Schrödinger's hats

A puzzle about parities and permutations

Matthew Brecknell

April 14, 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 $\{i \mid 0 \leq i \leq n\}$, 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 article, we will figure out how the cats do this. We'll start with some informal analysis, deriving the solution by repeatedly asking ourselves the question: if there is a solution, what must it look like? 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 a fundamental property of permutation groups, and we'll look at some of the basic techniques of formal mathematical proof.

2 Initial observations

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

2.1 Ordering the calls

The rules require each cat to make exactly one call, but do not specify the order in which they do this. We can see that the order we choose affects the distribution of information:

- Visible information remains constant over time, but cats towards the rear see more than cats towards the front.
- Audible information increases over time, but at any particular point in time, all cats have heard the same things.

We observe that the cats can only ever communicate information *forwards*, never backwards:

- When a cat makes a call, all of the information available to it is already known to all the cats behind it. Therefore, cats towards the rear can never learn anything from the choices made by cats towards the front.
- However, cats towards the front *can* learn things from choices made by cats towards the rear, because those choices might encode knowledge of hats which are not visible from the front.

We propose that the cats should take turns from the rearmost towards the front, ensuring that:

- The cat making the choice is always the one with the most information.
- Every call but the last has the potential to communicate useful information.

We won't attempt to prove that this ordering is necessary to solve the puzzle. However, by choosing an order, we drastically reduce the space of possible solutions we consider, so we should at least have an informal justificaton for the order we choose.

2.2 Limited information

Each cat sees the hats in front of it, and hears the calls made by those behind it, but otherwise receives no information. In particular, no cat knows the rearmost cat's number. Until Schrödinger reveals it at the end of the performance, it could be either of the two hats that are invisible to all cats.

To guarantee success, the cats must therefore assume the worst: that the rearmost cat got it wrong. But this means that all the other cats *must* get it right!

Surprisingly, knowing which cats must get it right makes our job easier. When considering how some cat k makes its choice, we can assume that all the cats $\{i \mid 0 < i < k\}$, i.e. those behind it, except the rearmost, have already made the right choices.

This might seem like circular reasoning, but it's not. In principle, we build up what we know from the rearmost cat, one cat at a time towards the front, using what we've already shown about cats $\{i \mid 0 \leq i < k\}$ when we're proving that cat k makes the right choice. Mathematical induction merely says that if all steps are alike, we can take them all at once by considering an arbitrary cat k, and assuming we've already considered all the cats $\{i \mid 0 \leq i < k\}$ behind it.

We can formalise this with a proof:

```
lemma spoken_correct_induct:
  assumes "⋀k. k ∈ {1 ..< length assigned}
               ⟹ ∀i ∈ {1 ..< k}. spoken ! i = assigned ! i
               ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
  by (induct k rule: nat_less_induct)
     (meson assms atLeastLessThan_iff less_imp_le less_le_trans)
```

2.3 Candidate selection

According to the rules, no cat may repeat a number already called by another cat behind it. With a little thought, 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 k called out a number that it saw on the hat of t who is in front of k. Hat numbers are unique, so k's number must be different from t's, and therefore k's call is wrong. But t may not repeat the number that k called, so t is also wrong.

This means that each cat k has to choose between exactly two candidate numbers: those left over after removing all the numbers it has seen and heard. Since none of the cats $\{i \mid 0 \leq i < k\}$ previously called k's number, k's own number is one of those candidates. Taking into account our assumption that all those $\{i \mid 0 < i < k\}$ except the rearmost called their own numbers, we can also say that the other candidate will be the same number which the rearmost cat chose *not* to call.

To solve the puzzle, we therefore just need to ensure that every cat k rejects the same number that the rearmost cat rejected.

This bears repeating, lest we miss its significance!

Working from the rear to the front, if each cat rejects all the numbers it has heard and seen, and of the remaining numbers, *additionally rejects the same number as the rearmost cat*, then the puzzle is solved.

3 The choice function

We'll now derive the method the cats use to ensure all of them reject the same hat. We assume that the cats have agreed beforehand on the algorithm each cat will *individually* apply, and have convinced themselves that the agreed algorithm will bring them *collective* success, no matter how the hats are assigned to them.

We can represent the individual algorithm as a function of the information an individual cat receives. We don't yet know its definition, but we can write its type:

```
type_synonym choice = "nat list ⇒ nat list ⇒ nat"
```

That is, when it is cat k's turn, we give the list of calls `heard` from behind, and the list of hats `seen` in front, both in order, and the function returns the number the cat should call. The lengths of the lists give the position of the cat in the line, so we can use a single function to represent the choices of all cats, without loss of generality.

We can partially implement the `choice` function, first calculating the `candidates` from which we must choose, by eliminating all those `heard` and `seen`. We defer the remaining work to a `classifier` function, which we'll take as a parameter until we know how to implement it:

```
type_synonym classifier = "nat ⇒ nat list ⇒ nat ⇒ nat list ⇒ bool"

definition
  candidates :: "nat list ⇒ nat set"
where
  "candidates xs ≡ {0 .. 1 + length xs} - set xs"

definition
  choice_of_classifier :: "classifier ⇒ choice"
where
  "choice_of_classifier classify heard seen ≡
    case sorted_list_of_set (candidates (heard @ seen)) of
      [a,b] ⇒ if (classify a heard b seen) then b else a"
```

The `classifier` receives the original arguments `heard` and `seen`, as well as the two `candidates`, `a` and `b`.

The order in which we pass these arguments is suggestive of one of the two possible orderings of the full set of hats consistent with what is `heard` and `seen` by the cat making the choice. The cat imagines hat `b` on its head, between those it has `heard` and `seen`, and imagines hat `a` placed on the floor behind the rearmost cat.

The classifier then returns a `bool` that indicates whether the given ordering should be accepted or rejected. If accepted, the cat calls the hat it had imagined on its own head. If rejected, it calls the other.

Since there must always be exactly one correct call, we require that the classifier accepts an ordering if and only if it would reject the alternative:

```
definition
  classifier_correct :: "classifier ⇒ bool"
where
  "classifier_correct classify ≡
    ∀a heard b seen. distinct (a # heard @ b # seen) ⟶
      (classify a heard b seen ⟷ ¬ classify b heard a seen)"
```

This means that we can say which is the accepted ordering, regardless of which ordering we actually passed to the classifier.

Although the refinement from choice to classifier might seem trivial, it gives us a different way of looking at the problem. Instead of asking what is the correct hat number, which is different for each cat, we can consider orderings of the complete set of hats, and whether or not those orderings are consistent with the information available to *all* of the cats.

In particular, we notice that for all but the rearmost cat to choose the correct hats, the accepted orderings must be the same for all cats. This is because the correct call for any cat must be what was `seen` by all cats to the rear, and will also be `heard` by all cats towards the front.

Surprisingly, this is true even for the rearmost cat! The only thing special about the rearmost cat is that its assigned number is irrelevant. The task of the rearmost cat is not to guess its assigned number, but to inform the other cats which ordering is both consistent with the information they will have, and also accepted by the classifier.

We can write down the required property that the accepted orderings must be consistent:

```
definition
  classifier_well_behaved :: "classifier ⇒ bool"
where
  "classifier_well_behaved classify ≡
    ∀a heard b seen a' heard' b' seen'.
      a # heard @ b # seen = a' # heard' @ b' # seen'
        ⟶ classify a heard b seen = classify a' heard' b' seen'"
```

So far, we have investigated some properties that a `classifier` must have, but have not thrown away any information. The classifier is given everything known to each cat. The lengths of the arguments `heard` and `seen` encode the cat's position in the line, so we even allow the classifier to behave differently for each cat.

But the property `classifier_well_behaved` suggests that the position in the line is redundant, and we can collapse the classifier's arguments into a single list. Given an existing classifier, we can derive such a function:

```
type_synonym parity = "nat list ⇒ bool"

definition
  parity_of_classifier :: "classifier ⇒ parity"
where
  "parity_of_classifier classify hats ≡
    case hats of a # b # seen ⇒ classify a [] b seen"
```

We can prove that all the behaviours of a well-behaved classifier are captured by the *parity* function derived from it. This confirms that we have not thrown away anything:

```
lemma parity_of_classifier_complete:
  "classifier_well_behaved classify ⟹
    ∀a heard b seen.
      classify a heard b seen = parity_of_classifier classify (a # heard @ b # seen)"
  unfolding classifier_well_behaved_def parity_of_classifier_def
  by (elim all_forward; case_tac heard) auto
```

We can also restate the required *classifier_correct* property in terms of *parity* functions, and prove a suitable equivalence:

```
definition
  parity_correct :: "parity ⇒ bool"
where
  "parity_correct parity ≡
    ∀a heard b seen. distinct (a # heard @ b # seen) ⟶
      (parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen))"

lemma parity_correct_classifier_correct:
  assumes "classifier_well_behaved classify"
  shows "classifier_correct classify ⟷ parity_correct (parity_of_classifier classify)"
  unfolding classifier_correct_def parity_correct_def
  apply (subst parity_of_classifier_complete[rule_format, OF assms])+
  by (rule refl)
```

Now that we're confident that a *parity* function is sufficient, so we can rephrase the *choice* function in terms of a *parity* function, and forget about *classifier* functions altogether:

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

lemma choice_of_parity_choice_of_classifier:
  assumes "classifier_well_behaved classify"
  shows "choice_of_parity (parity_of_classifier classify) = choice_of_classifier classify"
  unfolding choice_of_parity_def choice_of_classifier_def
  apply (subst parity_of_classifier_complete[rule_format, OF assms])
  by (rule refl)
```

Based on the informal derivation so far, our claim is that any function satisfying *parity_correct* is sufficient to solve the puzzle. Next, we'll derive an implementation of such a *parity* function, and then formally prove that it solves the puzzle.

4 The parity function

5 Proof

5.1 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 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) auto


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
```

5.2 Solving the puzzle

5.2.1 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"
```

5.2.2 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' []"
```

5.2.3 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
```

5.2.4 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')
```

5.2.5 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)
```

5.3 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
```