Schrödinger's hats

A puzzle about parities and permutations

Matthew Brecknell

May 2, 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.[1] 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.

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] Some exposure to Haskell, ML or similar, predicate logic with quantifiers, and naive set theory should be adequate.

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 Taking turns

The rules require each cat to say exactly one hat number, 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 said by the cats, in order from back to front.
  fixes spoken :: "nat list"
  — Each cat speaks exactly once.
  assumes length: "length spoken = length assigned"

— Each cat hears what was said by the cats behind it.
definition (in cats) "heard k ≡ take k spoken"

— Each cat sees the assigned hats in front of it.
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.

3 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.

4 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
```

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

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

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*. Logically, an assumption introduced with **assumes** is identical to one introduced with the implication arrow ($\Longrightarrow$). Which to use is a matter of aesthetics.

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:[6]

```
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
```

5 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 i said a number that it saw on the hat of j who is in front of i. Hat numbers are unique, so i's number must be different from j's, and therefore i's choice is wrong. But j may not repeat the number that i said, so j is also wrong.

Each cat i therefore has to choose between exactly two hats: those remaining after excluding all the numbers it has seen and heard. We'll call these the *candidates*, and we'll make our definition outside our locales, since it will form part of our final solution:

```
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"
```

We *calculate* the set of all hats by counting the number of *heard* and *seen*, so we're relying on the fact that the set of all hats is always the set containing 0 up to the number of cats.

For convenience, we'll make a corresponding definition in the *cats* locale:

```
definition (in cats)
  "candidates i ≡ candidates_excluding (heard i) (seen i)"
```

We now want to prove that *candidates* produces the right results. Consider cat *i*. If we take *heard i* and *seen i*, we know that we need to add two more hats to make up the complete set. Conversely, if

[6] Keyword *op* turns an infix operator into a prefix function.

we start with *heard i* and *seen i*, and add two hypothetical hats *a* and *b*, such that the result is the complete set of hats, then *candidates i* should be those hats *a* and *b*. Formally:

```
lemma (in cats) candidates_i:
  fixes a b i
  defines "view ≡ (a # heard i @ b # seen i)"
  assumes i_length: "i < length assigned"
  assumes distinct_view: "distinct view"
  assumes set_view: "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 length by auto
    have set: "set ?excluded = {0..length assigned} - {a,b}"
      apply (rule subset_minusI)
      using distinct_view set_view unfolding view_def by auto
    show ?thesis
      unfolding candidates_def candidates_excluding_def Let_def
      unfolding len set
      unfolding Diff_Diff_Int subset_absorb_r
      using set_view unfolding view_def
      by auto
  qed
```

Here, we've introduced the idea of a *view*. This is an hypothetical ordering of the complete set of hats, seen from the perspective of some cat. In this case, cat *i* imagines some hat *b* on its own head, between the hats it has *heard* and *seen*, and imagines the remaining hat *a* on the floor behind the rearmost cat, where no cat can see it. The order of the list does not matter here, though it will later, but it is still a nice visualisation. Here, we just need to know that the hats in the list are exactly those in full set of hats, and we capture this in the assumptions *distinct_view* and *set_view*.

6 The rejected hat

We now return to cat k of the *cat_k* locale. Since none of the cats $\{i \mid 0 \leq i < k\}$ previously said k's number, k's own number must be one of its *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. We'll call this the *rejected* hat.

To formalise this, we'll need to somehow define the rejected hat. We'll define the *rejected* hat in term of the choice made by cat *0*. We don't yet know how the cats choose their hats, but we can talk about their *candidates*. Before we can consider the rearmost cat, we first need to know that it exists, so let's make an assumption:

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

With this, we can safely extract the first *assigned* hat, and prove that the rearmost cat's *candidates* are as we expect. We make use of the *candidates_i* lemma, first defining a view, and proving lemmas to satisfy the *distinct_vew* and *set_view* premises.

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

lemma (in cat_0)
  distinct_0: "distinct view_0" and
  set_0: "set view_0 = {0..length assigned}"
  using distinct_hats assign
  unfolding seen_def Cons_nth_drop_Suc[OF exists_0]
  by auto

lemma (in cat_0) candidates_0: "candidates 0 = {spare, assigned ! 0}"
  using candidates_i exists_0 distinct_0 set_0
  unfolding heard_def seen_def
  by auto
```

Now we can define the `rejected` hat as whichever of those the rearmost cat does *not* say:

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

We now want to prove that `candidates k` consists of `k`'s `assigned` hat, and the `rejected` hat, but there's a problem. Since we defined `rejected` in the `cat_0` locale, it is not currently visible in `cat_k`. To make it visible, we need to *interpret* the `cat_0` locale in the context of `cat_k`.

To interpret a locale means to make all the *consequences* of that locale available in some new context, including definitions and lemmas proved. But for that to be logically sound, this means we need to *prove the assumptions* of the locale we are interpreting, in that same context.

In this case, we want to make the consequences of `cat_0` available in `cat_k`, so we need to prove the assumptions of `cat_0` in the context of `cat_k`. Thankfully, in `cat_k`, we can use the assumptions of `cat_k`, and `exists_0` follows easily from `k_min` and `k_max`.

To interpret one locale within another, we use the **sublocale** command:

```
sublocale cat_k < cat_0
  using k_min k_max by unfold_locales auto
```

Why did we not prove `exists_0` in locale `cat_k` in the first place? The reason is that later, we'll need `distinct_0` and `set_0` in a context where we don't have a cat `k`, but where we can prove `exists_0` by other means.

Now, we want to use `candidates_i`, but can't immediately satisfy the `distinct_view` and `set_view` premises of `candidate_i`, for cat `k`'s view. However, we notice that there is an ordering of the full set of hats which is a view for both cat `0` and cat `k`:

```
abbreviation (in cat_0) (input) "view_r ≡ rejected # spoken ! 0 # seen 0"
abbreviation (in cat_k) (input) "view_k ≡ rejected # heard k @ assigned ! k # seen k"
```

We expect these lists should be equal, because:

- the first thing that cat `k` would have `heard` was `spoken ! 0`, and
- under our `cat_k` assumptions, the rest of `view_k` is what cat `0` had `seen`.

This is interesting, because `view_r` gets us closer to `view_0`, for which we have already proved the `candidates_i` premises. If we can show that:

- `view_r` and `view_k` are equal, and that
- the first two hats in `view_r` are the same as the first two in `view_0`,

then we are close to proving the `candidates_i` premises for `view_k`. We can prove the first of these:

```
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_r = 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
```

To prove the second, 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`. Then we can prove the `view_r` lemmas directly:

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

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

Again, we're keeping a separate `cat_0` locale hierarchy, because we'll need this later. In any case, we can always recombine locales, as we do now to prove the `view_k` lemmas, and finally, the lemma for `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_r set_r 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 would trivially follow that cat k chooses its `assigned` hat. We don't gain much from formalising that now, but hopefully it's clear that the remaining task is to ensure that cat `k` does indeed reject the same hat as the rearmost cat.

7 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'll 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 locale 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"
```

We'll say more about the *classifier* in the next section. First, we'll define a function which assembles the choices of all the cats into a list. We'll need this to instantiate the *spoken* parameter of the *cats* locale.

We define the *choices* function in two steps. First, we recursively define *choices'*, taking two arguments: the numbers *heard* by the current cat; and the remaining *assigned* hats for the current cat and all cats towards the front. It recurses on the *assigned* hats, while building up the list of numbers *heard*. In the second case, we discard the hat *assigned* to the current cat, giving us exactly what is *seen* by the current cat, and which is also the remainder of the *assigned* hats for recursive call.

```
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)"
```

The *choices* function then specialises to the initial state where the rearmost cat begins having *heard* nothing:

```
definition (in classifier) "choices ≡ choices' []"
```

We can prove, in two steps, that the number of *choices* is the same as the number of *assigned* hats:

```
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)
```

We can also prove that the individual *choices* are as we expect. The *choices* lemma is important, because it makes clear that the *choices* function does not cheat. It agrees with the *choice* function, which is given exactly the information available to the respective cat. We know that *choice* cannot cheat, because the *choices* lemma is parametric in the list of *assigned* hats.

```
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)
```

8 The classifier

Like the views we used in the *candidates* lemmas, the order we pass arguments to the *classifier* is suggestive of one of the two possible orderings of the full set of hats that is consistent with what was *heard* and *seen* by the cat making the *choice*, with hat *b* in the position of the cat, and hat *a* on the floor behind the rearmost cat.

Rather than return a hat number, the *classifier* returns a *bool* that indicates whether the given ordering should be accepted or rejected. If accepted, the cat says the number it had imagined in its place. If rejected, it says 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 it's a small refinement from choice to classifier, it gives us a new 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 ask which is the ordering that is consistent with the information available to *every* cat.

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 when their turns come, and also accepted by their shared 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_consistent` suggests that the position in the line is redundant, and we can collapse the classifier's arguments into a single list. We define a `parity_classifier` locale which does just this. It is a specialisation of the `classifier_swap` locale, in which we instantiate the `classify` parameter with a classifier based on an arbitrary but fixed `parity` function. As a specialisation of the `classifier_swap` locale, `parity_classifier` assumes a specialised version of `classifier_swap`.

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

abbreviation (input)
  "classifier_of_parity parity ≡ λa heard b seen. parity (a # heard @ b # seen)"

locale parity_classifier = classifier_swap "classifier_of_parity parity"
  for parity :: "parity"
```

We can show that `parity_classifier` satisfies the `classifier_consistent` requirement, with a **sublocale** proof. This means that the only thing we require of our `parity` function is that it satisfies the `classifier_swap` property.

```
sublocale parity_classifier < classifier_consistent "classifier_of_parity parity"
  by unfold_locales simp
```

9 Solving the puzzle

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

First, we need a locale which combines `hats` and `parity_classifier`:

```
locale hats_parity = hats + parity_classifier
```

Previously, in the `cats` locale and its descendents, we had to take the numbers `spoken` by the cats as a locale parameter, since we did not know how they made their choices. Now, we want to instantiate this parameter with `choices assigned`, so we'll make a fresh batch of locales which do this. We'll also discharge the `cats` locale assumption for this instantiation, with a **sublocale** proof:

```
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
```

The following are just restatements of things we've already proved, but in terms slightly more convenient for the proofs further on.

```
lemma (in cat_0) 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_k_view) candidates_excluding_k:
  "candidates_excluding (heard k) (seen k) = {rejected, assigned ! k}"
  using candidates_k unfolding candidates_def by simp

lemma (in cat_0_parity) parity_swap_0:
  "parity (spare # assigned ! 0 # seen 0) ⟷ ¬ parity (assigned ! 0 # spare # seen 0)"
  using classifier_swap[of spare "[]"] distinct_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_k_parity) choices_k:
  "choices assigned ! k = choice (heard k) (seen k)"
  unfolding heard_def seen_def using choices[OF k_max] by simp
```

Since cat `0` uses the `parity` function to make its choice, we can prove a couple of results about how its choice relates to `view_0` and `view_r`. Note that the `parity` of `view_r` is always true!

```
lemma (in cat_0_parity) choice_0:
  "choices assigned ! 0 = (if parity view_0 then assigned ! 0 else spare)"
  using distinct_0 parity_swap_0
  unfolding choices_0 choice_def candidates_excluding_0
  by (subst sorted_list_of_set_distinct_pair) auto

lemma (in cat_0_parity) parity_r: "parity view_r"
  using distinct_0 parity_swap_0
  unfolding choices_0 choice_def candidates_excluding_0 rejected_def
  by auto
```

Since `view_r` and `view_k` are equal, we also have that `parity view_k` is always true:

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

At long last, we are almost ready to prove in `cat_k_parity` that cat `k` makes the right choice! But first, we need to make certain results about `view_k` available in the `cat_k_parity` locale. As usual, we'll use a sublocale proof:

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

Recall that our induction proof, `cat_k_induct`, showed that if every cat satisfying `cat_k` says the correct number, then every cat except the rearmost says the correct number. We've just shown that every cat satisfying `cat_k_parity` says the correct number, so to apply the induction lemma, we need to show that every cat satisfying `cat_k` also satisfies `cat_k_parity`. The only undischarged assumptions in `cat_k_parity`, relative to to `cat_k`, are the ones we make in `hats_parity`, so this implication is easy to prove in `hats_parity`:

```
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 by (rule assms)
    show ?thesis by unfold_locales
  qed
```

Finally, using our induction lemma, we get that in `hats_parity`, every cat except the rearmost says its assigned hat number.

```
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])
```

10 Legalities

There are a couple of rules which we've observed in our formal analysis, but for which we, so far, have no proof: every cat must say the number of some hat, and every cat must say a distinct number. We present the proofs without further comment.

```
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
```

11 Deriving the parity function

We have come a long way, but there is still one missing piece of the puzzle: a `parity` function which satisfies the `classifier_swap` property. Informally, the property requires that if we take a list of distinct naturals, and swap the *first* number with *any other number*, then the `parity` is inverted.

If we had such a function, what other properties must it have? For example, what happens to the `parity` when we swap two elements not including the first? By performing a sequence of three swaps with the first element, we can get the effect of an arbitrary swap, and derive the following property. This means that we actually require that if we swap *any* two elements, then the `parity` is inverted.

```
lemma (in parity_classifier) parity_swap_any:
  assumes "distinct (as @ b # cs @ d # es)"
  shows "parity (as @ b # cs @ d # es) ⟷ ¬ parity (as @ d # cs @ b # es)"
  proof (cases as)
    case Nil thus ?thesis using assms classifier_swap[of b] by simp
  next
    case (Cons a as)
    hence "parity (a # as @ b # cs @ d # es) ⟷ ¬ parity (b # as @ a # cs @ d # es)"
      using assms classifier_swap[of a as b "cs @ d # es"] by simp
    hence "parity (a # as @ b # cs @ d # es) ⟷ parity (d # as @ a # cs @ b # es)"
      using Cons assms classifier_swap[of b "as @ a # cs" d es] by simp
    hence "parity (a # as @ b # cs @ d # es) ⟷ ¬ parity (a # as @ d # cs @ b # es)"
      using Cons assms classifier_swap[of d as a "cs @ b # es"] by simp
    then show ?thesis using Cons by simp
  qed
```

How might we construct such a function? Let's start small, and consider only lists of exactly two distinct numbers. There are only two ways to order the elements, and four functions to a `bool` result. Two of those are constant functions which don't satisfy the `classifier_swap` property. One of the non-constant functions tests whether the numbers are in ascending order, and the other, descending order. They are mutual inverse, and both satisfy `classifier_swap`. We arbitrarily choose the first:

```
definition "parity_of_two xs ≡ case xs of [a,b] ⇒ a ≤ b"
```

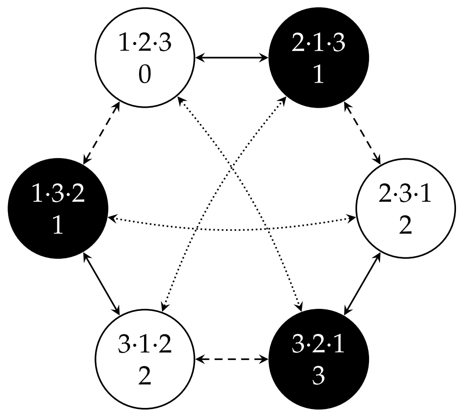

Figure 1: Permutations of three elements, by swaps of pairs.

Let's move on to lists of three distinct elements. There are six ways of ordering three numbers, and 64 possible functions to `bool`, but there are still only two mutually inverse functions that satisfy the `classifier_swap` property! We won't formalise this claim, but we can understand it by laying out the six permutations in a graph, as in figure 1. Each node shows one of the six possible orderings of the digits 1 to 3 at the top. Connecting lines indicate swaps of two elements: solid lines for the leftmost two digits, dashed lines for the rightmost two digits, and dotted lines for the outermost two digits.

If we choose a node, and assign it an arbitrary `parity`, then the `parity_swap_any` property tells us that we must assign the opposite `parity` to any node at the other end of a shared edge. We can continue traversing edges this way, and find that every parity is determined by our initial arbitrary choice. In the figure, we represent a `parity` which is `True` with a white fill, and `False` with a black fill.

Before we can extend this to lists of any length, we need to identify the pattern. For a list of length two, we performed a single comparison. With three elements, there are three comparisons we can perform, and for n elements, $\binom{n}{2}$ comparisons.[7] If we are to use comparisons to calculate the `parity`, we need to find a way to combine many `bool` results into one.

In figure 1, the number at the bottom of each node counts the number of *inversions* in the list. An inversion occurs when a list element is greater than some other element to the right of it. For example, in the list 2·3·1, the pairs 2·1 and 3·1 are inversions, but the pair 2·3 is not. We notice that all the white nodes have an even number of inversions, and the black nodes have an odd number of inversions.

So perhaps we can define the `parity` by counting the number of inversions, and defining the `parity` as whether or not the total number of inversions is even. This seems plausible, because when we swap two numbers within a distinct list:

- The swapped pair itself will change the number of inversions by one.
- The change in the number of inversions caused by moving one of the pair over the intervening numbers will be odd if and only the change caused by the other is also odd.

[7] The *binomial coefficient*, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of ways one can choose k things from n things.

12 Defining the parity function

We are now ready to write a recursive definition of a *parity* function! For the base case, we choose *True* as the *parity* of an empty list. For the recursive case, we calculate the *parity* of the tail, and also the number of inversions between the head and the tail. If both of these are odd, then the overall *parity* is even. Likewise, if both are even. However, if one is even and the other is odd, then the overall *parity* is odd.

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

We can prove that swapping two adjacent elements inverts the parity. Since the function performs a pattern match at the head of the list argument, we prove this by induction over the list preceding the first element being swapped.

```
lemma parity_swap_adj:
  "b ≠ c ⟹ parity (as @ b # c # ds) ⟷ ¬ parity (as @ c # b # ds)"
  proof (induct as)
    case Nil
    — In the Nil case, the parity function application simplifies away,
    — because b and c are at the head of the list.
    thus "parity ([] @ b # c # ds) ⟷ ¬ parity ([] @ c # b # ds)"
      by auto
  next
    case (Cons a as)
    — In the Cons case, b and c are not at the head of the list, so we can't simplify directly.
    — However, we get the following from the induction hypothesis.
    hence "parity (as @ b # c # ds) ⟷ ¬ parity (as @ c # b # ds)"
      by simp
    — Using the induction hypothesis, we can now prove the Cons case by simplification.
    thus "parity ((a # as) @ b # c # ds) ⟷ ¬ parity ((a # as) @ c # b # ds)"
      by auto
  qed
```

To prove that swapping any two elements inverts the parity, we use *parity_swap_adj* as the base case, and reason by induction on the list between the two elements we are swapping.

```
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
    — We get the following from the assumptions.
    hence "b ≠ d" by simp
    — From that and parity_swap_adj, we get the following.
    hence "parity (as @ b # d # es) ⟷ ¬ parity (as @ d # b # es)"
      using parity_swap_adj[of b d as es] by simp
    — The Nil case then follows by simplification
    thus "parity (as @ b # [] @ d # es) ⟷ ¬ parity (as @ d # [] @ b # es)"
      by simp
```

```
next
  case (Cons c cs)
  — We get the following by swapping b and c, which are adjacent.
  have " parity (as @ b # c # cs @ d # es) ⟷ ¬ parity (as @ c # b # cs @ d # es)"
    using Cons parity_swap_adj[of b c as "cs @ d # es"] by simp
  moreover
  — We get the following by swapping d and c, which are adjacent.
  have "parity (as @ d # c # cs @ b # es) ⟷ ¬ parity (as @ c # d # cs @ b # es)"
    using Cons parity_swap_adj[of d c as "cs @ b # es"] by simp
  moreover
  — We get the following from the induction hypothesis.
  have "parity (as @ c # b # cs @ d # es) ⟷ ¬ parity (as @ c # d # cs @ b # es)"
    using Cons(1)[where as="as @ [c]"] Cons(2) by auto
  ultimately
  — By combining the previous three swaps, we can prove the Cons case.
  show "parity (as @ b # (c # cs) @ d # es) ⟷ ¬ parity (as @ d # (c # cs) @ b # es)"
    by simp
qed
```

13 Top-level theorems

We now have what we need to discharge our remaining assumptions. By performing a **global_interpretation** of the `parity_classifier` locale, specialised using our concrete `parity` function, we make all the theorems and definitions of that locale available globally:

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

By interpreting the `hats_parity`, specialised using our concrete `parity` function, within the `hats` locale, we make all the theorems of `hats_parity` available in the `hats` locale:

```
sublocale hats < hats_parity spare assigned parity
  by unfold_locales
```

We can then plumb the important theorems into the global context, by locally assuming the same things as the `hats` locale:

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

We now have four top-level theorems which show that we have solved the puzzle. We'll present them in the traditional *rule* format, with premises above the line, and conclusions below the line. The first shows that we have not cheated:

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

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:

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

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

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

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

14 Conclusion

Solving algorithmic problems requires precise thinking. It requires us to keep account of the properties established by and required by any algorithm we are constructing. It helps enormously to have a language in which we can write down what we know and what we have assumed, and which allows us to check that our reasoning is logically sound. A mechanised theorem prover like Isabelle/HOL gives us such a language.

But precise thinking also requires *practice*. My hope in writing this is to convince you that exercising using a formal theorem prover is a useful personal discipline for developing some aspects of algorithmic problem-solving ability.

To learn more about theorem proving using Isabelle/HOL, read Tobias Nipkow and Gerwin Klein, *Concrete Semantics*, Springer 2014. There is a free PDF available here:

`http://concrete-semantics.org/`

This article was written as a literate Isabelle/HOL theory, so all definitions have been type-checked, and all proofs checked for validity. The source, with some other bits and pieces, is available here:

`https://github.com/mbrcknl/puzzle-parity-permutations`

In particular, there is a more direct bottom-up version of the proof, which is slightly shorter, because we did not need to invert the reasoning.