Galois: A Theory Development Project*

Peter Aczel
Departments of Computer Science and Mathematics
Manchester University, Manchester M13 9PL, U.K.

June 12, 1995

A report on work in progress, for the Turin meeting on the Representation of Mathematics in Logical frameworks, January 20-23, 1993

1 The aims of the project

This is intended to be a large scale, ambitious and perhaps collaborative project to develop a significant body of machine checked mathematics. The primary aim of the project is to produce enough algebra to cover Galois Theory and some of its applications, such as the unsolvability of the general polynomial of any degree greater than four.

In order to develop Galois Theory it will be necessary to define and develop, to some extent, several algebraic theories and then use combinations of the theories to create Galois Theory itself. In particular there will need to be chapters on Group Theory, Linear Algebra, Polynomial Rings, Field Extensions, etc.

One advantage in choosing Galois Theory is that there is a significant amount of algorithmic work to be done in applications of the theory, and it would be interesting to explore the possibilities of combining this theory and proof development work with the use of computer algebra systems.[1]

A reason to choose an ambitious project is so as to try to confront some of the issues of large scale: documentation and collaboration. We do not expect to be able to complete the project in just a few months on our own. Nevertheless I feel that even partial work on the project is worthwhile.

2 Present Status of the project

This project has only been under way for 2-3 months so that we have only had time to make a limited amount of progress. Gilles Barthe, an RA working on the Types project in Manchester since October, has had no previous practical experience with any proof development system. We decided to use Lego for the present, as this is the system I am reasonably familiar with. Rather than plunge into Galois Theory right away, Gilles has

*This work is supported by the ESPRIT Basic Research Action on Types and Proofs

[1]Actually, at present I am completely unfamiliar with the use of such systems, so that this idea is just a vague one.

been working on the key group theoretic result needed to apply the general theory to the unsolvability of polynomials. This result states that the group of permutations on any set, with five elements or more, is an unsolvable group. We initially thought that this would involve about a week's work for Gilles. In fact the work has taken several weeks. It is now completed, in some sense, though there is probably plenty of room to improve the organisation of the proof. This work will be presented in a separate report by Gilles.

I, myself, have had limited time to work on the project. Over Christmas I decided that in the longer term it was necessary to develop first a general theory of sets and then go on to develop the notion of a group and a proof that the set of permutations of any set form a group. This is what I have done. A lego file forms the appendix to this report. My work is intended, eventually, to form some of the initial parts of the work of Gilles. But, at present, our two lego developments are independent of each other.

By a set I mean a pair consisting of a type and an equivalence relation on that type. With this notion of set one can develop the notion of a category (the hom-sets being sets in this sense) and then show that the sets form a category. Note that a map in this category is not actually a function of the type theory, but a pair consisting of a function, together with a proof that the function is extensional with respect to the equivalence relations on the domain and codomain sets of the map. The work involved in showing that sets form a category seems to me to be an essential starting point for a chapter on sets that should be the basis for the treatment of algebra. It needs to be tackled in a thorough, non-piecemeal way, so as to create a fundamental tool that can be generally useful to the Lego community and presumably also potentially useful to the users of other computer sytems within the Types community.

3 A brief review of Galois Theory

My intention here is simply to put in the context of the overall Galois project the work carried out by Gilles. A version of the main result of Galois Theory may be expressed as follows:

Theorem 1 *Let p be a separable polynomial over a field K and let $L \supseteq K$ be a splitting field of p over K. Let $\mathcal{F}$ be the set of all fields Fsuch that $L \supseteq F \supseteq K$. Let $\mathcal{U}$ be the set of all subgroups of the group $Aut_K(L)$ of those field automorphisms of L that fix each element of K. Both the sets $\mathcal{F}$ and $\mathcal{U}$ are partially ordered by the subset relations on them. Let $\theta : \mathcal{F} \to \mathcal{U}$ be given by*

$$\theta(F) = Aut_F(L) \text{ for all } F \in \mathcal{F}.$$

Then θ is an isomorphism of the poset $\mathcal{F}$ with the dual poset $\mathcal{U}^{op}$, whose inverse is given by

$$\theta^{-1}(H) = \{x \in L \mid \forall g \in H\ g(x) = x\}$$

for all $H \in \mathcal{U}$.

The application of this result to the unsolvability of polynomials uses the following two results.

Theorem 2 *Let p be a separable polynomial over a field K and let $L \supseteq K$ be a splitting field of p over K. The polynomial p is solvable by radicals over K if and only if the group $Aut_K(L)$ is a solvable group.*

Theorem 3 *The group of permutations on any set, with five elements or more, is unsolvable.*

Appendix: The lego file "algebra.l"

This appendix is essentially an edited transcript of a lego file. There are three sections:

1. The Category of sets
2. The types of monoids and groups
3. The group of permutations of a set

In the last section I construct the group of permutations on a set as the group of inverse pairs of the monoid of endomorphisms of the set.

The file is initialised with the type system XCC (Extended Calculus of Constructions) and the command `Logic` inputs the standard Lego file of Logical definitions.

```
Init XCC; Logic;
```

1 THE CATEGORY OF SETS

1.1 The type of sets

We define a set to be a triple consisting of a type, an equivalence relation on that type and a proof that it is indeed an equivalence relation. We define functions `el, eq, pr` so that for any set `A` the three components of the set are `A.el, A.eq, A.pr`. The proof `A.pr` that `A.eq` is an equivalence relation is split into the three proofs `A.pr_refl, A.pr_symm`, `A.pr_tran` that `A.eq` is reflexive, symmetric and transitive, respectively.

```
[Rel[T:Type] = T -> T -> Prop];
[T|Type][R:Rel T];
[Refl = {x:T} R x x];
[Symm = {x,y|T} (R x y) -> (R y x)];
[Tran = {y,x,z|T} (R x y) -> (R y z) -> (R x z)];
[Equiv_rel = and3 Refl  Symm Tran];
Discharge T;

[Set = <T:Type><R:Rel T> Equiv_rel R];


[el[A:Set] = A.1];
[A|Set];
[eq = A.2.1];
[pr = A.2.2];
[pr_refl = pr.and3_out1];
[pr_symm = pr.and3_out2];
[pr_tran = pr.and3_out3];
Discharge A;
```

1.2 The set of maps between two sets

Given sets `A, B` we define the set `(Map_set A B)` of maps from `A` to `B`. Each map has to preserve the equivalence relations and the equality relation between maps is defined extensionally. If `f:(Map_set A B).el` then `f.ap : A.el -> B.el` and `f.pres` is a proof that `f.ap` preserves equivalence relations.

```
[Map_law[A,B|Set][f:A.el -> B.el] =
         {x,y:A.el} (x.eq y) ->  (f x).eq (f y)];
[Map[A,B:Set] = <f:A.el -> B.el>Map_law f];

[A,B|Set][f:Map A B];
[ap = f.1];
[pres = f.2];
Discharge A;

[Map_eq[A,B|Set][f,g:Map A B] = {x:A.el} (f.ap x).eq (g.ap x)];
```

Equality between maps is an equivalence relation

```
[A,B:Set];
Goal Equiv_rel (Map_eq|A|B);
    Refine pair3; Intros; Refine pr_refl;
    Intros; Refine pr_symm; Refine H;
    Intros; Refine pr_tran;
    Refine y.1 x1; Refine H; Refine H1;
Save equiv_map_eq;

[Map_set = ((Map A B),(Map_eq|A|B),equiv_map_eq):Set];
Discharge A;

[ap2[A,B,C|Set] = [M=Map_set][f:(M A (M B C)).el][a:A.el] ap (ap f a)];
```

1.3 The type of categories

We define a category to be a tuple with five components, a type `Ob` of objects, a homset map `H:Ob -> Ob -> Set`, a composition operation `o:Comp_Type`, an identity operation `i:Id_Type` and a proof of the conjunction of the associative law for the composition and the identity law for the identity with resepect to the composition.

```
[Ob:Type][H:Ob -> Ob -> Set]
$[M=Map_set]

[Comp_Type = {a,b,c|Ob} (M (H a b) (M (H b c) (H a c))).el];
[Id_Type = {a|Ob}((H a a).el)];
```

```
[o:Comp_Type];
$[O[a,b,c|Ob] = (o|a|b|c).ap2];

[Assoc_law = {a,b,c,d:Ob} {f:(H a b).el} {g:(H b c).el} {h:(H c d).el}

                    (f.O (g.O h)).eq ((f.O g).O h)
];

[Id_law[i:Id_Type] = {a,b:Ob} and ([X= H a b]{f:X.el}((i|a).O f).eq f)
                                  ([X= H b a]{f:X.el}(f.O (i|a)).eq f)
];
Discharge Ob;

[Category = <Ob:Type>
            <H:Ob -> Ob -> Set>
            <o:Comp_Type Ob H>
            <i:Id_Type Ob H>
                (Assoc_law Ob H o).and (Id_law Ob H o i)
];
```

1.4 The category of sets

We define the composition operation `comp`, for the category of sets, implicitly; i.e. we exploit the refinement proof mechanism of Lego to obtain the definition. This is sometimes a useful technique, but may be dangerous if one does not go on to check that the object defined is the expected one, rather than some alternative of the same type. In the present case we ought to prove the goal

```
{A,B,C|Set}{f:Map_set A B}{g:Map_set B C}

              ((comp f g).ap x).eq (f.ap (g.ap x))
```

But the first line of the proof below, defining `comp`, should indicate that `comp` has been correctly defined.

Having defined `comp` we go on to prove the associative law for it, define the identity operation, `id` for the category of sets and prove the identity law for it. We are then ready to define the category of sets.

```
Goal Comp_Type Set Map_set;
    Intros; Refine H1.ap (H.ap H2);
    Intros; Refine pres; Refine pres; Immed;
    Intros; Refine H1;
    Intros; Refine x1.2; Refine H;
Save comp;
```

```
Goal Assoc_law Set Map_set comp;
    Intros; Refine pr_refl;
Save assoc_law;

[id[A|Set] = (([x:A.el]x) , [x,y:A.el][u:eq x y]u) :Map A A];

Goal Id_law Set Map_set comp id;
    Intros a b; Refine pair;
    Intros; Refine pr_refl;
    Intros; Refine pr_refl;
Save id_law;

[SET = (Set,Map_set,comp,id,(pair assoc_law id_law)) :Category];
```

2 THE TYPES OF MONOIDS AND GROUPS

2.1 The type of monoids

If `A:Set` then we define `Bin_op A` to be the type of curried binary operations on `A`. We define a monoid to be a four-tuple with the components a set, a binary operation on that set, an element of the set and a proof of the conjunction of the two monoid laws.

We end with some useful definitions for monoids. In particular `cong_l` and `cong_r` are useful for equational reasoning with monoids. They express that the binary operation is a congruence with respect to its left and right arguments.

```
[Bin_op[A:Set] = Map_set A (Map_set A A)];

[A|Set][op:(Bin_op A).el];

$[c=[x:A.el] x.(op.ap).ap]
```

Associative law

```
[Monoid_law1 = {x,y,z:A.el} ((x.c y).c z).eq (x.c (y.c z))];
```

Unit law

```
[Monoid_law2[i:A.el] = {x:A.el} ((x.c i).eq x).and ((i.c x).eq x)];

Discharge A;

[Monoid = <A:Set><op:(Bin_op A).el><i:A.el>

             (Monoid_law1 op).and (Monoid_law2 op i)
];
```

```
[set[A:Monoid] = A.1:Set];
[A|Monoid];
$[A'=A.set];
[op   = A.2.1:(Bin_op A').el];
[unit = A.2.2.1:A'.el];
[monoid_law1 = A.2.2.2.fst];
[monoid_law2 = A.2.2.2.snd];
[cong_l[x, y, u:A'.el] = [z:x.eq y] (op.pres x y z u)];
[cong_r[x, y, u:A'.el] = [z:x.eq y] ((op.ap u).pres x y z)];
Discharge A;
```

2.2 The type of groups

We define a group to be a triple consisting of a monoid, a unary operation `inv` on the underlying set and a proof that `inv` satisfies the inverse law. The auxiliary relation `Inverses_rel` will be useful later.

```
[A|Monoid];
$[c=(op|A).ap2];
$[e=unit|A];
[Inverses_rel[a,a':A.set.el] = (((a.c a').eq e).and ((a'.c a).eq e))];

[Inverse_law = [B=A.set][inv:Map B B]
                         {a:B.el} Inverses_rel a a.(inv.ap)];
Discharge A;

[Group = <A:Monoid><inv:Map A.set A.set> Inverse_law inv];
```

3 THE GROUP OF PERMUTATIONS OF A SET

Given a set `X` we want to define the group, `PermGroup X`, of permutations of `X`. We have analysed this into two constructions. First, in 3.1, we define the monoid, `Endo_monoid X`. This is easy. Then, given a monoid `A` we define the group, `Inverses_group A`, of inverse pairs of `A`; i.e. pairs of elements of `A` together with a proof that the two elements are inverses of each other. Then we define

```
PermGroup[X:Set] = (Inverses_group (Endo_monoid X))
```

3.1 The monoid of Endomorphisms on a set

`Endo_monoid` is defined implicitly, again exploiting the refinement proof mechanism of Lego. By examining the first three lines of the proof we can see that we have defined the intended monoid.

```
[X:Set];
Goal Monoid;
    Intros #; Refine (Map_set X X);
    Intros # ; Refine comp;
    Intros #; Refine (id|X);
    Refine pair; Refine assoc_law;
    Intros x; Refine pair;
    Refine (id_law X X).fst; Refine (id_law X X).snd;
Save Endo_monoid;

Discharge X;
```

3.2 The monoid of inverse pairs on a monoid

```
[A:Monoid];
$[t = pr_tran|A.set];
```

The set, `Inverses_set`, of inverse pairs on `A`

```
[Inverses = <a,a': A.set.el> Inverses_rel a a'];

[Inverses_eq[a,a':Inverses] = a.1.eq a'.1];

Goal Equiv_rel Inverses_eq;
    Refine pair3;
    Intros _; Refine pr_refl;
    Intros __; Refine pr_symm;
    Intros ___; Refine t|y.1;
Save Inverses_equiv;

[Inverses_set = (Inverses,Inverses_eq, Inverses_equiv):Set];
```

Composition operation on `Inverses_set`

```
$[c = (op|A).ap2]$[e = unit|A];
$[g[a,b:Inverses_set.el] = ((a.1.c b.1), (b.2.1.c a.2.1))];
$[f = [x:Inverses] ((x.2.1 , x.1 , (pair x.2.2.snd x.2.2.fst)):Inverses)];
Goal {a,b:Inverses_set.el}[gab = g a b] (gab.1.c gab.2).eq e;
    Intros a b;
    Refine t|(a.1.c (b.1.c (c b.2.1 a.2.1)));
    Refine monoid_law1|A a.1 b.1 (b.2.1.c a.2.1);
    Refine t|(a.1.c a.2.1);
    Refine  cong_r;
    Refine t|((b.1.c b.2.1).c a.2.1);
```

```
    Refine pr_symm;
    Refine monoid_law1|A b.1 b.2.1 a.2.1;
    Refine t|(e.c a.2.1);
    Refine cong_l;
    Refine b.2.2.fst;
    Refine (monoid_law2|A a.2.1).snd;
    Refine a.2.2.fst;
Save gab_lemma;


Goal (Bin_op Inverses_set).el;
    Intros # a # b;
    Intros; Refine ((a.1).c b.1);
    Intros; Refine ((b.2.1).c a.2.1);
    Refine pair;
    Refine gab_lemma a b;
    Refine gab_lemma (f b) (f a);
    Intros x y;
    Intros; Refine cong_r; Immed;
    Intros; Refine cong_l; Immed;
Save Inverses_op;
```

The monoid of inverse pairs

```
Goal Monoid_law1|Inverses_set Inverses_op;
    Intros; Refine monoid_law1|A;
Save Inverses_ml1;

Goal Inverses_rel e e;
    Refine pair;
    Refine ?4;
    Refine (monoid_law2|A e).fst;
Save Inverses_unit_3;

[Inverses_unit = (e, e, Inverses_unit_3) :Inverses];

Goal Monoid_law2|Inverses_set Inverses_op Inverses_unit;
    Intros x; Refine monoid_law2|A;
Save Inverses_ml2;

[Inverses_monoid = (Inverses_set, Inverses_op, Inverses_unit,

                    (pair Inverses_ml1 Inverses_ml2) ) : Monoid];
```

3.3 The group of inverse pairs on a monoid

```
Goal Map_law|Inverses_set|Inverses_set f;
   Intros;
   Refine t|(y.2.1.c e);
   Refine +1 (monoid_law2|A y.2.1).fst;
   Refine t|(y.2.1.c (x.1.c x.2.1));
   Refine +1 cong_r;
   Refine +1 x.2.2.fst;
   Refine t|((y.2.1.c x.1).c x.2.1);
   Refine +1 monoid_law1|A;
   Refine t|((y.2.1.c y.1).c x.2.1);
   Refine +1 cong_l;
   Refine +1 cong_r;
   Refine +1 pr_symm;
   Refine +1 H;
   Refine t|(e.c x.2.1);
   Refine +1 cong_l;
   Refine +1 pr_symm;
   Refine +1 y.2.2.snd;
   Refine pr_symm;
   Refine (monoid_law2|A x.2.1).snd;
Save mlf;

[Inverses_inv = (f,mlf):Map Inverses_set Inverses_set];

Goal Inverse_law|Inverses_monoid Inverses_inv;
    Intros a; Refine pair;
    Refine a.2.2.fst;
    Refine a.2.2.snd;
Save inv_proof;

[Inverses_group = (Inverses_monoid , Inverses_inv , inv_proof):Group];
Discharge A;
```

3.4 The Group of permutations on a set

```
[PermGroup[X:Set] = (Inverses_group (Endo_monoid X)):Group];
```