Parallel Shear Sort

Manuel Eberl and Peter Lammich

September 30, 2024

Abstract

This entry provides a formalisation of *parallel shear sort*, a comparison-based sorting algorithm intended for highly parallel systems. It sorts n elements in $O(\log n)$ steps, each of which involves sorting $\sqrt{n}$ independent lists of $\sqrt{n}$ elements each.

If these smaller sort operations are done in parallel with a conventional $O(n \log n)$ sorting algorithm, this leads to an overall work of $O(n \log^2(n))$ and a span of $O(\sqrt{n} \log^2(n))$ – a considerable improvement over conventional non-parallel sorting.

Contents

1 Parallel shear sort

1.1 Auxiliary material

1.1.1 The $\lceil \log_2 n \rceil$ function

```
theory Ceil_Log2
  imports Complex_Main
begin

definition ceillog2 :: "nat ⇒ nat" where
  "ceillog2 n = (if n = 0 then 0 else nat ⌈log 2 (real n)⌉)"

lemma ceillog2_le_1 [simp]: "n ≤ 1 ⟹ ceillog2 n = 0"
  and ceillog2_2 [simp]: "ceillog2 2 = 1"
  by (auto simp: ceillog2_def)

lemma ceillog2_2_power [simp]: "ceillog2 (2 ^ n) = n"
  by (auto simp: ceillog2_def)

lemma ceillog2_ge_log:
  assumes "n > 0"
  shows   "real (ceillog2 n) ≥ log 2 (real n)"
proof -
  have "real_of_int ⌈log 2 (real n)⌉ ≥ log 2 (real n)"
    by linarith
  thus ?thesis
    using assms unfolding ceillog2_def by auto
qed

lemma ceillog2_less_log:
  assumes "n > 0"
  shows   "real (ceillog2 n) < log 2 (real n) + 1"
proof -
  have "real_of_int ⌈log 2 (real n)⌉ < log 2 (real n) + 1"
    by linarith
  thus ?thesis
    using assms unfolding ceillog2_def by auto
qed

lemma ceillog2_le_iff:
  assumes "n > 0"
  shows   "ceillog2 n ≤ l ⟷ n ≤ 2 ^ l"
proof -
  have "ceillog2 n ≤ l ⟷ real n ≤ 2 ^ l"
    unfolding ceillog2_def using assms by (auto simp: log_le_iff powr_realpow)
  also have "2 ^ l = real (2 ^ l)"
    by simp
  also have "real n ≤ real (2 ^ l) ⟷ n ≤ 2 ^ l"
    by linarith
```

```
  finally show ?thesis .
qed

lemma ceillog2_ge_iff:
  assumes "n > 0"
  shows   "ceillog2 n ≥ l ⟷ 2 ^ l < 2 * n"
proof -
  have "-1 < (0 :: real)"
    by auto
  also have "... ≤ log 2 (real n)"
    using assms by auto
  finally have "ceillog2 n ≥ l ⟷ real l - 1 < log 2 (real n)"
    unfolding ceillog2_def using assms by (auto simp: le_nat_iff le_ceiling_iff)
  also have "... ⟷ real l < log 2 (real (2 * n))"
    using assms by (auto simp: log_mult)
  also have "... ⟷ 2 ^ l < real (2 * n)"
    using assms by (subst less_log_iff) (auto simp: powr_realpow)
  also have "2 ^ l = real (2 ^ l)"
    by simp
  also have "real (2 ^ l) < real (2 * n) ⟷ 2 ^ l < 2 * n"
    by linarith
  finally show ?thesis .
qed

lemma le_two_power_ceillog2: "n ≤ 2 ^ ceillog2 n"
proof (cases "n = 0")
  case False
  thus ?thesis
    using ceillog2_le_iff[of n "ceillog2 n"] by simp
qed auto

lemma two_power_ceillog2_gt:
  assumes "n > 0"
  shows   "2 * n > 2 ^ ceillog2 n"
  using ceillog2_ge_iff[of n "ceillog2 n"] assms by simp

lemma ceillog2_eqI:
  assumes "n ≤ 2 ^ l" and "2 ^ l < 2 * n"
  shows   "ceillog2 n = l"
proof -
  from assms have "n > 0"
    by (intro Nat.gr0I) auto
  thus ?thesis using assms
    by (intro antisym[of _ l])
       (auto simp: ceillog2_le_iff ceillog2_ge_iff)
qed

lemma ceillog2_mono:
  assumes "m ≤ n"
```

```
  shows   "ceillog2 m ≤ ceillog2 n"
proof (cases "m = 0")
  case False
  have "⌈log 2 (real m)⌉ ≤ ⌈log 2 (real n)⌉"
    by (intro ceiling_mono) (use False assms in auto)
  hence "nat ⌈log 2 (real m)⌉ ≤ nat ⌈log 2 (real n)⌉"
    by linarith
  thus ?thesis using False assms
    unfolding ceillog2_def by simp
qed auto


lemma ceillog2_rec:
  assumes "n > 1"
  shows   "ceillog2 n = Suc (ceillog2 ((n + 1) div 2))"
proof -
  from assms have "log 2 n > 0"
    by force
  have "Suc (ceillog2 ((n + 1) div 2)) = nat (⌈log 2 (real ((n - 1) div
2 + 1))⌉) + 1"
    unfolding ceillog2_def using assms by (cases n) auto
  also have "... = nat (⌈log 2 (real ((n - 1) div 2 + 1))⌉ + 1)"
  proof (subst nat_add_distrib)
    have "log 2 (real ((n - 1) div 2 + 1)) ≥ 0"
      by force
    thus "⌈log 2 (real ((n - 1) div 2 + 1))⌉ ≥ 0"
      by linarith
  qed auto
  also have "... = ceillog2 n"
    unfolding ceillog2_def using assms by (subst (2) ceiling_log2_div2)
auto
  finally show ?thesis ..
qed

lemma ceillog2_rec_even:
  assumes "k > 0"
  shows   "ceillog2 (2 * k) = Suc (ceillog2 k)"
  by (rule ceillog2_eqI) (auto simp: le_two_power_ceillog2 two_power_ceillog2_gt
assms)

lemma ceillog2_rec_odd:
  assumes "k > 0"
  shows   "ceillog2 (Suc (2 * k)) = Suc (ceillog2 (Suc k))"
  using assms by (subst ceillog2_rec) auto

lemma funpow_div2_ceillog2_le_1:
  "((λn. (n + 1) div 2) ^^ ceillog2 n) n ≤ 1"
proof (induction n rule: less_induct)
  case (less n)
```

```
  show ?case
  proof (cases "n ≤ 1")
    case True
    thus ?thesis by auto
  next
    case False
    have "((λn. (n + 1) div 2) ^^ Suc (ceillog2 ((n + 1) div 2))) n ≤
1"
      using less.IH[of "(n+1) div 2"] False by (subst funpow_Suc_right)
auto
    also have "Suc (ceillog2 ((n + 1) div 2)) = ceillog2 n"
      using False by (subst ceillog2_rec[of n]) auto
    finally show ?thesis .
  qed
qed


fun ceillog2_aux :: "nat ⇒ nat ⇒ nat" where
  "ceillog2_aux acc n = (if n ≤ 1 then acc else ceillog2_aux (acc + 1)
((n + 1) div 2))"

lemmas [simp del] = ceillog2_aux.simps

lemma ceillog2_aux_correct: "ceillog2_aux acc n = ceillog2 n + acc"
proof (induction acc n rule: ceillog2_aux.induct)
  case (1 acc n)
  show ?case
  proof (cases "n ≤ 1")
    case False
    thus ?thesis using ceillog2_rec[of n] "1.IH"
      by (auto simp: ceillog2_aux.simps[of acc n])
  qed (auto simp: ceillog2_aux.simps[of acc n])
qed

lemma ceillog2_code [code]: "ceillog2 n = ceillog2_aux 0 n"
  by (simp add: ceillog2_aux_correct)


end

theory Parallel_Shear_Sort
  imports Complex_Main "HOL-Library.Multiset" "HOL-Library.FuncSet" Ceil_Log2
begin
```

1.1.2 Facts about multisets

```
lemma mset_concat: "mset (concat xss) = (∑ xs←xss. mset xs)"
  by (induction xss) auto
```

lemma *sum_mset_singleton_mset [simp]: "(∑x∈#A. {#f x#}) = image_mset f A"*
by *(induction A) auto*

lemma *sum_list_singleton_mset [simp]: "(∑x←xs. {#f x#}) = image_mset f (mset xs)"*
by *(induction xs) auto*

lemma *count_conv_size_mset: "count A x = size (filter_mset (λy. y = x) A)"*
by *(induction A) auto*

lemma *size_conv_count_bool_mset: "size A = count A True + count A False"*
by *(induction A) auto*

lemma *set_mset_sum: "finite A ⟹ set_mset (∑x∈A. f x) = (⋃x∈A. set_mset (f x))"*
by *(induction A rule: finite_induct) auto*

lemma *filter_image_mset:*
"filter_mset P (image_mset f A) = image_mset f (filter_mset (λx. P (f x)) A)"
by *(induction A) auto*

1.1.3 Facts about sorting

lemma *sort_replicate [simp]: "sort (replicate n x) = replicate n x"*
by *(intro properties_for_sort) auto*

lemma *mset_replicate [simp]: "mset (replicate n x) = replicate_mset n x"*
by *(induction n) auto*

lemma *sorted_wrt_induct [consumes 1, case_names Nil Cons]:*
assumes *"sorted_wrt R xs"*
assumes *"P []"*
"⋀x xs. (⋀y. y ∈ set xs ⟹ R x y) ⟹ P xs ⟹ P (x # xs)"
shows *"P xs"*
using *assms(1)* **by** *(induction xs) (auto intro: assms)*

lemma *sorted_wrt_trans_induct [consumes 2, case_names Nil single Cons]:*
assumes *"sorted_wrt R xs" "transp R"*
assumes *"P []" "⋀x. P [x]"*
"⋀x y xs. R x y ⟹ P (y # xs) ⟹ P (x # y # xs)"
shows *"P xs"*
using *assms(1)*
by *(induction xs rule: induct_list012)*
(auto intro: assms simp: sorted_wrt2[OF assms(2)])

```
lemmas sorted_induct [consumes 1, case_names Nil single Cons] =
  sorted_wrt_trans_induct[OF _ preorder_class.transp_on_le]

lemma sorted_wrt_map_mono:
  assumes "sorted_wrt R xs"
  assumes "⋀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ R x y ⟹ R' (f x) (f
y)"
  shows   "sorted_wrt R' (map f xs)"
  using assms by (induction rule: sorted_wrt_induct) auto

lemma sorted_map_mono:
  assumes "sorted xs" and "mono_on (set xs) f"
  shows   "sorted (map f xs)"
  using assms(1)
  by (rule sorted_wrt_map_mono) (use assms in ‹auto simp: mono_on_def›)

lemma sort_map_mono: "mono f ⟹ sort (map f xs) = map f (sort xs)"
  by (intro properties_for_sort sorted_map_mono)
     (use mono_on_subset in auto)

lemma sorted_boolE:
  assumes "sorted xs" "length xs = w"
  shows   "∃k≤w. xs = replicate k False @ replicate (w - k) True"
proof -
  define k where "k = length (filter (λb. ¬b) xs)"
  have "mset xs = replicate_mset k False + replicate_mset (w - k) True"
    unfolding k_def assms(2)[symmetric] by (induction xs) (auto simp:
Suc_diff_le)
  moreover have "sorted (replicate k False @ replicate (w - k) True)"
    by (auto simp: sorted_append)
  ultimately have "sort xs = replicate k False @ replicate (w - k) True"
    by (intro properties_for_sort) auto
  moreover have "k ≤ w"
    using assms by (auto simp: k_def)
  ultimately show ?thesis
    using assms(1) by (intro exI[of _ k]) (simp_all add: sorted_sort_id)
qed

lemma rev_sorted_boolE:
  assumes "sorted (rev xs)" "length xs = w"
  shows   "∃k≤w. xs = replicate k True @ replicate (w - k) False"
proof -
  from sorted_boolE[OF assms(1)] assms(2) obtain k
    where k: "k ≤ w" "rev xs = replicate k False @ replicate (w - k)
True" by auto
  note k(2)
  also have "rev (replicate k False @ replicate (w - k) True) =
             replicate (w - k) True @ replicate (w - (w - k)) False"
    using k(1) by auto
```

```
  finally show ?thesis
    by (intro exI[of _ "w - k"]) auto
qed

lemma sort_append:
  assumes "⋀x y. x ∈ set xs ⟹ y ∈ set ys ⟹ x ≤ y"
  shows   "sort (xs @ ys) = sort xs @ sort ys"
  using assms by (intro properties_for_sort) (auto simp: sorted_append)

lemma sort_append_replicate_left:
  "(⋀y. y ∈ set xs ⟹ x ≤ y) ⟹ sort (replicate n x @ xs) = replicate
n x @ sort xs"
  by (subst sort_append) auto

lemma sort_append_replicate_right:
  "(⋀y. y ∈ set xs ⟹ x ≥ y) ⟹ sort (xs @ replicate n x) = sort xs
@ replicate n x"
  by (subst sort_append) auto
```

1.1.4 Miscellaneous

```
lemma nth_append_left: "i < length xs ⟹ (xs @ ys) ! i = xs ! i"
  by (auto simp: nth_append)

lemma nth_append_right: "i ≥ length xs ⟹ (xs @ ys) ! i = ys ! (i -
length xs)"
  by (auto simp: nth_append)

lemma Times_insert_right: "A × insert y B = (λx. (x, y)) ` A ∪ A ×
B"
  by auto

lemma Times_insert_left: "insert x A × B = (λy. (x, y)) ` B ∪ A × B"
  by auto

lemma map_nth_shift:
  assumes "length xs = b - a"
  shows   "map (λj. xs ! (j - a)) [a..<b] = xs"
proof -
  have "map (λj. xs ! (j - a)) [a..<b] = map (λj. xs ! j) (map (λj. j
- a) [a..<b])"
    unfolding map_map o_def ..
  also have "map (λj. j - a) [a..<b] = [0..<b - a]"
  proof (cases "a ≤ b")
    case True
    thus ?thesis
      by (induction rule: dec_induct) (auto simp: Suc_diff_le)
  qed auto
  also have "map (λj. xs ! j) ... = xs"
```

```
    by (metis assms map_nth)
  finally show ?thesis .
qed
```

1.2 Auxiliary definitions

The following predicate states that all elements of a list are equal to one another.

```
definition all_same :: "'a list ⇒ bool"
  where "all_same xs = (∃x. set xs ⊆ {x})"

lemma all_same_replicate [intro]: "all_same (replicate n x)"
  unfolding all_same_def by auto

lemma all_same_altdef: "all_same xs ⟷ xs = replicate (length xs) (hd
xs)"
proof
  assume *: "xs = replicate (length xs) (hd xs)"
  show "all_same xs"
    by (subst *) auto
next
  assume "all_same xs"
  thus   "xs = replicate (length xs) (hd xs)"
    by (cases xs) (auto simp: all_same_def intro!: replicate_eqI)
qed

lemma all_sameE:
  assumes "all_same xs"
  obtains n x where "xs = replicate n x"
  using assms that unfolding all_same_altdef by metis
```

The following predicate states that a list is sorted in ascending or descending order, depending on the boolean flag.

```
definition sorted_asc_desc :: "bool ⇒ 'a :: linorder list ⇒ bool"
  where "sorted_asc_desc asc xs = (if asc then sorted xs else sorted
(rev xs))"
```

Analogously, we define a sorting function that takes such a flag.

```
definition sort_asc_desc :: "bool ⇒ 'a :: linorder list ⇒ 'a list"
  where "sort_asc_desc asc xs = (if asc then sort xs else rev (sort xs))"

lemma length_sort_asc_desc [simp]: "length (sort_asc_desc asc xs) = length
xs"
  by (auto simp: sort_asc_desc_def)

lemma mset_sort_asc_desc [simp]: "mset (sort_asc_desc asc xs) = mset
xs"
  by (auto simp: sort_asc_desc_def)
```

```
lemma sort_asc_desc_map_mono: "mono f ⟹ sort_asc_desc b (map f xs)
= map f (sort_asc_desc b xs)"
  by (auto simp: sort_asc_desc_def sort_map_mono rev_map)

lemma sort_asc_desc_all_same: "all_same xs ⟹ sort_asc_desc asc xs
= xs"
  by (auto simp: sort_asc_desc_def elim!: all_sameE)
```

1.3 Matrices

We represent matrices as functions mapping index pairs to elements. The first index is the row, the second the column. For convenience, we also fix explicit lower and upper bounds for the indices so that we can easily talk about minors of a matrix (or submatrices). The lower bound is inclusive, the upper bound exclusive.

```
type_synonym 'a mat = "nat × nat ⇒ 'a"

locale shearsort =
  fixes lrow urow lcol ucol :: nat and dummy :: "'a :: linorder"
  assumes lrow_le_urow: "lrow ≤ urow"
  assumes lcol_le_ucol: "lcol ≤ ucol"
begin
```

The set of valid indices:

```
definition idxs :: "(nat × nat) set" where "idxs = {lrow..<urow} × {lcol..<ucol}"
```

The multiset of all entries in the matrix:

```
definition mset_mat :: "(nat × nat ⇒ 'b) ⇒ 'b multiset"
  where "mset_mat m = image_mset m (mset_set idxs)"
```

The i-th row and j-th column of a matrix:

```
definition row :: "(nat × nat ⇒ 'b) ⇒ nat ⇒ 'b list"
  where "row m i = map (λj. m (i, j)) [lcol..<ucol]"
definition col :: "(nat × nat ⇒ 'b) ⇒ nat ⇒ 'b list"
  where "col m j = map (λi. m (i, j)) [lrow..<urow]"

lemma length_row [simp]: "length (row m i) = ucol - lcol"
  and length_col [simp]: "length (col m i) = urow - lrow"
  by (simp_all add: row_def col_def)

lemma nth_row [simp]: "j < ucol - lcol ⟹ row m i ! j = m (i, lcol
+ j)"
  unfolding row_def by (subst nth_map) auto

lemma set_row: "set (row m i) = (λj. m (i, j)) ` {lcol..<ucol}"
  unfolding row_def by simp
```

```
lemma set_col: "set (col m j) = (λi. m (i, j)) ‘ {lrow..<urow}"
  unfolding col_def by simp

lemma mset_row: "mset (row m i) = image_mset (λj. m (i, j)) (mset [lcol..<ucol])"
  unfolding row_def by simp

lemma mset_col: "mset (col m j) = image_mset (λi. m (i, j)) (mset [lrow..<urow])"
  unfolding col_def by simp

lemma nth_col [simp]: "i < urow - lrow ⟹ col m j ! i = m (lrow + i,
j)"
  unfolding col_def by (subst nth_map) auto
```

The following helps us to restrict a matrix operation to the valid indices. Here, `m` is the original matrix and `m'` the changed matrix that we obtained after applying some operation on it.

```
definition restrict_mat :: "'a mat ⇒ 'a mat ⇒ 'a mat" where
  "restrict_mat m m' = (λij. if ij ∈ idxs then m' ij else m ij)"

lemma row_restrict_mat [simp]:
  "row (restrict_mat m m') i = (if i ∈ {lrow..<urow} then row m' i else
row m i)"
  by (auto simp: restrict_mat_def idxs_def row_def)

lemma col_restrict_mat [simp]:
  "col (restrict_mat m m') j = (if j ∈ {lcol..<ucol} then col m' j else
col m j)"
  by (auto simp: restrict_mat_def idxs_def col_def)
```

The following lemmas allow us to prove that two matrices are equal by showing that their rows (or columns) are the same.

```
lemma matrix_eqI_rows:
  assumes "⋀i. i ∈ {lrow..<urow} ⟹ row m1 i = row m2 i"
  assumes "⋀i j. (i, j) ∉ idxs ⟹ m1 (i, j) = m2 (i, j)"
  shows   "m1 = m2"
  using assms by (auto simp: fun_eq_iff row_def idxs_def atLeastLessThan_def)

lemma matrix_eqI_cols:
  assumes "⋀j. j ∈ {lcol..<ucol} ⟹ col m1 j = col m2 j"
  assumes "⋀i j. (i, j) ∉ idxs ⟹ m1 (i, j) = m2 (i, j)"
  shows   "m1 = m2"
  using assms by (auto simp: fun_eq_iff col_def idxs_def atLeastLessThan_def)
```

The following lemmas express the multiset of elements as a sum of rows (or columns):

```
lemma mset_mat_conv_sum_rows: "mset_mat m = (∑ i∈{lrow..<urow}. mset
(row m i))"
```

```
  using lrow_le_urow unfolding mset_mat_def idxs_def
proof (induction rule: dec_induct)
  case (step n)
  have "image_mset m (mset_set ({lrow..<Suc n} × {lcol..<ucol})) =
        image_mset m (mset_set ((λj. (n, j)) ‘ {lcol..<ucol} ∪ {lrow..<n}
× {lcol..<ucol}))"
    using step.prems step.hyps by (auto simp: Times_insert_left atLeastLessThanSuc)
  also have "... = image_mset m (mset_set ((λj. (n, j)) ‘ {lcol..<ucol}))
+
                  image_mset m (mset_set ({lrow..<n} × {lcol..<ucol}))"
    by (subst mset_set_Union) (auto simp flip: image_mset_mset_set)
  also have "... = image_mset m (image_mset (λj. (n, j)) (mset_set {lcol..<ucol}))
+
                  image_mset m (mset_set ({lrow..<n} × {lcol..<ucol}))"
    by (subst image_mset_mset_set [symmetric]) (auto simp: inj_on_def)
  also have "... = (∑ i = lrow..<Suc n. mset (row m i))"
    using step by (simp add: row_def multiset.map_comp o_def)
  finally show ?case .
qed auto

lemma mset_mat_conv_sum_cols: "mset_mat m = (∑ j∈{lcol..<ucol}. mset
(col m j))"
  using lcol_le_ucol unfolding mset_mat_def idxs_def
proof (induction rule: dec_induct)
  case (step n)
  have "image_mset m (mset_set ({lrow..<urow} × {lcol..<Suc n})) =
        image_mset m (mset_set ((λi. (i, n)) ‘ {lrow..<urow} ∪ {lrow..<urow}
× {lcol..<n}))"
    using step.prems step.hyps by (auto simp: Times_insert_right atLeastLessThanSuc)
  also have "... = image_mset m (mset_set ((λi. (i, n)) ‘ {lrow..<urow}))
+
                  image_mset m (mset_set ({lrow..<urow} × {lcol..<n}))"
    by (subst mset_set_Union) (auto simp flip: image_mset_mset_set)
  also have "... = image_mset m (image_mset (λi. (i, n)) (mset_set {lrow..<urow}))
+
                  image_mset m (mset_set ({lrow..<urow} × {lcol..<n}))"
    by (subst image_mset_mset_set [symmetric]) (auto simp: inj_on_def)
  also have "... = (∑ i = lcol..<Suc n. mset (col m i))"
    using step by (simp add: col_def multiset.map_comp o_def)
  finally show ?case .
qed auto
```

Lastly, we define the transposition operation:

```
definition transpose_mat :: "((nat × nat) ⇒ 'a) ⇒ (nat × nat) ⇒ 'a"
  where "transpose_mat m = (λ(i,j). m (j, i))"

lemma transpose_mat_apply: "transpose_mat m (j, i) = m (i, j)"
  by (simp add: transpose_mat_def)
```

sublocale *transpose: shearsort lcol ucol lrow urow*
  by *unfold_locales (fact lcol_le_ucol lrow_le_urow)+*

lemma *row_transpose [simp]: "transpose.row (transpose_mat m) i = col m i"*
  and *col_transpose [simp]: "transpose.col (transpose_mat m) i = row m i"*
  by *(simp_all add: transpose.row_def col_def transpose.col_def row_def transpose_mat_def)*

lemma *in_transpose_idxs_iff: "(j, i) ∈ transpose.idxs ⟷ (i, j) ∈ idxs"*
  by *(auto simp: idxs_def transpose.idxs_def)*

1.4 Snake-wise sortedness

Next, we define snake-wise sortedness. For this, even-numbered rows must be sorted ascendingly, the odd-numbered ones descendingly, etc. We will show a nicer characterisation of this below.

definition *snake_sorted :: "'a mat ⇒ bool"* **where**
  "snake_sorted m ⟷
    (∀i∈{lrow..<urow}. sorted_asc_desc (even i) (row m i)) ∧
    (∀i i' x y. lrow ≤ i ∧ i < i' ∧ i' < urow ∧ x ∈ set (row m i) ∧ y ∈ set (row m i') ⟶ x ≤ y)"

Next, we define the list of elements encountered on the snake-like path through the matrix, i.e. when traversing the matrix top to bottom, even-numbered rows left-to-right and odd-numbered rows right-to-left.

context
  fixes *m :: "'a mat"*
begin

function *snake_aux :: "nat ⇒ 'a list"* **where**
  "snake_aux i =
    (if i ≥ urow then [] else (if even i then row m i else rev (row m i)) @ snake_aux (Suc i))"
  by *auto*
termination by *(relation "measure (λi. urow - i)") auto*

lemmas *[simp del] = snake_aux.simps*

definition *snake :: "'a list"*
  where *"snake = snake_aux lrow"*

lemma *mset_snake_aux: "mset (snake_aux lrow') = (∑i∈{lrow'..<urow}. mset (row m i))"*
  by *(induction lrow' rule: snake_aux.induct; subst snake_aux.simps)*

```
    (simp_all add: sum.atLeast_Suc_lessThan)

lemma set_snake_aux: "set (snake_aux lrow') = (⋃i∈{lrow'..<urow}. set
(row m i))"
proof -
  have "set (snake_aux lrow') = set_mset (mset (snake_aux lrow'))"
    by simp
  thus ?thesis
    by (subst (asm) mset_snake_aux) (simp_all add: set_mset_sum)
qed
```

We can now show that snake-wise sortedness is equivalent to saying that *snake* is sorted.

```
lemma sorted_snake_aux_iff:
  "sorted (snake_aux lrow') ⟷
     (∀i∈{lrow'..<urow}. sorted_asc_desc (even i) (row m i)) ∧
     (∀i i' x y. lrow' ≤ i ∧ i < i' ∧ i' < urow ∧ x ∈ set (row m i)
∧ y ∈ set (row m i') ⟶ x ≤ y)"
proof -
  define sorted1 where
    "sorted1 = (λlrow'. ∀i∈{lrow'..<urow}. sorted_asc_desc (even i) (row
m i))"
  define sorted2 where
    "sorted2 = (λlrow'. ∀i i' x y. lrow' ≤ i ∧ i < i' ∧ i' < urow ∧
                        x ∈ set (row m i) ∧ y ∈ set (row m i') ⟶ x
≤ y)"
  have ivl_split: "{lrow'..<urow} = insert lrow' {Suc lrow'..<urow}" if
"lrow' < urow" for lrow'
    using that by auto

  have "sorted (snake_aux lrow') ⟷ sorted1 lrow' ∧ sorted2 lrow'"
  proof (induction lrow' rule: snake_aux.induct)
    case (1 lrow')
    show ?case
    proof (cases "lrow' ≥ urow")
      case True
      thus ?thesis
        by (subst snake_aux.simps) (auto simp: sorted1_def sorted2_def)
    next
      case False
      hence "sorted (snake_aux lrow') ⟷
               (sorted_asc_desc (even lrow') (row m lrow') ∧ sorted1 (Suc
lrow')) ∧
               (sorted2 (Suc lrow') ∧
                  (∀j∈{lcol..<ucol}. ∀i∈{Suc lrow'..<urow}. ∀y∈set
(row m i). y ≥ m (lrow', j)))"
        (is "_ ⟷ ?A ∧ (_ ∧ ?B)") using 1
        by (subst snake_aux.simps)
           (simp_all add: sorted_append set_row set_snake_aux sorted_asc_desc_def)
```

```
      also have "?A ⟷ sorted1 lrow'"
        using False unfolding sorted1_def by (auto simp: ivl_split)
      also have "?B ⟷ (∀i∈{Suc lrow'..<urow}. ∀x∈set (row m lrow').
∀y∈set (row m i). x ≤ y)"
        by (auto simp: set_row)
      also have "sorted2 (Suc lrow') ∧ ... ⟷ sorted2 lrow'"
      proof (safe, goal_cases)
        case 1
        thus ?case
          using False
          unfolding sorted2_def by (metis Suc_leI atLeastLessThan_iff
le_neq_implies_less)
      next
        case 2
        thus ?case
          unfolding sorted2_def using False by (auto simp: sorted2_def
dest: Suc_leD)
      next
        case 3
        thus ?case
          using False by (auto simp: sorted2_def dest!: Suc_le_lessD)
      qed
      finally show ?thesis .
    qed
  qed
  thus ?thesis by (simp add: sorted1_def sorted2_def)
qed

lemma sorted_snake_iff: "sorted snake ⟷ snake_sorted m"
  by (simp add: snake_def sorted_snake_aux_iff snake_sorted_def)

end
```

1.5 Definition of the abstract algorithm

We can now define shear sort on matrices. We will also show that the multiset of elements is preserved.

1.5.1 Sorting the rows

```
definition step1 :: "'a mat ⇒ 'a mat" where
  "step1 m = restrict_mat m (λ(i,j). sort_asc_desc (even i) (row m i)
! (j - lcol))"

lemma step1_outside [simp]: "z ∉ idxs ⟹ step1 m z = m z"
  by (simp add: step1_def restrict_mat_def)

lemma row_step1:
```

```
  "row (step1 m) i = (if i ∈ {lrow..<urow} then sort_asc_desc (even i)
(row m i) else row m i)"
  unfolding step1_def row_restrict_mat by (subst (2) row_def) (simp add:
map_nth_shift)

lemma mset_mat_step1 [simp]: "mset_mat (step1 m) = mset_mat m"
  by (simp add: mset_mat_conv_sum_rows row_step1)
```

1.5.2 Sorting the columns

```
definition step2 :: "'a mat ⇒ 'a mat" where
  "step2 m = restrict_mat m (λ(i,j). sort (col m j) ! (i - lrow))"

lemma step2_outside [simp]: "z ∉ idxs ⟹ step2 m z = m z"
  by (simp add: step2_def restrict_mat_def)

lemma col_step2: "col (step2 m) j = (if j ∈ {lcol..<ucol} then sort
(col m j) else col m j)"
  unfolding step2_def col_restrict_mat by (subst (2) col_def) (simp add:
map_nth_shift)

lemma mset_mat_step2 [simp]: "mset_mat (step2 m) = mset_mat m"
  by (simp add: mset_mat_conv_sum_cols col_step2)

lemma step2_height_le_1:
  assumes "urow ≤ lrow + 1"
  shows   "step2 m = m"
proof (rule matrix_eqI_cols, goal_cases)
  case (1 j)
  with assms have "length (col m j) ≤ 1"
    by auto
  hence "sort (col m j) = col m j"
    using sorted01 sorted_sort_id by blast
  thus ?case using 1
    by (auto simp: col_step2)
qed (auto simp: step2_def restrict_mat_def)
```

We also show the alternative definiton of *step2* involving transposition and sorting rows:

```
definition step2' :: "'a mat ⇒ 'a mat" where
  "step2' m = restrict_mat m (λ(i,j). sort (row m i) ! (j - lcol))"

lemma step2'_outside [simp]: "z ∉ idxs ⟹ step2' m z = m z"
  by (simp add: step2'_def restrict_mat_def)

lemma row_step2': "row (step2' m) i = (if i ∈ {lrow..<urow} then sort
(row m i) else row m i)"
  unfolding step2'_def row_restrict_mat by (subst (2) row_def) (simp add:
map_nth_shift)
```

end

context *shearsort*
begin

lemma *step2_altdef: "step2 m = transpose.transpose_mat (transpose.step2' (transpose_mat m))"*
  by *(rule matrix_eqI_cols, goal_cases)*
    (simp_all add: col_step2 transpose.row_step2' transpose_mat_apply in_transpose_idxs_iff)

1.5.3 Combining the two steps

definition *step* **where** *"step = step2 ∘ step1"*

lemma *step_outside [simp]: "z ∉ idxs ⟹ step m z = m z"*
  by *(auto simp: step_def step1_def step2_def restrict_mat_def)*

lemma *row_step_outside [simp]: "i ∉ {lrow..<urow} ⟹ row (step m) i = row m i"*
  by *(auto simp add: step_def step2_def row_step1)*

lemma *mset_mat_step [simp]: "mset_mat (step m) = mset_mat m"*
  by *(simp add: step_def)*

The overall algorithm now simply alternates between steps 1 and 2 sufficiently often for the result to stabilise. We will show below that a logarithmic number of steps suffices.

definition *shearsort :: "'a mat ⇒ 'a mat"* **where**
  "shearsort = step ^^ (ceillog2 (urow - lrow) + 1)"

The preservation of the multiset of elements is very easy to show:

theorem *mset_mat_shearsort [simp]: "mset_mat (shearsort m) = mset_mat m"*
proof -
  define *l* **where** *"l = ceillog2 (urow - lrow) + 1"*
  have *"mset_mat ((step ^^ l) m) = mset_mat m"*
    by *(induction l) auto*
  thus *?thesis* **by** *(simp add: shearsort_def l_def)*
qed

end

1.6 Restriction to boolean matrices

To more towards the proof of sortedness, we first take a closer look at shear sort on boolean matrices. Our ultimate goal is to show that shear sort

correctly sorts any boolean matrix in $\lceil log_2\ h \rceil$ + 1 steps, where h is the height of the matrix. By the 0–1 principle, this implies that shear sort works on a matrix of any type.

1.6.1 Preliminary definitions

We first define predicates that tell us whether a list is all zeros (i.e. False) or all ones (i.e. True). The significance of such lists is that we call all-zero rows at the top of the matrix and all-one rows at the bottom "clean", and we will show that even in the worst case, the number of non-clean rows halves in every step.

```
definition all0 :: "bool list ⇒ bool" where "all0 xs = (set xs ⊆ {False})"
definition all1 :: "bool list ⇒ bool" where "all1 xs = (set xs ⊆ {True})"

lemma all0_nth: "all0 xs ⟹ i < length xs ⟹ xs ! i = False"
  and all1_nth: "all1 xs ⟹ i < length xs ⟹ xs ! i = True"
  unfolding all0_def all1_def using nth_mem[of i xs] by blast+

lemma all0_imp_all_same [dest]: "all0 xs ⟹ all_same xs"
  and all1_imp_all_same [dest]: "all1 xs ⟹ all_same xs"
  unfolding all0_def all1_def all_same_def by blast+


locale shearsort_bool =
  fixes lrow urow lcol ucol :: nat
  assumes lrow_le_urow: "lrow ≤ urow"
  assumes lcol_le_ucol: "lcol ≤ ucol"
begin

sublocale shearsort lrow urow lcol ucol True
  by unfold_locales (fact lrow_le_urow lcol_le_ucol)+
```

We say that a matrix m of height h has a clean decomposition of order n if there are at most n non-clean rows, i.e. there exists a k such that m has k lines that are all 0 at the top and h - n - k lines that are all 1 at the bottom.

```
definition clean_decomp where
  "clean_decomp n m ⟷ (∃k. lrow ≤ k ∧ k + n ≤ urow ∧
     (∀i∈{lrow..<k}. all0 (row m i)) ∧ (∀i∈{k+n..<urow}. all1 (row m
i)))"
```

A matrix of height h trivially has a clean decomposition of order h.

```
lemma clean_decomp_initial: "clean_decomp (urow - lrow) m"
  unfolding clean_decomp_def by (rule exI[of _ lrow]) (use lrow_le_urow
in auto)

lemma all0_rowI:
```

```
  assumes "i ∈ {lrow..<urow}" "⋀j. j ∈ {lcol..<ucol} ⟹ ¬m (i, j)"
  shows   "all0 (row m i)"
  using assms unfolding set_row all0_def by auto

lemma all1_rowI:
  assumes "i ∈ {lrow..<urow}" "⋀j. j ∈ {lcol..<ucol} ⟹ m (i, j)"
  shows   "all1 (row m i)"
  using assms unfolding set_row all1_def by auto
```

The *step2* function on boolean matrices has the following nice characterisation: *step2 m* has a *1* at position *(i, j)* iff the number of *0*s in the column *j* is at most *i*.

```
lemma step2_bool:
  assumes "(i, j) ∈ idxs"
  shows   "step2 m (i, j) ⟷ i ≥ lrow + size (count (mset (col m j))
False)"
proof -
  have "∃k≤urow - lrow. sort (col m j) = replicate k False @ replicate
(urow - lrow - k) True"
    by (rule sorted_boolE) auto
  then obtain k where k:
    "k ≤ urow - lrow"
    "sort (col m j) = replicate k False @ replicate (urow - lrow - k)
True"
    by blast
  have "k = size (count (mset (sort (col m j))) False)"
    by (subst k(2)) auto
  hence k_eq: "k = size (count (mset (col m j)) False)"
    by simp

  have "step2 m (i, j) = col (step2 m) j ! (i - lrow)"
    using assms by (subst nth_col) (auto simp: idxs_def)
  also have "... = sort (col m j) ! (i - lrow)"
    using assms by (simp add: col_step2 idxs_def)
  also have "... ⟷ i ≥ lrow + k"
    using assms by (auto simp: k nth_append idxs_def)
  finally show ?thesis
    by (simp only: k_eq)
qed

end
```

1.6.2 Shearsort steps ignore clean rows

We now look at a at the matrix minor consisting of the *n* (possibly) non-clean rows in the middle of a matrix with a clean decomposition of order *n*. We call the new upper and lower index bounds for the rows *lrow'* and *urow'*.

```
locale sub_shearsort_bool = shearsort_bool +
  fixes lrow' urow' :: nat and m :: "bool mat"
  assumes subrows: "lrow ≤ lrow'" "lrow' ≤ urow'" "urow' ≤ urow"
  assumes all0_first: "⋀i. i ∈ {lrow..<lrow'} ⟹ all0 (row m i)"
  assumes all1_last: "⋀i. i ∈ {urow'..<urow} ⟹ all1 (row m i)"
begin

sublocale sub: shearsort_bool lrow' urow' lcol ucol
  by unfold_locales (use subrows lcol_le_ucol in auto)

lemma idxs_subset: "sub.idxs ⊆ idxs"
  using subrows by (auto simp: sub.idxs_def idxs_def)
```

It is easy to see that *step1* does not touch the clean rows at all (i.e. it can be seen as operating entirely on the minor):

```
lemma sub_step1: "sub.step1 m = step1 m"
proof (rule sym, rule matrix_eqI_rows, goal_cases)
  case (1 i)
  show "row (step1 m) i = row (sub.step1 m) i"
  proof (cases "i ∈ {lrow'..<urow'}")
    case True
    thus ?thesis using subrows
      by (simp add: row_step1 sub.row_step1)
  next
    case False
    hence "all_same (sub.row m i)"
      using all0_first[of i] all1_last[of i] 1 by auto
    thus ?thesis using 1
      by (simp add: row_step1 sub.row_step1 sort_asc_desc_all_same)
  qed
next
  case (2 i j)
  with idxs_subset have "(i, j) ∉ sub.idxs"
    by auto
  thus ?case
    using 2 by auto
qed
```

Every column of the matrix has *lrow' - lrow 0*s at the top and *urow - urow' 1*s at the bottom:

```
lemma col_conv_sub_col:
  assumes "j ∈ {lcol..<ucol}"
  shows   "col m j = replicate (lrow' - lrow) False @ sub.col m j @ replicate
(urow - urow') True"
proof (intro nth_equalityI, goal_cases)
  case 1
  thus ?case using subrows by auto
next
  case (2 i)
```

```
  consider "i < lrow' - lrow" | "i ∈ {lrow'-lrow..<urow'-lrow}" | "i ∈
{urow'-lrow..<urow-lrow}"
    using 2 by force
  thus ?case
  proof cases
    case 1
    hence "all0 (row m (lrow + i))"
      by (intro all0_first) auto
    hence "row m (lrow + i) ! (j - lcol) = False"
      using assms by (intro all0_nth) auto
    thus ?thesis using 1 assms subrows
      by (auto simp: nth_append)
  next
    case 2
    have "(replicate (lrow' - lrow) False @ sub.col m j @ replicate (urow
- urow') True) ! i =
          (sub.col m j @ replicate (urow - urow') True) ! (i - (lrow'
- lrow))"
      using 2 by (subst nth_append_right) auto
    also have "... = sub.col m j ! (i - (lrow' - lrow))"
      using 2 by (subst nth_append_left) auto
    also have "... = m (lrow + i, j)"
      using 2 subrows by (subst sub.nth_col) (auto simp: algebra_simps)
    also have "... = col m j ! i"
      using 2 subrows by (subst nth_col) auto
    finally show ?thesis ..
  next
    case 3
    hence "all1 (row m (lrow + i))"
      by (intro all1_last) auto
    hence "row m (lrow + i) ! (j - lcol) = True"
      using assms by (intro all1_nth) auto
    hence "col m j ! i = True"
      using 3 assms by auto
    also have "True = (replicate (lrow' - lrow) False @ sub.col m j @
replicate (urow - urow') True) ! i"
      by (subst append_assoc [symmetric], subst nth_append_right) (use
3 subrows in auto)
    finally show ?thesis .
  qed
qed
```

mset *step2* preserves the clean rows at the bottom and top.

```
lemma all0_step2:
  assumes "i ∈ {lrow..<lrow'}"
  shows   "all0 (row (step2 m) i)"
proof -
  have *: "row (step2 m) i ! (j - lcol) = False" if j: "j ∈ {lcol..<ucol}"
for j
```

```
  proof -
    have "row (step2 m) i ! (j - lcol) = step2 m (i, j)"
      using assms j subrows by (subst nth_row) auto
    also have "step2 m (i, j) = col (step2 m) j ! (i - lrow)"
      using assms j subrows by (subst nth_col) auto
    also have "... = False"
      using j assms by (auto simp: col_step2 col_conv_sub_col sort_append_replicate_left
                                   sort_append_replicate_right nth_append)
    finally show ?thesis .
  qed
  have "row (step2 m) i ! j = False" if "j < length (row (step2 m) i)"
for j
    using *[of "j + lcol"] that by auto
  thus ?thesis unfolding all0_def set_conv_nth
    by blast
qed

lemma all1_step2:
  assumes "i ∈ {urow'..<urow}"
  shows   "all1 (row (step2 m) i)"
proof -
  have *: "row (step2 m) i ! (j - lcol) = True" if j: "j ∈ {lcol..<ucol}"
for j
  proof -
    have "row (step2 m) i ! (j - lcol) = step2 m (i, j)"
      using assms j subrows by (subst nth_row) auto
    also have "step2 m (i, j) = col (step2 m) j ! (i - lrow)"
      using assms j subrows by (subst nth_col) auto
    also have "... = ((replicate (lrow' - lrow) False @ sort (sub.col
m j)) @ replicate (urow - urow') True) ! (i - lrow)"
      using j assms by (simp add: col_step2 col_conv_sub_col sort_append_replicate_left
sort_append_replicate_right)
    also have "... = True"
      using j assms subrows by (subst nth_append_right) auto
    finally show ?thesis .
  qed
  have "row (step2 m) i ! j = True" if "j < length (row (step2 m) i)"
for j
    using *[of "j + lcol"] that by auto
  thus ?thesis unfolding all1_def set_conv_nth
    by blast
qed
```

Consequently, *step2* can also be seen as operating only on the minor.

```
lemma sub_step2: "sub.step2 m = step2 m"
proof (rule sym, rule matrix_eqI_cols, goal_cases)
  case (1 j)
  interpret step2: sub_shearsort_bool lrow urow lcol ucol lrow' urow' "sub.step2
m"
```

```
    by unfold_locales
       (use subrows all0_step2 all1_step2 all0_first all1_last in <auto
simp: sub.step2_def>)
  have "col (step2 m) j =
          sort (replicate (lrow' - lrow) False @ sub.col m j @ replicate
(urow - urow') True)"
    using 1 by (simp add: col_step2 col_conv_sub_col)
  also have "... = replicate (lrow' - lrow) False @ sort (sub.col m j)
@ replicate (urow - urow') True"
    by (simp add: sort_append_replicate_left sort_append_replicate_right)
  also have "... = col (sub.step2 m) j"
    using 1 subrows by (simp add: sub.col_step2 step2.col_conv_sub_col)
  finally show ?case .
next
  case (2 i j)
  with idxs_subset have "(i, j) ∉ sub.idxs"
    by auto
  thus ?case
    using 2 by auto
qed
```

Thus, the same holds for the combined shear sort step.

```
lemma sub_step: "sub.step m = step m"
proof -
  interpret step1: sub_shearsort_bool lrow urow lcol ucol lrow' urow' "step1
m"
    using all0_first all1_last subrows
    by unfold_locales (auto simp: sub.row_step1 simp flip: sub_step1)
  show ?thesis
    unfolding step_def o_def by (simp add: step1.sub_step2 sub.step_def
sub_step1)
qed

end
```

1.6.3 Correctness of boolean shear sort

We are now ready for the final push. The main work in this section is to show that if we run a single shear sort step on a matrix of height h, the number of non-clean rows in the result is no greater than $\lceil h/2 \rceil$.

Together with the fact from above that the step preserves clean rows and can such be thought of as operating solely on the non-clean minor, this means that the number of non-clean rows at least halves in every step, leading to a matrix with at most one non-clean row after $\lceil log_2\ h \rceil$ steps.

```
context shearsort_bool
begin
```

If we look at two rows, one of which is sorted in ascending order and one

in descending order, there exists a boolean value `x` such that every column contains an `x` (i.e. for every column index `j`, at least one of the two rows has an `x` at index `j`).

```
lemma clean_decomp_step2_aux:
  fixes m :: "bool mat"
  assumes "i ∈ {lrow..<urow}" "i' ∈ {lrow..<urow}"
  assumes "sorted (row m i)" "sorted (rev (row m i'))"
  shows   "∃x. ∀j∈{lcol..<ucol}. x ∈ {m (i, j), m (i', j)}"
proof -
  obtain k1 where k1: "k1 ≤ ucol - lcol"
     "row m i = replicate k1 False @ replicate (ucol - lcol - k1) True"
    using sorted_boolE[OF assms(3), of "ucol - lcol"] by auto
  obtain k2 where k2: "k2 ≤ ucol - lcol"
     "row m i' = replicate k2 True @ replicate (ucol - lcol - k2) False"
    using rev_sorted_boolE[OF assms(4), of "ucol - lcol"] by auto

  define x where "x = (k2 > k1)"
  have "x ∈ {m (i, j), m (i', j)}" if j: "j ∈ {lcol..<ucol}" for j
  proof -
    have "x ∈ {row m i ! (j - lcol), row m i' ! (j - lcol)}"
      using j k1 k2 by (auto simp: x_def nth_append)
    thus "x ∈ {m (i, j), m (i', j)}"
      using assms(1,2) j by auto
  qed
  thus ?thesis by metis
qed
```

`step1` leaves every even-numbered row in the matrix sorted in ascending order and every odd-numbered row in descending order:

```
lemma sorted_asc_desc_row_step1:
  "i ∈ {lrow..<urow} ⟹ sorted_asc_desc (even i) (row (step1 m) i)"
  by (auto simp: row_step1 sorted_asc_desc_def sort_asc_desc_def)
```

These two facts imply that applying `step2` to such a matrix indeed leads to at most $\lceil h/2 \rceil$ non-clean rows. The argument is as follows: we go through the matrix top-to-bottom, grouping adjacent rows into pairs of two (ignoring the last row if the matrix has odd height).

The above lemma proves that each such pair of rows either has a `1` in every column or a `0` in every column. Thus, the maximum number k_0 such that every column contains at least k_0 `0`s plus the maximum number k_1 such that every column contains at least k_1 1s is at least $\lfloor h/2 \rfloor$. Thus, after applying `step2`, we have at least k_0 all-zero rows at the top and at least k_1 all-one rows at the bottom, and therefore at least $\lfloor h/2 \rfloor$ clean lines in total.

```
lemma clean_decomp_step2:
  assumes "⋀i. i ∈ {lrow..<urow} ⟹ sorted_asc_desc (even i) (row m
i)"
  shows   "clean_decomp ((urow - lrow + 1) div 2) (step2 m)"
```

```
proof -
  define r where [simp]: "r = row m"

  have "∃x. ∀j∈{lcol..<ucol}. x ∈ {m (lrow+2*i, j), m (lrow+2*i+1, j)}"
    if i: "i < (urow - lrow) div 2" for i
  proof -
    have *: "lrow + 2 * i ∈ {lrow..<urow}" "lrow + 2 * i + 1 ∈ {lrow..<urow}"
      using i by auto
    have "sorted_asc_desc (even (lrow+2*i)) (row m (lrow+2*i))"
         "sorted_asc_desc (even (lrow+2*i+1)) (row m (lrow+2*i+1))"
      by (intro assms *)+
    hence "sorted (row m (lrow+2*i)) ∧ sorted (rev (row m (lrow+2*i+1)))
∨
          sorted (rev (row m (lrow+2*i))) ∧ sorted (row m (lrow+2*i+1))"
      using assms[of i] by (auto simp: sorted_asc_desc_def split: if_splits)
    thus ?thesis
      using clean_decomp_step2_aux[of "lrow+2*i" "lrow+2*i+1" m]
            clean_decomp_step2_aux[of "lrow+2*i+1" "lrow+2*i" m] *
      by (metis insert_iff)
  qed

  then obtain f where f: "⋀i j. i < (urow - lrow) div 2 ⟹ j ∈ {lcol..<ucol}
⟹
                                  f i ∈ {m (lrow+2*i, j), m (lrow+2*i+1,
j)}" by metis

  define I where "I = (λx. filter_mset (λi. f i = x) (mset [0..<(urow
- lrow) div 2]))"
  have size_I: "size (I x) ≤ (urow - lrow) div 2" for x
    unfolding I_def by (rule order.trans[OF size_filter_mset_lesseq])
auto

  have size_I_le: "count (mset (col m j)) x ≥ size (I x)" if j: "j ∈
{lcol..<ucol}" for x j
  proof -
    define g where "g = (λi. if m (lrow+2*i, j) = x then lrow + 2 * i
else lrow + 2 * i + 1)"
    have "size (I x) = card {i. i < (urow - lrow) div 2 ∧ f i = x}"
      by (simp add: I_def)
    also have "... = card (g ‘ {i. i < (urow - lrow) div 2 ∧ f i = x})"
      by (intro card_image [symmetric] inj_onI) (auto simp: g_def split:
if_splits)
    also have "... ≤ card {i. lrow ≤ i ∧ i < urow ∧ m (i, j) = x}"
      using j f by (intro card_mono) (auto simp: g_def)
    also have "... = count (mset (col m j)) x"
      by (simp add: count_conv_size_mset mset_col filter_image_mset)
    finally show ?thesis .
  qed
```

```
  show ?thesis
    unfolding clean_decomp_def
  proof (intro exI[of _ "lrow + size (I False)"] conjI ballI)
    show "lrow + size (I False) + (urow - lrow + 1) div 2 ≤ urow"
      using size_I[of False] lrow_le_urow by linarith
  next
    fix i assume i: "i ∈ {lrow..<lrow + size (I False)}"
    show "all0 (row (step2 m) i)"
    proof (intro all0_rowI)
      fix j assume j: "j ∈ {lcol..<ucol}"
      show "¬step2 m (i, j)"
        using i j size_I[of False] size_I_le[of j False]
        by (subst step2_bool) (auto simp: idxs_def)
    qed (use i size_I[of False] in auto)
  next
    fix i assume i: "i ∈ {lrow + size (I False) + (urow - lrow + 1) div
2..<urow}"
    show "all1 (row (step2 m) i)"
    proof (intro all1_rowI)
      fix j assume j: "j ∈ {lcol..<ucol}"

      have "size (I True) ≤ count (mset (col m j)) True"
        using j by (intro size_I_le) auto
      moreover have "size (I False + I True) = size (mset [0..<(urow
- lrow) div 2])"
        unfolding I_def by (subst union_filter_mset_complement) auto
      hence "size (I True) + size (I False) = (urow - lrow) div 2"
        by simp
      moreover have "count (mset (col m j)) True + count (mset (col m
j)) False =
                     size (mset (col m j))"
        by (simp add: size_conv_count_bool_mset)
      hence "count (mset (col m j)) True + count (mset (col m j)) False
= urow - lrow"
        by simp
      ultimately have "count (mset (col m j)) False ≤ size (I False)
+ Suc (urow - lrow) div 2"
        by linarith
      with i have "lrow + count (mset (col m j)) False ≤ i"
        by simp
      thus "step2 m (i, j)"
        using i j by (subst step2_bool) (auto simp: idxs_def)
    qed (use i size_I[of False] in auto)
  qed auto
qed

lemma clean_decomp_step_aux:
  "clean_decomp ((urow - lrow + 1) div 2) (step m)"
  unfolding step_def o_def by (intro clean_decomp_step2 sorted_asc_desc_row_step1)
```

We can now finally show that the number of non-clean rows halves in every step:

```
lemma clean_decomp_step:
  assumes "clean_decomp n m"
  shows   "clean_decomp ((n + 1) div 2) (step m)"
proof -
  from assms obtain k where k:
    "lrow ≤ k" "k + n ≤ urow" "∀ i∈{lrow..<k}. all0 (row m i)" "∀ i∈{k+n..<urow}.
all1 (row m i)"
    unfolding clean_decomp_def by metis
  interpret sub_shearsort_bool lrow urow lcol ucol k "k + n"
    by unfold_locales (use k lcol_le_ucol in auto)
  have idxs_subset: "sub.idxs ⊆ idxs"
    using k by (auto simp: sub.idxs_def idxs_def)

  have "sub.clean_decomp ((n + 1) div 2) (sub.step m)"
    using sub.clean_decomp_step_aux[of m] by simp
  then obtain k' where k':
    "k ≤ k'" "k' + (n + 1) div 2 ≤ k + n"
    "∀ i∈{k..<k'}. all0 (row (sub.step m) i)"
    "∀ i∈{k'+(n+1) div 2..<k + n}. all1 (row (sub.step m) i)"
    unfolding sub.clean_decomp_def by blast

  have "∀ i∈{lrow..<k'}. all0 (sub.row (step m) i)"
    using k'(3) k(3) by (auto simp flip: sub_step)
  moreover have "(∀ i∈{k' + (n + 1) div 2..<urow}. all1 (sub.row (step
m) i))"
    using k'(4) k(4) by (auto simp flip: sub_step)
  moreover have "lrow ≤ k'" "k' + (n + 1) div 2 ≤ urow"
    using k(1,2) k'(1,2) by linarith+
  ultimately show ?thesis
    unfolding clean_decomp_def by metis
qed
```

Moreover, if we have a matrix that has at most one non-clean row, applying one last step of shear sort leads to a snake-sorted matrix. This is because

1. *step1* leaves the clean rows untouched and sorts the non-clean row (if it exists) in the correct order.
2. *step2* leaves the clean parts of the columns untouched, and since the non-clean part has height at most 1, it also leaves that part untouched.

```
lemma snake_sorted_step_final:
  assumes "clean_decomp n m" and "n ≤ 1"
  shows   "snake_sorted (step m)"
proof -
  define n' where "n' = (n + 1) div 2"
```

```
  have "n' ≤ 1"
    using ‹n ≤ 1› unfolding n'_def by linarith
  have "clean_decomp n' (step m)"
    unfolding n'_def by (rule clean_decomp_step) fact
  from assms(1) obtain k where k: "k ≥ lrow" "k + n ≤ urow"
     "⋀i. i ∈ {lrow..<k} ⟹ all0 (row m i)"
     "⋀i. i ∈ {k + n..<urow} ⟹ all1 (row m i)"
    unfolding clean_decomp_def by metis
  interpret sub_shearsort_bool lrow urow lcol ucol k "k + n"
    by unfold_locales (use k in auto)

  show ?thesis
    unfolding snake_sorted_def
  proof safe
    fix i assume "i ∈ {lrow..<urow}"
    thus "sorted_asc_desc (even i) (row (step m) i)"
      by (metis ‹n ≤ 1› comp_eq_dest_lhs nat_add_left_cancel_le
          sorted_asc_desc_row_step1 sub.step2_height_le_1 sub.step_def
sub_step sub_step1)
  next
    fix i i' x y
    assume *: "lrow ≤ i" "i < i'" "i' < urow"
              "x ∈ set (row (step m) i)" "y ∈ set (row (step m) i')"
    have **: "row (step m) i = row m i" if "i ≠ k" for i
    proof -
      have "row (sub.step m) i = row m i"
        using that assms(2) by force
      thus ?thesis
        by (simp add: sub_step)
    qed

    from *(1-3) consider "i < k" | "i' > k"
      by linarith
    thus "x ≤ y"
    proof cases
      assume "i < k"
      hence "all0 (row m i)"
        using * by (intro k) auto
      thus "x ≤ y"
        using * ‹i < k› by (auto simp: all0_def **)
    next
      assume "i' > k"
      hence "all1 (row m i')"
        using * assms(2) by (intro k) auto
      thus "x ≤ y"
        using * ‹i' > k› by (auto simp: all1_def **)
    qed
  qed
qed
```

It is now easy to show that shear sort is indeed correct for boolean matrices.

```
lemma snake_sorted_shearsort_bool: "snake_sorted (shearsort m)"
proof -
  define div2 where "div2 = (λx::nat. (x + 1) div 2)"
  define l where "l = ceillog2 (urow - lrow)"
  have "clean_decomp ((div2 ^^ l) (urow - lrow)) ((step ^^ l) m)"
  proof (induction l)
    case (Suc k)
    have "clean_decomp (((div2 ^^ k) (urow - lrow) + 1) div 2) (step ((step
^^ k) m))"
      by (intro clean_decomp_step Suc.IH)
    thus ?case by (simp add: div2_def)
  qed (auto intro: clean_decomp_initial)
  moreover have "(div2 ^^ l) (urow - lrow) ≤ 1"
    unfolding l_def div2_def by (rule funpow_div2_ceillog2_le_1)
  ultimately have "snake_sorted (step ((step ^^ l) m))"
    by (rule snake_sorted_step_final)
  thus ?thesis
    by (simp add: shearsort_def l_def)
qed

end
```

1.7 Shearsort commutes with monotone functions

To invoke the 0–1 principle, we must now prove that shear sort commutes with monotone functions. We will only show it for functions that return booleans, since that is all we need, but it could easily be shown the same way for a more general result type as well.

```
context shearsort
begin

interpretation bool: shearsort_bool lrow urow lcol ucol
  by unfold_locales (fact lrow_le_urow lcol_le_ucol)+

context
  fixes f :: "'a ⇒ bool"
begin

lemma row_commute: "row (f ∘ m) i = map f (row m i)"
  and col_commute: "col (f ∘ m) i = map f (col m i)"
  by (simp_all add: row_def col_def)

lemma restrict_mat_commute:
  assumes "⋀i j. (i, j) ∈ idxs ⟹ f (m' (i, j)) = fm' (i, j)"
  shows   "bool.restrict_mat (f ∘ m) fm' = f ∘ restrict_mat m m'"
  using assms by (auto simp: restrict_mat_def bool.restrict_mat_def fun_eq_iff)
```

```
lemma step1_mono_commute: "mono f ⟹ bool.step1 (f ∘ m) = f ∘ step1
m"
  unfolding step1_def bool.step1_def
  by (intro restrict_mat_commute) (auto simp: idxs_def sort_asc_desc_map_mono
row_commute)

lemma step2_mono_commute: "mono f ⟹ bool.step2 (f ∘ m) = f ∘ step2
m"
  unfolding step2_def bool.step2_def
  by (intro restrict_mat_commute) (auto simp: idxs_def sort_map_mono col_commute)

lemma step_mono_commute: "mono f ⟹ bool.step (f ∘ m) = f ∘ step m"
  by (simp add: step_def bool.step_def step1_mono_commute step2_mono_commute)

lemma snake_aux_commute: "bool.snake_aux (f ∘ m) lrow' = map f (snake_aux
m lrow')"
  by (induction lrow' rule: snake_aux.induct;
      subst snake_aux.simps; subst bool.snake_aux.simps)
     (auto simp: row_commute rev_map simp del: o_apply)

lemma snake_commute: "bool.snake (f ∘ m) = map f (snake m)"
  by (simp add: snake_def bool.snake_def snake_aux_commute)

lemma shearsort_mono_commute:
  assumes "mono f"
  shows   "bool.shearsort (f ∘ m) = f ∘ shearsort m"
proof -
  have "(bool.step ^^ k) (f ∘ m) = f ∘ (step ^^ k) m" for k
    by (induction k) (simp_all add: step_mono_commute assms del: o_apply
                                add: o_apply[of "bool.step"] o_apply[of
step])
  from this[of "ceillog2 (urow - lrow) + 1"] show ?thesis
    unfolding shearsort_def bool.shearsort_def .
qed

end
```

1.8 Final correctness theorem

All that is left now is a routine application of the 0–1 principle.

```
theorem snake_sorted_shearsort: "snake_sorted (shearsort m)"
proof (rule ccontr)
  define xs where "xs = snake (shearsort m)"
  assume "¬snake_sorted (shearsort m)"
  hence "¬sorted (snake (shearsort m))"
    by (simp add: sorted_snake_iff)
  then obtain i j where ij: "i < j" "j < length xs" "xs ! i > xs ! j"
    by (auto simp: sorted_iff_nth_mono_less xs_def)
  define f where "f = (λx. x > xs ! j)"
```

```
  have [simp]: "mono f"
    by (auto simp: f_def mono_def)

  have "sorted (bool.snake (bool.shearsort (f ∘ m)))"
    by (simp add: bool.sorted_snake_iff bool.snake_sorted_shearsort_bool)
  also have "bool.snake (bool.shearsort (f ∘ m)) = map f xs"
    by (simp add: xs_def shearsort_mono_commute snake_commute)
  finally have "f (xs ! i) ≤ f (xs ! j)"
    using ij unfolding sorted_iff_nth_mono_less by auto
  hence "xs ! i ≤ xs ! j"
    by (auto simp: f_def)
  with ij(3) show False by simp
qed

end
```

1.9 Refinement to lists

Next, we define a refinement of matrices to lists of lists and show the correctness of the corresponding shear sort implementation. Note that this is not useful as an actual implementation in practice since the fact that we have to transpose the list of lists once in every step negates all the advantage of having a parallel algorithm.

```
primrec step1_list :: "bool ⇒ 'a :: linorder list list ⇒ 'a list list"
where
  "step1_list b [] = []"
| "step1_list b (xs # xss) = sort_asc_desc b xs # step1_list (¬b) xss"

definition step2_list :: "'a :: linorder list list ⇒ 'a list list"
  where "step2_list xss =
           (if xss = [] ∨ hd xss = [] then xss else transpose (map sort
(transpose xss)))"

definition shearsort_list :: "bool ⇒ 'a :: linorder list list ⇒ 'a list
list" where
  "shearsort_list b xss = ((step2_list ∘ step1_list b) ^^ (ceillog2 (length
xss) + 1)) xss"

primrec snake_list :: "bool ⇒ 'a list list ⇒ 'a list" where
  "snake_list asc [] = []"
| "snake_list asc (xs # xss) = (if asc then xs else rev xs) @ snake_list
(¬asc) xss"

lemma mset_snake_list: "mset (snake_list b xss) = mset (concat xss)"
  by (induction xss arbitrary: b) auto

definition (in shearsort) mat_of_list :: "'a list list ⇒ 'a mat"
  where "mat_of_list xss = (λ(i,j). xss ! (i - lrow) ! (j - lcol))"
```

The following relator relates a matrix to a list of rows. It ensure that the dimensions and the entries are the same.

```
definition (in shearsort) mat_list_rel :: "'a mat ⇒ 'a list list ⇒ bool"
where
  "mat_list_rel m xss ⟷
     length xss = urow - lrow ∧ (∀xs∈set xss. length xs = ucol - lcol)
∧
     (∀i j. i < urow - lrow ∧ j < ucol - lcol ⟶ xss ! i ! j = m (lrow
+ i, lcol + j))"

lemma (in shearsort) mat_list_rel_transpose [intro]:
  assumes "mat_list_rel m xss" "xss ≠ []"
  shows   "transpose.mat_list_rel (transpose_mat m) (transpose xss)"
proof -
  have "transpose xss = map (λi. map (λj. xss ! j ! i) [0..<length xss])
[0..<ucol - lcol]"
    using assms
    by (intro transpose_rectangle[where n = "ucol - lcol"]) (auto simp:
mat_list_rel_def)
  thus ?thesis
    using assms by (auto simp: mat_list_rel_def transpose.mat_list_rel_def
transpose_mat_def)
qed

lemma (in shearsort) mat_list_rel_row [intro]:
  assumes "mat_list_rel m xss" "i ∈ {lrow..<urow}"
  shows   "row m i = xss ! (i - lrow)"
  using assms unfolding row_def mat_list_rel_def by (intro nth_equalityI)
auto

lemma (in shearsort) mat_list_rel_mset:
  assumes "mat_list_rel m xss"
  shows   "mset_mat m = (∑xs←xss. mset xs)"
proof -
  define S where "S = (SIGMA i:{..<length xss}. {..<length (xss ! i)})"
  have "mset_mat m = image_mset m (mset_set ({lrow..<urow} × {lcol..<ucol}))"
    by (simp add: mat_list_rel_def mset_mat_def idxs_def)
  also have "... = image_mset (λ(i,j). xss ! (i - lrow) ! (j - lcol))
                   (mset_set ({lrow..<urow} × {lcol..<ucol}))"
    using lcol_le_ucol lrow_le_urow diff_less_mono
    by (intro image_mset_cong) (use assms in ‹auto simp: mat_list_rel_def›)
  also have "... = image_mset (λ(i,j). xss ! i ! j)
                   (image_mset (λ(i,j). (i - lrow, j - lcol))
                     (mset_set ({lrow..<urow} × {lcol..<ucol})))"
    by (simp add: multiset.map_comp o_def case_prod_unfold)
  also have "image_mset (λ(i,j). (i - lrow, j - lcol)) (mset_set ({lrow..<urow}
× {lcol..<ucol})) =
               mset_set ((λ(i,j). (i - lrow, j - lcol)) ‘ ({lrow..<urow}
× {lcol..<ucol}))"
```

```
    by (rule image_mset_mset_set) (auto intro!: inj_onI)
  also have "bij_betw (λ(i,j). (i - lrow, j - lcol)) ({lrow..<urow} ×
{lcol..<ucol})
              ({..<urow-lrow} × {..<ucol-lcol})"
    by (rule bij_betwI[of _ _ _ "λ(i,j). (i + lrow, j + lcol)"]) auto
  hence "(λ(i,j). (i - lrow, j - lcol)) ‘ ({lrow..<urow} × {lcol..<ucol})
=
             {..<urow-lrow} × {..<ucol-lcol}"
    by (simp add: bij_betw_def)
  also have "{..<urow-lrow} × {..<ucol-lcol} = S"
    unfolding S_def by (rule Sigma_cong) (use assms in ‹simp_all add:
mat_list_rel_def›)
  also have "image_mset (λ(i,j). xss ! i ! j) (mset_set S) = (∑(i,j)∈#mset_set
S. {#xss ! i ! j#})"
    by (simp add: case_prod_unfold)
  also have "... = (∑(i,j)∈S. {#xss ! i ! j#})"
    by (rule sum_unfold_sum_mset [symmetric])
  also have "... = (∑xs←xss. ∑x←xs. {#x#})"
    unfolding S_def
    by (subst sum.Sigma [symmetric])
       (auto simp: atLeast0LessThan sum.list_conv_set_nth simp del: sum_list_singleton_mset)
  also have "... = (∑xs←xss. mset xs)"
    by simp
  finally show ?thesis .
qed

lemma (in shearsort) mat_list_rel_of_list:
  assumes "length xss = urow - lrow" "⋀xs. xs ∈ set xss ⟹ length xs
= ucol - lcol"
  shows   "mat_list_rel (mat_of_list xss) xss"
  using assms by (auto simp: mat_list_rel_def mat_of_list_def)

lemma (in shearsort) mset_mat_of_list:
  assumes "length xss = urow - lrow" "⋀xs. xs ∈ set xss ⟹ length xs
= ucol - lcol"
  shows   "mset_mat (mat_of_list xss) = (∑xs←xss. mset xs)"
  using mat_list_rel_mset[OF mat_list_rel_of_list[OF assms]] by simp


context shearsort
begin

lemma mat_list_rel_col [intro]:
  assumes "mat_list_rel m xss" "j ∈ {lcol..<ucol}" "xss ≠ []"
  shows   "col m j = transpose xss ! (j - lcol)"
  using transpose.mat_list_rel_row[OF mat_list_rel_transpose[OF assms(1,3)]
assms(2)] by simp

lemma length_step1_list [simp]: "length (step1_list b xss) = length xss"
```

```
  by (induction xss arbitrary: b) auto

lemma nth_step1_list:
  "i < length xss ⟹ step1_list b xss ! i = sort_asc_desc (b = even i)
(xss ! i)"
proof -
  have [simp]: "(¬b1) = b2 ⟷ b1 = (¬b2)" for b1 b2
    by auto
  show ?thesis if "i < length xss"
    using that by (induction xss arbitrary: b i) (auto simp: nth_Cons
split: nat.splits)
qed

lemma mat_list_rel_step1:
  assumes "mat_list_rel m xss"
  shows   "mat_list_rel (step1 m) (step1_list (even lrow) xss)"
  using assms mat_list_rel_row[OF assms]
  unfolding mat_list_rel_def
  by (force simp: step1_def nth_step1_list restrict_mat_def set_conv_nth
idxs_def)

lemma mat_list_rel_step2:
  assumes [intro]: "mat_list_rel m xss"
  shows   "mat_list_rel (step2 m) (step2_list xss)"
proof (cases "lcol ≥ ucol ∨ lrow ≥ urow")
  case False
  define m' xss' where "m' = transpose_mat m" and "xss' = transpose xss"
  define step2' where
    "step2' = transpose.restrict_mat m' (λ(i, j). sort (transpose.row
m' i) ! (j - lrow))"

  from False assms have "xss ≠ []"
    by (auto simp: mat_list_rel_def not_le set_conv_nth hd_conv_nth)
  from False assms this have "hd xss ≠ []"
    by (cases xss) (auto simp: mat_list_rel_def)
  have "xss' ≠ []"
    using ‹xss ≠ []› ‹hd xss ≠ []› unfolding xss'_def by (subst transpose_empty)
auto

  have *: "transpose.mat_list_rel m' xss'"
    using ‹xss ≠ []› unfolding m'_def xss'_def by blast
  hence "transpose.mat_list_rel step2' (map sort xss')"
    unfolding step2'_def using transpose.mat_list_rel_row[OF *]
    by (auto simp: transpose.mat_list_rel_def transpose.restrict_mat_def
transpose.idxs_def)
  hence "mat_list_rel (transpose_mat step2') (transpose (map sort xss'))"
    using ‹xss' ≠ []› by (intro transpose.mat_list_rel_transpose) auto
  also have "transpose_mat step2' = step2 m"
    by (simp add: step2_def step2'_def restrict_mat_def transpose.restrict_mat_def
```

```
                transpose.idxs_def idxs_def transpose_mat_def fun_eq_iff
m'_def
                transpose.row_def col_def)
  also have "transpose (map sort xss') = step2_list xss"
    using ‹xss ≠ []› ‹hd xss ≠ []› by (simp add: step2_list_def xss'_def)
  finally show ?thesis .
qed (use assms in ‹auto simp: mat_list_rel_def step2_list_def›)

lemma mat_list_rel_step:
  "mat_list_rel m xss ⟹ mat_list_rel (step m) (step2_list (step1_list
(even lrow) xss))"
  by (simp add: step_def mat_list_rel_step1 mat_list_rel_step2)

lemma mat_list_rel_shearsort:
  assumes "mat_list_rel m xss"
  shows   "mat_list_rel (shearsort m) (shearsort_list (even lrow) xss)"
proof -
  define l where "l = ceillog2 (urow - lrow) + 1"
  have "mat_list_rel ((step ^^ l) m) (((step2_list ∘ step1_list (even
lrow)) ^^ l) xss)"
    using assms by (induction l) (auto simp: mat_list_rel_step)
  moreover have "length xss = urow - lrow"
    using assms by (simp add: mat_list_rel_def)
  ultimately show ?thesis
    by (simp add: shearsort_list_def shearsort_def l_def)
qed

lemma mat_list_rel_snake_aux:
  assumes "mat_list_rel m xss" "lrow' ∈ {lrow..urow}"
  shows   "snake_aux m lrow' = snake_list (even lrow') (drop (lrow' -
lrow) xss)"
  using assms(2)
proof (induction lrow' rule: snake_aux.induct)
  case (1 lrow')
  have len: "length xss = urow - lrow"
    using assms(1) by (auto simp: mat_list_rel_def)

  show ?case
  proof (cases "lrow' < urow")
    case False
    thus ?thesis using len
      by (subst snake_aux.simps) auto
  next
    case True
    hence "snake_aux m lrow' =
             (if even lrow' then row m lrow' else rev (row m lrow')) @
snake_aux m (Suc lrow')"
      by (subst snake_aux.simps) simp
    also have "snake_aux m (Suc lrow') = snake_list (odd lrow') (drop
```

```
(Suc (lrow' - lrow)) xss)"
      using True "1.prems" by (subst 1) (auto simp: Suc_diff_le)
    also have "(if even lrow' then row m lrow' else rev (row m lrow'))
@ ... =
                snake_list (even lrow') (drop (lrow' - lrow) xss)"
      using mat_list_rel_row[OF assms(1), of lrow'] "1.prems" True
      by (subst (2) Cons_nth_drop_Suc [symmetric]) (simp_all add: len)
    finally show ?thesis .
  qed
qed

lemma mat_list_rel_snake:
  assumes "mat_list_rel m xss"
  shows   "snake m = snake_list (even lrow) xss"
  using mat_list_rel_snake_aux[OF assms, of lrow] lrow_le_urow by (auto
simp: snake_def)

end
```

The final correctness theorem for shear sort on lists of lists:

```
theorem shearsort_list_correct:
  assumes "⋀xs. xs ∈ set xss ⟹ length xs = ncols"
  shows   "mset (concat (shearsort_list True xss)) = mset (concat xss)"
    and   "sorted (snake_list True (shearsort_list True xss))"
proof -
  define nrows where "nrows = length xss"
  interpret shearsort 0 nrows 0 ncols
    by standard auto
  define m where "m = mat_of_list xss"
  have m: "mat_list_rel m xss"
    unfolding m_def
    by (rule mat_list_rel_of_list) (use assms in ‹auto simp: nrows_def›)
  hence m': "mat_list_rel (shearsort m) (shearsort_list True xss)"
    using mat_list_rel_shearsort[of m xss] by simp
  hence "sum_list (map mset (shearsort_list True xss)) = mset_mat (shearsort
m)"
    by (rule mat_list_rel_mset [symmetric])
  also have "... = mset_mat m"
    by simp
  also have "... = mset (concat xss)"
    unfolding m_def
    by (subst mset_mat_of_list) (use assms in ‹auto simp: nrows_def mset_concat›)
  finally show "mset (concat (shearsort_list True xss)) = mset (concat
xss)"
    by (simp add: mset_concat)

  have "sorted (snake (shearsort m))"
    by (simp add: sorted_snake_iff snake_sorted_shearsort)
  also have "snake (shearsort m) = snake_list True (shearsort_list True
```

```
xss)"
    using mat_list_rel_snake[OF m'] by simp
  finally show "sorted (snake_list True (shearsort_list True xss))" .
qed

value "shearsort_list True [[5, 8, 2], [9, 1, 7], [3, 6, 4 :: int]]"

end
```

References

[1] S. Sen, I. D. Scherson, and A. Shamir. Shear sort: A true two-dimensional sorting techniques for VLSI networks. In *International Conference on Parallel Processing, ICPP'86, University Park, PA, USA, August 1986*, pages 903–908. IEEE Computer Society Press, 1986.