Matrix

Steven Obua

November 22, 2007

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*) = $y \Longrightarrow \forall\ a\ b.\ x\ a\ b = y\ a\ b$ ⟨*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* $\Longrightarrow$ *Rep-matrix* *A m n* = *0*
⟨*proof*⟩

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

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

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

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

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

lemma *le-nrows*: (*n* <= *nrows A*) = (∀ *m*. (∀ *j i*. *m* <= *j* ⟶ (*Rep-matrix A j i*) = *0*) ⟶ *n* <= *m*) (**is** *?concl*)
⟨*proof*⟩

lemma *nrows-notzero*: *Rep-matrix A m n* ≠ *0* ⟹ *m* < *nrows A*
⟨*proof*⟩

lemma *ncols-notzero*: *Rep-matrix A m n* ≠ *0* ⟹ *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* ⟶ *x j i* = *0* (**is** *?em*) **and** *aen*:*? n*. ! *j i*. (*n* <= *i* ⟶ *x j i* = *0*) (**is** *?en*)
shows (*Rep-matrix* (*Abs-matrix x*)) = *x*
⟨*proof*⟩

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

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

lemma *expand-combine-infmatrix*[*simp*]: *combine-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-infmatrix f*)
⟨*proof*⟩

lemma *combine-matrix-commute*:
commutative f $\Longrightarrow$ *commutative* (*combine-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-positions* (*combine-infmatrix f A B*) $\subseteq$ (*nonzero-positions A*) $\cup$ (*nonzero-positions B*)
⟨*proof*⟩

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

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

lemma *combine-ncols*: *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* ⇒ *'b::zero* ⇒ *'a*) ⇒ *bool*
zero-r-neutral f == ! *a*. *f a 0 = a*

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

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

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

lemma *foldseq-assoc* : *associative f* ⟹ *foldseq f* = *foldseq-transposed f*
⟨*proof*⟩

lemma *foldseq-distr*: ⟦*associative f*; *commutative f*⟧ ⟹ *foldseq f* (% *k*. *f* (*u k*) (*v k*)) *n* = *f* (*foldseq f u n*) (*foldseq f v n*)
⟨*proof*⟩

theorem ⟦*associative f*; *associative g*; ∀ *a b c d*. *g* (*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*); *? x y*. (*f x*) ≠ (*f y*); *? x y*. (*g x*) ≠ (*g y*); *f x x* = *x*; *g x x* = *x*⟧ ⟹ *f*=*g* | (! *y*. *f y x* = *y*) | (! *y*. *g y x* = *y*)
⟨*proof*⟩

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

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

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

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

$\langle proof \rangle$

lemma *foldseq-zerotail2*:
assumes $!\ x.\ f\ x\ 0 = x$
and $!\ i.\ n < i \longrightarrow s\ i = 0$
and *nm*: $n <= m$
shows
foldseq f s n = *foldseq f s m* (**is** *?concl*)
$\langle proof \rangle$

lemma *foldseq-zerostart*:
$!\ x.\ f\ 0\ (f\ 0\ x) = f\ 0\ x \Longrightarrow\ !\ i.\ i <= n \longrightarrow s\ i = 0 \Longrightarrow$ *foldseq f s* $(Suc\ n) = f\ 0\ (s\ (Suc\ n))$
$\langle proof \rangle$

lemma *foldseq-zerostart2*:
$!\ x.\ f\ 0\ x = x \Longrightarrow\ !\ i.\ i < n \longrightarrow s\ i = 0 \Longrightarrow$ *foldseq f s n* $= s\ n$
$\langle proof \rangle$

lemma *foldseq-almostzero*:
assumes *f0x*:$!\ x.\ f\ 0\ x = x$ **and** *fx0*: $!\ x.\ f\ x\ 0 = x$ **and** *s0*:$!\ i.\ i \neq j \longrightarrow s\ i = 0$
shows *foldseq f s n* = (*if* ($j <= n$) *then* ($s\ j$) *else* 0) (**is** *?concl*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

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

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

$\langle proof \rangle$

lemma *combine-matrix-zero-l-neutral*: *zero-l-neutral f* $\Longrightarrow$ *zero-l-neutral* (*combine-matrix f*)
$\langle proof \rangle$

lemma *combine-matrix-zero-r-neutral*: *zero-r-neutral f* $\Longrightarrow$ *zero-r-neutral* (*combine-matrix f*)
$\langle proof \rangle$

lemma *mult-matrix-zero-closed*: $[\![$*fadd 0 0* $=$ *0*; *zero-closed fmul*$]\!]$ $\Longrightarrow$ *zero-closed* (*mult-matrix fmul fadd*)
$\langle proof \rangle$

lemma *mult-matrix-n-zero-right*[*simp*]: $[\![$*fadd 0 0* $=$ *0*; !*a*. *fmul a 0* $=$ *0*$]\!]$ $\Longrightarrow$ *mult-matrix-n n fmul fadd A 0* $=$ *0*
$\langle proof \rangle$

lemma *mult-matrix-n-zero-left*[*simp*]: $[\![$*fadd 0 0* $=$ *0*; !*a*. *fmul 0 a* $=$ *0*$]\!]$ $\Longrightarrow$ *mult-matrix-n n fmul fadd 0 A* $=$ *0*
$\langle proof \rangle$

lemma *mult-matrix-zero-left*[*simp*]: $[\![$*fadd 0 0* $=$ *0*; !*a*. *fmul 0 a* $=$ *0*$]\!]$ $\Longrightarrow$ *mult-matrix fmul fadd 0 A* $=$ *0*
$\langle proof \rangle$

lemma *mult-matrix-zero-right*[*simp*]: $[\![$*fadd 0 0* $=$ *0*; !*a*. *fmul a 0* $=$ *0*$]\!]$ $\Longrightarrow$ *mult-matrix fmul fadd A 0* $=$ *0*
$\langle proof \rangle$

lemma *apply-matrix-zero*[*simp*]: *f 0* $=$ *0* $\Longrightarrow$ *apply-matrix f 0* $=$ *0*
$\langle proof \rangle$

lemma *combine-matrix-zero*: *f 0 0* $=$ *0* $\Longrightarrow$ *combine-matrix f 0 0* $=$ *0*
$\langle proof \rangle$

lemma *transpose-matrix-zero*[*simp*]: *transpose-matrix 0* $=$ *0*
$\langle proof \rangle$

lemma *apply-zero-matrix-def*[*simp*]: *apply-matrix* (% *x*. *0*) *A* $=$ *0*
$\langle proof \rangle$

constdefs
singleton-matrix :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ ('*a*::*zero*) $\Rightarrow$ '*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* $\Rightarrow$ *int* $\Rightarrow$ *int* $\Rightarrow$ '*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* $\Rightarrow$ *nat* $\Rightarrow$ '*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:

$?\ e.\ (!\ a\ b.\ a \neq b \longrightarrow fmul\ a\ e \neq fmul\ b\ e)$
and *fprems*:
$!\ a.\ fmul\ 0\ a = 0$
$!\ a.\ fmul\ a\ 0 = 0$
$!\ a.\ fadd\ a\ 0 = a$
$!\ a.\ fadd\ 0\ a = a$
and *contraprems*:
$mult\text{-}matrix\ fmul\ fadd\ A = mult\text{-}matrix\ fmul\ fadd\ B$
shows
$A = B$
$\langle proof \rangle$

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

lemma *foldmatrix-transpose*:
assumes
$!\ a\ b\ c\ d.\ g(f\ a\ b)\ (f\ c\ d) = f\ (g\ a\ c)\ (g\ b\ d)$
shows
$foldmatrix\ f\ g\ A\ m\ n = foldmatrix\text{-}transposed\ g\ f\ (transpose\text{-}infmatrix\ A)\ n\ m$ (**is** *?concl*)
$\langle proof \rangle$

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
$foldseq\ g\ (\%\ j.\ foldseq\ f\ (A\ j)\ n)\ m = foldseq\ f\ (\%\ j.\ foldseq\ g\ ((transpose\text{-}infmatrix\ A)\ j)\ m)\ n$
$\langle proof \rangle$

lemma *mult-n-nrows*:
assumes
$!\ a.\ fmul\ 0\ a = 0$
$!\ a.\ fmul\ a\ 0 = 0$
$fadd\ 0\ 0 = 0$
shows $nrows\ (mult\text{-}matrix\text{-}n\ n\ fmul\ fadd\ A\ B) \leq nrows\ A$
$\langle proof \rangle$

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
$transpose\text{-}matrix \ (mult\text{-}matrix \ fmul \ fadd \ A \ B) = mult\text{-}matrix \ fmul \ fadd \ (transpose\text{-}matrix \ B) \ (transpose\text{-}matrix \ A)$
$\langle proof \rangle$

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

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

instance *matrix* :: ({*ord*, *zero*}) *ord*
le-matrix-def: $A \leq B \equiv \forall j \ i. \ Rep\text{-}matrix \ A \ j \ i \leq Rep\text{-}matrix \ B \ j \ i$
less-def: $A < B \equiv A \leq B \wedge A \neq B$ $\langle proof \rangle$

instance *matrix* :: ({*order*, *zero*}) *order*
$\langle proof \rangle$

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

lemma *le-combine-matrix*:
assumes
$f \ 0 \ 0 = 0$
$! \ a \ b \ c \ d. \ a <= b \ \& \ c <= d \longrightarrow f \ a \ c <= f \ b \ d$
$A <= B$
$C <= D$
shows
$combine\text{-}matrix \ f \ A \ C <= combine\text{-}matrix \ f \ B \ D$
$\langle proof \rangle$

lemma *le-left-combine-matrix*:
assumes
$f \ 0 \ 0 = 0$

$!\ a\ b\ c.\ a <= b \longrightarrow f\ c\ a <= f\ c\ b$
$A <= B$
shows
combine-matrix f C A $<=$ *combine-matrix f C B*
⟨*proof*⟩

lemma *le-right-combine-matrix*:
assumes
$f\ 0\ 0 = 0$
$!\ a\ b\ c.\ a <= b \longrightarrow f\ a\ c <= f\ b\ c$
$A <= B$
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
$!\ a\ b\ c\ d\ .\ a <= b\ \&\ c <= d \longrightarrow f\ a\ c <= f\ b\ d$
$!\ i.\ i <= n \longrightarrow s\ i <= t\ i$
shows
foldseq f s n $<=$ *foldseq f t n*
⟨*proof*⟩

lemma *le-left-mult*:
assumes
$!\ a\ b\ c\ d.\ a <= b\ \&\ c <= d \longrightarrow$ *fadd a c* $<=$ *fadd b d*
$!\ c\ a\ b.\quad 0 <= c\ \&\ a <= b \longrightarrow$ *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
$!\ a\ b\ c\ d.\ a <= b\ \&\ c <= d \longrightarrow$ *fadd a c* $<=$ *fadd b d*
$!\ c\ a\ b.\ 0 <= c\ \&\ a <= b \longrightarrow$ *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* $\Longrightarrow$ *P j i* ⟨*proof*⟩
lemma *neg-imp*: ($\neg$ *Q* $\longrightarrow$ $\neg$ *P*) $\Longrightarrow$ *P* $\longrightarrow$ *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* $\Longrightarrow$ *0* <= *i* $\Longrightarrow$ (*move-matrix A j i* <= *0*) = (*A* <= (*0*::('*a*::{*order*,*zero*}) *matrix*))
⟨*proof*⟩

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

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

end

theory *Matrix*
imports *MatrixGeneral*
begin

instance *matrix* :: ({*zero*, *lattice*}) *lattice*
inf $\equiv$ *combine-matrix inf*
sup $\equiv$ *combine-matrix sup*
⟨*proof*⟩

instance *matrix* :: ({*plus*, *zero*}) *plus*
plus-matrix-def: *A* + *B* $\equiv$ *combine-matrix* (*op* +) *A B* ⟨*proof*⟩

instance *matrix* :: ({*minus*, *zero*}) *minus*
minus-matrix-def: $-$ *A* $\equiv$ *apply-matrix uminus A*
diff-matrix-def: *A* $-$ *B* $\equiv$ *combine-matrix* (*op* $-$) *A B* ⟨*proof*⟩

instance *matrix* :: ({*plus*, *times*, *zero*}) *times*
times-matrix-def: $A * B \equiv$ *mult-matrix* (*op* $*$) (*op* $+$) A B ⟨*proof*⟩

instance *matrix* :: (*lordered-ab-group-add*) *abs*
abs-matrix-def: *abs* $A \equiv$ *sup* A ($-$ A) ⟨*proof*⟩

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

lemma *Rep-one-matrix*[*simp*]: *Rep-matrix* (*one-matrix n*) *j i* = (*if* (*j* = *i* & *j* < *n*) *then 1 else 0*)
⟨*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*′) ≤ δ*A*
  b ≤ *b*′
  abs (*c* − *c*′) ≤ δ*c*
  abs x ≤ *r*
  shows
  c ∗ *x* ≤ *y* ∗ *b*′ + (*y* ∗ δ*A* + *abs* (*y* ∗ *A*′ − *c*′) + δ*c*) ∗ *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
$A * x \leq (b::'a::lordered\text{-}ring)$
$0 \leq y$
$A1 \leq A$
$A \leq A2$
$c1 \leq c$
$c \leq c2$
$r1 \leq x$
$x \leq r2$
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

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

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

consts
$sparse\text{-}row\text{-}vector :: ('a::lordered\text{-}ring)\ spvec \Rightarrow 'a\ matrix$
$sparse\text{-}row\text{-}matrix :: ('a::lordered\text{-}ring)\ spmat \Rightarrow 'a\ matrix$

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

lemma *sparse-row-vector-empty*[*simp*]: $sparse\text{-}row\text{-}vector\ [] = 0$
⟨*proof*⟩

lemma *sparse-row-matrix-empty*[*simp*]: $sparse\text{-}row\text{-}matrix\ [] = 0$
⟨*proof*⟩

lemma *foldl-distrstart*[*rule-format*]: $!\ a\ x\ y.\ (f\ (g\ x\ y)\ a = g\ x\ (f\ y\ a)) \Longrightarrow !\ x\ y.\ (foldl\ f\ (g\ x\ y)\ l = g\ x\ (foldl\ f\ y\ l))$
⟨*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*]: $m <= n \longrightarrow$ *sorted-spvec* $((n,a)\#arr)$ $\longrightarrow$ *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-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*)

$\longrightarrow$ *sorted-spvec* $((aa,\ b + y * ba)\ \#\ (addmult\text{-}spvec\ (y,\ arr,\ brr)))$
⟨*proof*⟩

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

consts
mult-spvec-spmat :: (*'a*::*lordered-ring*) *spvec* $*$ *'a spvec* $*$ *'a spmat* $\Rightarrow$ *'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*) $\longrightarrow$ *sorted-spvec B* $\longrightarrow$
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*) $\longrightarrow$ *sorted-spmat B* $\longrightarrow$ *sorted-spvec* (*mult-spvec-spmat* (*c*, *a*, *B*))
⟨*proof*⟩

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

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

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

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

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

consts
  add-spvec :: (*'a::lordered-ab-group-add*) *spvec* $*$ *'a spvec* $\Rightarrow$ *'a spvec*
  add-spmat :: (*'a::lordered-ab-group-add*) *spmat* $*$ *'a spmat* $\Rightarrow$ *'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* $\longrightarrow$ (*ab* = *a* | (*brr* $\neq$ [] & *ab* = *fst* (*hd brr*)))
  ⟨*proof*⟩

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

lemma *sorted-add-spvec-helper*[*rule-format*]: *add-spvec* (*arr*, *brr*) = (*ab*, *bb*) # *list* $\longrightarrow$ ((*arr* $\neq$ [] & *ab* = *fst* (*hd arr*)) | (*brr* $\neq$ [] & *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*

$le\text{-}spmat\ (a\#as,\ []) = (le\text{-}spvec\ (snd\ a,\ [])\ \&\ (le\text{-}spmat\ (as,\ [])))$
$le\text{-}spmat\ ([],\ b\#bs) = (le\text{-}spvec\ ([],\ snd\ b)\ \&\ (le\text{-}spmat\ ([],\ bs)))$
$le\text{-}spmat\ (a\#as,\ b\#bs) = ($
$if\ fst\ a < fst\ b\ then$
$(le\text{-}spvec(snd\ a,[])\ \&\ le\text{-}spmat(as,\ b\#bs))$
$else\ (if\ (fst\ b < fst\ a)\ then$
$(le\text{-}spvec([],\ snd\ b)\ \&\ le\text{-}spmat(a\#as,\ bs))$
$else$
$(le\text{-}spvec(snd\ a,\ snd\ b)\ \&\ le\text{-}spmat\ (as,\ bs))))$

constdefs
$disj\text{-}matrices :: ('a::zero)\ matrix \Rightarrow 'a\ matrix \Rightarrow bool$
$disj\text{-}matrices\ A\ B == (!\ j\ i.\ (Rep\text{-}matrix\ A\ j\ i \neq 0) \longrightarrow (Rep\text{-}matrix\ B\ j\ i = 0))\ \&\ (!\ j\ i.\ (Rep\text{-}matrix\ B\ j\ i \neq 0) \longrightarrow (Rep\text{-}matrix\ A\ j\ i = 0))$

declare $[[simp\text{-}depth\text{-}limit = 6]]$

lemma $disj\text{-}matrices\text{-}contr1$: $disj\text{-}matrices\ A\ B \Longrightarrow Rep\text{-}matrix\ A\ j\ i \neq 0 \Longrightarrow Rep\text{-}matrix\ B\ j\ i = 0$
⟨proof⟩

lemma $disj\text{-}matrices\text{-}contr2$: $disj\text{-}matrices\ A\ B \Longrightarrow Rep\text{-}matrix\ B\ j\ i \neq 0 \Longrightarrow Rep\text{-}matrix\ A\ j\ i = 0$
⟨proof⟩

lemma $disj\text{-}matrices\text{-}add$: $disj\text{-}matrices\ A\ B \Longrightarrow disj\text{-}matrices\ C\ D \Longrightarrow disj\text{-}matrices\ A\ D \Longrightarrow disj\text{-}matrices\ B\ C \Longrightarrow$
$(A + B <= C + D) = (A <= C\ \&\ B <= (D::('a::lordered\text{-}ab\text{-}group\text{-}add)\ matrix))$
⟨proof⟩

lemma $disj\text{-}matrices\text{-}zero1[simp]$: $disj\text{-}matrices\ 0\ B$
⟨proof⟩

lemma $disj\text{-}matrices\text{-}zero2[simp]$: $disj\text{-}matrices\ A\ 0$
⟨proof⟩

lemma $disj\text{-}matrices\text{-}commute$: $disj\text{-}matrices\ A\ B = disj\text{-}matrices\ B\ A$
⟨proof⟩

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

lemma $disj\text{-}matrices\text{-}add\text{-}zero\text{-}le$: $disj\text{-}matrices\ A\ B \Longrightarrow$
$(0 <= A + B) = (0 <= A\ \&\ 0 <= (B::('a::lordered\text{-}ab\text{-}group\text{-}add)\ matrix))$
⟨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*) $\longrightarrow$ (*sorted-spmat A*) $\longrightarrow$ (*sorted-spvec B*) $\longrightarrow$ (*sorted-spmat B*) $\longrightarrow$
le-spmat(A, B) = (*sparse-row-matrix A* <= *sparse-row-matrix B*)

$\langle proof \rangle$

declare $[[simp\text{-}depth\text{-}limit = 999]]$

consts
$abs\text{-}spmat :: ('a::lordered\text{-}ring)\ spmat \Rightarrow 'a\ spmat$
$minus\text{-}spmat :: ('a::lordered\text{-}ring)\ spmat \Rightarrow 'a\ spmat$

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

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

lemma *sparse-row-matrix-minus*:
$sparse\text{-}row\text{-}matrix\ (minus\text{-}spmat\ A) = -\ (sparse\text{-}row\text{-}matrix\ A)$
$\langle proof \rangle$

lemma *Rep-sparse-row-vector-zero*: $x \neq 0 \Longrightarrow Rep\text{-}matrix\ (sparse\text{-}row\text{-}vector\ v)\ x\ y = 0$
$\langle proof \rangle$

lemma *sparse-row-matrix-abs*:
$sorted\text{-}spvec\ A \Longrightarrow sorted\text{-}spmat\ A \Longrightarrow sparse\text{-}row\text{-}matrix\ (abs\text{-}spmat\ A) = abs\ (sparse\text{-}row\text{-}matrix\ A)$
$\langle proof \rangle$

lemma *sorted-spvec-minus-spmat*: $sorted\text{-}spvec\ A \Longrightarrow sorted\text{-}spvec\ (minus\text{-}spmat\ A)$
$\langle proof \rangle$

lemma *sorted-spvec-abs-spmat*: $sorted\text{-}spvec\ A \Longrightarrow sorted\text{-}spvec\ (abs\text{-}spmat\ A)$
$\langle proof \rangle$

lemma *sorted-spmat-minus-spmat*: $sorted\text{-}spmat\ A \Longrightarrow sorted\text{-}spmat\ (minus\text{-}spmat\ A)$
$\langle proof \rangle$

lemma *sorted-spmat-abs-spmat*: $sorted\text{-}spmat\ A \Longrightarrow sorted\text{-}spmat\ (abs\text{-}spmat\ A)$
$\langle proof \rangle$

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

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

$\langle proof \rangle$

lemma *sorted-spvec-diff-spmat*: *sorted-spvec A* $\Longrightarrow$ *sorted-spvec B* $\Longrightarrow$ *sorted-spvec* (*diff-spmat A B*)
$\langle proof \rangle$

lemma *sparse-row-diff-spmat*: *sparse-row-matrix* (*diff-spmat A B*) = (*sparse-row-matrix A*) − (*sparse-row-matrix B*)
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *sorted-sparse-matrix-imp-spmat*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spmat A*
$\langle proof \rangle$

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

lemma *bool1*: ($\neg$ *True*) = *False* $\langle proof \rangle$
lemma *bool2*: ($\neg$ *False*) = *True* $\langle proof \rangle$
lemma *bool3*: ((*P*::*bool*) $\wedge$ *True*) = *P* $\langle proof \rangle$
lemma *bool4*: (*True* $\wedge$ (*P*::*bool*)) = *P* $\langle proof \rangle$
lemma *bool5*: ((*P*::*bool*) $\wedge$ *False*) = *False* $\langle proof \rangle$
lemma *bool6*: (*False* $\wedge$ (*P*::*bool*)) = *False* $\langle proof \rangle$
lemma *bool7*: ((*P*::*bool*) $\vee$ *True*) = *True* $\langle proof \rangle$
lemma *bool8*: (*True* $\vee$ (*P*::*bool*)) = *True* $\langle proof \rangle$
lemma *bool9*: ((*P*::*bool*) $\vee$ *False*) = *P* $\langle proof \rangle$
lemma *bool10*: (*False* $\vee$ (*P*::*bool*)) = *P* $\langle proof \rangle$
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*)
$\langle proof \rangle$

consts
pprt-spvec :: (*'a*::{*lordered-ab-group-add*}) *spvec* $\Rightarrow$ *'a spvec*
nprt-spvec :: (*'a*::{*lordered-ab-group-add*}) *spvec* $\Rightarrow$ *'a spvec*
pprt-spmat :: (*'a*::{*lordered-ab-group-add*}) *spmat* $\Rightarrow$ *'a spmat*
nprt-spmat :: (*'a*::{*lordered-ab-group-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* ⟹ *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*) = *move-matrix* (*pprt A*) *j i*
⟨*proof*⟩

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

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

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

lemma *sparse-row-matrix-nprt*: *sorted-spvec m* $\Longrightarrow$ *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

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

end

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

end

theory *Compute-Oracle* **imports** *CPure*
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* $\Longrightarrow$ *x* == *y* ⟨*proof*⟩
lemma *meta-eq-imp-eq*: *x* == *y* $\Longrightarrow$ *x* = *y* ⟨*proof*⟩
lemma *eq-trivial*: *x* = *y* $\Longrightarrow$ *x* = *y* ⟨*proof*⟩

lemma *bool-to-true*: $x :: bool \Longrightarrow x == True$ ⟨*proof*⟩
lemma *transmeta-1*: $x = y \Longrightarrow y == z \Longrightarrow x = z$ ⟨*proof*⟩
lemma *transmeta-2*: $x == y \Longrightarrow y = z \Longrightarrow x = z$ ⟨*proof*⟩
lemma *transmeta-3*: $x == y \Longrightarrow y == z \Longrightarrow x = z$ ⟨*proof*⟩

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

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

lemma *bool1*: $(\neg\ True) = False$ ⟨*proof*⟩
lemma *bool2*: $(\neg\ False) = True$ ⟨*proof*⟩
lemma *bool3*: $(P \wedge True) = P$ ⟨*proof*⟩
lemma *bool4*: $(True \wedge P) = P$ ⟨*proof*⟩
lemma *bool5*: $(P \wedge False) = False$ ⟨*proof*⟩
lemma *bool6*: $(False \wedge P) = False$ ⟨*proof*⟩
lemma *bool7*: $(P \vee True) = True$ ⟨*proof*⟩
lemma *bool8*: $(True \vee P) = True$ ⟨*proof*⟩
lemma *bool9*: $(P \vee False) = P$ ⟨*proof*⟩
lemma *bool10*: $(False \vee P) = P$ ⟨*proof*⟩
lemma *bool11*: $(True \longrightarrow P) = P$ ⟨*proof*⟩
lemma *bool12*: $(P \longrightarrow True) = True$ ⟨*proof*⟩
lemma *bool13*: $(True \longrightarrow P) = P$ ⟨*proof*⟩
lemma *bool14*: $(P \longrightarrow False) = (\neg\ P)$ ⟨*proof*⟩
lemma *bool15*: $(False \longrightarrow P) = True$ ⟨*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*: $((a,\ b) = (c,\ d)) = (a = c \wedge b = d)$ ⟨*proof*⟩

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

lemmas *bitnorm* = *Pls-0-eq Min-1-eq*

lemma *neg1*: *neg Numeral.Pls* = *False* ⟨*proof*⟩
lemma *neg2*: *neg Numeral.Min* = *True* ⟨*proof*⟩
lemma *neg3*: *neg* (*x BIT Numeral.B0*) = *neg x* ⟨*proof*⟩
lemma *neg4*: *neg* (*x BIT Numeral.B1*) = *neg x* ⟨*proof*⟩
lemmas *bitneg* = *neg1 neg2 neg3 neg4*

lemma *iszero1*: *iszero Numeral.Pls = True* ⟨*proof*⟩
lemma *iszero2*: *iszero Numeral.Min = False* ⟨*proof*⟩
lemma *iszero3*: *iszero* (*x BIT Numeral.B0*) = *iszero x* ⟨*proof*⟩
lemma *iszero4*: *iszero* (*x BIT Numeral.B1*) = *False* ⟨*proof*⟩
lemmas *bitiszero = iszero1 iszero2 iszero3 iszero4*

constdefs
lezero x == ($x \leq 0$)
lemma *lezero1*: *lezero Numeral.Pls = True* ⟨*proof*⟩
lemma *lezero2*: *lezero Numeral.Min = True* ⟨*proof*⟩
lemma *lezero3*: *lezero* (*x BIT Numeral.B0*) = *lezero x* ⟨*proof*⟩
lemma *lezero4*: *lezero* (*x BIT Numeral.B1*) = *neg x* ⟨*proof*⟩
lemmas *bitlezero = lezero1 lezero2 lezero3 lezero4*

lemma *biteq1*: (*Numeral.Pls = Numeral.Pls*) = *True* ⟨*proof*⟩
lemma *biteq2*: (*Numeral.Min = Numeral.Min*) = *True* ⟨*proof*⟩
lemma *biteq3*: (*Numeral.Pls = Numeral.Min*) = *False* ⟨*proof*⟩
lemma *biteq4*: (*Numeral.Min = Numeral.Pls*) = *False* ⟨*proof*⟩
lemma *biteq5*: (*x BIT Numeral.B0 = y BIT Numeral.B0*) = (*x = y*) ⟨*proof*⟩
lemma *biteq6*: (*x BIT Numeral.B1 = y BIT Numeral.B1*) = (*x = y*) ⟨*proof*⟩
lemma *biteq7*: (*x BIT Numeral.B0 = y BIT Numeral.B1*) = *False* ⟨*proof*⟩
lemma *biteq8*: (*x BIT Numeral.B1 = y BIT Numeral.B0*) = *False* ⟨*proof*⟩
lemma *biteq9*: (*Numeral.Pls = x BIT Numeral.B0*) = (*Numeral.Pls = x*) ⟨*proof*⟩
lemma *biteq10*: (*Numeral.Pls = x BIT Numeral.B1*) = *False* ⟨*proof*⟩
lemma *biteq11*: (*Numeral.Min = x BIT Numeral.B0*) = *False* ⟨*proof*⟩
lemma *biteq12*: (*Numeral.Min = x BIT Numeral.B1*) = (*Numeral.Min = x*) ⟨*proof*⟩
lemma *biteq13*: (*x BIT Numeral.B0 = Numeral.Pls*) = (*x = Numeral.Pls*) ⟨*proof*⟩
lemma *biteq14*: (*x BIT Numeral.B1 = Numeral.Pls*) = *False* ⟨*proof*⟩
lemma *biteq15*: (*x BIT Numeral.B0 = Numeral.Min*) = *False* ⟨*proof*⟩
lemma *biteq16*: (*x BIT Numeral.B1 = Numeral.Min*) = (*x = Numeral.Min*) ⟨*proof*⟩
lemmas *biteq = biteq1 biteq2 biteq3 biteq4 biteq5 biteq6 biteq7 biteq8 biteq9 biteq10 biteq11 biteq12 biteq13 biteq14 biteq15 biteq16*

lemma *bitless1*: (*Numeral.Pls* < *Numeral.Min*) = *False* ⟨*proof*⟩
lemma *bitless2*: (*Numeral.Pls* < *Numeral.Pls*) = *False* ⟨*proof*⟩
lemma *bitless3*: (*Numeral.Min* < *Numeral.Pls*) = *True* ⟨*proof*⟩
lemma *bitless4*: (*Numeral.Min* < *Numeral.Min*) = *False* ⟨*proof*⟩
lemma *bitless5*: (*x BIT Numeral.B0* < *y BIT Numeral.B0*) = (*x* < *y*) ⟨*proof*⟩
lemma *bitless6*: (*x BIT Numeral.B1* < *y BIT Numeral.B1*) = (*x* < *y*) ⟨*proof*⟩
lemma *bitless7*: (*x BIT Numeral.B0* < *y BIT Numeral.B1*) = ($x \leq y$) ⟨*proof*⟩
lemma *bitless8*: (*x BIT Numeral.B1* < *y BIT Numeral.B0*) = (*x* < *y*) ⟨*proof*⟩
lemma *bitless9*: (*Numeral.Pls* < *x BIT Numeral.B0*) = (*Numeral.Pls* < *x*) ⟨*proof*⟩
lemma *bitless10*: (*Numeral.Pls* < *x BIT Numeral.B1*) = (*Numeral.Pls* ≤ *x*) ⟨*proof*⟩

lemma *bitless11*: $(Numeral.Min < x\ BIT\ Numeral.B0) = (Numeral.Pls \le x)$ ⟨*proof*⟩
lemma *bitless12*: $(Numeral.Min < x\ BIT\ Numeral.B1) = (Numeral.Min < x)$ ⟨*proof*⟩
lemma *bitless13*: $(x\ BIT\ Numeral.B0 < Numeral.Pls) = (x < Numeral.Pls)$ ⟨*proof*⟩
lemma *bitless14*: $(x\ BIT\ Numeral.B1 < Numeral.Pls) = (x < Numeral.Pls)$ ⟨*proof*⟩
lemma *bitless15*: $(x\ BIT\ Numeral.B0 < Numeral.Min) = (x < Numeral.Pls)$ ⟨*proof*⟩
lemma *bitless16*: $(x\ BIT\ Numeral.B1 < Numeral.Min) = (x < Numeral.Min)$ ⟨*proof*⟩
lemmas *bitless* = *bitless1 bitless2 bitless3 bitless4 bitless5 bitless6 bitless7 bitless8 bitless9 bitless10 bitless11 bitless12 bitless13 bitless14 bitless15 bitless16*

lemma *bitle1*: $(Numeral.Pls \le Numeral.Min) = False$ ⟨*proof*⟩
lemma *bitle2*: $(Numeral.Pls \le Numeral.Pls) = True$ ⟨*proof*⟩
lemma *bitle3*: $(Numeral.Min \le Numeral.Pls) = True$ ⟨*proof*⟩
lemma *bitle4*: $(Numeral.Min \le Numeral.Min) = True$ ⟨*proof*⟩
lemma *bitle5*: $(x\ BIT\ Numeral.B0 \le y\ BIT\ Numeral.B0) = (x \le y)$ ⟨*proof*⟩
lemma *bitle6*: $(x\ BIT\ Numeral.B1 \le y\ BIT\ Numeral.B1) = (x \le y)$ ⟨*proof*⟩
lemma *bitle7*: $(x\ BIT\ Numeral.B0 \le y\ BIT\ Numeral.B1) = (x \le y)$ ⟨*proof*⟩
lemma *bitle8*: $(x\ BIT\ Numeral.B1 \le y\ BIT\ Numeral.B0) = (x < y)$ ⟨*proof*⟩
lemma *bitle9*: $(Numeral.Pls \le x\ BIT\ Numeral.B0) = (Numeral.Pls \le x)$ ⟨*proof*⟩
lemma *bitle10*: $(Numeral.Pls \le x\ BIT\ Numeral.B1) = (Numeral.Pls \le x)$ ⟨*proof*⟩
lemma *bitle11*: $(Numeral.Min \le x\ BIT\ Numeral.B0) = (Numeral.Pls \le x)$ ⟨*proof*⟩
lemma *bitle12*: $(Numeral.Min \le x\ BIT\ Numeral.B1) = (Numeral.Min \le x)$ ⟨*proof*⟩
lemma *bitle13*: $(x\ BIT\ Numeral.B0 \le Numeral.Pls) = (x \le Numeral.Pls)$ ⟨*proof*⟩
lemma *bitle14*: $(x\ BIT\ Numeral.B1 \le Numeral.Pls) = (x < Numeral.Pls)$ ⟨*proof*⟩
lemma *bitle15*: $(x\ BIT\ Numeral.B0 \le Numeral.Min) = (x < Numeral.Pls)$ ⟨*proof*⟩
lemma *bitle16*: $(x\ BIT\ Numeral.B1 \le Numeral.Min) = (x \le Numeral.Min)$ ⟨*proof*⟩
lemmas *bitle* = *bitle1 bitle2 bitle3 bitle4 bitle5 bitle6 bitle7 bitle8 bitle9 bitle10 bitle11 bitle12 bitle13 bitle14 bitle15 bitle16*

lemmas *bitsucc* = *succ-Pls succ-Min succ-1 succ-0*

lemmas *bitpred* = *pred-Pls pred-Min pred-1 pred-0*

lemmas *bituminus* = *minus-Pls minus-Min minus-1 minus-0*

lemmas *bitadd* = *add-Pls add-Pls-right add-Min add-Min-right add-BIT-11 add-BIT-10 add-BIT-0*[**where** *b*=*Numeral.B0*] *add-BIT-0*[**where** *b*=*Numeral.B1*]

lemma *mult-Pls-right*: $x * Numeral.Pls = Numeral.Pls$ ⟨*proof*⟩
lemma *mult-Min-right*: $x * Numeral.Min = - x$ ⟨*proof*⟩
lemma *multb0x*: $(x\ BIT\ Numeral.B0) * y = (x * y)\ BIT\ Numeral.B0$ ⟨*proof*⟩
lemma *multxb0*: $x * (y\ BIT\ Numeral.B0) = (x * y)\ BIT\ Numeral.B0$ ⟨*proof*⟩
lemma *multb1*: $(x\ BIT\ Numeral.B1) * (y\ BIT\ Numeral.B1) = (((x * y)\ BIT\ Numeral.B0) + x + y)\ BIT\ Numeral.B1$
⟨*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* (*Numeral.Pls*) ⟨*proof*⟩
lemma *natnorm1*: $(1 :: nat)$ = *number-of* (*Numeral.Pls BIT Numeral.B1*) ⟨*proof*⟩

lemmas *natnorm* = *natnorm0 natnorm1 nat-norm-number-of*

lemma *natsuc*: *Suc* (*number-of* x) = (*if neg* x *then* 1 *else number-of* (*Numeral.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*: $(((number\text{-}of\ x)::nat) < (number\text{-}of\ y)) = ((x < y) \wedge (\neg\ (lezero\ y)))$
  ⟨*proof*⟩

lemma *natle*: $(((number\text{-}of\ x)::nat) \leq (number\text{-}of\ y)) = (y < x \longrightarrow lezero\ x)$
  ⟨*proof*⟩

fun *natfac* :: $nat \Rightarrow nat$
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*: $(((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) = (number\text{-}of\ y)) = (x = y)$
  ⟨*proof*⟩

lemma *number-le*: $(((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) \leq (number\text{-}of\ y)) = (x \leq y)$
  ⟨*proof*⟩

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

lemma *number-diff*: $((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) - number\text{-}of\ y = number\text{-}of\ (x + (-\ y))$
  ⟨*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*: $real\ ((number\text{-}of\ v)::nat) = (if\ neg\ v\ then\ 0\ else\ number\text{-}of\ v)$
  ⟨*proof*⟩

lemma *compute-nat-of-int-number-of*: $nat\ ((number\text{-}of\ v)::int) = (number\text{-}of\ v)$
  ⟨*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* $0 \leq r - b$ *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* $0 \leq a$ *then*
if $0 \leq b$ *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* (*Numeral.Pls*) = *True*
⟨*proof*⟩

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

lemma *even-B0*: *even* (*x BIT Numeral.B0*) = *True*
⟨*proof*⟩

lemma *even-B1*: *even* (*x BIT Numeral.B1*) = *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