A formalization of the mutilated chessboard problem

Jeremy Avigad

July 19, 2019

1 Introduction

Consider an eight-by-eight chessboard and a set of dominos with the property that each domino can cover exactly two adjacent chessboard squares. Remove the bottom left and top right corners. The *mutilated chessboard problem* asks whether it is possible to cover the remaining squares with nonoverlapping dominos. A simple observation shows that the answer is no: every domino covers exactly one black square and one white square, and the two corners that have been removed have the same color—say, white—so there are more black squares to be covered than white squares.

The mutilated chessboard problem is often presented as an example of a problem where an "aha!" insight makes the solution obvious. The argument has been formalized in a number of interactive theorem provers, and some of these formalizations were recently discussed in a paper by Fenner Tanswell.[1] Silvia De Toffoli asked me if I could formalize the theorem in a manner that fills out the intuitive argument in a straightforward way. Section 5 contains my attempt to do so,[2] using the *Lean* interactive theorem prover.

The philosophy of mathematics currently hosts a spirited discussion of the extent to which mathematics is formal, or formalizable, whatever that may mean. Insofar as the practice of interactive theorem proving is relevant to the discussion, this note aims to provide some data. My goal here is not to make any strong philosophical claims, but merely to report on some aspects of the formalization process. I don't think anything I say below is controversial, and I believe the sentiments expressed are generally shared by those in interactive theorem proving community.

2 Overview of the formalization

Formalization in an interactive theorem prover can be viewed as an extreme form of mathematization. To express the mutilated chessboard problem in more mathematical terms, we might write something like this:

> We can represent the mutilated the chessboard as the set $\{0, \ldots, 7\} \times \{0, \ldots, 7\}$ with the elements $\{(0, 0), (7, 7)\}$ removed. A horizontal domino at position (i, j) covers the set of squares $\{(i, j), (i + 1, j)\}$ and a vertical domino at position (i, j) covers the set of squares $\{(i, j), (i, j + 1)\}$. So the question is whether it is possible to write the mutilated chessboard as a union of disjoint sets of those two types.

[1] Tanswell, Fenner, "A problem with the dependence of informal Proofs on formal proof." *Phiosophia Mathematica*, 23(3):295–310, 2015.

[2] It also online at `https://gist.github.com/avigad/1080e327ad6806884c9c5091f5c449bd`.

In this case, mathematization is somewhat gratuitous, since the original problem is clear. But if there were any ambiguity in the statement of the problem itself or in the proof, the mathematical representation should help clear it up. That is part of what mathematics is supposed to do.

Interactive theorem proving requires us to go further and express all statements in terms of symbols recognized by the system, identifiers denoting definitions and theorems that are already in the library, and so on. This is a matter of pushing the mathematization to the point where a computer can check the syntax against a formally specified grammar, and check the proofs against the rules of a formal axiomatic system.

The first two lines of the formalization imports some relevant background and open some namespaces, so that, for example, the theorem named `nat.even_or_odd` can be written `even_or_odd`. Lines 4–34 then contain some basic definitions.

```
def chessboard := zmod 8 × zmod 8

def mutilated : finset chessboard := univ \ {(0, 0), (7, 7)}

def right (u : chessboard) := (u.1 + 1, u.2)
def left  (u : chessboard) := (u.1 - 1, u.2)
def up    (u : chessboard) := (u.1, u.2 + 1)
def down  (u : chessboard) := (u.1, u.2 - 1)
```

The chessboard is presented as $\mathbb{Z}_8 \times \mathbb{Z}_8$, i.e. the data type of pairs (i, j) where i and j are integers modulo 8. Lean has a more basic type, `fin 8`, that more directly represents that values $0, \ldots, 7$, but it is convenient to use `zmod 8`, because the latter has a ring structure, allowing us in particular to use addition and subtraction modulo 8. The mutilated chessboard is the set of all elements of this data type, minus the two corners. The definitions `right`, `left`, `up`, and `down` specify the four positions of four squares that are adjacent to the square at position `u`.

We then define the notion of a domino:

```
inductive domino
| horizontal : chessboard → domino
| vertical   : chessboard → domino

def squares : domino → finset chessboard
| (horizontal u) := {u, right u}
| (vertical u)   := {u, up u}
```

Every domino is of the form `horizontal u` or `vertical u`, where `x` is a chessboard position. If `d` is a domino, `squares d` denotes the finite set of squares that it covers: `{u, right u}` if `d` is of the form `horizontal u`, and `{u, up u}` if `d` is of the form `vertical u`. This is already enough to state the main theorem:

```
theorem mutilated_checkboard_problem {s : finset domino}
    (h : ∀ u ∈ s, ∀ v ∈ s, u ≠ v → squares u ∩ squares v = ∅) :
  s.bind squares ≠ mutilated
```

The notation `s.bind squares` would be written in more conventional mathematical notation as $\bigcup_{d \in s} \mathit{squares}(s)$. In words, the theorem says that if s is a set of dominos that do not cover overlapping squares, then the union of the sets of squares covered by those dominos is not equal the mutilated chessboard.

The proof requires the notion of what it means for a square to be white. Note that this property does not play a role in the statement of the theorem, but, rather, only in the proof. A moment's reflection shows that the color of the square at (i, j) is determined by the parity of $i + j$, so we say that (i, j) is white if $i + j$ is even.

```
def white (p : chessboard) : Prop := even (p.1.val + p.2.val)

abbreviation white_squares : finset chessboard := filter white finset.univ
abbreviation black_squares := univ \ white_squares
```

Some other lines in the file—the ones that begin with `instance` for example—relate to the fact that Lean's foundation has a computational framework. We can ask Lean to evaluate expressions. For example, in response to the input

```
#eval if white (2, 3) then "white!" else "black!"
```

Lean returns `"black!"`. The purpose of the line

```
instance : decidable_pred white := by intro; unfold white; apply_instance
```

gets Lean to recognize that it is able to evaluate the predicate `white`. These computational nuances are not essential to the proof of the theorem.

The careful reader may have noticed that there is something fishy in our formalization. One of the dominos in our set `s` might be `horizontal 3 7`, and our specification says that this domino wraps around the board, covering squares $(3, 7)$ and $(3, 0)$. We could easily add a hypothesis that rules out such dominos by specifying that horizontal dominos have second coordinate strictly less than 7, and vertical dominos have first coordinate strictly less than 7: define

```
def reasonable : domino → Prop
| (horizontal (i, j)) := j ≠ 7
| (vertical   (i, j)) := i ≠ 7
```

and add the hypothesis `h' : ∀ d ∈ s, reasonable d`. But this would not change the proof: the hypothesis would simply be unused. In other words, we proved a more general theorem, showing that it is impossible to cover the mutilated chessboard even if we allow unreasonable dominos.

Lines 36–90 prove some really obvious facts. For example,

```
lemma right_left (u : chessboard) : right (left u) = u
```

says that the square to the right of the square to the left of `u` is `u`,

```
lemma right_ne (u : chessboard) : right u ≠ u
```

says that the square to the right of `u` is not the same as `u`, and

```
lemma white_right {u : chessboard} : white (right u) ↔ ¬ white u
```

says that the square to the right of `u` is white if and only if `u` is not white. It is annoying to have to prove these—they are so damn obvious! But in each case, the proof is just a matter of unfolding definition, using some basic properties of modular arithmetic and the like, and relying on some simple automation to fill in details. Marking lemmas with the attribute `[simp]` tells Lean's automation to use the corresponding identities to simplify expressions in the future.

The remainder of the file consists of three lemmas and then the main theorem. The first lemma says that the number of white squares is equal to the number of black squares on a full chessboard.

```
lemma card_white_squares : card white_squares = card black_squares
```

If you think this is obvious, then—quick!—is it true of a 7×7 chessboard? The fact that it may have taken you a second or two to realize why not justifies a short argument that it is true in this case. The one given here is that the white squares can be paired in a one-to-one fashion with the black squares by associating to each white square the black square immediately to its right (wrapping around the chessboard for squares in the last column).

The second lemma uses the previous one to show that on the mutilated chessboard, there are two extra black squares:

```
lemma card_mutilated_inter_black_squares :
  card (mutilated ∩ black_squares) = card (mutilated ∩ white_squares) + 2
```

This follows from the fact that the mutilated chessboard removes two white squares and no black ones.

The third lemma says that a finite set of nonoverlapping dominoes covers just as many white squares as black squares:

```
lemma card_bind_squares_inter_white_squares {s : finset domino}
    (h : ∀ u ∈ s, ∀ v ∈ s, u ≠ v → squares u ∩ squares v = ∅) :
  card (s.bind squares ∩ white_squares) = card (s.bind squares ∩ black_squares)
```

It follows from the fact that the dominoes cover disjoint sets of squares, and each one covers one white one and one black one.

We now have everything we need to prove the main theorem. Suppose `s` is a finite set of dominos covering disjoint sets of squares. If we assume `s.bind squares = mutilated`, we can replace the left-hand side by the right-hand side in the conclusion of the third lemma to obtain the following:

```
card (mutilated ∩ black_squares) = card (mutilated ∩ white_squares)
```

This contradicts the second lemma. Hence `s.bind squares` $\neq$ `mutilated`.

3 Observations

The first thing to note is that the proof is somewhat tedious—at least, a lot longer than the one-line argument in the introduction. Spelling out the details requires, first, expressing the original problem in more mathematical terms, and then translating the mathematics into the precise syntax required by the theorem prover. But mathematics is about making ideas precise, and that is often tedious. A grade-school exercise asks children to write careful instructions for making a peanut-butter-and-jelly sandwich, and then has them laugh at all the things that go wrong when their teacher follows their instructions to the letter while misinterpreting all the things that are left out. Being precise is hard work, but that is what mathematics is about. Being fully formal requires you to be *very* precise—usually much more precise than one would like—but the possibility of doing so stems from the fact that we are dealing with mathematics rather than peanut-butter-and-jelly sandwiches.

Second, there is an act of translation involved. In this case, we had to express notions like "chessboard" and "white" in mathematical terms that were alien to the original formulation. This is common with word problems. Other types of problems leave less room for interpretation: if we were asked to show that any group in which every element is idempotent is necessarily abelian, we

wouldn't have to think as carefully about how to represent the statement. But even the mathematical notion of a group can be formulated in various ways, and to be precise one has to pick one.

Which leads to the next observation: often there are *many* possible representations, that is, multiple ways of spelling out the details. Mathematically, it doesn't matter whether we take the unit element to be part of the group structure or rely on an axiom that merely asserts that a unit exists. Similarly, it doesn't matter whether we take a tiling to be a finite *set* of dominos or a finite *sequence* of dominos. There are lots of reasonable ways to fill out the mathematical details, and even more choices to be made when it comes to formalization.

When formalizing a mathematical theorem, some choices that have to be made are insignificant, whereas other choices can result in more or less convenient manners of expression. Sometimes a choice of representation is even more important than that: it can be crucial to seeing one's way to the desired conclusion, or make a big difference in how the argument plays out. Even though the specific challenges are different, everything I just said it true of informal mathematics as well. Some representational choices are insignificant, but other choices of representation structure mathematical arguments in more substantial ways. Novel ways of representing algebraic structures, spaces, and combinatorial objects sometimes yield crucial insights.

The main moral I would like to extract from this exercise is that formalization is continuous with the usual mathematical procedures for making claims and arguments precise. When one comes across a mathematical theorem, one may wonder whether the proof is correct, but once one convinces oneself that the proof *is* correct, there is no further question as to whether it can be formalized. Being correct *means* that one can supply details to any level of precision, down to the axioms and rules of a formal foundation if necessary. Spelling out spatial or visual intuitions in mathematical terms can be inordinately difficult, but we know how to do that, too, and the fact that we can do it is part and parcel of what we take such arguments to be properly mathematical. In particular, anyone who has ever formalized a nontrivial mathematical theorem will immediately recognize that the mutilated chessboard problem can be formalized. One may wonder how best to do it and how much effort it will require (and the answer is, invariably, "more than we would like"). But there is never a question as to whether it is possible.

Reducing an informal argument to formal terms is not always mathematically interesting, though sometimes valuable insights emerge. And although formalization sometimes has its charms, it is often unpleasant. Struggling with the incidental features of a theorem prover and its libraries can feel like a distraction from the core mathematical ideas, and we still have a lot to learn about how to bridge the gap between informal and formal mathematics efficiently and effectively. But the point is simply that it can be done, and that the practice of interactive theorem proving only corroborates the claim that, for a piece of mathematics, being correct is tantamount to being formalizable.

4 A note on "translation"

Responding to an earlier draft, Silvia challenged my use of the word "translation" in the last section on the grounds that it may suggest some sort of equivalence of content or meaning. In this case, the formal argument is much longer and more detailed than the original one, conforms to a rigid syntax, and contains additional information. More generally, different presentations of a mathematical object or argument can have very different features. Silvia is right to point out that we currently lack a good vocabulary to account for what changes, and what remains the same, between the different presentations.

The word "representation" is commonly used in situations like this. For example, one might say that the formalization described here is a representation of the informal argument. Ken Manders worried that such uses of the word "representation" results in excessive focus on the thing being represented, rather than important features of the representation itself. As a result, he preferred using the term "artifact" in its place. But this choice downplays the fact that our mathematical artifacts are often related to one another in special ways, which is what we are getting at when we take them to be representations of a common thing.

I would like to propose using words like "rendering," "portrayal," or "depiction" instead. Using terms generally associated with works of art can keep us mindful of similarities between artistic and mathematical representation. Artists are free to choose their media; they can render a portrait in oil, or in watercolor, or digitally. Artists also choose their perspective—they can render a subject from up close or from afar—and their style—they can be realistic or abstract. Different portrayals of the Virgin Mary or one of Napoleon's battles or a scene of nature can serve different purposes and have different effects, and the artist need not be explicit about the purpose or fully conscious of the effects. This choice of terminology is subject to Manders' concern, but we can get over it: we can talk about depictions of unicorns in medieval art or Basil Rathbone's portrayal of Sherlock Holmes without pretending that unicorns and Sherlock Holmes are real objects.

What follows, then, is a formal rendering, or portrayal, of the mutilated chessboard theorem.

5 The formalization

```
1  import data.finset data.zmod.basic data.nat.parity tactic.norm_num
2  open finset nat
3
4  /- definitions -/
5
6  def chessboard := zmod 8 × zmod 8
7
8  instance : fintype chessboard := by unfold chessboard; apply_instance
9  instance : decidable_eq chessboard := by unfold chessboard; apply_instance
10
11 def mutilated : finset chessboard := univ \ {(0, 0), (7, 7)}
12
13 def right (u : chessboard) := (u.1 + 1, u.2)
14 def left  (u : chessboard) := (u.1 - 1, u.2)
15 def up    (u : chessboard) := (u.1, u.2 + 1)
16 def down  (u : chessboard) := (u.1, u.2 - 1)
17
18 @[derive decidable_eq]
19 inductive domino
20 | horizontal : chessboard → domino
21 | vertical   : chessboard → domino
22
23 open domino
24
25 def squares : domino → finset chessboard
26 | (horizontal u) := {u, right u}
```

```
| (vertical u)   := {u, up u}

def white (p : chessboard) : Prop := even (p.1.val + p.2.val)

instance : decidable_pred white := by intro; unfold white; apply_instance

abbreviation white_squares : finset chessboard := filter white finset.univ
abbreviation black_squares := univ \ white_squares

/- straightforward facts -/

private lemma aux₀ (n : nat) : even (n % (8 : pnat)) ↔ even n :=
by change even (n % 8) ↔ _; simp [even_iff, mod_mod_of_dvd _ (dec_trivial : 2 ∣ 8)]

private lemma aux₁ : (1 : zmod 8).val = 1 := rfl

private lemma aux₂ : (-1 : zmod 8).val = 7 := rfl

private lemma aux₃ (u : zmod 8) : ¬ (u + 1 = u) :=
begin
  intro h,
  have : u + 0 = u + 1, by simp [h],
  have : (0 : zmod 8) = (1 : zmod 8), from add_left_cancel this,
  have : 0 = 1, from congr_arg fin.val this,
  contradiction
end

local attribute [simp] aux₀ aux₁ aux₂ aux₃ filter_inter filter_insert filter_singleton

@[simp] lemma right_left (u : chessboard) : right (left u) = u :=
by simp [left, right]

@[simp] lemma left_right (u : chessboard) : left (right u) = u :=
by simp only [left, right, add_sub_cancel, prod.mk.eta]

@[simp] lemma right_ne (u : chessboard) : right u ≠ u :=
by cases u; simp [right]

@[simp] lemma up_ne (u : chessboard) : up u ≠ u :=
by cases u; simp [up]

@[simp] lemma white_right {u : chessboard} : white (right u) ↔ ¬ white u :=
by { simp [right, white, zmod.add_val] with parity_simps, by_cases even u.fst.val; simp * }

@[simp] lemma white_left {u : chessboard} : white (left u) ↔ ¬ white u :=
by { simp [left, white, zmod.add_val] with parity_simps, by_cases even u.fst.val; simp * }

@[simp] lemma white_up {u : chessboard} : white (up u) ↔ ¬ white u :=
by { simp [up, white, zmod.add_val] with parity_simps, by_cases even u.fst.val; simp * }
```

```
@[simp] lemma white_down {u : chessboard} : white (down u) ↔ ¬ white u :=
by { simp [down, white, zmod.add_val] with parity_simps, by_cases even u.fst.val; simp * }

@[simp] lemma card_squares (d : domino) : card (squares d) = 2 :=
by cases d; simp [*, squares]

lemma card_squares_inter_white_squares : ∀ d, card (squares d ∩ white_squares) = 1
| (horizontal u) := by repeat { simp [squares]; split_ifs }
| (vertical u)   := by repeat { simp [squares]; split_ifs }

lemma card_squares_inter_black_squares (d : domino) : card (squares d ∩ black_squares) = 1 :=
have squares d ∩ black_squares = squares d \ (squares d ∩ white_squares),
  by { ext x, simp, tauto },
by rw [this, card_sdiff (inter_subset_left _ _), card_squares_inter_white_squares]; simp

/- the main argument -/

lemma card_white_squares : card white_squares = card black_squares :=
have h₁ : image right white_squares = black_squares,
  begin
    ext x, simp [@mem_filter _ white],
    show (∃ y, white y ∧ right y = x) ↔ ¬white x,
      from ⟨λ ⟨y, wy, eq⟩, by simp [eq.symm, wy], λ h, ⟨left x, by simp *⟩⟩,
  end,
have h₂ : ∀ u ∈ white_squares, ∀ v ∈ white_squares, right u = right v → u = v,
  by intros u _ v _ eq; rw ←[left_right u, eq, left_right],
by rw ←[h₁, card_image_of_inj_on h₂]

lemma card_mutilated_inter_black_squares :
  card (mutilated ∩ black_squares) = card (mutilated ∩ white_squares) + 2 :=
have h₁ : mutilated ∩ black_squares = black_squares,
  begin
    ext x, simp [mutilated],
    refine ⟨λ h, h.1, λ h, ⟨h, λ h', h _⟩ ⟩,
    cases h'; simp [*, white] with parity_simps
  end,
have h₂ : mutilated ∩ white_squares ∪ {(0, 0), (7, 7)} = white_squares,
  begin
    ext x, simp [mutilated], split,
    { rintro (rfl | rfl | ⟨_, h⟩), iterate 2 { simp [white] with parity_simps}, exact h },
    intro h, simp [h], rw ←[or_assoc], apply classical.em
  end,
begin
  conv { to_lhs, rw [h₁, ←card_white_squares, ←h₂] },
  rw [card_disjoint_union],
  { simp, rw [card_insert_of_not_mem]; simp, intro h,
    have := congr_arg fin.val h, contradiction },
  simp [disjoint, white], refl
end
```

```
lemma card_bind_squares_inter_white_squares {s : finset domino}
    (h : ∀ u ∈ s, ∀ v ∈ s, u ≠ v → squares u ∩ squares v = ∅) :
  card (s.bind squares ∩ black_squares) = card (s.bind squares ∩ white_squares) :=
begin
  rw [bind_inter, card_bind], swap,
  { intros x xs y yt e, rw [inter_right_comm, ←inter_assoc, h x xs y yt e], simp },
  rw [bind_inter, card_bind], swap,
  { intros x xs y yt e, rw [inter_right_comm, ←inter_assoc, h x xs y yt e], simp },
  simp only [card_squares_inter_white_squares, card_squares_inter_black_squares]
end

theorem mutilated_checkboard_problem {s : finset domino}
    (h : ∀ u ∈ s, ∀ v ∈ s, u ≠ v → squares u ∩ squares v = ∅) :
  s.bind squares ≠ mutilated :=
begin
  intro h',
  have := card_mutilated_inter_black_squares,
  rw ←[h', card_bind_squares_inter_white_squares h] at this,
  have : 0 = 2, from add_left_cancel this,
  contradiction
end
```