```
theory Example_Lomuto
  imports Hnr_Diff_Arr Hnr_Array  Definition_Utils "HOL-Library.Multiset" "HOL-Library.Rew
begin

definition swap :: "nat ⇒ nat ⇒ 'a list ⇒ 'a list" where
  "swap i j xs ≡ (xs[i := xs!j])[j := xs!i]"

fun partition :: "nat ⇒ nat ⇒ ('a::linorder) list ⇒ ('a list * nat)" where
  "partition i j xs = (if 1 < j then
      (if xs ! 0 < xs ! (j - 1)
       then partition (i - 1) (j - 1) (swap (i - 1) (j - 1) xs)
       else partition i (j - 1) xs)
    else (swap (i - 1) 0 xs, i - 1)
  )"

declare partition.simps[simp del]

abbreviation partition' where
  "partition' xs ≡ partition (length xs) (length xs) xs"

definition inv :: "nat ⇒ nat ⇒ ('a::linorder) list ⇒ bool" where
  "inv i j xs ≡
    let p = xs ! 0 in
    0 < length xs ∧
    0 < i ∧
    i ≤ length xs ∧
    j ≤ length xs ∧
    j ≤ i ∧
   (∀h ∈ set (drop i xs). p < h) ∧
   (∀l ∈ set (take (i - j) (drop j xs)). l ≤ p)"

definition is_valid_partition where
  "is_valid_partition ys m ≡ ∀l ∈ set (take m ys). ∀h ∈ set (drop m ys). l ≤ h"

lemma mset_swap' [simp]:"i < length xs ⟹ j < length xs ⟹ mset (swap i j xs) = mset xs"
  unfolding swap_def
  using mset_swap by auto

lemma swap_length [simp]: "length (swap i j xs) = length xs"
  unfolding swap_def
  by auto

lemma swap_pivot: "swap (Suc i) (Suc j) (p#xs) = p # (swap i j xs)"
  unfolding swap_def
  by auto

lemma swap_pivot_2: "0 < i ⟹ 0 < j ⟹ swap i j (p#xs) = p # (swap (i - 1) (j - 1) xs)"
  unfolding swap_def
  apply(cases i; cases j)
  by auto

lemma swap_nth_absorb: "n < i ⟹ n < j ⟹ swap i j xs ! n = xs ! n"
  unfolding swap_def
  by auto

lemma drop_swap: "n ≤ i ⟹ n ≤ j ⟹ drop n (swap i j xs) = swap (i - n) (j - n) (drop n
  unfolding swap_def
  by (metis drop_eq_Nil drop_update_swap le_add_diff_inverse linorder_le_cases list_update

lemma drop_swap': "⟦
  Suc 0 < j;
  xs ≠ [];
  i ≤ length xs;
```

```
  j ≤ i⟧ ⟹
  drop (i - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs) = xs ! (j - Suc 0) # (drop i xs)"
  unfolding swap_def
  by(smt (verit, best) Cons_nth_drop_Suc One_nat_def drop_upd_irrelevant dual_order.strict

lemma drop_swap'_2: "⟦
  Suc 0 < j;
  xs ! 0 < xs ! (j - Suc 0);
  xs ≠ [];
  i ≤ length xs;
  j ≤ i⟧ ⟹ drop (j - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs) = swap (i - 1 - (j - Suc 0
  using drop_swap[of "j - 1" "i - 1" "j - 1" xs]
  by auto

lemma test_2: "n < length xs ⟹ drop n xs = drop n xs ! 0 # drop (Suc n) xs"
  apply auto
  by (simp add: Cons_nth_drop_Suc)

lemma test_3: "⟦Suc 0 < j; i ≤ length xs; j ≤ i⟧
  ⟹ (drop (j - Suc 0) xs)[0 := xs ! (i - 1)] = xs ! (i - 1) # drop j xs"
  by (smt (verit, ccfv_SIG) Cons_nth_drop_Suc Suc_diff_le Suc_le_eq diff_Suc_Suc le_trans

lemma test_3_1: "⟦Suc 0 < j; j ≤ length xs⟧ ⟹ (drop (j - Suc 0) xs)[0 := y] = y # drop j
  by (metis dual_order.refl list_update_code(2) list_update_overwrite test_3)

lemma test_3_2: "(take (i - j) (drop (j - Suc 0) xs))[0 := xs ! (i - Suc 0)] = (take (i -
  by simp

lemma test_4: "⟦Suc 0 < j; i ≤ length xs; j ≤ i⟧
  ⟹ take (i - j) (drop (j - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs)) = take (i -j) (xs
  unfolding swap_def
  apply auto
  by (smt (z3) One_nat_def Suc_le_mono diff_Suc_Suc diff_self_eq_0 drop_update_swap dual_c

lemma test_5_1: "⟦l ∈L take (i - j) (xs ! (i - Suc 0) # drop j xs); l ≠ xs ! (i - Suc 0)⟧
  apply auto
  by (smt (verit) Suc_diff_Suc Suc_le_eq diff_is_0_eq less_or_eq_imp_le linorder_not_less

lemma test_5_2: "l ∈L take (i - Suc j) (drop j xs) ⟹ l ∈L take (i - j) (drop j xs)"
  by (meson Suc_n_not_le_n diff_le_mono2 nle_le set_take_subset_set_take subset_code(1))

lemma test_5: "⟦
  xs ≠ [];
  Suc 0 < j;
  i ≤ length xs;
  j ≤ i;
  i ≠ j;
  ∀l∈set (take (i - j) (drop j xs)). l ≤ xs ! 0;
  l ∈L take (i - j) (xs ! (i - Suc 0) # drop j xs);
  xs ! (i - Suc 0) ≤ xs ! 0
⟧ ⟹ (l::'a::linorder) ≤ xs ! 0"
  apply(cases "l = xs ! (i - Suc 0)")
   apply auto
  apply(drule test_5_1)
  by(auto simp: test_5_2)

lemma test_6_1: "j < i ⟹ i ≤ length xs ⟹ xs ! (i - Suc 0) ∈L drop j xs"
  by (metis Suc_leI Suc_le_mono Suc_pred in_set_drop_conv_nth less_imp_Suc_add nz_le_conv_

lemma test_6: "j < i ⟹ i ≤ length xs ⟹ xs ! (i - Suc 0) ∈L take (i - j) (drop j xs)"
  using test_6_1
  by (smt (verit, ccfv_SIG) diff_less drop_take le_SucE le_zero_eq length_take less_zeroE
```

```
lemma test_7: "⟦∀l∈set xs. P l; x ∈L xs⟧ ⟹ P x"
  by simp

lemma aux_1_1: "⟦
  Suc 0 < j;
  xs ! 0 < xs ! (j - Suc 0);
  xs ≠ [];
  i ≤ length xs;
  j ≤ i;
  ∀x∈set (drop i xs). xs ! 0 < x;
  x ∈L drop (i - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs)
⟧ ⟹ xs ! 0 < x"
  using drop_swap'[of j xs i]
  by auto

lemma aux_1_2: "⟦
  Suc 0 < j;
  xs ! 0 < xs ! (j - Suc 0);
  xs ≠ [];
  i ≤ length xs;
  j ≤ i;
  ∀l∈set (take (i - j) (drop j xs)). l ≤ xs ! 0;
  l ∈L take (i - j) (drop (j - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs))
⟧ ⟹ (l::'a::linorder) ≤ xs ! 0"
  apply(auto simp: test_4)
  apply(cases "i = j")
   apply auto
  apply(subgoal_tac "xs ! (i - Suc 0) ≤ xs ! 0")
  using test_5[of xs]
   apply simp
  using test_6[of j i xs]
  by simp

lemma aux_2_1:  "⟦l ∈L take n xs⟧
  ⟹  l ∈L take (Suc n) xs"
  apply(induction xs)
   apply auto
  by (metis lessI less_or_eq_imp_le set_ConsD set_take_subset_set_take subset_code(1) take

lemma aux_2_2: "0 < j ⟹ j ≤ length xs ⟹ drop (j - Suc 0) xs =  xs ! (j - Suc 0) # drop
  by (metis Cons_nth_drop_Suc Suc_pred nz_le_conv_less)

lemma aux_3_1: "⟦¬ Suc 0 < j; xs ≠ []; 0 < i; i ≤ length xs; l ∈L (take (i - Suc 0) xs)[0
   ⟹ l ∈L take (i - j) (drop j xs)"
  by (smt (verit, ccfv_threshold) Cons_nth_drop_Suc One_nat_def Suc_diff_Suc Suc_inject Su

lemma aux_3_2: "⟦
  ¬ Suc 0 < j;
  (xs::('a::linorder) list) ≠ []; 0 < i;
  i ≤ length xs;
  ∀h ∈set (drop i xs). xs ! 0 ≤ h
⟧ ⟹ ∀h ∈set  (drop (i - Suc 0) (xs[i - Suc 0 := xs ! 0, 0 := xs ! (i - Suc 0)])). xs ! 0
  apply(cases "i = 1")
   apply auto
  subgoal for x
    apply(cases "x = xs ! 0")
     apply auto
    by (metis Cons_nth_drop_Suc drop_0 length_pos_if_in_set set_ConsD)
  by (smt (verit, best) Nat.diff_diff_eq One_nat_def Suc_diff_Suc diff_diff_cancel drop_up

lemma partition: "inv i j xs ⟹ partition i j xs = (ys, m) ⟹ is_valid_partition ys m"
proof(induction i j xs arbitrary: ys m rule: partition.induct)
  case (1 i j xs)
```

```
  then have unfolded_partition: "(
        if Suc 0 < j
        then if xs ! 0 < xs ! (j - 1)
             then partition (i - 1) (j - 1) (swap (i - 1) (j - 1) xs)
             else partition i (j - 1) xs
        else (swap (i - 1) 0 xs, i - 1)) = (ys, m)"
    by(simp add: partition.simps)

  have recursive_branch_1:
    "⟦1 < j; xs ! 0 < xs ! (j - Suc 0)⟧
        ⟹  inv (i - Suc 0) (j - Suc 0) (swap (i - Suc 0) (j - Suc 0) xs)"
  using "1.prems"
  unfolding inv_def Let_def
  apply(auto simp: swap_nth_absorb)
  by(auto simp: aux_1_1 aux_1_2)

  have recursive_branch_2:
    "⟦1 < j; ¬ xs ! 0 < xs ! (j - Suc 0)⟧
        ⟹ inv i (j - Suc 0) xs"
  using "1.prems"
   unfolding inv_def Let_def
  apply auto
  subgoal for l
    apply(cases "l = xs ! (j - Suc 0)")
     apply(auto simp: aux_2_2)
    using take_Suc_Cons[of "i - j" "xs ! (j - Suc 0)"  "drop j xs"]
    by auto
  done

  have terminating_branch:
     "¬ Suc 0 < j ⟹ is_valid_partition (swap (i - Suc 0) 0 xs) (i - Suc 0)"
     using "1.prems"
    unfolding inv_def Let_def is_valid_partition_def
  apply(auto)
  unfolding swap_def
  subgoal for l h
  using aux_3_1[of j xs i l] aux_3_2[of j xs i]
  apply(auto)
  by (meson order_less_imp_le order_trans)
  done

  from unfolded_partition 1 recursive_branch_1 recursive_branch_2 terminating_branch
  show ?case
    by(auto split: if_splits)
qed

lemma inv: "inv (length (p#xs)) (length (p#xs)) (p#xs)"
  unfolding inv_def
  by auto

lemma partition':
  "partition' (p#xs) = (ys, m) ⟹ is_valid_partition ys m"
  using inv[of p xs] partition[of "length (p # xs)" "length (p # xs)" "p # xs" ys m]
  by auto

(* TODO: If c2 and c3 exchange then not linear anymore *)
definition swap_opt :: "nat ⇒ nat ⇒ 'a list ⇒ 'a list option" where
  "swap_opt i j xs = do {
      let c1 = xs!j;
      let c2 = xs!i;
      let c3 = xs[i := c1];
      let c4 = c3[j := c2];
      Some c4
  }"
```

```
lemma swap_opt_termination: "swap_opt i j xs = Some (swap i j xs)"
  unfolding swap_opt_def swap_def
  by auto

synth_definition swap_impl is [hnr_rule_diff_arr]:
  "hnr (master_assn' (insert (xs, xsi) F) * id_assn i ii * id_assn j ji)(¤:: ?'a Heap) ?Γ'
  unfolding swap_opt_def
  by hnr_diff_arr

synth_definition swap_impl_2 is [hnr_rule_arr]:
  "hnr (array_assn xs xsi * id_assn i ii * id_assn j ji)(¤:: ?'a Heap) ?Γ' (swap_opt i j x
  unfolding swap_opt_def
  by hnr_arr

definition partition_opt :: "nat × nat × ('a::linorder) list ⇒ (('a::linorder) list × na
  where
    "partition_opt ≡ option.fixp_fun (λpartition_opt p. case p of (i, j, xs) ⇒ do {
      let c1 = 1;
      let c2 = c1 < j;
      if c2 then do {
        let c99 = 0;
        let c3 = xs ! c99;
        let c4 = 1;
        let c5 = j - c4;
        let c6 = xs ! c5;
        let c7 = c3 < c6;
        if c7 then do {
           let c8 = 1;
           let c9 = i - c8;
           let c10 = 1;
           let c11 = j - c10;
           c12 ← swap_opt c9 c11 xs;
           let c13 = (c9, c11, c12);
           partition_opt c13
        }
        else do {
          let c14 = 1;
          let c15 = j - c14;
          let c16 = (i, c15, xs);
          partition_opt c16
        }
      }
      else do {
        let c17 = 1;
        let c18 = i - c17;
        let c19 = 0;
        c20 ← swap_opt c18 c19 xs;
        Some (c20, c18)
      }
    }
  )"

schematic_goal partition_opt_unfold: "partition_opt p ≡ ?v"
  apply(rule gen_code_thm_option_fixp[OF partition_opt_def])
  by(partial_function_mono)

lemma partition_opt_termination: "partition_opt (i, j, xs) = Some (partition i j xs)"
  apply(induction i j xs rule: partition.induct)
  apply(rewrite partition_opt_unfold)
  apply(rewrite partition.simps)
  by(auto simp: Let_def swap_opt_termination)

context
```

```
  fixes xsi :: "('a::{linorder,heap}) cell ref"
begin

synth_definition partition_impl is [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) F) * id_assn i ii * id_assn j ji)
    (⌑:: ?'a Heap)
    ?Γ'
    (partition_opt (i, j, xs))"
  unfolding partition_opt_def
  apply(hnr_recursion
          "(λF p pi.
                master_assn' (insert (snd(snd p), snd (snd pi)) F) *
                id_assn (fst p) (fst pi) *
                id_assn (fst (snd p)) (fst (snd pi)))"
          "(λF p pi r ri.
                master_assn' (insert (snd(snd p), snd (snd pi)) (insert (fst r, fst ri) F)
                id_assn (snd r) (snd ri) *
                id_assn (fst p) (fst pi) *
                id_assn (fst (snd p)) (fst (snd pi)) *
                true
                )"
          hnr_diff_arr_match_atom
        )
     apply hnr_diff_arr
                        apply(hnr_solve_recursive_call hnr_diff_arr_match_atom)
                        apply hnr_diff_arr
                        apply(hnr_solve_recursive_call hnr_diff_arr_match_atom)
  by hnr_diff_arr
  sorry

end

end
```