Groups and Subgroups in Formal Set Theory

September 5, 2019

In these notes we will walk through the formalization of a few easy group theory definitions and statements of theorems. The main definitions are groups, abelian groups, subgroups and normal subgroups. The main theorems are that every subgroup of an abelian group is normal, the subgroup relation is transitive, there is a group with a non-normal subgroup, and the normal subgroup relation is not transitive.

The formalization is done in a formal system based on higher-order set theory (Egal [2]), but the same set-theoretic style of formalization could be done in a type theory system like Coq or Lean. In Coq or Lean one can either assume a type `set` of sets along with ZFC-style constructors and axioms, or construct such a type `set` using the Aczel inductive type [1]. Egal was written with set theory in mind, so the parser supports basic set theoretic notation, but a front end with a parser could be written to translate set theoretic notation into Lean. The Egal formalation without proofs is at
`https://github.com/JUrban/ForSet/blob/master/egalexamples/NormalSubgroupExample_minimal.egal`
and a version with full proofs is at the same url without the `minimal` suffix.

We first define groups. In fact we define groups twice. The first time we assume we have a set G and a binary operation $*$ (on the whole set theoretic universe) and define when G and $*$ form an `explicit_Group`:

```
Variable G:set.
Variable op:set -> set -> set.
Infix * 355 right := op.

Definition explicit_Group : prop :=
      (forall a b :e G, a * b :e G)
   /\ (forall a b c :e G, a * (b * c) = (a * b) * c)
   /\ exists e :e G,
          (forall a :e G, e * a = a /\ a * e = a)
        /\ (forall a :e G, exists b :e G, a * b = e /\ b * a = e).
```

(In Egal `:e` is ASCII for $\in$ and `c=` is ASCII for $\subseteq$.)

Note that we have chosen to define groups without making the identity and inverse function explicit. These can be defined using a choice operator and proven to have the expected properties. We can further define the explicit version of being abelian in the obvious way:

```
Definition explicit_abelian : prop := forall a b :e G, a * b = b * a.
```

We now forget about the set G and operation above, but remember the definitions like `explicit_Group`. Now `explicit_Group` expects two arguments (a set G and operation $*$) and returns a proposition.

The second time we define groups as mathematical objects. Groups will be sets that package explicit groups.

```
Definition Group : set -> prop :=
  fun G => struct_b G /\ unpack_b prop G explicit_Group.
```

The first conjunct $\texttt{struct}_\texttt{b}\ G$ is defined to mean that G is an ordered pair of the form $(G', \hat{f})$ where f is a binary operation and $\hat{f}$ is its encoding as a set when restricted to the set G'. The second conjunct $\texttt{unpack}_\texttt{b}\ \texttt{prop}\ G\ \texttt{explicit\_Group}$. means that when we "unpack" G then its components G' and f form

an explicit group.[1] A constructor $\texttt{pack}_\texttt{b}$ takes a set G' and a binary operator f and forms $(G', \hat{f})$. It is provable that G' and f form an explicit group if and only if $\texttt{pack}_\texttt{b}\ G'\ f$ forms a group. Furthermore, if G is a group, then it is provably of the form $\texttt{pack}_\texttt{b}\ G'\ f$ for some explicit group formed by G' and f. (Also, as expected, the group does not depend on the behavior of f outside of G'.)

With a better parser than Egal currently has, the definition of Group could be written in a form like

```
Definition: Group G iff
            struct_b G
        and let_b (G',op) := G in
            explicit_Group G' op.
```

This could then be translated into precisely the form above using $\texttt{unpack}_\texttt{b}$ instead of let.

Informal claim: When working inside particular groups, it is more convenient to consider the explicit group with components. When working outside, it is more convenient to work with the packaged set representation.

An abelian group is a group that is explicitly abelian when we unpack it:

```
Definition abelian_Group : set -> prop :=
  fun G => Group G /\ unpack_b prop G explicit_abelian.
```

Next we define subgroups and normal subgroups. Again we will first define an explicit version working with components and then define packaged versions where the groups are sets. Assume we have a set G and a binary operation $*$. Let e be the (defined) identity element and let a^- denote the (defined) inverse of a for $a \in G$. Next assume we have another set H. We define when G, $*$ and H satisfy the explicit subgroup property if $\texttt{pack}_\texttt{b}\ H\ *$ is a group and $H \subseteq G$.

```
Definition explicit_subgroup : prop := Group (pack_b H op) /\ H c= G.
```

Influenced by proofwiki (`https://proofwiki.org/wiki/Definition:Subgroup`) have not required G and $*$ to be an explicit group in order for the subgroup relation to hold.

We also define an explicit notion of being normal. Here a typical definition is to require $x * H * x^- \subseteq H$ for all $x \in G$. Here $x * H * x-$ is interpreted pointwise and we can simply use set theoretic notation:

```
Definition explicit_normal : prop := forall x :e G, {x * a * x - | a :e H} c= H.
```

A replacement operator $\texttt{Repl} : \texttt{set} \to (\texttt{set} \to \texttt{set}) \to \texttt{set}$ is internally used to represent the set notation using $\{\cdots\}$. In this case $\{x * a * x^- | a \in H\}$ would internally be $\texttt{Repl}\ H\ (\lambda a. x * a * x^-)$. Having term level binders like this is no problem if one is working within a reasonable type theory.

We now forget about G, $*$ and H and define the subgroup relation and normal subgroup relation for groups (as sets). The definition of `subgroup` is slightly tricky and uses two $\texttt{unpack}_\texttt{b}$ operations.

```
Definition subgroup : set -> set -> prop :=
  fun H G =>
   struct_b G /\ struct_b H /\
   unpack_b prop G
     (fun G' op =>
       unpack_b prop H
         (fun H' _ => H = pack_b H' op /\ Group (pack_b H' op) /\ H' c= G')).
```

Again, with a better front end parser, we could write this with two lets:

```
Definition: subgroup H G iff
     struct_b G and struct_b H
 and let_b (G',op) := G in
     let_b (H',_) := H in
     H = pack_b H' op and Group (pack_b H' op) and H' c= G'.
```

[1] The "unpack" operator is essentially a let or match construct encoded as a higher order function.

The first two conjuncts simply say G and H are pairs of a carrier set and encoded binary operation (for each). We then unpack G and H to say the remaining conditions. Let $(G', *)$ be the unpacked version of G and $(H', *')$ be the unpacked version of H. We actually completely ignore the operation $*'$. We state three further conjuncts:

- H is $\texttt{pack}_\texttt{b}\ H'\ *$. (This forces the unmentioned operation $*'$ of H to be the same as the operation $*$ for G when both are restricted to H'.)
- $\texttt{pack}_\texttt{b}\ H'\ *$ is a group.
- $H' \subseteq G'$.

With some work, it is possible to prove expected relationships between `subgroup` and `explicit_subgroup`.

We can write $H \leq G$ when H and G satisfy the `subgroup` property.

```
Infix <= 400 := subgroup.
```

The definition of subset is the first time (in these examples) one sees a significant difference between defining concepts in a set theoretic style versus a type theoretic style. One way to define groups in type theory is as a record type with a carrier type and operations. There is no direct way to define subgroups of such a group since there is no way to say one carrier type is a "subset" of another carrier type (unless the type theory has some nontrivial subtyping relation). One could instead reformulate statements about subgroups into statements about monomorphisms between groups, but this diverges from mathematical practice. A different approach taken in type theory is to always work with groups that only have elements from some fixed "ambient" carrier type. For example, in HOL-light's `grouptheory.ml` library a type "α `group`" is defined in which all elements of a group must have type α. One α `group` could be considered a subgroup of another α `group`, but an α `group` could not directly be considered a subgroup of a β `group` when $\alpha \neq \beta$.

We define the normal subgroup relation to hold if $H \leq G$ and when we unpack the explicit subgroup is explicitly normal.

```
Definition normal_subgroup : set -> set -> prop :=
  fun H G => H <= G /\
  unpack_b prop G (fun G' op =>
     unpack_b prop H (fun H' _ => explicit_normal G' op H')).
```

We can now state the main theorems.

Every subgroup of an abelian group is normal.

```
Theorem abelian_group_normal_subgroup:
  forall G, abelian_Group G -> forall H, H <= G -> normal_subgroup H G.
```

The subgroup relation is transitive.

```
Theorem subgroup_transitive: forall K H G, K <= H -> H <= G -> K <= G.
```

There is a group with a non-normal subgroup.

```
Theorem nonnormal_subgroup: exists H G, Group G /\ H <= G /\ ~normal_subgroup H G.
```

The normal subgroup relation is not transitive.

```
Theorem normal_subgroup_not_transitive: exists K H G,
   Group G /\ normal_subgroup K H /\ normal_subgroup H G /\ ~normal_subgroup K G.
```

If we were to follow the "ambient" approach, then the last two "theorems" would only be provable under assumptions about the material available in the ambient type for the carrier. In HOL-light, the last two theorems would only be theorems for α `group`s when α has a sufficient number of elements.

Aside from the "packing" and "unpacking" (which could likely be hidden as a "let" by a front end) in the definitions, the statements read more like their informal mathematical versions that type theoretic versions

likely would. To confirm this, an advocate for doing mathematics in type theory could reformulate the definitions and statements in a more traditional type theoretic manner and mathematicians could compare.

Again, the message is not that formal abstracts should use a different prover than Lean. The message is that there are different ways to formalize mathematical definitions and statements in Lean, and one option is to work set theoretically within type theory. The set theoretic versions might be harder to prove, but their statements would hopefully be easier to create, understand, communicate and debug. There are two clear reasons why set theoretic versions might be harder to prove:

1. Much of the "type checking" for type theoretic version would often correspond to explicitly proving set membership in the set theoretic version.

2. Terms considered the same up to conversion in the type theoretic version would often correspond to two sets that need to be explicitly proven equal in the set theoretic version.

When it comes to proving, it is easy to imagine type theorists proving type theoretic versions of statements and then using the type theoretic versions to conclude the "official" set theoretic version.

References

[1] Aczel, P.: On relating type theories and set theories. In: T. Altenkirch, W. Naraschewski, B. Reus (eds.) TYPES, *Lecture Notes in Computer Science*, vol. 1657, pp. 1–18. Springer (1998)

[2] Brown, C.E., Pak, K.: A tale of two set theories. In: C. Kaliszyk, E. Brady, A. Kohlhase, C.S. Coen (eds.) Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings, *Lecture Notes in Computer Science*, vol. 11617, pp. 44–60. Springer (2019). DOI 10.1007/978-3-030-23250-4. URL `https://doi.org/10.1007/978-3-030-23250-4`