟩

lemma *int-of-real-0*[*simp*]: *int-of-real* (*0*::*real*) = (*0*::*int*)

⟨*proof*⟩

lemma *int-of-real-1*[*simp*]: *int-of-real* (*1*::*real*) = (*1*::*int*)
⟨*proof*⟩

lemma *int-of-real-number-of*[*simp*]: *int-of-real* (*number-of b*) = *number-of b*
⟨*proof*⟩

lemma *float-transfer-even*: *even a* ⟹ *float* (*a*, *b*) = *float* (*a div 2*, *b*+*1*)
⟨*proof*⟩

lemma *int-div-zdiv*: *int* (*a div b*) = (*int a*) *div* (*int b*)
⟨*proof*⟩

lemma *int-mod-zmod*: *int* (*a mod b*) = (*int a*) *mod* (*int b*)
⟨*proof*⟩

lemma *abs-div-2-less*: *a* ≠ *0* ⟹ *a* ≠ −*1* ⟹ *abs*((*a*::*int*) *div 2*) < *abs a*
⟨*proof*⟩

function *norm-float* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
norm-float a b = (*if a* ≠ *0* ∧ *even a then norm-float* (*a div 2*) (*b* + *1*)
else if a = *0 then* (*0*, *0*) *else* (*a*, *b*))
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *norm-float*: *float x* = *float* (*split norm-float x*)
⟨*proof*⟩

lemma *float-add-l0*: *float* (*0*, *e*) + *x* = *x*
⟨*proof*⟩

lemma *float-add-r0*: *x* + *float* (*0*, *e*) = *x*
⟨*proof*⟩

lemma *float-add*:
float (*a1*, *e1*) + *float* (*a2*, *e2*) =
(*if e1*<=*e2 then float* (*a1*+*a2*∗*2*^(*nat*(*e2*−*e1*)), *e1*)
else float (*a1*∗*2*^(*nat* (*e1*−*e2*))+*a2*, *e2*))
⟨*proof*⟩

lemma *float-add-assoc1*:
(*x* + *float* (*y1*, *e1*)) + *float* (*y2*, *e2*) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-add-assoc2*:
(*float* (*y1*, *e1*) + *x*) + *float* (*y2*, *e2*) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-add-assoc3*:
float (*y1*, *e1*) + (*x* + *float* (*y2*, *e2*)) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-add-assoc4*:
float (*y1*, *e1*) + (*float* (*y2*, *e2*) + *x*) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-mult-l0*: *float* (*0*, *e*) ∗ *x* = *float* (*0*, *0*)
⟨*proof*⟩

lemma *float-mult-r0*: *x* ∗ *float* (*0*, *e*) = *float* (*0*, *0*)
⟨*proof*⟩

definition
lbound :: *real* ⇒ *real*
where
lbound x = *min 0 x*

definition
ubound :: *real* ⇒ *real*
where
ubound x = *max 0 x*

lemma *lbound*: *lbound x* ≤ *x*
⟨*proof*⟩

lemma *ubound*: *x* ≤ *ubound x*
⟨*proof*⟩

lemma *float-mult*:
float (*a1*, *e1*) ∗ *float* (*a2*, *e2*) =
(*float* (*a1* ∗ *a2*, *e1* + *e2*))
⟨*proof*⟩

lemma *float-minus*:
− (*float* (*a*,*b*)) = *float* (−*a*, *b*)
⟨*proof*⟩

lemma *zero-less-pow2*:
0 < *pow2 x*
⟨*proof*⟩

lemma *zero-le-float*:
(*0* <= *float* (*a*,*b*)) = (*0* <= *a*)
⟨*proof*⟩

lemma *float-le-zero*:

(*float* (*a*,*b*) <= *0*) = (*a* <= *0*)
⟨*proof*⟩

lemma *float-abs*:
abs (*float* (*a*,*b*)) = (*if 0* <= *a then* (*float* (*a*,*b*)) *else* (*float* (−*a*,*b*)))
⟨*proof*⟩

lemma *float-zero*:
float (*0*, *b*) = *0*
⟨*proof*⟩

lemma *float-pprt*:
pprt (*float* (*a*, *b*)) = (*if 0* <= *a then* (*float* (*a*,*b*)) *else* (*float* (*0*, *b*)))
⟨*proof*⟩

lemma *pprt-lbound*: *pprt* (*lbound x*) = *float* (*0*, *0*)
⟨*proof*⟩

lemma *nprt-ubound*: *nprt* (*ubound x*) = *float* (*0*, *0*)
⟨*proof*⟩

lemma *float-nprt*:
nprt (*float* (*a*, *b*)) = (*if 0* <= *a then* (*float* (*0*,*b*)) *else* (*float* (*a*, *b*)))
⟨*proof*⟩

lemma *norm-0-1*: (*0*::-::*number-ring*) = *Numeral0* & (*1*::-::*number-ring*) = *Numeral1*
⟨*proof*⟩

lemma *add-left-zero*: *0* + *a* = (*a*::'*a*::*comm-monoid-add*)
⟨*proof*⟩

lemma *add-right-zero*: *a* + *0* = (*a*::'*a*::*comm-monoid-add*)
⟨*proof*⟩

lemma *mult-left-one*: *1* * *a* = (*a*::'*a*::*semiring-1*)
⟨*proof*⟩

lemma *mult-right-one*: *a* * *1* = (*a*::'*a*::*semiring-1*)
⟨*proof*⟩

lemma *int-pow-0*: (*a*::*int*)^(*Numeral0*) = *1*
⟨*proof*⟩

lemma *int-pow-1*: (*a*::*int*)^(*Numeral1*) = *a*
⟨*proof*⟩

lemma *zero-eq-Numeral0-nring*: (*0*::'*a*::*number-ring*) = *Numeral0*
⟨*proof*⟩

lemma *one-eq-Numeral1-nring*: $(1::'a::number\text{-}ring) = Numeral1$
⟨*proof*⟩

lemma *zero-eq-Numeral0-nat*: $(0::nat) = Numeral0$
⟨*proof*⟩

lemma *one-eq-Numeral1-nat*: $(1::nat) = Numeral1$
⟨*proof*⟩

lemma *zpower-Pls*: $(z::int)\hat{}Numeral0 = Numeral1$
⟨*proof*⟩

lemma *zpower-Min*: $(z::int)\hat{}((-1)::nat) = Numeral1$
⟨*proof*⟩

lemma *fst-cong*: $a=a' \Longrightarrow fst\ (a,b) = fst\ (a',b)$
⟨*proof*⟩

lemma *snd-cong*: $b=b' \Longrightarrow snd\ (a,b) = snd\ (a,b')$
⟨*proof*⟩

lemma *lift-bool*: $x \Longrightarrow x=True$
⟨*proof*⟩

lemma *nlift-bool*: $\sim x \Longrightarrow x=False$
⟨*proof*⟩

lemma *not-false-eq-true*: $(\sim False) = True$ ⟨*proof*⟩

lemma *not-true-eq-false*: $(\sim True) = False$ ⟨*proof*⟩

lemmas *binarith* =
normalize-bin-simps
pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps

lemma *int-eq-number-of-eq*:
$(((number\text{-}of\ v)::int)=(number\text{-}of\ w)) = iszero\ ((number\text{-}of\ (v + uminus\ w))::int)$
⟨*proof*⟩

lemma *int-iszero-number-of-Pls*: $iszero\ (Numeral0::int)$
⟨*proof*⟩

lemma *int-nonzero-number-of-Min*: $\sim(iszero\ ((-1)::int))$
⟨*proof*⟩

lemma *int-iszero-number-of-Bit0*: $iszero\ ((number\text{-}of\ (Int.Bit0\ w))::int) = iszero$
$((number\text{-}of\ w)::int)$
⟨*proof*⟩

lemma *int-iszero-number-of-Bit1*: $\neg$ *iszero* ((*number-of* (*Int.Bit1 w*))::*int*)
⟨*proof*⟩

lemma *int-less-number-of-eq-neg*: (((*number-of x*)::*int*) < *number-of y*) = *neg* ((*number-of* (*x* + (*uminus y*)))::*int*)
⟨*proof*⟩

lemma *int-not-neg-number-of-Pls*: $\neg$ (*neg* (*Numeral0*::*int*))
⟨*proof*⟩

lemma *int-neg-number-of-Min*: *neg* (−*1*::*int*)
⟨*proof*⟩

lemma *int-neg-number-of-Bit0*: *neg* ((*number-of* (*Int.Bit0 w*))::*int*) = *neg* ((*number-of w*)::*int*)
⟨*proof*⟩

lemma *int-neg-number-of-Bit1*: *neg* ((*number-of* (*Int.Bit1 w*))::*int*) = *neg* ((*number-of w*)::*int*)
⟨*proof*⟩

lemma *int-le-number-of-eq*: (((*number-of x*)::*int*) ≤ *number-of y*) = ($\neg$ *neg* ((*number-of* (*y* + (*uminus x*)))::*int*))
⟨*proof*⟩

lemmas *intarithrel* =
int-eq-number-of-eq
lift-bool[*OF int-iszero-number-of-Pls*] *nlift-bool*[*OF int-nonzero-number-of-Min*] *int-iszero-number-of-Bit0*
lift-bool[*OF int-iszero-number-of-Bit1*] *int-less-number-of-eq-neg nlift-bool*[*OF int-not-neg-number-of-Pls*] *lift-bool*[*OF int-neg-number-of-Min*]
int-neg-number-of-Bit0 int-neg-number-of-Bit1 int-le-number-of-eq

lemma *int-number-of-add-sym*: ((*number-of v*)::*int*) + *number-of w* = *number-of* (*v* + *w*)
⟨*proof*⟩

lemma *int-number-of-diff-sym*: ((*number-of v*)::*int*) − *number-of w* = *number-of* (*v* + (*uminus w*))
⟨*proof*⟩

lemma *int-number-of-mult-sym*: ((*number-of v*)::*int*) ∗ *number-of w* = *number-of* (*v* ∗ *w*)
⟨*proof*⟩

lemma *int-number-of-minus-sym*: − ((*number-of v*)::*int*) = *number-of* (*uminus v*)
⟨*proof*⟩

lemmas *intarith = int-number-of-add-sym int-number-of-minus-sym int-number-of-diff-sym int-number-of-mult-sym*

lemmas *natarith = add-nat-number-of diff-nat-number-of mult-nat-number-of eq-nat-number-of less-nat-number-of*

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

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 pprt-lbound nprt-ubound

lemmas *arith = binarith intarith intarithrel natarith powerarith floatarith not-false-eq-true not-true-eq-false*

⟨*ML*⟩

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

end
theory *ComputeHOL*
imports *Complex-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*: $(\neg\ True) = False\ \langle proof\rangle$
lemma *bool2*: $(\neg\ False) = True\ \langle proof\rangle$
lemma *bool3*: $(P \wedge True) = P\ \langle proof\rangle$
lemma *bool4*: $(True \wedge P) = P\ \langle proof\rangle$
lemma *bool5*: $(P \wedge False) = False\ \langle proof\rangle$
lemma *bool6*: $(False \wedge P) = False\ \langle proof\rangle$
lemma *bool7*: $(P \vee True) = True\ \langle proof\rangle$
lemma *bool8*: $(True \vee P) = True\ \langle proof\rangle$
lemma *bool9*: $(P \vee False) = P\ \langle proof\rangle$
lemma *bool10*: $(False \vee P) = P\ \langle proof\rangle$
lemma *bool11*: $(True \longrightarrow P) = P\ \langle proof\rangle$
lemma *bool12*: $(P \longrightarrow True) = True\ \langle proof\rangle$
lemma *bool13*: $(True \longrightarrow P) = P\ \langle proof\rangle$
lemma *bool14*: $(P \longrightarrow False) = (\neg\ P)\ \langle proof\rangle$
lemma *bool15*: $(False \longrightarrow P) = True\ \langle proof\rangle$
lemma *bool16*: $(False = False) = True\ \langle proof\rangle$
lemma *bool17*: $(True = True) = True\ \langle proof\rangle$
lemma *bool18*: $(False = True) = False\ \langle proof\rangle$
lemma *bool19*: $(True = False) = False\ \langle proof\rangle$

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*: $fst\ (x,y) = x\ \langle proof\rangle$
lemma *compute-snd*: $snd\ (x,y) = y\ \langle proof\rangle$
lemma *compute-pair-eq*: $((a,\ b) = (c,\ d)) = (a = c \wedge b = d)\ \langle proof\rangle$

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

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

lemma *compute-the*: $the\ (Some\ x) = x\ \langle proof\rangle$
lemma *compute-None-Some-eq*: $(None = Some\ x) = False\ \langle proof\rangle$
lemma *compute-Some-None-eq*: $(Some\ x = None) = False\ \langle proof\rangle$
lemma *compute-None-None-eq*: $(None = None) = True\ \langle proof\rangle$
lemma *compute-Some-Some-eq*: $(Some\ x = Some\ y) = (x = y)\ \langle proof\rangle$

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

lemmas *biteq* = *eq-bin-simps*

lemmas *bitless* = *less-bin-simps*

lemmas *bitle* = *le-bin-simps*

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 *divmod*: *IntDiv.divmod 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 divmod 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 *SparseMatrix LP* ~~/*src*/*HOL*/*Tools*/*ComputeFloat* ~~/*src*/*HOL*/*Tools*/*ComputeNumeral*
uses *Cplex-tools.ML CplexMatrixConverter.ML FloatSparseMatrixBuilder.ML*
fspmlp.ML (*matrixlp.ML*)
begin

lemma *spm-mult-le-dual-prts*:
assumes
sorted-sparse-matrix A1
sorted-sparse-matrix A2
sorted-sparse-matrix c1
sorted-sparse-matrix c2
sorted-sparse-matrix y
sorted-sparse-matrix r1
sorted-sparse-matrix r2
sorted-spvec b
le-spmat ([], *y*)
sparse-row-matrix A1 ≤ *A*
A ≤ *sparse-row-matrix A2*
sparse-row-matrix c1 ≤ *c*
c ≤ *sparse-row-matrix c2*
sparse-row-matrix r1 ≤ *x*
x ≤ *sparse-row-matrix r2*

$A * x \leq$ *sparse-row-matrix* $(b::('a::lordered\text{-}ring)\ spmat)$
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 $\leq A$
$A \leq$ *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* $(b::('a::lordered\text{-}ring)\ spmat)$
shows
$c * x \leq$ *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*⟩

⟨*ML*⟩

end