

# Matrix

Steven Obua

April 19, 2009

```
theory Matrix
imports Main
begin
```

```
types 'a infmatrix = nat  $\Rightarrow$  nat  $\Rightarrow$  'a
```

```
definition nonzero-positions :: (nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero)  $\Rightarrow$  nat  $\times$  nat  $\Rightarrow$  bool
where
```

```
  nonzero-positions A = {pos. A (fst pos) (snd pos)  $\sim$  0}
```

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

```
proof -
```

```
  have ( $\lambda j$  i. 0)  $\in$  {(f::(nat  $\Rightarrow$  nat  $\Rightarrow$  'a::zero)). finite (nonzero-positions f)}
```

```
    by (simp add: nonzero-positions-def)
```

```
  then show ?thesis by auto
```

```
qed
```

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

```
lemma finite-nonzero-positions : finite (nonzero-positions (Rep-matrix A))
```

```
apply (rule Abs-matrix-induct)
```

```
by (simp add: Abs-matrix-inverse matrix-def)
```

```
constdefs
```

```
  nrows :: ('a::zero) matrix  $\Rightarrow$  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  $\Rightarrow$  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  $\leq$  m
```

```
  shows (Rep-matrix A m n) = 0 (is ?concl)
```

```
proof cases
```

```
  assume nonzero-positions(Rep-matrix A) = {}
```

```

    then show (Rep-matrix A m n) = 0 by (simp add: nonzero-positions-def)
next
  assume a: nonzero-positions(Rep-matrix A) ≠ {}
  let ?S = fst'(nonzero-positions(Rep-matrix A))
  have c: finite (?S) by (simp add: finite-nonzero-positions)
  from hyp have d: Max (?S) < m by (simp add: a n rows-def)
  have m ∉ ?S
  proof -
    have m ∈ ?S ⇒ m ≤ Max(?S) by (simp add: Max-ge [OF c])
    moreover from d have ~ (m ≤ Max ?S) by (simp)
    ultimately show m ∉ ?S by (auto)
  qed
  thus Rep-matrix A m n = 0 by (simp add: nonzero-positions-def image-Collect)
qed

```

**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  
 by ((rule ext)+, simp)

**lemma** transpose-infmatrix: transpose-infmatrix (% j i. P j i) = (% j i. P i j)  
 apply (rule ext)+  
 by (simp add: transpose-infmatrix-def)

**lemma** transpose-infmatrix-closed[simp]: Rep-matrix (Abs-matrix (transpose-infmatrix (Rep-matrix x))) = transpose-infmatrix (Rep-matrix x)  
 apply (rule Abs-matrix-inverse)  
 apply (simp add: matrix-def nonzero-positions-def image-def)  
 proof -

```

    let ?A = {pos. Rep-matrix x (snd pos) (fst pos) ≠ 0}
    let ?swap = % pos. (snd pos, fst pos)
    let ?B = {pos. Rep-matrix x (fst pos) (snd pos) ≠ 0}
    have swap-image: ?swap' ?A = ?B
    apply (simp add: image-def)
    apply (rule set-ext)
    apply (simp)
  proof

```

```

    fix y
    assume hyp: ∃ a b. Rep-matrix x b a ≠ 0 ∧ y = (b, a)
    thus Rep-matrix x (fst y) (snd y) ≠ 0
    proof -
      from hyp obtain a b where (Rep-matrix x b a ≠ 0 & y = (b,a)) by blast

```

```

      then show Rep-matrix  $x$  (fst  $y$ ) (snd  $y$ )  $\neq 0$  by (simp)
    qed
  next
  fix  $y$ 
  assume hyp: Rep-matrix  $x$  (fst  $y$ ) (snd  $y$ )  $\neq 0$ 
  show  $\exists a b. (\text{Rep-matrix } x \ b \ a \neq 0 \ \& \ y = (b, a))$ 
    by (rule exI[of - snd  $y$ ], rule exI[of - fst  $y$ ]) (simp add: hyp)
  qed
  then have finite (?swap‘ $A$ )
  proof -
    have finite (nonzero-positions (Rep-matrix  $x$ )) by (simp add: finite-nonzero-positions)
    then have finite ? $B$  by (simp add: nonzero-positions-def)
    with swap-image show finite (?swap‘ $A$ ) by (simp)
  qed
  moreover
  have inj-on ?swap ? $A$  by (simp add: inj-on-def)
  ultimately show finite ? $A$  by (rule finite-imageD[of ?swap ? $A$ ])
qed

```

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

```

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

```

```

lemma transpose-matrix-inject: (transpose-matrix  $A = \text{transpose-matrix } B$ ) = ( $A = B$ )
apply (simp add: transpose-matrix-def)
apply (subst Rep-matrix-inject[THEN sym])+
apply (simp only: transpose-infmatrix-closed transpose-infmatrix-inject)
done

```

```

lemma transpose-matrix[simp]: Rep-matrix(transpose-matrix  $A$ )  $j \ i = \text{Rep-matrix } A \ i \ j$ 
by (simp add: transpose-matrix-def)

```

```

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

```

```

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

```

```

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

```

```

lemma ncols: ncols A <= n  $\implies$  Rep-matrix A m n = 0
proof -
  assume ncols A <= n
  then have nrows (transpose-matrix A) <= n by (simp)
  then have Rep-matrix (transpose-matrix A) n m = 0 by (rule nrows)
  thus Rep-matrix A m n = 0 by (simp add: transpose-matrix-def)
qed

lemma ncols-le: (ncols A <= n) = (! j i. n <= i  $\longrightarrow$  (Rep-matrix A j i) = 0) (is
- = ?st)
apply (auto)
apply (simp add: ncols)
proof (simp add: ncols-def, auto)
  let ?P = nonzero-positions (Rep-matrix A)
  let ?p = snd' ?P
  have a:finite ?p by (simp add: finite-nonzero-positions)
  let ?m = Max ?p
  assume  $\sim$ (Suc (?m) <= n)
  then have b:n <= ?m by (simp)
  fix a b
  assume (a,b)  $\in$  ?P
  then have ?p  $\neq$  {} by (auto)
  with a have ?m  $\in$  ?p by (simp)
  moreover have !x. (x  $\in$  ?p  $\longrightarrow$  (? y. (Rep-matrix A y x)  $\neq$  0)) by (simp add:
nonzero-positions-def image-def)
  ultimately have ? y. (Rep-matrix A y ?m)  $\neq$  0 by (simp)
  moreover assume ?st
  ultimately show False using b by (simp)
qed

lemma less-ncols: (n < ncols A) = (? j i. n <= i & (Rep-matrix A j i)  $\neq$  0) (is
?concl)
proof -
  have a: !! (a::nat) b. (a < b) = ( $\sim$ (b <= a)) by arith
  show ?concl by (simp add: a ncols-le)
qed

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)
apply (auto)
apply (subgoal-tac ncols A <= m)
apply (simp)
apply (simp add: ncols-le)
apply (drule-tac x=ncols A in spec)
by (simp add: ncols)

lemma nrows-le: (nrows A <= n) = (! j i. n <= j  $\longrightarrow$  (Rep-matrix A j i) = 0)
(is ?s)

```

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

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

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

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

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

**lemma finite-natarray1:**  $\text{finite } \{x. x < (n::\text{nat})\}$   
**apply** (*induct n*)  
**apply** (*simp*)  
**proof** –  
 fix  $n$   
 have  $\{x. x < \text{Suc } n\} = \text{insert } n \ \{x. x < n\}$  **by** (*rule set-ext, simp, arith*)  
 moreover assume  $\text{finite } \{x. x < n\}$   
 ultimately show  $\text{finite } \{x. x < \text{Suc } n\}$  **by** (*simp*)  
**qed**

**lemma finite-natarray2:**  $\text{finite } \{\text{pos}. (\text{fst pos}) < (m::\text{nat}) \ \& \ (\text{snd pos}) < (n::\text{nat})\}$   
**apply** (*induct m*)  
**apply** (*simp+*)  
**proof** –

```

fix m::nat
let ?s0 = {pos. fst pos < m & snd pos < n}
let ?s1 = {pos. fst pos < (Suc m) & snd pos < n}
let ?sd = {pos. fst pos = m & snd pos < n}
assume f0: finite ?s0
have f1: finite ?sd
proof -
  let ?f = % x. (m, x)
  have {pos. fst pos = m & snd pos < n} = ?f ‘ {x. x < n} by (rule set-ext,
simp add: image-def, auto)
  moreover have finite {x. x < n} by (simp add: finite-natarray1)
  ultimately show finite {pos. fst pos = m & snd pos < n} by (simp)
qed
have su: ?s0 ∪ ?sd = ?s1 by (rule set-ext, simp, arith)
from f0 f1 have finite (?s0 ∪ ?sd) by (rule finite-UnI)
with su show finite ?s1 by (simp)
qed

```

**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
apply (rule Abs-matrix-inverse)
apply (simp add: matrix-def nonzero-positions-def)
proof -
  from aem obtain m where a: ! j i. m <= j ⟶ x j i = 0 by (blast)
  from aen obtain n where b: ! j i. n <= i ⟶ x j i = 0 by (blast)
  let ?u = {pos. x (fst pos) (snd pos) ≠ 0}
  let ?v = {pos. fst pos < m & snd pos < n}
  have c: !! (m::nat) a. ~ (m <= a) ⟹ a < m by (arith)
  from a b have (?u ∩ (~ ?v)) = {}
  apply (simp)
  apply (rule set-ext)
  apply (simp)
  apply auto
  by (rule c, auto)+
  then have d: ?u ⊆ ?v by blast
  moreover have finite ?v by (simp add: finite-natarray2)
  ultimately show finite ?u by (rule finite-subset)
qed

```

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

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

**constdefs**

```

commutative :: ('a ⇒ 'a ⇒ 'b) ⇒ bool
commutative f == ! x y. f x y = f y x
associative :: ('a ⇒ 'a ⇒ 'a) ⇒ 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 situation: Assume that we have sets  $A$  and  $B$  with  $B \subset A$  and an abstraction  $u : A \rightarrow B$ . This abstraction has to fulfill  $u(b) = b$  for all  $b \in B$ , but is arbitrary otherwise. Each function  $f : A \times A \rightarrow A$  now induces a function  $f' : B \times B \rightarrow B$  by  $f' = u \circ f$ . It is obvious that commutativity of  $f$  implies commutativity of  $f'$ :  $f'xy = u(fxy) = u(fyx) = f'yx$ .

**lemma** *combine-infmatrix-commute*:  
*commutative f ⇒ commutative (combine-infmatrix f)*  
**by** (*simp add: commutative-def combine-infmatrix-def*)

**lemma** *combine-matrix-commute*:  
*commutative f ⇒ commutative (combine-matrix f)*  
**by** (*simp add: combine-matrix-def commutative-def combine-infmatrix-def*)

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

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

but on the other hand we have

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

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

**lemma** *nonzero-positions-combine-infmatrix[simp]*: *f 0 0 = 0 ⇒ nonzero-positions (combine-infmatrix f A B) ⊆ (nonzero-positions A) ∪ (nonzero-positions B)*  
**by** (*rule subsetI, simp add: nonzero-positions-def combine-infmatrix-def, auto*)

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

**by** (*insert Rep-matrix [of A], simp add: matrix-def*)

**lemma** *combine-infmatrix-closed* [simp]:

*f 0 0 = 0  $\implies$  Rep-matrix (Abs-matrix (combine-infmatrix f (Rep-matrix A) (Rep-matrix B))) = combine-infmatrix f (Rep-matrix A) (Rep-matrix B)*

**apply** (*rule Abs-matrix-inverse*)

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

**apply** (*rule finite-subset[of - (nonzero-positions (Rep-matrix A))  $\cup$  (nonzero-positions (Rep-matrix B))]*)

**by** (*simp-all*)

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

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

**by** (*rule subsetI, simp add: nonzero-positions-def apply-infmatrix-def, auto*)

**lemma** *apply-infmatrix-closed* [simp]:

*f 0 = 0  $\implies$  Rep-matrix (Abs-matrix (apply-infmatrix f (Rep-matrix A))) = apply-infmatrix f (Rep-matrix A)*

**apply** (*rule Abs-matrix-inverse*)

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

**apply** (*rule finite-subset[of - nonzero-positions (Rep-matrix A)]*)

**by** (*simp-all*)

**lemma** *combine-infmatrix-assoc*[simp]: *f 0 0 = 0  $\implies$  associative f  $\implies$  associative (combine-infmatrix f)*

**by** (*simp add: associative-def combine-infmatrix-def*)

**lemma** *comb*: *f = g  $\implies$  x = y  $\implies$  f x = g y*

**by** (*auto*)

**lemma** *combine-matrix-assoc*: *f 0 0 = 0  $\implies$  associative f  $\implies$  associative (combine-matrix f)*

**apply** (*simp(no-asm) add: associative-def combine-matrix-def, auto*)

**apply** (*rule comb [of Abs-matrix Abs-matrix]*)

**by** (*auto, insert combine-infmatrix-assoc[of f], simp add: associative-def*)

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

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

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

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



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

**by** (*simp add: nrows-le*)

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

**by** (*simp add: ncols-le*)

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

**by** (*simp add: nrows-le*)

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

**by** (*simp add: ncols-le*)

**constdefs**

*zero-r-neutral* ::  $('a \Rightarrow 'b::\text{zero} \Rightarrow 'a) \Rightarrow \text{bool}$

*zero-r-neutral*  $f == ! a. f\ a\ 0 = a$

*zero-l-neutral* ::  $('a::\text{zero} \Rightarrow 'b \Rightarrow 'a) \Rightarrow \text{bool}$

*zero-l-neutral*  $f == ! a. f\ 0\ a = a$

*zero-closed* ::  $(( 'a::\text{zero} \Rightarrow ('b::\text{zero} \Rightarrow ('c::\text{zero})) \Rightarrow \text{bool}$

*zero-closed*  $f == (!x. f\ x\ 0 = 0) \ \&\ (!y. f\ 0\ y = 0)$

**consts** *foldseq* ::  $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a$

**primrec**

*foldseq*  $f\ s\ 0 = s\ 0$

*foldseq*  $f\ s\ (\text{Suc } n) = f\ (s\ 0)\ (\text{foldseq } f\ (\% k. s(\text{Suc } k))\ n)$

**consts** *foldseq-transposed* ::  $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a$

**primrec**

*foldseq-transposed*  $f\ s\ 0 = s\ 0$

*foldseq-transposed*  $f\ s\ (\text{Suc } n) = f\ (\text{foldseq-transposed } f\ s\ n)\ (s\ (\text{Suc } n))$

**lemma** *foldseq-assoc* : *associative*  $f \implies \text{foldseq } f = \text{foldseq-transposed } f$

**proof** –

**assume** *a:associative*  $f$

**then have** *sublemma*:  $!! n. ! N\ s. N \leq n \longrightarrow \text{foldseq } f\ s\ N = \text{foldseq-transposed } f\ s\ N$

**proof** –

**fix**  $n$

**show**  $!N\ s. N \leq n \longrightarrow \text{foldseq } f\ s\ N = \text{foldseq-transposed } f\ s\ N$

**proof** (*induct*  $n$ )

**show**  $!N\ s. N \leq 0 \longrightarrow \text{foldseq } f\ s\ N = \text{foldseq-transposed } f\ s\ N$  **by** *simp*

**next**

**fix**  $n$

**assume**  $b: !N\ s. N \leq n \longrightarrow \text{foldseq } f\ s\ N = \text{foldseq-transposed } f\ s\ N$

**have**  $c: !N\ s. N \leq n \implies \text{foldseq } f\ s\ N = \text{foldseq-transposed } f\ s\ N$  **by** (*simp add: b*)

```

show !  $N\ t.\ N \leq \text{Suc } n \longrightarrow \text{foldseq } f\ t\ N = \text{foldseq-transposed } f\ t\ N$ 
proof (auto)
  fix  $N\ t$ 
  assume  $N\text{suc}: N \leq \text{Suc } n$ 
  show  $\text{foldseq } f\ t\ N = \text{foldseq-transposed } f\ t\ N$ 
  proof cases
    assume  $N \leq n$ 
    then show  $\text{foldseq } f\ t\ N = \text{foldseq-transposed } f\ t\ N$  by (simp add: b)
  next
    assume  $\sim(N \leq n)$ 
    with  $N\text{suc}$  have  $N\text{suceq}: N = \text{Suc } n$  by simp
    have  $\text{negz}: n \neq 0 \implies ?m.\ n = \text{Suc } m \ \&\ \text{Suc } m \leq n$  by arith
    have  $\text{assocf}: !!\ x\ y\ z.\ f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z$  by (insert a, simp add:
associative-def)
    show  $\text{foldseq } f\ t\ N = \text{foldseq-transposed } f\ t\ N$ 
    apply (simp add: Nsuceq)
    apply (subst c)
    apply (simp)
    apply (case-tac n = 0)
    apply (simp)
    apply (drule negz)
    apply (erule exE)
    apply (simp)
    apply (subst assocf)
    proof –
      fix  $m$ 
      assume  $n = \text{Suc } m \ \&\ \text{Suc } m \leq n$ 
      then have  $\text{mless}: \text{Suc } m \leq n$  by arith
      then have  $\text{step1}: \text{foldseq-transposed } f\ (\% k.\ t\ (\text{Suc } k))\ m = \text{foldseq } f$ 
 $(\% k.\ t\ (\text{Suc } k))\ m$  (is  $?T1 = ?T2$ )
      apply (subst c)
      by simp+
      have  $\text{step2}: f\ (t\ 0)\ ?T2 = \text{foldseq } f\ t\ (\text{Suc } m)$  (is  $- = ?T3$ ) by simp
      have  $\text{step3}: ?T3 = \text{foldseq-transposed } f\ t\ (\text{Suc } m)$  (is  $- = ?T4$ )
      apply (subst c)
      by (simp add: mless)+
      have  $\text{step4}: ?T4 = f\ (\text{foldseq-transposed } f\ t\ m)\ (t\ (\text{Suc } m))$  (is  $- = ?T5$ )
by simp
      from  $\text{step1 step2 step3 step4}$  show  $\text{sowhat}: f\ (f\ (t\ 0)\ ?T1)\ (t\ (\text{Suc}$ 
 $(\text{Suc } m))) = f\ ?T5\ (t\ (\text{Suc } (\text{Suc } m)))$  by simp
      qed
    qed
  qed
  qed
  qed
  show  $\text{foldseq } f = \text{foldseq-transposed } f$  by ((rule ext)+, insert sublemma, auto)
qed

```

**lemma** *foldseq-distr*:  $\llbracket \text{associative } f; \text{commutative } f \rrbracket \implies \text{foldseq } f\ (\% k.\ f\ (u\ k)\ (v$

```

k)) n = f (foldseq f u n) (foldseq f v n)
proof -
  assume assoc: associative f
  assume comm: commutative f
  from assoc have a:!! x y z. f (f x y) z = f x (f y z) by (simp add: associative-def)
  from comm have b:!! x y. f x y = f y x by (simp add: commutative-def)
  from assoc comm have c:!! x y z. f x (f y z) = f y (f x z) by (simp add:
commutative-def associative-def)
  have !! n. (! u v. foldseq f (%k. f (u k) (v k)) n = f (foldseq f u n) (foldseq f v
n))
    apply (induct-tac n)
    apply (simp+, auto)
    by (simp add: a b c)
  then show foldseq f (% k. f (u k) (v k)) n = f (foldseq f u n) (foldseq f v n) by
simp
qed

```

```

theorem [[associative f; associative g;  $\forall a b c d. g (f a b) (f c d) = f (g a c) (g b d)$ ;
 $? x y. (f x) \neq (f y)$ ;  $? x y. (g x) \neq (g y)$ ;  $f x x = x$ ;  $g x x = x$ ]]  $\implies f=g \mid (! y. f y x = y) \mid (! y. g y x = y)$ 
oops

```

```

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 -
  have !! n. ! s. (! i. i <= n  $\longrightarrow$  s i = 0)  $\longrightarrow$  foldseq f s n = 0
    apply (induct-tac n)
    apply (simp)
    by (simp add: fz)
  then show foldseq f s n = 0 by (simp add: sz)
qed

```

```

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 -
  have !! m . ! s t. (! i. i <= m  $\longrightarrow$  s i = t i)  $\longrightarrow$  foldseq f s m = foldseq f t m
    apply (induct-tac m)
    apply (simp)
    apply (simp)
    apply (auto)
  proof -
    fix n
    fix s::nat $\Rightarrow$ 'a
    fix t::nat $\Rightarrow$ 'a
    assume a:  $\forall s t. (\forall i \leq n. s i = t i) \longrightarrow foldseq f s n = foldseq f t n$ 
    assume b:  $\forall i \leq \text{Suc } n. s i = t i$ 

```

```

    have c:!! a b. a = b  $\implies$  f (t 0) a = f (t 0) b by blast
    have d:!! s t. ( $\forall i \leq n. s\ i = t\ i$ )  $\implies$  foldseq f s n = foldseq f t n by (simp
add: a)
    show f (t 0) (foldseq f ( $\lambda k. s\ (Suc\ k)$ ) n) = f (t 0) (foldseq f ( $\lambda k. t\ (Suc\ k)$ ) n) by (rule c, simp add: d b)
  qed
  with p show ?concl by simp
qed

```

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

**proof** –

```

  have suc: !! a b.  $\llbracket a \leq Suc\ b; a \neq Suc\ b \rrbracket \implies a \leq b$  by arith
  have a:!! a b c . a = b  $\implies$  f c a = f c b by blast
  have !! n. ! m s. 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
  apply (induct-tac n)
  apply (simp)
  apply (simp)
  apply (auto)
  apply (case-tac m = Suc na)
  apply (simp)
  apply (rule a)
  apply (rule foldseq-significant-positions)
  apply (auto)
  apply (drule suc, simp+)
  proof –
    fix na m s
    assume suba:  $\forall m \leq na. \forall s. \text{foldseq } f\ s\ na = \text{foldseq } f\ (\lambda k. \text{if } k < m \text{ then } s\ k \text{ else } \text{foldseq } f\ (\lambda k. s\ (k + m))\ (na - m))\ m$ 
    assume subb: m <= na
    from suba have subc:!! m s. m <= na  $\implies$  foldseq f s na = foldseq f ( $\lambda k. \text{if } k < m \text{ then } s\ k \text{ else } \text{foldseq } f\ (\lambda k. s\ (k + m))\ (na - m))\ m$  by simp
    have subd: foldseq f ( $\lambda k. \text{if } k < m \text{ then } s\ (Suc\ k) \text{ else } \text{foldseq } f\ (\lambda k. s\ (Suc\ (k + m)))\ (na - m))\ m =$ 
      foldseq f (% k. s(Suc k)) na
    by (rule subc[of m % k. s(Suc k), THEN sym], simp add: subb)
    from subb have sube: m  $\neq 0 \implies$  ? mm. m = Suc mm & mm <= na by
arith
    show f (s 0) (foldseq f ( $\lambda k. \text{if } k < m \text{ then } s\ (Suc\ k) \text{ else } \text{foldseq } f\ (\lambda k. s\ (Suc\ (k + m)))\ (na - m))\ m) =$ 
      foldseq f ( $\lambda k. \text{if } k < m \text{ then } s\ k \text{ else } \text{foldseq } f\ (\lambda k. s\ (k + m))\ (Suc\ na - m))\ m
    apply (simp add: subd)
    apply (case-tac m=0)
    apply (simp)
    apply (drule sube)
    apply (auto)
    apply (rule a)
  qed$ 
```

```

      by (simp add: subc if-def)
    qed
  then show ?p  $\implies$  ?concl by simp
qed

```

**lemma** *foldseq-zerotail*:

```

  assumes
    fz:  $f\ 0\ 0 = 0$ 
  and sz:  $\! i. \ n \leq i \longrightarrow s\ i = 0$ 
  and nm:  $n \leq m$ 
  shows
     $foldseq\ f\ s\ n = foldseq\ f\ s\ m$ 
  proof -
    show  $foldseq\ f\ s\ n = foldseq\ f\ s\ m$ 
      apply (simp add: foldseq-tail[OF nm, of f s])
      apply (rule foldseq-significant-positions)
      apply (auto)
      apply (subst foldseq-zero)
      by (simp add: fz sz)+
  qed

```

**lemma** *foldseq-zerotail2*:

```

  assumes  $\! x. \ f\ x\ 0 = x$ 
  and  $\! i. \ n < i \longrightarrow s\ i = 0$ 
  and nm:  $n \leq m$ 
  shows
     $foldseq\ f\ s\ n = foldseq\ f\ s\ m$  (is ?concl)
  proof -
    have  $f\ 0\ 0 = 0$  by (simp add: prems)
    have b:  $\! m\ n. \ n \leq m \implies m \neq n \implies \exists k. \ m - n = Suc\ k$  by arith
    have c:  $0 \leq m$  by simp
    have d:  $\! k. \ k \neq 0 \implies \exists l. \ k = Suc\ l$  by arith
    show ?concl
      apply (subst foldseq-tail[OF nm])
      apply (rule foldseq-significant-positions)
      apply (auto)
      apply (case-tac m=n)
      apply (simp+)
      apply (drule b[OF nm])
      apply (auto)
      apply (case-tac k=0)
      apply (simp add: prems)
      apply (drule d)
      apply (auto)
      by (simp add: prems foldseq-zero)
  qed

```

**lemma** *foldseq-zerostart*:

```

 $\! x. \ f\ 0\ (f\ 0\ x) = f\ 0\ x \implies \! i. \ i \leq n \longrightarrow s\ i = 0 \implies foldseq\ f\ s\ (Suc\ n) = f$ 

```

```

0 (s (Suc n))
proof -
  assume f00x: ! x. f 0 (f 0 x) = f 0 x
  have ! s. (! i. i <= n ⟶ s i = 0) ⟶ foldseq f s (Suc n) = f 0 (s (Suc n))
    apply (induct n)
    apply (simp)
    apply (rule allI, rule impI)
    proof -
      fix n
      fix s
      have a: foldseq f s (Suc (Suc n)) = f (s 0) (foldseq f (% k. s (Suc k)) (Suc
n)) by simp
      assume b: ! s. ((∀ i ≤ n. s i = 0) ⟶ foldseq f s (Suc n) = f 0 (s (Suc n)))
      from b have c: ! s. (∀ i ≤ n. s i = 0) ⟹ foldseq f s (Suc n) = f 0 (s (Suc
n)) by simp
      assume d: ! i. i <= Suc n ⟶ s i = 0
      show foldseq f s (Suc (Suc n)) = f 0 (s (Suc (Suc n)))
        apply (subst a)
        apply (subst c)
        by (simp add: d f00x)+
      qed
    then show ! i. i <= n ⟶ s i = 0 ⟹ foldseq f s (Suc n) = f 0 (s (Suc n))
  by simp
qed

```

**lemma** foldseq-zerostart2:

```

! x. f 0 x = x ⟹ ! i. i < n ⟶ s i = 0 ⟹ foldseq f s n = s n
proof -
  assume a: ! i. i < n ⟶ s i = 0
  assume x: ! x. f 0 x = x
  from x have f00x: ! x. f 0 (f 0 x) = f 0 x by blast
  have b: ! i l. i < Suc l = (i <= l) by arith
  have d: ! k. k ≠ 0 ⟹ ? l. k = Suc l by arith
  show foldseq f s n = s n
    apply (case-tac n=0)
    apply (simp)
    apply (insert a)
    apply (drule d)
    apply (auto)
    apply (simp add: b)
    apply (insert f00x)
    apply (drule foldseq-zerostart)
    by (simp add: x)+
  qed

```

**lemma** foldseq-almostzero:

```

assumes f0x: ! 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 -

```

```

from  $s0$  have  $a: ! i. i < j \longrightarrow s\ i = 0$  by simp
from  $s0$  have  $b: ! i. j < i \longrightarrow s\ i = 0$  by simp
show  $?concl$ 
  apply auto
  apply (subst foldseq-zerotail2[of f, OF fx0, of j, OF b, of n, THEN sym])
  apply simp
  apply (subst foldseq-zerostart2)
  apply (simp add: f0x a)+
  apply (subst foldseq-zero)
  by (simp add: s0 f0x)+
qed

```

```

lemma foldseq-distr-unary:
  assumes  $!! a\ b. g\ (f\ a\ b) = f\ (g\ a)\ (g\ b)$ 
  shows  $g(\text{foldseq}\ f\ s\ n) = \text{foldseq}\ f\ (\% x. g(s\ x))\ n$  (is  $?concl$ )
proof  $-$ 
  have  $! s. g(\text{foldseq}\ f\ s\ n) = \text{foldseq}\ f\ (\% x. g(s\ x))\ n$ 
    apply (induct-tac n)
    apply (simp)
    apply (simp)
    apply (auto)
    apply (drule-tac x=% k. s (Suc k) in spec)
    by (simp add: prems)
  then show  $?concl$  by simp
qed

```

```

constdefs
   $\text{mult-matrix-}n :: \text{nat} \Rightarrow ((a::\text{zero}) \Rightarrow (b::\text{zero}) \Rightarrow (c::\text{zero})) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c)$ 
 $\Rightarrow 'a\ \text{matrix} \Rightarrow 'b\ \text{matrix} \Rightarrow 'c\ \text{matrix}$ 
   $\text{mult-matrix-}n\ n\ \text{fmul}\ \text{fadd}\ A\ B == \text{Abs-matrix}(\% j\ i. \text{foldseq}\ \text{fadd}\ (\% k. \text{fmul}$ 
 $(\text{Rep-matrix}\ A\ j\ k)\ (\text{Rep-matrix}\ B\ k\ i))\ n)$ 
   $\text{mult-matrix} :: ((a::\text{zero}) \Rightarrow (b::\text{zero}) \Rightarrow (c::\text{zero})) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'a$ 
 $\text{matrix} \Rightarrow 'b\ \text{matrix} \Rightarrow 'c\ \text{matrix}$ 
   $\text{mult-matrix}\ \text{fmul}\ \text{fadd}\ A\ B == \text{mult-matrix-}n\ (\max\ (\text{ncols}\ A)\ (\text{nrows}\ B))\ \text{fmul}$ 
 $\text{fadd}\ A\ B$ 

```

```

lemma mult-matrix-n:
  assumes  $\text{prems: ncols}\ A \leq n$  (is  $?An$ )  $\text{nrows}\ B \leq n$  (is  $?Bn$ )  $\text{fadd}\ 0\ 0 = 0$   $\text{fmul}\ 0\ 0 = 0$ 
  shows  $c:\text{mult-matrix}\ \text{fmul}\ \text{fadd}\ A\ B = \text{mult-matrix-}n\ n\ \text{fmul}\ \text{fadd}\ A\ B$  (is  $?concl$ )
proof  $-$ 
  show  $?concl$  using prems
    apply (simp add: mult-matrix-def mult-matrix-n-def)
    apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
    by (rule foldseq-zerotail, simp-all add: nrows-le ncols-le prems)
qed

```

```

lemma mult-matrix-nm:
  assumes  $\text{prems: ncols}\ A \leq n$   $\text{nrows}\ B \leq n$   $\text{ncols}\ A \leq m$   $\text{nrows}\ B \leq m$ 

```

```

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 -
  from prems have mult-matrix-n n fmul fadd A B = mult-matrix fmul fadd A B
  by (simp add: mult-matrix-n)
  also from prems have ... = mult-matrix-n m fmul fadd A B by (simp add:
  mult-matrix-n[THEN sym])
  finally show mult-matrix-n n fmul fadd A B = mult-matrix-n m fmul fadd A B
  by simp
qed

```

**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 **by** (arith)

**lemma** max2: !! b x y. (b::nat) <= y ⇒ b <= max x y **by** (arith)

**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 -
  from prems show ?concl
  apply (simp add: r-distributive-def mult-matrix-def, auto)
  proof -
    fix a::'a matrix
    fix u::'b matrix
    fix v::'b matrix
    let ?mx = max (ncols a) (max (nrows u) (nrows v))
    from prems show mult-matrix-n (max (ncols a) (nrows (combine-matrix fadd
u v))) fmul fadd a (combine-matrix fadd u v) =
      combine-matrix fadd (mult-matrix-n (max (ncols a) (nrows u)) fmul fadd a
u) (mult-matrix-n (max (ncols a) (nrows v)) fmul fadd a v)
    apply (subst mult-matrix-nm[of - - ?mx fadd fmul])
    apply (simp add: max1 max2 combine-nrows combine-ncols)+
    apply (subst mult-matrix-nm[of - - v ?mx fadd fmul])
    apply (simp add: max1 max2 combine-nrows combine-ncols)+

```



```

    apply (subst mult-matrix-nm[of - - u ?mx fadd fmul])
    apply (simp add: max1 max2 combine-rows combine-cols)+
    apply (simp add: mult-matrix-n-def r-distributive-def foldseq-distr[of fadd])
    apply (simp add: combine-matrix-def combine-infmatrix-def)
    apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
    apply (simplesubst RepAbs-matrix)
    apply (simp, auto)
    apply (rule exI[of - nrow a], simp add: nrow-le foldseq-zero)
    apply (rule exI[of - ncol v], simp add: ncol-le foldseq-zero)
    apply (subst RepAbs-matrix)
    apply (simp, auto)
    apply (rule exI[of - nrow a], simp add: nrow-le foldseq-zero)
    apply (rule exI[of - ncol u], simp add: ncol-le foldseq-zero)
  done
qed
qed

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 -
    from prems show ?concl
    apply (simp add: l-distributive-def mult-matrix-def, auto)
    proof -
      fix a::'b matrix
      fix u::'a matrix
      fix v::'a matrix
      let ?mx = max (nrow a) (max (ncol u) (ncol v))
      from prems show mult-matrix-n (max (ncol (combine-matrix fadd u v))
        (nrow a)) fmul fadd (combine-matrix fadd u v) a =
        combine-matrix fadd (mult-matrix-n (max (ncol u) (nrow a)) fmul
        fadd u a) (mult-matrix-n (max (ncol v) (nrow a)) fmul fadd v a)
      apply (subst mult-matrix-nm[of v - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-rows combine-cols)+
      apply (subst mult-matrix-nm[of u - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-rows combine-cols)+
      apply (subst mult-matrix-nm[of - - - ?mx fadd fmul])
      apply (simp add: max1 max2 combine-rows combine-cols)+
      apply (simp add: mult-matrix-n-def l-distributive-def foldseq-distr[of fadd])
      apply (simp add: combine-matrix-def combine-infmatrix-def)
      apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
      apply (simplesubst RepAbs-matrix)
      apply (simp, auto)
    end
  end

```

```

    apply (rule exI[of - nrows v], simp add: nrows-le foldseq-zero)
    apply (rule exI[of - ncols a], simp add: ncols-le foldseq-zero)
    apply (subst RepAbs-matrix)
    apply (simp, auto)
    apply (rule exI[of - nrows u], simp add: nrows-le foldseq-zero)
    apply (rule exI[of - ncols a], simp add: ncols-le foldseq-zero)
  done
qed
qed

instantiation matrix :: (zero) zero
begin

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

instance ..

end

lemma Rep-zero-matrix-def[simp]: Rep-matrix 0 j i = 0
  apply (simp add: zero-matrix-def)
  apply (subst RepAbs-matrix)
  by (auto)

lemma zero-matrix-def-nrows[simp]: nrows 0 = 0
proof -
  have a:!! (x::nat). x <= 0  $\implies$  x = 0 by (arith)
  show nrows 0 = 0 by (rule a, subst nrows-le, simp)
qed

lemma zero-matrix-def-ncols[simp]: ncols 0 = 0
proof -
  have a:!! (x::nat). x <= 0  $\implies$  x = 0 by (arith)
  show ncols 0 = 0 by (rule a, subst ncols-le, simp)
qed

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

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

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

```

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

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

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

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

**lemma** *apply-matrix-zero*[simp]:  $f \ 0 = 0 \implies$  *apply-matrix* *f* *0* = 0  
**apply** (simp add: *apply-matrix-def* *apply-infmatrix-def*)  
**by** (simp add: *zero-matrix-def*)

**lemma** *combine-matrix-zero*:  $f \ 0 \ 0 = 0 \implies$  *combine-matrix* *f* *0* *0* = 0  
**apply** (simp add: *combine-matrix-def* *combine-infmatrix-def*)  
**by** (simp add: *zero-matrix-def*)

**lemma** *transpose-matrix-zero*[simp]: *transpose-matrix* *0* = 0  
**apply** (simp add: *transpose-matrix-def* *transpose-infmatrix-def* *zero-matrix-def* *RepAbs-matrix*)  
**apply** (subst *Rep-matrix-inject*[*symmetric*], (rule *ext*)+)  
**apply** (simp add: *RepAbs-matrix*)  
**done**

**lemma** *apply-zero-matrix-def*[simp]: *apply-matrix* (% *x*. 0) *A* = 0  
**apply** (simp add: *apply-matrix-def* *apply-infmatrix-def*)  
**by** (simp add: *zero-matrix-def*)

**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*  $\Rightarrow$  *nat*  $\Rightarrow$  '*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*  $\Rightarrow$  *nat*  $\Rightarrow$  'a *matrix*  
*column-of-matrix* A n == take-columns (move-matrix A 0 (- int n)) 1  
*row-of-matrix* :: ('a::zero) *matrix*  $\Rightarrow$  *nat*  $\Rightarrow$  '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)  
**apply** (simp add: singleton-matrix-def)  
**apply** (auto)  
**apply** (subst RepAbs-matrix)  
**apply** (rule exI[of - Suc m], simp)  
**apply** (rule exI[of - Suc n], simp+)  
**by** (subst RepAbs-matrix, rule exI[of - Suc j], simp, rule exI[of - Suc i], simp+)+

**lemma** *apply-singleton-matrix[simp]*: f 0 = 0  $\implies$  apply-matrix f (singleton-matrix j i x) = (singleton-matrix j i (f x))  
**apply** (subst Rep-matrix-inject[symmetric])  
**apply** (rule ext)+  
**apply** (simp)  
**done**

**lemma** *singleton-matrix-zero[simp]*: singleton-matrix j i 0 = 0  
**by** (simp add: singleton-matrix-def zero-matrix-def)

**lemma** *nrows-singleton[simp]*: nrows(singleton-matrix j i e) = (if e = 0 then 0 else Suc j)  
**proof**–  
**have** th:  $\neg (\forall m. m \leq j) \exists n. \neg n \leq i$  **by** arith+  
**from** th **show** ?thesis  
**apply** (auto)  
**apply** (rule le-anti-sym)  
**apply** (subst nrows-le)  
**apply** (simp add: singleton-matrix-def, auto)  
**apply** (subst RepAbs-matrix)  
**apply** auto  
**apply** (simp add: Suc-le-eq)  
**apply** (rule not-leE)  
**apply** (subst nrows-le)  
**by** simp  
**qed**

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

```

apply (auto)
apply (rule le-anti-sym)
apply (subst ncols-le)
apply (simp add: singleton-matrix-def, auto)
apply (subst RepAbs-matrix)
apply auto
apply (simp add: Suc-le-eq)
apply (rule not-leE)
apply (subst ncols-le)
by simp
qed

```

```

lemma combine-singleton: f 0 0 = 0  $\implies$  combine-matrix f (singleton-matrix j i a) (singleton-matrix j i b) = singleton-matrix j i (f a b)
apply (simp add: singleton-matrix-def combine-matrix-def combine-infmatrix-def)
apply (subst RepAbs-matrix)
apply (rule exI[of - Suc j], simp)
apply (rule exI[of - Suc i], simp)
apply (rule comb[of Abs-matrix Abs-matrix], simp, (rule ext)+)
apply (subst RepAbs-matrix)
apply (rule exI[of - Suc j], simp)
apply (rule exI[of - Suc i], simp)
by simp

```

```

lemma transpose-singleton[simp]: transpose-matrix (singleton-matrix j i a) = singleton-matrix i j a
apply (subst Rep-matrix-inject[symmetric], (rule ext)+)
apply (simp)
done

```

```

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)))
apply (simp add: move-matrix-def)
apply (auto)
by (subst RepAbs-matrix,
  rule exI[of - (nrows A) + (nat (abs y))], auto, rule nrows, arith,
  rule exI[of - (ncols A) + (nat (abs x))], auto, rule ncols, arith)+

```

```

lemma move-matrix-0-0[simp]: move-matrix A 0 0 = A
by (simp add: move-matrix-def)

```

```

lemma move-matrix-ortho: move-matrix A j i = move-matrix (move-matrix A j 0) 0 i
apply (subst Rep-matrix-inject[symmetric])
apply (rule ext)+
apply (simp)
done

```

```

lemma transpose-move-matrix[simp]:
  transpose-matrix (move-matrix A x y) = move-matrix (transpose-matrix A) y x
apply (subst Rep-matrix-inject[symmetric], (rule ext)+)
apply (simp)
done

lemma move-matrix-singleton[simp]: move-matrix (singleton-matrix u v x) j i =
  (if (j + int u < 0) | (i + int v < 0) then 0 else (singleton-matrix (nat (j + int
u)) (nat (i + int v)) x))
apply (subst Rep-matrix-inject[symmetric])
apply (rule ext)+
apply (case-tac j + int u < 0)
apply (simp, arith)
apply (case-tac i + int v < 0)
apply (simp add: neg-def, arith)
apply (simp add: neg-def)
apply arith
done

lemma Rep-take-columns[simp]:
  Rep-matrix (take-columns A c) j i =
  (if i < c then (Rep-matrix A j i) else 0)
apply (simp add: take-columns-def)
apply (simplesubst RepAbs-matrix)
apply (rule exI[of - nrows A], auto, simp add: nrows-le)
apply (rule exI[of - ncols A], auto, simp add: ncols-le)
done

lemma Rep-take-rows[simp]:
  Rep-matrix (take-rows A r) j i =
  (if j < r then (Rep-matrix A j i) else 0)
apply (simp add: take-rows-def)
apply (simplesubst RepAbs-matrix)
apply (rule exI[of - nrows A], auto, simp add: nrows-le)
apply (rule exI[of - ncols A], auto, simp add: ncols-le)
done

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

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

lemma column-of-matrix: ncols A ≤ n  $\implies$  column-of-matrix A n = 0
apply (subst Rep-matrix-inject[THEN sym])

```

```

apply (rule ext)+
by (simp add: ncols)

```

```

lemma row-of-matrix: nrow A ≤ n ⇒ row-of-matrix A n = 0
apply (subst Rep-matrix-inject[THEN sym])
apply (rule ext)+
by (simp add: nrow)

```

```

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))
  apply (simp add: mult-matrix-def)
  apply (subst mult-matrix-nm[of - - max (ncols A) (Suc j)])
  apply (auto)
  apply (simp add: prems)+
  apply (simp add: mult-matrix-n-def apply-matrix-def apply-infmatrix-def)
  apply (rule comb[of Abs-matrix Abs-matrix], auto, (rule ext)+)
  apply (subst foldseq-almostzero[of - j])
  apply (simp add: prems)+
  apply (auto)
  apply (metis comm-monoid-add.mult-1 le-anti-sym le-diff-eq not-neg-nat zero-le-imp-of-nat
    zle-int)
  done

```

```

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(rule contrapos-np[of False], simp)
  assume a: A ≠ B
  have b: !! f g. (! x y. f x y = g x y) ⟹ f = g by ((rule ext)+, auto)
  have ? j i. (Rep-matrix A j i) ≠ (Rep-matrix B j i)
    apply (rule contrapos-np[of False], simp+)
    apply (insert b[of Rep-matrix A Rep-matrix B], simp)
    by (simp add: Rep-matrix-inject a)

```

```

then obtain  $J\ I$  where  $c:(\text{Rep-matrix } A\ J\ I) \neq (\text{Rep-matrix } B\ J\ I)$  by blast
from eprem obtain  $e$  where eprops:  $(! a\ b. a \neq b \longrightarrow \text{fmul } a\ e \neq \text{fmul } b\ e)$  by
blast
let ?S = singleton-matrix  $I\ 0\ e$ 
let ?comp = mult-matrix fmul fadd
have  $d: !!x\ f\ g. f = g \implies f\ x = g\ x$  by blast
have  $e: (\% x. \text{fmul } x\ e)\ 0 = 0$  by (simp add: prems)
have  $\sim(?comp\ A\ ?S = ?comp\ B\ ?S)$ 
  apply (rule notI)
  apply (simp add: fprems eprops)
  apply (simp add: Rep-matrix-inject[THEN sym])
  apply (drule d[of - - J], drule d[of - - 0])
  by (simp add: e c eprops)
with contraprems show False by simp
qed

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

lemma foldmatrix-transpose:
  assumes
     $! a\ b\ c\ d. g(f\ a\ b)\ (f\ c\ d) = f\ (g\ a\ c)\ (g\ b\ d)$ 
  shows
     $\text{foldmatrix } f\ g\ A\ m\ n = \text{foldmatrix-transposed } g\ f\ (\text{transpose-infmatrix } A)\ n\ m$ 
  (is ?concl)
  proof -
    have forall:  $!! P\ x. (! x. P\ x) \implies P\ x$  by auto
    have tworows:  $! A. \text{foldmatrix } f\ g\ A\ 1\ n = \text{foldmatrix-transposed } g\ f\ (\text{transpose-infmatrix } A)\ n\ 1$ 
      apply (induct n)
      apply (simp add: foldmatrix-def foldmatrix-transposed-def prems)+
      apply (auto)
      by (drule-tac  $x=(\% j\ i. A\ j\ (\text{Suc } i))$ ) in forall, simp)
    show  $\text{foldmatrix } f\ g\ A\ m\ n = \text{foldmatrix-transposed } g\ f\ (\text{transpose-infmatrix } A)\ n\ m$ 
      apply (simp add: foldmatrix-def foldmatrix-transposed-def)
      apply (induct m, simp)
      apply (simp)
      apply (insert tworows)
      apply (drule-tac  $x=\% j\ i. (\text{if } j = 0 \text{ then } (\text{foldseq-transposed } g\ (\lambda u. A\ u\ i)\ m) \text{ else } (A\ (\text{Suc } m)\ i))$ ) in spec)
      by (simp add: foldmatrix-def foldmatrix-transposed-def)
  qed

```



**lemma** *foldseq-foldseq*:  
**assumes**  
   *associative f*  
   *associative g*  
   ! *a b c d. g(f a b) (f c d) = f (g a c) (g b d)*  
**shows**  
   *foldseq g (% j. foldseq f (A j) n) m = foldseq f (% j. foldseq g ((transpose-infmatrix A) j) m) n*  
   **apply** (*insert foldmatrix-transpose[of g f A m n]*)  
   **by** (*simp add: foldmatrix-def foldmatrix-transposed-def foldseq-assoc[THEN sym]*  
*prems*)

**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*  
**apply** (*subst nrows-le*)  
**apply** (*simp add: mult-matrix-n-def*)  
**apply** (*subst RepAbs-matrix*)  
**apply** (*rule-tac x=nrows A in exI*)  
**apply** (*simp add: nrows prems foldseq-zero*)  
**apply** (*rule-tac x=ncols B in exI*)  
**apply** (*simp add: ncols prems foldseq-zero*)  
**by** (*simp add: nrows prems foldseq-zero*)

**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*  
**apply** (*subst ncols-le*)  
**apply** (*simp add: mult-matrix-n-def*)  
**apply** (*subst RepAbs-matrix*)  
**apply** (*rule-tac x=nrows A in exI*)  
**apply** (*simp add: nrows prems foldseq-zero*)  
**apply** (*rule-tac x=ncols B in exI*)  
**apply** (*simp add: ncols prems foldseq-zero*)  
**by** (*simp add: ncols prems foldseq-zero*)

**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*

**by** (*simp add: mult-matrix-def mult-n-rows prems*)

**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*

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

**lemma** *nrows-move-matrix-le:* *nrows (move-matrix A j i) ≤ nat((int (nrows A)) + j)*

**apply** (*auto simp add: nrows-le*)

**apply** (*rule nrows*)

**apply** (*arith*)

**done**

**lemma** *ncols-move-matrix-le:* *ncols (move-matrix A j i) ≤ nat((int (ncols A)) + i)*

**apply** (*auto simp add: ncols-le*)

**apply** (*rule ncols*)

**apply** (*arith*)

**done**

**lemma** *mult-matrix-assoc:*

**assumes** *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** –

**have** *comb-left:* **!!** *A B x y. A = B ⇒ (Rep-matrix (Abs-matrix A)) x y = (Rep-matrix (Abs-matrix B)) x y* **by** *blast*

**have** *fmul2fadd1fold:* **!!** *x s n. fmul2 (foldseq fadd1 s n) x = foldseq fadd1 (% k. fmul2 (s k) x) n*

**by** (*rule-tac g1 = % y. fmul2 y x in ssubst [OF foldseq-distr-unary], simp-all!*)

**have** *fmul1fadd2fold:* **!!** *x s n. fmul1 x (foldseq fadd2 s n) = foldseq fadd2 (% k. fmul1 x (s k)) n*

**by** (*rule-tac g1 = % y. fmul1 x y in ssubst [OF foldseq-distr-unary], simp-all!*)

```

let ?N = max (ncols A) (max (ncols B) (max (nrows B) (nrows C)))
show ?concl
  apply (simp add: Rep-matrix-inject[THEN sym])
  apply (rule ext)+
  apply (simp add: mult-matrix-def)
  apply (simplesubst mult-matrix-nm[of - max (ncols (mult-matrix-n (max (ncols
A) (nrows B)) fmul1 fadd1 A B)) (nrows C) - max (ncols B) (nrows C)])
  apply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simplesubst mult-matrix-nm[of - max (ncols A) (nrows (mult-matrix-n
(max (ncols B) (nrows C)) fmul2 fadd2 B C)) - max (ncols A) (nrows B)])  ap-
ply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simplesubst mult-matrix-nm[of - - ?N])
  apply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simplesubst mult-matrix-nm[of - - ?N])
  apply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simplesubst mult-matrix-nm[of - - ?N])
  apply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simplesubst mult-matrix-nm[of - - ?N])
  apply (simp add: max1 max2 mult-n-ncols mult-n-nrows prems)+
  apply (simp add: mult-matrix-n-def)
  apply (rule comb-left)
  apply ((rule ext)+, simp)
  apply (simplesubst RepAbs-matrix)
  apply (rule exI[of - nrows B])
  apply (simp add: nrows prems foldseq-zero)
  apply (rule exI[of - ncols C])
  apply (simp add: prems ncols foldseq-zero)
  apply (subst RepAbs-matrix)
  apply (rule exI[of - nrows A])
  apply (simp add: nrows prems foldseq-zero)
  apply (rule exI[of - ncols B])
  apply (simp add: prems ncols foldseq-zero)
  apply (simp add: fmul2fadd1fold fmul1fadd2fold prems)
  apply (subst foldseq-foldseq)
  apply (simp add: prems)+
  by (simp add: transpose-infmatrix)
qed

```

**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)
apply (rule ext)+
apply (simp add: comp-def )
by (simp add: mult-matrix-assoc prems)

```

**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 -
have !! a b c d. fadd (fadd a b) (fadd c d) = fadd (fadd a c) (fadd b d)
by (simp! add: associative-def commutative-def)
then show ?concl
apply (subst mult-matrix-assoc)
apply (simp-all!)
by (simp add: associative-def distributive-def l-distributive-def r-distributive-def)+
qed

```

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

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

**lemma** *Rep-mult-matrix*:

```

assumes
! 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))
  apply (simp add: mult-matrix-def mult-matrix-n-def)
  apply (subst RepAbs-matrix)
  apply (rule exI[of - nrows A], simp! add: nrows foldseq-zero)
  apply (rule exI[of - ncols B], simp! add: ncols foldseq-zero)
  by simp

```

**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)
  apply (simp add: Rep-matrix-inject[THEN sym])
  apply (rule ext)+
  by (simp! add: Rep-mult-matrix max-ac)

```

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

```

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

```

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

```

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

```

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

**definition**

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

**definition**

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

**instance** ..

**end**

**instance** *matrix* :: ( $\{\text{zero}, \text{order}\}$ ) *order*

```

  apply intro-classes
  apply (simp-all add: le-matrix-def less-def)

```

```

apply (auto)
apply (drule-tac  $x=j$  in spec, drule-tac  $x=j$  in spec)
apply (drule-tac  $x=i$  in spec, drule-tac  $x=i$  in spec)
apply (simp)
apply (simp add: Rep-matrix-inject[THEN sym])
apply (rule ext)+
apply (drule-tac  $x=xa$  in spec, drule-tac  $x=xa$  in spec)
apply (drule-tac  $x=xb$  in spec, drule-tac  $x=xb$  in spec)
apply simp
done

```

```

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

```

```

lemma le-combine-matrix:
  assumes
     $f\ 0\ 0 = 0$ 
     $! a\ b\ c\ d. a \leq b \ \&\ c \leq d \longrightarrow f\ a\ c \leq f\ b\ d$ 
     $A \leq B$ 
     $C \leq D$ 
  shows
     $\text{combine-matrix}\ f\ A\ C \leq \text{combine-matrix}\ f\ B\ D$ 
  by (simp! add: le-matrix-def)

```

```

lemma le-left-combine-matrix:
  assumes
     $f\ 0\ 0 = 0$ 
     $! a\ b\ c. a \leq b \longrightarrow f\ c\ a \leq f\ c\ b$ 
     $A \leq B$ 
  shows
     $\text{combine-matrix}\ f\ C\ A \leq \text{combine-matrix}\ f\ C\ B$ 
  by (simp! add: le-matrix-def)

```

```

lemma le-right-combine-matrix:
  assumes
     $f\ 0\ 0 = 0$ 
     $! a\ b\ c. a \leq b \longrightarrow f\ a\ c \leq f\ b\ c$ 
     $A \leq B$ 
  shows
     $\text{combine-matrix}\ f\ A\ C \leq \text{combine-matrix}\ f\ B\ C$ 
  by (simp! add: le-matrix-def)

```

```

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

```

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

**lemma** *le-foldseq*:  
**assumes**  
 $! a \ b \ c \ d. a \leq b \ \& \ c \leq d \longrightarrow f \ a \ c \leq f \ b \ d$   
 $! i. i \leq n \longrightarrow s \ i \leq t \ i$   
**shows**  
 $foldseq \ f \ s \ n \leq foldseq \ f \ t \ n$   
**proof** -  
**have**  $! s \ t. (! i. i \leq n \longrightarrow s \ i \leq t \ i) \longrightarrow foldseq \ f \ s \ n \leq foldseq \ f \ t \ n$  **by**  
(*induct-tac n, simp-all!*)  
**then show**  $foldseq \ f \ s \ n \leq foldseq \ f \ t \ n$  **by** (*simp!*)  
**qed**

**lemma** *le-left-mult*:  
**assumes**  
 $! a \ b \ c \ d. a \leq b \ \& \ c \leq d \longrightarrow fadd \ a \ c \leq fadd \ b \ d$   
 $! c \ a \ b. 0 \leq c \ \& \ a \leq b \longrightarrow fmul \ c \ a \leq fmul \ c \ b$   
 $! a. fmul \ 0 \ a = 0$   
 $! a. fmul \ a \ 0 = 0$   
 $fadd \ 0 \ 0 = 0$   
 $0 \leq C$   
 $A \leq B$   
**shows**  
 $mult-matrix \ fmul \ fadd \ C \ A \leq mult-matrix \ fmul \ fadd \ C \ B$   
**apply** (*simp! add: le-matrix-def Rep-mult-matrix*)  
**apply** (*auto*)  
**apply** (*simplesubst foldseq-zerotail[of - - - max (ncols C) (max (nrows A) (nrows B))], simp-all add: nrows ncols max1 max2*)  
**apply** (*rule le-foldseq*)  
**by** (*auto*)

**lemma** *le-right-mult*:  
**assumes**  
 $! a \ b \ c \ d. a \leq b \ \& \ c \leq d \longrightarrow fadd \ a \ c \leq fadd \ b \ d$   
 $! c \ a \ b. 0 \leq c \ \& \ a \leq b \longrightarrow fmul \ a \ c \leq fmul \ b \ c$   
 $! a. fmul \ 0 \ a = 0$   
 $! a. fmul \ a \ 0 = 0$   
 $fadd \ 0 \ 0 = 0$   
 $0 \leq C$   
 $A \leq B$   
**shows**  
 $mult-matrix \ fmul \ fadd \ A \ C \leq mult-matrix \ fmul \ fadd \ B \ C$   
**apply** (*simp! add: le-matrix-def Rep-mult-matrix*)  
**apply** (*auto*)  
**apply** (*simplesubst foldseq-zerotail[of - - - max (nrows C) (max (ncols A) (ncols B))], simp-all add: nrows ncols max1 max2*)  
**apply** (*rule le-foldseq*)

```

by (auto)

lemma spec2: ! j i. P j i  $\implies$  P j i by blast
lemma neg-imp: ( $\neg$  Q  $\longrightarrow$   $\neg$  P)  $\implies$  P  $\longrightarrow$  Q by blast

lemma singleton-matrix-le[simp]: (singleton-matrix j i a <= singleton-matrix j i
b) = (a <= (b::order))
  by (auto simp add: le-matrix-def)

lemma singleton-le-zero[simp]: (singleton-matrix j i x <= 0) = (x <= (0::'a::{order,zero}))
  apply (auto)
  apply (simp add: le-matrix-def)
  apply (drule-tac j=j and i=i in spec2)
  apply (simp)
  apply (simp add: le-matrix-def)
  done

lemma singleton-ge-zero[simp]: (0 <= singleton-matrix j i x) = ((0::'a::{order,zero})
<= x)
  apply (auto)
  apply (simp add: le-matrix-def)
  apply (drule-tac j=j and i=i in spec2)
  apply (simp)
  apply (simp add: le-matrix-def)
  done

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

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

lemma move-matrix-le-move-matrix-iff[simp]: 0 <= j  $\implies$  0 <= i  $\implies$  (move-matrix
A j i <= move-matrix B j i) = (A <= (B::('a::{order,zero}) matrix))
  apply (auto simp add: le-matrix-def neg-def)
  apply (drule-tac j=ja+(nat j) and i=ia+(nat i) in spec2)
  apply (auto)
  done

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

```



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

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

**instance**

**by** *default (auto simp add: inf-le1 inf-le2 le-infI le-matrix-def inf-matrix-def sup-matrix-def)*

**end**

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

**definition**

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

**instance** ..

**end**

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

**definition**

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

**instance** ..

**end**

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

**definition**

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

**instance** ..

**end**

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

**definition**

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

**instance** ..

**end**

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

**definition**

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

**instance** ..

**end**

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

**proof**

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

**show** *A* + *B* + *C* = *A* + (*B* + *C*)

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

**apply** (*rule combine-matrix-assoc*[*simplified associative-def*, *THEN spec*, *THEN spec*, *THEN spec*])

**apply** (*simp-all add: add-assoc*)

**done**

**show** *0* + *A* = *A*

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

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

**apply** (*simp*)

**done**

**show** *A* + *0* = *A*

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

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

**apply** (*simp*)

**done**

**qed**

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

**proof**

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

**show** *A* + *B* = *B* + *A*

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

**apply** (*rule combine-matrix-commute*[*simplified commutative-def*, *THEN spec*, *THEN spec*])

**apply** (*simp-all add: add-commute*)

**done**

**show** *0* + *A* = *A*

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

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

**apply** (*simp*)

done  
qed

**instance** *matrix* :: (group-add) group-add  
**proof**  
  **fix** *A B* :: 'a *matrix*  
  **show**  $- A + A = 0$   
    **by** (*simp add: plus-matrix-def minus-matrix-def Rep-matrix-inject[symmetric]*  
*ext*)  
  **show**  $A - B = A + - B$   
    **by** (*simp add: plus-matrix-def diff-matrix-def minus-matrix-def Rep-matrix-inject*  
*[symmetric] ext diff-minus*)  
qed

**instance** *matrix* :: (ab-group-add) ab-group-add  
**proof**  
  **fix** *A B* :: 'a *matrix*  
  **show**  $- A + A = 0$   
    **by** (*simp add: plus-matrix-def minus-matrix-def Rep-matrix-inject[symmetric]*  
*ext*)  
  **show**  $A - B = A + - B$   
    **by** (*simp add: plus-matrix-def diff-matrix-def minus-matrix-def Rep-matrix-inject[symmetric]*  
*ext*)  
qed

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

**instance** *matrix* :: (lordered-ab-group-add) lordered-ab-group-add-meet ..  
**instance** *matrix* :: (lordered-ab-group-add) lordered-ab-group-add-join ..

**instance** *matrix* :: (ring) ring  
**proof**  
  **fix** *A B C* :: 'a *matrix*  
  **show**  $A * B * C = A * (B * C)$   
    **apply** (*simp add: times-matrix-def*)  
    **apply** (*rule mult-matrix-assoc*)  
    **apply** (*simp-all add: associative-def algebra-simps*)  
  **done**  
  **show**  $(A + B) * C = A * C + B * C$   
    **apply** (*simp add: times-matrix-def plus-matrix-def*)

```

    apply (rule l-distributive-matrix[simplified l-distributive-def, THEN spec, THEN
spec, THEN spec])
    apply (simp-all add: associative-def commutative-def algebra-simps)
  done
show  $A * (B + C) = A * B + A * C$ 
  apply (simp add: times-matrix-def plus-matrix-def)
  apply (rule r-distributive-matrix[simplified r-distributive-def, THEN spec, THEN
spec, THEN spec])
  apply (simp-all add: associative-def commutative-def algebra-simps)
  done
qed

```

```

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

```

```

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

```

```

lemma Rep-matrix-add[simp]:
  Rep-matrix ((a::('a::monoid-add) matrix)+b) j i = (Rep-matrix a j i) + (Rep-matrix
b j i)
  by (simp add: plus-matrix-def)

```

```

lemma Rep-matrix-mult: Rep-matrix ((a::('a::ring) matrix) * b) j i =
  foldseq (op +) (% k. (Rep-matrix a j k) * (Rep-matrix b k i)) (max (ncols a)
(nrows b))
  apply (simp add: times-matrix-def)
  apply (simp add: Rep-mult-matrix)
  done

```

```

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

```

```

     $\implies \text{apply-matrix } f ((a::('a::\text{monoid-add}) \text{ matrix}) + b) = (\text{apply-matrix } f a) +$ 
     $(\text{apply-matrix } f b)$ 
  apply (subst Rep-matrix-inject[symmetric])
  apply (rule ext)+
  apply (simp)
done

```

```

lemma singleton-matrix-add: singleton-matrix j i ((a::monoid-add)+b) = (singleton-matrix
j i a) + (singleton-matrix j i b)
  apply (subst Rep-matrix-inject[symmetric])
  apply (rule ext)+
  apply (simp)
done

```

```

lemma nrows-mult: nrows ((A::('a::ring) matrix) * B) <= nrows A
by (simp add: times-matrix-def mult-nrows)

```

```

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

```

**definition**

```

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

```

```

lemma Rep-one-matrix[simp]: Rep-matrix (one-matrix n) j i = (if (j = i & j <
n) then 1 else 0)
  apply (simp add: one-matrix-def)
  apply (simpsubst RepAbs-matrix)
  apply (rule exI[of - n], simp add: split-if)+
  by (simp add: split-if)

```

```

lemma nrows-one-matrix[simp]: nrows ((one-matrix n) :: ('a::zero-neq-one)matrix)
= n (is ?r = -)
proof -
  have ?r <= n by (simp add: nrows-le)
  moreover have n <= ?r by (simp add: le-nrows, arith)
  ultimately show ?r = n by simp
qed

```

```

lemma ncols-one-matrix[simp]: ncols ((one-matrix n) :: ('a::zero-neq-one)matrix)
= n (is ?r = -)
proof -
  have ?r <= n by (simp add: ncols-le)
  moreover have n <= ?r by (simp add: le-ncols, arith)
  ultimately show ?r = n by simp
qed

```

```

lemma one-matrix-mult-right[simp]: ncols A <= n  $\implies$  (A::('a::{ring-1}) matrix)
* (one-matrix n) = A

```

```

apply (subst Rep-matrix-inject[THEN sym])
apply (rule ext)+
apply (simp add: times-matrix-def Rep-mult-matrix)
apply (rule-tac j1=xa in ssubst[OF foldseq-almostzero])
apply (simp-all)
by (simp add: max-def ncols)

```

```

lemma one-matrix-mult-left[simp]: nrow A ≤ n ⇒ (one-matrix n) * A =
(A::('a::ring-1) matrix)
apply (subst Rep-matrix-inject[THEN sym])
apply (rule ext)+
apply (simp add: times-matrix-def Rep-mult-matrix)
apply (rule-tac j1=x in ssubst[OF foldseq-almostzero])
apply (simp-all)
by (simp add: max-def nrow)

```

```

lemma transpose-matrix-mult: transpose-matrix ((A::('a::comm-ring) matrix)*B)
= (transpose-matrix B) * (transpose-matrix A)
apply (simp add: times-matrix-def)
apply (subst transpose-mult-matrix)
apply (simp-all add: mult-commute)
done

```

```

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

```

```

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

```

```

lemma transpose-matrix-minus: transpose-matrix (-(A::('a::group-add) matrix))
= - transpose-matrix (A::'a matrix)
by (simp add: minus-matrix-def transpose-apply-matrix)

```

**constdefs**

```

right-inverse-matrix :: ('a::{ring-1}) matrix ⇒ 'a matrix ⇒ bool
right-inverse-matrix A X == (A * X = one-matrix (max (nrow A) (ncol X)))
∧ nrow X ≤ ncol A
left-inverse-matrix :: ('a::{ring-1}) matrix ⇒ 'a matrix ⇒ bool
left-inverse-matrix A X == (X * A = one-matrix (max(nrow X) (ncol A))) ∧
ncol X ≤ nrow A
inverse-matrix :: ('a::{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 ⇒ nrow A = ncol X
apply (insert ncol-mult[of A X], insert nrow-mult[of A X])
by (simp add: right-inverse-matrix-def)

```

```

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

lemma left-right-inverse-matrix-unique:
  assumes left-inverse-matrix  $A$   $Y$  right-inverse-matrix  $A$   $X$ 
  shows  $X = Y$ 
proof –
  have  $Y = Y * \text{one-matrix } (\text{nrows } A)$ 
    apply (subst one-matrix-mult-right)
    apply (insert prems)
    by (simp-all add: left-inverse-matrix-def)
  also have  $\dots = Y * (A * X)$ 
    apply (insert prems)
    apply (frule right-inverse-matrix-dim)
    by (simp add: right-inverse-matrix-def)
  also have  $\dots = (Y * A) * X$  by (simp add: mult-assoc)
  also have  $\dots = X$ 
    apply (insert prems)
    apply (frule left-inverse-matrix-dim)
    apply (simp-all add: left-inverse-matrix-def right-inverse-matrix-def one-matrix-mult-left)
  done
  ultimately show  $X = Y$  by (simp)
qed

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

lemma one-matrix-inverse: inverse-matrix (one-matrix  $n$ ) (one-matrix  $n$ )
  by (simp add: inverse-matrix-def left-inverse-matrix-def right-inverse-matrix-def)

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

lemma Rep-matrix-zero-imp-mult-zero:
  !  $j \ i \ k$ . (Rep-matrix  $A \ j \ k = 0$ )  $\mid$  (Rep-matrix  $B \ k \ i = 0$ )  $\implies A * B =$ 
  ( $0 :: ('a :: \text{lordered-ring}) \text{ matrix}$ )
  apply (subst Rep-matrix-inject[symmetric])
  apply (rule ext) $+$ 
  apply (auto simp add: Rep-matrix-mult foldseq-zero zero-imp-mult-zero)
  done

lemma add-nrows: nrows ( $A :: ('a :: \text{monoid-add}) \text{ matrix}$ )  $\leq u \implies \text{nrows } B \leq u$ 
 $\implies \text{nrows } (A + B) \leq u$ 
  apply (simp add: plus-matrix-def)
  apply (rule combine-nrows)
  apply (simp-all)

```

done

**lemma** *move-matrix-row-mult*: *move-matrix* ((*A*::('a::ring) matrix) \* *B*) *j* 0 =  
 (*move-matrix* *A* *j* 0) \* *B*  
**apply** (*subst Rep-matrix-inject*[*symmetric*])  
**apply** (*rule ext*) +  
**apply** (*auto simp add: Rep-matrix-mult foldseq-zero*)  
**apply** (*rule-tac foldseq-zerotail*[*symmetric*])  
**apply** (*auto simp add: nrows zero-imp-mult-zero max2*)  
**apply** (*rule order-trans*)  
**apply** (*rule ncols-move-matrix-le*)  
**apply** (*simp add: max1*)  
 done

**lemma** *move-matrix-col-mult*: *move-matrix* ((*A*::('a::ring) matrix) \* *B*) 0 *i* = *A*  
 \* (*move-matrix* *B* 0 *i*)  
**apply** (*subst Rep-matrix-inject*[*symmetric*])  
**apply** (*rule ext*) +  
**apply** (*auto simp add: Rep-matrix-mult foldseq-zero*)  
**apply** (*rule-tac foldseq-zerotail*[*symmetric*])  
**apply** (*auto simp add: ncols zero-imp-mult-zero max1*)  
**apply** (*rule order-trans*)  
**apply** (*rule nrows-move-matrix-le*)  
**apply** (*simp add: max2*)  
 done

**lemma** *move-matrix-add*: ((*move-matrix* (*A* + *B*) *j* *i*)::('a::monoid-add) matrix))  
 = (*move-matrix* *A* *j* *i*) + (*move-matrix* *B* *j* *i*)  
**apply** (*subst Rep-matrix-inject*[*symmetric*])  
**apply** (*rule ext*) +  
**apply** (*simp*)  
 done

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

**constdefs**

*scalar-mult* :: ('a::ring)  $\Rightarrow$  'a matrix  $\Rightarrow$  'a matrix  
*scalar-mult* *a* *m* == *apply-matrix* (*op* \* *a*) *m*

**lemma** *scalar-mult-zero*[*simp*]: *scalar-mult* *y* 0 = 0  
**by** (*simp add: scalar-mult-def*)

**lemma** *scalar-mult-add*: *scalar-mult* *y* (*a*+*b*) = (*scalar-mult* *y* *a*) + (*scalar-mult* *y*  
*b*)  
**by** (*simp add: scalar-mult-def apply-matrix-add algebra-simps*)

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



```

a j i)
by (simp add: scalar-mult-def)

lemma scalar-mult-singleton[simp]: scalar-mult y (singleton-matrix j i x) = singleton-matrix
j i (y * x)
apply (subst Rep-matrix-inject[symmetric])
apply (rule ext)+
apply (auto)
done

lemma Rep-minus[simp]: Rep-matrix (-(A:::group-add)) x y = - (Rep-matrix
A x y)
by (simp add: minus-matrix-def)

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

end

theory SparseMatrix
imports Matrix
begin

types
'a svec = (nat * 'a) list
'a smat = ('a svec) svec

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

lemma sparse-row-matrix-empty [simp]: sparse-row-matrix [] = 0
  by (simp add: sparse-row-matrix-def)

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

lemma foldl-distrstart[rule-format]: ! a x y. (f (g x y) a = g x (f y a))  $\implies$  ! x y.

```

```

(foldl f (g x y) l = g x (foldl f y l))
  by (induct l, auto)

```

```

lemma sparse-row-vector-cons[simp]:
  sparse-row-vector (a # arr) = (singleton-matrix 0 (fst a) (snd a)) + (sparse-row-vector
arr)
  apply (induct arr)
  apply (auto simp add: sparse-row-vector-def)
  apply (simp add: foldl-distrstart [of  $\lambda m x. m + \text{singleton-matrix } 0 \text{ (fst } x) \text{ (snd } x) \lambda x m. \text{ singleton-matrix } 0 \text{ (fst } x) \text{ (snd } x) + m]$ ])
  done

```

```

lemma sparse-row-vector-append[simp]:
  sparse-row-vector (a @ b) = (sparse-row-vector a) + (sparse-row-vector b)
  by (induct a) auto

```

```

lemma nrows-spvec[simp]: nrows (sparse-row-vector x) <= (Suc 0)
  apply (induct x)
  apply (simp-all add: add-nrows)
  done

```

```

lemma sparse-row-matrix-cons: sparse-row-matrix (a#arr) = ((move-matrix (sparse-row-vector
(snd a)) (int (fst a)) 0)) + sparse-row-matrix arr
  apply (induct arr)
  apply (auto simp add: sparse-row-matrix-def)
  apply (simp add: foldl-distrstart [of  $\lambda m x. m + (\text{move-matrix (sparse-row-vector (snd } x) \text{ (int (fst } x)) 0})$ 
 $\% a m. (\text{move-matrix (sparse-row-vector (snd } a) \text{ (int (fst } a)) 0) + m]$ ])
  done

```

```

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

```

```

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

```

```

primrec sorted-spmat :: 'a spat  $\Rightarrow$  bool where
  sorted-spmat [] = True
  | sorted-spmat (a#as) = ((sorted-spvec (snd a)) & (sorted-spmat as))

```

```

declare sorted-spvec.simps [simp del]

```

```

lemma sorted-spvec-empty[simp]: sorted-spvec [] = True
by (simp add: sorted-spvec.simps)

```

```

lemma sorted-spvec-cons1: sorted-spvec (a#as)  $\implies$  sorted-spvec as
apply (induct as)
apply (auto simp add: sorted-spvec.simps)
done

```

```

lemma sorted-spvec-cons2: sorted-spvec (a#b#t)  $\implies$  sorted-spvec (a#t)
apply (induct t)
apply (auto simp add: sorted-spvec.simps)
done

```

```

lemma sorted-spvec-cons3: sorted-spvec(a#b#t)  $\implies$  fst a < fst b
apply (auto simp add: sorted-spvec.simps)
done

```

```

lemma sorted-sparse-row-vector-zero[rule-format]: m <= n  $\longrightarrow$  sorted-spvec ((n,a)#arr)
 $\longrightarrow$  Rep-matrix (sparse-row-vector arr) j m = 0
apply (induct arr)
apply (auto)
apply (frule sorted-spvec-cons2, simp) +
apply (frule sorted-spvec-cons3, simp)
done

```

```

lemma sorted-sparse-row-matrix-zero[rule-format]: m <= n  $\longrightarrow$  sorted-spvec ((n,a)#arr)
 $\longrightarrow$  Rep-matrix (sparse-row-matrix arr) m j = 0
apply (induct arr)
apply (auto)
apply (frule sorted-spvec-cons2, simp)
apply (frule sorted-spvec-cons3, simp)
apply (simp add: sparse-row-matrix-cons neg-def)
done

```

```

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

```

```

primrec abs-spvec :: ('a::lordered-ab-group-add-abs) spvec  $\Rightarrow$  '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)
apply (induct v)
apply (simp-all add: sparse-row-vector-cons)
apply (simp add: Rep-matrix-inject[symmetric])
apply (rule ext) +
apply simp
done

```

```

instance matrix :: (ordered-ab-group-add-abs) ordered-ab-group-add-abs
apply default
unfolding abs-matrix-def ..

```

```

lemma sparse-row-vector-abs:
  sorted-spvec (v :: 'a::ordered-ring spvec)  $\implies$  sparse-row-vector (abs-spvec v) =
  abs (sparse-row-vector v)
  apply (induct v)
  apply simp-all
  apply (frule-tac sorted-spvec-cons1, simp)
  apply (simp only: Rep-matrix-inject[symmetric])
  apply (rule ext)+
  apply auto
  apply (subgoal-tac Rep-matrix (sparse-row-vector v) 0 a = 0)
  apply (simp)
  apply (rule sorted-sparse-row-vector-zero)
  apply auto
done

```

```

lemma sorted-spvec-minus-spvec:
  sorted-spvec v  $\implies$  sorted-spvec (minus-spvec v)
  apply (induct v)
  apply (simp)
  apply (frule sorted-spvec-cons1, simp)
  apply (simp add: sorted-spvec.simps split:list.split-asm)
done

```

```

lemma sorted-spvec-abs-spvec:
  sorted-spvec v  $\implies$  sorted-spvec (abs-spvec v)
  apply (induct v)
  apply (simp)
  apply (frule sorted-spvec-cons1, simp)
  apply (simp add: sorted-spvec.simps split:list.split-asm)
done

```

```

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

```

```

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

```

```

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

```

```

consts addmult-spvec :: ('a::ring) * 'a spvec * 'a spvec  $\Rightarrow$  '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*  
**by** (induct a) auto

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

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

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

**lemma** *sparse-row-vector-addmult-spvec*: *sparse-row-vector (addmult-spvec (y::'a::ordered-ring,*  
*a, b)) =*  
*(sparse-row-vector a) + (scalar-mult y (sparse-row-vector b))*  
**apply** (rule addmult-spvec.induct[of - y])  
**apply** (simp add: scalar-mult-add smult-spvec-cons sparse-row-vector-smult singleton-matrix-add)+  
**done**

**lemma** *sorted-smult-spvec*[rule-format]: *sorted-spvec a  $\implies$  sorted-spvec (smult-spvec*  
*y a)*  
**apply** (auto simp add: smult-spvec-def)  
**apply** (induct a)  
**apply** (auto simp add: sorted-spvec.simps split:list.split-asm)  
**done**

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

```

apply (auto)
done

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

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

lemma sorted-addmult-spvec[rule-format]:  $\text{sorted-spvec } a \longrightarrow \text{sorted-spvec } b \longrightarrow$ 
 $\text{sorted-spvec } (\text{addmult-spvec } (y, a, b))$ 
apply (rule addmult-spvec.induct[of - y a b])
apply (simp-all add: sorted-smult-spvec)
apply (rule conjI, intro strip)
apply (case-tac  $\sim(a < aa)$ )
apply (simp-all)
apply (frule-tac  $as=brr$  in sorted-spvec-cons1)
apply (simp add: sorted-spvec-addmult-spvec-helper)
apply (intro strip | rule conjI)+
apply (frule-tac  $as=arr$  in sorted-spvec-cons1)
apply (simp add: sorted-spvec-addmult-spvec-helper2)
apply (intro strip)
apply (frule-tac  $as=arr$  in sorted-spvec-cons1)
apply (frule-tac  $as=brr$  in sorted-spvec-cons1)
apply (simp)
apply (simp-all add: sorted-spvec-addmult-spvec-helper3)
done

consts
  mult-spvec-spmat :: ('a::lordered-ring) spvec * 'a spvec * 'a smat  $\Rightarrow$  '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 -
  have comp-1: !! a b. a < b  $\implies$  Suc 0 <= nat ((int b)-(int a)) by arith
  have not-iff: !! a b. a = b  $\implies$  ( $\sim$  a) = ( $\sim$  b) by simp
  have max-helper: !! a b.  $\sim$  (a <= max (Suc a) b)  $\implies$  False
    by arith
  {
    fix a
    fix v
    assume a:a < nrows(sparse-row-vector v)
    have b:nrows(sparse-row-vector v) <= 1 by simp
    note dummy = less-le-trans[of a nrows (sparse-row-vector v) 1, OF a b]
    then have a = 0 by simp
  }
  note nrows-helper = this
  show ?thesis
    apply (rule mult-spvec-spmat.induct)
    apply simp +
    apply (rule conjI)
    apply (intro strip)
    apply (frule-tac as=brr in sorted-spvec-cons1)
    apply (simp add: algebra-simps sparse-row-matrix-cons)
    apply (simpsubst Rep-matrix-zero-imp-mult-zero)
    apply (simp)
    apply (intro strip)
    apply (rule disjI2)
    apply (intro strip)
    apply (subst nrows)
    apply (rule order-trans[of - 1])
    apply (simp add: comp-1) +
    apply (subst Rep-matrix-zero-imp-mult-zero)
    apply (intro strip)
    apply (case-tac k <= aa)
    apply (rule-tac m1 = k and n1 = a and a1 = b in ssust[OF sorted-sparse-row-vector-zero])
    apply (simp-all)
    apply (rule impI)
    apply (rule disjI2)
    apply (rule nrows)
    apply (rule order-trans[of - 1])
    apply (simp-all add: comp-1)

    apply (intro strip | rule conjI) +
    apply (frule-tac as=arr in sorted-spvec-cons1)
    apply (simp add: algebra-simps)
    apply (subst Rep-matrix-zero-imp-mult-zero)

```

```

apply (simp)
apply (rule disjI2)
apply (intro strip)
apply (simp add: sparse-row-matrix-cons neg-def)
apply (case-tac a <= aa)
apply (erule sorted-sparse-row-matrix-zero)
apply (simp-all)
apply (intro strip)
apply (case-tac a=aa)
apply (simp-all)
apply (frule-tac as=arr in sorted-spvec-cons1)
apply (frule-tac as=brr in sorted-spvec-cons1)
apply (simp add: sparse-row-matrix-cons algebra-simps sparse-row-vector-addmult-spvec)
apply (rule-tac B1 = sparse-row-matrix brr in ssubst[OF Rep-matrix-zero-imp-mult-zero])
apply (auto)
apply (rule sorted-sparse-row-matrix-zero)
apply (simp-all)
apply (rule-tac A1 = sparse-row-vector arr in ssubst[OF Rep-matrix-zero-imp-mult-zero])
apply (auto)
apply (rule-tac m=k and n = aa and a = b and arr=arr in sorted-sparse-row-vector-zero)
apply (simp-all)
apply (simp add: neg-def)
apply (drule nrows-notzero)
apply (drule nrows-helper)
apply (arith)

apply (subst Rep-matrix-inject[symmetric])
apply (rule ext)+
apply (simp)
apply (subst Rep-matrix-mult)
apply (rule-tac j1=aa in ssubst[OF foldseq-almostzero])
apply (simp-all)
apply (intro strip, rule conjI)
apply (intro strip)
apply (drule-tac max-helper)
apply (simp)
apply (auto)
apply (rule zero-imp-mult-zero)
apply (rule disjI2)
apply (rule nrows)
apply (rule order-trans[of - 1])
apply (simp)
apply (simp)
done
qed

```

**lemma** *sorted-mult-spvec-spmat*[*rule-format*]:  
 $\text{sorted-spvec } (c::('a::\text{lordered-ring}) \text{ spvec}) \longrightarrow \text{sorted-spmat } B \longrightarrow \text{sorted-spvec}$   
 $(\text{mult-spvec-spmat } (c, a, B))$



```

apply (rule mult-spvec-spmat.induct[of - c a B])
apply (simp-all add: sorted-addmult-spvec)
done

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)
apply (induct A)
apply (auto simp add: sparse-row-matrix-cons sparse-row-mult-spvec-spmat algebra-simps
  move-matrix-mult)
done

lemma sorted-spvec-mult-spmat[rule-format]:
  sorted-spvec (A::('a::lordered-ring) spmat)  $\longrightarrow$  sorted-spvec (mult-spmat A B)
apply (induct A)
apply (auto)
apply (drule sorted-spvec-cons1, simp)
apply (case-tac A)
apply (auto simp add: sorted-spvec.simps)
done

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

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
by (induct a, auto)

```

```

lemma add-spvec-empty2[simp]: add-spvec (a, []) = a
  by (induct a, auto)

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

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)
  apply (rule add-spmat.induct)
  apply (auto simp add: sparse-row-matrix-cons sparse-row-vector-add move-matrix-add)
  done

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

lemma sorted-add-spvec-helper1[rule-format]: add-spvec ((a, b)#arr, brr) = (ab,
bb) # list  $\longrightarrow$  (ab = a | (brr  $\neq$  [] & ab = fst (hd brr)))
  proof -
    have (! x ab a. x = (a, b)#arr  $\longrightarrow$  add-spvec (x, brr) = (ab, bb) # list  $\longrightarrow$ 
(ab = a | (ab = fst (hd brr))))
      by (rule add-spvec.induct[of - - brr], auto)
    then show ?thesis
      by (case-tac brr, auto)
  qed

lemma sorted-add-spmat-helper1[rule-format]: add-spmat ((a, b)#arr, brr) = (ab,
bb) # list  $\longrightarrow$  (ab = a | (brr  $\neq$  [] & ab = fst (hd brr)))
  proof -
    have (! x ab a. x = (a, b)#arr  $\longrightarrow$  add-spmat (x, brr) = (ab, bb) # list  $\longrightarrow$ 
(ab = a | (ab = fst (hd brr))))
      by (rule add-spmat.induct[of - - brr], auto)
    then show ?thesis
      by (case-tac brr, auto)
  qed

```

```

lemma sorted-add-spvec-helper[rule-format]: add-spvec (arr, brr) = (ab, bb) # list
→ ((arr ≠ [] & ab = fst (hd arr)) | (brr ≠ [] & ab = fst (hd brr)))
  apply (rule add-spvec.induct[of - arr brr])
  apply (auto)
  done

lemma sorted-add-spmat-helper[rule-format]: add-spmat (arr, brr) = (ab, bb) #
list → ((arr ≠ [] & ab = fst (hd arr)) | (brr ≠ [] & ab = fst (hd brr)))
  apply (rule add-spmat.induct[of - arr brr])
  apply (auto)
  done

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

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

lemma sorted-add-spvec-helper2: add-spvec ((a,b)#arr, brr) = (ab, bb) # list ⇒
aa < a ⇒ sorted-spvec ((aa, ba) # brr) ⇒ aa < ab
  apply (drule sorted-add-spvec-helper1)
  apply (auto)
  apply (case-tac brr)
  apply (simp-all)
  apply (drule-tac sorted-spvec-cons3)
  apply (simp)
  done

lemma sorted-add-spmat-helper2: add-spmat ((a,b)#arr, brr) = (ab, bb) # list
⇒ aa < a ⇒ sorted-spvec ((aa, ba) # brr) ⇒ aa < ab
  apply (drule sorted-add-spmat-helper1)
  apply (auto)
  apply (case-tac brr)
  apply (simp-all)
  apply (drule-tac sorted-spvec-cons3)
  apply (simp)
  done

lemma sorted-spvec-add-spvec[rule-format]: sorted-spvec a → sorted-spvec b →
sorted-spvec (add-spvec (a, b))
  apply (rule add-spvec.induct[of - a b])
  apply (simp-all)
  apply (rule conjI)
  apply (intro strip)
  apply (simp)
  apply (frule-tac as=brr in sorted-spvec-cons1)

```

```

apply (simp)
apply (subst sorted-spvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (simp add: sorted-add-spvec-helper2)
apply (clarify)
apply (rule conjI)
apply (case-tac a=aa)
apply (simp)
apply (clarify)
apply (frule-tac as=arr in sorted-spvec-cons1, simp)
apply (subst sorted-spvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (simp add: sorted-add-spvec-helper2 add-spvec-commute)
apply (case-tac a=aa)
apply (simp-all)
apply (clarify)
apply (frule-tac as=arr in sorted-spvec-cons1)
apply (frule-tac as=brr in sorted-spvec-cons1)
apply (simp)
apply (subst sorted-spvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (drule-tac sorted-add-spvec-helper)
apply (auto)
apply (case-tac arr)
apply (simp-all)
apply (drule sorted-spvec-cons3)
apply (simp)
apply (case-tac brr)
apply (simp-all)
apply (drule sorted-spvec-cons3)
apply (simp)
done

```

```

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

```

```

apply (clarify)
apply (rule conjI)
apply (case-tac a=aa)
apply (simp)
apply (clarify)
apply (frule-tac as=as in sorted-spvec-cons1, simp)
apply (subst sorted-spvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (simp add: sorted-add-spmat-helper2 add-spmat-commute)
apply (case-tac a=aa)
apply (simp-all)
apply (clarify)
apply (frule-tac as=as in sorted-spvec-cons1)
apply (frule-tac as=bs in sorted-spvec-cons1)
apply (simp)
apply (subst sorted-spvec-step)
apply (simp split: list.split)
apply (clarify, simp)
apply (drule-tac sorted-add-spmat-helper)
apply (auto)
apply (case-tac as)
apply (simp-all)
apply (drule sorted-spvec-cons3)
apply (simp)
apply (case-tac bs)
apply (simp-all)
apply (drule sorted-spvec-cons3)
apply (simp)
done

```

```

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

```

**consts**

```

le-spvec :: ('a::lordered-ab-group-add) spvec * 'a spvec  $\Rightarrow$  bool
le-spmat :: ('a::lordered-ab-group-add) spmat * 'a spmat  $\Rightarrow$  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
  by (simp add: disj-matrices-def)

lemma disj-matrices-contr2: disj-matrices A B ==> Rep-matrix B j i ≠ 0 ==>
Rep-matrix A j i = 0
  by (simp add: disj-matrices-def)

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))
  apply (auto)
  apply (simp (no-asm-use) only: le-matrix-def disj-matrices-def)
  apply (intro strip)
  apply (erule conjE)+
  apply (drule-tac j=j and i=i in spec2)+
  apply (case-tac Rep-matrix B j i = 0)
  apply (case-tac Rep-matrix D j i = 0)
  apply (simp-all)
  apply (simp (no-asm-use) only: le-matrix-def disj-matrices-def)
  apply (intro strip)
  apply (erule conjE)+
  apply (drule-tac j=j and i=i in spec2)+
  apply (case-tac Rep-matrix A j i = 0)

```

```

apply (case-tac Rep-matrix C j i = 0)
apply (simp-all)
apply (erule add-mono)
apply (assumption)
done

lemma disj-matrices-zero1[simp]: disj-matrices 0 B
by (simp add: disj-matrices-def)

lemma disj-matrices-zero2[simp]: disj-matrices A 0
by (simp add: disj-matrices-def)

lemma disj-matrices-commute: disj-matrices A B = disj-matrices B A
by (auto simp add: disj-matrices-def)

lemma disj-matrices-add-le-zero: disj-matrices A B  $\implies$ 
  (A + B <= 0) = (A <= 0 & (B::('a::lordered-ab-group-add) matrix) <= 0)
by (rule disj-matrices-add[of A B 0 0, simplified])

lemma disj-matrices-add-zero-le: disj-matrices A B  $\implies$ 
  (0 <= A + B) = (0 <= A & 0 <= (B::('a::lordered-ab-group-add) matrix))
by (rule disj-matrices-add[of 0 0 A B, simplified])

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

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

lemma disj-sparse-row-singleton: i <= j  $\implies$  sorted-spvec((j,y)#v)  $\implies$  disj-matrices
  (sparse-row-vector v) (singleton-matrix 0 i x)
apply (simp add: disj-matrices-def)
apply (rule conjI)
apply (rule neg-imp)
apply (simp)
apply (intro strip)
apply (rule sorted-sparse-row-vector-zero)
apply (simp-all)
apply (intro strip)
apply (rule sorted-sparse-row-vector-zero)
apply (simp-all)
done

lemma disj-matrices-x-add: disj-matrices A B  $\implies$  disj-matrices A C  $\implies$  disj-matrices
  (A::('a::lordered-ab-group-add) matrix) (B+C)
apply (simp add: disj-matrices-def)
apply (auto)

```

```

apply (drule-tac j=j and i=i in spec2)+
apply (case-tac Rep-matrix B j i = 0)
apply (case-tac Rep-matrix C j i = 0)
apply (simp-all)
done

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

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)
by (auto simp add: disj-matrices-def)

lemma disj-move-sparse-vec-mat[simplified disj-matrices-commute]:
j <= a  $\implies$  sorted-spvec((a,c)#as)  $\implies$  disj-matrices (move-matrix (sparse-row-vector
b) (int j) i) (sparse-row-matrix as)
apply (auto simp add: neg-def disj-matrices-def)
apply (drule nrows-notzero)
apply (drule less-le-trans[OF - nrows-spvec])
apply (subgoal-tac ja = j)
apply (simp add: sorted-sparse-row-matrix-zero)
apply (arith)
apply (rule nrows)
apply (rule order-trans[of - 1 -])
apply (simp)
apply (case-tac nat (int ja - int j) = 0)
apply (case-tac ja = j)
apply (simp add: sorted-sparse-row-matrix-zero)
apply arith+
done

lemma disj-move-sparse-row-vector-twice:
j  $\neq$  u  $\implies$  disj-matrices (move-matrix (sparse-row-vector a) j i) (move-matrix
(sparse-row-vector b) u v)
apply (auto simp add: neg-def disj-matrices-def)
apply (rule nrows, rule order-trans[of - 1], simp, drule nrows-notzero, drule
less-le-trans[OF - nrows-spvec], arith)+
done

lemma le-spvec-iff-sparse-row-le[rule-format]: (sorted-spvec a)  $\longrightarrow$  (sorted-spvec
b)  $\longrightarrow$  (le-spvec (a,b)) = (sparse-row-vector a <= sparse-row-vector b)
apply (rule le-spvec.induct)
apply (simp-all add: sorted-spvec-cons1 disj-matrices-add-le-zero disj-matrices-add-zero-le

disj-sparse-row-singleton[OF order-refl] disj-matrices-commute)
apply (rule conjI, intro strip)
apply (simp add: sorted-spvec-cons1)
apply (subst disj-matrices-add-x-le)

```



```

apply (simp add: disj-sparse-row-singleton[OF less-imp-le] disj-matrices-x-add
disj-matrices-commute)
apply (simp add: disj-sparse-row-singleton[OF order-refl] disj-matrices-commute)
apply (simp, blast)
apply (intro strip, rule conjI, intro strip)
apply (simp add: sorted-spvec-cons1)
apply (subst disj-matrices-add-le-x)
apply (simp-all add: disj-sparse-row-singleton[OF order-refl] disj-sparse-row-singleton[OF
less-imp-le] disj-matrices-commute disj-matrices-x-add)
apply (blast)
apply (intro strip)
apply (simp add: sorted-spvec-cons1)
apply (case-tac a=aa, simp-all)
apply (subst disj-matrices-add)
apply (simp-all add: disj-sparse-row-singleton[OF order-refl] disj-matrices-commute)
done

```

```

lemma le-spvec-empty2-sparse-row[rule-format]: (sorted-spvec b)  $\longrightarrow$  (le-spvec (b, []))
= (sparse-row-vector b <= 0))
apply (induct b)
apply (simp-all add: sorted-spvec-cons1)
apply (intro strip)
apply (subst disj-matrices-add-le-zero)
apply (simp add: disj-matrices-commute disj-sparse-row-singleton sorted-spvec-cons1)
apply (rule-tac y = snd a in disj-sparse-row-singleton[OF order-refl])
apply (simp-all)
done

```

```

lemma le-spvec-empty1-sparse-row[rule-format]: (sorted-spvec b)  $\longrightarrow$  (le-spvec ([], b))
= (0 <= sparse-row-vector b))
apply (induct b)
apply (simp-all add: sorted-spvec-cons1)
apply (intro strip)
apply (subst disj-matrices-add-zero-le)
apply (simp add: disj-matrices-commute disj-sparse-row-singleton sorted-spvec-cons1)
apply (rule-tac y = snd a in disj-sparse-row-singleton[OF order-refl])
apply (simp-all)
done

```

```

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)
apply (rule le-spmat.induct)
apply (simp add: sparse-row-matrix-cons disj-matrices-add-le-zero disj-matrices-add-zero-le
disj-move-sparse-vec-mat[OF order-refl]
disj-matrices-commute sorted-spvec-cons1 le-spvec-empty2-sparse-row le-spvec-empty1-sparse-row)+
apply (rule conjI, intro strip)
apply (simp add: sorted-spvec-cons1)

```

```

apply (subst disj-matrices-add-x-le)
apply (rule disj-matrices-add-x)
apply (simp add: disj-move-sparse-row-vector-twice)
apply (simp add: disj-move-sparse-vec-mat[OF less-imp-le] disj-matrices-commute)
apply (simp add: disj-move-sparse-vec-mat[OF order-refl] disj-matrices-commute)
apply (simp, blast)
apply (intro strip, rule conjI, intro strip)
apply (simp add: sorted-spvec-cons1)
apply (subst disj-matrices-add-le-x)
apply (simp add: disj-move-sparse-vec-mat[OF order-refl])
apply (rule disj-matrices-x-add)
apply (simp add: disj-move-sparse-row-vector-twice)
apply (simp add: disj-move-sparse-vec-mat[OF less-imp-le] disj-matrices-commute)
apply (simp, blast)
apply (intro strip)
apply (case-tac a=aa)
apply (simp-all)
apply (subst disj-matrices-add)
apply (simp-all add: disj-matrices-commute disj-move-sparse-vec-mat[OF order-refl])
apply (simp add: sorted-spvec-cons1 le-spvec-iff-sparse-row-le)
done

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

consts
  abs-spmat :: ('a::lordered-ring) spmat  $\Rightarrow$  'a spmat
  minus-spmat :: ('a::lordered-ring) spmat  $\Rightarrow$  '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)
apply (induct A)
apply (simp-all add: sparse-row-vector-minus sparse-row-matrix-cons)
apply (subst Rep-matrix-inject[symmetric])
apply (rule ext)+
apply simp
done

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

```

```

have r:nrows (sparse-row-vector v) <= Suc 0 by (rule nrows-spvec)
show ?thesis
  apply (rule nrows)
  apply (subgoal-tac Suc 0 <= x)
  apply (insert r)
  apply (simp only:)
  apply (insert x)
  apply arith
  done
qed

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

lemma sorted-spvec-minus-spmat: sorted-spvec A  $\implies$  sorted-spvec (minus-spmat
A)
  apply (induct A)
  apply (simp)
  apply (frule sorted-spvec-cons1, simp)
  apply (simp add: sorted-spvec.simps split:list.split-asm)
  done

lemma sorted-spvec-abs-spmat: sorted-spvec A  $\implies$  sorted-spvec (abs-spmat A)
  apply (induct A)
  apply (simp)
  apply (frule sorted-spvec-cons1, simp)
  apply (simp add: sorted-spvec.simps split:list.split-asm)
  done

lemma sorted-spmat-minus-spmat: sorted-spmat A  $\implies$  sorted-spmat (minus-spmat
A)
  apply (induct A)
  apply (simp-all add: sorted-spvec-minus-spvec)
  done

```

```

lemma sorted-spmat-abs-spmat: sorted-spmat  $A \implies \text{sorted-spmat } (\text{abs-spmat } A)$ 
  apply (induct  $A$ )
  apply (simp-all add: sorted-spvec-abs-spvec)
  done

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

lemma sorted-spmat-diff-spmat: sorted-spmat  $A \implies \text{sorted-spmat } B \implies \text{sorted-spmat } (\text{diff-spmat } A\ B)$ 
  by (simp add: diff-spmat-def sorted-spmat-minus-spmat sorted-spmat-add-spmat)

lemma sorted-spvec-diff-spmat: sorted-spvec  $A \implies \text{sorted-spvec } B \implies \text{sorted-spvec } (\text{diff-spmat } A\ B)$ 
  by (simp add: diff-spmat-def sorted-spvec-minus-spmat sorted-spvec-add-spmat)

lemma sparse-row-diff-spmat: sparse-row-matrix (diff-spmat  $A\ B$ ) = (sparse-row-matrix  $A$ ) - (sparse-row-matrix  $B$ )
  by (simp add: diff-spmat-def sparse-row-add-spmat sparse-row-matrix-minus)

constdefs
  sorted-sparse-matrix :: 'a spmat  $\Rightarrow$  bool
  sorted-sparse-matrix  $A == (\text{sorted-spvec } A) \ \& \ (\text{sorted-spmat } A)$ 

lemma sorted-sparse-matrix-imp-spvec: sorted-sparse-matrix  $A \implies \text{sorted-spvec } A$ 
  by (simp add: sorted-sparse-matrix-def)

lemma sorted-sparse-matrix-imp-spmat: sorted-sparse-matrix  $A \implies \text{sorted-spmat } A$ 
  by (simp add: sorted-sparse-matrix-def)

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

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

```

**lemma** *if-case-eq*:  $(\text{if } b \text{ then } x \text{ else } y) = (\text{case } b \text{ of True} \Rightarrow x \mid \text{False} \Rightarrow y)$  **by** *simp*

**consts**

*pprt-spvec* ::  $('a::\{\text{lordered-ab-group-add}\}) \text{ spvec} \Rightarrow 'a \text{ spvec}$   
*nprrt-spvec* ::  $('a::\{\text{lordered-ab-group-add}\}) \text{ spvec} \Rightarrow 'a \text{ spvec}$   
*pprt-spmat* ::  $('a::\{\text{lordered-ab-group-add}\}) \text{ smat} \Rightarrow 'a \text{ smat}$   
*nprrt-spmat* ::  $('a::\{\text{lordered-ab-group-add}\}) \text{ smat} \Rightarrow 'a \text{ smat}$

**primrec**

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

**primrec**

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

**primrec**

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

**primrec**

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

**lemma** *pprt-add*:  $\text{disj-matrices } A \text{ } (B::(-::\text{lordered-ring}) \text{ matrix}) \Longrightarrow \text{pprt } (A+B)$   
 $= \text{pprt } A + \text{pprt } B$   
**apply** (*simp add: pprt-def sup-matrix-def*)  
**apply** (*simp add: Rep-matrix-inject[symmetric]*)  
**apply** (*rule ext*) +  
**apply** *simp*  
**apply** (*case-tac Rep-matrix A x xa ≠ 0*)  
**apply** (*simp-all add: disj-matrices-contr1*)  
**done**

**lemma** *nprrt-add*:  $\text{disj-matrices } A \text{ } (B::(-::\text{lordered-ring}) \text{ matrix}) \Longrightarrow \text{nprrt } (A+B)$   
 $= \text{nprrt } A + \text{nprrt } B$   
**apply** (*simp add: nprrt-def inf-matrix-def*)  
**apply** (*simp add: Rep-matrix-inject[symmetric]*)  
**apply** (*rule ext*) +  
**apply** *simp*  
**apply** (*case-tac Rep-matrix A x xa ≠ 0*)  
**apply** (*simp-all add: disj-matrices-contr1*)  
**done**

**lemma** *pprt-singleton[simp]*:  $\text{pprt } (\text{singleton-matrix } j \ i \ (x::(-::\text{lordered-ring}))) = \text{singleton-matrix}$

```

j i (pprt x)
  apply (simp add: pprt-def sup-matrix-def)
  apply (simp add: Rep-matrix-inject[symmetric])
  apply (rule ext)+
  apply simp
done

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma sparse-row-matrix-pprr: sorted-spvec (m :: 'a::lordered-ring spmat)  $\implies$ 
sorted-spmat m  $\implies$  sparse-row-matrix (pprr-spmat m) = pprp (sparse-row-matrix
m)
  apply (induct m)
  apply simp
  apply simp
  apply (frule sorted-spvec-cons1)
  apply (simp add: sparse-row-matrix-cons sparse-row-vector-pprr)
  apply (subst pprp-add)
  apply (subst disj-matrices-commute)
  apply (rule disj-move-sparse-vec-mat)
  apply auto
  apply (simp add: sorted-spvec.simps)
  apply (simp split: list.split)
  apply auto
  apply (simp add: pprp-move-matrix)
done

```

```

lemma sparse-row-matrix-nprp: sorted-spvec (m :: 'a::lordered-ring spmat)  $\implies$ 
sorted-spmat m  $\implies$  sparse-row-matrix (nprp-spmat m) = nprp (sparse-row-matrix
m)
  apply (induct m)
  apply simp
  apply simp
  apply (frule sorted-spvec-cons1)
  apply (simp add: sparse-row-matrix-cons sparse-row-vector-nprp)
  apply (subst nprp-add)
  apply (subst disj-matrices-commute)
  apply (rule disj-move-sparse-vec-mat)
  apply auto
  apply (simp add: sorted-spvec.simps)
  apply (simp split: list.split)
  apply auto
  apply (simp add: nprp-move-matrix)
done

```

```

lemma sorted-pprr-spvec: sorted-spvec v  $\implies$  sorted-spvec (pprr-spvec v)
  apply (induct v)
  apply (simp)
  apply (frule sorted-spvec-cons1)

```

```

apply simp
apply (simp add: sorted-spvec.simps split:list.split-asm)
done

lemma sorted-nprt-spvec: sorted-spvec v  $\implies$  sorted-spvec (npert-spvec v)
apply (induct v)
apply (simp)
apply (frule sorted-spvec-cons1)
apply simp
apply (simp add: sorted-spvec.simps split:list.split-asm)
done

lemma sorted-spvec-pprt-spmat: sorted-spvec m  $\implies$  sorted-spvec (pprt-spmat m)
apply (induct m)
apply (simp)
apply (frule sorted-spvec-cons1)
apply simp
apply (simp add: sorted-spvec.simps split:list.split-asm)
done

lemma sorted-spvec-nprt-spmat: sorted-spvec m  $\implies$  sorted-spvec (npert-spmat m)
apply (induct m)
apply (simp)
apply (frule sorted-spvec-cons1)
apply simp
apply (simp add: sorted-spvec.simps split:list.split-asm)
done

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

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

constdefs
  mult-est-spmat :: ('a::lordered-ring) spmat  $\Rightarrow$  'a spmat  $\Rightarrow$  'a spmat  $\Rightarrow$  'a spmat
   $\Rightarrow$  '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) (npert-spmat r2),
      add-spmat (mult-spmat (npert-spmat s2) (pprt-spmat r1), mult-spmat (npert-spmat
        s1) (npert-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*:  $(0::\text{number-ring}) = \text{Numeral0}$  **by** *simp*

**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*  
*nprrt-spmat.simps nprrt-spvec.simps*  
*mult-est-spmat-def*

**end**

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

**lemma** *linprog-dual-estimate*:  
**assumes**  
 $A * x \leq (b::'a::\text{lordered-ring})$   
 $0 \leq y$   
 $\text{abs } (A - A') \leq \delta A$   
 $b \leq b'$   
 $\text{abs } (c - c') \leq \delta c$   
 $\text{abs } x \leq r$   
**shows**  
 $c * x \leq y * b' + (y * \delta A + \text{abs } (y * A' - c') + \delta c) * r$   
**proof** –  
**from** *prems* **have**  $1: y * b \leq y * b'$  **by** (*simp add: mult-left-mono*)

```

    from prems have 2:  $y * (A * x) \leq y * b$  by (simp add: mult-left-mono)
    have 3:  $y * (A * x) = c * x + (y * (A - A') + (y * A' - c') + (c' - c)) * x$ 
    by (simp add: algebra-simps)
    from 1 2 3 have 4:  $c * x + (y * (A - A') + (y * A' - c') + (c' - c)) * x \leq$ 
 $y * b'$  by simp
    have 5:  $c * x \leq y * b' + \text{abs}((y * (A - A') + (y * A' - c') + (c' - c)) * x)$ 
    by (simp only: 4 estimate-by-abs)
    have 6:  $\text{abs}((y * (A - A') + (y * A' - c') + (c' - c)) * x) \leq \text{abs}(y * (A -$ 
 $A') + (y * A' - c') + (c' - c)) * \text{abs } x$ 
    by (simp add: abs-le-mult)
    have 7:  $(\text{abs}(y * (A - A') + (y * A' - c') + (c' - c))) * \text{abs } x \leq (\text{abs}(y * (A -$ 
 $A') + (y * A' - c') + \text{abs}(c' - c)) * \text{abs } x$ 
    by (rule abs-triangle-ineq [THEN mult-right-mono]) simp
    have 8:  $(\text{abs}(y * (A - A') + (y * A' - c') + \text{abs}(c' - c))) * \text{abs } x \leq (\text{abs}(y * (A -$ 
 $A') + \text{abs}(y * A' - c') + \text{abs}(c' - c)) * \text{abs } x$ 
    by (simp add: abs-triangle-ineq mult-right-mono)
    have 9:  $(\text{abs}(y * (A - A')) + \text{abs}(y * A' - c') + \text{abs}(c' - c)) * \text{abs } x \leq (\text{abs } y * \text{abs}(A - A') + \text{abs}(y * A' - c') + \text{abs}(c' - c)) * \text{abs } x$ 
    by (simp add: abs-le-mult mult-right-mono)
    have 10:  $c' - c = -(c - c')$  by (simp add: algebra-simps)
    have 11:  $\text{abs}(c' - c) = \text{abs}(c - c')$ 
    by (subst 10, subst abs-minus-cancel, simp)
    have 12:  $(\text{abs } y * \text{abs}(A - A') + \text{abs}(y * A' - c') + \text{abs}(c' - c)) * \text{abs } x \leq (\text{abs } y * \text{abs}(A - A') + \text{abs}(y * A' - c') + \delta c) * \text{abs } x$ 
    by (simp add: 11 prems mult-right-mono)
    have 13:  $(\text{abs } y * \text{abs}(A - A') + \text{abs}(y * A' - c') + \delta c) * \text{abs } x \leq (\text{abs } y * \delta A + \text{abs}(y * A' - c') + \delta c) * \text{abs } x$ 
    by (simp add: prems mult-right-mono mult-left-mono)
    have r:  $(\text{abs } y * \delta A + \text{abs}(y * A' - c') + \delta c) * \text{abs } x \leq (\text{abs } y * \delta A + \text{abs}(y * A' - c') + \delta c) * r$ 
    apply (rule mult-left-mono)
    apply (simp add: prems)
    apply (rule-tac add-mono[of 0::'a - 0, simplified]) +
    apply (rule mult-left-mono[of 0  $\delta A$ , simplified])
    apply (simp-all)
    apply (rule order-trans[where  $y = \text{abs}(A - A')$ , simp-all add: prems])
    apply (rule order-trans[where  $y = \text{abs}(c - c')$ , simp-all add: prems])
    done
    from 6 7 8 9 12 13 r have 14:  $\text{abs}((y * (A - A') + (y * A' - c') + (c' - c)) * x) \leq (\text{abs } y * \delta A + \text{abs}(y * A' - c') + \delta c) * r$ 
    by (simp)
    show ?thesis
    apply (rule-tac le-add-right-mono[of - -  $\text{abs}((y * (A - A') + (y * A' - c') + (c' - c)) * x)$ ])
    apply (simp-all only: 5 14 [simplified abs-of-nonneg[of y, simplified prems]])
    done
  qed

```

lemma le-ge-imp-abs-diff-1:

```

assumes
   $A1 \leq (A::'a::\text{ordered-ring})$ 
   $A \leq A2$ 
shows  $\text{abs } (A - A1) \leq A2 - A1$ 
proof -
  have  $0 \leq A - A1$ 
  proof -
    have  $1: A - A1 = A + (- A1)$  by simp
    show ?thesis by (simp only: 1 add-right-mono[of A1 A - A1, simplified, simplified prems])
  qed
  then have  $\text{abs } (A - A1) = A - A1$  by (rule abs-of-nonneg)
  with prems show  $\text{abs } (A - A1) \leq (A2 - A1)$  by simp
qed

lemma mult-le-prts:
assumes
   $a1 \leq (a::'a::\text{ordered-ring})$ 
   $a \leq a2$ 
   $b1 \leq b$ 
   $b \leq b2$ 
shows
   $a * b \leq \text{pprt } a2 * \text{pprt } b2 + \text{pprt } a1 * \text{nprt } b2 + \text{nprt } a2 * \text{pprt } b1 + \text{nprt } a1 * \text{nprt } b1$ 
proof -
  have  $a * b = (\text{pprt } a + \text{nprt } a) * (\text{pprt } b + \text{nprt } b)$ 
  apply (subst prts[symmetric]) +
  apply simp
  done
  then have  $a * b = \text{pprt } a * \text{pprt } b + \text{pprt } a * \text{nprt } b + \text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$ 
  by (simp add: algebra-simps)
  moreover have  $\text{pprt } a * \text{pprt } b \leq \text{pprt } a2 * \text{pprt } b2$ 
  by (simp-all add: prems mult-mono)
  moreover have  $\text{pprt } a * \text{nprt } b \leq \text{pprt } a1 * \text{nprt } b2$ 
  proof -
    have  $\text{pprt } a * \text{nprt } b \leq \text{pprt } a * \text{nprt } b2$ 
    by (simp add: mult-left-mono prems)
    moreover have  $\text{pprt } a * \text{nprt } b2 \leq \text{pprt } a1 * \text{nprt } b2$ 
    by (simp add: mult-right-mono-neg prems)
    ultimately show ?thesis
    by simp
  qed
  moreover have  $\text{nprt } a * \text{pprt } b \leq \text{nprt } a2 * \text{pprt } b1$ 
  proof -
    have  $\text{nprt } a * \text{pprt } b \leq \text{nprt } a2 * \text{pprt } b$ 
    by (simp add: mult-right-mono prems)
    moreover have  $\text{nprt } a2 * \text{pprt } b \leq \text{nprt } a2 * \text{pprt } b1$ 
    by (simp add: mult-left-mono-neg prems)

```

```

ultimately show ?thesis
  by simp
qed
moreover have  $\text{nprt } a * \text{nprt } b \leq \text{nprt } a1 * \text{nprt } b1$ 
proof -
  have  $\text{nprt } a * \text{nprt } b \leq \text{nprt } a * \text{nprt } b1$ 
    by (simp add: mult-left-mono-neg prems)
  moreover have  $\text{nprt } a * \text{nprt } b1 \leq \text{nprt } a1 * \text{nprt } b1$ 
    by (simp add: mult-right-mono-neg prems)
  ultimately show ?thesis
    by simp
qed
ultimately show ?thesis
  by - (rule add-mono | simp)+
qed

lemma mult-le-dual-prts:
  assumes
     $A * x \leq (b :: 'a :: \text{lordered-ring})$ 
     $0 \leq y$ 
     $A1 \leq A$ 
     $A \leq A2$ 
     $c1 \leq c$ 
     $c \leq c2$ 
     $r1 \leq x$ 
     $x \leq r2$ 
  shows
     $c * x \leq y * b + (\text{let } s1 = c1 - y * A2; s2 = c2 - y * A1 \text{ in } \text{pprt } s2 * \text{pprt } r2$ 
     $+ \text{pprt } s1 * \text{nprt } r2 + \text{nprt } s2 * \text{pprt } r1 + \text{nprt } s1 * \text{nprt } r1)$ 
    (is -  $\leq$  - + ?C)
  proof -
    from prems have  $y * (A * x) \leq y * b$  by (simp add: mult-left-mono)
    moreover have  $y * (A * x) = c * x + (y * A - c) * x$  by (simp add:
    algebra-simps)
    ultimately have  $c * x + (y * A - c) * x \leq y * b$  by simp
    then have  $c * x \leq y * b - (y * A - c) * x$  by (simp add: le-diff-eq)
    then have  $cx: c * x \leq y * b + (c - y * A) * x$  by (simp add: algebra-simps)
    have  $s2: c - y * A \leq c2 - y * A1$ 
      by (simp add: diff-def prems add-mono mult-left-mono)
    have  $s1: c1 - y * A2 \leq c - y * A$ 
      by (simp add: diff-def prems add-mono mult-left-mono)
    have prts:  $(c - y * A) * x \leq ?C$ 
      apply (simp add: Let-def)
      apply (rule mult-le-prts)
      apply (simp-all add: prems s1 s2)
    done
    then have  $y * b + (c - y * A) * x \leq y * b + ?C$ 
      by simp
    with cx show ?thesis

```

```

    by(simp only:)
qed

```

```

end

```

## 1 Floating Point Representation of the Reals

```

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

```

**definition**

```

  pow2 :: int ⇒ real where
  pow2 a = (if (0 ≤ a) then (2nat a) else (inverse (2nat (-a))))

```

**definition**

```

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

```

```

lemma pow2-0[simp]: pow2 0 = 1
by (simp add: pow2-def)

```

```

lemma pow2-1[simp]: pow2 1 = 2
by (simp add: pow2-def)

```

```

lemma pow2-neg: pow2 x = inverse (pow2 (-x))
by (simp add: pow2-def)

```

```

lemma pow2-add1: pow2 (1 + a) = 2 * (pow2 a)

```

**proof** –

```

  have h: ! n. nat (2 + int n) - Suc 0 = nat (1 + int n) by arith
  have g: ! a b. a - -1 = a + (1::int) by arith
  have pos: ! n. pow2 (int n + 1) = 2 * pow2 (int n)
    apply (auto, induct-tac n)
    apply (simp-all add: pow2-def)
    apply (rule-tac m1=2 and n1=nat (2 + int na) in ssust[OF realpow-num-eq-if])
    by (auto simp add: h)
  show ?thesis
proof (induct a)
    case (1 n)
    from pos show ?case by (simp add: algebra-simps)
  next
    case (2 n)
    show ?case
      apply (auto)
      apply (subst pow2-neg[of - int n])
      apply (subst pow2-neg[of -1 - int n])
      apply (auto simp add: g pos)

```

```

    done
  qed
qed

lemma pow2-add: pow2 (a+b) = (pow2 a) * (pow2 b)
proof (induct b)
  case (1 n)
  show ?case
  proof (induct n)
    case 0
    show ?case by simp
  next
    case (Suc m)
    show ?case by (auto simp add: algebra-simps pow2-add1 prems)
  qed
next
  case (2 n)
  show ?case
  proof (induct n)
    case 0
    show ?case
    apply (auto)
    apply (subst pow2-neg[of a + -1])
    apply (subst pow2-neg[of -1])
    apply (simp)
    apply (insert pow2-add1[of -a])
    apply (simp add: algebra-simps)
    apply (subst pow2-neg[of -a])
    apply (simp)
    done
  case (Suc m)
  have a: int m - (a + -2) = 1 + (int m - a + 1) by arith
  have b: int m - -2 = 1 + (int m + 1) by arith
  show ?case
  apply (auto)
  apply (subst pow2-neg[of a + (-2 - int m)])
  apply (subst pow2-neg[of -2 - int m])
  apply (auto simp add: algebra-simps)
  apply (subst a)
  apply (subst b)
  apply (simp only: pow2-add1)
  apply (subst pow2-neg[of int m - a + 1])
  apply (subst pow2-neg[of int m + 1])
  apply auto
  apply (insert prems)
  apply (auto simp add: algebra-simps)
  done
  qed
qed

```

**lemma** *float* (a, e) + *float* (b, e) = *float* (a + b, e)  
**by** (*simp add: float-def algebra-simps*)

**definition**

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

**definition**

*real-is-int* :: *real*  $\Rightarrow$  *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))  
**by** (*auto simp add: real-is-int-def int-of-real-def*)

**lemma** *float-transfer*: *real-is-int* ((*real* a)\*(*pow2* c))  $\Longrightarrow$  *float* (a, b) = *float* (*int-of-real* ((*real* a)\*(*pow2* c)), b - c)  
**by** (*simp add: float-def real-is-int-def2 pow2-add[symmetric]*)

**lemma** *pow2-int*: *pow2* (*int* c) =  $2^c$   
**by** (*simp add: pow2-def*)

**lemma** *float-transfer-nat*: *float* (a, b) = *float* (a \*  $2^c$ , b - *int* c)  
**by** (*simp add: float-def pow2-int[symmetric] pow2-add[symmetric]*)

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

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

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

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

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

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

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

**apply** (*simp add: int-of-real-sub real-int-of-real*)  
**done**

**lemma** *real-is-int-rep*: *real-is-int*  $x \implies ?!(a::\text{int}). \text{real } a = x$   
**by** (*auto simp add: real-is-int-def*)

**lemma** *int-of-real-mult*:  
**assumes** *real-is-int*  $a$  *real-is-int*  $b$   
**shows** (*int-of-real* ( $a*b$ )) = (*int-of-real*  $a$ ) \* (*int-of-real*  $b$ )  
**proof** –  
**from** *prems* **have**  $a: ?!(a'::\text{int}). \text{real } a' = a$  **by** (*rule-tac real-is-int-rep, auto*)  
**from** *prems* **have**  $b: ?!(b'::\text{int}). \text{real } b' = b$  **by** (*rule-tac real-is-int-rep, auto*)  
**from**  $a$  **obtain**  $a'::\text{int}$  **where**  $a':a = \text{real } a'$  **by** *auto*  
**from**  $b$  **obtain**  $b'::\text{int}$  **where**  $b':b = \text{real } b'$  **by** *auto*  
**have**  $r: \text{real } a' * \text{real } b' = \text{real } (a' * b')$  **by** *auto*  
**show** *?thesis*  
**apply** (*simp add: a' b'*)  
**apply** (*subst r*)  
**apply** (*simp only: int-of-real-real*)  
**done**  
**qed**

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

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

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

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

**lemma** *real-is-int-number-of[simp]*: *real-is-int* ((*number-of*  $:: \text{int} \Rightarrow \text{real}$ )  $x$ )  
**proof** –  
**have** *neg1*: *real-is-int* ( $-1::\text{real}$ )  
**proof** –  
**have** *real-is-int* ( $-1::\text{real}$ ) = *real-is-int*(*real* ( $-1::\text{int}$ )) **by** *auto*



```

    also have ... = True by (simp only: real-is-int-real)
    ultimately show ?thesis by auto
qed

{
  fix x :: int
  have real-is-int ((number-of :: int  $\Rightarrow$  real) x)
    unfolding number-of-eq
    apply (induct x)
    apply (induct-tac n)
    apply (simp)
    apply (simp)
    apply (induct-tac n)
    apply (simp add: neg1)
  proof -
    fix n :: nat
    assume rn: (real-is-int (of-int (- (int (Suc n)))))
    have s: -(int (Suc (Suc n))) = -1 + - (int (Suc n)) by simp
    show real-is-int (of-int (- (int (Suc (Suc n)))))
      apply (simp only: s of-int-add)
      apply (rule real-is-int-add)
      apply (simp add: neg1)
      apply (simp only: rn)
    done
  qed
}
note Abs-Bin = this
{
  fix x :: int
  have ? u. x = u
    apply (rule exI[where x = x])
    apply (simp)
  done
}
then obtain u::int where x = u by auto
with Abs-Bin show ?thesis by auto
qed

lemma int-of-real-0[simp]: int-of-real (0::real) = (0::int)
by (simp add: int-of-real-def)

lemma int-of-real-1[simp]: int-of-real (1::real) = (1::int)
proof -
  have 1: (1::real) = real (1::int) by auto
  show ?thesis by (simp only: 1 int-of-real-real)
qed

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

```

```

have real-is-int (number-of b) by simp
then have uu: ?! u::int. number-of b = real u by (auto simp add: real-is-int-rep)
then obtain u::int where u: number-of b = real u by auto
have number-of b = real ((number-of b)::int)
  by (simp add: number-of-eq real-of-int-def)
have ub: number-of b = real ((number-of b)::int)
  by (simp add: number-of-eq real-of-int-def)
from uu u ub have unb: u = number-of b
  by blast
have int-of-real (number-of b) = u by (simp add: u)
with unb show ?thesis by simp
qed

```

```

lemma float-transfer-even: even a  $\implies$  float (a, b) = float (a div 2, b+1)
  apply (subst float-transfer[where a=a and b=b and c=-1, simplified])
  apply (simp-all add: pow2-def even-def real-is-int-def algebra-simps)
  apply (auto)
proof -
  fix q::int
  have a:b - (-1::int) = (1::int) + b by arith
  show (float (q, (b - (-1::int)))) = (float (q, ((1::int) + b)))
    by (simp add: a)
qed

```

```

lemma int-div-zdiv: int (a div b) = (int a) div (int b)
by (rule zdiv-int)

```

```

lemma int-mod-zmod: int (a mod b) = (int a) mod (int b)
by (rule zmod-int)

```

```

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

```

```

function norm-float :: int  $\Rightarrow$  int  $\Rightarrow$  int  $\times$  int where
  norm-float a b = (if a  $\neq$  0  $\wedge$  even a then norm-float (a div 2) (b + 1)
    else if a = 0 then (0, 0) else (a, b))
by auto

```

```

termination by (relation measure (nat o abs o fst))
(auto intro: abs-div-2-less)

```

```

lemma norm-float: float x = float (split norm-float x)
proof -
  {
    fix a b :: int
    have norm-float-pair: float (a, b) = float (norm-float a b)
    proof (induct a b rule: norm-float.induct)
      case (1 u v)
      show ?case

```

```

proof cases
  assume u:  $u \neq 0 \wedge \text{even } u$ 
  with prems have ind:  $\text{float } (u \text{ div } 2, v + 1) = \text{float } (\text{norm-float } (u \text{ div } 2)$ 
( $v + 1$ )) by auto
  with u have  $\text{float } (u, v) = \text{float } (u \text{ div } 2, v + 1)$  by (simp add: float-transfer-even)
  then show ?thesis
    apply (subst norm-float.simps)
    apply (simp add: ind)
    done
  next
    assume  $\sim(u \neq 0 \wedge \text{even } u)$ 
    then show ?thesis
      by (simp add: prems float-def)
    qed
  qed
}
note helper = this
have  $? a \ b. x = (a, b)$  by auto
then obtain a b where  $x = (a, b)$  by blast
then show ?thesis by (simp add: helper)
qed

```

```

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

```

```

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

```

```

lemma float-add:
   $\text{float } (a1, e1) + \text{float } (a2, e2) =$ 
  (if  $e1 \leq e2$  then  $\text{float } (a1 + a2 * 2^{\text{nat}(e2 - e1)}, e1)$ 
  else  $\text{float } (a1 * 2^{\text{nat}(e1 - e2)} + a2, e2)$ )
  apply (simp add: float-def algebra-simps)
  apply (auto simp add: pow2-int[symmetric] pow2-add[symmetric])
  done

```

```

lemma float-add-assoc1:
   $(x + \text{float } (y1, e1)) + \text{float } (y2, e2) = (\text{float } (y1, e1) + \text{float } (y2, e2)) + x$ 
by simp

```

```

lemma float-add-assoc2:
   $(\text{float } (y1, e1) + x) + \text{float } (y2, e2) = (\text{float } (y1, e1) + \text{float } (y2, e2)) + x$ 
by simp

```

```

lemma float-add-assoc3:
   $\text{float } (y1, e1) + (x + \text{float } (y2, e2)) = (\text{float } (y1, e1) + \text{float } (y2, e2)) + x$ 
by simp

```

```

lemma float-add-assoc4:

```

```

float (y1, e1) + (float (y2, e2) + x) = (float (y1, e1) + float (y2, e2)) + x
by simp

lemma float-mult-l0: float (0, e) * x = float (0, 0)
by (simp add: float-def)

lemma float-mult-r0: x * float (0, e) = float (0, 0)
by (simp add: float-def)

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
by (simp add: lbound-def)

lemma ubound: x ≤ ubound x
by (simp add: ubound-def)

lemma float-mult:
  float (a1, e1) * float (a2, e2) =
  (float (a1 * a2, e1 + e2))
by (simp add: float-def pow2-add)

lemma float-minus:
  - (float (a, b)) = float (-a, b)
by (simp add: float-def)

lemma zero-less-pow2:
  0 < pow2 x
proof -
{
  fix y
  have 0 ≤ y ⇒ 0 < pow2 y
    by (induct y, induct-tac n, simp-all add: pow2-add)
}
note helper=this
show ?thesis
  apply (case-tac 0 ≤ x)
  apply (simp add: helper)
  apply (subst pow2-neg)
  apply (simp add: helper)
done

```

qed

**lemma** *zero-le-float*:

$(0 \leq \text{float } (a,b)) = (0 \leq a)$   
**apply** (auto simp add: float-def)  
**apply** (auto simp add: zero-le-mult-iff zero-less-pow2)  
**apply** (insert zero-less-pow2[of b])  
**apply** (simp-all)  
**done**

**lemma** *float-le-zero*:

$(\text{float } (a,b) \leq 0) = (a \leq 0)$   
**apply** (auto simp add: float-def)  
**apply** (auto simp add: mult-le-0-iff)  
**apply** (insert zero-less-pow2[of b])  
**apply** auto  
**done**

**lemma** *float-abs*:

$\text{abs } (\text{float } (a,b)) = (\text{if } 0 \leq a \text{ then } (\text{float } (a,b)) \text{ else } (\text{float } (-a,b)))$   
**apply** (auto simp add: abs-if)  
**apply** (simp-all add: zero-le-float[symmetric, of a b] float-minus)  
**done**

**lemma** *float-zero*:

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

**lemma** *float-pprt*:

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

**lemma** *pprt-lbound*:  $\text{pprt } (\text{lbound } x) = \text{float } (0, 0)$

**apply** (simp add: float-def)  
**apply** (rule prpt-eq-0)  
**apply** (simp add: lbound-def)  
**done**

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

**apply** (simp add: float-def)  
**apply** (rule nprrt-eq-0)  
**apply** (simp add: ubound-def)  
**done**

**lemma** *float-npr*:

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

**lemma** *norm-0-1*:  $(0::\text{number-ring}) = \text{Numeral0} \ \& \ (1::\text{number-ring}) = \text{Numeral1}$

```

    by auto

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

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

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

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

lemma int-pow-0:  $(a::int)^{(Numeral0)} = 1$ 
  by simp

lemma int-pow-1:  $(a::int)^{(Numeral1)} = a$ 
  by simp

lemma zero-eq-Numeral0-nring:  $(0::'a::number-ring) = Numeral0$ 
  by simp

lemma one-eq-Numeral1-nring:  $(1::'a::number-ring) = Numeral1$ 
  by simp

lemma zero-eq-Numeral0-nat:  $(0::nat) = Numeral0$ 
  by simp

lemma one-eq-Numeral1-nat:  $(1::nat) = Numeral1$ 
  by simp

lemma zpower-Pls:  $(z::int)^{Numeral0} = Numeral1$ 
  by simp

lemma zpower-Min:  $(z::int)^{((-1)::nat)} = Numeral1$ 
proof -
  have  $1::((-1)::nat) = 0$ 
  by simp
  show ?thesis by (simp add: 1)
qed

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

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

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

```

**by** *simp*

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

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

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

**lemmas** *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*:  
 $((\text{number-of } v)::\text{int}) = (\text{number-of } w) \iff \text{iszero } ((\text{number-of } (v + \text{uminus } w))::\text{int})$   
**by** (*rule eq-number-of-eq*)

**lemma** *int-iszero-number-of-Pls*:  $\text{iszero } (\text{Numeral0}::\text{int})$   
**by** (*simp only: iszero-number-of-Pls*)

**lemma** *int-nonzero-number-of-Min*:  $\sim(\text{iszero } ((-1)::\text{int}))$   
**by** *simp*

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

**lemma** *int-iszero-number-of-Bit1*:  $\neg \text{iszero } ((\text{number-of } (\text{Int.Bit1 } w))::\text{int})$   
**by** *simp*

**lemma** *int-less-number-of-eq-neg*:  $((\text{number-of } x)::\text{int}) < \text{number-of } y \iff \text{neg } ((\text{number-of } (x + (\text{uminus } y)))::\text{int})$   
**unfolding** *neg-def number-of-is-id* **by** *simp*

**lemma** *int-not-neg-number-of-Pls*:  $\neg (\text{neg } (\text{Numeral0}::\text{int}))$   
**by** *simp*

**lemma** *int-neg-number-of-Min*:  $\text{neg } (-1::\text{int})$   
**by** *simp*

**lemma** *int-neg-number-of-Bit0*:  $\text{neg } ((\text{number-of } (\text{Int.Bit0 } w))::\text{int}) = \text{neg } ((\text{number-of } w)::\text{int})$   
**by** *simp*

**lemma** *int-neg-number-of-Bit1*:  $\text{neg } ((\text{number-of } (\text{Int.Bit1 } w))::\text{int}) = \text{neg } ((\text{number-of } w)::\text{int})$   
**by** *simp*

**lemma** *int-le-number-of-eq*:  $((\text{number-of } x)::\text{int}) \leq \text{number-of } y = (\neg \text{neg } ((\text{number-of } (y + (\text{uminus } x)))::\text{int}))$

**unfolding** *neg-def number-of-is-id* **by** (*simp add: not-less*)

**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*:  $((\text{number-of } v)::\text{int}) + \text{number-of } w = \text{number-of } (v + w)$

**by** *simp*

**lemma** *int-number-of-diff-sym*:  $((\text{number-of } v)::\text{int}) - \text{number-of } w = \text{number-of } (v + (\text{uminus } w))$

**by** *simp*

**lemma** *int-number-of-mult-sym*:  $((\text{number-of } v)::\text{int}) * \text{number-of } w = \text{number-of } (v * w)$

**by** *simp*

**lemma** *int-number-of-minus-sym*:  $-(\text{number-of } v)::\text{int} = \text{number-of } (\text{uminus } v)$

**by** *simp*

**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 pprrt-lbound nprrt-ubound*

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

**use** *~~/src/HOL/Tools/float-arith.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:  $\text{Trueprop } X == (X == \text{True})$  by (simp add: atomize-eq)
lemma meta-eq-trivial:  $x == y \implies x == y$  by simp
lemma meta-eq-imp-eq:  $x == y \implies x = y$  by auto
lemma eq-trivial:  $x = y \implies x == y$  by auto
lemma bool-to-true:  $x :: \text{bool} \implies x == \text{True}$  by simp
lemma transmeta-1:  $x = y \implies y == z \implies x = z$  by simp
lemma transmeta-2:  $x == y \implies y = z \implies x = z$  by simp
lemma transmeta-3:  $x == y \implies y == z \implies x = z$  by simp


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


lemmas compute-if = If-True If-False


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

```

**lemma** *bool19*:  $(True = False) = False$  **by** *blast*

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

**lemma** *compute-fst*:  $\text{fst } (x,y) = x$  **by** *simp*

**lemma** *compute-snd*:  $\text{snd } (x,y) = y$  **by** *simp*

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

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

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

**lemma** *compute-the*:  $\text{the } (Some \ x) = x$  **by** *simp*

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

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

**lemma** *compute-None-None-eq*:  $(None = None) = True$  **by** *auto*

**lemma** *compute-Some-Some-eq*:  $(Some \ x = Some \ y) = (x = y)$  **by** *auto*

**definition**

*option-case-compute* ::  $'b \ \text{option} \Rightarrow 'a \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$

**where**

*option-case-compute* *opt a f* = *option-case a f opt*

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

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

**lemma** *option-case-compute-None*: *option-case-compute* *None* =  $(\lambda \ a \ f. \ a)$

**apply** (*rule ext*)+

**apply** (*simp add: option-case-compute-def*)

**done**

**lemma** *option-case-compute-Some*: *option-case-compute*  $(Some \ x) = (\lambda \ a \ f. \ f \ x)$

**apply** (*rule ext*)+

**apply** (*simp add: option-case-compute-def*)

**done**

**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*:  $\text{length } (x \# xs) = 1 + (\text{length } xs)$   
**by** *simp*

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

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

**definition**

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

**where**

*list-case-compute* l a f = *list-case* a f l

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

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

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

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

**lemma** *compute-list-nth*:  $((x \# xs) ! n) = (\text{if } n = 0 \text{ then } x \text{ else } (xs ! (n - 1)))$   
**by** (*cases n, auto*)

**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**  $\langle\langle$   
*signature* *ComputeHOL* =  
*sig*  
  *val* *prep-thms* : *thm list*  $\rightarrow$  *thm list*  
  *val* *to-meta-eq* : *thm*  $\rightarrow$  *thm*  
  *val* *to-hol-eq* : *thm*  $\rightarrow$  *thm*  
  *val* *symmetric* : *thm*  $\rightarrow$  *thm*  
  *val* *trans* : *thm*  $\rightarrow$  *thm*  $\rightarrow$  *thm*  
*end*  
  
*structure* *ComputeHOL* : *ComputeHOL* =  
*struct*  
  
  *local*  
  *fun* *lhs-of eq* = *fst* (*Thm.dest-equals* (*cprop-of eq*));  
  *in*  
  *fun* *rewrite-conv* [] *ct* = *raise CTERM* (*rewrite-conv*, [])  
  | *rewrite-conv* (*eq :: eqs*) *ct* =  
    *Thm.instantiate* (*Thm.match* (*lhs-of eq*, *ct*)) *eq*  
    *handle* *Pattern.MATCH*  $\Rightarrow$  *rewrite-conv eqs ct*;  
  *end*  
  
  *val* *convert-conditions* = *Conv.fconv-rule* (*Conv.premis-conv*  $\sim 1$  (*Conv.try-conv*  
  (*rewrite-conv* [{} *thm Trueprop-eq-eq*]))))  
  
  *val* *eq-th* = @{*thm HOL.eq-reflection*}  
  *val* *meta-eq-trivial* = @{*thm ComputeHOL.meta-eq-trivial*}  
  *val* *bool-to-true* = @{*thm ComputeHOL.bool-to-true*}  
  
  *fun* *to-meta-eq th* = *eq-th* *OF* [*th*] *handle* *THM* -  $\Rightarrow$  *meta-eq-trivial* *OF* [*th*] *handle*  
  *THM* -  $\Rightarrow$  *bool-to-true* *OF* [*th*]  
  
  *fun* *to-hol-eq th* = @{*thm meta-eq-imp-eq*} *OF* [*th*] *handle* *THM* -  $\Rightarrow$  @{*thm*  
  *eq-trivial*} *OF* [*th*]  
  
  *fun* *prep-thms ths* = *map* (*convert-conditions* *o* *to-meta-eq*) *ths*  
  
  *fun* *symmetric th* = @{*thm HOL.sym*} *OF* [*th*] *handle* *THM* -  $\Rightarrow$  @{*thm Pure.symmetric*}  
  *OF* [*th*]  
  
  *local*  
  *val* *trans-HOL* = @{*thm HOL.trans*}

```

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

end
>>

end

theory ComputeNumeral
imports ComputeHOL ComputeFloat
begin

lemmas bitnorm = normalize-bin-simps

lemma neg1: neg Int.Pls = False by (simp add: Int.Pls-def)
lemma neg2: neg Int.Min = True apply (subst Int.Min-def) by auto
lemma neg3: neg (Int.Bit0 x) = neg x apply (simp add: neg-def) apply (subst
  Bit0-def) by auto
lemma neg4: neg (Int.Bit1 x) = neg x apply (simp add: neg-def) apply (subst
  Bit1-def) by auto
lemmas bitneg = neg1 neg2 neg3 neg4

lemma iszero1: iszero Int.Pls = True by (simp add: Int.Pls-def iszero-def)
lemma iszero2: iszero Int.Min = False apply (subst Int.Min-def) apply (subst
  iszero-def) by simp
lemma iszero3: iszero (Int.Bit0 x) = iszero x apply (subst Int.Bit0-def) apply
  (subst iszero-def)+ by auto
lemma iszero4: iszero (Int.Bit1 x) = False apply (subst Int.Bit1-def) apply
  (subst iszero-def)+ apply simp by arith
lemmas bitiszero = iszero1 iszero2 iszero3 iszero4

constdefs
  lezero x == (x ≤ 0)
lemma lezero1: lezero Int.Pls = True unfolding Int.Pls-def lezero-def by auto
lemma lezero2: lezero Int.Min = True unfolding Int.Min-def lezero-def by auto
lemma lezero3: lezero (Int.Bit0 x) = lezero x unfolding Int.Bit0-def lezero-def
  by auto
lemma lezero4: lezero (Int.Bit1 x) = neg x unfolding Int.Bit1-def lezero-def

```

```

neg-def by auto
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:  $x * \text{Int.Pl} = \text{Int.Pl}$  by (simp add: Pls-def)
lemma mult-Min-right:  $x * \text{Int.Min} = - x$  by (subst mult-commute, simp add:
mult-Min)
lemma multb0x:  $(\text{Int.Bit0 } x) * y = \text{Int.Bit0 } (x * y)$  by (rule mult-Bit0)
lemma multxb0:  $x * (\text{Int.Bit0 } y) = \text{Int.Bit0 } (x * y)$  unfolding Bit0-def by simp
lemma multb1:  $(\text{Int.Bit1 } x) * (\text{Int.Bit1 } y) = \text{Int.Bit1 } (\text{Int.Bit0 } (x * y) + x + y)$ 
  unfolding Bit0-def Bit1-def by (simp add: algebra-simps)
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)
  apply (simp add: nat-norm-number-of-def)
  unfolding lezero-def iszero-def neg-def
  apply (simp add: numeral-simps)
  done

```

```

lemma natnorm0: (0::nat) = number-of (Int.Pls) by auto
lemma natnorm1: (1 :: nat) = number-of (Int.Bit1 Int.Pls) by auto
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))
by (auto simp add: number-of-is-id)

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))))
  unfolding nat-number-of-def number-of-is-id neg-def
  by auto

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))))))
  unfolding nat-norm-number-of
  by (auto simp add: number-of-is-id neg-def lezero-def iszero-def Let-def nat-number-of-def)

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)))
  unfolding nat-number-of-def number-of-is-id neg-def
  by (simp add: nat-mult-distrib)

lemma nateq: (((number-of x)::nat) = (number-of y)) = ((lezero x ∧ lezero y) ∨
(x = y))
  by (auto simp add: iszero-def lezero-def neg-def number-of-is-id)

lemma natless: (((number-of x)::nat) < (number-of y)) = ((x < y) ∧ (¬ (lezero
y)))
  by (simp add: lezero-def numeral-simps not-le)

lemma natle: (((number-of x)::nat) ≤ (number-of y)) = (y < x → lezero x)
  by (auto simp add: number-of-is-id lezero-def nat-number-of-def)

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)
  unfolding number-of-eq
  apply simp

```

**done**

**lemma** *number-le*:  $((\text{number-of } x)::'a::\{\text{number-ring}, \text{ordered-idom}\}) \leq (\text{number-of } y) = (x \leq y)$   
**unfolding** *number-of-eq*  
**apply** *simp*  
**done**

**lemma** *number-less*:  $((\text{number-of } x)::'a::\{\text{number-ring}, \text{ordered-idom}\}) < (\text{number-of } y) = (x < y)$   
**unfolding** *number-of-eq*  
**apply** *simp*  
**done**

**lemma** *number-diff*:  $((\text{number-of } x)::'a::\{\text{number-ring}, \text{ordered-idom}\}) - \text{number-of } y = \text{number-of } (x + (-y))$   
**apply** (*subst diff-number-of-eq*)  
**apply** *simp*  
**done**

**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*:  $\text{real } ((\text{number-of } v)::\text{nat}) = (\text{if } \text{neg } v \text{ then } 0 \text{ else } \text{number-of } v)$   
**by** (*simp only: real-of-nat-number-of number-of-is-id*)

**lemma** *compute-nat-of-int-number-of*:  $\text{nat } ((\text{number-of } v)::\text{int}) = (\text{number-of } v)$   
**by** *simp*

**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*:  $\text{adjust } b \ (q, r) = (\text{if } 0 \leq r - b \text{ then } (2 * q + 1, r - b) \text{ else } (2 * q, r))$   
**by** (*auto simp only: adjust-def*)

**lemma** *negateSnd*:  $\text{negateSnd } (q, r) = (q, -r)$   
**by** (*simp add: negateSnd-def*)

**lemma** *divmod*:  $\text{IntDiv.divmod } a \ b = (\text{if } 0 \leq a \text{ then}$



```

      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)))
  by (auto simp only: IntDiv.divmod-def)

```

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

**lemma** *even-Pls: even (Int.Pls) = True*  
**apply** (*unfold Pls-def even-def*)  
**by** *simp*

**lemma** *even-Min: even (Int.Min) = False*  
**apply** (*unfold Min-def even-def*)  
**by** *simp*

**lemma** *even-B0: even (Int.Bit0 x) = True*  
**apply** (*unfold Bit0-def*)  
**by** *simp*

**lemma** *even-B1: even (Int.Bit1 x) = False*  
**apply** (*unfold Bit1-def*)  
**by** *simp*

**lemma** *even-number-of: even ((number-of w)::int) = even w*  
**by** (*simp only: number-of-is-id*)

**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*  $\sim\sim$  */src/HOL/Tools/ComputeFloat*  $\sim\sim$  */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 ≤ 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) (nprrt-spmat r2),
      add-spmat (mult-spmat (nprrt-spmat s2) (pprt-spmat r1), mult-spmat (nprrt-spmat
s1) (nprrt-spmat r1)))))))))
apply (simp add: Let-def)
apply (insert assms)
apply (simp add: sparse-row-matrix-op-simps algebra-simps)
apply (rule mult-le-dual-prts[where A=A, simplified Let-def algebra-simps])
apply (auto)
done

```

**lemma** *spm-mult-le-dual-prts-no-let:*

```

assumes
  sorted-sparse-matrix A1
  sorted-sparse-matrix A2
  sorted-sparse-matrix c1
  sorted-sparse-matrix c2
  sorted-sparse-matrix y
  sorted-sparse-matrix r1
  sorted-sparse-matrix r2
  sorted-spvec b
  le-spmat ([], y)
  sparse-row-matrix A1 ≤ A
  A ≤ sparse-row-matrix A2
  sparse-row-matrix c1 ≤ c
  c ≤ sparse-row-matrix c2

```

```

sparse-row-matrix r1 ≤ x
x ≤ sparse-row-matrix r2
A * x ≤ sparse-row-matrix (b::('a::lordered-ring) spmat)
shows
c * x ≤ sparse-row-matrix (add-spmat (mult-spmat y b,
mult-est-spmat r1 r2 (diff-spmat c1 (mult-spmat y A2)) (diff-spmat c2 (mult-spmat
y A1))))
by (simp add: assms mult-est-spmat-def spm-mult-le-dual-prts[where A=A, simplified Let-def])

use matrixlp.ML

end

```