A formal proof of the unsolvability of the symmetric group over a set with five or more elements

Gilles Barthe*

Department of Computer Science
University of Nijmegen

Abstract. *We present a formal proof of the unsolvability of the symmetric group over a discrete set with five or more elements. The proof is using a simulation of the notion of class; as a result, some limited but convenient form of inheritance is available.*

1 Introduction

The symmetric group over n elements S_n (i.e the group of permutations over $\{1, \ldots, n\}$) is a fundamental notion in algebra. One of the most surprising application of the symmetric group comes from Galois theory. Every polynomial f over a field K determines a subgroup G of S_n, where n is the degree of f. The fundamental result of Galois theory states that the roots of the polynomial can be obtained from K by arithmetic operations (addition, substraction, multiplication and division) and m^{th}-roots if and only if the group G is solvable (see for example [15]). If the polynomial is of degree 2, 3 or 4, we can use some formulae to get, when they exist, the roots of f. One of the most striking achievements of Galois was to prove that there is no general algorithm to compute the roots of a polynomial of degree greater than 4 by arithmetic operations and m^{th}-roots. In modern algebra, this result is stated as a corollary of the Galois isomorphism and of the unsolvability of S_n, for $n \geq 5$. We present a machine-checked proof of the second result, namely:

Theorem 1 *For $n \geq 5$, the set of permutations over $\{1, \ldots, n\}$ is unsolvable.*

The work reported here is part of a large project GALOIS whose principal aim is the formal development of Galois theory ([1],[5]). Galois is a large scale, ambitious project which has been instigated in order to confront the possibility of formalising a large body of mainstream mathematics interactively with a proof-development system. The project is in some sense open ended, as the subject matter can be extended to the whole field of polynomial algebra, field theory ...We do not expect to cover a substantial amount of material in a couple of years on our own. Nevertheless we think that the enterprise is worthwhile, as it raises some fundamental questions such as the potential of type theory as a

* email:gillesb@cs.kun.nl

framework for formal mathematics, the formal representation of mathematical notions...

The proof has been implemented in Lego, a proof-checker based on the Extended Calculus of Constructions ([13]). Lego has been succesfully used to formalise some parts of mathematics[2]. However, we faced a difficult problem when we tried to formalise the hierarchy of algebraic structures; we wanted every group to be a set as well as a monoid and a group. Unfortunately, such a facility is not available in Lego. In order to allow for some form of inheritance, we have simulated the notion of class as it exists in object-oritented languages or some functional languages (e.g. Haskell). As a result, we have been able to use single inheritance: the result is that the syntax is more readable (e.g. we can write 1_M for the unit of a monoid M but also 1_G for the unit of a group G), results can be reused (e.g. the construction of the monoid generated by a subset S of a monoid M can be used when M is a group). We believe that the use of classes makes the proof more elegant and closer to informal mathematics. However, our simulation of classes is just an implementation trick and the concept of class in a type theory with strong sums has to be studied further.

The paper is divided as follows: section 2 provides an overview of Lego. Section 3 introduces the concept of class in type theory. A full description of the implementation is out of the range of this paper and we refer the interested reader to [2] and [7]. The proof is documented in section 4. Section 5 studies the difference between a standard mathematical proof and the proof we followed. In section 6, we draw the conclusions of our work. We chose not to present any Lego code for the sake of conciseness. It can be obtained on request from the author.

2 An overview of Lego

Lego is an interactive proof-development system developed by R.Pollack in Edinburgh ([13]). It is based on the Extended Calculus of Constructions with inductive types: the Extended Calculus of Constructions or ECC ([11]) is an example of Pure Type System ([4]) with strong sums (i.e. Σ-types) and a cumulative hierarchy of universes. ECC is a very powerful type theory in which mathematical notions can be formalised ([12]). However, it is much more convenient to add inductive types to ECC, so that basic notions (e.g. booleans, integers, lists...) can be easily formalised without any tedious encoding.

2.1 Proving in Lego

In the cumulative hierarchy of ECC, the universe of propositions is included in the universe of types: intuitively, a proposition is lifted to the type of its proofs.

[2] Examples of formalisation include the construction of the reals from the rational numbers ([9]), the strong normalisation of second-order λ-calculus ([3]), the theory of Pure Type Systems ([10])...

We can enter the proof state of Lego both to prove a proposition and to construct an inhabitant of a type. The two basic tactics of are introduction and refinement.

- **Introduction:** in order to construct an inhabitant of $A \to B$, where A and B are types, we can introduce a variable $x : A$ and construct a term of type B.
- **Refinement:** the construction of an inhabitant of a type B can be reduced to the construction of an inhabitant of a type A and of a map $f : A \to B$.

The other tactics are mainly variants of this two tactics. One of the distinguishing features of Lego is that the set of tactics is fixed and is very simple. One advantage of the approach is that proofs are then developed in a unified framework and we do not have the problem of understanding the tactics developed by somebody else in order to understand his files. One inconvenient is that the pattern of a proof is sometimes obscured by the absence of tactics.

2.2 Defining in Lego

There is nothing special about defining notions in Lego. In general a notion is defined by an equality, with its name of the left and its meaning on the right. As mentioned above, you can also enter the proof state to define an object. It is sometimes quite useful to use such a technique and has been done in several occasions during our proof.

3 Classes

3.1 Intuition

In order to formalise mathematics, it is convenient to use some overloading mechanism. That is, we want some maps to accept as arguments not only objects of a given type, but objects of several types. An example coming from informal mathematics is $+$: $+$ is used for addition on the natural numbers, in an abelian group, in a ring ...We say that $+$ is an overloaded operator. If we formalise routinely algebra in type theory, we cannot overload operators, so we would have to introduce a new function symbol for every use of $+$ we make: e.g. we will have to use $+_N$ to denote addition on natural numbers, $+_G^{abgroup}$ to denote addition in an abelian group G, $+_R^{ring}$ to denote addition in a ring R...The syntax becomes too heavy! So we must look for a technique to implement some form of overloading in a proof-checker. The syntax and semantics of overloading is problematic if we do not impose restrictions. Inheritance is a restricted form of overloading which is powerful enough for our purposes: the idea is to impose a partial order on types and every type will inherit of the maps of its supertypes. Let us exemplify how our example would work in a system with inheritance: the type of rings can be defined as a subtype of abelian groups. If we define $+$ as the polymorphic addition over abelian groups (i.e $+_G$ is the addition over G for every abelian group), then the type of rings inherits of $+$ so that we can write

$+_R$ for every ring R. The idea is simple enough but its formalisation is quite problematic. It is not clear yet how to combine subtyping with dependent types and so we needed to turn to something else.

The approach we have chosen circumvenes the use of a partial order on types. We separate between classes and types. Roughly, a class is a type T with a finite set $Incl$ of closed terms $\{t_1 : B \rightarrow A_1, \ldots, t_n : B \rightarrow A_n\}$. Elements of $incl$ are meant as inclusion maps. For example, one possible set of inclusions to form the class of rings is $\{rag : Ring \rightarrow AGroup,\ rg : Ring \rightarrow Group,\ rm : Ring \rightarrow Monoid, rs : Ring \rightarrow Set\}$ where

- $Ring$ is the (large) type of rings,
- $AGroup$ is the (large) type of abelian groups,
- $Group$ is the (large) type of groups,
- $Monoid$ is the (large) type of monoids,
- Set is the (large) type of sets.

and rag, rg, rm, rs are the obvious forgetful maps. The only requirement on a class is that there should be at most one inclusion from the underlying type of the class to any other type. Precisely, $(T, Incl)$ is a class if T is a type, $Incl$ is a finite set of closed terms $\{t_1 : B \rightarrow A_1, \ldots, t_n : B \rightarrow A_n\}$ and $A_i \neq_\beta A_j$ for $i \neq j$. We want to develop a "class theory" that extends the type theory with a more liberal syntax. First we assign to every type T a set of closed terms $Incl$ such that $(T, Incl)$ is a class. The type theory is now extended so as to allow expressions such as fa where $f : A \rightarrow B$, $a : A'$, provided there exists an inclusion map from A' to A. In this case, the term fa will be a valid term of type B. In the example of the rings, we can define a map $+$ with domain $AGroup$ and apply it to a ring, as there exists an inclusion map from rings to abelian groups. Note that there is a natural ordering relation on classes: a class (T', I') is a subclass of (T, I) if there exists an inclusion map i_0 from T' to T and for every inclusion map i in I, the composition of i_0 with i lies in I'. For the purpose of our proof, it is enough for the ordering to be a tree; in fact, the following holds: if (T', I') is a subclass of (T, I), then every inclusion map in I' can be obtained by composing the inclusion map i_0 from T' to T with an inclusion map in I. The following picture indicates which inclusions are assumed:

$$\begin{array}{ccc} & Group & \\ & \downarrow & \\ & Monoid & \\ & \downarrow & \\ Poset \rightarrow & Set & \leftarrow DSet \end{array}$$

where $Poset$ is the type of posets and $DSet$ is the type of discrete sets (as we are working in a constructive framework, we distinguish between sets and sets with a decidable equality).

3.2 Implementation

The simulation of classes we have been using has been implemented within the language of the proof-checker itself, by exploiting the argument synthesis mechanisms. As we did not want to modify the proof-checker, we needed to use a trick in order to assign a set of inclusions to a type. As mentioned above, the inclusion maps assigned to a subclass (T', I') of a class (T, I) are all obtained by composing a single map from T to T' with the inclusions maps of I. What we did is to start with:

- a type of classes $Class$ and a map $ob : Class \rightarrow Type$ that extracts from a class its first component,
- a base class $\mathcal{SET}$ whose type is Set and whose set of inclusions is empty,
- an operator $newclass$ to form new classes from old ones: given a class C, a type T and a map $f : T \rightarrow (ob\ C)$, we get a new class C' whose underlying type is T and whose inclusions are obtained from the inclusions of C by composing them with f.

The following equalities are required: $ob\ \mathcal{SET} = Set$ and $ob\ (newclass\ C\ T\ f) = T$ where C is class, T is a type and $f : T \rightarrow (ob\ C)$.

Although we do not have a notion of subclass, we are able to define a lift operator that takes as argument any map $f : (ob\ C) \rightarrow T$, where T is a class and T is a type, such that $lift\ f\ a$ is valid (i.e. accepted and typable by the computer) for every element $a : ob\ C'$, where C' is a class that has been obtained from C by several (possibly 0) iterations of the newclass constructor. Maps of the form $lift\ f$ are called methods. For example, $+$ can be defined as a method on abelian groups.

The full details of the implementation can be found in [7], together with a complete formalisation of algebraic notions up to discrete fields.

Note: in order to present the proof, we have found convenient to introduce the following syntax: if C is a class, then $x : C$ will denote that x is an object of a subclass of C. For example, $g : \mathcal{MONOID}$, where $\mathcal{MONOID}$ is the class of monoids, for every group g.

4 The proof

The lego code can conveniently be organised into modules. We have used this facility to separate the various ingredients required for the proof. Each module will fall into one of this category:

- Basic modules: natural numbers, booleans, vectors, some properties of existential quantification.
- Fundamental modules: examples include the definition of the basic mathematical structures used in the proof, such as sets, categories, posets, monoids, groups, the group S_n ... as well as some basic constructions such as the submonoid generated by a subset of a monoid, the construction of the subcategory of isomorphisms of a category ...

– Specialised modules: the specialised modules are devoted two two topics: first, the definition and characterisation of solvability. Second, the proof of Theorem 1.

We hope that the proof is organised in such a way that the code can be easily re-used for further work in algebra. A substantial body of the results proved in the basic modules are already found in the Lego library, so we will not document such modules.

Note: We have followed the standard convention to drop subscripts when there is no risk of confusion.

4.1 Fundamental Modules

Sets, discrete sets and posets The representation of sets in Lego has not yet reached a consensus. Some authors prefer to model sets as a total equivalence relations, some others as a partial equivalence relation. An account on the representation of the basic notions of set, map... can be found in [6], where we justify all the choices made here.

Definition 2 *A* **set** *consists of a type T, a binary relation $=_T$ on T and a proof that $=_T$ is an equivalence relation. Sets form a class $\mathcal{SET}$ whose type of objects is Set.*

The methods on $\mathcal{SET}$ are:

– set: If $s : \mathcal{SET}$, then $set\ s : Set$ is its underlying set.
– el: If $s : \mathcal{SET}$, then $el\ s : Type$ is the type of elements of s.
– $=$: If $s : \mathcal{SET}$, then $=_s$ is the equality of $set\ s$.
– $ax - set$: If $s : \mathcal{SET}$, then $ax - set\ s$ is a proof that $=_s$ is an equivalence relation. We have defined three auxiliary methods $eq - refl$, $eq - sym$, $eq - trans$ that extract the proofs of the relexivity, symmetry and transitivity of $=_s$.

In some cases, it is necessary for the equality to be decidable. We could formalise discrete sets, i.e. sets with a decidable equality, as sets with a proof that any two elements of the set are provably equal or different. However, the formalisation is too weak (for example, we cannot define maps by cases) and we have to introduce a boolean-valued equality.

Definition 3 *A* **discrete set** *consists of a type T, a binary boolean-valued relation $\doteq_T$ on T and a proof that the induced proposition-valued binary relation $=_T$ is an equivalence relation. Discrete sets form a class $\mathcal{DSET}$ whose type of objects is DSet. $\mathcal{DSET}$ is a child of $\mathcal{SET}$.*

The methods on $\mathcal{DSET}$ are:

– $dset$: If $s : \mathcal{DSET}$, then $dset\ s : DSet$ is its underlying discrete set.
– $\doteq$: If $s : \mathcal{DSET}$, then $\doteq_s$ is the boolean-valued equality of $dset\ s$.
– $\neq$: If $s : \mathcal{DSET}$, then $\neq_s$ is the boolean-valued inequality of $dset\ s$.

Posets can be introduced as sets with an order relation respecting equality or as a type with a reflexive and transitive binary relation. Equality is then defined as the greatest anti-symmetric subrelation contained in the partial order.

Definition 4 *A* **poset** *consists of a type T, a binary relation $\leq_T$ on T and a proof that $\leq_T$ is reflexive and transitive. Posets form a class $\mathcal{POSET}$ whose type of objects is Poset. $\mathcal{POSET}$ is a child of $\mathcal{SET}$: given a triple $(T, \leq_T, pr)$ and $x, y : T$, we define $x =_T y$ as $x \leq_T y$ and $y \leq_T x$.*

The methods on $\mathcal{POSET}$ are:

- $poset$: If $s : \mathcal{POSET}$, then $poset\ s : Poset$ is its underlying poset.
- $\leq$: If $s : \mathcal{POSET}$, then $\leq_s$ is the partial order of s.
- $ax-poset$: If $s : \mathcal{POSET}$, then $ax-posets$ is the proof that $\leq_s$ is reflexive, transitive.

Maps Set-theoretical maps are not mere maps between the underlying type of elements of two sets. They need to preserve equalities. So maps from a set A to a set B will be a pair, consisting of an inhabitant f of $(el\ A) \rightarrow (el\ B)$ with a proof that f is extensional. In our formalisation, maps are defined as a method over $\mathcal{SET}$, so that we can talk about set-theoretical maps over discrete sets or monoids.

Definition 5 *The type of maps between two sets is defined as:*

$$map = \lambda A, B : \mathcal{SET}.\Sigma f : (el\ A) \rightarrow (el\ B).\Pi x, y : (el\ A).(x =_A y) \rightarrow (fx) =_B (fy)$$

The type of binary maps is defined as $bmap = \lambda A, B, C : \mathcal{SET}.map\ (A \times B)\ C$ where $\times$ is the cartesian product of two sets.

The type of unary operations is $op1 = \lambda A : \mathcal{SET}.map\ A\ A$. The type of unary operations is $op2 = \lambda A : \mathcal{SET}.bmap\ A\ A\ A$.

Most of the maps occuring in the proof are proved to be extensional. However, we shall not refer to extensionality any further as it is a tedious routine underlying our approach. We shall merely distinguish between extensional and non-extensional maps by using the notation $f : map\ A\ B$ where A and B are sets, if f is extensional. Otherwise we use $f : (el\ A) \rightarrow (el\ B)$. Moreover, we shall write fa where $f : map\ A\ B$ and $a : el\ A$ for two sets A and B: the pseudo-term fa is not typable but its intended meaning $f_1 a$ is clear.

Similarly, poset-theoretical maps will consist of a map on the underlying type of elements of the posets and a proof that it preserves inequalities.

Definition 6 *The type of maps between two posets is defined as:*

$$\begin{aligned} map{-}poset = \lambda A, B : \mathcal{POSET}.\Sigma f : (el\ A) \rightarrow (el\ B).& \\ \Pi x, y : (el\ A).(x \leq_A y) \rightarrow (fx) \leq_B (fy)& \end{aligned}$$

Powerset and powerposet A predicate over a set A is an element of $map\ A\ \Omega$, where Ω is the set whose underlying type is $Prop$ and whose underlying equality is logical equivalence. It is more suitable, in order to get an elegant proof, to define powerset and powerposet as methods.

Definition 7 *The* **powerset** Px *of an element* $x : \mathcal{SET}$ *is the set* $(map\ A\ \Omega, =_{Px})$ *where for every* $P, P' : map\ A\ \ \Omega$, $P =_{Px} P'$ *if for every* $y : el\ x$, $(Px) \leftrightarrow (P'x)$. *The* **powerposet** $P_{\leq}x$*of an element* $x : \mathcal{SET}$ *is the poset* $(map\ A\ \Omega, \subseteq)$, *where for every* $P, P' : map\ A\ \Omega$, $P \subseteq P'$ *if for every* $y : el\ x$, $(Px) \rightarrow (P'x)$.

In order to keep the syntax readable, we shall write $x \in P$ as a shorthand for $P_1 x$ where x is an element of A and P a predicate over A. Similarly, given two predicates P, P' over A, we write $P \subseteq P'$ for $P \leq P'$.

Categories

Definition 8 *A* **category** *consists of a type Obj of objects, a polymorphic set of morphisms* $Hom : Obj \rightarrow Obj \rightarrow Set$, *a polymorphic composition operator* $\bullet : \lambda A, B, C : Obj.bmap\ (Hom\ A\ B)\ (Hom\ B\ C)\ (Hom\ A\ C)$ *and a polymorphic identity* $id : \lambda A : Obj.(Hom\ A\ A)$ *satisfying the following axioms:*

- *for every objects* A, B, C, D *and morphisms* $f : el\ (Hom\ A\ B)$, $g : el\ (Hom\ B\ C)$, $h : el\ (Hom\ C\ D)$, $f \bullet (g \bullet h) = (f \bullet g) \bullet h$,
- *for every objects* A, B *and morphism* $f : el\ (Hom\ A\ B)$, $(id\ A) \bullet f = f$ *and* $f \bullet (id\ B) = f$.

Let C be a category. An **isomorphism** consists of two morphisms which are mutually inverse. The type of isomorphisms from A to B is $Iso\ A\ B$.

Proposition 9 *The isomorphisms of a given category* C *form a category* C^{iso}.

Proof: The isomorphims between two objects A and B can be turned into a set by taking as equality the equality on the first component. Formally, $f =_{(Iso\ A\ B)} g$ if $f_1 =_{(Hom\ A\ B)} g_1$. Composition is defined as follows: let $f,\ g$ be two isomorphisms from A to B and B to C respectively. Then $f \bullet g = (f_1 \bullet g_1, g_2 \bullet f_2)$. The composition of two isomorphisms is still an isomorphism. Moreover, the composition of isomorphisms is associative. The identity morphism of an object A in C^{iso} is $(id\ A, id\ A)$.□

One example of category is the category of sets.

Proposition 10 *Sets form a category SET.*

Monoids

Definition 11 *A monoid consists of a set* S, *a binary map* $\circ : op2\ S$ *and an element* e *of* S *such that* $\circ$ *is associative and* e *is a unit for* $\circ$. *Monoids form a class* $\mathcal{MONOID}$ *whose type of objects is Monoid.* $\mathcal{MONOID}$ *is a child of* $\mathcal{SET}$.

The methods on $\mathcal{MONOID}$ are:

- mon: for every $s : \mathcal{MONOID}$, $mon\ s : Monoid$ is the underlying monoid of s.
- $\circ$:for every $s : \mathcal{MONOID}$, $\circ_s : op2\ s$ is the composition operator over s.
- e: for every $s : \mathcal{MONOID}$, $e_s : el\ s$ is the unit of s.
- $ax-monoid$: if $s : \mathcal{MONOID}$, $ax-monoid\ s$ is the proof that s satisfies the axioms of a monoid. For more convenience, we have defined four methods $left-ass$, $right-ass$, $left-unit$, $right-unit$ which extract the respective proofs that $\circ$ is left and right-associative and that e is left and right-identity.

Definition 12 *A* **submonoid** *of a monoid M is a subset of M containing e and closed under $\circ$.*

Let M be a monoid. We can form, for every n-tuple of elements of M, their product.

Definition 13 *The product fp of a n-tuple of elements of M is defined inductively as follows:*

- $fp[] = e$,
- $fp((x_{n+1}, \ldots, x_1)) = x_n \circ (fp((x_n, \ldots, x_1)))$.

As $\circ$ is associative, we write informally $x_3 \circ x_2 \circ x_1$ for $x_3 \circ (x_2 \circ x_1)$. Lego does not accept such a syntax and the user has to prove himself the associativity results he needs. We have only used one result:

Lemma 14 *Let M be a monoid and $x_1, \ldots, x_n$ be elements of M. Then*

$$x_1 \circ (x_2 \circ (\ldots \circ x_n) \ldots) = (\ldots (x_1 \circ x_2) \circ \ldots) \circ x_n$$

Given a subset S of M, the set of finite products over S is a submonoid of M. Formally, we define for every subset S of M the subset

$$Submonoid\ S = \{y : (el\ M) | \exists (x_1, \ldots, x_n) : (el\ M)^n . y = fp((x_1, \ldots, x_n))\}$$

Groups

Definition 15 *A group consists of a monoid M and a map $(-)^{-1} : op1\ M$ such that for every element x of M, we have $x \circ x^{-1} = e$ and $x^{-1} \circ x = e$. Groups form a class $\mathcal{GROUP}$ whose type of objects is Group. $\mathcal{GROUP}$ is a child of $\mathcal{MONOID}$.*

The methods on $\mathcal{GROUP}$ are:

- $group$: if $s : \mathcal{GROUP}$, then $group\ s : Group$ is its underlying group.
- $(-)^{-1}$: if $s : \mathcal{GROUP}$, then $(-)^{-1}_s : op1\ s$ is the inverse operation of s.
- $ax-group$: if $s : \mathcal{GROUP}$, then $ax-group\ s$ is the proof that $(-)^{-1}$ satisifies the axioms of an inverse function (with respect to $mon\ s$). We have defined, for more convenience, two methods $left-inv, right-inv$ extracting the proofs that $(-)^{-1}$ is a left and right-inverse respectively.

Definition 16 *A subgroup of a group G is a subset of G containing e and closed under composition and inverse.*

In order to construct the subgroup generated by a subset of a group, we prove:

Proposition 17 *Let G be a group. Let S be a subset of G closed under inverse. Then $Submonoid\ S$ is a subgroup.*

Given a subset S of a group G, we form the closure of S under inverse S'. Then we define $Subgroup\ S = Submonoid\ S'$. $Subgroup\ S$ is the subgroup generated by S.

Definition 18 *Let G be a group and x, y be two elements of G. The* **conjuguate** *of x, y is $y \circ x \circ y^{-1}$. The* **commutator** *$[x, y]$ of x, y is $x \circ y \circ x^{-1} \circ y^{-1}$.*

We prove several equational results for groups:

Proposition 19 *Let G be a group. Let x, y, z be elements of G.*

- *Cancelation law: $x \circ z = y \circ z \Rightarrow x = y$.*
- *Unicity of inverse: $x \circ y = e \Rightarrow y = x^{-1}$.*
- *Distributivity of $(-)^{-1}$: $(x \circ y)^{-1} = y^{-1} \circ x^{-1}$.*
- *$(-)^{-1}$ is an involution: $(x^{-1})^{-1} = x$.*
- *Inverse of a conjuguate: $(y \circ x \circ y^{-1})^{-1} = y \circ x^{-1} \circ y^{-1}$.*
- *Product of conjuguates: $(z \circ x \circ z^{-1}) \circ (z \circ y \circ z^{-1}) = z \circ (x \circ y) \circ z^{-1}$.*
- *Inverse of a commutator: $[x, y]^{-1} = [y, x]$.*
- *Conjuguate of a commutator: $z \circ [x, y] \circ z^{-1} = [z \circ x \circ z^{-1}, z \circ y \circ z^{-1}]$.*

The symmetric group Our construction of the symmetric group is slighly different from the one of [1]. Aczel defines the symmetric group of a set to be the group of invertible elements of its monoid of endomorphisms. We work with the category of isomorphisms of sets. We prove:

Proposition 20 *For every category C and object A, $Hom_C\ A\ A$ can be given the structure of a monoid and $Hom_{C^{iso}}\ A\ A$ can be given the structure of a group. $Hom_{C^{iso}}\ A\ A$ is called the symmetric group of A in C.*

We define the **symmetric group** of a set to be its symmetric group in SET. S is defined as a method on $\mathcal{SET}$ so that it can equally be applied to discrete sets.

4.2 Specialised Modules

Solvable groups

Definition 21 *Let H_1, H_2 be two subsets of a group G. (H_1, H_2) is a* **solvable step** *if:*

- *Both H_1 and H_2 are subgroups of G and H_2 is included in H_1,*

– *For every* $x \in H_2$ *and* $y \in H_1$, $(y \circ x \circ y^{-1}) \in H_1$,
– *for every* $x, y \in H_1$, $[x, y] \in H_2$.

The second assertion is equivalent to H_2 is normal in H_1 and the third one to H_1/H_2 is abelian. However, we feel our definition is more convenient to work with.

Definition 22 *A group* G *is* **solvable** *if there exists subsets* $H_0, \ldots, H_n$ *such that* $H_0 = G$, $H_n = \{e\}$ *and for* $i = 1, \ldots, n$, (H_{i-1}, H_i) *is a solvable step.*

$H_0, \ldots, H_n$ *is called a* **solvable chain**.

One should keep in mind that the H_i's are subsets of G, so the first two equalities of the definition above have to be formalised consequently (for example, the first equality is formalised as $x \in H_0$ for every element x of G). Note also that we are using a strong version of solvability, i.e. we formalise the existence property by a Σ-type. Typically, we have $solvable : Group \rightarrow Type$.

The main result we have proved concerning solvability is its characterisation in terms of derived subgroups. Intuitively, every group G has a canonical decreasing sequence of subsets $G^0, \ldots, G^n, \ldots$, such that G is solvable if and only if the sequence is stationary and equal to $\{e\}$ from a certain rank.

Definition 23 *The* **derived subset** *of a subset* H *of a group* G *is the subset of elements of the form* $[x_1, y_1] \circ \cdots \circ [x_n, y_n]$, *where* $x_i, y_i \in H$ *for* $i = 1, \ldots, n$.

Formally, we have $D = \lambda G : Group.op1\ (P_{\leq}\ G)$. To define D, we first define $D_0 = \lambda G : Group.op1\ (P_{\leq}\ G)$, which assigns to every $H : el\ (P_{\leq}\ G))$ the subset of commutators of H. Given a group G and $H : el\ (P_{\leq}\ G))$, we define $D(H)$ as $Submonoid(D_0(H))$.

The characterisation theorem states:

Theorem 24 *A group* G *is solvable if and only if there exists a natural number* n *such that* $D^n(G) = \{e\}$.

The existence of n is strong, i.e. modelled by a Σ-type. The intuition behind the characterisation is that a group G is solvable if and only if there exists a natural number n such that $D^0(G), \ldots, D^n(G)$ is a solvable chain. The result can be analyzed in poset-theoretical terms. We have a poset A (the powerposet of a group), a unary relation P (being a subgroup of G), a binary relation Δ on A (being a solvable step) and a map $f : A \rightarrow A$ (derive). The poset also has two elements $\top$ and $\bot$ both satisfying P. We want to prove that the following statements are equivalent:

statement 1: there exists a natural number n such that $f^n \top = \bot$,
statement 2: there exists elements $a_0, \ldots, a_n \in A$ such that $a_0 = \top$, $a_n = \bot$ and for $i = 1, \ldots, n$, $\Delta(a_{i-1}, a_i)$.

In order to prove the equivalence between the two statements, some further assumptions are needed:

– f preserves inequalities,

- if $a_1, a_2 \in A$ satisfy $\Delta(a_1, a_2)$, then both Pa_1 and Pa_2 hold,
- if $a_1, a_2 \in A$ satisfy $\Delta(a_1, a_2)$, then $fa_1 \leq a_2$,
- for every $a \in A$ such that Pa holds, $\Delta(a, fa)$,
- $\top$ is the greatest element of A,
- $\bot$ is the lowest element of A satisfying P,

Assuming the assumptions above, we prove:

Proposition 25 *Statement 1 and statement 2 above are equivalent.*

Proof: Assume $f^n\top = \bot$ for some natural number n. Then $f^0\top = \top$, $f^n\top = \bot$ and for $i = 1, \ldots, n$, $\Delta(f^{i-1}\top, f^i\top)$. So Statement 2 holds.

Assume there exists elements $a_0, \ldots, a_n \in A$ such that $a_0 = \top$, $a_n = \bot$ and for $i = 1, \ldots, n$, $\Delta(a_{i-1}, a_i)$. We can prove by induction on n that $f^i\top \leq a_i$. Therefore, we get $f^n\top \leq \bot$. The other inequality is also proved by induction on n.□

In order to apply the result to the case of solvable groups, we need to prove the following results:

Proposition 26 *Let G be a group.*

(i) D is a map of posets,
(ii) D maps subgroups to subgroups,
(iii) if (H_1, H_2) is a solvable step, then $D(H_1)$ is included in H_2,
(iv) if H is a subgroup of G, then $(H, D(H))$ is a solvable step.

Proof: (i) We have to prove that for every two subsets H_1, H_2 of a group G, $D(H_1) \subseteq D(H_2)$ whenever $H_1 \subseteq H_2$. An element of $D(H_1)$ is of the form $[x_1, y_1] \circ \cdots \circ [x_n, y_n]$, where for $i = 1, \ldots, n$, $x_i, y_i \in H_1$. As $H_1 \subseteq H_2$, every element of $D(H_1)$ is also an element of $D(H_2)$. Formally, the proof is carried by induction on n.

(ii) We have already proved that the submonoid generated by a subset H of G closed under inverse is a subgroup. So we are left to prove that the subset of commutators over H is closed under inverse. Let x, y be any two elements of G. Then $[y, x] = [x, y]^{-1}$. So the subset of commutators over a subset of G is closed under inverse. It follows that D maps subgroups to subgroups.

(iii) Let (H_1, H_2) be a solvable step. An element of $D(H_1)$ is of the form $[x_1, y_1] \circ \cdots \circ [x_n, y_n]$, where for $i = 1, \ldots, n$, $x_i, y_i \in H_1$. The proof is carried by induction on n. The base case reduces to prove that H_2 contains e, which is obvious as it is a subgroup. The induction step is solved by using the closure properties of H_2. We will sketch informally the argument. Let y be an element of $D(H_1)$. Assume $y = [x_1, y_1] \circ \cdots \circ [x_n, y_n]$, where for $i = 1, \ldots, n$, $x_i, y_i \in H_1$. Proving that $y \in H_2$ reduces to prove that both $[x_1, y_1]$ and $[x_2, y_2] \circ \cdots \circ [x_n, y_n]$ lie in H_2 (as H_2 is closed under composition). The first premise follows from the definition of solvable step and the second from the induction hypothesis.

(iv) We need to prove that $D(H)$ is normal in H, i.e. for every element $x \in D(H)$ and $y \in H$, $y \circ x \circ y^{-1} \in D(H)$. The proof relies on the fact that for every x, y, z in G, we have $z \circ [x, y] \circ z^{-1} = [z \circ x \circ z^{-1}, z \circ y \circ z^{-1}]$.

We prove by induction on n that for every $z \in H$ and $x_i, y_i \in H$, where i varies over $1, \ldots, n$,

$$z\circ([x_1, y_1]\circ\cdots\circ[x_n, y_n])\circ z^{-1} = [z\circ x_1\circ z^{-1}, z\circ y_1\circ z^{-1}]\circ\cdots\circ[z\circ x_n\circ z^{-1}, z\circ y_n\circ z^{-1}]$$

As H_2 is closed under composition, it is enough to prove $y \in D(H)$ that $[z \circ x_i \circ z^{-1}, z \circ y_i \circ z^{-1}] \in D(H)$, for $i = 1, \ldots, n$. The problem reduces to prove that both $z \circ x_i \circ z^{-1}$ and $z \circ y_i \circ z^{-1}$ belong to H for $i = 1, \ldots, n$. This follows from the subgroup axioms.□

We can apply Proposition 25 to prove Theorem 24

Theorem 24 1 (Characterisation of solvability) *A group G is solvable if and only if $D^n(G) = \{e\}$ for some natural number n.*

We are concerned with proving the unsolvability of a certain group. We present a poset-theoretical criterion of unsolvability. Let us assume again the hypotheses of Proposition 25. Let a_0 be an element of A such that $\perp < a_0 \leq f a_0$.

Lemma 27 *Under the assumptions above, Statement 1 (or equivalently Statement 2) does not hold.*

Proof: We prove by induction that for every n, we have $a_0 \leq f^n \top$.□

We deduce the following criterion of unsolvability.

Lemma 28 *Let G be a group. If there exists a subset H of G satisfying $H \subseteq D(H)$ and $\{e\}$ is strictly contained in H, then G is unsolvable.*

3-Cycles

Definition 29 *Let A be a discrete set and x, y, z be three distinct elements of A. Then the* **3-cycle** *generated by x, y, z is written $(x\ y\ z)$ and defined by:*

Formally we define 3-cycles by using if ... then ... else ... So we are first forced to check that the formal definition is actually the intended one. The main theorem to prove with 3-cycles is the following equality:

Proposition 30 *Let A be a discrete set and a_1, a_2, a_3, a_4, a_5 be five distinct elements of A. Then $(a1\ a2\ a3) = [(a1\ a5\ a3), (a1\ a4\ a2)]$.*

Proving the equality formally requires lots of intermediate lemmas to be proved and intermediate computations to be done. The process is very basic: two maps are equal if they are extensionally equal, i.e. produce the same value when applied to an element of their domain. Thus we have to prove that for every element x of A, we have $(a1\ a2\ a3)x = [(a1\ a5\ a3), (a1\ a4\ a2)]x$. The equality is proved by cases: $x = a_i$ for some $i = 1, \ldots, 5$ or $x \neq a_i$, for every $i = 1, \ldots, 5$.

The proof of Theorem 1 Given a discrete set A and five distinct elements a_1, a_2, a_3, a_4, a_5, consider the set P of 3-cycles generated by any three of these elements. P is a subset of the symmetric group over A.

Proposition 31 $P \subseteq D(P)$.

One surprising detour of the proof is that the following statement has to be proved: given any three pairwise distinct elements of $\{a_1, a_2, a_3, a_4, a_5\}$, there are two more elements such that the five of them are pairwise distinct. It is absolutely trivial but the proof is tedious. Once this is proved, the proposition follows.

Corollary 32 *If A is a discrete set with five or more elements, then the symmetric group over A is unsolvable.*

The result can be applied, for $n \geq 5$, to S_n which is defined to be the symmetric group over the set $\{1, \ldots, n\}$.

5 Alternative proofs and related results

5.1 Alternative proofs

We proved the unsolvability of S_n for $n \geq 5$ by by-passing several results which would have been difficult to prove in Lego. In fact, the proof we have followed differs slightly from the standard textbooks ([15],[8]). Moreover, we believe it would be extremely difficult to formalise a proof of Theorem 1 following any of the standard textbooks: they use several results whose implementation in a proof-checker would be problematic. In particular, proving the simplicity of the alternating group A_n for $n \geq 5$ requires a lot of insight on the structure of S_n and would be rather difficult to prove in Lego. We mention some of the key steps that we believe to be challenging:

1. to prove that every permutation can be decomposed as a product of two cycles. The proof could be divided into two parts: first, prove that every permutation can be written as a product of disjoint cycles. Second, prove that every cycle can be written as a product of 2-cycles (required to define A_n as the subset of permutations that can be written as an even product of permutations).
2. to prove that A_n is closed under inverses (required to establish that A_n is a subgroup of S_n).
3. to prove that A_n is generated by 3-cycles,
4. to prove that every normal subgroup of A_n containing one 3-cycle contains all 3-cycles.

Further, Stewart's proof uses a combinatorial argument, namely that A_n is of order $n!/2$, which is not prime for $n \geq 5$. Proving that the cardinal of A_n is equal to $n!/2$ is likely to be a difficult puzzle in Lego.

5.2 Related results and future work

Several algebraic results can be formalised relatively easily on top of our proof. The results mainly concern solvable and nilpotent groups. However, it would be more interesting, in the context of the GALOIS project, to focus on actions of groups over a set. We would like to prove the class equation and eventually Sylow's theorem[3]. Some preliminary work is needed: in particular, the notions of prime number, greatest common divisor, finite set and cardinal will have to be implemented. Some basic group theory, such as the Isomorphisms Theorems, has to be formalised as well prior to proving Sylow's theorem.

6 Conclusion

The work presented here was originally thought to be a matter of few weeks. It turned out to involve a lot more work than expected. However, most of the effort has been put in producing a reasonably organised proof. We have tried to present the material in a natural way, close to informal style: for example, we gave a construction of the submonoid generated by a subset of a monoid, while this construction was only needed for groups. The ability to follow such a style is largely relying on the simulation of classes we have been using. It would have been very unaesthetic and tedious to use inheritance manually, i.e. by making explicit every time it is needed the forgetful functors from groups to monoids or monoids to sets. However, our simulation of classes is of limited interest, as it is quite slow and does not support multiple inheritance. We believe that proof-checkers supporting multiple inheritance are a very promising perspective; if multiple inheritance were implemented in the current proof-checkers, the task of formalising algebra would be greatly simplified.

A fair amount of the code is devoted to prove trivial lemmas at the level of propositional logic or about existential quantification. We think it is a rather uninteresting part of the work and it should be avoided in the future. There are two solutions to avoid such low-levels details: use a vast library of low-level details or combine proof-checkers with theorem-provers. One of the superiorities of proof-checkers versus theorem-provers lie in their ability to adhere to informal reasoning. Such an advantage is certainly minimized if the process of formalising a proof in a proof-checker involves a considerable amount of work on trivial problems. We believe that one should not be too dogmatic about formal verification and allow trivial problems to be solved automatically by the computer.

Another source of great difficulty is equational reasoning. Proving that every 3-cycles can be written as a commutator of 3-cycles is nothing but proving the equality of two entities. It turned out to be a tedious task. One could hope that future proof-checkers will have some ability to do equational reasoning. There are several systems in which equational reasoning is efficiently automated which could hopefully be combined with proof-checkers.

[3] Sylow's theorem is used to prove that the field C of complex numbers is algebraically closed.

Worse, simple combinatorics problems involving natural numbers are more than often hopeless for being formalised. Natural numbers are introduced as an inductive type and calculations involving natural numbers have to be done manually. A very simple solution to the problem would be to integrate as a primitive the type of natural numbers and support it with computational power. Moreover, it would be handy to be able to write 5 instead of $s(s(s(s(s0))))$.

Our overall feeling is that a considerable amount of energy has to be spent on trivial matters. However, proof-checkers are really efficient as far as abstract reasoning and constructions are concerned. Constructing the category of isomorphisms of a category or proving Proposition 25 were straightforward. High-level reasoning in a proof-checker is surprisingly easy and close to informal mathematical practice. We believe that formalising mathematics interactively with a proof-checker is very instructive about mathematics themselves, as it forces us to have a better understanding of mathematical proofs. On the contrary, a theorem-prover could hinder the underlying mathematical reasoning by using resolution techniques remote from the usual mathematical line of thinking. Indeed, proof-checkers seem to have, out of the different families of computer systems designed for the mechanisation of mathematics, the greatest ability to capture mathematical reasoning and respect its spirit. However, they lack some of the qualities of its concurrents. A very promising area of reasearch that has just been scratched is the design of a computer system with reasoning, algorithmic and calculating abilities. One could imagine a system combining the existing families of systems: proof-checkers, theorem-provers, computer algebra systems and systems for equational reasoning and rewriting. Ideally, the user of such a system would be free to choose at every moment which component of the system should be given preference. From our point of view, we are interested in formal proofs that adhere to the style of informal reasoning, so we would presumably desire the proof-checker component of the system to be preponderant. The other component would relieve the user from the burden of proving a great amount of trivial lemmas. Of course, everyone would still have to decide for himself what are the essential parts of a theory and what are its trivial results.

Acknowledgements

Most of this work was performed in collaboration with Peter Aczel at Manchester University during the year 1992-1993 with the support of the BRA 'Types' project.

References

1. P.Aczel. *Galois: a theory development project*, manuscript, University of Manchester, January 1993.
2. P.Aczel and G.Barthe. *A notion of class for type theory*, note, University of Manchester, August 1993.

3. T.Altenkirch. *A formalisation of the strong normalisation proof for system F in Lego*, in the Proceedings of Typed Lambda Calculi and applications, LNCS 664, March 1993.
4. H.Barendregt. *Typed λ-calculi*, Handbook of logic in computer science, Abramsky and *al* eds, OUP 1992.
5. G.Barthe. *Formalising Galois theory*, manuscript, University of Manchester, May 1993.
6. G.Barthe. *Mathematical concepts in type theory*, manuscript, University of Nijmegen, March 1994.
7. G.Barthe. *A note on the simulation of classes in Lego*, in preparation.
8. T.Hungerford. *Algebra*, Graduate texts in mathematics 73, Springer-Verlag, 1974.
9. C.Jones. *Completing the rationals and metric spaces in Lego*, in Proceedings of the Second Workshop on Logical Frameworks, Huet and Plotkin Eds, May 1991.
10. J.Mac Kinna and R.Pollack *Pure Type Systems Formalised*, in the Proceedings of Typed Lambda Calculi and applications, LNCS 664, March 1993.
11. Z.Luo. *ECC: an Extended Calculus of Constructions*, Ph.D thesis, University of Edinburgh, July 1990.
12. Z.Luo. *Program Specification and Data Refinement in Type Theory*, Mathematical Foundations of Computer Science, 1993.
13. Z.Luo and R.Pollack. *LEGO Proof Development System: User's manual*, Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
14. R.Pollack. *Implicit Syntax*, in the electronic proceedings of the Workshop on Logical Frameworks, May 1990.
15. I.Stewart. *Galois theory*, Chapman and Hall, 1973.