Schrödinger's hats

A puzzle about parities and permutations

Matthew Brecknell

April 5, 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 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 Introduction

In this document, we will figure out how the cats do this. We'll start with some informal analysis, deriving the solution by a series of small logical steps. Once we've identified the key ingredient of the solution, we'll turn to formal proof in Isabelle/HOL, ultimately showing that the method always works.

Along the way, we'll rediscover some properties of permutation groups, and we'll look at some of the basic techniques of formal mathematical proof.

1.1 Initial observations

We can begin to structure our thinking by making some initial observations.

1.1.1 Order of calls

The order in which the cats call out numbers is not specified, but there is a clear ordering of information available. The cat at the rear of the line should go first, because it initially has the most information: it can see all hats except its own and the spare hat. The cat second from the rear should go next: it can see all hats but three, and has heard the rearmost cat's call. Likewise, each subsequent cat should wait until all the cats behind it have made their calls.

1.1.2 Limited information

The cats receive limited information: they see the hats in front of them, and they hear the calls made by cats behind. In particular, no cat has any information about the rearmost cat's number. It might as well be a probability density function, which only collapses into a particular number when Scrödinger reveals all the hats at the end of the performance.

The cats must assume the worst: that the rearmost cat got it wrong. And this means that all the other cats must get it right!

Counterintuitively, this makes our job easier: when considering how some particular cat makes its choice, we can assume that all the cats behind it, except the rearmost, have already made the right choice.

1.1.3 Candidate selection

According to the rules, no cat may repeat a number already called by another cat behind it. But we can also say that no cat may call a number that it can see ahead of it. If it did, there would be at least two incorrect calls.

To see this, suppose some cat Felix called out a number that it saw on some other cat Tigger who is ahead of Felix. Hat numbers are unique, so Felix's number must be different from Tigger's, so Felix's call is wrong. But Tigger may not repeat the number that Felix called, so Tigger is also wrong.

2 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 of indices *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
```

3 Solving the puzzle

3.1 Individual choice function

Given a list of all hat numbers either *seen* or *heard*, we can reconstruct the set of all hat numbers from the length of that list. Excluding the members from the

```
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"
```

3.2 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' []"
```

3.3 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
```

3.4 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')
```

3.5 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)
```

3.6 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
```