A formalization of elementary group theory in the proof assistant Lean

Andrew Zipperer
Department of Philosophy
Carnegie Mellon University

July 28, 2016

Abstract

In this thesis, we describe a formalization of elementary group theory in dependent type theory. In particular, we use an implementation of the calculus of inductive constructions in a proof assistant — Lean — to define the relevant mathematical objects and prove the relevant results. The documentation here culminates in a presentation of the first group isomorphism theorem in the formal setting.

We begin by describing features of type theories — first one with simple types, then one with dependent types after we motivate dependent types. Next, we explain how to use a type theory with dependent types — as encoded by Lean — to define logical objects and operations. Then, we describe features of Lean which facilitate the formalization of mathematical objects. Lastly, after presenting the group theory concepts informally, we present our definitions of these concepts within the type theory, and we state and prove results about these formal objects.

Contents

1 Introduction **4**
- 1.1 Interactive theorem proving . . . 4

2 Background on Lean **6**
- 2.1 Dependent type theory . . . 7
  - 2.1.1 Formal systems . . . 7
  - 2.1.2 Syntax for terms and types in simply typed lambda calculus . . . 7
  - 2.1.3 The need for dependent types . . . 10
  - 2.1.4 The calculus of inductive constructions . . . 11
- 2.2 Using dependent type theory . . . 12
  - 2.2.1 Defining propositional connectives and proof rules . . . 13
  - 2.2.2 Defining predicates, relations, and quantifiers . . . 17
  - 2.2.3 Types dependent on types; types dependent on terms . . . 18
- 2.3 Algebraic structures . . . 19
  - 2.3.1 Type class inference . . . 19

3 Background from group theory **22**

4 The formalization **24**
- 4.1 Sets . . . 24
  - 4.1.1 Operations on sets . . . 25
  - 4.1.2 Relations between sets . . . 27
  - 4.1.3 Example of proof . . . 27
- 4.2 Functions . . . 29
- 4.3 Operations on a type . . . 29
  - 4.3.1 Functions interacting with operations . . . 30
- 4.4 Groups . . . 30
  - 4.4.1 Building the group type: overview . . . 31
  - 4.4.2 Building the group type: increments . . . 32
  - 4.4.3 Example proof . . . 33
- 4.5 Objects dependent on group structure . . . 36
  - 4.5.1 Subgroups . . . 36
  - 4.5.2 Cosets . . . 38
  - 4.5.3 Normal sets . . . 38

4.6 Relations dependent on group structure 39
4.7 Homomorphism and kernel 40
4.8 Quotient types and quotient groups 40
4.8.1 Quotients in Lean: introducing constants 41
4.8.2 Quotient groups overview: informal and formal 42
4.8.3 Quotient type and quotient group: the first construction 43
4.8.4 Quotient type and quotient group: the second construction 53
4.9 Other formalizations 56

Appendices **58**

5 Bibliography **97**

Acknowledgement

Jeremy Avigad is an excellent advisor. Thank you, Jeremy, for your generosity, patience, and tirelessness. It was a privilege to work with you on this project.

Chapter 1

Introduction

Mathematics is distinguished by the inferences permitted in reasoning for claims. The reasoning of one mathematician can be checked by another by checking that each inference is among those permitted.

Mathematics is typically written in natural language; results are stated this way, and proofs are given this way. Mathematicians use special symbols to denote mathematical objects and operations, but the reasoning is presented in natural langauge.

Logicians have defined formal languages for writing statements and representing reasoning. For these formal languages, the permitted rules of inference are stipulated. Given such a formal language and the stipulated rules of inference, reasoning is correct if each step is licensed by a stipulated rule of inference, and we can check the correctness of reasoning by checking that each step is licensed by a stipulated rule of inference; so, since the rules are mechanical, the process of checking correctness is mechanical. However, checking the correctness of such reasoning by hand is tedious; checking the correctness by hand introduces the possibility of human error in checking; and further, producing reasoning of this character is exhausting. Yet, given that (i) it is possible to express reasoning such that its correctness is checkable by machine and (ii) we wish for our reasoning to be correct, it is desirable to represent reasoning in this way. This set of circumstances motivates the development of methods for checking reasoning by machine and for using machines to assist in producing reasoning.

1.1 Interactive theorem proving

Interactive theorem proving is one method of (i) making mathematical reasoning machine-checkable and (ii) supporting the production of mathematical reasoning. Given a formal language on paper, we can translate this into a computer-readable language — we will call such a language a proof language. And, given such a proof language, we can write computer programs to check the correctness of reasoning with respect to the stipulated rules of inference. Software which packages (i) an encoding of a formal language (i.e. a proof language) and (ii) a program for checking reasoning represented in this language is called a *proof assistant*.

Using the formal languages on paper, we can write statements about properties and relations and reason about them. Further, we can define mathematical objects within the

language and reason about them. So, once we have translated the formal language from paper to a computer-readable form, we can define mathematical objects in the computer-readable language. Further, the proof-checking program now checks reasoning about mathematical objects.

We can proceed in the proof language as we do in natural language. That is, we can proceed to define mathematical objects, properties, functions, and relations; prove things about them; steadily build up a collection of facts; and, organize the facts into theories.

Chapter 2

Background on Lean

Lean[1] is one proof assistant. The formal language Lean encodes is a version of the calculus of inductive constructions (CIC).[2] CIC is a variety of dependent type theory extensible by inductive definitions. In addition to encoding the CIC and providing a method for checking derivations in that language, Lean has features which assist the user in defining objects, constructing derivations, and organizing theories. These include: (1) a powerful and fast elaborator which allows the user to leave much information implicit when writing in the language, (2) automated methods for filling in missing information in expressions, (3) an object-oriented-programming-like class defining mechanism which allows the definition of compound data objects and allows such classes to inherit from other classes, and (4) a type class inference mechanism which allows the user to suppress information and which facilitates the use of natural notations [9][10][2]. Using this class definition feature, we can elegantly define algebraic structures.

In the next sections, we discuss features of Lean mentioned above that are essential in the formalization below. In order to establish a shared vocabulary and give background for the calculus of inductive constructions, we first discuss formal systems. We distinguish from these a subset — type theories. Among type theories, we distinguish between those with simple types and with dependent types. We motivate the use of dependent types for the purposes of formalizing logical and mathematical reasoning. And we describe features of the particular type theory with dependent types which Lean encodes, the CIC. After this, we demonstrate how this type theory can be used to define logical objects (e.g. propositions, connectives, predicates, quantifiers) and prove statements. We demonstrate that the CIC is bettter suited to formalize reasoning than a simple type theory, because the resources of the CIC can be used to define objects which cannot be defined using the resources of simple type theory. Lastly, we describe specific tools Lean provides to facilitate the definition of mathematical objects in the formal system.

[1] `http://leanprover.github.io/`

[2] For the original calculus of constructions, see [8]. For the calculus of inductive constructions, see [7, 16, 19]

2.1 Dependent type theory

In this section, we describe features of the formal language which Lean encodes. We do this in steps. First, we introduce the notion of a formal system. Then, we expand this into the notion of a formal system including simple types. From there, we describe motivations for developing and using a formal system with dependent types, and we describe features of such a system — the CIC. It should be noted that the system which Lean encodes is not discussed until the section titled 'Calculus of inductive constructions'. The systems which are presented before this are just examples and are not to be taken as incremental developments of the CIC.

After we describe features of the CIC, we show how we can encode familiar logical reasoning in this system using inductive types. We do this by (i) describing the *propositions-as-types* interpretation; (ii) using the resources of the CIC to define both logical connectives and quantifiers via inductive types; and (iii) using the resources of the CIC to define predicates and relations. We note here but will describe more carefully later that a *dependent type theory* is a formal system with types where *dependent types* are permitted.

2.1.1 Formal systems

Let an *alphabet* be a collection of distinct symbols.[3] Let an *expression* be a string of symbols from the alphabet. Let a *well-formed expression* be an expression which meets stipulated syntactic criteria. Later, we will distinguish between well-formed expressions labeled *terms*, *types*, and *judgments*. Let a *derivation rule* be a means of creating a judgment from one or more judgments. Let an *axiom* be a judgment taken as given.

Let a *formal system* be: (i) an alphabet, (ii) criteria for expressions to be well-formed expressions, (iii), critera for expressions to be judgments, (iv) a set of derivation rules, and (v) a set of axioms. Informally stated, a type theory is a formal system in which types are assigned to terms. In a type theory, there are judgments which assert that, given a term and a type, the term has the type.

2.1.2 Syntax for terms and types in simply typed lambda calculus

In this section, we exhibit instances of the concepts defined above. We consider examples of alphabets along with syntactic criteria for well-formed expressions; first for terms, then for types. Then, we present examples of derivation rules. In these examples, we restrict ourselves to discussing a language for defining expressions and for showing that certain expressions have certain types.

Term expressions

The set of expressions we describe in this example is the set Λ of terms in the untyped

[3]Many of these definitions and notations follow [15].

lambda calculus.[4] Let $V := \{x, y, z...\}$. Let $S := \{(,), \lambda, .\}$. Let the alphabet be the set $V \cup S$. The following inductive definition provides the syntactic criteria for being a well-formed *term expression* (alternately, *term*):

(i) if $u \in V$, then $u \in \Lambda$

(ii) if $M \in \Lambda$ and $N \in \Lambda$, then $(MN) \in \Lambda$

(iii) if $u \in V$ and $M \in \Lambda$, then $(\lambda u.M) \in \Lambda$

This inductive definition is expressed equivalently using Backus-Naur Form (BNF) notation:

$$\Lambda := V|(\Lambda\Lambda)|(\lambda V.\Lambda)$$

V is the set of term variables, and Λ is the set of terms built from the variables in V using the construction rules and the punctuation symbols in S.

Type expressions

The above is an example of an alphabet along with syntactic criteria for well-formed term expressions. We continue by describing another set of expressions; we call this set $\mathbb{T}$. It is the set of *type expressions* (alternately, *types*) in the simply typed lambda calculus.[5]

---

[4]For the reader who has not seen the untyped lambda calculus before, we state the following. It is a formalism for carefully describing the behavior of functions. Thus — though we do not discuss this here (see, e.g. [5, 15]) — it provides means for describing the notions of *variable binding*, *substitution*, and *evaluation*, and it can be used to describe concrete functions of interest, encode natural numbers, and represent computations.

For aid in reading the expressions in the maintext example above, we note the intended interpretation for the symbols and expressions in the untyped lambda calculus:

> In general, an expression represents a function or value.
>
> The intended interpretation of the symbol λ is a function-creating operator; λ binds a variable, and an expression beginning with a λ is a function. Thus, given a variable x, $(\lambda x.x)$ is the function which takes an input x and returns x; given variables x and y, $(\lambda x.\lambda y.y)$ is the function which takes an input and returns the function which takes an input y and returns y.
>
> The intended interpretation of juxtaposing expressions in the lambda calculus is this: the left expression is a function taking as input the right expression — alternately put, the left expression is *applied to* the right expression. Thus, given variables x and y, $(\lambda x.\lambda y.yx)$ takes an input, takes a second input, then applies the second input to the first. Moreover, $(((\lambda x.\lambda y.yx)z)\,f)$ reduces to $f\,z$. For more on reductions and the use of the calculus, see [5, 15].

[5]For the reader who has not seen the syntax for expressions in the simply typed lambda calculus, we state the following. The simply typed lambda calculus takes as given the term expressions of the untyped lambda calculus and makes additional stipulations. One stipulation is that term variables are assigned 'types'. A second stipulation is that compound term expressions are assigned types in accordance with rules, where the rules for assigning types to the compound term expressions use the types of the component term expressions. As an example, suppose that x is a variable and x has type A. Then, the function $\lambda x.x$ is assigned the type $A \to A$; that is, $\lambda x.x$ is a term of type $A \to A$. The interpretation of $A \to A$ is that it is the type of all functions from terms of type A to terms of type A. Moreover, given these assignments, the function $\lambda x.x$ can be applied only to inputs of type A.

Suppose that we have a set $\mathbb{V} := \{\alpha, \beta, \gamma, ...\}$ and a set $\mathbb{S} := \{(,), \rightarrow\}$. Let the alphabet for this example be the set $\mathbb{V} \cup \mathbb{S}$. The following inductive defintion provides syntactic criteria for being a well-formed type expression:

(i) if $\alpha \in \mathbb{V}$, then $\alpha \in \mathbb{T}$

(ii) if $\alpha \in \mathbb{T}$ and $\beta \in \mathbb{T}$, then $(\alpha \rightarrow \beta) \in \mathbb{T}$

In BNF,

$$\mathbb{T} := \mathbb{V}|(\mathbb{T} \rightarrow \mathbb{T}).$$

The intended interpretation of the arrow type is a function type, so $(\alpha \rightarrow \beta)$ is the type of functions from type α to type β.

Judgments

Thus, we have term expressions and type expressions. So, we have the components for the sort of assertions foreshadowed above — assertions of the sort 'this term has that type'. We now define this third set of well-formed expressions — judgments. Let ':' be a new symbol.[6] Then, a *judgment* is a string of the form '$M : \sigma$' where $M \in \Lambda$ and $\sigma \in \mathbb{T}$. The judgment '$M : \sigma$' asserts that M has type σ. M is called the subject of the judgment; σ is called the type of the judgment.

In a setting with a set of terms Λ and a set of types $\mathbb{T}$: a *declaration* is a judgment of the form $w : \sigma$ where $w \in V$ and $\sigma \in \mathbb{T}$; typed variables can be used in lambda abstractions; a *context* is a list of declarations with different subjects. Let '$\vdash$' be a new symbol. A *judgment* (with a context) is a string of the form $\Gamma \vdash M : \sigma$, where Γ is a context and $M : \sigma$ is a judgment. $\Gamma \vdash M : \sigma$ is read 'given context Γ, M has type σ'.

Derivation rules

Thus, we have judgments which assert that given a context a term has a certain type. Given such judgments and the definition of constructing compound terms, we wish to derive the type of compound terms. We give rules for deriving judgments from other judgments; for each of the term construction rules, there is a type derivation rule. Note: '$M : \sigma \in \Gamma$' is '$M : \sigma$ is an element of the list Γ'.

$$\frac{M : \sigma \in \Gamma}{\Gamma \vdash M : \sigma}\,\text{var} \qquad \frac{\Gamma, w : \sigma \vdash M : \tau}{\Gamma \vdash \lambda(w : \sigma).M : \sigma \rightarrow \tau}\,\text{abst} \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}\,\text{appl}$$

These rules are read as 'given what is above the line, we can derive/infer what is below the line'. In sum, given Λ and $\mathbb{T}$ and these rules, we can write judgments, construct contexts, and derive judgments from other judgments.

We give the following examples of derivations. First, a derivation of the example in the footnote. Note: $(x : A)$ denotes the list with one element, $x : A$.

[6] At this point, our alphabet is $V \cup S \cup \mathbb{V} \cup \mathbb{S} \cup \{:\}$

$$\dfrac{\dfrac{x : A \in (x : A)}{(x : A) \vdash x : A}\,\text{var}}{\vdash \lambda x.x : A \to A}\,\text{abst}$$

A more complex derivation. Note: we truncate the contexts for simplicity.

$$\dfrac{\dfrac{v : B \in (v : B)}{(v : B) \vdash v : B}\,\text{var} \qquad \dfrac{\dfrac{w : A \in (w : A)}{(w : A) \vdash w : A}\,\text{var} \qquad \dfrac{\dfrac{\dfrac{z : C \in (x : A, y : B, z : C)}{(x : A, y : B, z : C) \vdash z : C}\,\text{var}}{(x : A, z : C) \vdash (\lambda y.z) : (B \to C)}\,\text{abst}}{(z : C) \vdash (\lambda x.\lambda y.z) : (A \to (B \to C))}\,\text{abst}}{(w : A, z : C) \vdash ((\lambda x.\lambda y.z)w) : (B \to C)}\,\text{appl}}{(w : A, v : B, z : C) \vdash (((\lambda x.\lambda y.z)w)v) : C}\,\text{appl}$$

2.1.3 The need for dependent types

The set of types $\mathbb{T}$ is limited; for, the only types permitted are the base types from $\mathbb{V}$ (i.e. $\alpha, \beta, \gamma, ...$) and what can be constructed via the $\to$ rule. In a formal system with types, the types delimit the varieties of objects describable in the language; so, in the formal system we have given above, the only objects describable in the language are (a) terms of the types, (b) functions between types, and (c) functionals between types. Using this language, we can describe some objects and types encountered in mathematics or computer science. For example, we could add to the set of base types the familiar types of $\mathbb{N}$ or nat, **2** or bool, $\mathbb{Z}$ or int, sequences or lists. From these and the type construction rules, we could describe the types of functions and functionals, e.g. $\mathbb{N} \to \mathbb{N}$, nat $\to$ bool, int $\to$ int, $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$.

However, in mathematics and logic and computer science, we encounter types which cannot be described in the system above. Consider the following type: lists of terms, where the terms are all of type A. In this example, we have a type (i.e. list A), which depends on a type (i.e A).[7] Now, consider vectors of length n, where the terms in the vector are all of type A and n is a natural number. In this example, we have a type (i.e. vector A n) which depends on a type (i.e. A) *and* a term (i.e. n, where there is another type in play because n has type nat). In this latter example, the type is not in the base set of types and it cannot be constructed from other types using the $\to$ rule since it depends on a term. The type in this examples differs from the types in $\mathbb{T}$, because the type in the example *depends* on (i) other types in a different way than being composed by $\to$, and (ii) terms.

The formal system with simple types described above has no means for creating the type in this example, but we wish to define and reason about this and similar types; so, this system is not sufficient for our purposes. In order to describe types like that in the example and objects of those types, we need a formal system with types, where that system has provisions for constructing (a) types which depend on types and (b) types which depend on terms.[8] Types which depend on types and types which depend on terms

[7] It is possible to describe types dependent on types in simple type theory. See [17]

[8] For an outline of a series of extensions of the formal system with simple types, see [15] or [4].

are called *dependent types*. A formal system with types which depend on terms is called a *dependent type theory*.

2.1.4 The calculus of inductive constructions

The formal language Lean encodes is a dependent type theory called the *calculus of inductive constructions*. In this section, we give a brief overview of features of this language.[9] To do this, we describe properties of types and terms in the language and then the intuitions for inductive types.

Types and terms

Types and terms are treated differently in the CIC than they are in the examples in the preceding section. In the examples of the preceding section, types and terms are distinct. However, in the CIC, this distinction does not hold. For, in the CIC, all types have a type; so, since they have a type, types are also terms. We will see later that this allows the uniform treatment of types, terms, propositions, and proofs – they are all terms.[10]

The derivation rules in the CIC simultaneously give the means for constructing terms and for assigning types to those terms. In the example in the preceding section, a new type could be constructed from two types and an $\to$. The CIC includes a generalization of this: the Π. Below, to give examples of derivation rules in the CIC, we exhibit rules for Π. We also indicate how this generalizes $\to$.

In the rules, $Type$ denotes some particular type in the hierarchy. We use the syntax from the previous section for judgments, contexts, and derivations. In one rule, we use the notation $z[x/y]$ to denote the expression z with all occurrences of y replaced by x.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\, var \qquad \frac{\Gamma \vdash \sigma : Type \qquad \Gamma, x : \sigma \vdash \tau : Type_*}{\Gamma \vdash \Pi(x : \sigma).\tau : Type_*}\, \Pi form$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau \qquad \Gamma \vdash \Pi(x : \sigma).\tau : Type}{\Gamma \vdash \lambda(x : \sigma).M : \Pi(x : \sigma).\tau}\, \Pi abst \qquad \frac{\Gamma \vdash M : \Pi(x : \sigma).\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau[N/x]}\, \Pi appl$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \upsilon : Type}{\Gamma \vdash M : \upsilon}\, conv. \text{ side-condition: } \tau \text{ reduces to } \upsilon$$

In $\Pi form$, $\Pi abst$, and $\Pi appl$, x is a variable and the Π binds its occurrences in the whole expression. Consider $\Pi form$; in cases where τ does not depend on σ (i.e. in cases where τ contains no occurrences of x), we abbreviate $\Pi(x : \sigma).\tau$ as $\sigma \to \tau$, and terms of this type are the functions described in the previous section.

[9] For specifications of the language, see [8], [7], [19], [16], [6], [18], [15]

[10] In order to have both (a) all types having a type and (b) a consistent formal system, there is an infinite universe of types:: $Type_0, Type_1, Type_2, ...$ In different varieties of dependent type theory, different hierarchies are imposed on this universe; in some, for all i and j, if $i < j$, then $Type_i : Type_j$ (this is referred to as cumulative inclusion); in others, for each i, $Type_i : Type_{i+1}$ and $Type_i$ has no other type (this is referred to as non-cumulative inclusion). Lean implements a non-cumulative hierarchy.

In *conv*, we mention reduction; again, for an explanation of reduction/computation rules, see [5, 15]. In short, expressions in the CIC have a computational interpretation – expressions can be reduced according to certain rules. Thus, one means of proving an equality is to reduce two terms to the same term; we will see examples of this in the formalization — in those cases, the proof term indicating that the proof proceeds by reduction is `rfl`.

Inductive Types

The above term expressions, type expressions, universe of types, and derivation rules are features of the calculus of constructions. The calculus of inductive constructions contains these and adds a schema for defining new types. A type defined according to this schema is called an *inductive type*. The schema allows the user to define types from the bottom up by stipulating (i) terms in the type and (ii) constructors, i.e. functions which have as return type the type being defined. When an inductive type is defined, it comes with a recursor (alternately, eliminator), which is a means for defining functions with domain type the type being defined.[11] Instead of specifying the schema, we give examples and direct the reader to references (e.g. [7], [16], [11]) for a specification.

Canonical and familiar inductive types include nat and list. nat can be defined by stipulating that 0 is a nat, and for each nat n, succ n is a nat. So, 0 is a base term, and succ is a constructor. list can be defined inductively by stipulating that null is a list, and for each term a and list l, cons a l is a list. So, null is a base term, and cons is a constructor. bool can be defined inductively by giving two base terms: tt is a bool, and ff is a bool. Below, we give many examples of inductive types within the formal language.

The implementation of the calculus of inductive constructions in Lean

In addition to this formal system with types, Lean includes three items which are neither native to the calculus of constructions nor definable via inductive definitions. These are quotients, propositional extensionality, and a Hilbert choice operator. We discuss quotients below in the 'The formalization' section. In short, for the other two: propositional extensionality states that if two propositions are equivalent, the propositions are equal; and, the Hilbert choice operator allows the user, given a proposition $\exists x, P(x)$, to extract a term y such that $P(y)$.[12] The Hilbert choice operator is a classical reasoning principle and allows the derivation of other classical reasoning principles (e.g. the law of excluded middle, double negation elimination).

2.2 Using dependent type theory

We saw in the previous section that the calculus of inductive constructions consists of: term expressions, type expressions, a universe of types, a schema for defining new types (inductive types), and derivation rules which determine the types of compound terms from

[11]Terms of the inductive type are uniquely decomposable. The inductive definitions are deterministic. See [1].

[12]For more information about each of these, see Lean tutorial [2].

the types of component terms. Using these resources, we define logical and mathematical objects and reason about them. In this section, we describe how we use these resources to define in the type theory: propositions, predicates, relations, quantifiers, types dependent on types (e.g. list A), and types dependent on terms (e.g. vector A n).

2.2.1 Defining propositional connectives and proof rules

Propositions as types

So far, we have seen sketches of how to define in CIC types including nat, list, and bool. Now, we define propositions — that is the familiar propositions of logic. To encode propositions in the CIC, one method is to adopt the *propositions-as-types* interpretation. This interpretation amounts to making the following stipulations:

(i) there is a new type: Prop

(ii) terms of type Prop are themselves types

(iii) we refer to terms of type Prop as propositions (thus, propositions are types)

(iv) given a term P of type Prop and a term p of type P, p is a proof of P
(i.e. if (P : Prop) and (p : P), then p is a proof of P)

In presentations of the CIC, this interpretation is implemented as follows. $Type_0$ and Prop are synonomous. In addition to this, Prop has two defining characteristics. (1) Given a term P of type Prop (recall that this makes P a type), the result of applying the Π-appl rule to P is a term of type Prop.[13] (2) Given a term P of type Prop and given two terms p_1 and p_2 both of type P, p_1 and p_2 are viewed as identical by the system.[14]

Propositions as types and machine-checkable proofs

Encoding propositions as types has dramatic consequences. For, given a term in the CIC and a type, checking whether the term has the given type is decidable. So, since proofs are terms and propositions are types, given a term and a proposition, checking whether that term is a proof of the given proposition is decidable. In other words, through encoding propositions as types, whether a proposed term (i.e. argument) is a proof can be decided by machine.[15]

Defining logical connectives

As mentioned above, we define logical connectives and quantifiers via inductive definitions. Let's start with 'and'. For abitrary (P : Prop) and (Q : Prop), what we want from the type 'and P Q' is the following behavior: (1) we want to be able to construct a term

[13] This makes Prop *impredicative*.

[14] This makes Prop *proof irrelevant*.

[15] We note that whether proofs can be checked mechanically depends on the nature of the rules, not on the variety of formal system used. The above paragraph is intended only to convey how the checking is done for this formal system, not to suggest that type theory has special status regarding machine-checkability of proofs.

of the type 'and P Q' from a proof of P and a proof of Q, and (2) given a term of type 'and P Q', we want to be able to extract a proof of P and a proof of Q. We get the desired behavior from this definition.

```
inductive and (P : Prop) (Q : Prop) : Prop :=
intro : P → Q → and P Q
```

This is Lean syntax for an inductive definition. `inductive` is a keyword indicating the beginning of an inductive definition. `and` is an identifier. Since

`(P : Prop)` and `(Q : Prop)`

appear before the colon, they are arguments for the type that is being defined. So, `and` takes two arguments of type `Prop`. After the colon is the type of the term being defined, here `Prop`. So, overall, this definition assigns to the identifier `and` the type

`Π(P : Prop) (Q : Prop), Prop`

(alternately, `Prop → Prop → Prop`). Since this is an inductive definition, to the right of `:=` is the list of base terms of the type and the constructors of the type.[16]

This inductive definition has no base terms, and it has one constructor. The constructor is referred to by `and.intro`. As required for inductive definitions, the return type of the constructor is the type being defined — here, `and P Q`. The constructor takes as arguments terms (i.e. proofs) of `P` and `Q`. So, using this constructor, we can — as desired — construct a term of type `and P Q` from a proof of `P` and a proof of `Q`:

given `(p : P)` and `(q : Q)`, `and.intro p q` is a term of type `and P Q`

Since the constructor is the means for creating a term of the type, it corresponds to what is typically called an introduction rule.

When a type is defined inductively, Lean automatically generates a 'destructor'/'recursor'; the 'destructor'/'recursor' allows the user to construct a function, where the argument type of the function is the inductively defined type. We give the reader a sense of these automatically generated recursors through examples. In the case of the type `and P Q`, the recursor allows us to define a function with argument type `and P Q`. This recursor provides the means for defining the desired elimination rules:

```
definition and.elim_left (H : and P Q) : P := and.rec_on H (λ x y, x)
definition and.elim_right (H : and P Q): Q := and.rec_on H (λ x y, y)
```

This is Lean syntax for a non-inductive definition. `definition` is a keyword indicating the beginning of a non-inductive definition. `and.elim_left` is an identifier. Since

`(H : and P Q)`

appears before the colon, `and.elim_left` is defined to take an argument of type `and P Q`. After the colon is the type of the term being defined, here `P`. So, overall, this defintion assigns to the identifier `and.elim_left` the type

[16] For non-inductive definitions, to the right of the `:=` is the term of the CIC assigned to the identifer by the definition. We give an example of this shortly.

```
Π{P : Prop} {Q : Prop} (H : and P Q), P
```

(alternately, Π `{P : Prop} {Q : Prop}, and P Q → P`). The identifier is assigned this type because the definition assigns to the identifer the term from the CIC after `:=`. Here, the term assigned to `and.elim_left` is

```
λ (H : and P Q), and.rec_on H (λ x y, x)
```

The initial λ comes from the arguments to the left of the colon; putting arguments to the left of the colon in a definition is an implicit lambda abstraction. So, given a term of type `and P Q` (e.g. `(hyp : and P Q)`), `and.elim_left hyp` is a term of type `P`.

`and.rec_on` is the automatically generated recursor. Its type is

```
Π {P : Prop}{Q : Prop}{C : Type}, and P Q → (P → Q → C) → C
```

In the definitions above, we take `C` to be `P` and `Q`. Thus, through using the recursor in the terms assigned to `and.elim_left` and `and.elim_right`, we get the other behavior we want from the `and P Q` type. Specifically, from a proof of `and P Q`, we can extract a proof of `P` and a proof of `Q`.

Lastly, we use Lean's resources for defining notation.

```
infix ∧ := and
```

Having given a concrete example of each, we give the general syntax for inductive definitions and non-inductive definitions.

```
inductive <identifier><hypotheses/implicit lambda abstractions> <:>
  <type expression> :=
| <identifier1> : <identifier>
...
| <identifierN> : <identifier>
| <identifierM> : ... → <identifier>
...
| <identiferNplusM> : ... → <identifier>

definition <identifier><hypotheses/implicit lambda abstractions> <:>
  <type expression> :=
  <term expression>
```

Defining 'or'

For arbitrary `(P : Prop)` and `(Q : Prop)`, the behavior we want from the type

```
or P Q
```

is this: (1) we want to be able to construct a term of type `or P Q` from a proof of `P` or from a proof of `Q`, (2) if we can (i) construct a term of type `C` from a term of type `P` and (ii) construct a term of type `C` from a term of type `Q`, then we can construct a term of type `C` from a term of type `or P Q`. We get the desired behavior from this definition.

```
inductive or (P : Prop) (Q : Prop) : Prop :=
|inl : P → or P Q
|inr : Q → or P Q
```

This inductive definition has no base terms and two constructors: `or.inl` and `or.inr`. `or.inl` is a function from a proof of `P` to a proof of `or P Q`. `or.inr` is a function from a proof of `Q` to a proof of `or P Q`. The automatically generated recursor is used to define the elimination rule.

```
definition or.elim (H : or P Q)(H1 : P → C)(H2 : Q → C) : C :=
  or.rec_on H H1 H2

infix ∨:= or
```

Defining 'implies'

The connective 'if...then...'/'implies' requires no work; for, we already have in the formal system a type that has the desired behavior. Instances of the Π type where the return type does not depend on the input type (i.e. what we represent with the →) provide what we want. To see that this is the case, let us consider the desired behavior for the 'implies' type: (1) we want to be able to construct a term of type `Q` by applying a term of type `implies P Q` to a term of type `P`, and (2) we want to be able to construct a term of type `implies P Q` by taking a term of type `P` and producing a term of type `Q`. This is exactly the behavior of the Π rules where the return type does not depend on the input term; see the derivation rules for Π above.

Defining 'false' and 'not'

For an arbitrary `(P : Prop)`, the definition for `not P` takes two steps. First, we define a type `false`. Then, we define `not P` as `P → false`.

```
inductive false : Prop
```

This inductive definition has no base terms and no constructors. The recursor allows us to construct a function from `false` to any type. These facts correspond to the desired behavior of: (1) there is no proof of `false`, and (2) from a proof of `false`, we can conclude anything.

```
definition not (P : Prop) : Prop := P → false

notation ¬ := not
```

Observe that we can present the rules as we would in a sequent calculus / natural deduction setting.

$$\frac{\Gamma \vdash \texttt{p : P} \qquad \Gamma \vdash \texttt{q : Q}}{\Gamma \vdash \texttt{and.intro p q : P} \land \texttt{Q}}$$

$$\frac{\Gamma \vdash \texttt{H : P} \land \texttt{Q}}{\Gamma \vdash \texttt{and.elim_left H : P}} \qquad \frac{\Gamma \vdash \texttt{H : P} \land \texttt{Q}}{\Gamma \vdash \texttt{and.elim_right H : Q}}$$

$$\frac{\Gamma \vdash \texttt{p : P}}{\Gamma \vdash \texttt{or.inl _ p : P} \vee \texttt{Q}} \qquad \frac{\Gamma \vdash \texttt{q : Q}}{\Gamma \vdash \texttt{or.inr _ q : P} \vee \texttt{Q}}$$

$$\frac{\Gamma \vdash \texttt{H : P} \vee \texttt{Q} \qquad \Gamma \vdash \texttt{H1 : P} \to \texttt{C} \qquad \Gamma \vdash \texttt{H2 : Q} \to \texttt{C}}{\Gamma \vdash \texttt{or.elim H H1 H2 : C}}$$

$$\frac{\Gamma \vdash \texttt{H : P} \to \texttt{Q} \qquad \Gamma \vdash \texttt{p : P}}{\Gamma \vdash \texttt{H p : Q}} \qquad \frac{\Gamma, (\texttt{p : P}) \vdash \texttt{q : Q}}{\Gamma \vdash \lambda \texttt{ p, q : P} \to \texttt{Q}}$$

$$\frac{\Gamma, (\texttt{p : P}) \vdash \texttt{c : false}}{\Gamma \vdash \lambda \texttt{ p, c : not P}} \qquad \frac{\Gamma \vdash \texttt{h : false}}{\Gamma \vdash \texttt{false.elim h : C}}$$

That concludes the presentation of the propositional connectives.

2.2.2 Defining predicates, relations, and quantifiers

Defining predicates and relations

We now define in CIC predicates and relations. Each is defined as a Prop-valued function. In the case that the Prop-valued function takes one argument, the function is called a predicate; in the case of more than one argument, the function is called a relation.

Given `(X : Type)`, the type of predicates on `X` is

$$\Pi \texttt{ (x : X), Prop}\text{; alternately, } \texttt{X} \to \texttt{Prop}$$

For example, if `(P : Π(x : X), Prop)` and `(z : X)`, then `P z : Prop`.

Given `(X : Type)` and `(Y : Type)`, the type of relations on `X` and `Y` is

$$\Pi\texttt{(x : X) (y : Y), Prop}\text{; alternately, } \texttt{X} \to \texttt{Y} \to \texttt{Prop}$$

For example, if `(R : Π(x : X) (y : Y), Prop)` and `(z : X)` and `(w : Y)`, then `R z w : Prop`.

Defining quantifiers

Given predicates and relations, we define in CIC universal and existential quantifiers. For the universal quantifier, the situation is similar to the situation for implication above; there is already an object in the language which has the desired properties: the Π type. However, this case is different from the implication case above, because for this case the return type will depend on the input term. Recall that the desired properties of the universal quantifier are as follows: (1) to introduce a universal quantifier, we take an arbitrary element of a domain and prove that the predicate holds for that element, and (2) to use/eliminate a universal quantifier, we apply the universally quantified statement to an element of the domain, resulting in the statement with the first universal quantifier removed and the element substituted for the quantified variable throughout the statement. These are exactly the rules for the Π type; again, see the rules in the previous section.

```
notation ∀ := Π
```

For the existential quantifier, there is work. The behavior we want for the existential quantifier is this: (1) to introduce an existential quantifier, we must produce an element of a domain and a proof that the predicate holds of that element, and (2) to use/eliminate an existential quantifier, if we can (a) take an arbitrary element of the domain, (b) assume that the predicate holds of that element, and (c) construct a term of type C, then from the existentially quantified statement we can construct a term of type C. A type with this behavior is defined in this way:

```
inductive exists (P : X → Prop): Prop :=
intro : Π (x : X), P x → exists P

notation ∃ := exists
```

So, given

$$\texttt{(X : Type) (x : X) (P : X} \rightarrow \texttt{Prop) (H : P x)}$$

`exists.intro x H` is a term of type `exists P`.

Using the recursor, we define the elimination rule.

```
definition exists.elim {A : Type} {P : A → Prop} {B : Prop}
  (H1 : ∃ x, P x) (H2 : ∀ (a : A), P a → B) : B :=
  exists.rec_on H1 H2
```

Again, we can represent the above rules in the natural deduction / sequent calculus format.

$$\frac{\Gamma \vdash \texttt{x : A} \qquad \Gamma \vdash \texttt{H : P x}}{\Gamma \vdash \texttt{exists.intro x H : } \exists \texttt{ y : A, P y}} \qquad \frac{\Gamma \vdash \texttt{H : } \Pi \texttt{ x : A, P x} \qquad \Gamma \vdash \texttt{y : A}}{\Gamma \vdash \texttt{H y : P y}}$$

$$\frac{\Gamma, \texttt{(x : A)} \vdash \texttt{H : P x}}{\Gamma \vdash \lambda \texttt{x : A, H} : \Pi \texttt{x : A, P x}}$$

$$\frac{\Gamma \vdash \texttt{M : } \exists \texttt{ x : A, P x} \qquad \Gamma \texttt{ (x : A), (H : P x)} \vdash \texttt{L : Q}}{\Gamma \vdash \texttt{exists.elim M (} \lambda \texttt{ x : A,} \lambda \texttt{ H : P x, L) : Q}}$$

We could also restate the rules with Π with ∀ to reflect its use as the universal quantifier.

2.2.3 Types dependent on types; types dependent on terms

In a previous section, we demonstrated the need for a formal system with dependent types by showing that a type theory with simple types could not describe certain objects of interest — i.e. types dependent on terms (e.g. vector $A\ n$). In this section, we define these in the calculus of inductive constructions.

We define lists of terms of type `A` exactly as one might expect. `nil` is a list of terms of type `A`, and — given a term `a` of type `A` and a list `l` of terms of type `A` – `cons a l` is a list of terms of type `A`. Using Lean notation, we define this type as follows.

```
inductive list (A : Type) : Type :=
| nil {} : list A
| cons   : A → list A → list A
```

Similarly, we define vectors of `n` terms of type `A`. `nil` is a vector of zero terms of type `A`, and — given a term `a` of type `A` and a vector `v` of `n` terms of type `A` — `cons a v` is a vector of `succ n` terms of type `A`. In Lean:

```
inductive vector (A : Type) : nat → Type :=
| nil {} : vector A zero
| cons   : Π {n}, A → vector A n → vector A (succ n)
```

2.3 Algebraic structures

It is fruitful to isolate and treat specially a certain subset of inductive types. Specifically, we isolate inductive types with one constructor and provide special means of working with them.

Lean provides a special syntax for defining inductive types of this sort, and types defined in this way are called *structures*. The method of defining a structure shares features with the method of defining objects in an object-oriented programming language; structures are like objects the following ways: (1) they store data and functions, (2) one structure can inherit from one or more other structures.

For example, we define a type of points in an integer-valued grid. We define it by defining a type `x_coordinate` which possesses one piece of data, and a type `y_coordinate` which also possesses one piece of data. The type `point` will inherit from each of these.

```
structure x_coordinate := (x : int)
structure y_coordinate := (y : int)
structure point extends x_coordinate, y_coordinate
```

Thus, given a term of type `x_coordinate` or `y_coordinate`, that term carries an `int`. To construct a term of type `x_coordinate` or `y_coordinate`, we must provide an `int`. For structures, the default name for the single constructor is `mk`; e.g., suppose `z : int`, then `x_coordinate.mk z` has type `x_coordinate`. Since it inherits from `x_coordinate` and `y_coordinate`, the type `point` possesses two pieces of data; so, to construct a term of this type, we must provide two `int`s. Given a term `pt` of type `point`, the data carried by the term can be accessed by `point.x pt` and `point.y pt`. If the `point` type possessed a field with a function, that field could be accessed in the same way. Accessing the fields of the structure using this '.' is nothing new: it is simply notation for using the automatically generated recursor to project out the components required to make a term of that type.

2.3.1 Type class inference

Proof assistants like Lean are designed to facilitate the construction of formal axiomatic derivations. Formal axiomatic derivations include many details. So, one way in which proof assistants can help users is by providing means through which users can leave out details, allowing the assistant to provide them. One of these means is referred to as 'type class inference', and Lean supports this.

For the purposes here, it suffices to characterize type class inference in the following way. It is a method of leaving out information in expressions. The method is invoked by

labeling certain types as `[class]`. Then, given a type `T` that has been labeled in this way, when writing expressions which require a term of type `T`, the user omits the term. To fill in this missing information, Lean consults a list of the terms in the context. We illustrate this with examples below.

Type class inference and structures can be used together to remarkable effect. We achieve this as follows. First, a structure is marked as a class. Next, the user introduces an instance of the structure into the context; this can be done by either adding it as a hypothesis or proving that the presence of the structure follows from hypotheses in the context. Then, in any definition/lemma/theorem in which the structure and its fields are used, the user can forgo mentioning the structure, allowing the type class inference mechanism to find the relevant structure.

An example will clarify the forgoing paragraphs. Suppose we wish to consider types with a multiplication operation. We can define a structure with a field that is an operation in this way.

```
structure has_mul (A : Type) := (mul : A → A → A)
```

So, given `(X : Type)`, `has_mul X` is a type; and, given a term of this type

`h : has_mul X`

the term has one field (i.e. `has_mul.mul h`), a function of type `X → X → X`. Now that we have this function, we want to use it. With `(x : X) (y : X)`, we can apply the function with `has_mul.mul h x y`. This expression is long and untidy. We can make it nicer by defining notation.

```
notation * := has_mul.mul
```

Now, we can apply the function with `* h x y`. This is an improvement; but, if there is only one `has_mul` we are considering, then we may wish to suppress the `h` and write something like `* x y` or change the notation to infix (i.e. `x * y`). Using type class inference allows us to do both of these things. To accomplish this, we change the definition of `has_mul` by marking the defined type as a class.

```
structure has_mul [class] (A : Type) := (mul : A → A → A)
```

Then, with `[h : has_mul X]` in the context[17], we can apply the function with the expression `* x y`. Alternately, we could change the notation

```
infix * := has_mul.mul
```

and write `x * y`. By marking the type as a class, the user indicates to Lean to make sense of expressions by filling in omitted terms with terms from the context. Here, `h` is omitted in the expressions and is filled in by Lean as needed.

Further, type class inference can use the information that one type marked as a class inherits from another type marked as a class. For example, consider the following.

```
structure has_one [class] (A : Type) := (one : A)
structure monoid [class] extends has_mul A, has_one A :=
  (one_mul : ∀ (x : A), mul one x = x)
  (mul_one : ∀ (x : A), mul x one = x)
```

[17]The hard brackets indicate that this hypothesis is available for use in type class inference.

Ignore the fields of `monoid`. Consider only the fact that it inherits `has_mul A`. Since it inherits from `has_mul A`, one of the implicit fields of `monoid` is `(mul : A → A → A)`. Given `[h : monoid A]`, we may want to use `h`'s `mul`. As desired, we can do so by writing `x * y`. Lean makes sense of the expression `x * y` by invoking type class inference. It find the instance of `monoid` (i.e. `h`). It observes that `monoid` inherits from `has_mul` and thus finds the `mul` needed to make sense of the expression.

Later, we build complex algebraic structures using these formal *structures*, and we reason seamlessly about these using the properties of *structures*, inheritance, and type class inference.

Chapter 3

Background from group theory

In this section, we present elementary concepts from group theory as they are presented informally. In the next section, we present the concepts formally.

Definition: group Suppose that G is a nonempty set. Suppose that $*$ is a binary operation; we call $*$ multiplication. Then G is a *group* iff

- G is closed under multiplication: for all a and b, if $a \in G$ and $b \in G$, then $a * b \in G$
- multiplication in G is associative: for all a, b, and c in G, $(a * b) * c = a * (b * c)$
- there is a unit element for the multiplication: there is an e in G such that for all a in G, $a * e = a$ and $e * a = a$.
- G contains inverse elements for the multiplication: for all a in G, there is a b in G such that $a * b = e$ and $b * a = e$

We denote the inverse element for a with a^{-1}.

Definition: subgroup Suppose that G is a group with multiplication $*$. Suppose that H is a nonempty subset of G. Then H is a *subgroup* of G iff

- H is closed under the multiplication in G: for all a and b, if $a \in H$ and $b \in H$, then $a * b \in H$
- H contains inverse elements for the multiplication in G: for all a in H, there is a b in H such that $a * b = e$ and $b * a = e$

Definition: left coset Suppose that G is a group with multiplication $*$. Suppose that H is a subset of G. Suppose that g is an element of G. Then the *left coset of H by g* is $\{g * h | h \in H\}$.

We denote the *left coset of H by g* with $g * H$ (alternately, gH), reusing the notation $*$. Similarly, we define the *right coset of H by g* and denote it with $H * g$ (alternately, Hg).

Definition: normalizer Suppose that G is a group with multiplication $*$. Suppose that H is a subset of G. Then the *normalizer of H* is $\{g \in G | g * H = H * g\}$.

Definition: normal subgroup Suppose that G is a group with multiplication $*$. Suppose that H is a subgroup of G. Then H is a *normal subgroup* of G iff for all $g \in G$, $g*H = H*g$.

Given a group G with multiplication $*$ and a subset H of G, we use the notion of *left coset of H* to define a relation on the elements of G.

Definition: coset relation Suppose that G is a group with multiplication $*$. Suppose that H is a subset of G. Then, for all g_1 and g_2 in G, $g_1 \sim g_2$ iff $g_1 * H = g_2 * H$.

By the reflexivity, symmetry, and transitivity of equality, this relation is an equivalence relation.

Definition: quotient group Given a group G with multiplication $*$ and a subset H of G, we can consider the new set $\{g * H | g \in G\}$. We denote this set with G/H. On this new set, we define a binary operation $\star$: given g_1 and g_2 in G, $(g_1 * H) \star (g_2 * H) := (g_1 * g_2) * H$. For economy of notation, instead of using $\star$, we reuse $*$; so, $(g_1 * H) * (g_2 * H) := (g_1 * g_2) * H$. Further, if H is a *normal subgroup*, then — using this operation as the multiplication — the new set is a group; we call this group the *quotient group of G by H.*

Given two groups G_1 and G_2, we consider functions between them. From the functions between G_1 and G_2, we distinguish functions with certain behavior with respect to the multiplications.

Definition: homomorphism Suppose G_1 and G_2 are groups. Let $*_{G_1}$ and $*_{G_2}$ be the multiplications in G_1 and G_2 respectively. A function f from G_1 to G_2 is a *homomorphism* iff for all a and b in G_1, $f(a *_{G_1} b) = f(a) *_{G_2} f(b)$.

We also define particular sets using functions between groups.

Definition: kernel Suppose that G_1 and G_2 are groups. Suppose f is a function from G_1 to G_2. Let 1_{G_2} denote the multiplicative identity in G_2. Then, the *kernel of f* is the set $\{g \in G_1 | f(g) = 1_{G_2}\}$.

The result which is the focus of this formalization is the first isomorphism theorem which relates the above concepts.

First isomorphism theorem: Suppose that G_1 and G_2 are groups. Suppose that f is a homomorphism from G_1 to G_2. Suppose f is onto from G_1 to G_2. Suppose K is the kernel of f. Then, K is a normal subgroup of G_1; there is a homomorphism g from G_1/K onto G_2; and, g is injective.

Chapter 4

The formalization

We saw above that the calculus of inductive constructions has two means for defining new types: (i) using the Π constructor on (a) base types and (b) types defined from base types and (ii) defining new types via inductive definitions and applying the Π constructor to these. Through the propositions-as-types interpretation, we defined propositions in the CIC; further, we defined connectives and quantifiers using inductive definitions. In this section, we define mathematical objects in the CIC.

Our goal in the formalization is this: find the best representation in type theory for familiar mathematical objects. For a representation to be acceptable, it must have the same properties as the familiar object. A representation is preferable to another if, compared to the other, (i) it is easier to reason about or (ii) it looks more similar to the familiar object. Throughout, we define notations which mimic familiar informal mathematics.

A couple notes

Because it is used frequently below, we restate the syntax for assigning an identifier to a term:

```
definition <identifier> <:> <type expression> :=
  <term expression>
```

In many cases, the system can infer the type, in which cases `<:> <type expression>` is optional.

For each of the objects defined, the reader can find the definitions and examples of their use in the relevant Lean files. For example[1]:

data.set.basic.lean

algebra.homomorphism.lean

theories.group_theory.basic.lean

4.1 Sets

In the standard library of Lean, the set type is defined as follows.

[1] The directory is here: `https://github.com/leanprover/lean/tree/master/library`

```
definition set := λ (A : Type), A → Prop
```

By this definition, we have assigned to the string `set` the term

$$\lambda\ \texttt{(A : Type), A} \to \texttt{Prop}$$

So, `set` has type Π`(A : Type), Type` (alternately, `Type → Type`). Further, given a type `X`, `set X` is the term `X → Prop` and has type `Type`.

Since it has type `Type`, `set X` is a type. And since `set X` is `X → Prop`, terms of type `set X` (e.g. `S` where `S : set X`) are functions from `X` to `Prop`. So, given

$$\texttt{(X : Type) (x : X) (S : set X)}$$

`S x : Prop`. Lastly, we define notation so that `x ∈ S` is `S x`.

```
notation x ∈ S := S x
```

Recall that `Prop` is a type, that terms of type `Prop` (e.g. `P : Prop`) are types and are interpreted as propositions, and that terms with type a proposition (e.g. `p : P`) are interpreted as proofs of the propositions. Given this and the notation above, a hypothesis that an element is in a set has the natural representation `(h : x ∈ S)`.

Before proceeding to further definitions, we recall an alternate syntax for the definition above. The above definition assigns to the identifier `set` the term `λ (A : Type), A → Prop`. We can acheive an identical assignment using this syntax:

```
definition set (A : Type) := A → Prop
```

In this alternate syntax, instead of having the explicit λ abstraction over the type variable `A` to the right of `:=`, we have an implicit λ abstraction to the left of `:=`. This alternate syntax is used frequently in the Lean library, and we use it below. Also, it is a notational convention to use one λ followed by multiple variables to abbreviate $\lambda\, var1, \lambda\, var2, ...$; later examples use this convention.

4.1.1 Operations on sets

In the standard library of Lean, operations on sets are defined as follows.

```
definition intersection :=
  λ {A : Type} (S : set A) (T : set A), λ (a : A), a ∈ S ∧ a ∈ T
definition union :=
  λ {A : Type} (S : set A) (T : set A), λ (a : A), a ∈ S ∨ a ∈ T
```

By these definitions, we have assigned to the strings `intersection` and `union` the corresponding terms. There are squiggly braces around `A : Type`; the consequence of the squiggly braces is this: when we write `intersection ...`, we do not pass as argument the type of the sets. For example, given `(X : Type)`$($`S`$_1$ `: set X)`$($`S`$_2$ `: set X)`, we write `intersection` S_1 S_2, rather than `intersection X` S_1 S_2. `intersection` S_1 S_2 is the term `λ (a : X), a ∈` S_1 `∧ a ∈` S_2; so, `intersection` S_1 S_2 has type `X → Prop`; and so, it has the right type to be a term of `set X`.

Also, we define notation

```
infix ∩  := intersection
infix ∪  := union
```

Thus, we can introduce the hypothesis that an element is in one of these sets with the natural notation `(h_a : x ∈ S₁ ∩ S₂)` or `(h_b : x ∈ S₁ ∪ S₂)`. And Lean can verify the identity between `x ∈ S₁ ∩ S₂` and `x ∈ S₁ ∧ x ∈ S₂` by reducing them to a common term.

```
example {X : Type} (S₁ : set X) (S₂ : set X) (x : X) :
  (x ∈ (S₁ ∩ S₂)) = ((x ∈ S₁) ∧ (x ∈ S₂)) := rfl
```

Lean permits users to set precedences on operators. As a result, users can omit parentheses as desired.

As we did in the section for sets, we present an alternate syntax for the above definitions. In the definitions for `intersection` and `union`, we λ-abstract over the type variable, but we surround this λ-abstraction with squiggly braces because we do not want to pass the type variable as explicit argument. In a sense, the type is in the background. We can simultaneously (a) represent that the type is in the background and (b) achieve the same definitions as above using the following syntax:

```
section
  variable {A : Type}

  definition intersection (S : set A) (T : set A) : set A :=
    λ (a : A), a ∈ S ∧ a ∈ T
  definition union (S : set A) (T : set A) : set A :=
    λ (a : A), a ∈ S ∨ a ∈ T
end
```

Two things changed in the transition between definitions: first, we used the implicit λ abstraction to the left of `:=` as discussed in the section for sets above; second, we used a variable declaration (i.e. `variable {A : Type}`). Variable declarations make the declared terms available for use in definitions. Definitions which do not use one or more of the declared variables do not depend on those variables. Here, the definitions of `intersection` and `union` do rely on the type variable, so the definitions assign a term which includes this; e.g. to `intersection` is assigned

```
λ {A : Type} (S : set A) (T : set A), λ (a : A), a ∈ S ∧ a ∈ T
```

as desired.[2]

[2]For more on `section`s and variable declarations, see tutorial [2].

Lastly, we exhibit a defined set-builder notation for sets.

```
section
  variable {A : Type}

  definition intersection (S : set A) (T : set A) : set A :=
    {a : A | a ∈ S ∧ a ∈ T}
  definition union (S : set A) (T : set A) : set A :=
    {a : A | a ∈ S ∨ a ∈ T}
end
```

`{a : A | a ∈ S ∧ a ∈ T}`

is notation for `λ (a : A), a ∈ S ∧ a ∈ T`.

4.1.2 Relations between sets

We define a relation on sets as follows.

```
section
  variable {A : Type}

  definition subset (S : set A) (T : set A) : Prop :=
    ∀ (a : A), a ∈ S → a ∈ T
end
```

The definition assigns to the string `subset` the corresponding term. Thus, the type of `subset` is

`Π {A : Type}, set A → set A → Prop`

Again, we write `subset` S_1 S_2 rather than `subset X` S_1 S_2. `subset` S_1 S_2 is

`∀ (a : X), a ∈` S_1 `→ a ∈` S_2

Hence, `subset` S_1 S_2 is a term of `Prop`.

We define notation:

```
infix ⊆ := subset
```

Given this notation, a hypothesis that one set is a subset of another has the natural form: `(h :` $S_1 \subseteq S_2$`)`.

4.1.3 Example of proof

We have now defined enough logical operations, mathematical objects, relations, and notation to state and prove the following statement: Given sets A, B, and C, if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.

An informal argument is as follows. Suppose that $A \subseteq B$. Suppose that $B \subseteq C$. Take an arbitrary x. Suppose $x \in A$. Since $x \in A$ and $A \subseteq B$, $x \in B$. Since $x \in B$ and $B \subseteq C$,

$x \in C$. Thus, if $x \in A$, then $x \in C$. Because x is arbitrary, for all x if $x \in A$ then $x \in C$; that is, $A \subseteq C$.

This informal argument is represented by the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \in A} \quad \dfrac{\dfrac{A \subseteq B}{\forall w, w \in A \to w \in B}}{x \in A \to x \in B}}{x \in B} \quad \dfrac{\dfrac{B \subseteq C}{\forall y, y \in B \to y \in C}}{x \in B \to x \in C}}{x \in C}}{x \in A \to x \in C}}{\forall x, x \in A \to x \in C}}{A \subseteq C}$$

The expressions on this proof tree are propositions. In the setting of the CIC, propositions are types; so, below, we construct a tree in which we represent this. Having a hypothesis is taking as given a term of the proposition-type. Later terms are built from hypotheses in accordance with the derivation rules. Let H_{AB} be the hypothesis that $A \subseteq B$; similarly H_{BC} for $B \subseteq C$. With these annotations, the proof tree is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H_x : x \in A} \quad \dfrac{\dfrac{H_{AB} : A \subseteq B}{H_{AB} : \forall w, w \in A \to w \in B} \quad \dfrac{}{x : A}}{(H_{AB}\ x) : x \in A \to x \in B}}{(H_{AB}\ x) H_x : x \in B} \quad \dfrac{\dfrac{H_{BC} : B \subseteq C}{H_{BC} : \forall y, y \in B \to y \in C} \quad \dfrac{}{x : A}}{(H_{BC}\ x) : x \in B \to x \in C}}{(H_{BC}\ x)((H_{AB}\ x)\ H_x) : x \in C}}{\lambda H_x, (H_{BC}\ x)((H_{AB}\ x)\ H_x) : x \in A \to x \in C}}{\lambda x, \lambda H_x, (H_{BC}\ x)((H_{AB}\ x)\ H_x) : \forall x, x \in A \to x \in C}}{\lambda x, \lambda H_x, (H_{BC}\ x)((H_{AB}\ x)\ H_x) : A \subseteq C}$$

The term on the bottom line of the proof tree is a term of the desired type. We present the proof in Lean:

```
example {X: Type} (A : set X) (B : set X) (C : set X) (HAB : A ⊆ B)
  (HBC : B ⊆ C) : A ⊆ C :=
  λ x Hx, (HBC x)((HAB x) Hx)
```

In these lines, we have stated that, given

```
{X: Type} (A : set X) (B : set X) (C : set X) (HAB : A ⊆ B)
(HBC : B ⊆ C)
```

the term `λ x Hx, (HBC x)((HAB x) Hx)` has type `A ⊆ C`. Moreover, since

`A ⊆ C : Prop`,

`λ x Hx, (HBC x)((HAB x) Hx)` is a proof of `A ⊆ C`.

4.2 Functions

Function types are instances of the Π type, and functions are terms of these types. In this section, we define relations and predicates involving functions. As examples, we define the relations of the image of a function on a set and the preimage of a function on a set and the predicates of injectivity and surjectivity.

```
section
  variables {A B : Type}

  definition image (f : A → B) (S : set A) : set B :=
    { y : B | ∃ x : A, x ∈ S ∧ f x = y }
  definition preimage (f : A → B) (T : set B) : set A :=
    { x : A | f x ∈ T }
  definition injective (f : A → B) : Prop :=
    ∀ (x : A)(y : A), (f x = f y → x = y)
  definition inj_on (f : A → B) (S : set A) : Prop :=
    ∀ {x : A}{y : A}, (x ∈ S → y ∈ S → f x = f y → x = y)
  definition surjective (f : A → B) : Prop :=
    ∀ (y : B), ∃ (x : A), f x = y
  definition surj_on (f : A → B)(S : set A)(T : set B) : Prop :=
    ∀ (y : B), (y ∈ T → ∃ (x : A), (x ∈ S ∧ f x = y))

end
```

Given `(f : A → B)`, `injective f` asserts that `f` is injective on the whole type A; whereas given `(S : set A)`, `inj_on f S` asserts only that `f` is injective on the set `S`. Similarly for `surjective` and `surj_on f S`. So, we see that we can state and prove function properties with respect to the whole type or restricted to a set on the type.

4.3 Operations on a type

In preparation for constructing groups, let us consider defining operations on a type and constraining the operations.

```
definition unary_operation (A : Type) := A → A
definition binary_operation (A : Type) := A → A → A
definition is_commutative (A : Type) (op : A → A → A) :=
  ∀ (a : A)(b : A), (op a b = op b a)
```

`unary_operation` and `binary_operation` are types. Given `(X : Type)`, `(f : unary_operation X)` states that `f` is a unary operation on `X` — a function from `X` to `X`. Similarly, `(g : binary_operation X)` states that `g` is a binary operation on `X` — a function from `X` to `X` to `X`.

As stated, `f` and `g` are arbitrary functions; we know nothing about their behavior. However, using other defined terms, we can stipulate behavior. Consider

`is_commutative X g;`

it reduces to the term `∀ (a : X)(b : X), (g a b = g b a)`; its type is `Prop`. A term of type `is_commutative X g` — alternately stated: a term of type

```
∀ (a : X)(b : X), (g a b = g b a)
```

— is a proof that g is commutative. Consider `(h : is_commutative X g)`; given

```
(x : X) (y : X)
```

`h x` is a proof that `∀ (b : X), g x b = g b x`, and `h x y` is a proof that

```
g x y = g y x
```

4.3.1 Functions interacting with operations

Given a type, an operation on that type, and a function with that type as domain, we can stipulate the behavior of the function with respect to that operation. For example,

```
section
  variable (A : Type)

  definition is_distributive (f : A → A) (op : A → A → A) :=
    ∀ (x : A)(y : A),(f (op x y) = op (f x) (f y))
end
```

Thus, given `(X : Type) (g : X → X) (op_ex : X → X → X)`,

```
is_distributive X g op_ex
```

reduces to `∀ (x : X)(y : X), (g (op_ex x y) = op_ex (g x) (g y))`. So, if we have a term `h` of this type, then `h` is a proof that `g` distributes over `op_ex`. And, using `h` and terms of type `X`, we can prove equalities for those terms.

4.4 Groups

We build the type of groups using structures.[3] Throughout the construction of the group type, we mark the `structure`s as type classes (i.e. `[class]`). We claimed above and demonstrate below that this combination of structures and type classes facilitates reasoning with these types and provides means for natural notation. Additional benefits of defining types incrementally include: (1) we can state and prove results at the exact level of generality at which they hold — e.g. we can prove results about monoids using only the facts about them; (2) we can define objects using exactly the hypotheses they need and no more — e.g. we can define homomorphisms between any two types with `has_muls`, the concept does not require groups on each type. These benefits make the structures general and reusable in other contexts.

[3] Recall that (1) structures are a special case of an inductive type: an inductive type with one constructor and (2) structures can be built incrementally through inheritance.

Our goal is to define a group type such that: (1) the group type depends on a type — with this, we can form groups on arbitrary types (e.g. `(A : Type)`) or on concrete types (e.g. `(nat : Type)`); (2) the group type has a multiplication; (3) it has a unit element with respect to the multiplication; (4) it has a means for referring to inverse elements; (5) there is a constraint on the multiplication — it is associative; (6) there is a constraint on the unit element – given `(A : Type) (a : A)`, multiplying `a` on the left or right by the unit element is equal to `a`; (7) there is a constraint on the inverse elements — given `(A : Type) (a : A)`, multiplying `a` on the left or the right by its inverse is equal to the unit element.

We build this type through a few steps. In the next section, we give an overview of the steps and provide a picture. In the section after that, we consider each step individually.

4.4.1 Building the group type: overview

Suppose that we have `(A : Type)`. The first structure we define is `has_mul A`. `has_mul A` has one field `(mul : A → A → A)`. Next, we define a structure `semigroup A` which inherits from `has_mul A`. `semigroup A` has one field

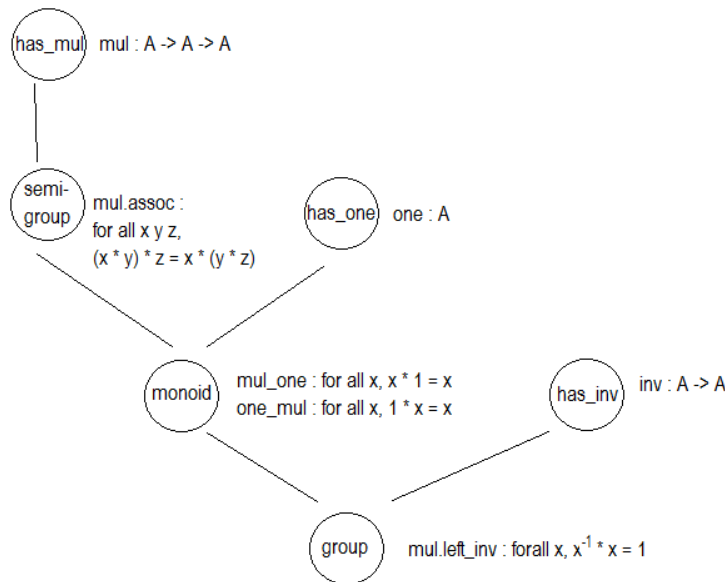
```
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))
```

Now, we define a structure separate from `has_mul A` and `semigroup A`; we define `has_one A`, which has one field `(one : A)`. Next, we define a structure `monoid A` which inherits from `semigroup A` and `has_one A`. `monoid A` has two fields:

```
(mul_one : ∀ (x : A), mul x one = x)
(one_mul : ∀ (x : A), mul one x = x)
```

Now, we again define a separate structure: we define `has_inv A` with one field `(inv : A → A)`. Finally, the structure `group A` inherits from `monoid A` and `has_inv A`. `group A` has the new field `(mul_left_inv : ∀ (x : A), mul (inv x) x = one)`.

The picture below depicts the steps described above.



4.4.2 Building the group type: increments

We now consider each step in the construction of the group type and that step's relation to our goal. The first structure we define is `has_mul`.

```
structure has_mul [class] (A : Type) := (mul : A → A → A)
```

The first observation is that the structure depends on a parameter — that is, it takes an argument. The argument is a type. So, given types `(X: Type)` `(nat : Type)`, we can have `has_mul X` and `has_mul nat`. The second observation is that the field of this structure — `mul` — is a binary operation. For `has_mul X`, the field is `mul : X → X → X`, and similarly for `has_mul nat`, it is `mul : nat → nat → nat`.

Next, we define `semigroup`.

```
structure semigroup [class] (A : Type) extends has_mul A :=
  (mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))
```

Again, the structure depends on a parameter. The keyword `extends` indicates that `semigroup` inherits from `has_mul`. Specifically, inheritance is the following behavior: suppose S_1 and S_2 are structures; if S_2 inherits from S_1, then S_2 has all the fields of S_1 in addition to the fields specific to S_2. So, `semigroup` has two fields, `mul` and `mul_assoc`. Note that the second field uses the first — `mul_assoc` uses the `mul`. We use these two fields as the desired (a) multiplication and (b) stipulation that the multiplication is associative.

Next, we define `has_one`.

```
structure has_one [class] (A : Type) := (one : A)
```

The field `(one : A)` provides a term of the type. In `monoid`, we use this term as the unit element with respect to the multiplication.

```
structure monoid [class] (A : Type) extends semigroup A, has_one A :=
  (mul_one : ∀ (x : A), mul x one = x)
  (one_mul : ∀ (x : A), mul one x = x)
```

`monoid` provides an example of a structure inheriting from two separate structures. It also provides an example of a structure with multiple fields. Its fields — `mul_one` and `one_mul` — provide the desired stipulations on the behavior of `one` with respect to the multiplication.

Now, we define `has_inv`.

```
structure has_inv [class] (A : Type) := (inv : A → A)
```

The field of `has_inv` is a unary operation. We use this operation to represent inverse elements.

Finally, we define the group type.

```
structure group [class] (A : Type) extends monoid A, has_inv A :=
  (mul_left_inv : ∀ (x : A), mul (inv x) x = one)
```

`group` possesses all the fields of `has_inv` and `monoid`, and all the fields of the structures `monoid` inherits from, recursively. The new field `mul_left_inv` provides the desired stipulation that left multiplication of an element by its inverse element equals the unit element. The corresponding constraint on right multiplication is derivable from what we have.

Consequence of structures and type classes: natural notation

We define notation for the multiplication, inverse, and one.

```
infix *    := has_mul.mul
postfix ⁻¹ := has_inv.inv
notation 1 := has_one.one
```

By relying on type class inference and using these notations, we represent group elements and group operations with natural expressions. In the example below, we declare a type, terms of that type, and a term of the group type for the given type. Using these, we write expressions. In the example, for the declaration of the term of the group type, I enclose the declaration in hard brackets (i.e. '[' and ']'). These hard brackets tell Lean to use type class inference to find this term.

```
section
  variables {X : Type} [G : group X] (x : X) (y : X)

  check x                    -- x : X
  check x⁻¹                  -- x⁻¹ : X
  check x * y                -- x * y : X
  check (x * y⁻¹) * y = x    -- (x * y⁻¹) * y = x : Prop

end
```

Notice that the expressions `x⁻¹` and `x * y` do not mention `G`. We declared as classes the types related to the notations. In the expressions, the presence of the relevant type is presupposed. When $^{-1}$ or `*` or `1` are used in an expression, Lean invokes class type inference to look for the presupposed types — in this case, a `has_mul` or `has_inv`. Here, Lean finds a `group`; and, in the `group`'s fields, it finds the `has_mul` and `has_inv`.

In this example, there is only one term of the group type in the context. Because we do not use its name and because there is no need to distinguish it from other terms of this type in the context, we can declare it anonymously.

```
section
  variables {X : Type} [group X] ...
end
```

In these expressions, the notation and type class inference allow the user to suppress much information. Without the notation, `x * y` would be `mul x y`. Without type class inference tracing inheritance, the user would have to show Lean where to find the `mul` in the fields of `group`. Without the implicit arguments allowed by type class inference, in each expression where one of the fields is used, the user would need to state the term the field's of which are being accessed.

4.4.3 Example proof

We have now defined enough operations and stipulations to state and prove claims about group types. For example, we can state and prove claims about terms in a type

which has a group type. An informally-stated claim of this sort is the following: Suppose that G is a group. Suppose a and b are elements of G. Then, $(a * b^{-1}) * b = a$.

An informal argument is as follows.

$$
\begin{aligned}
(a * b^{-1}) * b &= a * (b^{-1} * b) && \text{associativity of multiplication} \\
&= a * 1 && \text{substitution of } (b^{-1} * b) = 1 \\
&= a && \text{right multiplication by identity}
\end{aligned}
$$

This informal argument is represented in the following proof tree:

$$
\dfrac{
  \dfrac{
    \dfrac{\forall xyz, (x * y) * z = x * (y * z)}{(a * b^{-1}) * b = a * (b^{-1} * b)}
    \quad
    \dfrac{
      \dfrac{\forall \{x\}, x = x}{a * (b^{-1} * b) = a * (b^{-1} * b)}
      \quad
      \dfrac{\forall x, x^{-1} * x = 1}{b^{-1} * b = 1}
    }{a * (b^{-1} * b) = a * 1}
  }{(a * b^{-1}) * b = a * 1}
  \quad
  \dfrac{\forall x, x * 1 = x}{a * 1 = a}
}{(a * b^{-1}) * b = a}
$$

As we did in a previous section for the proof regarding sets, we annotate this proof tree with terms. The term on the bottom line is a proof of the conclusion.

$$
\dfrac{
\dfrac{
\dfrac{\text{mul_assoc: } \forall xyz, (x*y)*z = x*(y*z)}{\text{mul.assoc a b}^{-1}\text{ b} : (a*b^{-1})*b = a*(b^{-1}*b)}
\quad
\dfrac{
\dfrac{\text{rfl} : \forall\{x\}, x = x}{\text{rfl} : a*(b^{-1}*b) = a*(b^{-1}*b)}
\quad
\dfrac{\text{mul_left_inv} : \forall x, x^{-1}*x = 1}{\text{mul_left_inv b} : b^{-1}*b = 1}
}{\text{eq.subst (mul_left_inv b) rfl} : a*(b^{-1}*b) = a*1}
}{\text{eq.trans (mul_assoc a b}^{-1}\text{ b) (eq.subst (mul_left_inv b) rfl)} : (a*b^{-1})*b = a*1}
\quad
\dfrac{\text{mul_one} : \forall x, x*1 = x}{\text{mul_one a} : a*1 = a}
}{\text{eq.trans (eq.trans (mul_assoc a b}^{-1}\text{ b) (eq.subst (mul_left_inv b) rfl)) (mul_one a)} : (a*b^{-1})*b = a}
$$

In Lean, one representation of the proof is:

```
example : (a * b^{-1}) * b = a :=
eq.trans (eq.trans (mul_assoc a b^{-1} b) (eq.subst (mul_left_inv b) rfl))
      (mul_one a)
```

Alternately, we can represent this proof using another tool provided by Lean — the calculation environment. This environment is designed for working with relation symbols which support transitivity reasoning. For proofs involving equality, this environment provides a means of representing formal proofs exactly like their informal counterparts.

```
example : (a * b^{-1}) * b = a :=
  calc
         (a * b^{-1}) * b = a * (b^{-1} * b) : mul_assoc
                   ... = a * 1          : mul_left_inv
                   ... = a              : mul_one
```

[4]

4.5 Objects dependent on group structure

With the group type in place, we construct types which depend on some or all of the attributes of groups. We construct subgroups, cosets, normal sets, and homomorphisms.

4.5.1 Subgroups

Our goal is to define a subgroup predicate such that: (1) a subgroup depends on a group, (2) a group can have multiple subgroups, (3) subgroups are closed under the group multiplication, (4) subgroups are closed under the group inverse. To accomplish this, we construct the subgroup predicate incrementally using structures[5], where intermediate structures have some of the properties we want the subgroup predicate to have. As we did for groups, we describe the steps, depict the process with a picture, then consider each step.

Suppose `(A : Type) [group A] (S : set A)`. The first structure we define is `is_mul_closed S`. `is_mul_closed S` has one field:

```
(mul_mem : ∀ (x : A)(y : A), x ∈ S → y ∈ S → x * y ∈ S)
```

Next, we define `is_inv_closed S`. `is_inv_closed S` has one field:

```
(inv_mem : ∀ (x : A)(y : A), x ∈ S → x^{-1} ∈ S)
```

Then, we define `is_one_closed S`. `is_one_closed S` has one field

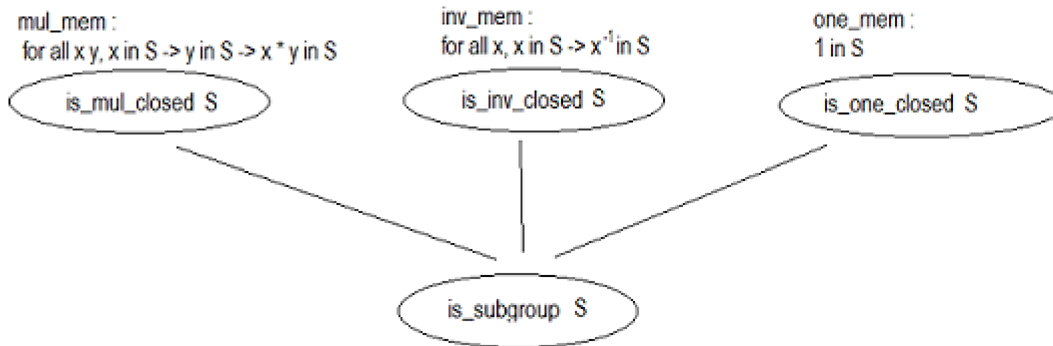
```
(one_mem : one ∈ S)
```

[4]Another option is rewriting:
`example : (a * b^{-1}) * b = a := by rewrite[mul_assoc, mul_left_inv, mul_one]`

[5]Notice that in this use of structures the type defined has type `Prop`. Previously, we used structures to define types with type `Type`.

Finally, `is_subgroup S` inherits from these three structures and adds no new fields.

The picture below depicts the steps described above.



We now consider the steps of the construction. The first structure is `is_mul_closed`. It takes an implicit argument — a type. In the context where it is defined, `A` has been declared as a type (i.e. `(A : Type)`). Outside this context, this same structure can be used for different types just as we indicated for groups.

```
structure is_mul_closed [class] [has_mul A] (S : set A) :=
(mul_mem : ∀ (x : A)(y : A), x ∈ S → y ∈ S → x * y ∈ S)
```

The structure is marked as a class. Later, `is_subgroup` inherits from this structure; and the upshot is that, if we have `(T : set A)` and `[is_subgroup T]` in the context, Lean can find the `[is_mul_closed T]`. Since this structure does not use `inv` or `one`, it only requires that there is a `has_mul` in the context, rather than a `group`. The field of `is_mul_closed` stipulates that the set is closed under the multiplication.

Next, we define `is_inv_closed`.

```
structure is_inv_closed [class] [has_inv A] (S : set A) : Prop :=
(inv_mem : ∀ (x : A), x ∈ S → x⁻¹ ∈ S)
```

The field provides the stipulation that the set is closed under the inverse operation.

Next, we define `is_one_closed`.

```
structure is_one_closed [class] [has_one A] (S : set A) : Prop :=
(one_mem : one ∈ S)
```

The field provides the stipulation that the group identity element is in the set.

Finally, we define `is_subgroup`.

```
structure is_subgroup [class] [group A] extends is_mul_closed A,
  is_inv_closed A, is_one_closed A
```

All of the above structures take as an argument a `set` on the type. Because there can be multiple sets on a type, given a group on the type there can be multiple subgroups of the group. Between this and the stipulations imposed by the fields `is_subgroup` inherits, we have constructed a type that achieves our goal for the subgroup type.

4.5.2 Cosets

Given `{X : Type} (S : set X) [has_mul X] (x : X)`, we consider left and right cosets of S by x.

```
section
  variables {X : Type}[has_mul X]

  definition lcoset (S : set X) (x : X) := image (mul x) S
  definition rcoset (S : set X) (x : X) := image (mul^~ x) S

end
```

The `^~` in the definition of `rcoset` is notation for the following procedure: take a binary operation, reverse the order in which the operands are given. So here, it reverses the order in which the operands for `mul` are given; with `arg1` and `arg2`, `mul^~ arg1 arg2` is `mul arg2 arg1`.

Given a particular set `(T : set X)` and a particular element `y : X`, `lcoset T y` is the `set` composed of elements of the form `y * t` for some `t` in `T`. This corresponds with the informal coset object, as does the notation.

```
infix * := lcoset
infix * := rcoset
```

So, `y*T` and `T*y` are the left and right coset of `T` by `y`. Notice that Lean supports the overloading used informally.

4.5.3 Normal sets

Given `(A : Type) (S : set A) [has_mul A]`, we consider whether `S` is normal. In informal group theory, *normal subgroups* are considered. In our formalization, we separate the properties of a set being a *subgroup* and a set being *normal*. Again, this allows us to isolate and prove the results depending on one property but not the other.

```
section
  variables {A : Type} [has_mul A]

  definition normalizes (a : A)(S : set A) : Prop := a * S = S * a

  definition is_normal [class] (S : set A) : Prop :=
    ∀ (a : A), normalizes a S

  definition normalizer (S : set A) : set A :=
    { a : A | normalizes a S}
end
```

4.6 Relations dependent on group structure

Recall that we can consider a relation on an arbitrary type. For example, given `(X : Type)`, a two-place relation on `X` is a term of the type `X → X → Prop`. We can also consider relations on particular types. For example, a two-place relations on `set X` is a term of the type `set X → set X → Prop`.

One such relation is essential to the formalization. That relation, expressed informally is this: Given a group G, a subset H of G, and elements a and b in G

$$a \sim b \iff aH = bH$$

This is an equivalence relation, and it is the equivalence relation we later use to define the quotient type and quotient group. In the formalization, we represent this relation as follows.

```
section
  variables {X : Type} [has_mul X]

  definition lcoset_equiv (H : set X) (a : X) (b : X) := a*H = b*H

end
```

That `lcoset_equiv` is an equivalence relation follows immediately from the reflexivity, symmetry, and transitivity of equality. These proofs can be given as:

```
lemma lcoset_equiv_refl (S : set X): reflexive (lcoset_equiv S) :=
  λ x, rfl
lemma lcoset_equiv_symm (S : set X) : symmetric (lcoset_equiv S) :=
  λ x y H, eq.symm H
lemma lcoset_equiv_trans (S : set X) : transitive (lcoset_equiv S) :=
  λ x y z Hxy Hyz, eq.trans Hxy Hyz
```

where `reflexive`, `symmetric`, and `transitive` are defined as expected.

```
section
  variables {X: Type} (R : X → X → Prop)
  definition reflexive := ∀ x, R x x
  definition symmetric := ∀ x y, R x y → R y x
  definition transitive := ∀ x y z, R x y → R y z → R x z
  definition equivalence :=
    (reflexive R) ∧ (symmetric R) ∧ (transitive R)

  lemma equivalence_lcoset_equiv (H : set X) :
    equivalence (lcoset_equiv H) :=
    and.intro (lcoset_equiv_refl H)
              (and.intro (lcoset_equiv_symm H) (lcoset_equiv_trans H))
end
```

4.7 Homomorphism and kernel

Given two types `(X : Type) (Y : Type)`, we consider functions between them — i.e. terms of the type `X → Y`. Further, given 'structure' on the types (e.g. operations like multiplication), we can stipulate the behavior of functions with respect to this 'structure'. In the section 'Functions interacting with operations', `is_distributive` is an example of this.

Suppose that `X` and `Y` have multiplication operations. That is, suppose `[has_mul X] [has_mul Y]`. One stipulated behavior is of special interest — the behavior of a function being *homomorphic* with respect to the multiplications.

```
section
  variables {X Y : Type} [has_mul X] [has_mul Y]

  structure is_hom [class] (f : X → Y): Prop :=
    (hom_mul : ∀ a b, f (a * b) = f a * f b)
end
```

There is a subtlety in this definition. The objects `a` and `b` are terms in `X`, and `a * b` is an application of the multiplication in `X`. The objects `f a` and `f b` are terms in `Y`, and `f a * f b` is an application of the multiplication in `Y`. The use of the same `*` hides this difference, but it reflects informal notation. Lean supports this reuse of notation; that it can is another consequence of type class inference.

A second object dependent on 'structure' and a function is the kernel. The relevant 'structure' is a unit element on the target type.

```
section
  variables {X Y : Type} [has_one Y]

  definition ker (f : X → Y) : set X := {x : X | f x = 1}
end
```

4.8 Quotient types and quotient groups

In informal mathematics, the procedure of forming quotients can be described as follows. Suppose that we have a collection of objects S. Suppose that we define a binary relation $\approx$ on S. Suppose that $\approx$ is reflexive, symmetric, and transitive. Then we do the following:

- for each x in S, consider $[x]$; where $[x] := \{s \in S \mid s \approx x\}$
- define a new collection of objects T, where the objects in T are exactly the $[x]$ for x in S (i.e. $T := \{\ [x] \mid x \in S\}$)
- given a function f with domain S, if we prove

  for all a b in S, if $a \approx b$, then $f\,a = f\,b$

  then we can construct a function g with domain T s.t. $g\,[x] = f\,x$, for all x in S.

· Given a predicate P, to prove a statement of the form

for all t in T, P t

it suffices to prove

for all s in S, P [s]

4.8.1 Quotients in Lean: introducing constants

The calculus of inductive constructions does not have a builtin notion of quotient. Moreover, there is no way to *define* quotients using the resources of CIC. So, to get the feature of quotients, we must *add* them. In Lean, we do this by adding constants — that is, we do this by stipulating that certain identifers have certain types. These constants allow us to do in the formal setting what we do informally. Specifically, we add the constants `quot`, `quot.mk`, `quot.sound`, `quot.lift`, and `quot.ind` .[6]

Setoids

These constants make use of a structure we define: `setoid`. We have not yet discussed `setoids`; so, we first discuss these then present the constants. Given `(A : Type)`, a `setoid A` is a compound object which contains a relation on `A` and a proof that the relation is an equivalence relation.

```
structure setoid [class] (A : Type) :=
(r : A → A → Prop) (iseqv : equivalence r)
```

Just as in the informal case, the data for the formal quotient construction are: a collection (here, a type), a relation on the collection (here, the relation on the type), and a proof that the relation is an equivalence. In the formal setting, we bundle this data in a `setoid`.

Quotient constants

Given this, we present the constants:

```
constant quot.{l} : Π{A : Type.{l}}, setoid A → Type.{l}

constant quot.mk  : Π {A : Type} [s : setoid A], A → quot s

constant sound    : Π {A : Type} [s : setoid A] {a b : A},
                      a ≈ b → [a] = [b]

constant lift     : Π {A B : Type} [s : setoid A] (f : A → B),
                      (∀ a b, a ≈ b → f a = f b)→ quot s → B

constant ind      : ∀ {A : Type} [s : setoid A] {B : quot s → Prop},
                      (∀ a, B [a]) → ∀ q, B q
```

[6]From these we can derive others, e.g. `quot.lift₂`, `quot.ind₂`. In short, these others are extensions of the constants, the purpose of which extensions is defining relations and proving statements about them.

These constants correspond to the actions from informal mathematics listed above.

`quot` constructs the new collection of objects. From the data of (i) a type and (ii) a setoid on the type, `quot` constructs a new type.[7] Since a setoid is simply a packaged equivalence relation, we see concretely that the data for this formal version is the same as the data for the informal case.

`quot.mk` takes an element in the base type to its corresponding element in the quotient type. Applying `quot.mk` to an element corresponds to making $[x]$ from an x in the original collection.

`quot.sound` encodes the principle that two elements which are related in the base type are equal in the quotient type.

`quot.lift` encodes the principle that, if there is a function `f` on the original type and we prove that `f` respects the relation, then we can use this function `f` to construct a function `g` on the quotient type. That the two functions are such that `g [a]` = `f a` for all `a : A` is stipulated.[8]

Finally, `quot.ind` encodes the principle that to show that a property holds for all elements of the quotient type, it suffices to show that the property holds of all equivalence classes.

Together, these constants stipulate how to construct the new type, stipulate how the elements of the new type are related to the old type, and provide the means of reasoning about the new type as we reason about it informally.

4.8.2 Quotient groups overview: informal and formal

Recall that in informal group theory, there is a construction referred to as a quotient group. In this construction, we take a group G with multiplication $*$ and a subset H of G; we consider the new set $G/H := \{g * H | g \in G\}$; on this new set, we define a particular multiplication operation and inverse operation, and we distinguish a particular element; and, if H is a normal subgroup of G, then the new set G/H is a group with the defined multiplication, inverse, and element. In that case, G/H is referred to as the quotient of G by H.

In the formal setting, the construction is similar. One aspect of our formal construction which is of interest is the part that corresponds to considering the new set $G/H := \{g * H | g \in G\}$. Informally, this is a new collection of objects constructed from the givens. Correspondingly in the formal case, we define a new collection of objects from the givens. This new collection is a new type; we call it the *quotient type*.

[7]The indices (i.e. the `{l}` on `quot` and `Type`) refer to hierarchy-levels in the hierarchy on the universe of types, and they constrain the position of the quotient type in the hierarchy. We can ignore them here.

[8]This stipulation is recorded in the library as `lift_beta` in quot.lean. That these two expressions (i.e. `g [a]` and `f a`) are reduced to the same term by the Lean kernel is a consequence of the command `init_quotient` in quot.lean. This command instructs the kernel to reduce them to the same term; it is a stipulation.

In the description of the informal construction above, H is a normal subgroup. That is, H is normal and H is a subgroup. In the formal setting, we separate the properties of being normal and being a subgroup. Because of this, there is room for two separate constructions of the quotient type. In one construction of the quotient type, we suppose H is normal, and we do not suppose that it is a subgroup. The second construction uses the first but begins with a different supposition. Using this alternate supposition, we construct a normal set for use in the first. More specifically, in the second construction:

(1) we suppose that H is a subgroup, but we do not suppose it is normal;

(2) we consider the normalizer of H;

(3) we construct a new type using the normalizer of H[9];

(4) there is a function from the original type to the type constructed using the normalizer of H;

(5) under that function, the image of H is a normal in the new type;

(6) so, now we have a type (i.e. the type constructed from normalizer H) and a normal set in that type (i.e. the image of H under the function from base type to new type), and we can proceed as we do in the first construction.

Below, we discuss each construction more slowly. Further, once we have the quotient type, we define a multiplication, an inverse, and a one on this type. Then we show that these operations and one form a group on the quotient type. The quotient type with this group is the formal version of the *quotient group*, the quotient of the group by H.

4.8.3 Quotient type and quotient group: the first construction

Because the second construction of the quotient type (i) requires a tool we have not yet discussed (i.e. `subtype`) and (ii) uses the first construction, we discuss it after the first construction. Moreover, after we discuss the first construction of the quotient type, we construct the group on this type, prove lemmas about these objects, and discuss the first isomorphism theorem.

First construction of quotient type

Suppose that `{A : Type} [group A] (H : set A) [is_normal G]`. Given the first three suppositions[10], we can prove that we have an equivalence relation; in particular, `equivalence (lcoset_equiv H)` (see section on relations). So, we have a type (i.e. `A`) and an equivalence relation on this type (i.e. `lcoset_equiv H`). Thus, we have the data for making a `setoid A`. With a `setoid A`, we can construct the desired quotient type using `quot`.

```
section
  variables {A : Type} [group A] (H : set) [is_normal H]
```

[9]We construct a new type using this set using `subtype` which we discuss shortly.

[10]To construct the quotient type, we do not need the assumption that `H` is normal. However, to define the desired multiplication and inverse on the quotient type, we need this assumption.

```
  definition lcoset_setoid [instance] : setoid A :=
    setoid.mk (lcoset_equiv H) (equivalence_lcoset_equiv H)

  definition quotient := quot (lcoset_setoid H)
end
```

Since `quotient` is defined in a section with an explicit variable of type `set`, `quotient` takes an argument of type `set`. By inspecting the type of `quot`, we see that `(quotient H : Type)`. `quotient H` is the quotient type.

Constructing the group on the quotient type

In order to construct a group on the quotient type, we need to

define an element which will serve as a one

define an operation which will serve as an inverse

define an operation which will serve as a multiplication, and

prove that this element and these operations satisfy the fields of the `group` structure.

We proceed through these steps.

Defining the one

Recall that `quot.mk` is a function from the base type to the type formed by `quot`. We define the notation `[x]` for the application of `quot.mk` to an `x`. With this, we define the element which will serve as a one as `[1]`, where this is the `1` from the base type.[11]

Defining the inverse and multiplication

Recall that `quot.lift` and `quot.lift`$_2$ are terms used to construct one-place or two-place operations on the quotient type. For each of these, the term takes as argument an operation on the base type and a proof that the operation respects the equivalence relation. For example, consider the case of a one-place operation. We restate `quot.lift`:

```
constant quot.lift : Π {A B : Type} [setoid A] (f : A → B),
                       (∀ a b, a ≈ b → f a = f b) → quot s → B
```

So, with `{A B : Type} [s: setoid A]` available, `quot.lift` takes as arguments some `(f : A → B)` and a proof that `∀ a b, a ≈ b → f a = f b`. The result is a function of type `quot s → B`.

In our case, we have `{A : Type} [lcoset_setoid H : setoid A]` and ≈ is `lcoset_equiv H`. So, to define an inverse operation on the quotient type, we apply `quot.lift` to

[11]One of the assumptions is `[group A]`; so, such a `1` is available.

```
λ a, [a⁻¹]
```
[12]

and a proof that `∀ a b, lcoset_equiv H a b → [a⁻¹] = [b⁻¹]`. And, to define a multiplication on the quotient type, we apply `quot.lift₂` to `(λ a b, [a * b])` and a proof that

```
∀ a₁ a₂ b₁ b₂, lcoset_equiv H a₁ b₁ → lcoset_equiv H a₂ b₂ →
  [a₁ * a₂] = [b₁ * b₂]
```

This is accomplished as follows.

```
section
  variables {A : Type} [group A] (H : set A) [is_normal H]

  definition qone : quotient H := [1]

  definition qinv : quotient H → quotient H :=
    quot.lift
      (λ a, [a⁻¹])
      (λ a₁ a₂ e, quot.sound (lcoset_equiv_inv H e))

  definition qmul : quotient H → quotient H → quotient H :=
    quot.lift₂
      (λ a b, [a * b])
      (λ a₁ a₂ b₁ b₂ e₁ e₂, quot.sound (lcoset_equiv_mul H e₁ e₂))
end
```

Note that, since the quotient type depends explicitly on a set (here, `H`), we refer to the above by `qone H`, `qinv H`, `qmul H`. Thereby, the distinguished element and the operations for the quotient group are parameterized by the set by which the group is quotiented. Thus, for our quotient type (`quotient H`), we have a distinguished element (`qone H`), a one-place operation (`qinv H`), and a two-place operation (`qmul H`).

Proving that the one, inverse, and multiplication satisfy the fields of the group structure

Recall the definition of the `group` structure. In total, it depends on `(A : Type)`, and it has the fields `mul`, `inv`, `one`, `mul_assoc`, `mul_one`, `one_mul`, and `mul.left_inv`. Thus, in order to demonstrate that we have a `group` on the quotient type (i.e. `quotient H`), we need to construct a structure with each of these fields. So, far we have `qmul` for `mul`, `qinv` for `inv`, and `qone` for `one`. It remains to demonstrate that these operations and element behave as required by `mul_assoc`, `mul_one`, `one_mul`, and `mul.left_inv`.

We prove each of these using two consequences (`quot.induction_on` and `quot.induction_on₂`) of one of the quotient constants above (`quot.ind`).[13]
`quot.induction_on` has the form

[12] the $^{-1}$ is notation for the inverse operation from the `[group A]`

[13] To see the statement and derivation of these consequences, see quot.lean. Each is an immediate consequence of `quot.ind`.

```
definition quot.induction_on {A : Type} [s : setoid A]
  {B : quot s → Prop} (q  : quot s) (H : ∀ a, B [a]) : B q :=
  quot.ind H q
```

That is, given a type, a setoid on the type, and a predicate on the type

`{A : Type} [s : setoid A] {B : quot s→ Prop}`

to show that the predicate holds of an element in the quotient type (i.e. to show `B q` for a `(q : quot s)`), it suffices to show that the predicate holds for all equivalence classes (i.e. it suffices to show `∀ a, B [a]`).

For example, consider the claim that the operations and the one satisfy `mul_one`. That is, consider the claim that `∀ q, qmul H q (qone H) = q`. This claim is about all elements in the quotient type. Using `quot.induction_on`, we prove the claim by proving that the claim holds for all equivalence classes (i.e. `∀ a, qmul H [a] [one] = [a]`).[14] By the definition of `qmul`, `qmul H [a] [1] = [a * 1]`. And, by the `mul_one` from the group on the base type, `a * 1 = a`. With this sketch, the reader can see how the proof proceeds. The other proofs are similar.

```
section
  variables {A : Type} [group A] (H : set A)

  proposition qmul_one (a : quotient H) : qmul H a (qone H) = a :=
    quot.induction_on a (λ a', show [a' * 1] = [a'], by rewrite mul_one)

  proposition qone_qmul (a : quotient H) : qmul H (qone H) a = a :=
    quot.induction_on a (λ a', show [1 * a'] = [a'], by rewrite one_mul)

  proposition qmul_left_inv (a : quotient H) :
    qmul H (qinv H a) a = qone H :=
      quot.induction_on a
        (λ a', show [a'⁻¹ * a'] = [1], by rewrite mul.left_inv)

  proposition qmul_assoc (a b c : quotient H) :
    qmul H (qmul H a b) c = qmul H a (qmul H b c) :=
      quot.induction_on₂ a b
        (λ a b, quot.indution_on c
          (λ c,
           have H : [(a * b) * c] = [a * (b * c)], by rewrite mul.assoc,
           H))
end
```

[14] In this expression, the `a` is implicitly typed. `qmul H` takes as argument elements of the quotient type; and `[a]` takes `a` to the quotient type. From these constraints, Lean infers that `a` is an element of the base type.

Defining the quotient group

We establish that there is a group on the quotient type by filling in each of the fields of the `group` structure.[15]

```
definition group [instance] : group (quotient H) :=
{ group,
  mul           := qmul H,
  inv           := qinv H,
  one           := qone H,
  mul_assoc     := qmul_assoc H,
  mul_one       := qmul_qone H,
  one_mul       := qone_qmul H,
  mul_left_inv  := qmul_left_inv H
}
```

Thus, under the assumptions

`{A : Type} [group A] (H : set A) [is_normal H]`

we have constructed a quotient type (i.e. `quotient H`); and on the quotient type we have constructed a quotient group (i.e. `quotient_group.group H : group (quotient H)`).

Characterizing the quotient group: general lemmas

In the general setting of

`{A : Type} [group A] (H : set A) [is_normal H]`

we prove lemmas about the quotient type, quotient group, and its features. With these general lemams in place, in any setting where we have (i) a type, (ii) a group on the type, (iii) a set, and (iv) a proof that the set is normal, we can apply the lemmas in that setting. In particular, for the first isomorphism theorem, we have — among other hypotheses — (i) a type, (ii) a group on the type, (iii) a set (i.e. the kernel of a homomorphism), and (we prove that) (iv) the kernel of a homomorphism is normal. So, given those hypotheses, we will be able to apply these general lemmas to that case.

In the next section, we prove general lemmas about creating functions on the quotient type using functions on the base type. Again, with the hypotheses of the first isomorphism theorem, we have the data for applying these lemmas. From these applications and those of the previous paragraph, we see that the constructions of the quotient type and the quotient group, along with the associated lemmas, constitute a reusable interface for creating and reasoning about group quotients.

Preliminaries: notation

Recall that one of the general quotient constants we added is a function (`quot.mk`, with notation `[]`) which takes an element from the base type to the quotient type. In

[15] In this definition, we use the identifier `group` for the quotient group being defined. This appears to clash with our previous use of the identifier `group`. For the above, I transcribe the definition from the library; and the reason `group` is a permissible identifier in this definition is that in the library this definition occurs within a namespace such that the full name of the identifier is `quotient_group.group`, rather than just `group`. So, there is no conflict, despite appearances.

order to distinguish this general function from its use in our particular setting, we give it a new name.

```
definition qproj (a : A) : quotient H := [a]
```

Also, we add notation for an operation and a set. First, the operation: given (i) a type, (ii) a group on the type, (iii) an element in the base type, (iv) a set on the base type, and (v) a proof that the set is normal, we consider the result of applying to that element the base-type-to-quotient-type function (`qproj`) for that set's quotient type. And, the set: instead of applying the function (`qproj`) to an element, we apply it to a set to obtain the image. We denote the operation with `'*` and the set with `/`.

```
infix '*    := λ {A' : Type} [group A'] a H' [is_normal H'], qproj H' a
infix  /    := λ {A' : Type} [group A'] G H' [is_normal H'], qproj H' ' G
```

For example, consider the setting

```
{A : Type} [group A] (H : set A) [is_normal H]
```

Given an element `(a : A)`, the corresponding element in the quotient group (`group_quotient.group H`) is `a '* H`. And, given a set `(K : set A)`, the image of `K` under the base-type-to-quotient-type function (`qproj`) is `K / H`.

Lemmas

First, we prove that the base-type-to-quotient-type function (`qproj`) is a homomorphism. Due to how we defined the multiplication on the quotient group, this is immediate.

```
proposition is_hom_qproj [instance] : is_hom (qproj H) :=
  is_mul_hom.mk (λ a b, rfl)
```

The simplicity of this proof is made possible by type class inference and `structure`s. That `qproj H` is well-defined depends on Lean finding a proof of `is_normal H` — a procedure we have flagged for type class inference. That there is a `group` on the domain type (`A`) and the target type (`group_quotient.group H`) is determined by type class inference. Then, since what is required to construct a multiplicative homomorphism is a `has_mul` on the domain type and target type, Lean traces the class inheritances to find within each `group` the appropriate `has_mul` and the behavior of its multiplication. Lastly, and unrelated to type class inference and `structure`s, Lean's reduction engine allows the proof of the equality

```
qproj H (a * b) = qmul (qproj H a) (qproj H b)
```

to be generated by `rfl`. The kernel automatically reduces the terms using our definitions and checks that the terms reduce to the same term, establishing the equality.

Next, we show that the base-type-to-quotient-type function is surjective. This depends only on the properties of the quotient constants.

```
proposition surjective_qproj : surjective (qproj H) :=
  take y, quot.induction_on y (λ a, exists.intro a rfl)
```

Also, using only the quotient constants and their consequences[16], we prove the lemmas that two elements in the base type are related iff their images under the base-type-to-quotient-type function are equal.

[16] For a defintion of `quot.exact`, see quot.lean.

```
proposition qproj_eq_qproj {a b : A} (h : a * H = b * H):
  a '* H = b '* H :=
    quot.sound h

proposition lcoset_eq_lcoset_of_qproj_eq_qproj {a b : A}
  (h : a '* H = b '* H) : a * H = b * H :=
    quot.exact h
```

Using these facts and a new hypothesis that `is_subgroup H`, we prove:

- the kernel of the base-type-to-quotient-type function is the set used to define the quotient (i.e. `proposition ker_proj : ker (qproj H) = H`)

- the image of an element in the base type under `qproj` is the `one` in the quotient type iff the element is in the set used to define the quotient (i.e. `proposition qproj_eq_one_iff : a '* H = 1 ↔ a ∈ H`)

- the image under `qproj` of the set used to define the quotient is the singleton containing the `one` in the quotient type (i.e. `proposition image_qproj_self : H / H = '{1}`, where `'{x}` denotes the set containing only `x`)

Characterizing the quotient group:
extending functions from base type to quotient type

The results in the previous section are restricted to relations (i) between elements on the base type and quotient type and (ii) between sets on the base type and quotient type. In this section, we extend the results to relations between functions on the base type and functions on the quotient type. In particular, we exhibit how to use a function on the base type to construct a function on the quotient type. We have done this before in defining `qinv` and `qmul` via `quot.lift` and `quot.lift₂`. However, in those cases, we applied `quot.lift` or `quot.lift₂` to a function for which both the domain type and return type were the base type. In this section, we consider functions for which the return type is not the base type.

To obtain the results, we introduce the new hypotheses

```
{B : Type} {f : A → B} (respf : ∀ a₁ a₂, a₁ * H = a₂ * H → f a₁ = f a₂)
```

By these, we introduce a new type `B`, a function from `A` to `B`, and a stipulation that the function respects the `lcoset_equiv H` relation. In the previous section, we noted that we first prove results with general hypotheses and later apply the results to situations in which we can satisfy the hypotheses, e.g. when given the hypotheses of the first isomorphism theorem. Here too, we prove results with general hypotheses and later apply them. Specifically, the hypotheses of the first isomorphism theorem give us a `B` and an `f`, and we prove that `f` satisfies the equivalence relation `lcoset_equiv (ker f)`.

Under these new general hypotheses, we extend this function `f` on the base type to a function on the quotient type similarly to how we extended `mul` and `inv`: we use `quot.lift`, passing (a) the proposed function and (b) a proof that the function respects the equivalence relation.

```
definition extend : quotient H → B := quot.lift f respf
```

`extend` depends on `respf`, so in future uses `(extend respf : quotient H → B)`.

Recall that the quotient constants are added to the CIC. Consequently, their behavior is stipulated rather than defined. One of the stipulations about quotient constants regards the relationship between a function `f` and a function constructed from this by `quot.lift`. Suppose that we have an equivalence relation ≈ and a proof that `f` respects this relation (i.e. `(c : ∀ a b, a ≈ b → f a = f b)`). The relationship between `f` and its extension is this: for an element `a` in the domain type of `f`, `f a = lift f c [a]`. That is, the output of `f` on `a` is the same as the output of `quot.lift f c` on the equivalence class of `a`.[17] In the setting of this section, this fact gives us the behavior of `extend`.

```
proposition extend_qproj (a : A) : extend respf (a '* H) = f a := rfl

proposition extend_comp_qproj : extend respf ∘ (qproj H) = f := rfl

proposition image_extend (G : set A) : (extend respf) ' (G / H) = f ' G :=
  by rewrite [-image_comp]
```

`extend_qproj` is the stipulated behavior mentioned above: the result of applying `f` to an element `a` of the base type is the same as the result of applying the extended version of `f` to the equivalence class (i.e. the element in the quotient type which corresponds to the element in the base type) of `a`. `extend_comp_qproj` says that applying `f` to an element of the base type is the same as (i) applying to an element of the base type the base-type-to-quotient-type function then (ii) applying the extended version of `f` to the result of (i). `image_extend` is `extend_qproj` as applied to sets instead of to an element: the image of `f` on a set `G` on the base type is the same as the image of the extended version of `f` on the set of equivalence classes of elements in `G`.

Under additional hypotheses, additional behavior of `extend respf` follows immediately from the behavior of `f`. In particular, suppose that there is a group on `B` (i.e. `[group B]`) and `f` is a homomorphism from `A` to `B` (i.e. `[is_hom f]`). Then, `extend respf` is a homomorphism.

```
section
  variable [group B]

  proposition is_hom_extend [instance] [is_hom f] :
    is_hom (extend respf) :=
      is_mul_hom.mk (take a b,
      show (extend respf (a * b)) = (extend respf a) * (extend respf b),
        from quot.induction_on₂ a b (take a b, hom_mul f a b))
end
```

Keeping the assumption that `[group B]` but dropping the assumption that `[is_hom f]`, we prove a lemma about the kernel of `extend respf`. The lemma is that

[17] See quot.lean; in particular, `lift.beta`. And, in the tutorial, see the documentation for the command `init_quotient` on line 28.

two sets are the same: (i) the kernel of `extend respf` and (ii) the image of `ker f` under the base-type-to-quotient-type function. That is,

```
proposition ker_extend : ker (extend respf) = ker f / H := ...
```

That concludes our characterization of the quotient group under general hypotheses. In the next section, we use these results to prove the first isomorphism theorem for this construction of the quotient type and quotient group.

The first isomorphism theorem

Recall the informal statement of the first isomorphism theorem:

> Suppose that G_1 and G_2 are groups. Suppose that f is a homomorphism from G_1 to G_2. Suppose f is onto from G_1 to G_2. Suppose K is the kernel of f. Then, K is a normal subgroup of G_1; there is a homomorphism g from G_1/K onto G_2; and, g is injective.

We state an equivalent version.

> Suppose that G_1 and G_2 are groups. Suppose that f is a homomorphism from G_1 to G_2. Suppose K is the kernel of f. Then, K is a normal subgroup of G_1; there is a homomorphism g from G_1/K onto the image of f; and, g is injective.

Using this equivalent version, we alter the variable names to align with the variable names used in previous examples.

> Suppose that A and B are groups. Suppose that f is a homomorphism from A to B. Suppose K is the kernel of f. Then, K is a normal subgroup of A; there is a homomorphism $\overline{f}$ from A/K onto the image of f; and, $\overline{f}$ is injective.

We display the informal and formal representations side-by-side.

informal	formal
A is a group	`{A : Type}` `[group A]`
B is a group	`{B : Type}` `[group B]`
$f : A \to B$	`f : A → B`
f is a homomorphism	`[is_hom f]`
kernel of f	`ker f`
kernel of f is a normal subgroup of A	`[is_subgroup (ker f)]` `[is_normal (ker f)]`
$A/(\text{kernel } f)$	`quotient (ker f)`

$\overline{f} : A/(\text{kernel } f) \to B$
is a homomorphism
onto the image of f
	`bar f : quotient (ker f) → B`
	`[is_hom (bar f)]`
	`surj_on_bar f : surj_on (bar f) univ (f ' univ)`

$\overline{f}$ is injective	`injective_bar f : injective (bar f)`

There are idenifiers in the table above that we have not yet defined; e.g.

`univ, bar f, surj_on_bar , injective_bar`

`univ` is the name of the set on a type which set contains all the elements on the type.[18] Constructing the terms assigned to the latter three identifiers is the work of proving the first isomorphism theorem for this construction of the quotient type.

The first observation about the theorem quoted above is that it discusses the quotient A/K; so, in the formal setting, we need to construct the quotient type `quotient (ker f)`. A second observation is that there is a homomorphism, the domain of which is the quotient. For a homomorphism to be defined, there must be a binary operation on the domain; so, in the formal setting, we need to construct such a binary operation on the quotient type. As foreshadowed, both the desired quotient type and the binary operation on that type are consequences of applying the general quotient type construction and quotient group construction to the data of the theorem's hypotheses. In short, we get the desired type and the operation from the general lemmas of the previous section.

Constructing the function on the quotient type

In order to extend the function `f` from the base type to the quotient type, we prove that `f` respects the `lcoset_equiv (ker f)` equivalence relation.[19] After this, we apply a general lemma from the previous section in order to extend `f` to the quotient type.

```
definition bar : quotient (ker f) → B :=
  extend (eq_of_lcoset_equiv_ker f)
```

In the expression `quotient (ker f)`, there is an implicit requirement that `ker f` is a normal subgroup. This is represented by an implicit argument that is filled in by Lean automatically.

Let's consider how this is done. Before we write the above definition in the library,

(i) we have proven `[is_normal (ker f)]`[20]

(ii) in the general quotient type construction, we have marked for type class inference the proof that the relevant set is normal (i.e. `(H : set A) [is_normal H]` — note the hard brackets indicating that this hypothesis should be found by type class inference)

(iii) in the general quotient type construction, we have marked as an `instance` the `setoid`[21]

In the definition of `bar`, we apply `extend`. In the definition of `extend`, we apply `quot.lift`. `quot.lift` depends on — among other things — the presence of a `setoid` on the base type. In the definition of `quot.lift`, this setoid is marked for type class inference. Thus, given the entire scenario just described, when `extend` uses `quot.lift`, Lean searches

[18] `definition univ {X : Type} : set X := λ x, true`

[19] `proposition eq_of_lcoset_equiv_ker`

[20] And Lean has access to this proof because, in the file where `bar` is defined, we have imported the file in which the proof appears

[21] Recall that, when something is marked as an `instance`, it can be found via type class inference.

for a `setoid`; it can find this setoid by finding the proof that `[is_normal (ker f)]` and constructing the setoid. Given the setoid, the quotient type can be constructed. Further — though we don't use it in the defnition of `bar` — there is a group on the quotient type since the `group` defined on the quotient type is marked as an `instance`.

Proving the properties of the function

In the section about characterizing the quotient group, we proved that `extend` is a homomorphism (i.e. see `is_hom_extend`). Since `bar f` is `extend` on the data from the hypotheses in this section, that `bar f` is a homomorphism follows from the result in the previous section.

```
proposition is_hom_bar [instance] : is_hom (bar f) := is_hom_extend _
```

The `_` indicates to Lean to find the required argument (here, `[is_hom f]`) automatically.

Also in the section characterizing the quotient group, we exhibited the stipulations relating the behavior of the function on the base type to the extended function on the quotient type. Here, we use these stipulations and the results we proved in the last section to prove that the extended function `bar f` is surjective on the image of `f`.[22]

Lastly, to prove that `bar` is injective, we again use the general results from the previous section. Before this, we prove that: if the kernel of a homomorphism is the singleton containing the identity element, the homomorphism is injective.[23] Then, we use the previous results to show that the kernel of the homomorphism (i.e. `ker (bar f)`) is the singleton containing the identity element; that is, we use the previous results `ker_extend` and `image_qproj_self` to show that `(ker (bar f))` is `'{1}`.

```
proposition ker_bar_eq : ker (bar f) = '{1} :=
  by rewrite [↑bar, ker_extend, image_qproj_self]
```

So, combining the above, we get the result that `injective (bar f)`.

```
proposition injective_bar : injective (bar f) :=
  injective_of_ker_eq_singleton_one (ker_bar_eq f)
```

This concludes the proof of the first isomorphism theorem for this construction of the quotient type. In the next section, we proceed with the second construction of the quotient type.

4.8.4 Quotient type and quotient group: the second construction

In the first construction of the quotient type, the hypotheses were

```
{A : Type} [group A] (H : set A) [is_normal H]
```

We can construct the quotient type in a second way using different hypotheses. In particular, suppose

[22] See `surj_on_bar`.

[23] We call this `injective_of_ker_eq_singleton_one`. It is in group_theory.basic.lean.

```
{A : Type} [group A] (H : set A) [is_subgroup H]
```

Note that any set H is normal in the normalizer of H. Suppose that:

(i) we can make a type from `normalizer H`

(ii) we can construct a group on this new type

(iii) in the new type, we can construct a normal set using `H`

Then, we have the hypotheses for the first construction of the quotient type: a type, a group on the type, and a normal set on the type. So, using the first construction on this data, we get a quotient type and quotient group. Below, we proceed in this way, thereby giving a means to take a quotient by an arbitrary subgroup.

Supposition (i): creating a type from a set

The first of the three suppositions above is that we can construct a new type from `normalizer H`. We can do this. In fact, we can construct a new type from any predicate on a type. In this section, we describe a tool (i.e. `subtype`) and use the tool to construct the desired type.

Description of `subtype`

`subtype` is an inductive type.[24]

```
structure subtype {A : Type} (P : A → Prop) :=
  tag :: (elt_of : A) (has_property : P elt_of)
```

Given a type `(X : Type)` and a predicate on the type `(S : X → Prop)`, `subtype S` is a new type. Terms of the new type have two components: the first component is a term of the base type, and second component is a proof that the first component satisfies the predicate. To construct a term of `subtype S`, we need a term of type `X` and a proof that the term satifies the predicate. Given a term of type of `subtype S`, we can extract a term of type `X` and a proof that the element satisfies `S`.

Using `subtype` *to construct a type from* `normalizer H`

Recall that we have defined `set`s as predicates. So, since `normalizer H` is a `set`, we can use `subtype` to construct a new type: `subtype (normalizer H) : Type`. Terms on this new type are bundled objects — a bundle of (i) term from base type and (ii) proof that

[24]In previous `structure` definitions, after the `:=` is a sequence of type decalarations; for example, in the definition of `has_mul`, after `:=` is `(mul : A→ A→ A)`. And, for other `structure`s, to make a term of the inductive type, we write `<structure_name>.mk` and provide arguments for the required components. For example, suppose that we have `(X : Type)` and `(op : X→ X→ X)`; then, `(has_mul.mk op : has_mul X)`.

However, the `structure` definition for `subtype` differs in one way from these previous definitions. In it, before the type declarations appears `tag ::`. The effect of `tag ::` is simply to rename the default `subtype.mk` to `subtype.tag`. Hence, with

```
(X : Type) (S : X→ Prop) (w : X) (y : S w)
```

we have `(subtype.tag w y : subtype S)`

the element is a member of `normalizer H`. We can use `subtype.tag` and `subtype.elt_of` as functions to move back and forth between the base type and the subtype.

Supposition (ii): constructing a group on the new type

Using `subtype`, we construct a new type. Next, we wish to construct a group on this new type. Since

1. elements on the new type are a bundle including a term from the base type;
2. we have a multiplication, inverse, and one for terms on the base type from the assumption that there is a group on the base type; and,
3. we can move between elements on the base type and elements on the subtype using `subtype.tag` and `subtype.elt_of`,

it is straightforward to define a multiplication, inverse, and one for the subtype in terms of those on the base type. In short, we project from the subtype to the base type, apply the desired operation, and then project back. Further, the proofs that these operations and distinguished term have the desired properties are immediate consequences of the properties of the `group` on the base type.[25] The fact that `is_subgroup H` is required in these steps.

Supposition (iii): creating a normal set on the new type

The final hypothesis of the first construction is that there is a normal set on the type. In this second construction, we construct such a set in two steps.

1. use `subtype.tag` to define a function (i.e. `to_group_of (normalizer H)`) from base type to subtype
2. consider the image of `H` under this function

The image of `H` under this function (i.e. `to_group_of (normalizer H) 'H`) is normal in `subtype (normalizer H)`.

Thus, we have a type (i.e. `subtype (normalizer H))`), a group on the type (see immediately previous section), and a normal set on the type (i.e. `to_group_of (normalizer H) 'H`). So, we have the requisite hypotheses for using the first construction of the quotient type. Using that construction, we construct a quotient type (i.e. `quotient (to_group_of (normalizer H) 'H)`).

Note that now we have three types: the base type, the subtype, and the quotient type. We construct functions from the base type to the quotient type by composing (a) functions from base type to subtype with (b) functions from subtype to quotient type. The development of the first isomorphism theorem for this construction is parallel to that for the other construction, with the following divergences:

> we state and prove a general lemma about homomorphisms and extending functions from one type to another (see results for `gen_extend`), and we obtain from this lemma the function for the isomorphism theorem.

[25] Both the definitions and proofs are in subgroup_to_group.lean

the results are 'local' in the sense that the results hold for sets on the type rather than the whole type; for example, rather than proving that the homomorphism is injective on the type, we prove that it is injective on a certain set.

This concludes the description of the second construction of the quotient type.

4.9 Other formalizations

There are other formalizations of group theory using proof assistants. These include one in Isabelle and one in the SSReflect extension of Coq.[12, 13, 14, 3] We point to a few of the similarities and disimilarities.

The formal language Isabelle encodes is simple type theory. As a result of this, the features of the formal language differ from those in Lean, and mechanisms are introduced to Isabelle to provide features native to dependent type theory and consequently Lean. For example, in dependent type theory variables can range over structures, and much information in expressions can be left implicit and inferred from the context. In Isabelle, the mechanism of *locales* is introduced to assist with these tasks. Locales share features with `section`s in Lean. For, locales provide 'arbitrary but fixed' objects for use in definitions, statements, theorems, and proofs; inside the locales, information can be left implicit because the relevant objects can be inferred from the stated locale objects; and, outside the locales, the definitions, statements, theorems, and proofs that use the objects are parameterized by inputs of the relevant types – i.e. the locale objects provide the behavior of variables ranging over structures.

The formal language Coq encodes is the calculus of inductive constructions. So, it is possible for our formalization to be very similar to the formalization in SSReflect. In fact, we use ideas from that formalization.[26] The sizes of the two projects differ: the goal of our formalization is the group isomorphism theorems; the goal of the SSReflect formalization is the Feit-Thompson theorem. Also, aspects of the approaches of the two projects differ: the SSReflect formalization is concerned with being constructive and computational throughout, and consequently it restricts to finite sets. In our formalization, we do not restrict to finite sets; and, for example, in cases where we want a definition to depend on whether an element is a member of some set, we use classical reasoning. Our results hold for arbitrary sets and so can be applied to finite sets.[27] A second difference concerns the tools used to handle implicit information in expressions. In order to suppress information in expressions, define natural notation, and mimic informal mathematics we use type classes and type class inference. In order to accomplish the same ends, the SSReflect formalization uses a similar mechanism called *canonical structures*.

[26] In particular, the second construction of the quotient type follows a construction in the SSReflect formalization. As there, for an arbitrary subgroup, we consider the normalizer of the subgroup, a type constructed from the normalizer, and quotient groups on this type. Further, for the second construction of the quotient type, we prove the same general morphism property as they do in order to obtain the isomorphism for the first isomorphism theorem.

[27] In the standard library of Lean, there is a predicate which asserts that a set is finite. The results can be applied to sets for which this prediate holds. Also, there is a formalization in Lean based on a rendering of finite sets called finsets. For that, see the directories data.finset and finite_group_theory.

Our formalization provides a library of group theoretic results to Lean. It exhibits the tools available in that proof assistant, and it serves as an assessment of those tools – it can be used to assess the degree to which the system permits natural, convenient representations of mathematical objects and the degree to which it provides support for reasoning about these objects. In particular, we feel that the formalization shows how the language of the calculus of inductive constructions along with the tools of sections, structures, type class inference, calculation environments, rewriting, and defined notation permit natural, convenient representations and provide support for machine-verified reasoning.

Appendices

The appendices contain the files comprising the bulk of the formalization. At the date of writing, these files can be accessed here:

https://github.com/leanprover/lean/blob/master/library/theories/group_theory/basic.lean
https://github.com/leanprover/lean/blob/master/library/theories/group_theory/subgroup_to_group.lean
https://github.com/leanprover/lean/blob/master/library/theories/group_theory/quotient.lean

```
/-
Copyright (c) 2016 Andrew Zipperer. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Andrew Zipperer, Jeremy Avigad

Basic group theory: subgroups, homomorphisms on a set, homomorphic images, cosets,
  normal cosets and the normalizer, the kernel of a homomorphism, the centralizer, etc.

For notation a * S and S * a for cosets, open the namespace "coset_notation".
For notation a^b and S^a, open the namespace "conj_notation".

TODO: homomorphisms on sets should be refactored and moved to algebra.
-/
import data.set algebra.homomorphism theories.move
open eq.ops set function

namespace group_theory

variables {A B C : Type}

/- subgroups -/

structure is_one_closed [class] [has_one A] (S : set A) : Prop :=
(one_mem : one ∈ S)

proposition one_mem [has_one A] {S : set A} [is_one_closed S] : 1 ∈ S :=
is_one_closed.one_mem _ S

structure is_mul_closed [class] [has_mul A] (S : set A) : Prop :=
(mul_mem : ∀₀ a ∈ S, ∀₀ b ∈ S, a * b ∈ S)

proposition mul_mem [has_mul A] {S : set A} [is_mul_closed S] {a b : A} (aS : a ∈ S) (bS : b ∈ S) :
  a * b ∈ S :=
is_mul_closed.mul_mem _ S aS bS

structure is_inv_closed [class] [has_inv A] (S : set A) : Prop :=
(inv_mem : ∀₀ a ∈ S, a⁻¹ ∈ S)

proposition inv_mem [has_inv A] {S : set A} [is_inv_closed S] {a : A} (aS : a ∈ S) : a⁻¹ ∈ S :=
is_inv_closed.inv_mem _ S aS

structure is_subgroup [class] [group A] (S : set A)
  extends is_one_closed S, is_mul_closed S, is_inv_closed S : Prop

section groupA
  variable [group A]
```

```
  proposition mem_of_inv_mem {a : A} {S : set A} [is_subgroup S] (H : a⁻¹ ∈ S) : a ∈ S :=
  have (a⁻¹)⁻¹ ∈ S, from inv_mem H,
  by rewrite inv_inv at this; apply this

  proposition inv_mem_iff (a : A) (S : set A) [is_subgroup S] : a⁻¹ ∈ S ↔ a ∈ S :=
  iff.intro mem_of_inv_mem inv_mem

  proposition is_subgroup_univ [instance] : is_subgroup (@univ A) :=
  ⦃ is_subgroup,
    one_mem := trivial,
    mul_mem := λ a au b bu, trivial,
    inv_mem := λ a au, trivial ⦄

  proposition is_subgroup_inter [instance] (G H : set A) [is_subgroup G] [is_subgroup H] :
    is_subgroup (G ∩ H) :=
  ⦃ is_subgroup,
    one_mem := and.intro one_mem one_mem,
    mul_mem := λ a ai b bi, and.intro (mul_mem (and.left ai) (and.left bi))
                  (mul_mem (and.right ai) (and.right bi)),
    inv_mem := λ a ai, and.intro (inv_mem (and.left ai)) (inv_mem (and.right ai)) ⦄
end groupA


/- homomorphisms on sets -/

section has_mulABC
  variables [has_mul A] [has_mul B] [has_mul C]

  -- in group theory, we can use is_hom for is_mul_hom
  abbreviation is_hom := @is_mul_hom

  definition is_hom_on [class] (f : A → B) (S : set A) : Prop :=
    ∀₀ a₁ ∈ S, ∀₀ a₂ ∈ S, f (a₁ * a₂) = f a₁ * f a₂

  proposition hom_on_mul (f : A → B) {S : set A} [H : is_hom_on f S] {a₁ a₂ : A}
      (a₁S : a₁ ∈ S) (a₂S : a₂ ∈ S) : f (a₁ * a₂) = (f a₁) * (f a₂) :=
    H a₁S a₂S

  proposition is_hom_on_of_is_hom (f : A → B) (S : set A) [H : is_hom f] : is_hom_on f S :=
  forallb_of_forall₂ S S (hom_mul f)

  proposition is_hom_of_is_hom_on_univ (f : A → B) [H : is_hom_on f univ] : is_hom f :=
  is_mul_hom.mk (forall_of_forallb_univ₂ H)

  proposition is_hom_on_univ_iff (f : A → B) : is_hom_on f univ ↔ is_hom f :=
```

```
  iff.intro (λH, is_hom_of_is_hom_on_univ f) (λ H, is_hom_on_of_is_hom f univ)

  proposition is_hom_on_of_subset (f : A → B) {S T : set A} (ssubt : S ⊆ T) [H : is_hom_on f T] :
    is_hom_on f S :=
  forallb_of_subset₂ ssubt ssubt H

  proposition is_hom_on_id (S : set A) : is_hom_on id S :=
  have H : is_hom (@id A), from is_mul_hom_id,
  is_hom_on_of_is_hom id S

  proposition is_hom_on_comp {S : set A} {T : set B} {g : B → C} {f : A → B}
    (H₁ : is_hom_on f S) (H₂ : is_hom_on g T) (H₃ : maps_to f S T) : is_hom_on (g ∘ f) S :=
  take a₁, assume a₁S, take a₂, assume a₂S,
  have f a₁ ∈ T, from H₃ a₁S,
  have f a₂ ∈ T, from H₃ a₂S,
  show g (f (a₁ * a₂)) = g (f a₁) * g (f a₂), by rewrite [H₁ a₁S a₂S, H₂ ‘f a₁ ∈ T‘ ‘f a₂ ∈ T‘]
end has_mulABC

section groupAB
  variables [group A] [group B]

  proposition hom_on_one (f : A → B) (G : set A) [is_subgroup G] [H : is_hom_on f G] : f 1 = 1 :=
  have f 1 * f 1 = f 1 * 1, by rewrite [-H one_mem one_mem, *mul_one],
  eq_of_mul_eq_mul_left' this

  proposition hom_on_inv (f : A → B) {G : set A} [is_subgroup G] [H : is_hom_on f G]
      {a : A} (aG : a ∈ G) :
    f a⁻¹ = (f a)⁻¹ :=
  have f a⁻¹ * f a = 1, by rewrite [-H (inv_mem aG) aG, mul.left_inv, hom_on_one f G],
  eq_inv_of_mul_eq_one this

  proposition is_subgroup_image [instance] (f : A → B) (G : set A)
    [is_subgroup G] [is_hom_on f G] :
  is_subgroup (f ' G) :=
  ⦃ is_subgroup,
    one_mem := mem_image one_mem (hom_on_one f G),
    mul_mem := λ a afG b bfG,
      obtain c (cG : c ∈ G)(Hc : f c = a), from afG,
      obtain d (dG : d ∈ G)(Hd : f d = b), from bfG,
      show a * b ∈ f ' G, from mem_image (mul_mem cG dG) (by rewrite [hom_on_mul f cG dG, Hc, Hd]),
    inv_mem := λ a afG,
      obtain c (cG : c ∈ G)(Hc : f c = a), from afG,
      show a⁻¹ ∈ f ' G, from mem_image (inv_mem cG) (by rewrite [hom_on_inv f cG, Hc]) ⦄
end groupAB
```

```
/- cosets -/

definition lcoset [has_mul A] (a : A) (N : set A) : set A := (mul a) 'N
definition rcoset [has_mul A] (N : set A) (a : A) : set A := (mul^~ a) 'N

-- overload multiplication
namespace coset_notation
  infix * := lcoset
  infix * := rcoset
end coset_notation

open coset_notation

section has_mulA
  variable [has_mul A]

  proposition mul_mem_lcoset {S : set A} {x : A} (a : A) (xS : x ∈ S) : a * x ∈ a * S :=
  mem_image_of_mem (mul a) xS

  proposition mul_mem_rcoset [has_mul A] {S : set A} {x : A} (xS : x ∈ S) (a : A) :
    x * a ∈ S * a :=
  mem_image_of_mem (mul^~ a) xS

  definition lcoset_equiv (S : set A) (a b : A) : Prop := a * S = b * S

  proposition equivalence_lcoset_equiv (S : set A) : equivalence (lcoset_equiv S) :=
  mk_equivalence (lcoset_equiv S) (λ a, rfl) (λ a b, !eq.symm) (λ a b c, !eq.trans)

  proposition lcoset_subset_lcoset {S T : set A} (a : A) (H : S ⊆ T) : a * S ⊆ a * T :=
  image_subset _ H

  proposition rcoset_subset_rcoset {S T : set A} (H : S ⊆ T) (a : A) : S * a ⊆ T * a :=
  image_subset _ H

  proposition image_lcoset_of_is_hom_on {B : Type} [has_mul B] {f : A → B} {S : set A} {a : A}
      {G : set A} (SsubG : S ⊆ G) (aG : a ∈ G) [is_hom_on f G] :
    f ' (a * S) = f a * f ' S :=
  ext (take x, iff.intro
    (assume fas : x ∈ f ' (a * S),
      obtain t [s (sS : s ∈ S) (seq : a * s = t)] (teq : f t = x), from fas,
      have x = f a * f s, by rewrite [-teq, -seq, hom_on_mul f aG (SsubG sS)],
      show x ∈ f a * f ' S, by rewrite this; apply mul_mem_lcoset _ (mem_image_of_mem _ sS))
    (assume fafs : x ∈ f a * f ' S,
      obtain t [s (sS : s ∈ S) (seq : f s = t)] (teq : f a * t = x), from fafs,
      have x = f (a * s), by rewrite [-teq, -seq, hom_on_mul f aG (SsubG sS)],
      show x ∈ f ' (a * S), by rewrite this; exact mem_image_of_mem _ (mul_mem_lcoset _ sS)))
```

```
  proposition image_rcoset_of_is_hom_on {B : Type} [has_mul B] {f : A → B} {S : set A} {a : A}
    {G : set A} (SsubG : S ⊆ G) (aG : a ∈ G) [is_hom_on f G] :
  f ' (S * a) = f ' S * f a :=
  ext (take x, iff.intro
    (assume fas : x ∈ f ' (S * a),
      obtain t [s (sS : s ∈ S) (seq : s * a = t)] (teq : f t = x), from fas,
      have x = f s * f a, by rewrite [-teq, -seq, hom_on_mul f (SsubG sS) aG],
      show x ∈ f ' S * f a, by rewrite this; exact mul_mem_rcoset (mem_image_of_mem _ sS) _)
    (assume fafs : x ∈ f ' S * f a,
      obtain t [s (sS : s ∈ S) (seq : f s = t)] (teq : t * f a = x), from fafs,
      have x = f (s * a), by rewrite [-teq, -seq, hom_on_mul f (SsubG sS) aG],
      show x ∈ f ' (S * a), by rewrite this; exact mem_image_of_mem _ (mul_mem_rcoset sS _)))

  proposition image_lcoset_of_is_hom {B : Type} [has_mul B] (f : A → B) (a : A) (S : set A)
      [is_hom f] :
    f ' (a * S) = f a * f ' S :=
  have is_hom_on f univ, from is_hom_on_of_is_hom f univ,
  image_lcoset_of_is_hom_on (subset_univ S) !mem_univ

  proposition image_rcoset_of_is_hom {B : Type} [has_mul B] (f : A → B) (S : set A) (a : A)
      [is_hom f] :
    f ' (S * a) = f ' S * f a :=
  have is_hom_on f univ, from is_hom_on_of_is_hom f univ,
  image_rcoset_of_is_hom_on (subset_univ S) !mem_univ
end has_mulA

section semigroupA
  variable [semigroup A]

  proposition rcoset_rcoset (S : set A) (a b : A) : S * a * b = S * (a * b) :=
  have H : (mul^~ b) ∘ (mul^~ a) = mul^~ (a * b), from funext (take x, !mul.assoc),
  calc
    S * a * b = ((mul^~ b) ∘ (mul^~ a)) 'S : image_comp
          ... = S * (a * b)                 : by rewrite [↑rcoset, H]

  proposition lcoset_lcoset (S : set A) (a b : A) : a * (b * S)  = (a * b) * S :=
  have H : (mul a) ∘ (mul b) = mul (a * b), from funext (take x, !mul.assoc⁻¹),
  calc
    a * (b * S) = ((mul a) ∘ (mul b)) 'S : image_comp
            ... = (a * b) * S            : by rewrite [↑lcoset, H]

  proposition lcoset_rcoset [semigroup A] (S : set A) (a b : A) : a * S * b = a * (S * b) :=
  have H : (mul^~ b) ∘ (mul a) = (mul a) ∘ (mul^~ b), from funext (take x, !mul.assoc),
  calc
    a * S * b = ((mul^~ b) ∘ (mul a)) 'S : image_comp
```

```
          ... = ((mul a) ∘ (mul^~ b)) 'S : H
          ... = a * (S * b)              : image_comp
end semigroupA

section monoidA
  variable [monoid A]

  proposition one_lcoset (S : set A) : 1 * S = S :=
  ext (take x, iff.intro
    (suppose x ∈ 1 * S,
      obtain s (sS : s ∈ S) (eqx : 1 * s = x), from this,
      show x ∈ S, by rewrite [-eqx, one_mul]; apply sS)
    (suppose x ∈ S,
      have 1 * x ∈ 1 * S, from mem_image_of_mem (mul 1) this,
      show x ∈ 1 * S, by rewrite one_mul at this; apply this))

  proposition rcoset_one (S : set A) : S * 1 = S :=
  ext (take x, iff.intro
    (suppose x ∈ S * 1,
      obtain s (sS : s ∈ S) (eqx : s * 1 = x), from this,
      show x ∈ S, by rewrite [-eqx, mul_one]; apply sS)
    (suppose x ∈ S,
      have x * 1 ∈ S * 1, from mem_image_of_mem (mul^~ 1) this,
      show x ∈ S * 1, by rewrite mul_one at this; apply this))
end monoidA

section groupA
  variable [group A]

  proposition lcoset_inv_lcoset (a : A) (S : set A) : a * (a⁻¹ * S) = S :=
  by rewrite [lcoset_lcoset, mul.right_inv, one_lcoset]

  proposition inv_lcoset_lcoset (a : A) (S : set A) : a⁻¹ * (a * S) = S :=
  by rewrite [lcoset_lcoset, mul.left_inv, one_lcoset]

  proposition rcoset_inv_rcoset (S : set A) (a : A) : (S * a⁻¹) * a = S :=
  by rewrite [rcoset_rcoset, mul.left_inv, rcoset_one]

  proposition rcoset_rcoset_inv (S : set A) (a : A) : (S * a) * a⁻¹ = S :=
  by rewrite [rcoset_rcoset, mul.right_inv, rcoset_one]

  proposition eq_of_lcoset_eq_lcoset {a : A} {S T : set A} (H : a * S = a * T) : S = T :=
  by rewrite [-inv_lcoset_lcoset a S, -inv_lcoset_lcoset a T, H]

  proposition eq_of_rcoset_eq_rcoset {a : A} {S T : set A} (H : S * a = T * a) : S = T :=
  by rewrite [-rcoset_rcoset_inv S a, -rcoset_rcoset_inv T a, H]
```

```
proposition mem_of_mul_mem_lcoset {a b : A} {S : set A} (abaS : a * b ∈ a * S) : b ∈ S :=
have a⁻¹ * (a * b) ∈ a⁻¹ * (a * S), from mul_mem_lcoset _ abaS,
by rewrite [inv_mul_cancel_left at this, inv_lcoset_lcoset at this]; apply this

proposition mul_mem_lcoset_iff (a b : A) (S : set A) : a * b ∈ a * S ↔ b ∈ S :=
iff.intro !mem_of_mul_mem_lcoset !mul_mem_lcoset

proposition mem_of_mul_mem_rcoset {a b : A} {S : set A} (abSb : a * b ∈ S * b) : a ∈ S :=
have (a * b) * b⁻¹ ∈ (S * b) * b⁻¹, from mul_mem_rcoset abSb _,
by rewrite [mul_inv_cancel_right at this, rcoset_rcoset_inv at this]; apply this

proposition mul_mem_rcoset_iff (a b : A) (S : set A) : a * b ∈ S * b ↔ a ∈ S :=
iff.intro !mem_of_mul_mem_rcoset (λ H, mul_mem_rcoset H _)

proposition inv_mul_mem_of_mem_lcoset {a b : A} {S : set A} (abS : a ∈ b * S) : b⁻¹ * a ∈ S :=
have b⁻¹ * a ∈ b⁻¹ * (b * S), from mul_mem_lcoset b⁻¹ abS,
by rewrite inv_lcoset_lcoset at this; apply this

proposition mem_lcoset_of_inv_mul_mem {a b : A} {S : set A} (H : b⁻¹ * a ∈ S) : a ∈ b * S :=
have b * (b⁻¹ * a) ∈ b * S, from mul_mem_lcoset b H,
by rewrite mul_inv_cancel_left at this; apply this

proposition mem_lcoset_iff (a b : A) (S : set A) : a ∈ b * S ↔ b⁻¹ * a ∈ S :=
iff.intro inv_mul_mem_of_mem_lcoset mem_lcoset_of_inv_mul_mem

proposition mul_inv_mem_of_mem_rcoset {a b : A} {S : set A} (aSb : a ∈ S * b) : a * b⁻¹ ∈ S :=
have a * b⁻¹ ∈ (S * b) * b⁻¹, from mul_mem_rcoset aSb b⁻¹,
by rewrite rcoset_rcoset_inv at this; apply this

proposition mem_rcoset_of_mul_inv_mem {a b : A} {S : set A} (H : a * b⁻¹ ∈ S) : a ∈ S * b :=
have a * b⁻¹ * b ∈ S * b, from mul_mem_rcoset H b,
by rewrite inv_mul_cancel_right at this; apply this

proposition mem_rcoset_iff (a b : A) (S : set A) : a ∈ S * b ↔ a * b⁻¹ ∈ S :=
iff.intro mul_inv_mem_of_mem_rcoset mem_rcoset_of_mul_inv_mem

proposition lcoset_eq_iff_eq_inv_lcoset (a : A) (S T : set A) : (a * S = T) ↔ (S = a⁻¹ * T) :=
iff.intro (assume H, by rewrite [-H, inv_lcoset_lcoset])
          (assume H, by rewrite [H, lcoset_inv_lcoset])

proposition rcoset_eq_iff_eq_rcoset_inv (a : A) (S T : set A) : (S * a = T) ↔ (S = T * a⁻¹) :=
iff.intro (assume H, by rewrite [-H, rcoset_rcoset_inv])
          (assume H, by rewrite [H, rcoset_inv_rcoset])

proposition lcoset_inter (a : A) (S T : set A) [is_subgroup S] [is_subgroup T] :
```

```
    a * (S ∩ T) = (a * S) ∩ (a * T) :=
  eq_of_subset_of_subset
    (image_inter_subset _ S T)
    (take b, suppose b ∈ (a * S) ∩ (a * T),
      obtain [s [smem (seq : a * s = b)]] [t [tmem (teq : a * t = b)]], from this,
      have s = t, from eq_of_mul_eq_mul_left' (eq.trans seq (eq.symm teq)),
      show b ∈ a * (S ∩ T),
        begin
          rewrite -seq,
          apply mul_mem_lcoset,
          apply and.intro smem,
          rewrite this, apply tmem
        end)

  proposition inter_rcoset (a : A) (S T : set A) [is_subgroup S] [is_subgroup T] :
    (S ∩ T) * a = (S * a) ∩ (T * a) :=
  eq_of_subset_of_subset
    (image_inter_subset _ S T)
    (take b, suppose b ∈ (S * a) ∩ (T * a),
      obtain [s [smem (seq : s * a = b)]] [t [tmem (teq : t * a = b)]], from this,
      have s = t, from eq_of_mul_eq_mul_right' (eq.trans seq (eq.symm teq)),
      show b ∈ (S ∩ T) * a,
        begin
          rewrite -seq,
          apply mul_mem_rcoset,
          apply and.intro smem,
          rewrite this, apply tmem
        end)
end groupA

section subgroupG
  variables [group A] {G : set A} [is_subgroup G]

  proposition lcoset_eq_self_of_mem {a : A} (aG : a ∈ G) : a * G = G :=
  ext (take x, iff.intro
    (assume xaG, obtain g [gG xeq], from xaG,
      show x ∈ G, by rewrite -xeq; exact (mul_mem aG gG))
    (assume xG, show x ∈ a * G, from mem_image
      (show a⁻¹ * x ∈ G, from (mul_mem (inv_mem aG) xG)) !mul_inv_cancel_left))

  proposition rcoset_eq_self_of_mem {a : A} (aG : a ∈ G) : G * a = G :=
  ext (take x, iff.intro
    (assume xGa, obtain g [gG xeq], from xGa,
      show x ∈ G, by rewrite -xeq; exact (mul_mem gG aG))
    (assume xG, show x ∈ G * a, from mem_image
      (show x * a⁻¹ ∈ G, from (mul_mem xG (inv_mem aG))) !inv_mul_cancel_right))
```

```
  proposition mem_lcoset_self (a : A) : a ∈ a * G :=
  by rewrite [-mul_one a at {1}]; exact mul_mem_lcoset a one_mem

  proposition mem_rcoset_self (a : A) : a ∈ G * a :=
  by rewrite [-one_mul a at {1}]; exact mul_mem_rcoset one_mem a

  proposition mem_of_lcoset_eq_self {a : A} (H : a * G = G) : a ∈ G :=
  by rewrite [-H]; exact mem_lcoset_self a

  proposition mem_of_rcoset_eq_self {a : A} (H : G * a = G) : a ∈ G :=
  by rewrite [-H]; exact mem_rcoset_self a

  variable (G)

  proposition lcoset_eq_self_iff (a : A) : a * G = G ↔ a ∈ G :=
  iff.intro mem_of_lcoset_eq_self lcoset_eq_self_of_mem

  proposition rcoset_eq_self_iff (a : A) : G * a = G ↔ a ∈ G :=
  iff.intro mem_of_rcoset_eq_self rcoset_eq_self_of_mem

  variable {G}

  proposition lcoset_eq_lcoset {a b : A} (H : b⁻¹ * a ∈ G) : a * G = b * G :=
  have b⁻¹ * (a * G) = b⁻¹ * (b * G),
    by rewrite [inv_lcoset_lcoset, lcoset_lcoset, lcoset_eq_self_of_mem H],
  eq_of_lcoset_eq_lcoset this

  proposition inv_mul_mem_of_lcoset_eq_lcoset {a b : A} (H : a * G = b * G) : b⁻¹ * a ∈ G :=
  mem_of_lcoset_eq_self (by rewrite [-lcoset_lcoset, H, inv_lcoset_lcoset])

  proposition lcoset_eq_lcoset_iff (a b : A) : a * G = b * G ↔ b⁻¹ * a ∈ G :=
  iff.intro inv_mul_mem_of_lcoset_eq_lcoset lcoset_eq_lcoset

  proposition rcoset_eq_rcoset {a b : A} (H : a * b⁻¹ ∈ G) : G * a = G * b :=
  have G * a * b⁻¹ = G * b * b⁻¹,
    by rewrite [rcoset_rcoset_inv, rcoset_rcoset, rcoset_eq_self_of_mem H],
  eq_of_rcoset_eq_rcoset this

  proposition mul_inv_mem_of_rcoset_eq_rcoset {a b : A} (H : G * a = G * b) : a * b⁻¹ ∈ G :=
  mem_of_rcoset_eq_self (by rewrite [-rcoset_rcoset, H, rcoset_rcoset_inv])

  proposition rcoset_eq_rcoset_iff (a b : A) : G * a = G * b ↔ a * b⁻¹ ∈ G :=
  iff.intro mul_inv_mem_of_rcoset_eq_rcoset rcoset_eq_rcoset
end subgroupG
```

```
/- normal cosets and the normalizer -/

section has_mulA
  variable [has_mul A]

  abbreviation normalizes [reducible] (a : A) (S : set A) : Prop := a * S = S * a

  definition is_normal [class] (S : set A) : Prop := ∀ a, normalizes a S

  definition normalizer (S : set A) : set A := { a : A | normalizes a S }

  definition is_normal_in [class] (S T : set A) : Prop := T ⊆ normalizer S

  abbreviation normalizer_in [reducible] (S T : set A) : set A := T ∩ normalizer S

  proposition lcoset_eq_rcoset (a : A) (S : set A) [H : is_normal S] : a * S = S * a := H a

  proposition subset_normalizer (S T : set A) [H : is_normal_in T S] : S ⊆ normalizer T := H

  proposition lcoset_eq_rcoset_of_mem {a : A} (S : set A) {T : set A} [H : is_normal_in S T]
      (amemT : a ∈ T) :
    a * S = S * a := H amemT

  proposition is_normal_in_of_is_normal (S T : set A) [H : is_normal S] : is_normal_in S T :=
  forallb_of_forall T H

  proposition is_normal_of_is_normal_in_univ {S : set A} (H : is_normal_in S univ) :
    is_normal S :=
  forall_of_forallb_univ H

  proposition is_normal_in_univ_iff_is_normal (S : set A) : is_normal_in S univ ↔ is_normal S :=
  forallb_univ_iff_forall _

  proposition is_normal_in_of_subset {S T U : set A} (H : T ⊆ U) (H' : is_normal_in S U) :
    is_normal_in S T :=
  forallb_of_subset H H'

  proposition normalizes_of_mem_normalizer {a : A} {S : set A} (H : a ∈ normalizer S) :
    normalizes a S := H

  proposition mem_normalizer_iff_normalizes (a : A) (S : set A) :
    a ∈ normalizer S ↔ normalizes a S := iff.refl _

  proposition is_normal_in_normalizer [instance] (S : set A) : is_normal_in S (normalizer S) :=
  subset.refl (normalizer S)
```

```
end has_mulA

section groupA
  variable [group A]

  proposition is_normal_in_of_forall_subset {S G : set A} [is_subgroup G]
    (H : ∀₀ x ∈ G, x * S ⊆ S * x) :
  is_normal_in S G :=
  take x, assume xG,
  show x * S = S * x, from eq_of_subset_of_subset (H xG)
    (have x * (x⁻¹ * S) * x ⊆ x * (S * x⁻¹) * x,
      from rcoset_subset_rcoset (lcoset_subset_lcoset x (H (inv_mem xG))) x,
      show S * x ⊆ x * S,
        begin
         rewrite [lcoset_inv_lcoset at this, lcoset_rcoset at this, rcoset_inv_rcoset at this],
         exact this
        end)

  proposition is_normal_of_forall_subset {S : set A} (H : ∀ x, x * S ⊆ S * x) : is_normal S :=
  begin
    rewrite [-is_normal_in_univ_iff_is_normal],
    apply is_normal_in_of_forall_subset,
    intro x xuniv, exact H x
  end

  proposition subset_normalizer_self (G : set A) [is_subgroup G] : G ⊆ normalizer G :=
  take a, assume aG, show a * G = G * a,
    by rewrite [lcoset_eq_self_of_mem aG, rcoset_eq_self_of_mem aG]
end groupA

section normalG
  variables [group A] (G : set A) [is_normal G]

  proposition lcoset_equiv_mul {a₁ a₂ b₁ b₂ : A}
    (H₁ : lcoset_equiv G a₁ a₂) (H₂ : lcoset_equiv G b₁ b₂) : lcoset_equiv G (a₁ * b₁) (a₂ * b₂) :=
  begin
    unfold lcoset_equiv at *,
    rewrite [-lcoset_lcoset, H₂, lcoset_eq_rcoset, -lcoset_rcoset, H₁, lcoset_rcoset,
             -lcoset_eq_rcoset, lcoset_lcoset]
  end

  proposition lcoset_equiv_inv {a₁ a₂ : A} (H : lcoset_equiv G a₁ a₂) : lcoset_equiv G a₁⁻¹ a₂⁻¹ :=
  begin
    unfold lcoset_equiv at *,
    have a₁⁻¹ * G = a₂⁻¹ * (a₂ * G) * a₁⁻¹, by rewrite [inv_lcoset_lcoset, lcoset_eq_rcoset],
    rewrite [this, -H, lcoset_rcoset, lcoset_eq_rcoset, rcoset_rcoset_inv]
```

```
  end
end normalG


/- the normalizer is a subgroup -/

section semigroupA
  variable [semigroup A]

  proposition mul_mem_normalizer {S : set A} {a b : A}
    (Ha : a ∈ normalizer S) (Hb : b ∈ normalizer S) : a * b ∈ normalizer S :=
  show a * b * S = S * (a * b),
    by rewrite [-lcoset_lcoset, normalizes_of_mem_normalizer Hb, -lcoset_rcoset,
                normalizes_of_mem_normalizer Ha, rcoset_rcoset]
end semigroupA

section monoidA
  variable [monoid A]

  proposition one_mem_normalizer (S : set A) : 1 ∈ normalizer S :=
  by rewrite [↑normalizer, mem_set_of_iff, one_lcoset, rcoset_one]
end monoidA

section groupA
  variable [group A]

  proposition inv_mem_normalizer {S : set A} {a : A} (H : a ∈ normalizer S) : a⁻¹ ∈ normalizer S :=
  have a⁻¹ * S = S * a⁻¹,
  begin
    apply iff.mp (rcoset_eq_iff_eq_rcoset_inv _ _ _),
    rewrite [lcoset_rcoset, -normalizes_of_mem_normalizer H, inv_lcoset_lcoset]
  end,
  by rewrite [↑normalizer, mem_set_of_iff, this]

  proposition is_subgroup_normalizer [instance] (S : set A) : is_subgroup (normalizer S) :=
  ⦃ is_subgroup,
    one_mem := one_mem_normalizer S,
    mul_mem := λ a Ha b Hb, mul_mem_normalizer Ha Hb,
    inv_mem := λ a H, inv_mem_normalizer H⦄
end groupA

section subgroupG
  variables [group A] {G : set A} [is_subgroup G]

  proposition normalizes_image_of_is_hom_on [group B] {a : A} (aG : a ∈ G) {S : set A}
      (SsubG : S ⊆ G) (H : normalizes a S) (f : A → B) [is_hom_on f G] :
```

```
    normalizes (f a) (f ' S) :=
  by rewrite [-image_lcoset_of_is_hom_on SsubG aG, -image_rcoset_of_is_hom_on SsubG aG,
             ↑normalizes at H, H]

  proposition is_normal_in_image_image [group B] {S T : set A} (SsubT : S ⊆ T)
      [H : is_normal_in S T] (f : A → B) [is_subgroup T] [is_hom_on f T] :
    is_normal_in (f ' S) (f ' T) :=
  take a, assume afT,
  obtain b [bT (beq : f b = a)], from afT,
  show normalizes a (f ' S),
    begin rewrite -beq, apply (normalizes_image_of_is_hom_on bT SsubT (H bT)) end

  proposition normalizes_image_of_is_hom [group B] {a : A} {S : set A}
      (H : normalizes a S) (f : A → B) [is_hom f] :
    normalizes (f a) (f ' S) :=
  by rewrite [-image_lcoset_of_is_hom f a S, -image_rcoset_of_is_hom f S a,
             ↑normalizes at H, H]

  proposition is_normal_in_image_image_univ [group B] {S : set A}
      [H : is_normal S] (f : A → B) [is_hom f] :
    is_normal_in (f ' S) (f ' univ) :=
  take a, assume afT,
  obtain b [buniv (beq : f b = a)], from afT,
  show normalizes a (f ' S),
    begin rewrite -beq, apply (normalizes_image_of_is_hom (H b) f) end
end subgroupG


/- conjugation -/

definition conj [reducible] [group A] (a b : A) : A := b⁻¹ * a * b

definition set_conj [reducible] [group A] (S : set A)(a : A) : set A := a⁻¹ * S * a
-- conj^~ a ' S

namespace conj_notation
  infix `^` := conj
  infix `^` := set_conj
end conj_notation

open conj_notation

section groupA
  variables [group A]

  proposition set_conj_eq_image_conj (S : set A) (a : A) : S^a = conj^~ a 'S :=
```

```
  eq.symm !image_comp

  proposition set_conj_eq_self_of_normalizes {S : set A} {a : A} (H : normalizes a S) : S^a = S :=
  by rewrite [lcoset_rcoset, ↑normalizes at H, -H, inv_lcoset_lcoset]

  proposition normalizes_of_set_conj_eq_self {S : set A} {a : A} (H : S^a = S) : normalizes a S :=
  by rewrite [-H at {1}, ↑set_conj, lcoset_rcoset, lcoset_inv_lcoset]

  proposition set_conj_eq_self_iff_normalizes (S : set A) (a : A) : S^a = S ↔ normalizes a S :=
  iff.intro normalizes_of_set_conj_eq_self set_conj_eq_self_of_normalizes

  proposition set_conj_eq_self_of_mem_normalizer {S : set A} {a : A} (H : a ∈ normalizer S) :
    S^a = S := set_conj_eq_self_of_normalizes H

  proposition mem_normalizer_of_set_conj_eq_self {S : set A} {a : A} (H : S^a = S) :
    a ∈ normalizer S := normalizes_of_set_conj_eq_self H

  proposition set_conj_eq_self_iff_mem_normalizer (S : set A) (a : A) :
    S^a = S ↔ a ∈ normalizer S :=
  iff.intro mem_normalizer_of_set_conj_eq_self set_conj_eq_self_of_mem_normalizer

  proposition conj_one (a : A) : a ^ (1 : A) = a :=
  by rewrite [↑conj, one_inv, one_mul, mul_one]

  proposition conj_conj (a b c : A) : (a^b)^c = a^(b * c) :=
  by rewrite [↑conj, mul_inv, *mul.assoc]

  proposition conj_inv (a b : A) : (a^b)⁻¹ = (a⁻¹)^b :=
  by rewrite[mul_inv, mul_inv, inv_inv, mul.assoc]

  proposition mul_conj (a b c : A) : (a * b)^c = a^c * b^c :=
  by rewrite[↑conj, *mul.assoc, mul_inv_cancel_left]
end groupA


/- the kernel -/

definition ker [has_one B] (f : A → B) : set A := { x | f x = 1 }

section hasoneB
  variable [has_one B]

  proposition eq_one_of_mem_ker {f : A → B} {a : A} (H : a ∈ ker f) : f a = 1 := H

  proposition mem_ker_iff (f : A → B) (a : A) : a ∈ ker f ↔ f a = 1 := iff.rfl
```

```
  proposition ker_eq_preimage_one (f : A → B) : ker f = f '- '{1} :=
  ext (take x, by rewrite [mem_ker_iff, -mem_preimage_iff, mem_singleton_iff])

  definition ker_in (f : A → B) (S : set A) : set A := ker f ∩ S

  proposition ker_in_univ (f : A → B) : ker_in f univ = ker f :=
  !inter_univ
end hasoneB

section groupAB
  variables [group A] [group B]
  variable  {f : A → B}

  proposition eq_of_mul_inv_mem_ker [is_hom f] {a₁ a₂ : A} (H : a₁ * a₂⁻¹ ∈ ker f) :
    f a₁ = f a₂ :=
  eq_of_mul_inv_eq_one (by rewrite [-hom_inv f, -hom_mul f]; exact H)

  proposition mul_inv_mem_ker_of_eq [is_hom f] {a₁ a₂ : A} (H : f a₁ = f a₂) :
    a₁ * a₂⁻¹ ∈ ker f :=
  show f (a₁ * a₂⁻¹) = 1, by rewrite [hom_mul f, hom_inv f, H, mul.right_inv]

  proposition eq_iff_mul_inv_mem_ker [is_hom f] (a₁ a₂ : A) : f a₁ = f a₂ ↔ a₁ * a₂⁻¹ ∈ ker f :=
  iff.intro mul_inv_mem_ker_of_eq eq_of_mul_inv_mem_ker

  proposition eq_of_mul_inv_mem_ker_in {G : set A} [is_subgroup G] [is_hom_on f G]
      {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) (H : a₁ * a₂⁻¹ ∈ ker_in f G) :
    f a₁ = f a₂ :=
  eq_of_mul_inv_eq_one (by rewrite [-hom_on_inv f a₂G, -hom_on_mul f a₁G (inv_mem a₂G)];
    exact and.left H)

  proposition mul_inv_mem_ker_in_of_eq {G : set A} [is_subgroup G] [is_hom_on f G]
      {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) (H : f a₁ = f a₂) :
    a₁ * a₂⁻¹ ∈ ker_in f G :=
  and.intro
    (show f (a₁ * a₂⁻¹) = 1,
      by rewrite [hom_on_mul f a₁G (inv_mem a₂G), hom_on_inv f a₂G, H, mul.right_inv])
    (mul_mem a₁G (inv_mem a₂G))

  proposition eq_iff_mul_inv_mem_ker_in {G : set A} [is_subgroup G] [is_hom_on f G]
      {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) :
    f a₁ = f a₂ ↔ a₁ * a₂⁻¹ ∈ ker_in f G :=
  iff.intro (mul_inv_mem_ker_in_of_eq a₁G a₂G) (eq_of_mul_inv_mem_ker_in a₁G a₂G)

  -- Ouch! These versions are not equivalent to the ones before.

  proposition eq_of_inv_mul_mem_ker [is_hom f] {a₁ a₂ : A} (H : a₁⁻¹ * a₂ ∈ ker f) :
```

```
  f a₁ = f a₂ :=
eq.symm (eq_of_inv_mul_eq_one (by rewrite [-hom_inv f, -hom_mul f]; exact H))

proposition inv_mul_mem_ker_of_eq [is_hom f] {a₁ a₂ : A} (H : f a₁ = f a₂) :
  a₁⁻¹ * a₂ ∈ ker f :=
show f (a₁⁻¹ * a₂) = 1, by rewrite [hom_mul f, hom_inv f, H, mul.left_inv]

proposition eq_iff_inv_mul_mem_ker [is_hom f] (a₁ a₂ : A) : f a₁ = f a₂ ↔ a₁⁻¹ * a₂ ∈ ker f :=
iff.intro inv_mul_mem_ker_of_eq eq_of_inv_mul_mem_ker

proposition eq_of_inv_mul_mem_ker_in {G : set A} [is_subgroup G] [is_hom_on f G]
    {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) (H : a₁⁻¹ * a₂ ∈ ker_in f G) :
  f a₁ = f a₂ :=
eq.symm (eq_of_inv_mul_eq_one (by rewrite [-hom_on_inv f a₁G, -hom_on_mul f (inv_mem a₁G) a₂G];
  exact and.left H))

proposition inv_mul_mem_ker_in_of_eq {G : set A} [is_subgroup G] [is_hom_on f G]
    {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) (H : f a₁ = f a₂) :
  a₁⁻¹ * a₂ ∈ ker_in f G :=
and.intro
  (show f (a₁⁻¹ * a₂) = 1,
    by rewrite [hom_on_mul f (inv_mem a₁G) a₂G, hom_on_inv f a₁G, H, mul.left_inv])
  (mul_mem (inv_mem a₁G) a₂G)

proposition eq_iff_inv_mul_mem_ker_in {G : set A} [is_subgroup G] [is_hom_on f G]
    {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G) :
  f a₁ = f a₂ ↔ a₁⁻¹ * a₂ ∈ ker_in f G :=
iff.intro (inv_mul_mem_ker_in_of_eq a₁G a₂G) (eq_of_inv_mul_mem_ker_in a₁G a₂G)

proposition eq_one_of_eq_one_of_injective [is_hom f] (H : injective f) {x : A}
    (H' : f x = 1) :
  x = 1 :=
H (by rewrite [H', hom_one f])

proposition eq_one_iff_eq_one_of_injective [is_hom f] (H : injective f) (x : A) :
  f x = 1 ↔ x = 1 :=
iff.intro (eq_one_of_eq_one_of_injective H) (λ H', by rewrite [H', hom_one f])

proposition injective_of_forall_eq_one [is_hom f] (H : ∀ x, f x = 1 → x = 1) : injective f :=
take a₁ a₂, assume Heq,
have f (a₁ * a₂⁻¹) = 1, by rewrite [hom_mul f, hom_inv f, Heq, mul.right_inv],
eq_of_mul_inv_eq_one (H _ this)

proposition injective_of_ker_eq_singleton_one [is_hom f] (H : ker f = '{1}) : injective f :=
injective_of_forall_eq_one
  (take x, suppose x ∈ ker f, by rewrite [H at this]; exact eq_of_mem_singleton this)
```

```
proposition ker_eq_singleton_one_of_injective [is_hom f] (H : injective f) : ker f = '{1} :=
ext (take x, by rewrite [mem_ker_iff, mem_singleton_iff, eq_one_iff_eq_one_of_injective H])

variable (f)

proposition injective_iff_ker_eq_singleton_one [is_hom f] : injective f ↔ ker f = '{1} :=
iff.intro ker_eq_singleton_one_of_injective injective_of_ker_eq_singleton_one

variable {f}

proposition eq_one_of_eq_one_of_inj_on {G : set A} [is_subgroup G] [is_hom_on f G]
  (H : inj_on f G) {x : A} (xG : x ∈ G) (H' : f x = 1) :
  x = 1 :=
H xG one_mem (by rewrite [H', hom_on_one f G])

proposition eq_one_iff_eq_one_of_inj_on {G : set A} [is_subgroup G] [is_hom_on f G]
  (H : inj_on f G) {x : A} (xG : x ∈ G) [is_hom_on f G] :
  f x = 1 ↔ x = 1 :=
iff.intro (eq_one_of_eq_one_of_inj_on H xG) (λ H', by rewrite [H', hom_on_one f G])

proposition inj_on_of_forall_eq_one {G : set A} [is_subgroup G] [is_hom_on f G]
  (H : ∀₀ x ∈ G, f x = 1 → x = 1) : inj_on f G :=
take a₁ a₂, assume a₁G a₂G Heq,
have f (a₁ * a₂⁻¹) = 1,
  by rewrite [hom_on_mul f a₁G (inv_mem a₂G), hom_on_inv f a₂G, Heq, mul.right_inv],
eq_of_mul_inv_eq_one (H (mul_mem a₁G (inv_mem a₂G)) this)

proposition inj_on_of_ker_in_eq_singleton_one {G : set A} [is_subgroup G] [is_hom_on f G]
  (H : ker_in f G = '{1}) : inj_on f G :=
inj_on_of_forall_eq_one
  (take x, assume xG fxone,
    have x ∈ ker_in f G, from and.intro fxone xG,
    by rewrite [H at this]; exact eq_of_mem_singleton this)

proposition ker_in_eq_singleton_one_of_inj_on {G : set A} [is_subgroup G] [is_hom_on f G]
  (H : inj_on f G) : ker_in f G = '{1} :=
ext (take x,
  begin
    rewrite [↑ker_in, mem_inter_iff, mem_ker_iff, mem_singleton_iff],
    apply iff.intro,
      {intro H', cases H' with fxone xG, exact eq_one_of_eq_one_of_inj_on H xG fxone},
    intro xone, rewrite xone, split, exact hom_on_one f G, exact one_mem
  end)

variable (f)
```

```
proposition inj_on_iff_ker_in_eq_singleton_one (G : set A) [is_subgroup G] [is_hom_on f G] :
  inj_on f G ↔ ker_in f G = '{1} :=
iff.intro ker_in_eq_singleton_one_of_inj_on inj_on_of_ker_in_eq_singleton_one

variable {f}

proposition conj_mem_ker [is_hom f] {a₁ : A} (a₂ : A) (H : a₁ ∈ ker f) : a₁^a₂ ∈ ker f :=
show f (a₁^a₂) = 1,
  by rewrite [↑conj, *(hom_mul f), hom_inv f, eq_one_of_mem_ker H, mul_one, mul.left_inv]

variable (f)

proposition is_subgroup_ker_in [instance] (S : set A) [is_subgroup S] [is_hom_on f S] :
  is_subgroup (ker_in f S) :=
⦃ is_subgroup,
  one_mem := and.intro (hom_on_one f S) one_mem,
  mul_mem := λ a aker b bker,
             obtain (fa : f a = 1) (aS : a ∈ S), from aker,
             obtain (fb : f b = 1) (bS : b ∈ S), from bker,
             and.intro (show f (a * b) = 1, by rewrite [hom_on_mul f aS bS, fa, fb, one_mul])
               (mul_mem aS bS),
  inv_mem := λ a aker,
             obtain (fa : f a = 1) (aS : a ∈ S), from aker,
             and.intro (show f (a⁻¹) = 1, by rewrite [hom_on_inv f aS, fa, one_inv])
               (inv_mem aS)
⦄

proposition is_subgroup_ker [instance] [is_hom f] : is_subgroup (ker f) :=
begin
  rewrite [-ker_in_univ f],
  have is_hom_on f univ, from is_hom_on_of_is_hom f univ,
  apply is_subgroup_ker_in f univ
end

proposition is_normal_in_ker_in [instance] (G : set A) [is_subgroup G] [is_hom_on f G] :
  is_normal_in (ker_in f G) G :=
is_normal_in_of_forall_subset
  (take x, assume xG, take y, assume yker,
    obtain z [[(fz : f z = 1) zG] (yeq : x * z = y)], from yker,
    have y = x * z * x⁻¹ * x, by rewrite [yeq, inv_mul_cancel_right],
    show y ∈ ker_in f G * x,
      begin
        rewrite this,
        apply mul_mem_rcoset,
        apply and.intro,
        show f (x * z * x⁻¹) = 1,
```

```
          by rewrite [hom_on_mul f (mul_mem xG zG) (inv_mem xG), hom_on_mul f xG zG, fz,
                      hom_on_inv f xG, mul_one, mul.right_inv],
        show x * z * x⁻¹ ∈ G, from mul_mem (mul_mem xG zG) (inv_mem xG)
      end)

  proposition is_normal_ker [instance] [H : is_hom f] : is_normal (ker f) :=
  begin
    rewrite [-ker_in_univ, -is_normal_in_univ_iff_is_normal],
    apply is_normal_in_ker_in,
    exact is_hom_on_of_is_hom f univ
  end

end groupAB

section subgroupH
  variables [group A] [group B] {H : set A} [is_subgroup H]
  variables {f : A → B} [is_hom f]

  proposition subset_ker_of_forall (hyp : ∀ x y, x * H = y * H → f x = f y) : H ⊆ ker f :=
  take h, assume hH,
  have h * H = 1 * H, by rewrite [lcoset_eq_self_of_mem hH, one_lcoset],
  have f h = f 1, from hyp h 1 this,
  show f h = 1, by rewrite [this, hom_one f]

  proposition eq_of_lcoset_eq_lcoset_of_subset_ker {x y : A} (hyp₀ : x * H = y * H)
    (hyp₁ : H ⊆ ker f) :
    f x = f y :=
  have y⁻¹ * x ∈ H, from inv_mul_mem_of_lcoset_eq_lcoset hyp₀,
  eq.symm (eq_of_inv_mul_mem_ker (hyp₁ this))

  variables (H f)
  proposition subset_ker_iff : H ⊆ ker f ↔ ∀ x y, x * H = y * H → f x = f y :=
  iff.intro (λ h₁ x y h₀, eq_of_lcoset_eq_lcoset_of_subset_ker h₀ h₁) subset_ker_of_forall
end subgroupH

section subgroupGH
  variables [group A] [group B] {G H : set A} [is_subgroup G] [is_subgroup H]
  variables {f : A → B} [is_hom_on f G]

  proposition subset_ker_in_of_forall (hyp₀ : ∀₀ x ∈ G, ∀₀ y ∈ G, x * H = y * H → f x = f y)
      (hyp₁ : H ⊆ G) :
    H ⊆ ker_in f G :=
  take h, assume hH,
    have hG : h ∈ G, from hyp₁ hH,
    and.intro
      (have h * H = 1 * H, by rewrite [lcoset_eq_self_of_mem hH, one_lcoset],
```

```
        have f h = f 1, from hyp₀ hG one_mem this,
        show f h = 1, by rewrite [this, hom_on_one f G])
      hG

  proposition eq_of_lcoset_eq_lcoset_of_subset_ker_in {x : A} (xG : x ∈ G) {y : A} (yG : y ∈ G)
      (hyp₀ : x * H = y * H) (hyp₁ : H ⊆ ker_in f G) :
    f x = f y :=
  have y⁻¹ * x ∈ H, from inv_mul_mem_of_lcoset_eq_lcoset hyp₀,
  eq.symm (eq_of_inv_mul_mem_ker_in yG xG (hyp₁ this))

  variables (H f)

  proposition subset_ker_in_iff :
    H ⊆ ker_in f G ↔ (H ⊆ G ∧ ∀₀ x ∈ G, ∀₀ y ∈ G, x * H = y * H → f x = f y) :=
  iff.intro
    (λ h₁, and.intro
      (subset.trans h₁ (inter_subset_right _ _))
      (λ x xG y yG h₀, eq_of_lcoset_eq_lcoset_of_subset_ker_in xG yG h₀ h₁))
    (λ h, subset_ker_in_of_forall (and.right h) (and.left h))
end subgroupGH


/- the centralizer -/

section has_mulA
  variable [has_mul A]

  abbreviation centralizes [reducible] (a : A) (S : set A) : Prop := ∀₀ b ∈ S, a * b = b * a

  definition centralizer (S : set A) : set A := { a : A | centralizes a S }

  abbreviation is_centralized_by (S T : set A) : Prop := T ⊆ centralizer S

  abbreviation centralizer_in (S T : set A) : set A := T ∩ centralizer S

  proposition mem_centralizer_iff_centralizes (a : A) (S : set A) :
    a ∈ centralizer S ↔ centralizes a S := iff.refl _

  proposition normalizes_of_centralizes {a : A} {S : set A} (H : centralizes a S) :
    normalizes a S :=
  ext (take b, iff.intro
    (suppose b ∈ a * S,
      obtain s [ains (beq : a * s = b)], from this,
      show b ∈ S * a, by rewrite[-beq, H ains]; apply mem_image_of_mem _ ains)
    (suppose b ∈ S * a,
      obtain s [ains (beq : s * a = b)], from this,
```

```
      show b ∈ a * S, by rewrite[-beq, -H ains]; apply mem_image_of_mem _ ains))

  proposition centralizer_subset_normalizer (S : set A) : centralizer S ⊆ normalizer S :=
  λ a acent, normalizes_of_centralizes acent

  proposition centralizer_subset_centralizer {S T : set A} (ssubt : S ⊆ T) :
    centralizer T ⊆ centralizer S :=
  λ x xCentT s sS, xCentT _ (ssubt sS)
end has_mulA

section groupA
  variable [group A]

  proposition is_subgroup_centralizer [instance] [group A] (S : set A) :
    is_subgroup (centralizer S) :=
  ⦃ is_subgroup,
    one_mem := λ b bS, by rewrite [one_mul, mul_one],
    mul_mem := λ a acent b bcent c cS, by rewrite [mul.assoc, bcent cS, -*mul.assoc, acent cS],
    inv_mem := λ a acent c cS, eq_mul_inv_of_mul_eq
      (by rewrite [mul.assoc, -acent cS, inv_mul_cancel_left])⦄
end groupA


/- the subgroup generated by a set -/

section groupA
  variable [group A]

  inductive subgroup_generated_by (S : set A) : A → Prop :=
  | generators_mem : ∀ x, x ∈ S → subgroup_generated_by S x
  | one_mem        : subgroup_generated_by S 1
  | mul_mem        : ∀ x y, subgroup_generated_by S x → subgroup_generated_by S y →
                       subgroup_generated_by S (x * y)
  | inv_mem        : ∀ x, subgroup_generated_by S x → subgroup_generated_by S (x⁻¹)

  theorem generators_subset_subgroup_generated_by (S : set A) : S ⊆ subgroup_generated_by S :=
  subgroup_generated_by.generators_mem

  theorem is_subgroup_subgroup_generated_by [instance] (S : set A) :
    is_subgroup (subgroup_generated_by S) :=
  ⦃ is_subgroup,
    one_mem := subgroup_generated_by.one_mem S,
    mul_mem := λ a amem b bmem, subgroup_generated_by.mul_mem a b amem bmem,
    inv_mem := λ a amem, subgroup_generated_by.inv_mem a amem ⦄

  theorem subgroup_generated_by_subset {S G : set A} [is_subgroup G] (H : S ⊆ G) :
```

```
    subgroup_generated_by S ⊆ G :=
  begin
    intro x xgenS,
    induction xgenS with a aS a b agen bgen aG bG a agen aG,
      {exact H aS},
      {exact one_mem},
      {exact mul_mem aG bG},
    exact inv_mem aG
  end
end groupA

end group_theory
```

```
/-
Copyright (c) 2016 Jeremy Avigad. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Jeremy Avigad

Turn a subgroup into a group on the corresponding subtype. Given

  variables {A : Type} [group A] (G : set A) [is_subgroup G]

we have:

  group_of G      := G, viewed as a group
  to_group_of G a := if a is in G, returns the image in group_of G, or 1 otherwise
  to_subgroup a   := given a : group_of G, return the underlying element
-/
import .basic
open set function subtype classical

variables {A B C : Type}

namespace group_theory

definition group_of (G : set A) : Type := subtype G

definition subgroup_to_group {G : set A} ⦃a : A⦄ (aG : a ∈ G) : group_of G := tag a aG

definition to_subgroup {G : set A} (a : group_of G) : A := elt_of a

proposition to_subgroup_mem {G : set A} (a : group_of G) : to_subgroup a ∈ G := has_property a

variables [group A] (G : set A) [is_subgroup G]

definition group_of.group [instance] : group (group_of G) :=
⦃ group,
  mul          := λ a b,  subgroup_to_group (mul_mem (to_subgroup_mem a) (to_subgroup_mem b)),
  mul_assoc    := λ a b c, subtype.eq !mul.assoc,
  one          := subgroup_to_group (@one_mem A _ G _),
  one_mul      := λ a, subtype.eq !one_mul,
  mul_one      := λ a, subtype.eq !mul_one,
  inv          := λ a, tag (elt_of a)⁻¹ (inv_mem (to_subgroup_mem a)),
  mul_left_inv := λ a, subtype.eq !mul.left_inv
⦄

proposition is_hom_group_to_subgroup [instance] : is_hom (@to_subgroup A G) :=
is_mul_hom.mk
  (take g₁ g₂ : group_of G,
```

```
    show to_subgroup (g₁ * g₂) = to_subgroup g₁ * to_subgroup g₂,
      by cases g₁; cases g₂; reflexivity)

noncomputable definition to_group_of (a : A) : group_of G :=
if H : a ∈ G then subgroup_to_group H else 1

proposition is_hom_on_to_group_of [instance] : is_hom_on (to_group_of G) G :=
take g₁, assume g₁G, take g₂, assume g₂G,
show to_group_of G (g₁ * g₂) = to_group_of G g₁ * to_group_of G g₂,
  by rewrite [↑to_group_of, dif_pos g₁G, dif_pos g₂G, dif_pos (mul_mem g₁G g₂G)]

proposition to_group_to_subgroup : left_inverse (to_group_of G) to_subgroup :=
begin
  intro a, rewrite [↑to_group_of, dif_pos (to_subgroup_mem a)],
  apply subtype.eq, reflexivity
end

-- proposition to_subgroup_to_group {a : A} (aG : a ∈ G) : to_subgroup (to_group_of G a) = a :=
-- by rewrite [↑to_group_of, dif_pos aG]
-- curiously, in the next version, "by rewrite [↑to_group_of, dif_pos aG]" doesn't work.

proposition to_subgroup_to_group : left_inv_on to_subgroup (to_group_of G) G :=
λ a aG, by xrewrite [dif_pos aG]

variable {G}

proposition inj_on_to_group_of : inj_on (to_group_of G) G :=
inj_on_of_left_inv_on (to_subgroup_to_group G)

variable (G)

proposition surj_on_to_group_of_univ : surj_on (to_group_of G) G univ :=
take y, assume yuniv, mem_image (to_subgroup_mem y) (to_group_to_subgroup G y)

proposition image_to_group_of_eq_univ : to_group_of G ' G = univ :=
image_eq_of_maps_to_of_surj_on (maps_to_univ _ _) (surj_on_to_group_of_univ G)

proposition surjective_to_group_of : surjective (to_group_of G) :=
surjective_of_has_right_inverse (exists.intro _ (to_group_to_subgroup G))

variable {G}

proposition to_group_of_preimage_to_group_of_image {S : set A} (SsubG : S ⊆ G) :
  (to_group_of G) '- (to_group_of G ' S) ∩ G = S :=
ext (take x, iff.intro
  (assume H,
```

```
    obtain Hx (xG : x ∈ G), from H,
    have to_group_of G x ∈ to_group_of G ' S, from mem_of_mem_preimage Hx,
    obtain y [(yS : y ∈ S) (Heq : to_group_of G y = to_group_of G x)], from this,
    have y = x, from inj_on_to_group_of (SsubG yS) xG Heq,
    show x ∈ S, by rewrite -this; exact yS)
  (assume xS, and.intro
     (mem_preimage (show to_group_of G x ∈ to_group_of G ' S, from mem_image_of_mem _ xS))
     (SsubG xS)))

end group_theory
```

```
/-
Copyright (c) 2016 Jeremy Avigad. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Andrew Zipperer, Jeremy Avigad

We provide two versions of the quoptient construction. They use the same names and notation:
one lives in the namespace 'quotient_group' and the other lives in the namespace
'quotient_group_general'.

The first takes a group, A, and a normal subgroup, H. We have

  quotient H        := the quotient of A by H
  qproj H a         := the projection, with notation a' * G
  qproj H ' s       := the image of s, with notation s / G
  extend H respf    := given f : A → B respecting the equivalence relation, we get a function
                       f : quotient G → B
  bar f             := the above, G = ker f)

The definition is constructive, using quotient types. We prove all the characteristic properties.

As in the SSReflect library, we also provide a construction to quotient by an *arbitrary subgroup*.
Now we have

  quotient H        := the quotient of normalizer H by H
  qproj H a         := still denoted a '* H, the projection when a is in normalizer H,
                       arbitrary otherwise
  qproj H G         := still denoted G / H, the image of the above
  extend H G respf := given a homomorphism on G with ker_in G f ⊆ H, extends to a
                       homomorphism G / H
  bar G f           := the above, with H = ker_in f G

This quotient H is defined by composing the first one with the construction which turns
normalizer H into a group.
-/
import .subgroup_to_group theories.move
open set function subtype classical quot

namespace group_theory
open coset_notation

variables {A B C : Type}

/- the quotient group -/

namespace quotient_group
```

```
variables [group A] (H : set A) [is_normal H]

definition lcoset_setoid [instance] : setoid A :=
setoid.mk (lcoset_equiv H) (equivalence_lcoset_equiv H)

definition quotient := quot (lcoset_setoid H)

private definition qone : quotient H := ⟦ 1 ⟧

private definition qmul : quotient H → quotient H → quotient H :=
quot.lift₂
  (λ a b, ⟦ a * b ⟧)
  (λ a₁ a₂ b₁ b₂ e₁ e₂, quot.sound (lcoset_equiv_mul H e₁ e₂))

private definition qinv : quotient H → quotient H :=
quot.lift
  (λ a, ⟦ a⁻¹ ⟧)
  (λ a₁ a₂ e, quot.sound (lcoset_equiv_inv H e))

private proposition qmul_assoc (a b c : quotient H) :
  qmul H (qmul H a b) c = qmul H a (qmul H b c) :=
quot.induction_on₂ a b (λ a b, quot.induction_on c (λ c,
  have H :  ⟦ a * b * c ⟧ = ⟦ a * (b * c) ⟧, by rewrite mul.assoc,
  H))

private proposition qmul_qone (a : quotient H) : qmul H a (qone H) = a :=
quot.induction_on a (λ a', show ⟦ a' * 1 ⟧ = ⟦ a' ⟧, by rewrite mul_one)

private proposition qone_qmul (a : quotient H) : qmul H (qone H) a = a :=
quot.induction_on a (λ a', show ⟦ 1 * a' ⟧ = ⟦ a' ⟧, by rewrite one_mul)

private proposition qmul_left_inv (a : quotient H) : qmul H (qinv H a) a = qone H :=
quot.induction_on a (λ a', show ⟦ a'⁻¹ * a' ⟧ = ⟦ 1 ⟧, by rewrite mul.left_inv)

protected definition group [instance] : group (quotient H) :=
⦃ group,
  mul          := qmul H,
  inv          := qinv H,
  one          := qone H,
  mul_assoc    := qmul_assoc H,
  mul_one      := qmul_qone H,
  one_mul      := qone_qmul H,
  mul_left_inv := qmul_left_inv H
⦄

-- these theorems characterize the quotient group
```

```
definition qproj (a : A) : quotient H := ⟦ a ⟧

infix ` '* `:65  := λ {A' : Type} [group A'] a H' [is_normal H'], qproj H' a
infix ` / `      := λ {A' : Type} [group A'] G H' [is_normal H'], qproj H' ' G

proposition is_hom_qproj [instance] : is_hom (qproj H) :=
is_mul_hom.mk (λ a b, rfl)

variable {H}

proposition qproj_eq_qproj {a b : A} (h : a * H = b * H) : a '* H = b '* H :=
quot.sound h

proposition lcoset_eq_lcoset_of_qproj_eq_qproj {a b : A} (h : a '* H = b '* H) :  a * H = b * H :=
quot.exact h

variable (H)

proposition qproj_eq_qproj_iff (a b : A) : a '* H = b '* H ↔ a * H = b * H :=
iff.intro lcoset_eq_lcoset_of_qproj_eq_qproj qproj_eq_qproj

proposition ker_qproj [is_subgroup H] : ker (qproj H) = H :=
ext (take a,
  begin
    rewrite [↑ker, mem_set_of_iff, -hom_one (qproj H), qproj_eq_qproj_iff,
      one_lcoset],
    show a * H = H ↔ a ∈ H, from iff.intro mem_of_lcoset_eq_self lcoset_eq_self_of_mem
  end)

proposition qproj_eq_one_iff [is_subgroup H] (a : A) : a '* H = 1 ↔ a ∈ H :=
have H : qproj H a = 1 ↔ a ∈ ker (qproj H), from iff.rfl,
by rewrite [H, ker_qproj]

variable {H}

proposition qproj_eq_one_of_mem [is_subgroup H] {a : A} (aH : a ∈ H) : a '* H = 1 :=
iff.mpr (qproj_eq_one_iff H a) aH

proposition mem_of_qproj_eq_one [is_subgroup H] {a : A} (h : a '* H = 1) : a ∈ H :=
iff.mp (qproj_eq_one_iff H a) h

variable (H)

proposition surjective_qproj : surjective (qproj H) :=
take y, quot.induction_on y (λ a, exists.intro a rfl)
```

```
variable {H}

proposition quotient_induction {P : quotient H → Prop} (h : ∀ a, P (a '* H)) : ∀ a, P a :=
quot.ind h

proposition quotient_induction₂ {P : quotient H → quotient H → Prop}
    (h : ∀ a₁ a₂, P (a₁ '* H) (a₂ '* H)) :
  ∀ a₁ a₂, P a₁ a₂ :=
quot.ind₂ h

variable (H)

proposition image_qproj_self [is_subgroup H] : H / H = '{1} :=
eq_of_subset_of_subset
  (image_subset_of_maps_to
    (take x, suppose x ∈ H,
      show x '* H ∈ '{1},
        from mem_singleton_of_eq (qproj_eq_one_of_mem `x ∈ H`)))
  (take x, suppose x ∈ '{1},
    have x = 1, from eq_of_mem_singleton this,
    show x ∈ H / H, by rewrite this; apply mem_image_of_mem _ one_mem)

-- extending a function A → B to a function A / H → B

section respf

variable {H}
variables {f : A → B} (respf : ∀ a₁ a₂, a₁ * H = a₂ * H → f a₁ = f a₂)

definition extend : quotient H → B := quot.lift f respf

proposition extend_qproj (a : A) : extend respf (a '* H) = f a := rfl

proposition extend_comp_qproj : extend respf ∘ (qproj H) = f := rfl

proposition image_extend (G : set A) : (extend respf) ' (G / H) = f ' G :=
by rewrite [-image_comp]

variable [group B]

proposition is_hom_extend [instance] [is_hom f] : is_hom (extend respf) :=
is_mul_hom.mk (take a b,
  show (extend respf (a * b)) = (extend respf a) * (extend respf b), from
    quot.induction_on₂ a b (take a b, hom_mul f a b))
```

```
proposition ker_extend : ker (extend respf) = ker f / H :=
eq_of_subset_of_subset
  (quotient_induction
    (take a, assume Ha : qproj H a ∈ ker (extend respf),
      have f a = 1, from Ha,
      show a '* H ∈ ker f / H,
        from mem_image_of_mem _ this))
  (image_subset_of_maps_to
    (take a, assume h : a ∈ ker f,
      show extend respf (a '* H) = 1, from h))

end respf

end quotient_group


/- the first homomorphism theorem for the quotient group -/

namespace quotient_group
  variables [group A] [group B] (f : A → B) [is_hom f]

  lemma eq_of_lcoset_equiv_ker ⦃a b : A⦄ (h : lcoset_equiv (ker f) a b) : f a = f b :=
  have b⁻¹ * a ∈ ker f, from inv_mul_mem_of_lcoset_eq_lcoset h,
  eq.symm (eq_of_inv_mul_mem_ker this)

  definition bar : quotient (ker f) → B := extend (eq_of_lcoset_equiv_ker f)

  proposition bar_qproj (a : A) : bar f (a '* ker f) = f a := rfl

  proposition is_hom_bar [instance] : is_hom (bar f) := is_hom_extend _

  proposition image_bar (G : set A) : bar f ' (G / ker f) = f ' G :=
  by rewrite [↑bar, image_extend]

  proposition image_bar_univ : bar f ' univ = f ' univ :=
  by rewrite [↑bar, -image_eq_univ_of_surjective (surjective_qproj (ker f)),
      image_extend]

  proposition surj_on_bar : surj_on (bar f) univ (f ' univ) :=
  by rewrite [↑surj_on, image_bar_univ]; apply subset.refl

  proposition ker_bar_eq : ker (bar f) = '{1} :=
  by rewrite [↑bar, ker_extend, image_qproj_self]

  proposition injective_bar : injective (bar f) :=
  injective_of_ker_eq_singleton_one (ker_bar_eq f)
```

```
end quotient_group


/- a generic morphism extension property -/

section
  variables [group A] [group B] [group C]
  variables (G : set A) [is_subgroup G]
  variables (g : A → C) (f : A → B)

  noncomputable definition gen_extend : C → B := λ c, f (inv_fun g G 1 c)

  variables {G g f}

  proposition eq_of_ker_in_subset {a₁ a₂ : A} (a₁G : a₁ ∈ G) (a₂G : a₂ ∈ G)
      [is_hom_on g G] [is_hom_on f G] (Hker : ker_in g G ⊆ ker f) (H' : g a₁ = g a₂) :
    f a₁ = f a₂ :=
  have memG : a₁⁻¹ * a₂ ∈ G, from mul_mem (inv_mem a₁G) a₂G,
  have a₁⁻¹ * a₂ ∈ ker_in g G, from inv_mul_mem_ker_in_of_eq a₁G a₂G H',
  have a₁⁻¹ * a₂ ∈ ker_in f G, from and.intro (Hker this) memG,
  show f a₁ = f a₂, from eq_of_inv_mul_mem_ker_in a₁G a₂G this

  proposition gen_extend_spec [is_hom_on g G] [is_hom_on f G] (Hker : ker_in g G ⊆ ker f)
    {a : A} (aG : a ∈ G) : gen_extend G g f (g a) = f a :=
  eq_of_ker_in_subset (inv_fun_spec' aG) aG Hker (inv_fun_spec aG)

  proposition is_hom_on_gen_extend [is_hom_on g G] [is_hom_on f G] (Hker : ker_in g G ⊆ ker f) :
    is_hom_on (gen_extend G g f) (g ' G) :=
  have is_subgroup (g ' G), from is_subgroup_image g G,
  take c₁, assume c₁gG : c₁ ∈ g ' G,
  take c₂, assume c₂gG : c₂ ∈ g ' G,
  let ginv := inv_fun g G 1 in
  have Hginv : maps_to ginv (g ' G) G, from maps_to_inv_fun one_mem,
  have ginvc₁ : ginv c₁ ∈ G, from Hginv c₁gG,
  have ginvc₂ : ginv c₂ ∈ G, from Hginv c₂gG,
  have ginvc₁c₂ : ginv (c₁ * c₂) ∈ G, from Hginv (mul_mem c₁gG c₂gG),
  have HH : ∀₀ c ∈ g ' G, g (ginv c) = c,
    from λ a aG, right_inv_on_inv_fun_of_surj_on _ (surj_on_image g G) aG,
  have eq₁ : g (ginv c₁) = c₁, from HH c₁gG,
  have eq₂ : g (ginv c₂) = c₂, from HH c₂gG,
  have eq₃ : g (ginv (c₁ * c₂)) = c₁ * c₂, from HH (mul_mem c₁gG c₂gG),
  have g (ginv (c₁ * c₂)) = g ((ginv c₁) * (ginv c₂)),
    by rewrite [eq₃, hom_on_mul g ginvc₁ ginvc₂, eq₁, eq₂],
  have f (ginv (c₁ * c₂)) = f (ginv c₁ * ginv c₂),
    from eq_of_ker_in_subset (ginvc₁c₂) (mul_mem ginvc₁ ginvc₂) Hker this,
  show f (ginv (c₁ * c₂)) = f (ginv c₁) * f (ginv c₂),
```

```
    by rewrite [this, hom_on_mul f ginvc₁ ginvc₂]
end


/- quotient by an arbitrary group, not necessarily normal -/

namespace quotient_group_general

variables [group A] (H : set A) [is_subgroup H]

lemma is_normal_to_group_of_normalizer [instance] :
  is_normal (to_group_of (normalizer H) ' H) :=
have H1 : is_normal_in (to_group_of (normalizer H) ' H)
                        (to_group_of (normalizer H) ' (normalizer H)),
  from is_normal_in_image_image (subset_normalizer_self H) (to_group_of (normalizer H)),
have H2 : to_group_of (normalizer H) ' (normalizer H) = univ,
  from image_to_group_of_eq_univ (normalizer H),
is_normal_of_is_normal_in_univ (by rewrite -H2; exact H1)

section quotient_group
open quotient_group

noncomputable definition quotient : Type := quotient (to_group_of (normalizer H) ' H)

noncomputable definition group_quotient  [instance] : group (quotient H) :=
quotient_group.group (to_group_of (normalizer H) ' H)

noncomputable definition qproj : A → quotient H :=
qproj (to_group_of (normalizer H) ' H) ∘ (to_group_of (normalizer H))

infix ` '* `:65  := λ {A' : Type} [group A'] a H' [is_subgroup H'], qproj H' a
infix ` / `      := λ {A' : Type} [group A'] G H' [is_subgroup H'], qproj H' ' G

proposition is_hom_on_qproj [instance] : is_hom_on (qproj H) (normalizer H) :=
have H₀ : is_hom_on (to_group_of (normalizer H)) (normalizer H),
  from is_hom_on_to_group_of (normalizer H),
have H₁ : is_hom_on (quotient_group.qproj (to_group_of (normalizer H) ' H)) univ,
  from iff.mpr (is_hom_on_univ_iff (quotient_group.qproj (to_group_of (normalizer H) ' H)))
         (is_hom_qproj (to_group_of (normalizer H) ' H)),
is_hom_on_comp H₀ H₁ (maps_to_univ (to_group_of (normalizer H)) (normalizer H))

proposition is_hom_on_qproj' [instance] (G : set A) [is_normal_in H G] :
  is_hom_on (qproj H) G :=
is_hom_on_of_subset (qproj H) (subset_normalizer G H)

proposition ker_in_qproj : ker_in (qproj H) (normalizer H) = H :=
```

```
let tg := to_group_of (normalizer H) in
begin
  rewrite [↑ker_in, ker_eq_preimage_one, ↑qproj, preimage_comp, -ker_eq_preimage_one],
  have is_hom_on tg H, from is_hom_on_of_subset _ (subset_normalizer_self H),
  have is_subgroup (tg ' H), from is_subgroup_image tg H,
  krewrite [ker_qproj, to_group_of_preimage_to_group_of_image (subset_normalizer_self H)]
end

end quotient_group

variable {H}

proposition qproj_eq_qproj_iff {a b : A} (Ha : a ∈ normalizer H) (Hb : b ∈ normalizer H) :
   a '* H = b '* H ↔ a * H = b * H :=
by rewrite [lcoset_eq_lcoset_iff, eq_iff_inv_mul_mem_ker_in Ha Hb, ker_in_qproj,
            -inv_mem_iff, mul_inv, inv_inv]

proposition qproj_eq_qproj {a b : A} (Ha : a ∈ normalizer H) (Hb : b ∈ normalizer H)
    (h : a * H = b * H) :
  a '* H = b '* H :=
iff.mpr (qproj_eq_qproj_iff Ha Hb) h

proposition lcoset_eq_lcoset_of_qproj_eq_qproj {a b : A}
    (Ha : a ∈ normalizer H) (Hb : b ∈ normalizer H) (h : a '* H = b '* H) :
  a * H = b * H :=
iff.mp (qproj_eq_qproj_iff Ha Hb) h

variable (H)

proposition qproj_mem {a : A} {G : set A} (aG : a ∈ G) : a '* H ∈ G / H :=
mem_image_of_mem _ aG

proposition qproj_one : 1 '* H = 1 := hom_on_one (qproj H) (normalizer H)

variable {H}

proposition mem_of_qproj_mem {a : A} (anH : a ∈ normalizer H)
    {G : set A} (HsubG : H ⊆ G) [is_subgroup G] [is_normal_in H G]
  (aHGH : a '* H ∈ G / H): a ∈ G :=
have GH : G ⊆ normalizer H, from subset_normalizer G H,
obtain b [bG (bHeq : b '* H = a '* H)], from aHGH,
have b * H = a * H, from lcoset_eq_lcoset_of_qproj_eq_qproj (GH bG) anH bHeq,
have a ∈ b * H, by rewrite this; apply mem_lcoset_self,
have a ∈ b * G, from lcoset_subset_lcoset b HsubG this,
show a ∈ G, by rewrite [lcoset_eq_self_of_mem bG at this]; apply this
```

```
proposition qproj_eq_one_iff {a : A} (Ha : a ∈ normalizer H) : a '* H = 1 ↔ a ∈ H :=
by rewrite [-hom_on_one (qproj H) (normalizer H), qproj_eq_qproj_iff Ha one_mem, one_lcoset,
        lcoset_eq_self_iff]

proposition qproj_eq_one_of_mem {a : A} (aH : a ∈ H) : a '* H = 1 :=
iff.mpr (qproj_eq_one_iff (subset_normalizer_self H aH)) aH

proposition mem_of_qproj_eq_one {a : A} (Ha : a ∈ normalizer H) (h : a '* H = 1) : a ∈ H :=
iff.mp (qproj_eq_one_iff Ha) h

variable (H)

section
open quotient_group
proposition surj_on_qproj_normalizer : surj_on (qproj H) (normalizer H) univ :=
have H₀ : surj_on (to_group_of (normalizer H)) (normalizer H) univ,
  from surj_on_to_group_of_univ (normalizer H),
have H₁ : surj_on (quotient_group.qproj (to_group_of (normalizer H) ' H)) univ univ,
  from surj_on_univ_of_surjective univ (surjective_qproj _),
surj_on_comp H₁ H₀
end

variable {H}

proposition quotient_induction {P : quotient H → Prop} (hyp : ∀₀ a ∈ normalizer H, P (a '* H)) :
  ∀ a, P a :=
surj_on_univ_induction (surj_on_qproj_normalizer H) hyp

proposition quotient_induction₂ {P : quotient H → quotient H → Prop}
    (hyp : ∀₀ a₁ ∈ normalizer H, ∀₀ a₂ ∈ normalizer H, P (a₁ '* H) (a₂ '* H)) :
  ∀ a₁ a₂, P a₁ a₂ :=
surj_on_univ_induction₂ (surj_on_qproj_normalizer H) hyp

variable (H)

proposition image_qproj_self : H / H = '{1} :=
eq_of_subset_of_subset
  (image_subset_of_maps_to
    (take x, suppose x ∈ H,
      show x '* H ∈ '{1},
        from mem_singleton_of_eq (qproj_eq_one_of_mem `x ∈ H`)))
  (take x, suppose x ∈ '{1},
    have x = 1, from eq_of_mem_singleton this,
    show x ∈ H / H,
      by rewrite [this, -qproj_one H]; apply mem_image_of_mem _ one_mem)
```

```
section respf

variable (H)
variables [group B] (G : set A) [is_subgroup G] (f : A → B)

noncomputable definition extend : quotient H → B := gen_extend G (qproj H) f

variables [is_hom_on f G] [is_normal_in H G]

private proposition aux : is_hom_on (qproj H) G :=
is_hom_on_of_subset (qproj H) (subset_normalizer G H)

local attribute [instance] aux

variables {H f}

private proposition aux' (respf : H ⊆ ker f) : ker_in (qproj H) G ⊆ ker f :=
subset.trans
  (show ker_in (qproj H) G ⊆ ker_in (qproj H) (normalizer H),
    from inter_subset_inter_left _ (subset_normalizer G H))
  (by rewrite [ker_in_qproj]; apply respf)

variable {G}

proposition extend_qproj (respf : H ⊆ ker f) {a : A} (aG : a ∈ G) :
  extend H G f (a '* H) = f a :=
gen_extend_spec (aux' G respf) aG

proposition image_extend (respf : H ⊆ ker f) {s : set A} (ssubG : s ⊆ G) :
  extend H G f ' (s / H) = f ' s :=
begin
  rewrite [-image_comp],
  apply image_eq_image_of_eq_on,
  intro a amems,
  apply extend_qproj respf (ssubG amems)
end

variable (G)

proposition is_hom_on_extend [instance] (respf : H ⊆ ker f) : is_hom_on (extend H G f) (G / H) :=
by unfold extend; apply is_hom_on_gen_extend (aux' G respf)

variable {G}

proposition ker_in_extend [is_subgroup G] (respf : H ⊆ ker f) (HsubG : H ⊆ G) :
  ker_in (extend H G f) (G / H) = (ker_in f G) / H :=
```

```
begin
  apply ext,
  intro aH,
  cases surj_on_qproj_normalizer H (show aH ∈ univ, from trivial) with a atemp,
  cases atemp with anH aHeq,
  rewrite -aHeq,
  apply iff.intro,
  { intro akerin,
    cases akerin with aker ain,
    have a '* H ∈ G / H, from ain,
    have a ∈ G, from mem_of_qproj_mem anH HsubG this,
    have a '* H ∈ ker (extend H G f), from aker,
    have extend H G f (a '* H) = 1, from this,
    have f a = extend H G f (a '* H), from eq.symm (extend_qproj respf `a ∈ G`),
    have f a = 1, by rewrite this; assumption,
    have a ∈ ker_in f G, from and.intro this `a ∈ G`,
    show a '* H ∈ (ker_in f G) / H, from qproj_mem H this},
  intro aHker,
  have aker : a ∈ ker_in f G,
    begin
      have Hsub : H ⊆ ker_in f G, from subset_inter respf HsubG,
      have is_normal_in H (ker_in f G),
        from subset.trans (inter_subset_right (ker f) G) (subset_normalizer G H),
      apply (mem_of_qproj_mem anH Hsub aHker)
    end,
  have a ∈ G, from and.right aker,
  have f a = 1, from and.left aker,
  have extend H G f (a '* H) = 1,
    from eq.trans (extend_qproj respf `a ∈ G`) this,
  show a '* H ∈ ker_in (extend H G f) (G / H),
    from and.intro this (qproj_mem H `a ∈ G`)
end

end respf

attribute quotient [irreducible]

end quotient_group_general

/- the first homomorphism theorem for general quotient groups -/

namespace quotient_group_general

variables [group A] [group B] (G : set A) [is_subgroup G]
variables (f : A → B) [is_hom_on f G]
```

```
noncomputable definition bar : quotient (ker_in f G) → B :=
extend (ker_in f G) G f

proposition bar_qproj {a : A} (aG : a ∈ G) : bar G f (a '* ker_in f G) = f a :=
extend_qproj (inter_subset_left _ _) aG

proposition is_hom_on_bar [instance] : is_hom_on (bar G f) (G / ker_in f G) :=
have is_subgroup (ker f ∩ G), from is_subgroup_ker_in f G,
have is_normal_in (ker f ∩ G) G, from is_normal_in_ker_in f G,
is_hom_on_extend G (inter_subset_left _ _)

proposition image_bar {s : set A} (ssubG : s ⊆ G) : bar G f ' (s / ker_in f G) = f ' s :=
have is_subgroup (ker f ∩ G), from is_subgroup_ker_in f G,
have is_normal_in (ker f ∩ G) G, from is_normal_in_ker_in f G,
image_extend (inter_subset_left _ _) ssubG

proposition surj_on_bar : surj_on (bar G f) (G / ker_in f G) (f ' G) :=
by rewrite [↑surj_on, image_bar G f (@subset.refl _ G)]; apply subset.refl

proposition ker_in_bar : ker_in (bar G f) (G / ker_in f G) = '{1} :=
have H₀ : ker_in f G ⊆ ker f, from inter_subset_left _ _,
have H₁ : ker_in f G ⊆ G, from inter_subset_right _ _,
by rewrite [↑bar, ker_in_extend H₀ H₁, image_qproj_self]

proposition inj_on_bar : inj_on (bar G f) (G / ker_in f G) :=
inj_on_of_ker_in_eq_singleton_one (ker_in_bar G f)

end quotient_group_general

end group_theory
```

Chapter 5

Bibliography

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[2] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean.* `http://leanprover.github.io/tutorial/tutorial.pdf`, 2015.

[3] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* `http://isabelle.in.tum.de/doc/locales.pdf`, 2010.

[4] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types.* Perspectives in Logic. Cambridge University Press, 2013.

[5] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2013.

[6] Y. Bertot, G. Huet, P. Castéran, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2013.

[7] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[8] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[9] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. *The Lean Theorem Prover (System Description)*, pages 378–388. Springer International Publishing, 2015.

[11] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[12] George Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. *A Modular Formalisation of Finite Group Theory*, pages 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[13] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[14] Florian Kammüller and Lawrence C. Paulson. A Formal Proof of Sylow's Theorem - An Experiment in Abstract Algebra with Isabelle HOL. Technical report, Journal of Automated Reasoning, 1999.

[15] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof: An Introduction.* Cambridge University Press, 2014.

[16] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Dec 1996.

[17] Lawrence C. Paulson. *A formulation of the simple theory of types (for Isabelle)*, pages 246–274. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[18] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[19] Benjamin Werner. *Une Théorie des Constructions Inductives.* Theses, Université Paris-Diderot - Paris VII, May 1994.