Schrödinger's hats

A puzzle about parities and permutations

Matthew Brecknell

April 22, 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 saying a single number from the set $\{i \mid 0 \leq i \leq n\}$, without repeating any number said previously, and without any other communication. The cats are allowed a single incorrect guess, but otherwise every cat must say the number on its own hat.

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 asking what properties it must have, and refining these properties until we can realise them with a concrete algorithm. We'll also develop a formal proof that the method always works, using Isabelle/HOL.

Along the way, we'll rediscover a fundamental property of permutation groups, and we'll gain some familiarity with some basic methods of formal mathematical proof.

Although this is not intended as an Isabelle/HOL tutorial, we hope that it is accessible to readers with no formal theorem proving experience. We do assume familiarity with some fundamentals of functional programming and classical logic. We won't explain the detailed steps required to prove each lemma, but we will explain how each lemma fits into the overall progression of the proof.

For the informal analysis, we'll work from the top down, gradually unfolding the solution. Each refinement will be small, and may seem like it is the only possible step. As we do this, we'll use Isabelle/HOL to make each step of the analysis more precise, and to check that our reasoning is sound.

However, there's a problem with this approach: our proof is inherently bottom up, building from the solution we ultimately identify, to a theorem that it solves the puzzle. We do not attempt to show that our solution is the *only* possible solution, although our informal analysis suggests that it is[1].

To develop the proof as we work top down, we need a way to invert the proof. We'll do this by temporarily *assuming* things we believe must be true for the puzzle to have a solution, but which we don't yet know how to prove. But we'll typically need to carry these assumptions through many lemmas. So, to avoid repeating assumptions, we'll use the **locale** mechanism of Isabelle/HOL to create named bundles of assumptions. Later, we'll discharge our assumptions using locale *interpretataion*.

[1] at least, up to some fairly strong equivalence.

Our first locale, `hats`, describes the basic setup of the puzzle:[2]

```
locale hats =
  — the spare hat
  fixes spare :: "nat"
  — hats assigned to cats, in order from back to front
  fixes assigned :: "nat list"
  — the set of all hat numbers
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
```

The `hats` locale takes two *parameters* introduced with **fixes** declarations:

- `spare`, of type `nat`, represents the number of the spare hat,
- `assigned`, of type `nat list`,[3] represents the hats assigned to cats, in order from back to front.

By *parameter*, we mean a placeholder for an arbitrary value of the given type. It is bound, or fixed, to the locale, so every mention of it in the locale context refers to the same hypothetical value. The **fixes** declaration does not specify *which* value, although subsequent locale assumptions may have the effect of restricting the possible values of parameters.

The `hats` locale also has an **assumes** declaration, which introduces an assumption named `assign`. It asserts that if we take the set of all hats, including the `spare` and `assigned` hats,[4] then we have the set of natural numers from `0` up to the number of `assigned` hats, inclusive. This specifies the possible hat numbers, but because we use unordered sets, it does not say anything about the order of `assigned` hats, nor which is the `spare` hat.

We can now prove lemmas in the context of the `hats` locale, which means that these lemmas can talk about the `spare` and `assigned` hats, and implicitly make the `assign` assumption.

For example, from this assumption, we can show that the hats all have distinct numbers:

```
lemma (in hats) "distinct (spare # assigned)"
  proof -
    — We start by restating our locale assumption.
    have "set (spare # assigned) = {0 .. length assigned}"
      by (rule assign)
    — We apply the card (set cardinality) function to both sides.
    hence "card (set (spare # assigned)) = card {0 .. length assigned}"
      by (rule arg_cong[where f=card])
    — We substitute an equivalent right-hand side, using built-in simplifications.
    hence "card (set (spare # assigned)) = 1 + length assigned"
      by simp
    — We substitute another right-hand side.
    hence "card (set (spare # assigned)) = length (spare # assigned)"
      by simp
    — The library fact card_distinct says a list is distinct if its length equals the cardinality of its set.
    thus "distinct (spare # assigned)"
      by (rule card_distinct)
  qed
```

The above proof contains much more detail than Isabelle/HOL requires. In the rest of this article,

[2] When reading Isabelle/HOL, you may ignore double quotes. They are there for technical reasons which have very little to do with the specification or proof you are reading.

[3] In Isabelle/HOL, type constructor application is written right-to-left, so a `nat list` is a list of natural numbers.

[4] The `#` constructor builds a new list from an existing element and list; and the `set` function converts a `list` to an unordered `set` type.

we'll write much terser individual proofs, since we want to focus on the higher-level development.

For example, we could shorten this proof as follows:

```
lemma (in hats) distinct_hats: "distinct (spare # assigned)"
  by (rule card_distinct, subst assign, simp)
```

We've also given the lemma a name, `distinct_hats`, so we can refer to the proven fact later.

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 accumulates 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 chooses a number, 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.
- We maximise the amount each cat can learn before it makes its choice.

We'll use another locale, and some definitions in the locale context, to describe the information flow:

```
locale cats = hats +
  — numbers spoken by cats, in order from back to front
  fixes spoken :: "nat list"
  — each cat speaks exactly once
  assumes length: "length spoken = length assigned"

definition (in cats) "heard k ≡ take k spoken"
definition (in cats) "seen k ≡ drop (Suc k) assigned"
```

Informally, the declaration says that there is a list of numbers `spoken` by the cats, which is just as long as the list of `assigned` hats. We define functions `heard` and `seen`, such that `heard k` and `seen k` are the lists of numbers heard and seen by cat `k`.

The only remarkable thing about the `cats` locale is that it *extends* the `hats` locale. This means that the **fixes** and **assumes** declarations from the `hats` locale become available in the context of the `cats` locale.

Lemmas proved in the `hats` locale also become available in the `cats` locale, whether they were proved before or after the `cats` locale declaration.

Why did we not make these **fixes** and **assumes** declarations in the `hats` locale? Eventually, we want to discharge the `length` assumption, but we can never discharge the `assign` assumption. As we'll see later, this means we need a separate locale.

2.2 The role of the rearmost cat

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!

The role of the rearmost cat is therefore not to try to guess his own hat, but to pass the right information to the other cats.

2.3 Reasoning by induction

Knowing which cats must get it right makes our job easier, since we don't need to keep track of whether the cats have used up their free pass. 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 an arbitrary number of 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'll use a locale to package up the so-called *induction hypothesis*. That is, we'll fix some cat `k`, which is not the rearmost cat, and assume that all the cats behind it, except the rearmost cat, said the correct number:[5]

```
locale cat_k = cats +
  fixes k :: "nat"
  assumes k_min: "0 < k"
  assumes k_max: "k < length assigned"
  assumes IH: "∀ i ∈ {1 ..< k}. spoken ! i = assigned ! i"
```

Using this, we can already formalise the induction argument:

```
lemma (in cats) cat_k_induct:
  assumes "⋀k. cat_k spare assigned spoken k ⟹ spoken ! k = assigned ! k"
  shows "k ∈ {1 ..< length assigned} ⟹ spoken ! k = assigned ! k"
  apply (induct k rule: nat_less_induct)
  apply (rule assms)
  apply (unfold_locales)
  by auto
```

[5] Infix operator `!` retrieves the `nth` element from a `list`; and `{a..<b}` is the set $\{n \mid a \leq n < b\}$.

This says that, in the *cats* locale, if any cat *k* satisfying *cat_k* says the correct number, then every cat except the rearmost says the correct number. We more or less get the induction hypothesis for free.

Note the keywords **assumes** and **shows** in the **lemma** statement. The first allows us to make additional assumptions for this lemma. The second introduces the thing we want to prove from the assumptions. If we have no local **assumes** declarations, we can omit the **shows** keyword, as we did in *distict_hats*.

In *cat_k_induct*, we've slightly abused the locale mechanism, by using the *cat_k* locale as a logical predicate, applied to some arguments. We only do this a couple of times, but it saves us from having to repeat the induction hypothesis many times.

As an example of something we can prove *within* the *cat_k* locale, we show that the tail of *heard k* can be rewritten in terms of the *assigned* hats:

```
lemma (in cat_k) k_max_spoken: "k < length spoken"
  using k_max length by simp

lemma (in cat_k) heard_k:
  "heard k = spoken ! 0 # map (op ! assigned) [Suc 0 ..< k]"
  using heard_def[of k] IH
        take_map_nth[OF less_imp_le, OF k_max_spoken]
        range_extract_head[OF k_min]
  by auto
```

2.4 Candidate selection

According to the rules, no cat may repeat a number already said 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 said 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 choice is wrong. But t may not repeat the number that k said, so t is also wrong.

Each cat k therefore has to choose between exactly two candidate hats: those left over after excluding all the numbers it has seen and heard:

```
definition
  candidates_excluding :: "nat list ⇒ nat list ⇒ nat set"
where
  "candidates_excluding heard seen ≡
    let excluded = heard @ seen in {0 .. 1 + length excluded} - set excluded"

definition (in cats)
  "candidates k ≡ candidates_excluding (heard k) (seen k)"

lemma (in cats) candidates_i:
  fixes a b i
  defines v: "view ≡ (a # heard i @ b # seen i)"
  assumes i: "i < length assigned"
  assumes d: "distinct view"
  assumes s: "set view = {0..length assigned}"
  shows "candidates i = {a,b}"
  proof -
    let ?excluded = "heard i @ seen i"
    have len: "1 + length ?excluded = length assigned"
```

```
    unfolding heard_def seen_def using i length by auto
  have set: "set ?excluded = {0..length assigned} - {a,b}"
    apply (rule subset_minusI)
    using s d unfolding v by auto
  show ?thesis
    unfolding candidates_def candidates_excluding_def Let_def
    unfolding len set
    unfolding Diff_Diff_Int subset_absorb_r
    using s unfolding v
    by auto
qed
```

Since none of the cats $\{i \mid 0 \leq i < k\}$ previously said 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 said 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.

To formalise this, we'll first prove that the `candidates` for the rearmost cat are as we expect:

```
locale cat_0 = cats +
  assumes exists_0: "0 < length assigned"

sublocale cat_k < cat_0
  apply unfold_locales
  using k_min k_max
  by auto

lemma (in cat_0) assigned_0: "assigned ! 0 # drop (Suc 0) assigned = assigned"
  using Cons_nth_drop_Suc[OF exists_0] by simp

lemma (in cat_0) candidates_0: "candidates 0 = {spare, assigned ! 0}"
  apply (rule candidates_i[OF exists_0])
  using distinct_hats assign unfolding heard_def seen_def assigned_0
  by auto
```

We define the `rejected` has as whichever of those the rearmost cat does not choose:

```
definition (in cat_0)
  "rejected ≡ if spoken ! 0 = spare then assigned ! 0 else spare"
```

We can try to prove a similar theorem for cat k, but we discover that we need some additional assumptions that the expected `candidates`, together with what cat k has `heard` and `seen`, constitute the complete set of hats.

We express these assumptions with an abbreviation `view_k`. We'll discharge the assumptions by proving that this is an ordering of the complete set of hats.

We'll prove the assumptions by appealing to two other orderings of the complete set of hats. We'll use `view_0` as a step towards `view_n`. The latter is very close to the ordering used in the original specification of `hats`, so we can easily prove the relevant properties of `view_n`

```
abbreviation (in cat_0) (input) "view_n ≡ spare # assigned ! 0 # seen 0"
abbreviation (in cat_0) (input) "view_0 ≡ rejected # spoken ! 0 # seen 0"
```

```
abbreviation (in cat_k) (input) "view_k ≡ rejected # heard k @ assigned ! k # seen k"

lemma (in cat_0)
  distinct_n: "distinct view_n" and
  set_n: "set view_n = {0..length assigned}"
  using distinct_hats assign
  unfolding seen_def assigned_0
  by auto
```

Ordering `view_0` is the same as `view_k`, but seen from the rearmost cat's perspective. We prove they are equal:

```
lemmas (in cat_k) drop_maps =
  drop_map_nth[OF less_imp_le_nat, OF k_max]
  drop_map_nth[OF Suc_leI[OF exists_0]]

lemma (in cat_k) view_eq: "view_0 = view_k"
  unfolding heard_k seen_def
  apply (simp add: k_max Cons_nth_drop_Suc drop_maps)
  apply (subst map_append[symmetric])
  apply (rule arg_cong[where f="map _"])
  apply (rule range_app[symmetric])
  using k_max k_min less_imp_le Suc_le_eq by auto
```

Now, to prove the relevant properties about `view_k`, we just need to prove them for `view_0`. But to do that, we need to know something about `spoken ! 0`. We haven't yet figured out how that choice is made, so we'll just assume it's one of the `candidates`:

```
locale cat_0_spoken = cat_0 +
  assumes spoken_candidate_0: "spoken ! 0 ∈ candidates 0"

lemma (in cat_0_spoken)
  distinct_0: "distinct view_0" and
  set_0: "set view_0 = {0..length assigned}"
  using spoken_candidate_0 distinct_n set_n
  unfolding candidates_0 rejected_def
  by fastforce+
```

Finally, we can prove the properties we wanted for `view_k`, and use them to discharge the assumptions of `candidates_k`.

```
locale cat_k_view = cat_k + cat_0_spoken

lemma (in cat_k_view)
  distinct_k: "distinct view_k" and
  set_k: "set view_k = {0..length assigned}"
  using distinct_0 set_0 view_eq
  by auto

lemma (in cat_k_view) candidates_k: "candidates k = {rejected, assigned ! k}"
  using candidates_i[OF k_max] distinct_k set_k by simp
```

If we additionally assumed that cat k chooses one of it's `candidates`, but somehow avoids the `rejected` hat, it trivially follows that cat k chooses its `assigned` hat.

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`, and deferring 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"

locale classifier =
  fixes classify :: "classifier"

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

primrec (in classifier)
  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 (in classifier) "choices ≡ choices' []"

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

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

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

The order in which we pass arguments to the `classifier` 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. We imagine hat `b` on the cat's head, between those it has `heard` and `seen`, and imagine hat `a` placed on the floor behind the rearmost cat, where no cat can see it.

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:

```
locale classifier_swap = classifier +
  assumes classifier_swap:
    "⋀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 its a small refinement from choice to classifier, 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:

```
locale classifier_consistent = classifier_swap +
  assumes classifier_consistent:
    "⋀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.

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

locale parity_classifier =
  fixes parity :: "parity"
  assumes parity_swap_0_i:
    "⋀a heard b seen.
      distinct (a # heard @ b # seen) ⟹
        parity (a # heard @ b # seen) ⟷ ¬ parity (b # heard @ a # seen)"

sublocale parity_classifier < classifier_consistent "λa heard b seen. parity (a # heard @ b # seen)"
  apply (unfold_locales)
  apply (erule parity_swap_0_i)
  by auto
```

Based on the informal derivation so far, our claim is that any function satisfying *parity_correct* is sufficient to solve the puzzle. Let's first prove this is the case, and then finally, we'll derive a *parity* function.

```
locale hats_parity = hats + parity_classifier

sublocale hats_parity < cats spare assigned "choices assigned"
  using choices_length by unfold_locales

locale cat_0_parity = hats_parity spare assigned parity
  + cat_0 spare assigned "choices assigned"
  for spare assigned parity

locale cat_k_parity = cat_0_parity spare assigned parity
  + cat_k spare assigned "choices assigned" k
  for spare assigned parity k

lemma (in cat_0_parity) candidates_excluding_0:
  "candidates_excluding [] (seen 0) = {spare, assigned ! 0}"
  using candidates_0 unfolding candidates_def heard_def take_0 by simp

lemma (in cat_0_parity) choices_0: "choices assigned ! 0 = choice [] (seen 0)"
  using choices[OF exists_0] unfolding seen_def by simp

lemma (in cat_0_parity) parity_swap_0:
  "parity (spare # assigned ! 0 # seen 0) ⟷ ¬ parity (assigned ! 0 # spare # seen 0)"
  using parity_swap_0_i[of spare "[]"] distinct_n by simp

lemma (in cat_0_parity) parity_0: "parity view_0"
  using distinct_n parity_swap_0
  unfolding choices_0 choice_def candidates_excluding_0 rejected_def
  by auto
```

```
lemma (in cat_k_parity) parity_k: "parity view_k"
  using parity_0 view_eq by simp

lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_n then assigned ! 0 else spare)"
  using distinct_n parity_swap_0
  unfolding choices_0 choice_def candidates_excluding_0
  by (subst sorted_list_of_set_distinct_pair) auto

sublocale cat_k_parity < cat_k_view spare assigned "choices assigned" k
  apply unfold_locales
  unfolding choice_0 candidates_0
  by simp

lemma (in cat_k_parity) choices_k:
  "choices assigned ! k = choice (heard k) (seen k)"
  unfolding heard_def seen_def using choices[OF k_max] by simp

lemma (in cat_k_parity) candidates_excluding_k:
  "candidates_excluding (heard k) (seen k) = {rejected, assigned ! k}"
  using candidates_k unfolding candidates_def by simp

lemma (in cat_k_parity) choice_k: "choices assigned ! k = assigned ! k"
  using parity_swap_0_i[OF distinct_k] distinct_k parity_k
  unfolding choices_k choice_def candidates_excluding_k
  by (subst sorted_list_of_set_distinct_pair) auto

lemma (in hats_parity) cat_k_cat_k_parity:
  assumes "cat_k spare assigned (choices assigned) k"
  shows "cat_k_parity spare assigned parity k"
  proof -
    interpret cat_k spare assigned "choices assigned" k using assms by simp
    show ?thesis by unfold_locales
  qed

lemma (in hats_parity) choices_correct:
  "k ∈ {1..<length assigned} ⟹ choices assigned ! k = assigned ! k"
  by (rule cat_k_induct[OF cat_k_parity.choice_k, OF cat_k_cat_k_parity])

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

lemma (in hats_parity) choices_distinct: "distinct (choices assigned)"
  proof (cases "0 < length assigned")
    case True
    interpret cat_0_parity spare assigned parity
      using True by unfold_locales
    show ?thesis
      apply (clarsimp simp: distinct_conv_nth_less choices_length)
```

```
      apply (case_tac "i = 0")
      using True choices_correct choice_0 distinct_pointwise
      by (auto split: if_splits)
  next
    case False
    thus ?thesis using choices_length[of assigned] by simp
  qed

lemma (in hats_parity) choice_legal:
  assumes "i < length assigned"
  shows "choices assigned ! i ∈ set (spare # assigned)"
  proof (cases "i = 0")
    case True
    interpret cat_0_parity spare assigned parity
      using assms True by unfold_locales simp
    show ?thesis using choice_0 using assms True by simp
  next
    case False
    thus ?thesis using assms choices_correct by auto
  qed

lemma (in hats_parity) choices_legal:
  "set (choices assigned) ⊆ set (spare # assigned)"
  using choices_length choice_legal subsetI in_set_conv_nth
  by metis
```

4 The parity function

4.1 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 Top-level theorems

```
global_interpretation parity_classifier parity
  using parity_swap[where as="[]"] by unfold_locales simp

sublocale hats < hats_parity spare assigned parity
  by unfold_locales

context
  fixes spare assigned
  assumes assign: "set (spare # assigned) = {0 .. length assigned}"
begin
  interpretation hats using assign by unfold_locales
  lemmas choices_legal = choices_legal
  lemmas choices_distinct = choices_distinct
  lemmas choices_correct = choices_correct
end
```

We have four top-level theorems which show that we have solved the puzzle. The first shows that we have not cheated:

$$\frac{i < \mathit{length\ assigned} \qquad \mathit{spoken} = \mathit{choices\ assigned}}{\mathit{spoken}\ !\ i = \mathit{choice}\ (\mathit{take}\ i\ \mathit{spoken})\ (\mathit{drop}\ (\mathit{Suc}\ i)\ \mathit{assigned})}$$

We don't need to look at the implementation of *choices* or *choice* to know this! The theorem is parametric in the set of *spare* and *assigned* hats, so the *choice* function can only use what appears in its arguments. Even if *choices* cheats, it agrees with *choice*, which cannot.

The next two show that the *choices* are legal. That is, every cat chooses the number of some hat, and no number is repeated:

$$\frac{\mathit{set}\ (\mathit{spare}\ \#\ \mathit{assigned}) = \{0..\mathit{length\ assigned}\}}{\mathit{set}\ (\mathit{choices\ assigned}) \subseteq \mathit{set}\ (\mathit{spare}\ \#\ \mathit{assigned})}$$

$$\frac{\mathit{set}\ (\mathit{spare}\ \#\ \mathit{assigned}) = \{0..\mathit{length\ assigned}\}}{\mathit{distinct}\ (\mathit{choices\ assigned})}$$

Finally, every cat except the rearmost chooses the number of its assigned hat:

$$\frac{\mathit{set}\ (\mathit{spare}\ \#\ \mathit{assigned}) = \{0..\mathit{length\ assigned}\} \qquad k \in \{1..<\mathit{length\ assigned}\}}{\mathit{choices\ assigned}\ !\ k = \mathit{assigned}\ !\ k}$$

5.1 Solving the puzzle

5.1.1 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"
```

5.1.2 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' []"
```

5.1.3 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
```

5.1.4 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')
```

5.1.5 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)
```

5.2 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

lemmas assigned_0
  = Cons_nth_drop_Suc[OF exists, simplified]

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
```