Noname manuscript No.
(will be inserted by the editor)

Modelling Algebraic Structures and Morphisms in ACL2

Jónathan Heras · Francisco Jesús Martín–Mateos · Vico Pascual

Received: date / Accepted: date

Abstract In this paper, we present how algebraic structures and morphisms can be modelled in the ACL2 theorem prover. Namely, we illustrate a methodology for implementing a set of tools that facilitates the formalisations related to algebraic structures — as a result, an algebraic hierarchy ranging from setoids to vector spaces has been developed. The resultant tools can be used to simplify the development of generic theories about algebraic structures. In particular, the benefits of using the tools presented in this paper, compared to a from-scratch approach, are especially relevant when working with complex mathematical structures; for example, the structures employed in Algebraic Topology. This work shows that ACL2 can be a suitable tool for formalising algebraic concepts coming, for instance, from computer algebra systems.

Keywords Mathematical structures, ACL2, Algebraic hierarchy, Proof engineering, Computer Algebra systems, Formal verification.

This work was partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

J. Heras (corresponding author) and V. Pascual
Department of Mathematics and Computer Science, University of La Rioja, Edificio Vives, Luis de Ulloa, s/n. 26004 Logroño, Spain.
Tel.: (+34) 941299461
Fax: +34 941299460
E-mail: jonathan.heras@unirioja.es, vico.pascual@unirioja.es

F.J. Martín–Mateos
Computational Logic Group, Dept. of Computer Science and Artificial Intelligence, University of Seville, E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain.
E-mail: fjesus@us.es

1 Introduction

Computer Algebra Systems (CAS) are powerful tools used, almost on a daily basis, by researchers in different areas. The performance of these systems keeps improving thanks to the introduction of new algorithms and the use of more efficient structures. The correctness of these new algorithms and structures is usually provided by theoretical means; however, bugs can be introduced during their implementation — e.g. a bug in the computation of some determinants with big integers was found in Mathematica [22].

Interactive Theorem Provers (ITPs) have been broadly employed to increase the confidence in CAS using different approaches; some examples are: the formalisation of CAS algorithms [5,18,53], the creation of environments to develop certified programs for symbolic computation [55], the implementation of ITPs on top of CAS and vice versa [11,40], and the communication between CAS and ITPs to ensure the correctness of some computations [1,10,16,31].

Consider the user of an ITP who is interested in constructing mechanically checked proofs about CAS. The user might start by using a from-scratch approach but would soon realise that the "same" theorems and sequences of steps show up over and over again. At that point, the user has a choice: capture the commonality in higher-order definitions and theorems, or build programming tools to reduce the user input.

Users of ACL2 [41,43], a first order system for reasoning about Common Lisp programs, tend to program their way out of complexity. This paper illustrates how ACL2's strengths can be used to overcome ACL2's limited support for higher order reasoning. The strengths in question are the Common Lisp programming language, macros — which permit Lisp functions to create commands that are then executed — and a powerful automatic theorem prover. Namely, we illustrate these strengths in an instrumental component in most CAS: *algebraic structures*.

Algebraic structures are the basis for several constructions, and, therefore, it makes sense to use ITPs to certify the correctness of the implementation of those structures. The implementation of algebraic structures in ITPs is a well-known problem; and most ITPs offer a set of tools to deal with it. In the literature, several implementations of algebraic structures have been produced for different systems, and with different aims — most of these implementations were developed in the form of *algebraic hierarchies*.

Coq is probably the most prolific system in this sense. To the best of our knowledge, 4 different approaches have been considered in this system to formalise algebraic structures. An algebraic hierarchy that tries to imitate the hierarchy of the Axiom computer algebra system [38] was implemented in [56]. The formalisation of the Fundamental Theorem of Algebra, see [26], employed the hierarchy presented in [25]. The SSReflect hierarchy, introduced in [24], has played a key role in the formalisation of the Feit-Thompson theorem [27]. Another hierarchy was developed in [63] having as final goal the formalisation of practical exact real arithmetic.

Two Ph.D. theses have been devoted, at least partially, to this topic. A hierarchy for the Nuprl system appeared in Jackson's thesis [37], and was the basis for proving some results about abstract algebra — this hierarchy had as final aim the connection with the Weyl computer algebra system [66]. Bailey implemented in his Ph.D. thesis [7] an algebraic hierarchy that was used to formalise part of Galois theory in Lego.

There are also different approaches in the family of HOL theorem provers. A basic theory of groups was developed in HOL using the hierarchy presented in [29]. As can be seen in [9], the Isabelle/HOL system provides a library to formalise abstract algebra which has been successfully used to prove, for instance, Sylow theorems. In addition, there is also a hierarchy for relation and Kleene algebras in Isabelle [6, 23], and a hierarchy that models Axiom's hierarchy [8].

We can also find a classical set-theoretic treatment of algebra in Mizar. The different structures, like group, ring and field are defined in several papers by various authors [39]. A report about some formalisation issues faced during these developments can be seen in [60]. Abstract algebra has been also formalised in constructive set theory using the MetaPRL system [65].

These algebraic hierarchies are interesting on their own (they have been used to formalise an impressive number of results); but, most of them are not directly related to CAS. However, there are some hierarchies that have been applied to study CAS [8, 37, 56]. An interesting example is the use of different hierarchies to formalise several constructions coming from the Kenzo computer algebra system [21].

Kenzo is a Common Lisp system devoted to Algebraic Topology which was developed by Francis Sergeraert. The Kenzo system has obtained some results not confirmed nor refuted by theoretical or computational means [61], and also has been used to refute some computations obtained by theoretical means [57, 58]. This implies that increasing the user's trust in this system is a relevant issue.

Several ITPs have been used to formalise different instrumental results implemented in Kenzo. The Basic Perturbation Lemma [59] was formalised in Isabelle/HOL (see [3]) using the algebraic hierarchy presented in [9]. In the same line, the Effective Homology of the Bicomplexes was formalised in Coq (see [20]) extending the algebraic hierarchy of [25]. These formalisations were related to algorithms and not to the real programs implemented in Kenzo. The problem of extracting programs from the Isabelle/HOL proofs was studied [4], but even there, the programs are generated in ML, far from Kenzo.

The ACL2 theorem prover is oriented to prove properties of programs written in (a subset) of Common Lisp; and, it has been successfully employed to study some critical fragments of actual Kenzo code [2,35,36,50] — due to the restrictions of the ACL2 language, some well-known and *safe* transformations were required (e.g. loops were replaced by tail-recursive functions). The work presented in [2, 35, 36, 50] did not concern algebraic structures, but ACL2 has also been used to formalise constructions (the Normalisation theorem and the Eilenberg-Zilberg theorem [59]) involving algebraic structures implemented in

Kenzo, see [44, 47]. In contrast to the work in Isabelle and Coq, the algebraic structures involved in the ACL2 formalisations were developed from scratch instead of using, as a basis, a previously developed algebraic hierarchy.

The formalisations presented in [44, 47] show that it is possible to reason about algebraic concepts in ACL2; however, the first-order quantifier-free logic of ACL2 stands in the way of a widespread use of this ITP to work with algebraic structures. In this paper, we show how this weakness can be overcome thanks to two of ACL2's great strengths: *programmable extensibility* and *proof automation*. In particular, we illustrate a methodology for implementing a set of tools that makes working with algebraic structures in ACL2 easier — as a result, a hierarchy of algebraic structures ranging from setoids to vector spaces has been developed. Using these tools as a basis, we can easily develop generic theories that model statements like "Let A be an S algebraic structure, then ...". This can be applied, for instance, to define generic constructions (e.g. the definition of the direct product from two generic groups) — generic constructions are instrumental in CAS, since they simplify the creation of new objects from others that have been previously defined [62].

Once ACL2's weakness is overcome, the use of ACL2 by CAS developers is justified for two reasons. First, the ACL2 learning curve for CAS developers is probably not as steep as in other ITPs since ACL2's prover is semi-automatic, and its language (Common Lisp) is the same as in several CAS (e.g. Kenzo, Axiom [38], Maxima [52], Reduce [32] and Weyl [66]). In addition, since the language is the same, ACL2 formalisations are closer to actual CAS code than formalisations in other ITPs — this might require some code-transformations and proofs by successive refinements, as for instance applied in [50].

The rest of this paper is organized as follows. In the next section, we present a brief introduction to the ACL2 system and the tools employed in our development. Section 3 presents how algebraic structures can be modelled from scratch in ACL2 (this is the approach followed in [44, 47]), and the difficulties that are associated with this process. To overcome those problems, we present a methodology (by means of examples) to implement a set of tools for algebraic structures in Section 4. These tools are the basis to simplify the development of generic theories about algebraic structures, as explained in Section 5. In Section 6, the ideas presented in Sections 4 and 5 are extrapolated to the formalisation of a construction implemented in the Kenzo system: the cone of a chain complex morphism; additionally, this section shows a comparison of different approaches to tackle this formalisation. Finally, Section 7 concludes the paper.

2 A Brief Introduction to ACL2

ACL2 [41, 43] is a programming language, a logic, and a theorem prover supporting reasoning in the logic. The ACL2 programming language is an extension of an applicative subset of Common Lisp. The ACL2 logic is a first-order logic with equality, used for specifying properties and reasoning about the functions defined in the programming language. All the variables in the formulas allowed by the ACL2 system are implicitly universally quantified. The syntax of its terms and formulas is that of Common Lisp, and it includes axioms for propositional logic, equality, and for a number of predefined Common Lisp functions and data types. Rules of inference of the logic include those for propositional calculus, equality and instantiation.

One important rule of inference is the *principle of induction*, that allows proofs by well-founded induction on the ordinal ε_0. The logic has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers. The system also includes the usual well-founded order relation defined on this set of ordinals.

By the *principle of definition*, new function definitions are admitted as axioms only if there exists a measure and a well-founded relation with respect to which the arguments of each recursive call decrease, thus ensuring that the function terminates. In this way, new definitions do not introduce inconsistencies. Usually, the system can prove automatically termination properties using both a predefined ordinal measure and the built-in well-founded relation on ordinals. Nevertheless, if the termination proof is not trivial, the user has to explicitly provide a measure on the arguments and a well-founded relation with respect to which this measure decreases. In addition, new function definitions must be total on the language of terms, so when functions are naturally defined only working on a subset of terms, some behaviour must also be defined on arguments outside of that subset.

An additional way to introduce new function symbols in the logic is by means of the encapsulate mechanism [42]. Instead of giving their definitional body, only certain properties are assumed about them; to ensure consistency, witness functions (which are functions local to an encapsulate block) having the same properties have to be exhibited. Inside an encapsulate, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms.

A derived rule of inference, called *functional instantiation* [41], provides a limited higher-order-like reasoning mechanism instantiating the function symbols of a previously proved theorem. This rule replaces function symbols with other functions, provided it can be proved that the new functions satisfy the constraints or the definitional axioms of the replaced functions (depending on whether they were introduced by an encapsulate or by the principle of definition, respectively).

The ACL2 theorem prover mechanises the ACL2 logic, and is particularly successful obtaining mechanical proofs, mainly, based on simplification and induction. The role of the user in this mechanisation is important: usually a non-trivial result is not proved in a first attempt, and the user has to lead the prover to a successful proof providing a set of lemmas, inspired by the failed proof, that the prover uses mainly as rewriting rules.

In this work, we extensively use *macros* [41], a mechanism for creating specialised notation, and for abbreviating commonly occurring expressions in

ACL2. Macros are functions on s-expressions whose output is interpreted as an ACL2 command containing terms and formulas computed from the input.

We will skip many details and some of the function definitions will be omitted. The interested reader can consult the complete source code at `http://www.unirioja.es/cu/joheras/ahomsia/`. In addition, a detailed explanation of the implementation of the tools presented in this paper can be read in [34].

3 Defining Algebraic Structures in ACL2 From Scratch

In this section, we explain how algebraic structures can be modelled from scratch in ACL2, and the problems associated with this process. As a running example, we consider the definition of *setoids* [12].

A *setoid* $\mathcal{X} = (X, \sim_X)$ is a set X together with an equivalence relation $\sim_X$ on it. Setoids are commonly used in the mechanised development of algebraic structures (see [7,9,25,29,56,63]) mainly for two reasons. From a mathematical point of view, we can form the quotient of a set by changing its equivalence relation — we will provide an example in Section 5. Moreover, the representation of a set in a computer needs the encoding of the equality of such a set (this was deeply studied in [45]), and setoids can be used with this aim.

A setoid can be represented by means of two functions: the characteristic function of the underlying set (the *invariant*), and a binary function encoding the *equivalence relation.* ACL2 provides a way to define equivalence rules [28], but those equivalence rules must be total (i.e. they must be equivalence rules on the whole universe of ACL2 terms); so, they cannot be used in the case of setoids since the equivalence rules of setoids are restricted to the domain of a concrete set.

Example 1 The setoid whose underlying set is the integers, and whose equivalence relation makes integers with same absolute value equivalent can be modelled by an ACL2 user as follows. He could use the ACL2 `integerp` function as invariant (`integerp` is a recogniser for integer numbers that returns `true` if its argument is an integer, and `nil` otherwise), and the `eq-abs` function as equivalence relation.

```
(defun eq-abs (a b)
  (equal (abs a) (abs b)))
```

To prove that these two functions form a setoid, it is necessary to prove the theorems (events whose successful evaluation extends the ACL2 logic) ensuring that `eq-abs` is an equivalence relation on the set characterised by `integerp`. For instance, the reflexivity property is given by the following theorem (that is automatically proven by ACL2).

```
(defthm eq-abs-x-x
  (implies (integerp x) (eq-abs x x)))
```

Example 1 illustrates how concrete setoids can be modelled in ACL2; however, this representation is not enough to deal with the scenarios that involve *generic* setoids (e.g. the proof of *universal properties*). In order to tackle this problem, the ACL2 user should employ the encapsulate mechanism (see Section 2 or [42]). This tool can be used to define a *generic setoid*; namely, the user can define two generic functions `X-inv` (the invariant function) and `X-eq` (the equivalence relation) assuming the properties of setoids. The definition of a *generic* setoid can be seen as an *equational algebraic specification* of this mathematical structure where the type information is missed.

Using the functions (`X-inv` and `X-eq`) introduced in the encapsulate to define the generic setoid, the ACL2 user could prove universal properties that, afterwards, could be instantiated for concrete setoids using the *functional instantiation* mechanism [41].

Example 2 Based on the encapsulate that defines the generic functions `X-inv` and `X-eq` satisfying the definitional axioms of setoids, the ACL2 user can prove the following general property about setoids:

```
(defthm symmetry-transitive
   (implies (and (X-inv x) (X-inv y) (X-inv z)
                 (X-eq y x) (X-eq y z))
            (X-eq x z)))
```

and, subsequently, instantiate it for the integers setoid defined in Example 1.

The encapsulate mechanism is not only the basis to prove universal properties, but it also can be used to define generic constructions.

Example 3 Given two setoids $\mathcal{X} = (X, \sim_X)$ and $\mathcal{Y} = (Y, \sim_Y)$, the Cartesian product of $\mathcal{X}$ and $\mathcal{Y}$, denoted by $\mathcal{X} \times \mathcal{Y}$, is given by $\mathcal{X} \times \mathcal{Y} = (X \times Y, \sim_{X \times Y})$ where $X \times Y$ is the Cartesian product of sets, and $(x_1, y_1) \sim_{X \times Y} (x_2, y_2)$ if $x_1 \sim_X x_2$ and $y_1 \sim_Y y_2$.

To model this generic construction in ACL2, the user starts by calling the encapsulate mechanism to define two "generic setoids" whose components are (`X-inv`, `X-eq`) and (`Y-inv`, `Y-eq`) respectively. From these 4 functions, he can construct the two functions (invariant and equality) that encode the Cartesian product:

```
(defun X-Y-inv (pair)
   (and (= (len pair) 2)
        (X-inv (first pair)) (Y-inv (second pair))))

(defun X-Y-eq (pair1 pair2)
   (and (X-eq (first pair1) (first pair2))
        (Y-eq (second pair1) (second pair2))))
```

Finally, he needs to prove the definitional axioms of setoids for the functions `X-Y-inv` and `X-Y-eq`. Once that this is done, the generic Cartesian product construction can be instantiated for any two concrete setoids. This example and Example 2 illustrate how the encapsulate mechanism can be used to simulate higher-order logic in ACL2.

This from-scratch approach to define concrete and generic setoids, prove universal properties, and create generic constructions can be extrapolated to deal with any algebraic structure in ACL2. Nevertheless, there are several problems associated with this approach:

P.1. The functions that are used to define an algebraic structure do not form an entity, but they are defined separately.
P.2. The definitional axioms for the objects have to be stated *manually* and *for each* instance of an algebraic structure — this can be especially time-consuming and error-prone when there are several definitional axioms or several instances.
P.3. Related to the previous problems, algebraic structures share some properties and components (e.g Abelian groups are groups with an additional property), but this fact cannot be directly captured with the from-scratch representation of algebraic structures.
P.4. The use of encapsulates to create generic instances of algebraic structures can be also a time-consuming and an error-prone task for the user when the structure is complex, or when several encapsulates have to be defined.
P.5. The development of generic theories (e.g. the proof of generic properties, or the construction of generic objects from generic definitions) involves several repetitive steps that could be simplified.

The next sections explain how these problems have been solved.

4 A Set of Tools to Model Algebraic Structures and Morphisms

In this section, we illustrate a methodology for defining a set of tools that fulfils three goals: the creation of entities gathering the components of an algebraic structure, the simplification of the statement of the definitional axioms of an algebraic structure, and the simplification of the definition of generic instances of an algebraic structure — these goals correspond to Problems **P.1**–**P.4**, Problem **P.5** will be tackled in the next section.

We present how this set of tools can be defined for an algebraic structure; and, subsequently, how the same ideas can be extrapolated to implement a hierarchy of algebraic structures.

4.1 Tools to Model an Algebraic Structure

Let us present the creation of the three different tools for a given algebraic structure. As a running example, we retake the definition of setoids presented in Section 3.

Gathering the components of an algebraic structure. Most ITPs use records (called `class` or `record` in Isabelle [30, 54], `Class` or `Record` in Coq [15], `struct` in Mizar [60] and so on) to pack the operations of algebraic structures. In ACL2, the same approach can be used by gathering the operations of a structure in a record defined by means of the `defstructure` macro [13].

Given a structure S with operations $op_1, \ldots, op_n$, a record called `S` with slots `op1`, …, `opn` can be defined. Since ACL2 is an untyped system, there will not be type information attached to the field names of the record. A concrete S instance whose components are given by the functions $f_1, \ldots, f_n$ will be created using the macro `make-S` (a macro automatically generated when the `S` record is defined) having as arguments the names of the functions implementing $f_1, \ldots, f_n$ (those functions must have been previously introduced in ACL2) — since ACL2 is not a higher-order system, this is the *only* way of treating functions as data. Each instance of a given structure can be stored for future use as the value of a user-specified constant symbol (constant symbols in ACL2 are symbols beginning and ending with the character `*`).

This approach models more accurately an algebraic structure than having the functions separately (Problem **P.1**), and provides a name to the structure (the components of a structure could be gathered in a list, but that approach would not assign a name to the structure).

Example 4 In the case of setoids, the authors of the present paper have defined a record called `setoid` with two fields (`inv` and `eq`) that will store respectively the names of the invariant function and the intended equivalence relation.

```
(defstructure setoid
  inv eq)
```

Using this representation, if an ACL2 user wants to create the setoid from Example 1, he should construct an instance of the `setoid` record, where the values of the `inv` and `eq` slots are respectively the names `integerp` and `eq-abs`. Moreover, he could assign this instance to a constant, for example `*Zabs*`, for latter use in his development.

```
  (defconst *Zabs* (make-setoid :inv 'integerp :eq 'eq-abs))   (1)
```

Simplifying the statement of definitional axioms. In some ITPs (e.g. Coq or Isabelle), records not only pack the operations of the structure, but also include the axioms about such operations. On the contrary, in ITPs like Mizar, the axioms are external to the record. In ACL2, the only feasible approach is the latter — the macro `defstructure` allows ACL2 users to attach assertions to a record, but this is not possible if the slots of the record have a functional nature, as in the case of algebraic structures.

In order to facilitate the statement of the event that ensures the definitional axioms of an algebraic structure S (Problem **P.2**), a function called `S-algebraic-structure` and a macro called `check-S-p` should be defined.

The function `S-algebraic-structure` should take as argument an *S* instance, and produce a "textual" (quoted in Lisp terminology) conjunction of formulas with the definitional axioms of the *S* structure for the functions of the *S* instance. The macro `check-S-p` should take as argument a constant, `*X*`, storing an *S* instance; and, internally invoke the function `S-algebraic-structure` generating a `defthm` event, called `*X*-is-an-S`, that checks whether the *S* instance stored in `*X*` satisfies the definitional axioms of an *S* structure. The `check-S-p` macro can be seen as a characteristic function for the type of *S* structures.

Example 5 In our running example about setoids, the authors of the present paper have defined a function called `setoid-algebraic-structure` that takes as argument a `setoid` instance and produces a conjunction of formulas stating that the `eq` component of the `setoid` instance is an equivalence relation on the set characterised by the `inv` component of the `setoid` instance — i.e. the definitional axioms of setoids. Additionally, we have defined the macro `check-setoid-p`, that can be used to certify that a constant storing a `setoid` instance is really a setoid. For instance, given the constant storing the `setoid` from Example 4 (`*Zabs*`), the macro invocation

```
(check-setoid-p *Zabs*)                    (2)
```

expands into a call of `defthm` whose name is `*Zabs*-is-a-setoid` stating that the functions of the setoid instance `*Zabs*` satisfy the definitional axioms of setoids. In this example, ACL2 proves automatically such a theorem, but in other cases, ACL2 could require the user intervention to guide the proof.

To do this example, the only user input required are Commands 1 and 2.

As can be seen in the above example, the macros `check-S-p` greatly reduce the burden of checking whether *S* instances satisfy the axioms of *S* structures. This is especially relevant when working with several instances of a structure, or when there are several definitional axioms.

Simplifying the definition of generic instances. In mathematical textbooks, it is usual to start the statement of a theorem with a sentence of the form "Let *A* be an *S* structure". The translation of such a statement to ITPs like Isabelle or Coq is straightforward (in Coq, the statement will be given by `forall A:S`). However, in ACL2, the user needs to define an encapsulate providing both the components and the axioms of the generic *S* structure (Problem **P.4**). This approach is far from that of working mathematicians and the users of other ITPs.

This problem can be tackled by defining a macro, let us call it `defgeneric-S`, that constructs generic instances of an *S* structure. The macro should internally invoke the encapsulate mechanism; in particular, it should take as input a symbol, `<symbol>`, and generate a constant, `*<symbol>*`, storing a generic *S* instance. In addition, such a macro should generate the theorem `*<symbol>*-is-an-S` stating that the generic instance stored in `*<symbol>*`

is an S structure — this theorem can be generated using the `check-`*S*`-p` macro. To simplify the use of these macros, the names of the components of a generic instance created with a macro `defgeneric-`*S* should follow the convention `<symbol>-<slot>` where `<symbol>` is the symbol given in the macro call, and `<slot>` is the name of a slot in the S record.

Example 6 In the case of setoids, the authors of the present paper have defined the macro `defgeneric-setoid`. An ACL2 user could now invoke the macro

```
(defgeneric-setoid X)
```

and such a macro will expand into an encapsulate that produces the constant `*X*`, storing a generic setoid (which components are `X-inv` and `X-eq`), and the theorem `*X*-is-a-setoid`, that ensures that the setoid stored in the constant `*X*` satisfies the definitional axioms of setoids. Subsequently, he could state the property `symmetry-transitive` in the same way as in Example 2, but with the advantage of skipping the step of defining the encapsulate.

Similarly to the macros `check-`*S*`-p`, the macros `defgeneric-`*S* are especially useful when it is necessary the definition of several generic instances of an S structure, or when there are several definitional axioms. We will illustrate these benefits in Sections 5 and 6.

4.2 A Hierarchy of Algebraic Structures and Morphisms

In the previous subsection, we have introduced a set of tools that simplifies the interaction with algebraic structures in ACL2. Following that approach, structures like groups and rings could also be defined; but, the relations between those structures would not be captured (Problem **P.3**). The rest of this section explains how the ideas presented previously have been extrapolated to define the hierarchy of algebraic structures and morphisms depicted in Figure 1.

Let us start with some general comments about such a hierarchy. The mathematical structures of our hierarchy, ranging from *setoids* to *vector spaces*, are depicted in the left side of Figure 1. As in several algebraic hierarchies [7, 9, 25, 29, 56, 63], setoids are the top structure of the hierarchy; other hierarchies, like [60, 65], are set-based, and the SSReflect hierarchy [24] uses a choice structure as the top level object.

A continuous arrow with an open triangle as tip represents an *inheritance* relationship modelling that the source mathematical structure *is-a* target mathematical structure, e.g. an Abelian group *is a* group with some additional properties. Whereas, a continuous arrow with a normal tip describes a *use* relationship, in the sense that the target mathematical structure *is used* to define the source structure, e.g. a vector space uses a field in its definition.

The morphisms included in our hierarchy are presented in the right side of Figure 1. A morphism always consists of a source structure A, a target structure B of the same type as A, and a map between them.

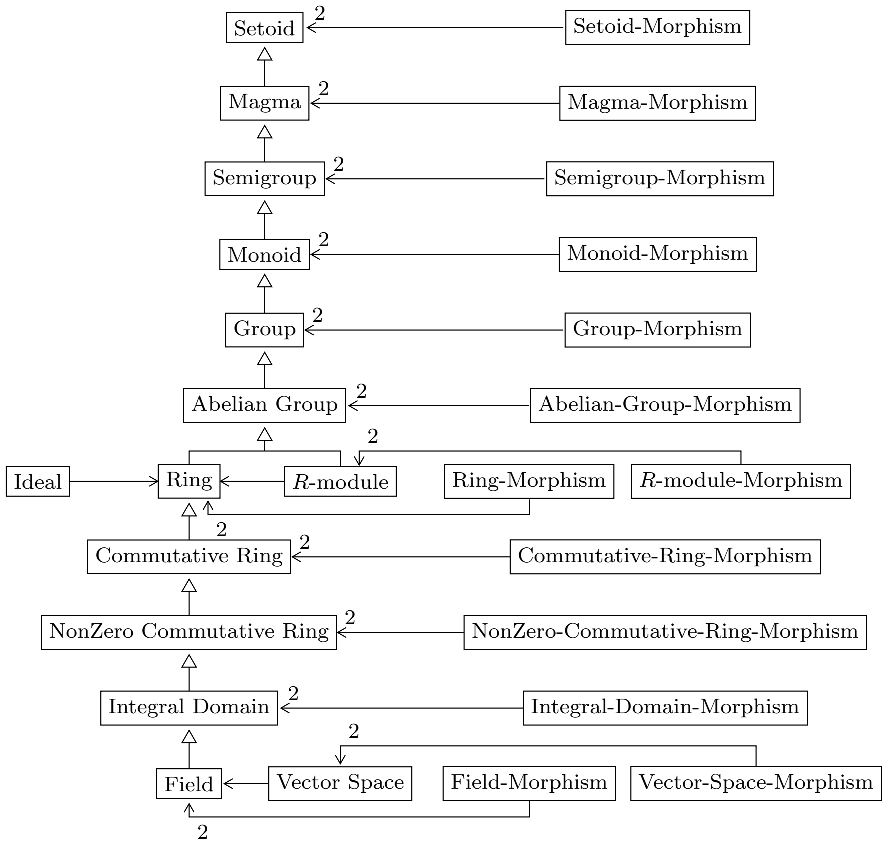

Fig. 1 Hierarchy of mathematical structures and morphisms.

For each structure and morphism of the hierarchy, the authors of the present paper have defined three tools (as we have explained in the previous subsection): a record S, and the macros `check-`S`-p` and `defgeneric-`S where S is the name of the structure or morphism. However, since these tools need to capture the relations between the structures, they are slightly different from the tools presented in the previous subsection.

Gathering the components of an algebraic structure. We have defined a record for each structure and morphism of our hierarchy. In the case of setoids, we use the representation introduced in Example 4 since this structure does not have any inheritance or use relationship. The rest of the structures can be split into three classes: (1) a structure that inherits from another structure, and has (in some cases) additional operations satisfying further properties; (2) a structure that uses one or more structures, and has (in some cases) additional operations satisfying further properties; and (3) a combination of (1) and (2). Let us see the ACL2 representation for an S structure belonging to Class (3), where S inherits from an A structure, uses a B structure, and has additional operations $op_1, \ldots, op_n$. In such a case, the record for S is defined as follows.

```
(defstructure S
  A B op1 ... opn)
```

The representation for structures that belong to Classes (1) and (2) is analogous; but for those structures, some slots will not be included.

In the construction of instances of the above S structure (using the macro `make-S`), the values of the `A` and `B` fields will be an `A` instance and a `B` instance respectively, and the values of `op1`, ..., `opn` slots will be function names. The construction of instances can be a cumbersome task since we have a hierarchy of nested structures. For instance, in order to construct an `Abelian-group`, it is necessary to use the definition of a `group`, which in turn needs the definition of a `monoid`, and so on. To overcome this problem, the authors of the present paper have defined a function called `create-S` for each structure S of the hierarchy. This function takes as arguments the names of the functions that are the components of the S structure, and builds an `S` instance with them. As we have explained in Subsection 4.1, the instances are stored in constants for latter use.

Example 7 Given a ring R, an R-module is an Abelian group with an external operation satisfying a number of properties, see [17]. Then, the representation of R-modules requires both *inheritance* and *use* relationships.

```
(defstructure R-module
  Abelian-group Ring external_operation)
```

If an ACL2 user wants to define an instance of the R-module structure using the macro `make-R-module`, he should proceed as follows:

```
(defconst *RM*
  (make-R-module
   :Abelian-group (make-Abelian-group :group (make-group ...))
   :Ring (make-ring :Abelian-group (make-Abelian-group ...) ...)
   :external_operation 'f))
```

The user can construct the same R-module using the `create-R-module` function taking as arguments the names of the functions used to define the `Abelian-group` instance (`a1`, ..., `ak`), the `Ring` instance (`r1`, ..., `rm`), and the external operation (`f`).

```
(defconst *RM* (create-R-module 'a1 ... 'ak 'r1 ... 'rm 'f))
```

Simplifying the statement of definitional axioms. We focus now on the statement of the event that generates the definitional axioms of an algebraic structure. As we have explained in Subsection 4.1, this is achieved thanks to a function `S-algebraic-structure` and a macro `check-S-p` where S is the name of a given structure. In the case of the structures and morphisms of the hierarchy, the advantage when defining these functions and macros is that it is not necessary to define them from scratch, but the functions previously defined for other structures can be re-used. This solves the re-usability problem stated at the end of Section 3 (Problem **P.3**).

Reconsider the case of an S structure that inherits from an A structure, uses a B structure, and has additional operations $op_1, \ldots, op_n$. For an `S` instance, the function `S-algebraic-structure` needs to state that: (1) the `A` component of the instance is an A structure, (2) the `B` component of the instance is a B structure, and (3) the operations of the instance (including the operations of the `A` and `B` slots) satisfy additional properties $P_1, \ldots, P_m$. For (1) and (2), the function `S-algebraic-structure` can re-use the respective functions for A and B structures; and for (3), it is necessary to define functions that state the properties $P_1, \ldots, P_m$. Finally, all these functions should be invoked by the function `S-algebraic-structure`.

The macro `check-S-p` should be defined as explained in Subsection 4.1. Such a macro should take as argument a constant, `*X*`, storing an `S` instance, and internally invoke the function `S-algebraic-structure` generating a `defthm` event, called `*X*-is-an-S`. This event should check (via a proof attempt) whether the `S` instance stored in the constant `*X*` satisfies the definitional axioms of the S structure.

Simplifying the definition of generic instances. Following the ideas presented in Subsection 4.1, the authors of the present paper have defined a set of macros (called `defgeneric-S` where `S` is the name of a given structure) to work with generic instances of the mathematical structures and morphisms of Figure 1; but, without explicitly using encapsulates. The macros take as argument a symbol and produce: a constant storing a generic instance of the structure or morphism, and a theorem stating the definitional axioms for the generic instance.

In addition, we have extended the functionality of the macros for those structures and morphisms that involve one or more use relationships. Consider a structure A that uses structures $A_1, \ldots, A_n$; for such a structure, the ACL2 user might be interested not only in creating generic A instances, but also in creating a generic A instance with fixed $A_1, \ldots, A_n$ instances (e.g. he might be interested in defining generic morphism between generic setoids, but also generic setoid morphisms where the source and target setoids are the integers setoid given in Example 4). Therefore, the macros coming from structures that use other structures have been modified to take a symbol, and optionally as many arguments as used structures — if only the symbol is provided, the macro behaves as explained in Subsection 4.1; otherwise, it constructs a generic instance parameterised by the given arguments.

Example 8 Consider the case of setoid morphisms (a structure with two use relationships), if the ACL2 user invokes the generic macro for creating a setoid morphism (i.e. the `defgeneric-setoid-morphism` macro) using only the symbol `X` as argument:

```
(defgeneric-setoid-morphism X)
```

the above macro will produce the constant `*X*` storing a generic setoid morphism with generic setoids as source and target of the morphism. However, if he invokes the macro with the symbol `X`, and the setoid `*Zabs*` (see Example 4) as arguments:

```
(defgeneric-setoid-morphism X *Zabs* *Zabs*)
```

this macro will create the constant `*X*` storing a generic setoid morphism with `*Zabs*` as both source and target setoid of the morphism.

5 Developing Generic Theories for Algebraic Structures

The tools presented in the previous section can be used to simplify the development of generic theories about algebraic structures. In general, the procedure to create a generic theory consists of three steps: (1) introduction of generic function symbols constraining them to have certain properties, this is achieved using the encapsulate mechanism; (2) definition of functions from the generic function symbols; and, (3) derivation of theorems from the generic function symbols. Once a generic theory is defined, it can be instantiated for concrete functions using the functional instantiation mechanism.

As we have explained in Problem **P.5**, the from-scratch creation of generic theories about algebraic structures has several difficulties. However, they can be overcome using the tools presented in the previous section; namely, the ACL2 user can take advantage of the tools presented in this paper in Steps (1) and (3). In this section, we illustrate this fact in the development of three generic theories: the Cartesian product of setoids, the subalgebra criterion, and the definition of homology groups. We finish the section with a general proof-scheme to create generic theories about algebraic structures using our tools (or tools developed using our methodology).

5.1 Cartesian Product of Setoids

Let us reconsider, using the tools presented in Section 4, the definition of the generic Cartesian product of setoids presented in Example 3. As a first step, the ACL2 user starts by defining two generic setoids:

```
(defgeneric-setoid X)     (3)
(defgeneric-setoid Y)     (4)
```

These macro calls will automatically produce two generic setoids `*X*` and `*Y*` with operations (`X-inv`, `X-eq`) and (`Y-inv`,`Y-eq`) respectively. Using the components of these setoids, the user can define the functions `X-Y-inv` and

`X-Y-eq`, that are exactly the same functions defined in Example 3, construct a `setoid` instance, and store it in the constant `*XxY*`.

```
(defconst *XxY* (create-setoid 'X-Y-inv 'X-Y-eq))        (5)
```

Finally, the user can certify that `*XxY*` is a setoid using the macro call:

```
(check-setoid-p *XxY*)        (6)
```

ACL2 finds automatically the proof of the event generated by this macro invocation. This approach is much simpler than the from-scratch approach presented in Example 3, and it requires less effort from the user point of view: using the tools presented in this paper, the user has to type 3 macros, and define 2 functions and 1 record (a total of 10 lines of code given by the definition of `X-inv` and `X-eq`, and Commands 3–6); on the other hand, using the from-scratch approach of Example 3, he needs to define 2 encapsulates (involving 4 functions and 8 theorems), 2 definitions and state 4 theorems (more than 50 lines of code).

5.2 Subalgebra Criterion

The *subalgebra criterion* [17] is a well-known result of Universal Algebra stating that given $\mathcal{X} = (X, op_1, \ldots, op_n)$ a mathematical structure where X is the underlying set of $\mathcal{X}$, and Y a subset of X closed with respect to $op_1, \ldots, op_n$; then, $\mathcal{Y} = (Y, op_1, \ldots, op_n)$ is of the same type as $\mathcal{X}$. This result has been proved for all the structures of our hierarchy, we consider here the proof of the subalgebra criterion for magmas — the proof of this result for the rest of structures is analogous.

Theorem 1 *Given $\mathcal{M} = (M, \sim_M, \circ_M)$ a magma and N a subset of M closed with respect to $\circ_M$; then, $\mathcal{N} = (N, \sim_M, \circ_M)$ is a magma.*

The ACL2 user can prove this result using the tools presented in the previous section as follows. First, he defines a generic magma instance using the `defgeneric-magma` macro (a macro defined by the authors of the paper analogously to the macro `defgeneric-setoid`) taking the symbol `M` as argument. Afterwards, a *generic subset* of M closed with respect to $\circ_M$ is defined using the encapsulate principle, where `N-inv` is the invariant of that generic subset. Note the benefits of using our tools: a generic magma is defined in one line using the `defgeneric-magma` macro; however, the definition of the generic subset requires an explicit encapsulate. This could be solved by defining a `defgeneric` macro, parameterised by a magma, that constructs the desired generic subset.

Once the generic subset is defined, the user constructs a `magma` instance where `N-inv` is the invariant, `M-eq` is the equivalence relation, and `M-binary-op` is the binary operation; and stores the result in the constant `*N*`.

```
(defconst *N* (create-magma 'N-inv 'M-eq 'M-binary-op))
```

Finally, he invokes the macro `check-magma-p` (another macro defined by the authors of the present paper) to certify that `*N*` is a magma:

```
(check-magma-p *N*)
```

ACL2 finds the proof of the event generated by this macro call without any external help.

5.3 Definition of Homology Groups

We consider now a more involved example: the definition of homology groups in the context of Homological Algebra — an introduction to this mathematical subject can be seen in [64].

Definition 1 Let $f : G_1 \to G_2$ and $g : G_2 \to G_3$ be Abelian group morphisms such that $\forall x \in G_1, gf(x) \sim_{G_3} 0_{G_3}$ (where 0_{G_3} is the neutral element of G_3); then, the *homology group* of (f, g), denoted by $H_{(f,g)}$, is the Abelian group $H_{(f,g)} = \ker(g)/im(f)$.

The condition $\forall x \in G_1, gf(x) \sim_{G_3} 0_{G_3}$, known as *nilpotency condition*, makes the above definition meaningful, since $im(f) \subseteq \ker(g)$. This definition involves several constructions of Universal Algebra such as subalgebras, morphisms and quotients [17].

Given $f : G_1 \to G_2$ and $g : G_2 \to G_3$ Abelian group morphisms such that $\forall x \in G_1, gf(x) \sim_{G_3} 0_{G_3}$; the ACL2 user can employ the tools presented in the previous section to define $H_{(f,g)}$ and prove that it is an Abelian group.

First of all, he starts by defining three generic Abelian groups (`*G1*`, `*G2*` and `*G3*`) using the `defgeneric-Abelian-group` macro.

```
(defgeneric-Abelian-group G1)
(defgeneric-Abelian-group G2)
(defgeneric-Abelian-group G3)
```

The components of these generic Abelian groups are: `G<i>-inv`, the invariant function of the underlying setoid of the group, `G<i>-eq`, the equivalence relation of the group, `G<i>-binary-op`, the binary operation, `G<i>-id-elem`, the identity element, and `G<i>-inverse`, the inverse function, with `<i>=1,2,3`.

Using the encapsulate principle, the ACL2 user defines two generic Abelian group morphisms $f : G_1 \to G_2$ and $g : G_2 \to G_3$ such that the nilpotency condition is satisfied.

```
(encapsulate
   ;  SIGNATURES
   (((f *) => *)
    ((g *) => *))

   ;  GENERIC ABELIAN GROUP MORPHISMS DEFINITION
   (defconst *f*
     (make-Abelian-group-morphism :source *G1*
                                   :target *G2* :map 'f))
   (defconst *g*
     (make-Abelian-group-morphism :source *G2*
                                   :target *G3* :map 'g))

   ;  ABELIAN GROUP MORPHISM AXIOMS
   (check-Abelian-group-morphism-p *f*)
   (check-Abelian-group-morphism-p *g*)

   ;  NILPOTENCY CONDITION
   (defthm nilpotency-condition
      (implies (G1-inv x)
               (G3-eq (g (f x)) (G3-id-elem))))
)
```

The above encapsulate must be read as follows. First of all, the signatures of the functions `f` and `g` are provided, the notation `((f *) => *)` means that the function `f` has an argument, that belongs to the universe of ACL2 terms, as input and returns another term as output — the encapsulate also requires witnesses for the functions `f` and `g`, but we do not include them here since they are not relevant. Subsequently, the constants `*f*` and `*g*` that store the two generic Abelian group morphisms are defined. Afterwards, using the `check-Abelian-group-morphism-p` macro, the axioms of Abelian group morphism to `*f*` and `*g*` are imposed. Finally, the nilpotency condition is imposed.

The macro `defgeneric-Abelian-group-morphism` cannot be used here since the user does not only want to build generic Abelian group morphisms, but also impose the nilpotency condition. As in the case of the subalgebra criterion, it is possible to define a `defgeneric` macro that constructs two Abelian group morphisms satisfying the nilpotency condition, and that is parameterised by three Abelian groups. Such a macro would reduce the above encapsulate to just one macro call.

Once that the user has defined the Abelian group morphisms f and g, he can introduce $ker(g)$, $im(f)$ and, subsequently, the quotient $ker(g)/im(f)$. The set $ker(g) = \{x \in G_2 : g(x) \sim_{G_3} 0_{G_3}\}$ (the invariant of such a set is encoded in ACL2 by means of a function called `ker-g-inv`) is a subset of the underlying set of G_2, and it is closed with respect to the group operations of G_2. These are the conditions of the subalgebra criterion for Abelian groups (cf. Subsection 5.2); therefore, the user can instantiate such a result for this concrete case and define the Abelian group $ker(g)$.

```
(defconst *ker-g*
  (create-Abelian-group 'ker-g-inv 'G2-eq 'G2-binary-op
                        'G2-id-elem 'G2-inverse))
```

Subsequently, he can define $im(f) = \{x \in G_2 : \exists y \in G_1, f(y) \sim_{G_2} x\}$ as a subgroup of G_2 using the same idea presented for $ker(g)$. The existential quantifier in the definition of the invariant function of $im(f)$ is introduced using `defun-sk` — the method supported by ACL2 to provide first-order quantification via Skolem functions [48].

```
(defun-sk im-f-inv (x)
  (exists (y)
          (and (G2-inv x) (G1-inv y) (G2-eq (f y) x))))
```

Then, using `im-f-inv` and the operations of `*G2*`, he can encode the Abelian group $im(f)$ using an `Abelian-group` instance.

Afterwards, he can tackle the definition of the homology group $H_{(f,g)}$ as the quotient $ker(g)/im(f)$. Quotienting a structure of the hierarchy is achieved by changing the equivalence relation of the underlying setoid of the structure with another equivalence relation compatible with the operations of the structure. This result has been proved by the authors of the present paper for each structure of the hierarchy following a similar process to the subalgebra criterion (cf. Subsection 5.2).

Since $im(f)$ is a subgroup of $ker(g)$, then $im(f)$ induces an equivalence relation on $ker(g)$ given in ACL2 by the following definition.

```
(defun im-f-eq (x y)
   (im-f-inv (G2-binary-op x (G2-inverse y))))
```

Therefore, $H_{(f,g)}$ should be defined using `ker-g-inv` as invariant, `im-f-eq` as equivalence relation, `G2-binary-op` as binary operation, `G2-id-elem` as the identity element, and `G2-inverse` as the inverse operation.

```
(defconst *homology-fg*
   (create-Abelian-group 'ker-g-inv 'im-f-eq 'G2-binary-op
                         'G2-id-elem 'G2-inverse))
```

The last step consists in certifying that `*homology-fg*` satisfies the definitional axioms of an Abelian group.

```
(check-Abelian-group-p *homology-fg*)
```

ACL2 does not find the proof of the event generated by the above macro call in the first attempt, and some auxiliary lemmas, suggested by the failed proof, are necessary. The way of facing that proof is the usual when trying to prove a result with ACL2: inspect the failed proof attempt and provide the necessary lemmas and hints, this is known in ACL2 as "*the Method*" [41]. Using this procedure, an ACL2 user proves that `*homology-fg*` is an Abelian group.

5.4 A General Proof-Scheme for Generic Theories

The three examples presented throughout this section follow a common proof-scheme. Such a proof-scheme can be applied to several generic theories about algebraic structures. In particular, given the generic objects $s_1, \ldots, s_n$, and the generic operations $op_1, \ldots, op_m$ satisfying the properties $p_1, \ldots, p_k$, the ACL2 user can prove that $t = (\widehat{op_1}, \ldots, \widehat{op_t})$ (where $\widehat{op_1}, \ldots, \widehat{op_t}$ are operations derived from the components of $s_1, \ldots, s_n$, and/or the operations $op_1, \ldots, op_m$) satisfies the definitional axioms of a T structure as follows.

S.1. Definition of the objects $s_1, \ldots, s_n$ using `defgeneric` macros.
S.2. Definition of the generic operations $op_1, \ldots, op_m$ satisfying the properties $p_1, \ldots, p_k$ using the encapsulate mechanism.
S.3. Definition of the functions $\widehat{op_1}, \ldots, \widehat{op_t}$ from the operations of Steps **S.1** and **S.2**.
S.4. Construction of a T instance with operations $\widehat{op_1}, \ldots, \widehat{op_t}$.
S.5. Proof that the instance built in the previous step satisfies the definitional axioms of a T structure using the macro `check-`*T*`-p` — this proof might require some user intervention.

Some of these steps might be unnecessary depending on the concrete problem. For instance, Step **S.2** is not required in the Cartesian product construction (cf. Subsection 5.1) since all the components of the Cartesian product are defined from the components of the generic setoid; and, Step **S.3** is unnecessary in the subalgebra criterion (cf. Subsection 5.2) since all the components of the new magma were introduced previously.

Once a generic theory is completed, it can be instantiated for concrete instances using the functional instantiation mechanism. As an example, the subalgebra criterion for Abelian groups (see Subsection 5.2) is instantiated for $\ker(f)$ and $im(g)$ in the definition of homology groups.

6 A Case Study: Formalising the Cone Construction

In the previous sections, we have presented a set of tools to work with the algebraic structures and morphisms of the hierarchy depicted in Figure 1; in addition, we have shown how these tools can facilitate the development of generic theories. The same ideas can be applied to more complex structures, and it is in those cases that the biggest impact occurs.

In this section, we study how the techniques previously presented can be extrapolated to an Algebraic Topology construction implemented in the Kenzo system: the cone of a chain complex morphism [51] — a representative example of Kenzo's constructions. Chain complexes and chain complex morphisms, as many other structures implemented in Kenzo (like simplicial sets), are defined as *indexed families of structures*; hence, the first step in the formalisation of the cone construction is the representation of indexed families of structures in ACL2. We will finish this section with a comparison of different approaches to formalise the cone construction.

6.1 Indexed Families of Structures

The structures used in Algebraic Topology, such as chain complexes or simplicial sets, are based on *families indexed* on a set, called the *index set.* The method that we have followed to represent these families is based on the approach presented in [46]. Roughly speaking, the representation of a graded structure indexed on a set is achieved thanks to the introduction of an additional parameter, that ranges the elements of the index set, in each operation of the structure. This method differs from the usual approach followed in other ITPs to represent indexed families of structures: a function that takes as input an index, and returns a structure as output (see [20]) — this approach is not feasible in the first-order setting of ACL2.

The tools described up to now in this paper cannot handle graded structures. However, we have extended them to do so, as described briefly in this section. Namely, in order to deal with families of structures in ACL2, we have created a hierarchy of graded structures that mirrors the hierarchy presented in the left side of Figure 1. As in the non-graded hierarchy (cf. Subsection 4.2), we have defined three tools for each structure of the graded-hierarchy, but these tools have some particularities that we explain as follows.

Gathering the components of a graded algebraic structure. For each structure of the graded hierarchy, we have defined a record that gathers the operations of the structure — the relations between structures are captured using the same ideas presented in Subsection 4.2.

In this new hierarchy, the basic object is the *graded setoid.* The record associated with this graded structure contains three fields: `graded-inv`, `graded-eq` and `index-sets`. When an ACL2 user defines a concrete graded setoid, the value of the `graded-inv` and `graded-eq` slots should be respectively a function symbol of arity 2 (representing the underlying graded set of the setoid) and a function symbol of arity 3 (encoding the intended equivalence relation). The value of the `index-sets` slot should be a list with a sole element: a function name that represents the characteristic function of the index set of the graded setoid.

Using the same record, the ACL2 user can also work with n-graded setoids (i.e. a family of setoids indexed on n sets). In the general case, the arities of `graded-inv` and `graded-eq` functions should be $n+1$ and $n+2$ respectively; and, the value of the `index-sets` slot should be a list whose elements are n function names encoding the n characteristic functions of the n sets. If the value of `index-sets` is an empty list, the ACL2 user has an object *"equivalent"* to a `setoid` instance as presented in Section 4. The `index-sets` slot is inherited from the `graded-setoid` structure to the rest of the graded structures of the hierarchy.

Simplifying the statement of definitional axioms. In addition to the records in charge of representing each graded structure, we have defined the macros that provide their definitional axioms (`check-`*S*`-p` macros). As we have just explained, the same record is used to encode an n-graded structure for different values of n (this value is determined by the length of the list stored in the `index-set` slot of the underlying graded setoid); hence, the `check-`$\mathcal{S}$`-p` macros have been defined to produce a term depending on the value of n.

Simplifying the definition of generic instances. Finally, we have introduced a number of macros to define generic instances of an indexed family of structures (`defgeneric-`$\mathcal{S}$ macros). These macros are parameterised by a list of function names. This list encodes the characteristic functions of the underlying index sets of the generic indexed family of structures.

6.2 Chain Complexes and the Cone Construction

The graded-hierarchy of algebraic structures is the basis to define a set of tools to work with chain complexes and chain complex morphisms.

Definition 2 A *chain complex*, C_*, is a family $C_* = (C_n, dC_n)_{n\in\mathbb{Z}}$ where $(C_n)_{n\in\mathbb{Z}}$ is a family of R-modules indexed on the integers and $(dC_n)_{n\in\mathbb{Z}}$ (the *differential* map) is a family of R-module endomorphisms of degree -1 ($dC_n : C_n \to C_{n-1}$) such that $dC_{n-1}dC_n = 0$ (this property is known as *nilpotency* condition).

Let $C_* = (C_n, dC_n)_{n\in\mathbb{Z}}$ and $D_* = (D_n, dD_n)_{n\in\mathbb{Z}}$ be two chain complexes, a *chain complex morphism* from C_* to D_* is a family of R-module morphisms $f = (f_n)_{n\in\mathbb{Z}}$ such that $dD_n f_n = f_{n-1} dC_n$ for each $n \in \mathbb{Z}$.

We include some comments about the records that encode chain complexes and chain complex morphisms. The instrumental notion in the definition of chain complexes is that of graded R-module. This graded structure *is-a* graded Abelian group that *uses* a ring as part of its definition; then, the graded and non-graded hierarchy are necessary to define a graded R-module. From this graded structure, we have represented chain complexes using the following record.

```
(defstructure chain-complex
  graded-R-module diff)
```

When an ACL2 user defines a `chain-complex`, the value of `graded-R-module` should be a `graded-R-module` instance, $\mathcal{C}$, indexed on the set of integer numbers, and `diff` should be a function symbol of arity 2 encoding the differential map, whose mathematical signature is `diff`: $\mathbb{Z} \times \mathcal{C} \to \mathcal{C}$ — i.e. the differential map is *uncurried*, the subscript of the differential map is now one of the inputs (uncurrying is a common mechanism to simulate higher-order using first-order tools).

Chain complexes are used to define chain complex morphisms; in particular, the representation of chain complex morphisms is given by the record

```
(defstructure chain-complex-morphism
  source target map)
```

where the value of both `source` and `target` slots will be `chain-complex` instances, and the value of `map` will be a function symbol whose arity is 2 — as in the case of the differential map in chain complexes, the map of the chain complex morphism is uncurried.

As in the rest of the structures and morphisms presented throughout the paper, the authors of the present paper have defined three tools for chain complexes (and chain complex morphisms): the record that we have just presented to gather the operations of the structure, a macro to check whether the definitional axioms of chain complexes (or chain complex morphisms) are fulfilled, and a macro to create generic chain complex (or chain complex morphism) instances. The behaviour of the macros is analogous to the macros presented previously.

Once that we have introduced the set of tools to work with chain complexes and chain complex morphisms, an ACL2 user can follow the steps of the proof-scheme presented in Subsection 5.4 to create generic theories about these notions. In particular, let us consider a construction implemented in Kenzo: the *cone of a chain complex morphism* [59] — this construction is important in Homological Algebra and plays an important role, for instance, in the proof of the *Short Exact Sequences theorems* [59].

Definition 3 Let $C_* = (C_n, dC_n)_{n\in\mathbb{Z}}$ and $D_* = (D_n, dD_n)_{n\in\mathbb{Z}}$ be two chain complexes and $\phi : D_* \to C_*$ be a chain complex morphism. Then, the cone of ϕ, denoted by $Cone(\phi) = (A_n, dA_n)_{n\in\mathbb{Z}}$, is defined as: $A_n := C_{n+1} \oplus D_n$ (an element $x \in A_n$ is a pair such that its first component belongs to C_{n+1} and the second component to D_n); and

$$\begin{aligned} dA_n : C_{n+1} \oplus D_n &\to C_n \oplus D_{n-1} \\ (c_{n+1}, d_n) &\mapsto (dC_{n+1}(c_{n+1}) + \phi(d_n), -dD_n(d_n)). \end{aligned}$$

In order to define this construction in ACL2, the ACL2 user should start by defining a generic chain complex morphism ϕ (Step **S.1**).

```
(defgeneric-chain-complex-morphism PHI)
```

The above macro call produces the constant `*PHI*` (storing a generic chain complex morphism), and the theorem that ensures that the components of `*PHI*` satisfy the chain complex morphism axioms.

From the components of `*PHI*`, the user needs to introduce the chain complex operations (9 operations are necessary to define a chain complex) defining the cone construction (Step **S.3**). Using these operations, he can create a `chain-complex` instance that is assigned to a new constant, called `*Cone-PHI*`, for latter use (Step **S.4**) — in this case, Step **S.2** is not required since the necessary tools to define the generic objects without explicitly using the encapsulate mechanism are available.

Subsequently, the ACL2 user can invoke the `check-chain-complex-p` macro with `*Cone-PHI*` as argument to prove the event which ensures that the definitional axioms of chain complex are satisfied by this instance (Step **S.5**). ACL2 is not able to find the proof of the event generated by this macro in

	Definition of generic chain complex morphism	Definition of cone construction	Proof of the correctness of the construction
from-scratch	19 function symbols 19 witnesses 84 axioms	9 definitions	49 theorems
half-way	19 function symbols 19 witnesses 84 axioms	9 definitions 1 record	1 macro call
hierarchical	1 macro call	9 definitions 1 record	1 macro call

Table 1 Comparison between the different approaches. The approach presented in this paper is called hierarchical. The columns represent the three steps required in the development of a generic theory: (1) the definition of generic function symbols, (2) the definition of functions from the generic function symbols, and (3) the derivation of theorems from the generic function symbols.

the first attempt. In particular, the user needs to guide the proof for the 9 "trickiest" definitional axioms of chain complexes; the trivial definitional axioms (40 axioms) are automatically proven by ACL2. This means that the user can focus on the difficult parts of the proof; additionally, the proof of these results is guided by the suggestions generated during the failed proof.

Once that these lemmas are proved, and in order to make the instantiation of the cone construction for concrete chain complex morphisms easier, the user could employ the *generic instantiation tool* [49] — a procedure which allows ACL2 users to instantiate generic theories in a simple way.

Several Kenzo constructions — e.g. the Easy Perturbation Lemma, the cone equivalence theorem, and the SES theorems [59] — have been formalised using the same ideas presented for the cone construction, see [33].

6.3 A Comparison with other Approaches

In the above development, we have shown how to take advantage of the tools presented throughout this paper. However, the same formalisation could be performed from scratch, but not without difficulty, as we will see.

First of all, the ACL2 user should employ the encapsulate mechanism to define the generic chain complex morphism ϕ. The definition of this generic object involves 19 function symbols (to define the operations of the chain complex morphism) the corresponding 19 witnesses and 84 axioms (to ensure that the 19 function symbols fulfil the properties that characterise the chain complex morphism operations). Afterwards, from the function symbols of the generic chain complex morphism, he should introduce the operations that define the chain complex associated with the cone construction; as we said previously, this means 9 new definitions. Finally, he could state the 49 events that claim that the 9 operations introduced in the previous step satisfy the definitional axioms of chain complexes. The non-trivial events are the same as before; then, the same auxiliary lemmas are necessary to prove 9 of them.

Table 1 shows a comparison between the two approaches. In addition, we also consider a half-way method presented in [33] where the macros in charge of certifying that an object satisfies the axioms that characterise an algebraic structure were defined (`check-`*S*`-p` macros), but not the functionality to generate generic instances of concrete structures (`defgeneric-`*S* macros). Since the result that we are proving is always the same, there are some figures that are repeated in all the cases (the number of definitions of the cone construction and the auxiliary lemmas).

To sum up, the use of the tools presented in this paper means a great improvement with respect to the other two approaches:

- it solves Problems **P.1**–**P.5** (cf. Section 3) that arise in the other two approaches,
- the amount of definitions and theorems is considerably reduced; then, both the number of lines of a given development and the chance of forgetting some results decrease,
- the developments are more readable thanks to the use of macros, an important issue when documenting a formalisation, and
- the user only has to focus on the difficult parts of the proofs.

We finish this section with a comparison between the ACL2 formalisation of the cone construction and the Coq formalisation of the same result [19]. As we explained in the Introduction, the gap between the ACL2 formalisation and the Kenzo code is much smaller than the one between Coq and Kenzo. In addition, there are several parts of the proof which are automated by ACL2 and, therefore, the user only has to focus on the difficult parts; on the contrary, in the Coq formalisation all the steps must be given by the user. Moreover, in the cases where ACL2 is not able to finish the proof on its own, the user receives feedback from the system, a valuable information that can help him to complete the proof. This shows that our tools made the ACL2 proof comparable to the Coq proof in terms of the user's demands.

7 Conclusions and Further work

In this paper, we have illustrated a methodology to develop tools that simplify the formalisations related to algebraic structures in ACL2. This methodology has been employed to create a hierarchy of algebraic structures — a task, that as far as we are aware, had not been undertaken up to now for this system. The resultant tools facilitate the development of generic theories about algebraic structures; this has been illustrated with several examples requiring different constructions (such as subalgebras, morphisms and quotients) coming from Universal Algebra. We have also shown that it is possible to extrapolate the same ideas to deal with more complex algebraic structures; for example, structures implemented in the Kenzo computer algebra system.

The benefits of using our tools and ideas have been illustrated throughout the paper, and are especially noticeable when working with complex algebraic structures or several instances of a structure. Using these tools, the limitation of ACL2's first-order setting can be overcome thanks to the strengths of this system.

With the acquired experience, the method presented in this paper could be extrapolated to other algebraic structures; for instance, Tarski Kleene Algebras which has been previously studied in Isabelle [6, 23]. We are also interested in the formalisation of the generic theory of Universal Algebra, see [14, 63], but this would require further work in issues like the definition of categories.

In addition, as we have seen in Section 4, the definition of morphism between structures always follows the same pattern; so, it would be desirable to have a tool able to automate, at least part of, the process to generate the tools related to morphisms between structures.

Our main research line for the future is the application of the tools that we have presented here to verify actual computer algebra systems. We are mainly interested in the Kenzo system, where the methodologies and tools presented in this paper can reduce the formalisation effort in works like [44, 47].

Acknowledgements

The authors would like to thank the reviewers for the useful suggestions and comments.

References

1. A. Adams et al. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, 2001.
2. M. Andrés, L. Lambán, J. Rubio, and J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. In *7th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2007)*, pages 34–39, 2007.
3. J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4):271–292, 2008.
4. J. Aransay, C. Ballarin, and J. Rubio. Generating certified code from formal proofs: a case study in homological algebra. *Formal Aspects of Computing*, 22(2):193–213, 2010.
5. J. Aransay and J. Divasón. Formalization and execution of Linear Algebra: from theorems to algorithms. In *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2013)*, Lecture Notes in Computer Science (In Press), 2013.
6. A. Armstrong, G. Struth, and T. Weber. Programming and Automating Mathematics in the Tarski-Kleene Hierarchy. *Journal of Logical and Algebraic Methods in Programming*, 83(2):87–102, 2014.
7. A. Bailey. *The machine-checked literate formalisation of algebra in type theory.* PhD thesis, Manchester University, 1999.
8. C. Ballarin. Algebraic structures in Axiom and Isabelle: attempt at a comparison. In *Programming Languages for Mechanized Mathematics (PLMMS 2007)*, number 07-10 in RISC-Linz Report Series, pages 75–80, 2007.
9. C. Ballarin, J. Aransay, S. Hohe, F. Kammller, and L. Paulson. The Isabelle/HOL Algebra Library, 2013. `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/document.pdf`.

10. C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: an interface between Isabelle and Maple. In *20th International Symposium on Symbolic and Algebraic Computation (ISSAC 1995)*, pages 150–157. ACM PRESS, 1995.
11. A. Bauer, E. M. Clarke, and X. Zhao. Analytica — an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.
12. E. A. Bishop. *Foundations of constructive analysis.* McGraw-Hill Publishing Company, Ltd., 1967.
13. B. Brock. `defstructure` for ACL2 version 2.0. Technical report, Computational Logic, Inc., 1997. `www.cs.utexas.edu/users/moore/publications/others/defstructure-brock.ps`.
14. V. Capretta. Universal Algebra in Type Theory. In *12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *Lecture Notes in Computer Science*, pages 131–148, 1999.
15. P. Castéran and M. Sozeau. A Gentle Introduction to Type Classes and Relations in Coq. Technical report, INRIA, 2014. `http://hal.inria.fr/hal-00702455`.
16. F. Chyzak, A. Mahboubi, T. Sibut-Pinote, and E. Tassi. A computer-algebra-based formal proof of the irrationality of $\zeta(3)$. In *5th International Conference on Interactive Theorem Proving (ITP 2014)*, volume 8558 of *Lecture Notes in Computer Science*, pages 160–176, 2014.
17. K. Denecke and S. L. Wismath. *Universal Algebra and Applications in Theoretical Computer Science.* Chapman Hall/CRC, 2002.
18. M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. In *3rd International Conference on Interactive Theorem Proving (ITP 2012)*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–96, 2012.
19. C. Domínguez and J. Rubio. Computing in Coq with Infinite Algebraic Data Structures. In *17th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus 2010)*, volume 6167 of *Lecture Notes in Artificial Intelligence*, pages 204–218, 2010.
20. C. Domínguez and J. Rubio. Effective Homology of Bicomplexes, formalized in Coq. *Theoretical Computer Science*, 412:962–970, 2011.
21. X. Dousson, J. Rubio, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.
22. A. J. Durán, M. Pérez, and J. L. Varona. Misfortunes of a mathematicians' trio using computer algebra systems: Can we trust? *CoRR*, abs/1312.3270, 2013.
23. S. Foster, G. Struth, and T. Weber. Automated Engineering of Relational and Algebraic Methods in Isabelle/HOL — (Invited Tutorial). In *12th International Conference Relational and Algebraic Methods in Computer Science (RAMICS 2011)*, pages 52–67, 2011.
24. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342, 2009.
25. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4):271–286, 2002.
26. H. Geuvers, F. Wiedijk, J. Zwanenburg, R. Pollack, and H. Barendregt. The "Fundamental Theorem of Algebra" Project. Technical report, 2000. `http://www.cs.kun.nl/gi/projects/fta`.
27. G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *4th International Conference on Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179, 2013.
28. D. Greve. Parameterized Congruences in ACL2. In *6th International Workshop on the ACL2 Theorem Prover and its Applications*, pages 28–34, 2006.
29. E. Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Department of Computer and Information Science, Moore School of Engineering, University of Pennsylvania, 1989. `http://repository.upenn.edu/cis_reports/789/`.
30. F. Haftmann. Haskell-style type classes with Isabelle/Isar. Technical report, Technische Universität München, 2014. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2014/doc/classes.pdf`.

31. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.
32. A. C. Hearn et al. Reduce, 2009. `http://www.reduce-algebra.com/index.htm`.
33. J. Heras. *Mathematical Knowledge Management in Algebraic Topology*, chapter An ACL2 infrastructure to formalize Kenzo Higher Order constructors, pages 293–312. PhD thesis, University of La Rioja, 2011. `http://www.unirioja.es/servicios/sp/tesis/22488.shtml`.
34. J. Heras, F. J. Martín-Mateos, and V. Pascual. Implementing Algebraic Structures in ACL2. Technical report, University of La Rioja, 2012. `http://www.unirioja.es/cu/joheras/ahomsia/`.
35. J. Heras, V. Pascual, and J. Rubio. A certified module to study digital images with the Kenzo system. In *13th International Conference on Computer Aided Systems Theory (EUROCAST 2011)*, volume 6927 of *Lecture Notes in Computer Science*, pages 113–120, 2011.
36. J. Heras, V. Pascual, and J. Rubio. Proving with ACL2 the correctness of simplicial sets in the Kenzo system. In *20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2010)*, volume 6564 of *Lecture Notes in Computer Science*, pages 37–51, 2011.
37. P. Jackson. *Enhancing the Nuprl proof-development system and applying it to computational abstract algebra*. PhD thesis, Cornell University, 1995.
38. R. Jenks and R. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.
39. Journal of Formalized Mathematics. 1990–present. `http://www.mizar.org/JFM/`.
40. C. Kaliszyk and F. Wiedijk. Certified computer algebra on top of an interactive theorem prover. In *14th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus 2007)*, volume 4108 of *Lecture Notes in Computer Science*, pages 94–105, 2007.
41. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
42. M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.
43. M. Kaufmann and J S. Moore. ACL2 version 6.5, 2014. http://www.cs.utexas.edu/users/moore/acl2/.
44. L. Lambán, F. J. Martín-Mateos, J. L. Ruiz-Reina, and J. Rubio. Formalization of a normalization theorem in simplicial topology. *Annals of Mathematics and Artificial Intelligence*, 64(1):1–37, 2012.
45. L. Lambán, V. Pascual, and J. Rubio. Specifying Implementations. In *24th International Symposium on Symbolic and Algebraic Computation (ISSAC 1999)*, ACM Press, pages 245–251, 1999.
46. L. Lambán, V. Pascual, and J. Rubio. An object-oriented interpretation of the EAT system. *Applicable Algebra in Engineering, Communication and Computing*, 14:187–215, 2003.
47. L. Lambán, J. Rubio, F. J. Martín-Mateos, and J. L. Ruiz-Reina. Verifying the bridge between simplicial topology and algebra: the Eilenberg-Zilber algorithm. *Logic Journal of the IGPL*, 22(1):39–65, 2014.
48. P. Manolios and J S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.
49. F. J. Martín-Mateos, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz-Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. In *3rd International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2002)*, pages 188–201, 2002.
50. F. J. Martín-Mateos, J. Rubio, and J. L. Ruiz-Reina. ACL2 verification of simplicial degeneracy programs in the Kenzo system. In *16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus 2009)*, volume 5625 of *Lecture Notes in Computer Science*, pages 106–121, 2009.
51. C. R. F. Maunder. *Algebraic Topology*. Dover, 1996.
52. Maxima, a Computer Algebra system, 2012. `http://maxima.sourceforge.net`.

53. I. Medina-Bulo, F. Palomo-Lozano, and J. L. Ruiz-Reina. A verified Common Lisp implementation of Buchberger's algorithm in ACL2. *Journal of Symbolic Computation*, 45(1):96–123, 2010.
54. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1998)*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366, 1998.
55. F. Pessaux, P. Weia, and D. Doligez. The FoCaLiZe essential. Technical report, 2010. `http://focalize.inria.fr/`.
56. L. Pottier. User contributions in Coq, Algebra. Technical report, 2001. `http://coq.inria.fr/pylons/pylons/contribs/view/Algebra/v8.4`.
57. A. Romero, J. Heras, J. Rubio, and F. Sergeraert. Defining and computing persistent Z-homology in the general case. *CoRR*, abs/1403.7086, 2014.
58. A. Romero and J. Rubio. Homotopy groups of suspended classifying spaces: An experimental approach. *Mathematics of Computation*, 82:2237–2244, 2013.
59. J. Rubio and F. Sergeraert. Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs. University of Genova, 2006.
60. P. Rudnicki, C. Schwarzweller, and A. Trybulec. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32:143–169, 2001.
61. F. Sergeraert. Effective homology, a survey. Technical report, Institut Fourier, 1992. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Survey.pdf`.
62. F. Sergeraert. Common Lisp, Typing and Mathematics. Technical report, Institut Fourier, 2001. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Ezcaray.pdf`.
63. B. Spitters and E. van der Weegen. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science*, 21:795–825, 2011.
64. C. A. Weibel. *An introduction to homological algebra*, volume 38 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1994.
65. X. Yu and J. Hickey. Formalizing Abstract Algebra in Constructive Set Theory. Technical report, California Institute of Technology, 2003. `http://authors.library.caltech.edu/27065/`.
66. R. Zippel. The weyl computer algebra substrate. In *International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO 1993)*, volume 722 of *Lecture Notes in Computer Science*, pages 303–318. 1993.