Eine Formalisierung des zweiten Satzes von Sylow aus der Gruppentheorie in Naproche im Vergleich zu einer Implementierung in Lean

Moritz Hartlieb, Jonas Lippert

3. März 2020

1 Einleitung

In der vorliegenden Arbeit stellen wir die Formalisierung des zweiten Sylow-Satzes in der Sprache Lean vor und vergleichen sie mit einer eigenen Formalisierung in Naproche.

Zu Beginn seien die wichtigsten Definitionen und Resultate der Gruppentheorie skizziert auf Basis des Skripts „Eine Einführung in die Algebra (Skript, WS 19/20, Bonn)“ von Prof. Dr. Jan Schröer [1].

1.1 Theoretische Grundlagen

Eine **Gruppe** ist eine Menge G zusammen mit einer Abbildung

$$\circ : G \times G \to G$$

$$(g, h) \mapsto g \circ h = gh$$

sodass gilt:

(i) $\circ$ ist assoziativ.

(ii) Es gibt ein neutrales Element $e \in G$, sodass $e \circ g = g = g \circ e$ für alle $g \in G$.

(iii) Für alle $g \in G$ existiert ein inverses Element $g^{-1} \in G$, sodass $g \circ g^{-1} = g^{-1} \circ g = e$.

Eine **Untergruppe** H von G, H ≤ G, ist eine Teilmenge von G mit

(i) $e \in H$.

(ii) H ist abgeschlossen bzgl. $\circ$ und Inversenbildung.

Seien H ≤ G gegeben. Für $g \in G$ ist

$$gH := \{gh \mid h \in H\}$$

die **Nebenklasse** von H zu g. Es bezeichne ${}^{G}\!/_{H}$ die Klasse der Nebenklassen von H. Weiter sei

$$[G : H] := |{}^{G}\!/_{H}|$$

der **Index** von H zu G.

Zwei Untergruppen H_1 und H_2 von G sind **konjugiert**, falls es ein $g \in G$ gibt mit

$$H_1 = gH_2g^{-1} := \{g\,h\,g^{-1} \mid h \in H\}.$$

Lemma 1.1. *Seien $H \leq G$ gegeben. Für $g_1, g_2 \in G$ gilt:*

(i) $g_1H \cap g_2H \neq \emptyset \iff g_1H = g_2H \iff g_1^{-1}g_2 \in H$

(ii) Für alle $g \in G$ is die Abbildung

$$H \to gH$$
$$h \mapsto gh$$

bijektiv. Insbesondere gilt im endlichen Fall $|H| = |gH|$.

□

Lemma 1.2 (Lagrange). *Seien $H \leq G$ endlich. Dann folgt aus vorherigem Lemma:*

$$|G| = [G : H] \cdot |H|.$$

□

Für eine Gruppe G und eine nichtleere Menge X ist die Abbildung

$$\phi : G \times X \to X$$
$$(g, x) \mapsto g.x$$

eine **Gruppenaktion**, falls gilt:

(i) $1.x = x$ für alle $x \in X$.

(ii) $(gh).x = g.(h.x)$ für alle $g, h \in G$ und $x \in X$.

Weiter definieren wir für $x \in X$:

(i) den **Orbit** $G.x := \{g.x \mid g \in G\}$ von x,

(ii) den **Stabilisator** $G_x := \{g \in G \mid g.x = x\}$ von x,

(iii) die Menge der **Fixpunkte** $X^G := \{x \in X \mid g.x = x \text{ für alle } g \in G\}$ von X.

Lemma 1.3. *Sei G eine Gruppe und X nichtleer. Die Funktion*

$$G/G_x \to G.x$$
$$(g\,G_x) \mapsto g.x$$

ist bijektiv.

□

Wie wir in Lemma 1.1 gesehen haben, ist eine Gruppe G disjunkte Vereinigung der Nebenklassen bzgl. einer Untergruppe $H \leq G$. Zusammen mit Lemma 1.3 erhalten wir, dass X disjunkte Vereinigung der Orbits bzgl. einer Gruppenaktion ist und die Kardinalität der Orbits entsprechend Lagrange die Gruppenordnung teilen müssen.

Lemma 1.4 (Bahnenformel). *Sei G eine endliche Gruppe, sei $X \neq \varnothing$ endlich und seien $x_1, \ldots, x_n$ gegeben, sodass X disjunkte Vereinigung der $G.x_i$ ist. Dann gilt*

$$|X| = \sum_{i=1}^{n} [G : G_{x_i}] = |X^G| + \sum_{\substack{1 \leq i \leq n \\ x_i \notin X^G}} [G : G_{x_i}].$$

□

Falls G eine p-Gruppe ist, das heißt $|G| = p^r$ für eine Primzahl p und ein $r \in \mathbb{N}$, dann folgt:

Lemma 1.5.

$$|X| \equiv |X^G| \mod p.$$

□

Für eine Gruppe G mit $|G| = p^r m$, wobei p prim und $p \nmid m$, ist die Menge der ***p*-Sylowgruppen** definiert durch

$$\mathrm{Syl}_p(G) := \{P \leq G \mid \}.$$

Wir können nun den zweiten Satz von Sylow definieren:

Theorem 1.6. *Sei p eine Primzahl und G eine endliche Gruppe mit $|G| = p^r * m$, sodass $p \nmid m$. Sei $U \leq G$ eine p-Untergruppe, und sei $P \leq G$ eine p-Sylowgruppe. Dann gilt:*

(i) Es gibt ein $g \in G$ mit

$$gUg^{-1} \subseteq P.$$

(ii) Je zwei p-Sylowgruppen sind konjugiert.

Beweis. (i) Setze $X := G/P$ und betrachte die Gruppenaktion

$$\phi : U \times X \to X$$

$$(u, gP) \mapsto ugP.$$

Nach Lemma 1.5 gilt

$$|X| = [G : P] = m \equiv |X^G| \mod p.$$

Da $p \nmid m$, gilt $X^G \neq \varnothing$. Es existiert also ein $g \in G$, sodass $ugP = gP$ für alle $u \in U$. Folglich ist $g^{-1}Ug \subseteq P$ und damit U konjugiert zu P bzgl. g^{-1}.

(ii) Dies gilt insbesondere im Falle $U \in \mathrm{Syl}_p(G)$. □

1.2 Vorüberlegungen zur Formalisierung

Es werden also zunächst Grundbegriffe der Gruppentheorie, endliche Mengen sowie natürliche Zahlen, Primzahlen und Modulo-Rechnung benötigt. Hierzu bieten sich unterschiedliche Herangehensweisen an. Es stellt sich heraus, dass Naproche für kleine Theorien gut geeignet ist, deren Grundlagen axiomatisch eingeführt werden, die ihrerseits in einer eigenen Theorie entwickelt werden (können). Die Implementierung in Lean baut hingegen auf bereits formalisierte Grundlagen auf und ist somit Teil einer einzigen großen Theorie der Mathematik.

Interessant ist die Formalisierung von Nebenklassen. Im Sinne einer kleinen Theorie bietet sich in Naproche eine direkte Konstruktion an:

$$Coset\,(g, H, G) := \left\{ g *^G h \;\middle|\; h << H \right\}.$$

Anschließend ist zu zeigen, dass G disjunkte Vereinigung von Nebenklassen bzgl. einer beliebigen Untergruppe H ist. In Lean wird dagegen bzgl. einer Untergruppe S von G folgende Äquivalenzrelation auf G eingeführt:

$$x \sim_S y :\Leftrightarrow x^{-1} * y \in S.$$

Lean erlaubt uns, den Quotient ${}^{G}\!/_{\sim_S}$ zu betrachten. Hier wird Lemma 1.1 implizit verwendet.

Im Folgenden wird zunächst auf die jeweiligen Formalisierungen im Detail eingegangen. Anschließend sollen Vor- und Nachteile der jeweiligen Sprachen verglichen und diskutiert werden.

2 Formalisierung in Lean

2.1 Quotienten in Lean

Endliche Mengen und Nebenklassen sind in Lean als Quotient formalisiert. In der core-Library von Lean sind folgende Konstanten definiert:

```
constant quot : Π {α : Sort u}, (α → α → Prop) → Sort u
constant quot.mk :
Π {α : Sort u} (r : α → α → Prop), α → quot r
axiom quot.ind :
∀ {α : Sort u} {r : α → α → Prop} {β : quot r → Prop},
(∀ a, β (quot.mk r a)) → ∀ (q : quot r), β q
axiom quot.sound :
∀ {α : Type u} {r : α → α → Prop} {a b : α},
r a b → quot.mk r a = quot.mk r b
constant quot.lift :
Π {α : Sort u} {r : α → α → Prop} {β : Sort u} (f : α → β)),
(∀ a b, r a b → f a = f b) → quot r → β
```

Die Klasse von a in `quot r` wird durch `quot.mk r a` erzeugt. Das Induktionsaxiom stellt sicher, dass alle Elemente von `quot r` von der Form `quot.mk r a` sind. Die Lifting-Eigenschaft erlaubt es, geeignete Funktionen auf `quot r` zu liften.
Ein `setoid α` ist ein Typ α zusammen mit einer Äquivalenzrelation:

```
class setoid (α : Sort u) :=
(r : α → α → Prop) (iseqv : equivalence r)
```

Der `quotient s` auf einem `s : setoid α` ist dann der Quotient bzgl. einer Äquivalenzrelation:

```
def quotient {α : Sort u} (s : setoid α) :=
@quot α setoid.r
```

Die obigen Eigenschaften von `quot` werden anschließend auf `quotient` übertragen.

2.2 Endliche Mengen in Lean

Endliche Mengen werden auf Basis von Listen definiert. Zunächst erhält man Multimengen als Quotienten, indem Permutationen mit Hilfe der Äquivalenzrelation `perm` miteinander identifiziert werden:

```
inductive perm : list α → list α → Prop
| nil   : perm [] []
| skip  : Π (x : α) {l₁ l₂ : list α},
```

```
          perm l₁ l₂ → perm (x::l₁) (x::l₂)
| swap  : Π (x y : α) (l : list α), perm (y::x::l) (x::y::l)
| trans : Π {l₁ l₂ l₃ : list α}, perm l₁ l₂ → perm l₂ l₃ → perm l₁ l₃
```

Unter Verwendung des Beweises `perm.eqv`, dass `perm` eine Äquivalenzrelation ist, wird eine Instanz von `setoid list α` eingeführt, welche dann in der Definition der Multimenge Verwendung findet:

```
instance is_setoid (α) : setoid (list α) :=
setoid.mk (@perm α) (perm.eqv α)

def {u} multiset (α : Type u) : Type u :=
quotient (list.is_setoid α)
```

Eine Multimenge wird zur endlichen Menge, wenn sie keine Duplikate beinhaltet:

```
structure finset (α : Type*) :=
(val : multiset α)
(nodup : nodup val)
```

Die Definition von nodup für Multimengen basiert auf nodup für Listen.

```
def nodup : list α → Prop := pairwise (≠)
```

Hierbei prüft `pairwise`, ob die Relation ≠ paarweise gilt.

```
variables (R : α → α → Prop)
inductive pairwise : list α → Prop
| nil {} : pairwise []
| cons : ∀ {a : α} {l : list α}, (∀ a' ∈ l, R a a') → pairwise l →
  pairwise (a::l)
```

Jetzt wird nodup für Listen auf Multimengen geliftet. Dazu werden `quot.lift_on` folgende Parameter übergeben: der Quotient s, die zu liftende Funktion nodup und ein Beweis, dass Permutation keine Duplikate erzeugt: Aus den zwei Listen `s t` und dem Beweis p, dass diese in Relation bzgl. `perm` zueinander stehen, erzeugt perm_nodup die Äquivalenz `nodup s ⇔ nodup t`. Dann wird das Lean-interne Axiom `propext` verwendet, nach dem äquivalente Propositionen gleich sind. (Die Rechtsklammerung bei der Funktionseinsetzung wird hier durch $ gewährleistet.)

```
def nodup (s : multiset α) : Prop :=
quot.lift_on s nodup (λ s t p, propext $ perm_nodup p)

def quot.lift_on {α : Sort u} {β : Sort v} {r : α → α → Prop}
(q : quot r) (f : α → β) (c : ∀ a b, r a b → f a = f b) : β
```

```
theorem perm_nodup {l₁ l₂ : list α} : l₁ ~ l₂ →
(nodup l₁ ↔ nodup l₂)
```

Der Beweis von `perm_nodup` kann in perm.lean nachgelesen werden. In der Regel wird nicht `finset` direkt verwendet, sondern die Typenklasse `fintype`, sodass der Typ α selber endlich ist.

```
class fintype (α : Type*) :=
(elems : finset α)
(complete : ∀ x : α, x ∈ elems)
```

2.3 MOD in Lean

Die Definition von $x \pmod{y}$ auf den natürlichen Zahlen basiert auf der Wohlfundiertheit der <-Relation auf $\mathbb{N}$:

```
inductive acc {α : Sort u} (r : α → α → Prop) : α → Prop
| intro (x : α) (h : ∀ y, r y x → acc y) : acc x

parameters {α : Sort u} {r : α → α → Prop}

local infix `<`:50    := r

inductive well_founded {α : Sort u} (r : α → α → Prop) : Prop
| intro (h : ∀ a, acc r a) : well_founded

class has_well_founded (α : Sort u) : Type u :=
(r : α → α → Prop) (wf : well_founded r)
```

Ein Element `x : α` kann nur die Eigenschaft `acc` haben, falls ein „kleinstes" Element bzgl. der Relation `R` existiert. Entsprechend wird ein Rekursions- und Induktionsprinzip für wohlfundierte Relationen eingeführt, auf die hier nicht näher eingegangen werden soll. Wichtig ist der Spezialfall, dass sich über wohlfundierte Relationen Funktionen definieren lassen (siehe `def fix` weiter unten):

```
parameter hwf : well_founded r
variable {C : α → Sort v}
variable F : Π x, (Π y, y < x → C y) → C x

def fix_F (x : α) (a : acc r x) : C x :=
acc.rec_on a (λ x₁ ac₁ ih, F x₁ ih)
```

Die Konstruktion des Fixpunktes nimmt gemäß `acc.rec_on` einen Beweis `a` für `acc r x` und liefert `C x`, falls folgender Sachverhalt gegeben ist:

```
(Π (x₁ : α), (∀ (y : α), r y x₁ → acc r y) →
(Π (y : α), r y x₁ → C y) → C x₁)
```

Durch `acc.rec_on` erhalten wir über den Konstruktor `intro`:

```
x₁ : α

ac₁ : ∀ (y : α), r y x₁ → acc r y

ih : Π (y : α), r y x₁ → C y
```

`F` liefert das Gewünschte. Wir können die Fixpunkteigenschaft nun auf wohlfundierte Relationen übertragen:

```
variables {α : Sort u} {C : α → Sort v} {r : α → α → Prop}

def fix (hwf : well_founded r) (F : Π x, (Π y, r y x → C y) → C x) (
   x : α) : C x :=
fix_F F x (apply hwf x)
```

Ist eine Funktion `F` gegeben, welche die Werte `C x` für bekannte Werte `C y` der „kleineren“ Elemente `y < x` liefert, so ist nach `fix` die Funktion `C` für alle `x : α` definiert. Für `F` setzen wir

```
private def mod.F (x : nat) (f : Π x₁, x₁ < x → nat → nat) (y : nat)
   : nat :=
if h : 0 < y ∧ y ≤ x then f (x - y) (div_rec_lemma h) y else x
```

und damit `C` auf $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Hier wurde folgendes Lemma verwendet:

```
div_rec_lemma {x y : nat} : 0 <  y ∧ y ≤ x → x - y <  x.
```

Weiter können wir `mod` und damit eine Instanz der Typenklasse `class has_mod` definieren, die bereits in `core.lean` vordefiniert ist. Sie besitzt einzig den Konstruktor `has_mod.mod :` $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

```
protected def mod := fix lt_wf mod.F

instance : has_mod nat :=
⟨nat.mod⟩
```

Die Infixnotation für `has_mod.mod n m` ist `n % m`. Schließlich wird `a ≡ b [MOD n]` definiert:

```
def modeq (n a b : ℕ) := a % n = b % n

notation a ‘ ≡ ‘:50 b ‘ [MOD ‘:50 n ‘]‘:0 := modeq n a b
```

2.4 Gruppen in Lean

Gruppen werden sukzessive durch Erweiterungen von `type calsses` definiert:

```
class has_mul (α : Type u) := (mul : α → α → α)

infix * := has_mul.mul

class semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c : α, a * b * c = a * (b * c))

class monoid (α : Type u) extends semigroup α, has_one α :=
(one_mul : ∀ a : α, 1 * a = a) (mul_one : ∀ a : α, a * 1 = a)

class group (α : Type u) extends monoid α, has_inv α :=
(mul_left_inv : ∀ a : α, a⁻¹ * a = 1)
```

Entsprechendes gilt für Untergruppen

```
variables {α : Type*} [monoid α] {s : set α}

class is_submonoid (s : set α) : Prop :=
(one_mem : (1:α) ∈ s)
(mul_mem {a b} : a ∈ s → b ∈ s → a * b ∈ s)

class is_subgroup (s : set α) extends is_submonoid s : Prop :=
(inv_mem {a} : a ∈ s → a⁻¹ ∈ s)
```

und Gruppenaktionen:

```
class has_scalar (α : Type u) (γ : Type v) :=
(smul : α → γ → γ)

infixr ‘ • ‘:73 := has_scalar.smul

class mul_action (α : Type u) (β : Type v) [monoid α] extends
  has_scalar α β :=
(one_smul : ∀ b : β, (1 : α) • b = b)
(mul_smul : ∀ (x y : α) (b : β), (x * y) • b = x • y • b)
```

Es folgen die üblichen Definitionen bzgl. Gruppenaktionen.

```
variables (α) [monoid α] [mul_action α β]

def orbit (b : β) := set.range (λ x : α, x · b)

variables (α) (β)

def stabilizer (b : β) : set α :=
{x : α | x · b = b}

def fixed_points : set β := {b : β | ∀ x, x ∈ stabilizer α b}
```

Nebenklassen werden als Quotient bzgl. der Relation `left_rel` definiert:

```
def left_rel [group α] (s : set α) [is_subgroup s] : setoid α :=
⟨λ x y, x⁻¹ * y ∈ s,
assume x, by simp [is_submonoid.one_mem],
assume x y hxy,
have (x⁻¹ * y)⁻¹ ∈ s, from is_subgroup.inv_mem hxy,
by simpa using this,
assume x y z hxy hyz,
have x⁻¹ * y * (y⁻¹ * z) ∈ s, from is_submonoid.mul_mem hxy hyz,
by simpa [mul_assoc] using this⟩

def left_cosets [group α] (s : set α) [is_subgroup s] : Type* :=
  quotient (left_rel s)
```

Für eine Untergruppe `H` von `G` ist dann `left_rel H` ein `setoid G` mit entsprechender Relation und dem zugehörigen Beweis, dass es sich um eine Äquivalenzrelation handelt. Mithilfe dieser Definition wird eine Instanz von `fintype` bzgl. `left_costes H` erstellt:

```
noncomputable instance [fintype G] (H : set G) [is_subgroup H] :
  fintype (left_cosets H) :=
quotient.fintype (left_rel H)
```

Der Grund dafür, dass die Instanz als `noncomputable` markiert werden muss, ist die Verwendung von `decidable.eq` in `quotient.fintype`. Das geht auf den Umstand zurück, dass das Bild einer Funktion auf einem endlichen Typ wieder ein endlicher Typ β ist. Hierbei wird das Bild zunächst als Multimenge betrachtet. Anschließend werden eventuelle Duplikate durch den Operator `to_finset` entfernt, was die Entscheidbarkeit der Gleichheitsrelation auf β voraussetzt. Um den 2. Sylowsatz in Lean zu formulieren, fehlen noch die Definitionen des `conjugate_set` und der p-Sylowgruppen:

```
def conjugate_set (x : G) (H : set G) : set G :=
```

```
(λ n, x⁻¹ * n * x) ⁻¹' H

class is_sylow [fintype G] (H : set G) {p : ℕ} (hp : prime p) extends
    is_subgroup H : Prop :=
(card_eq : card H = p ^ dlogn p (card G))

lemma sylow_2 [fintype G] {p : ℕ} (hp : nat.prime p)
(H K : set G) [is_sylow H hp] [is_sylow K hp] :
∃ g : G, H = conjugate_set g K
```

`dlogn p (card G)` ist die Vielfachheit von p in `card G`.

`#print axioms sylow.sylow_2` zeigt, dass `propext`, `quot.sound` und `calssical.choice` verwendet wird.

`quot.sound` wird benötigt, um zu beweisen, dass das kanonische Operieren einer Untergruppe H auf den Nebenklassen einer Untergruppe K eine Gruppenaktion ist:

```
def mul_left_cosets (L₁ L₂ : set G) [is_subgroup L₂] [is_subgroup L₁]
(x : L₂) (y : left_cosets L₁) : left_cosets L₁ :=
quotient.lift_on y (λ y, ⟦(x : G) * y⟧)
(λ a b (hab : _ ∈ L₁), quotient.sound
(show _ ∈ L₁, by rwa [mul_inv_rev, ← mul_assoc, mul_assoc (a⁻¹),
  inv_mul_self, mul_one]))
```

Dafür soll die Funktion `(λ y,⟦(x : G) * y⟧)` auf Nebenklassen geliftet werden. Für zwei Elemente a und b, die bzgl. `left_rel` äquivalent sind, ist also zu zeigen, dass

```
⟦(x : G) * a⟧ = ⟦(x : G) * b⟧.
```

Mit `quot_sound` genügt dann ein Beweis dafür, dass die erzeugenden Elemente äquivalent sind. Die nötigen Umformungen erledigt `simp`.

Im Beweis von Sylow 2 werden zunächst die vorangegangenen Resultate verwendet, um herzuleiten, dass die Anzahl der Fixpunkte bzgl. obiger Aktion ungleich Null ist. An dieser Stelle liefert `classical.choice` einen solchen Fixpunkt, ohne den die Konstruktion eines geeigneten `conjugate_set` nicht möglich wäre.

2.5 Sylow 2 in Lean

Wir wollen nun auf einige Details im Beweis des Satzes eingehen. Die verwendeten Lemmata, die für das sukzessive Umformen via `rw` oder `simp` genutzt werden, sind dabei weder mathematisch noch hinsichtlich des Formalisierungsprozesses besonders interessant. Es werden zahlreiche technische Lemma über natürliche Zahlen, Kardinalitäten, Funktionen, etc. zusammengetragen, deren Beweise für den Menschen trivial und in Lean im Nachhinein teilweise nur sehr schwer verständlich sind. Es ist prinzipiell klar, dass es auf Basis der konstruierten Begriffe und Strukturen möglich ist, entsprechende Beweise für Lean verständlich zu formulieren und es genügt in der Regel zu wissen, dass sie existieren.

Die ersten drei Hilfslemma liefern uns den gewünschten Fixpunkt.

```
lemma sylow_2 [fintype G] {p : ℕ} (hp : nat.prime p)
(H K : set G) [is_sylow H hp] [is_sylow K hp] :
∃ g : G, H = conjugate_set g K :=

have hs : card (left_cosets K) = card G / (p ^ dlogn p (card G)) :=
(nat.mul_right_inj (pos_pow_of_pos (dlogn p (card G)) hp.pos)).1
$ by rw [ ← card_sylow K hp, ← card_eq_card_cosets_mul_card_subgroup,
    card_sylow K hp,
nat.div_mul_cancel (dlogn_dvd _ hp.1)],

have hmodeq : card G / (p ^ dlogn p (card G)) ≡ card (fixed_points H (
    left_cosets K)) [MOD p] :=
eq.subst hs (mul_action.card_modeq_card_fixed_points hp (card_sylow H
    hp)),

have hfixed : 0 < card (fixed_points H (left_cosets K)) :=
nat.pos_of_ne_zero
(λ h, (not_dvd_div_dlogn (fintype.card_pos_iff.2 ⟨(1 : G)⟩) hp.1)
(by rwa [h, nat.modeq.modeq_zero_iff] at hmodeq)),
```

Zunächst wird in `hs` die Kardinalität der Klasse der Nebenklassen auf Basis der Definition von Sylow-Gruppen umgeformt. Hierbei wird das Lemma von Lagrange unter dem Namen `card_eq_card_cosets_mul_card_subgroup` benutzt.

In `hmodeq` kommt die Lean-Version von Lemma (??) zur Anwendung:

```
lemma card_modeq_card_fixed_points [fintype α] [fintype G]
[fintype (fixed_points G α)]
{p n : ℕ} (hp : nat.prime p) (h : card G = p ^ n) :
card α ≡ card (fixed_points G α) [MOD p]
```

Durch Substitution wird das allgemeine Resultat auf den Spezialfall übertragen.

Mit Hilfe von `hmodeq` lässt sich schließlich `hfixed` zeigen, dass nämlich die Kardinalität der Fixpunkte größer Null ist. Nach `nat.pos_of_ne_zero` genügt es zu zeigen, dass die Kardinalität der Fixpunkte ungleich Null ist. Es wird also ein Beweis von `false` gefordert unter Annahme `h`, die Kardinalität sei gleich Null. Der Widerspruch wird erzeugt, in dem erst

```
lemma not_dvd_div_dlogn {p a : ℕ} (ha : a > 0) (hp : p > 1) :
¬p ∣ a / (p ^ dlogn p a)
```

auf `a = card G` angewendet wird. Dass `p` aber ein Teiler der rechten Seite sein muss, folgt aus der Annahme `h` zusammen mit

```
modeq_zero_iff : b ≡ 0 [MOD n] ↔ n ∣ b,
```

wobei `b = card G / (p ^ dlogn p a)`.

Um aus dem abstrakten Kardinalitäts-Argument einen konkreten Fixpunkt zu gewinnen, wird das Axiom `classical.choice` in Form des Lemmas `fintype.card_pos_iff` verwendet:

```
let ⟨⟨x, hx⟩⟩ := fintype.card_pos_iff.1 hfixed in

begin

revert hx
```

Die Taktik `revert` generalisiert den Beweis `hx`, dass `x` ein Fixpunkt ist, sodass nun folgendes zu beweisen ist:

```
x ∈ fixed_points H (left_cosets K) →
(∃ (g : G), H = conjugate_set g K)
```

Der Grund dafür ist der, dass wir für die weitere Argumentation einen Vertreter `g : G` für die Äquivalenzklasse `x` benötigen. Dies wird möglich durch das Axiom `quot.ind` in Form von `quotient.induction_on`, nachdem es genügt, das Ziel für einen beliebigen Vertreter zu zeigen:

```
refine quotient.induction_on x
(λ g hg, ⟨g, set.eq_of_card_eq_of_subset _ _⟩)
```

Für ein Element `g : G` und unter der Annahme `hg`, dass ⟦`g`⟧ ein Fixpunkt ist, bleibt nun die ursprünglich Aussage `∃(g : G), H = conjugate_set g K` zu zeigen. Dem anonymen Konstruktor wurde zur Konstruktion der Existenzaussage das Element `g` und folgendes Lemma übergeben:

```
lemma eq_of_card_eq_of_subset {s t : set α} [fintype s] [fintype t]
(hcard : card s = card t) (hsub : s ⊆ t) : s = t
```

Durch die Verwendung der Taktik `refine` und der zwei Unterstriche anstelle der Beweise `hcard` und `hsub`, werden diese Bedingungen als neue Ziele innerhalb des Taktik-Blocks erstellt, wobei das ursprüngliche Ziel als bewiesen gilt. Die beiden Beweise seien hier der Vollständigkeit halber aufgeführt, auch wenn die technischen Details nicht weiter interessant sind:

```
{
rw [conjugate_set_eq_image, set.card_image_of_injective _
  conj_inj_left,
```

```
card_sylow K hp, card_sylow H hp] },
{
assume y hy,
have : (y⁻¹ * g)⁻¹ * g ∈ K :=
quotient.exact ((mem_fixed_points' (left_cosets K)).1 hg ⟦y⁻¹ * g⟧
⟨⟨y⁻¹, inv_mem hy⟩, rfl⟩),
simp [conjugate_set_eq_preimage],
simp only [*, mul_assoc, mul_inv_rev] at *,
simp [*, inv_inv] at *}
end
```

3 Formalisierung in Naproche

4 Vergleich

5 Diskussion

6 Bibliographie