Schrödinger's hats

A puzzle about parities and permutations

Matthew Brecknell

March 26, 2017

Meet Schrödinger, who travels the world with an unusually clever clowder of n talking cats. In their latest show, the cats stand in a line. Schrödinger asks a volunteer (not a plant!) to take $n + 1$ hats, numbered zero to n, and randomly assign one to each cat, so that there is one spare. Each cat sees all of the hats in front of it, but not its own hat, nor those behind, nor the spare hat. The cats then take turns, each calling out a single number from the set $\{0..n\}$, without repeating any number previously called, and without any other communication. Although the first call is allowed to be wrong, the remaining cats always call out the numbers on their own hats.

1 Parity of a list permutation

Define the parity of a list *xs* as the evenness of the number of inversions. Count an inversion for every pair *i* and *j*, such that *i < j*, but *xs!i > xs!j*.

```
primrec
  parity :: "nat list ⇒ bool"
where
  "parity [] = True"
| "parity (x # ys) = (parity ys = even (length [y ← ys. x > y]))"
```

In a list that is sufficiently distinct, swapping any two elements inverts the *parity*.

```
lemma parity_swap_adj:
  "b ≠ c ⟹ parity (as @ b # c # ds) ⟷ ¬ parity (as @ c # b # ds)"
  by (induct as; simp; blast)

lemma parity_swap:
  assumes "b ≠ d ∧ b ∉ set cs ∧ d ∉ set cs"
  shows "parity (as @ b # cs @ d # es) ⟷ ¬ parity (as @ d # cs @ b # es)"
  using assms
  proof (induct cs arbitrary: as)
    case Nil thus ?case using parity_swap_adj[of b d as es] by simp
  next
    case (Cons c cs) show ?case
      using parity_swap_adj[of b c as "cs @ d # es"]
            parity_swap_adj[of d c as "cs @ b # es"]
            Cons(1)[where as="as @ [c]"] Cons(2)
      by simp
  qed
```

2 Solving the puzzle

2.1 Individual choice function

```
definition
  "candidates xs ≡ {0 .. 1 + length xs} - set xs"

definition
  choice :: "nat list ⇒ nat list ⇒ nat"
where
  "choice heard seen ≡
    case sorted_list_of_set (candidates (heard @ seen)) of
      [a,b] ⇒ if parity (a # heard @ b # seen) then b else a"
```

2.2 Group choice function

```
primrec
  choices' :: "nat list ⇒ nat list ⇒ nat list"
where
  "choices' heard [] = []"
| "choices' heard (_ # seen)
    = (let c = choice heard seen in c # choices' (heard @ [c]) seen)"

definition "choices ≡ choices' []"
```

2.3 Examples

```
definition "example_even ≡ [4,2,3,6,0,5]"
lemma "parity (1 # example_even)" by eval
lemma "choices example_even = [4,2,3,6,0,5]" by eval

definition "example_odd ≡ [4,0,3,6,2,5]"
lemma "¬ parity (1 # example_odd)" by eval
lemma "choices example_odd = [1,0,3,6,2,5]" by eval
```

2.4 Group choice does not cheat

```
lemma choices':
  assumes "i < length assigned"
  assumes "spoken = choices' heard assigned"
  shows "spoken ! i = choice (heard @ take i spoken) (drop (Suc i) assigned)"
  using assms proof (induct assigned arbitrary: i spoken heard)
    case Cons thus ?case by (cases i) (auto simp: Let_def)
  qed simp

lemma choices:
  assumes "i < length assigned"
  assumes "spoken = choices assigned"
  shows "spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
  using assms by (simp add: choices_def choices')
```

2.5 Group choice has the correct length

```
lemma choices'_length: "length (choices' heard assigned) = length assigned"
  by (induct assigned arbitrary: heard) (auto simp: Let_def)

lemma choices_length: "length (choices assigned) = length assigned"
  by (simp add: choices_def choices'_length)
```

2.6 Correctness of choice function

```
context
  fixes spare :: "nat"
  fixes assigned :: "nat list"
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
begin

lemma distinct: "distinct (spare # assigned)"
  apply (rule card_distinct)
  apply (subst assign)
  by auto

lemma distinct_pointwise:
  assumes "i < length assigned"
  shows "spare ≠ assigned ! i
          ∧ (∀ j < length assigned. i ≠ j ⟶ assigned ! i ≠ assigned ! j)"
  using assms distinct by (auto simp: nth_eq_iff_index_eq)

context
  fixes spoken :: "nat list"
  assumes spoken: "spoken = choices assigned"
begin

lemma spoken_length: "length spoken = length assigned"
  using choices_length spoken by simp

lemma spoken_choice:
  "i < length assigned ⟹ spoken ! i = choice (take i spoken) (drop (Suc i) assigned)"
  using choices spoken by simp

context
  assumes exists: "0 < length assigned"
  notes parity.simps(2) [simp del]
begin

lemma assigned_0:
  "assigned ! 0 # drop (Suc 0) assigned = assigned"
  using exists by (simp add: Cons_nth_drop_Suc)

lemma candidates_0:
  "candidates (drop (Suc 0) assigned) = {spare, assigned ! 0}"
  proof -
    have len: "1 + length (drop (Suc 0) assigned) = length assigned"
      using exists by simp
    have set: "set (drop (Suc 0) assigned) = {0..length assigned} - {spare, assigned ! 0}"
      using Diff_insert2 Diff_insert_absorb assign assigned_0 distinct
            distinct.simps(2) list.simps(15)
      by metis
    show ?thesis
      unfolding candidates_def len set
      unfolding Diff_Diff_Int subset_absorb_r
      unfolding assign[symmetric]
      using exists by auto
  qed

lemma spoken_0:
  "spoken ! 0 = (if parity (spare # assigned) then assigned ! 0 else spare)"
```

```
  unfolding spoken_choice[OF exists] choice_def take_0 append_Nil candidates_0
  using parity_swap_adj[where as="[]"] assigned_0 distinct_pointwise[OF exists]
  by (cases "assigned ! 0 < spare") auto

context
  fixes rejected :: "nat"
  fixes initial_order :: "nat list"
  assumes rejected: "rejected = (if parity (spare # assigned) then spare else assigned ! 0)"
  assumes initial_order: "initial_order = rejected # spoken ! 0 # drop (Suc 0) assigned"
begin

lemma parity_initial: "parity initial_order"
  unfolding initial_order spoken_0 rejected
  using parity_swap_adj[of "assigned ! 0" "spare" "[]"]
        distinct_pointwise[OF exists] assigned_0
  by auto

lemma distinct_initial: "distinct initial_order"
  unfolding initial_order rejected spoken_0
  using assigned_0 distinct distinct_length_2_or_more
  by (metis (full_types))

lemma set_initial: "set initial_order = {0..length assigned}"
  unfolding initial_order assign[symmetric] rejected spoken_0
  using arg_cong[where f=set, OF assigned_0, symmetric]
  by auto

lemma spoken_correct:
  "i ∈ {1 ..< length assigned} ⟹ spoken ! i = assigned ! i"
  proof (induction i rule: nat_less_induct)
    case (1 i)

    have
      LB: "0 < i" and UB: "i < length assigned" and US: "i < length spoken" and
      IH: "∀ j ∈ {1 ..< i}. spoken ! j = assigned ! j"
      using 1 spoken_length by auto

    let ?heard = "take i spoken"
    let ?seen  = "drop (Suc i) assigned"

    have heard: "?heard = spoken ! 0 # map (op ! assigned) [Suc 0 ..< i]"
      using IH take_map_nth[OF less_imp_le, OF US] range_extract_head[OF LB] by auto

    let ?my_order = "rejected # ?heard @ assigned ! i # ?seen"

    have initial_order: "?my_order = initial_order"
      unfolding initial_order heard
      apply (simp add: UB Cons_nth_drop_Suc)
      apply (subst drop_map_nth[OF less_imp_le_nat, OF UB])
      apply (subst drop_map_nth[OF Suc_leI[OF exists]])
      apply (subst map_append[symmetric])
      apply (rule arg_cong[where f="map _"])
      apply (rule range_app)
      using UB LB less_imp_le Suc_le_eq by auto

    have distinct_my_order: "distinct ?my_order"
      using distinct_initial initial_order by simp
```

```
    have set_my_order: "set ?my_order = {0..length assigned}"
      using set_initial initial_order by simp

    have set: "set (?heard @ ?seen) = {0..length assigned} - {rejected, assigned ! i}"
      apply (rule subset_minusI)
      using distinct_my_order set_my_order by auto

    have len: "1 + length (?heard @ ?seen) = length assigned"
      using LB UB heard by simp

    have candidates: "candidates (?heard @ ?seen) = {rejected, assigned ! i}"
      unfolding candidates_def len set
      unfolding Diff_Diff_Int subset_absorb_r
      unfolding assign[symmetric]
      unfolding rejected
      using UB exists by auto

    show ?case
      apply (simp only: spoken_choice[OF UB] choice_def candidates)
      apply (subst sorted_list_of_set_distinct_pair)
       using distinct_my_order apply auto[1]
      apply (cases "assigned ! i < rejected"; clarsimp)
       apply (subst (asm) parity_swap[of _ _ _ "[]", simplified])
        apply (simp add: distinct_my_order[simplified])
       unfolding initial_order
       using parity_initial
       by auto
  qed

end
end
end

lemma choices_correct:
  "i ∈ {1 ..< length assigned} ⟹ choices assigned ! i = assigned ! i"
  apply (rule spoken_correct) by auto

lemma choices_distinct: "distinct (choices assigned)"
  proof (cases "0 < length assigned")
    case True show ?thesis
    apply (clarsimp simp: distinct_conv_nth_less choices_length)
    apply (case_tac "i = 0")
    using True choices_correct spoken_0[OF _ True] distinct_pointwise
    by (auto split: if_splits)
  next
    case False thus ?thesis using choices_length[of assigned] by simp
  qed

end
```