A symbolic approach to abstract algebra in HOL Light

Marco Maggesi**[1]

DiMaI - Dipartimento di Matematica e Informatica "U. Dini"
Università degli Studi Firenze, Italy
http://www.math.unifi.it/~maggesi/

Abstract. Formalising algebraic structures (groups, rings, fields, vector spaces, lattices, ...) is known to be challenging task which is often undertaken by exploiting various kind of extra-logical mechanisms (axiomatic classes, modules, locales, coercions, ...) provided by most modern theorem provers.
We want to explore an alternative strategy, where algebraic structures are implemented via a deep embedding of mathematical formulas and are managed as first class objects in the HOL theory.
Along the way, we provide a mechanism of *generalised conversions*, which extends Paulson's conversions by allowing to compute with equivalence relations. We developed generalised conversions to support rewriting in our system, but they can be used independently and may have an interest in its own.

Keywords: Mechanisation of mathematics, Algebraic Structures, HOL Light

1 Introduction

We aim to implement a large class of algebraic structures in HOL Light: monoids, groups, rings, fields, vector spaces, lattices, etc. These structures are typically presented by an abstract syntax (the allowed formulae of the theory) and a set of axioms (the symbolic manipulation permitted).

For instance, a group has a unit 'e', a multiplication '$x \cdot y$' and an inverse 'x^{-1}' obeying to the identities

- associatitvity: $(x \cdot y) \cdot z = (x \cdot y) \cdot z$;
- left identity: $e \cdot x = x$;
- left inverse: $x^{-1} \cdot x = e$.

Such description is essentially symbolical. We thus pursuit the idea of implementing, via a deep embedding, a symbolic system in HOL Light, which can be used to formalise algebraic structures. At a first sight, our framework will

** The author was supported by MIUR and GNSAGA-INdAM.

resemble the implementation of a Computer Algebra System (CAS) in the "programming language" HOL.

One key point of our approach is that it allows to encode and administer a hierarchy of algebraic structures without the need of extra-logical mechanisms, like axiomatic classes, modules, locales, coercions, etc.

We present a prototype implementation in its early stage of development. Various implementation choices have to be explored, tested and discussed. We argue that our methodology is sound and adeguate in principle. Whether this proof of concept can evolve into a workable approach in practice is the object of the present research.

The source code of our prototype can be obtained from the following repository: `https://bitbucket.org/maggesi/symbolic` .

2 Abstract terms

The starting point of our work is the definition of a type for representing symbolic expressions in HOL. We introduce the new type `‘:aterm‘` (think "aterm" as mnemonic for *abstract* terms) which is just the inductive type of the so called *ground terms*:

```
let aterm_INDUCT,aterm_RECUR = define_type
  "aterm = Symb string
         | App aterm aterm";;
```

Hence an aterm can be either a *symbol* `‘Symb s‘` or an *application* of two aterms `‘App f x‘`. The latter intends to denote a functional application $f(x)$.

We can consider several possible extensions and modifications to our type of abstract terms (a separate constructor for variables, polymorphic constants, binding constructions). However, we perceive ground terms as a kind of God-given abstract syntax which can conveniently encode any formula written on paper. So we will stick to this simple design choice for now.

We extended HOL Light by implementing a custom parser and printer for aterms, in order to allow the user to work with a more traditional concrete syntax. We use the notation `#( ... )` to quote and antiquote aterms. For instance, to represent the mathematical expression $x + y$, we will use the abstract term

```
‘App (App (Symb "+") (Symb "x")) (Symb "y")‘
```

which is represented in HOL Light with the notation `‘#(x + y)‘`. When a subterm of an aterm cannot be represented with the concrete syntax, the hash notation is used to antiquote the term. For instance, in the term

```
‘App (App (Symb "=") x) (Symb y)‘
```

the subterms x and y are HOL variables, not aterm symbols, thus they are preceded by the hash character in the concrete syntax:

```
‘#(#x = #y)‘
```

3 Generalized conversions

In the Higher-Order Logic, equality has a prominent role from its very foundation and HOL systems provide a rich set of tools to reason with equations. However, it is often the case that equivalence relations have to be used instead of plain equality, for which we have no specific support for rewriting in HOL Light.

In our settings, we are led to consider several kind of equivalence relations between aterms. We will give an example based on group theory in the next section. We thus develop our own mechanism which allows to perform rewriting with equivalence relations. As for the standard rewriting system, the key idea is the notion of *conversion*, introduced by Paulson in LCF. We propose a notion of *generalised conversions* (*gconvs* for short) which extends Paulson's conversions to work with equivalence relations. Our basic observation is that, given an equivalence relation `‘R‘` we have `‘R x y <=> R x = R y‘`, that is, two elements are equivalent if and only if they have the same associated class. Thus a generalised conversion is an ML procedure that for all term `‘R t‘` produces a theorem `|- R t = R t'` or fails. As for standard conversions, we define various conversionals (i.e., higher-order conversions) which allows to build complex conversions from basic building blocks.

We use the suffix `_GCONV` for generalised conversions and the prefix `G` in other cases. So, for instance, we have a conversion `REWRITE_GCONV` and a tactic `GREWRITE_TAC` that generalises the corresponding classical variants `REWRITE_CONV` and `REWRITE_TAC`.

4 A basic example: Group Theory

As a basic example, we illustrate a simple formalisation of group theory in HOL Light using our framework.

All the theory is encoded using a single judgment `‘GROUP #(`$x = y$`)‘` which states that x and y are equivalent 'expressions' (i.e., aterms) in group theory. Please notice the use of the hash notation to denote that the argument of `‘GROUP‘` is an aterm. In particular, we stress that the equality symbol there does not denote the standard equality of HOL, but a *judgmental equality* within our *object theory* of groups. (A more realistic example would also include a judgement for 'membership' or 'wellformedness', but we omit it here for simplicity.)

The inductive predicate `‘GROUP:aterm->bool‘` is inductively defined as follows:

```
let GROUP_RULES, GROUP_INDUCT, GROUP_CASES = new_inductive_definition
  ‘(!x. GROUP #(#x = #x)) /\
   (!x y. GROUP #(#x = #y) ==> GROUP #(#y = #x)) /\
   (!x y z. GROUP #(#x = #y) /\ GROUP #(#y = #z)
            ==> GROUP #(#x = #z)) /\
   (!x x' y y'. GROUP #(#x = #x') /\ GROUP #(#y = #y')
                ==> GROUP #(#x * #y = #x' * #y')) /\
   (!x x'. GROUP #(#x = #x') ==> GROUP #(inv #x = inv #x')) /\
```

```
  (!x. GROUP #(e * #x = #x)) /\
  (!x. GROUP #(inv #x * #x = e)) /\
  (!x y z. GROUP #(#x * (#y * #z) = (#x * #y) * #z))‘;;
```

The inductive definition is composed by eight rules. The first three of them are the reflexivity, symmetry and transitivity property of our judgemental equality. Next there are three rules that assert that multiplication and inversion of the group are compatible with equality. Finally, we have the three usual axioms for group theory: identity on left, inverse on left and associativity of the product.

By using our mechanism of generalised conversions, illustrated in the previous section, it is immediate to derive from the above axioms several other simple theorems and identities of the theory. For instance, the following basic computation $(x \cdot y^{-1}) \cdot y = x \cdot (y^{-1} \cdot y) = x \cdot e = x$ can be performed in one shot using `GREWRITE_TAC` as follows:

```
let GROUP_MUL_LINV = prove
 (‘!x y. GROUP #((#x * inv #y) * #y = #x)‘,
  GREWRITE_TAC[GROUP_REVASSOC; GROUP_LINV; GROUP_RID]);;
```

Finally, we define a HOL function `GROUP_SIMP` for putting a group expression in normal form. E.g., the expression $(x^{-1} \cdot e) \cdot y^{-1} \cdot (y \cdot x)$ can be simplified with the command

```
# GROUP_SIMP_CONV ‘GROUP_SIMP #((inv x * e) * inv y * (y * x))‘;;
val it : thm = |- GROUP_SIMP #((inv x * e) * inv y * y * x) = #(e)
```

We also prove the soundness of the `GROUP_SIMP` function

```
|- !x. GROUP #(#(GROUP_SIMP x) = #x)
```

thus providing another tool for automatically certify identities in group theory.

5 Conclusions

We presented a proof of concept for reasoning about ‘abstract’ algebraic formulas in HOL Light. The project is in its early stage of development and lacks essential functionalities. Nevertheless, our system it is already capable of supporting basic examples of algebraic theories, like reasoning about identities in group theory, in a completely certified way. We consider this as a preliminary step toward a general framework for formalising algebraic structures in HOL Light.