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

A Hierarchy of Mathematical Structures in ACL2

Jónathan Heras · Francisco-Jesús Martín-Mateos · Vico Pascual

Received: date / Accepted: date

Abstract In this paper, we present a methodology which allows one to deal with *mathematical structures* in the ACL2 theorem prover. Namely, we cope with the representation of mathematical structures, the certification that an object fulfills the axioms characterizing an algebraic structure and the generation of generic theories about concrete structures. As a by-product, an *ACL2 algebraic hierarchy* has been obtained. Our framework has been tested with the definition of *homology groups*, an example coming from Homological Algebra which involves several notions related to Universal Algebra. The method presented here means a great profit, with respect to a from scratch approach, when we work with complex mathematical structures; for instance, the ones coming from Algebraic Topology.

Keywords Mathematical structures, ACL2, Algebraic Hierarchy, Homological Algebra, Proof Engineering

1 Introduction

The implementation of algebraic structures in *theorem proving environments* is a well-known problem; and most of the theorem prover systems offer a set of tools

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 · V. Pascual
Department of Mathematics and Computer Science,
University of La Rioja, Edificio Vives, Luis de Ulloa, s/n. 26004 Logroño, SPAIN
Tel.: +034-941299461
Fax: +034-941299460
E-mail: {jonathan.heras,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
Tel.: +034-954557883
Fax: +034-954556599
E-mail: fjesus@us.es

that can be used to this aim. As a result, several *algebraic hierarchies* have been produced.

These algebraic hierarchies are, in some cases, the foundation for large proof developments. There are several proposals for the Coq system: the CCorn hierarchy [13] based on dependent records and used in the proof of the Fundamental Theorem of Algebra, the SSReflect hierarchy [12] based on packed classes and currently used in the development of the proof of the Feit-Thompson Theorem, and also an approach based on the Coq's type class mechanism [28]; other examples can be found in systems such as Isabelle [11,17], Nuprl [16] and Lego [3].

Nevertheless, up to the best of our knowledge, a similar development had not been undertaken for the ACL2 theorem prover until now. Then, we have tackled the challenge of devising a methodology to deal with mathematical structures in this system. As a result, we have obtained an algebraic hierarchy which allows one to build theories about usual mathematical structures (the hierarchy ranges from setoids to R-modules including structures such as groups or rings) taking advance of the ACL2 capabilities. In addition, this hierarchy is flexible enough to be extended with new structures following our methodology.

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 is devoted to introduce our methodology to deal with mathematical structures and the resultant algebraic hierarchy. As a benchmark to test our approach, the formalization of homology groups is explained in Section 4. The suitability of our method to cope with formalizations related to complex mathematical structures is presented in Section 5, showing the advantages of using our approach instead of working from scratch. The paper ends with a section of conclusions and further work, and the bibliography.

Some basic familiarity with the ACL2 system is necessary to fully understand this paper.

2 A brief introduction to ACL2

ACL2 [18,20] 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. The formulas allowed by the ACL2 system do not have quantifiers and all the variables in them 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. Although this is the only built-in well-founded relation, the user may define new well-founded relations, provided that an ordinal mapping is explicitly given and proved to be monotone.

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, no inconsistencies are introduced by new function definitions. Usually, the system can prove automatically this termination property using 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.

An additional way to introduce new function symbols in the logic is by means of the `encapsulate` mechanism [19]. Instead of giving their definitional body, only certain properties are assumed about them; to ensure consistency, witness local functions 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*, provides a limited higher-order-like reasoning mechanism allowing to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols, 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). See [18] for details.

The ACL2 theorem prover mechanizes the ACL2 logic, being particularly well suited for obtaining mechanical proofs mainly based on simplification and induction. The role of the user in this mechanization 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 that the prover uses mainly as rewrite rules.

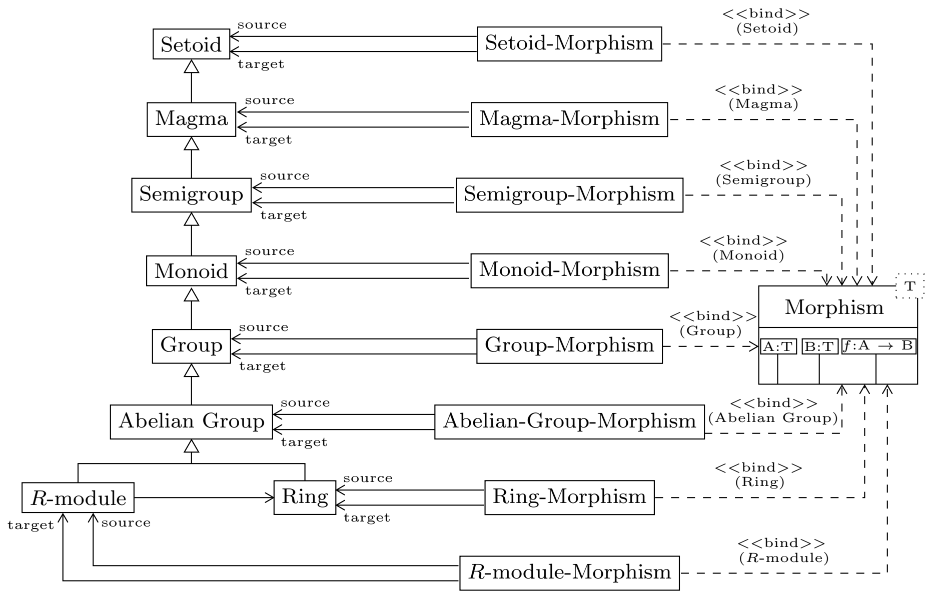
In addition to the built-in inference rules and tools provided by the ACL2 system we extensively use two tools developed by external authors: the defstructure tool [5], and the generic instantiation tool [23].

The defstructure tool provides the ACL2 `defstructure` macro that creates and characterizes general purpose record structures. Thus, this tool provides a convenient way to group and access data, rendering specifications more readable and modular, and it also creates an extensive theory about the creation, access, and update of the defined structures, providing many of the definitions and lemmas that a user will normally need to be able to effectively and automatically reason about ACL2 specifications using structures.

The generic instantiation tool generalizes the functional instantiation rule in ACL2 in such a way that given a “generic theory”, consisting of a set of “independent” functions and assumed properties about them, and a sequence of events (functions definitions and properties) built from the independent functions; then an instance of the whole generic theory can be obtained by means of an instantiation process provided a concrete version of the independent functions satisfying the assumed properties.

We will skip many details and some of the function definitions will be omitted. We urge the interested reader to consult the original and complete source code at `http://www.unirioja.es/cu/joheras/ahomsia/`

Fig. 1 Hierarchy of mathematical structures and morphisms



3 Modeling a hierarchy of algebraic structures in ACL2

In this section a framework to deal with both mathematical structures and morphisms between them is presented. Namely, we have developed a methodology to model the hierarchy depicted in Figure 1 using the *UML* notation [27]. Moreover, our methodology is flexible enough to extend this hierarchy without any special hindrance.

Let us present in a nutshell some remarks about our hierarchy, the concrete details will be provided throughout this section.

In the left side of Figure 1, we have depicted the mathematical structures of our hierarchy ranging from *setoids* to *R-modules*. A mathematical structure can be encoded by means of a structure with several components of functional nature, satisfying the definitional axioms of the intended mathematical structure.

A continuous arrow with an open triangle as tip represents an *inheritance* relationship modeling that the source mathematical structure *is-a* target mathematical structure. 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 one.

The morphisms included in our hierarchy are presented in the right side of Figure 1. It is worth noting that a morphism always consists of a source structure A of type T, a target structure B of type T, and a map $f : A \to B$. Therefore, a morphism can be seen as a *template*, using UML terminology, which is parameterized by a mathematical structure T. An actual morphism is produced by binding the parameter T to one of the concrete mathematical structures of our hierarchy. This is the meaning of dashed arrows in our diagram. As a final remark, the two continuous arrows between a T-morphism and the T-structure describe, as in the case of mathematical structures, a *use* relationship with *role*. Namely, one T-structure plays the *role* of the *source* in the definition of the T-morphism and the other one the *role* of the *target*.

Now, let us present how we implement the hierarchy in ACL2. We start by introducing in detail the implementation of the simplest algebraic structure: *setoid* [4]. Subsequently, we extrapolate the methodology to the rest of mathematical structures and morphisms.

3.1 Modeling setoids in ACL2

A *setoid* $\mathcal{X} = (X, \sim_X)$ is a set X together with an equivalence relation $\sim_X$ on it. Then, 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*.

Therefore, if we are interested in modeling, for instance, the setoid whose underlying set is the set of integer numbers having the same absolute value, we could use the ACL2 `integerp` function as invariant, and the `eq-abs` function as equivalence relation.

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

Moreover, it is necessary to prove the events ensuring that `eq-abs` is an equivalence relation on the set characterized by `integerp`.

```
(implies (integerp x) ;; REFLEXIVE
         (eq-abs x x))

(implies (and (integerp x) (integerp y) (eq-abs x y)) ;; SYMMETRY
         (eq-abs y x))

(implies (and (integerp x) (integerp y) (integerp z) ;; TRANSITIVE
              (eq-abs x y) (eq-abs y z))
         (eq-abs x z))
```

In this way, concrete setoids can be modeled in ACL2. However, this is not enough to work with setoids as in standard mathematical textbooks where, for example, *universal properties* about setoids are proved. In order to tackle this problem, we should use the `encapsulate` mechanism [18]. Namely, using this tool, a *generic setoid* can be defined as follows[1].

```
(encapsulate
   ; SIGNATURES
   (((X-inv *) => *)
    ((X-eq * *) => *))

   ; ASSUMPTIONS
```

[1] witnesses are not included in the `encapsulate`s presented in this paper but they are necessary.

```
   (defthm X-reflexive
      (implies (X-inv x)
               (X-eq x x))

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

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

Now, using the functions which define the generic setoid, we could prove universal properties which, afterwards, could be instantiated for concrete setoids using the *functional instantiation* mechanism [18]. For example, we could prove the following property:

```
(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 concrete setoid defined previously.

Then, this from scratch approach can be applied to deal with setoids in ACL2; nevertheless, from our point of view, several enhancements can be introduced to improve the use of this mathematical structure in ACL2.

First of all, it is worth noting that the use of a structure which gathers the functions encoding the invariant and the equivalence relation of a setoid would model the setoid more accurately than having the functions separately. This can be carried out in ACL2 by means of a record, implemented with the `defstructure` macro [5], with two fields (`inv` and `eq`) which store respectively the names of the invariant function and the intended equivalence relation (in addition, those functions must have been introduced previously). The following `defstructure` construction is used to define the setoid structure.

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

So, using this representation, the setoid whose underlying set is the set of integer numbers having the same absolute value can be encoded as an instance of the `setoid` record, where the values of the `inv` and `eq` slots are respectively the names `integerp` and `eq-abs`. Moreover this instance can be assigned to an ACL2 constant, called `*Zabs*`[2], for latter use in our development.

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

In order to facilitate the statement of the event which ensures the definitional setoid axioms for `setoid` instances, the `setoid-algebraic-structure` function has been defined.

[2] A constant in ACL2 is a symbol beginning and ending with the character `*`

```
(defun setoid-algebraic-structure (setoid)
  (let ((inv (setoid-inv setoid))
        (eq (setoid-eq setoid)))
    `(and (implies (and (,inv x) (,inv y))
                   (booleanp (,eq x y)))
          (implies (,inv x)
                   (,eq x x))
          ...)))
```

This function takes as argument a `setoid` instance and produces a list with the definitional axioms of setoids for the functions of the `setoid` (that is, the `eq` component of the `setoid` instance is Boolean, reflexive, symmetric and transitive on the set characterized by the `inv` component of the `setoid` instance). This function is internally invoked by a macro called `check-setoid-p` which can be used to certify that a concrete `setoid` instance is really a setoid. For example, if `*S*` is a constant storing a `setoid` instance, in the invocation:

```
(check-setoid-p *S*)
```

the macro expands into a call of `defthm` whose name is `*S*-is-a-setoid`. The term generated for the `defthm` is provided by the `setoid-algebraic-structure` function, and, as we have said, such term states that the functions of the setoid instance `*S*` satisfy the setoid definitional axioms. In this way, we have automatized the creation of the event associated with the definitional properties of setoids.

The last enhancement is related to the definition of generic setoids without providing, at least explicitly, an `encapsulate`. This question has been carried out by means of a macro called `defgeneric-setoid`. This macro takes as argument a symbol, for instance `X`, and expands into the following `encapsulate`.

```
(encapsulate
   ; Signatures
   (((X-inv *) => *)
    ((X-eq * *) => *))

   ; Constant Definition
   (defconst *X* (make-setoid :inv 'X-inv :eq 'X-eq))

   ; Setoid Axioms
   (check-setoid-p *X*)
)
```

Therefore, the macro produces, on the one hand, the constant `*X*` which stores a generic setoid; and, on the other hand, the theorem `*X*-is-a-setoid` which ensures that `*X*` satisfies the setoid axioms. Then, this macro makes the creation of generic setoids easier.

3.2 A hierarchy of algebraic structures

Following the same ideas presented for setoids, we have defined a number of algebraic structures in ACL2. In the left side of Figure 1 we have depicted the mathematical structures of our hierarchy with the relations among them. A detailed description of each one of these structures can be seen in [7].

The implementation of the *inheritance* relationship between structures is translated into a definition of the source mathematical structure in terms of the target one; this fact will allow us to reuse several code fragments.

We say that a B structure is defined in terms of an A structure if B is an A structure together with (in some cases) additional operations, $op_1, \dots, op_n$, and satisfying further properties, $P_1, \dots, P_m$. Then, the ACL2 representation of a B structure which is an A structure together with operations $op_1, \dots, op_n$ is:

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

where the value of the `A` field will be an `A` instance, and the values of `op1`, ..., `opn` slots will be function symbols. For instance, a *magma* is a setoid with a binary operation; so, a magma is represented as:

```
(defstructure magma
  setoid binary-op)
```

An instance of the `magma` structure is, for example, the magma whose underlying setoid is the set of integer numbers having the same absolute value (stored previously in the constant `*Zabs*`) and whose binary operation is the addition of integers (`+` in ACL2).

```
(defconst *Magma-Zabs* (make-magma :setoid *Zabs* :binary-op '+))
```

This idea is extrapolated to represent all the structures of our hierarchy. Just a remark, the representation of R-modules follows the same strategy but with a small nuance, the *use* relationship between R-modules and rings which is handled by means of a field in the definition of `R-module` whose value will be a `Ring` instance.

```
(defstructure R-module
  Abelian-Group Ring sc)
```

Now, we have to deal with the statement of the event which ensures the definitional axioms for an instance of a structure. Let us retake the general case of a B structure which is an A structure together with operations $op_1, \dots, op_n$ and satisfying properties $P_1, \dots, P_m$. First, we define the function `B-algebraic-structure`.

```
(defun B-algebraic-structure (B)
  (let ((A (B-A B))
        (op1 (B-op1 B))
        ...
        (opn (B-opn B)))
    (list 'and
       (A-algebraic-structure A)
       (P1 A op1 ... opn)
       ...
       (Pm A op1 ... opn))))
```

where `A-algebraic-structure` is a, previously defined, function which generates a term with the definitional axioms for an `A` structure and `P1`, ..., `Pm` are functions which produce, respectively, the statement of properties $P_1, \ldots, P_m$.

Therefore, the `B-algebraic-structure` function takes a `B` instance as argument and produces a list with the B definitional axioms for that instance. In addition, this function is invoked by a macro called `check-B-p` which has an analogous behavior to the one presented for `check-setoid-p` but for B structures.

As an example, we can consider the magma structure, which is a setoid with a binary operation *closed* on the set characterized by the invariant function of the setoid and *compatible* with the equivalence relation of the underlying setoid. In this case, we have the following functions.

```
(defun closed-op (inv binary-op)
   `(implies (and (,inv x) (,inv y))
             (,inv (,binary-op x y))))

(defun compatible-op (inv eq binary-op)
   `(implies (and (,inv x1) (,inv x2) (,inv y1) (,inv y2)
                  (,eq x1 x2) (,eq y1 y2))
             (,eq (,binary-op x1 y1) (,binary-op x2 y2))))

(defun magma-algebraic-structure (magma)
 (let* ((setoid (magma-setoid magma))
        (inv (setoid-inv setoid))
        (eq (setoid-eq setoid))
        (binary-op (magma-binary-op magma)))
  (list 'and
     (setoid-algebraic-structure setoid)
     (closed-op inv binary-op)
     (compatible-op inv eq binary-op))))
```

The `check-magma-p` macro, which given a constant `*M*`, that stores a `magma` instance, as argument, expands into a call of `defthm` whose name is `*M*-is-a-magma`. The term of this event states the magma definitional axioms for the components of `*M*`. Analogously, this method can be applied in the rest of the structures of our hierarchy.

The last set of tools that we have defined for the mathematical structures of Figure 1 allows us to work with generic instances of them. Namely, we have defined a number of macros called `defgeneric-<structure>` (where `<structure>` is the name of the structure). These macros take as argument a symbol, `X`, and produce, on the one hand, a constant, `*X*`, which stores a generic instance of the structure `<structure>` and, on the other hand, a theorem, `*X*-is-a-<structure>`, which states the definitional axioms of `<structure>` for the generic instance.

It is worth noting that the names of the components of the `<structure>` instance stored in the constant produced by `defgeneric-<structure>` always follow the same convention: `<symbol>-<slot>` where `<symbol>` is the symbol given in the call to `defgeneric-<structure>` and `<slot>` is the name of each one of the slots of `<structure>` (these components are introduced by means of the `encapsulate` principle). For example, given the symbol `M` as argument, the `defgeneric-magma` macro expands into:

```
(encapsulate
   ; SIGNATURES
   (((M-inv *) => *)
    ((M-eq * *) => *)
    ((M-binary-op * *) => *))

   ; CONSTANT DEFINITION
   (defconst *M* (make-magma
                   :setoid (make-setoid :inv 'M-inv :eq 'M-eq)
                   :binary-op 'M-binary-op))

   ; MAGMA AXIOMS
   (check-magma-p *M*)
)
```

Now, we could prove *universal properties* about magmas. For instance, the result which says that 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, will be proved as follows. First, we define a generic magma using the `defgeneric-magma` macro taking `M` as argument. Afterwards, a *generic subset* of M closed with respect to $\circ_M$ is defined using the `encapsulate` principle.

```
(encapsulate
   ; SIGNATURES
   (((N-inv *) => *))

   ; SUBSET OF M
   (defthm subset-of-M
      (implies (N-inv x) (M-inv x)))

   ; CLOSED WITH RESPECT TO ∘M
   (defthm closed-with-respect-to-binary-op
      (implies (and (N-inv x) (N-inv y))
               (N-inv (M-binary-op x y))))
)
```

Now, we can construct a `magma` instance where `N-inv` is the invariant function, `M-eq` is the equivalence relation and `M-binary-op` is the binary operation; and store it in the constant `*N*`.

```
(defconst *N* (make-magma :setoid (make-setoid :inv 'N-inv :eq 'M-eq)
                          :binary-op 'M-binary-op))
```

Eventually, we can certify that `*N*` is really a magma using `check-magma-p`.

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

It is worth noting that ACL2 is able to find the proof of this event without any external help and from now on, we could instantiate this result for concrete magmas. This property about magmas is a particular case of a well known result of Universal Algebra called *Subalgebra criterion* [7]. This criterion says that given

$\mathcal{X} = (X, op_1, \ldots, op_n)$ a T-structure where X is the underlying set of $\mathcal{X}$, and Y is a subset of X closed with respect to $op_1, \ldots, op_n$; then $\mathcal{Y} = (Y, op_1, \ldots, op_n)$ is also a T-structure. This result has been proved for all the structures of our hierarchy following the same schema presented for the magma case.

3.3 Morphisms between algebraic structures

Until now, we have presented how to model algebraic structures in ACL2; now we are going to tackle the task of representing *morphisms* over those mathematical structures as the ones presented in the right side of Figure 1.

In order to deal with morphisms we follow the same strategy presented for mathematical structures. First, we define a record to represent the morphism. It is worth remarking that a morphism consists of: the source structure, S, the target structure, T, and a map $f : S \to T$ (in addition, S and T belong to the same mathematical structure). Therefore, all the morphisms of our hierarchy, unlike what happened in the case of algebraic structures, can be encoded with the pattern:

```
(defstructure <structure>-morphism
  source target map)
```

where `<structure>` is the name of one of the mathematical structures of our hierarchy, the value of both `source` and `target` slots will be `<structure>` instances, and the value of `map` will be a function symbol. So, for instance, the identity setoid morphism on `*Zabs*` is defined as:

```
(defconst *id-Zabs*
  (make-setoid-morphism :source *Zabs* :target *Zabs* :map 'id))
```

where `id` is the identity function.

Following the same schema as in the case of structures, a function in charge of creating the event which ensures the axioms of a morphism between `<structure>`s; called `<structure>-morphism-structure`, is defined.

```
(defun <structure>-morphism-structure (morph)
  (let ((src (<structure>-morphism-source morph))
        (tgt (<structure>-morphism-target morph))
        (map (<structure>-morphism-map morph)))
    (list* 'and
      (<structure>-algebraic-structure src)
      (<structure>-algebraic-structure tgt)
      (<structure>-map-well-defined src tgt map))))
```

`<structure>-algebraic-structure` is a function, like the ones defined in the previous subsection, which provides the term with the axioms for `<structure>` instances, and `<structure>-map-well-defined` produces the statement with the well definition of morphism between `<structure>`s. For example, in the setoid morphisms case, we have the following function.

```
(defun setoid-map-well-defined (S T map)
 (let ((inv-S (setoid-inv S))
       (eq-S  (setoid-eq S))
       (inv-T (setoid-inv T))
       (eq-T  (setoid-eq T)))
   (list (setoid-function-preserves-inv inv-S inv-T map)
       (setoid-function-preserves-equivalence inv-S eq-S eq-T map))))
```

where `setoid-function-preserves-inv` generates the term which claims that `map` is closed and `setoid-function-preserves-equivalence` creates the term which says that `map` preserves the equivalence relation.

Furthermore, we also have defined the macro `check-<structure>-morphism-p`, which internally invokes to `<structure>-algebraic-structure`, whose behavior is analogous to the one presented for the macro `check-<structure>-p` presented in the previous subsection.

Eventually, we also have a macro, called `defgeneric-<structure>-morphism`, to define generic morphism instances between `<structure>`s. If we consider the particular case of setoid morphisms, the call to `defgeneric-setoid-morphism` with the symbol `Mrph` as argument expands into:

```
(encapsulate
  ; SIGNATURES
  (((Mrph-source-inv *) => *)
   ((Mrph-source-eq * *) => *)
   ((Mrph-target-inv *) => *)
   ((Mrph-target-eq * *) => *)
   ((Mrph-map *) => *))

  ; CONSTANT DEFINITION
  (defconst *Mrph*
   (make-setoid-morphism
     :source (make-setoid :inv 'Mrph-source-inv :eq 'Mrph-source-eq)
     :target (make-setoid :inv 'Mrph-target-inv :eq 'Mrph-target-eq)
     :map 'Mrph-map))

  ; SETOID FUNCTION AXIOMS
  (check-setoid-morphism-p *Mrph*)
)
```

To summarize this section, we have defined several tools which facilitate the use of mathematical structures and morphisms in ACL2. First of all, the representation of both structures and morphisms is improved thanks to the records which allow us to define the hierarchy of Figure 1 accurately. In addition, the certification that an object fulfills the definitional axioms of a structure or morphism has been enhanced. The reason is twofold: on the one hand, the generation of the necessary events is automatized with a bunch of macros, and, on the other hand, even if ACL2 is not able to find the proof of the event generated by these macros in the first attempt, the user only has to focus on the trickiest properties since the trivial ones are automatically proved. Eventually, the definition of generic instances of structures and morphisms is reduced to a macro call, making the development of generic theories about those structures and morphisms easier.

4 Application: Homological Algebra

In this section we present an example of the application of our tools to the context of Homological Algebra, an introduction to this mathematical subject can be seen in [29]. In particular, we are going to work with *homology groups*, an important concept in the Homological Algebra setting.

Definition 1 Let $f : G_1 \to G_2$ and $g : G_2 \to G_3$ be abelian group morphisms such that $\forall x \in G1, gf(x) \sim_{G_3} 0_{G_3}$ (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 $\forall x \in G1, gf(x) \sim_{G_3} 0_{G_3}$, known as *nilpotency condition*, makes the above definition meaningful, since $im(f) \subseteq ker(g)$. It is worth noting that this definition involves several constructions of Universal Algebra such as subalgebras, morphisms or quotients; for an introduction to Universal Algebra see [7].

Given $f : G_1 \to G_2$ and $g : G_2 \to G_3$ abelian group morphisms such that $\forall x \in G1, gf(x) \sim_{G_3} 0_{G_3}$; let us present how we can use our framework to define $H_{(f,g)}$ and prove that it is an abelian group.

First of all, we define 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`.

Now, using the `encapsulate` principle we define 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, in this case the abelian group morphism definitional axioms are stated with the `check-abelian-group-morphism-p` macro.

```
(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))))
)
```

Now, let us note that the set $ker(g) = \{x \in G_2 : g(x) \sim_{G_3} 0_{G_3}\}$ (encoded in ACL2 by means of a function called `ker-g-inv`) is both a subset of the underlying set of G_2 and closed with respect to the group operations of G_2; in addition, we have proved the *Subalgebra criterion* for abelian groups (in a similar way to the one presented for the magma case at the end of Subsection 3.2). Therefore, we can instantiate this criterion for our concrete case and define the abelian group $ker(g)$.

```
(defconst *ker-g*
   (make-abelian-group :group
        (make-group :monoid
             (make-monoid :semigroup
                  (make-semigroup :magma
                       (make-magma :setoid
                            (make-setoid :inv 'ker-g-inv :eq 'G2-eq)
                                  :binary-op 'G2-binary-op))
                          :id-elem 'G2-id-elem)
                    :inverse 'G2-inverse))
```

Now, we 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`, which is the way ACL2 provides support for first-order quantification.

```
(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*`, we construct an `abelian-group` instance which encodes the abelian group $im(f)$.

Afterwards, we can tackle the task of defining the homology group $H_{(f,g)}$ as the quotient $ker(g)/im(f)$. Quotienting a structure of our 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 for each one of the structures of our hierarchy following a similar process to the one presented to prove the Subalgebra criterion. In addition, we have proved that $im(f)$ is a subgroup of $ker(g)$, then $im(f)$ induces an equivalence relation on $ker(g)$ which is given in ACL2 by the following definition.

Equivalence Relation: $\forall x, y \in ker(g), x \sim_{im(f)} y \Leftrightarrow xy^{-1} \in im(f)$

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

Therefore, $H_{(f,g)}$ is 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*
  (make-abelian-group :group
       (make-group :monoid
            (make-monoid :semigroup
                 (make-semigroup :magma
                      (make-magma :setoid
                           (make-setoid :inv 'ker-g-inv :eq 'im-f-eq)
                                  :binary-op 'G2-binary-op))
                         :id-elem 'G2-id-elem)
                   :inverse 'G2-inverse))
```

The last step consists in certifying that `*homology-fg*` satisfies the definitional axioms of an abelian group, to this aim the `check-abelian-group-p` macro, taking as argument `*homology-fg*`, is invoked.

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

ACL2 is not able to find the proof of the event generated by the call to this macro in the first attempt and some previous lemmas, suggested by the failed proof, are necessary.

In this way, we have defined the homology group $H_{(f,g)}$ and proved that it is an abelian group.

5 Dealing with complex mathematical structures

The framework that we have presented in the previous sections can be enriched with more complex mathematical structures than the ones introduced up to now. It is in those cases when the profit of using our tools is most remarkable with respect to an ad-hoc approach. Let us illustrate this fact with an example coming from Algebraic Topology [25]; to that end, it is necessary to introduce how we deal with *indexed families of structures* in our context.

5.1 Indexed families of structures

In Algebraic Topology, we do not usually work with just an object of a structure but with a *family* of objects of that structure *indexed* on a set, called the *index set.* This can be seen, for instance, in the definition of two instrumental notions in Algebraic Topology.

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

Let $(C_n, d_n)_{n\in\mathbb{Z}}$ and $(D_n, \widehat{d_n})_{n\in\mathbb{Z}}$ be two chain complexes, a *chain complex morphism* between them is a family of R-module morphism $(f_n)_{n\in\mathbb{Z}}$ such that $\widehat{d_n} f_n = f_{n-1} d_n$ for each $n \in \mathbb{Z}$.

The approach that we have followed to represent indexed families of structures in our framework is based on the one presented in [22]. Roughly speaking, the representation of a graded structure indexed on a set is achieved thanks to the introduction of an additional parameter, which ranges the elements of the index set, in each one of the operations of the structure.

Then, in order to deal with families of structures in our context, we have created a hierarchy of graded structures which is paralleled to the one presented in the left side of Figure 1. The basic object of this hierarchy is *graded setoid* which is defined using the `defstructure` construction with three fields: `inv`, `eq` and `index-sets`. When, we are working with a graded setoid, the value of `inv` and `eq` will be respectively a function symbol, whose arity is 2, representing the underlying graded set of the setoid and a function symbol, whose arity is 3, encoding the intended equivalence relation; and, the value of `index-sets` will be a list with a sole element which is a function name that represents the characteristic function of the index set of the graded setoid.

It is worth noting that we can deal with n-graded setoids (that is to say, a family of setoids indexed on n sets) using the same record structure. In the general case, the arities of `inv` and `eq` functions will be $n + 1$ and $n + 2$ respectively; and, the value of the `index-sets` slot will be a list whose elements are n function names encoding the n characteristic functions of the n sets. Let us note that if the value of `index-sets` is an empty list, we have an object *"equivalent"* to a `setoid` instance as we have presented it in Subsection 3.1.

The ideas presented in Subsection 3.2 to define structures in terms of others can also be applied in the case of graded structures; that is to say, if B is a graded structure defined in terms of an A graded structure, B is an A graded structure together with additional operations and satisfying further properties. Then, the `index-sets` slot will be inherited from the `graded-setoid` structure to the rest of the graded structures of the hierarchy.

In addition to the tools in charge of representing graded structures, we have also defined the functions which generate the events that provide the definitional axioms of the graded structures. Namely, those functions have been defined in order to produce a term depending on the size of the list stored in `index-sets`. Therefore, there are available `check-graded-<structure>-p` macros which behave as `check-<structure>-p` macros but for graded structures.

Furthermore, we also have `defgeneric-graded-<structure>` macros in order to define generic indexed families of `structure`s. The definition of these macros includes a *keyword* parameter called `index-sets` whose value will be a list of function names encoding the characteristic functions of the underlying index sets of the generic indexed family of `structure`s.

Now, we can tackle the task of working with chain complexes in our environment. To this aim, the instrumental notion is the one of graded R-module; this graded structure *is-a* graded abelian group that *uses* a ring as part of its definition; so both the graded hierarchy and the one depicted in Figure 1 are necessary to define graded R-modules. From this graded structure, we can represent chain complexes using the following `defstructure` construction.

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

The value of `graded-R-module` will be a `graded-R-module` instance, let us call it $\mathcal{C}$, indexed on the set of integer numbers, and `diff` will be a function symbol whose arity is 2 encoding the differential map, `diff`: $\mathbb{Z} \times \mathcal{C} \rightarrow \mathcal{C}$; that is, the differential map is *uncurried.* The macro `check-chain-complex-p`, which has been defined following the ideas presented in Section 3, is in charge of verifying that given a `chain-complex` instance, its `graded-R-module` component is really a graded R-module, and `diff` is a family of R-module endomorphism of degree -1 satisfying the nilpotency condition. Besides, we have defined the macro `defgeneric-chain-complex` which produces generic chain complex instances.

Finally, we can focus on the second notion included in Definition 2, *chain complex morphisms.* The record structure to represent chain complex morphisms is

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

where the value of both `source` and `target` slots will be a `chain-complex` instance, 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 structures and morphisms presented in this paper, we have introduced a macro to certify that `chain-complex-morphism` instances are really chain complex morphisms, `check-chain-complex-morphism-p`, and another one, called `defgeneric-chain-complex-morphism`, to generate generic chain complex morphisms. It is worth noting that the notion of chain complex morphism can be generalized to *chain complex morphism of degree q* (a family of R-module morphism $(f_n)_{n\in\mathbb{Z}}$ such that $\widehat{d}_{n+q} f_n = f_{n-1} d_n$ for each $n \in \mathbb{Z}$); in the general case, the representation of chain complex morphism of degree q is the same presented for chain complex morphisms; on the contrary, the macros in charge of verifying that an object is really a chain complex morphism of degree q and generating generic chain complex morphism of degree q take as input an additional argument which is the degree q.

In the next subsection, we are going to present several examples of the usage of the tools related to chain complexes and chain complexes morphisms and the great profit of using them.

5.2 Benefits of our approach

Once that we have introduced how we model chain complexes and chain complex morphisms; let us present the advantages of using our tools in the context of Algebraic Topology by means of the *Cone construction.*

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 $\phi : D_* \rightarrow C_*$ 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: $A_n := C_{n+1} \oplus D_n$; and

$$dA_n := \begin{bmatrix} dC_{n+1} & \phi \\ 0 & -dD_n \end{bmatrix}$$

A more detailed description of this construction can be seen in [26]. The task that we are going to tackle consists in verifying that given a chain complex morphism ϕ, then $Cone(\phi)$ is a chain complex.

Using the tools that have been presented previously, we can define a generic chain complex morphism ϕ.

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

As the rest of this kind of macros, the above macro call produces, on the one hand, the constant `*PHI*` which stores a generic chain complex morphism; and, on the other hand, the theorem which ensures that the components of `*PHI*` satisfy the chain complex morphism axioms.

Afterwards, we introduce the chain complex operations (9 operations are necessary to define a chain complex) associated with the cone construction from the components of `*PHI*`. From them, we create a `chain-complex` instance which is assigned to a new constant, called `*Cone-PHI*`, for latter use.

Subsequently, we use the `check-chain-complex-p` macro with `*Cone-PHI*` as argument to prove an event which ensures the chain complex axioms for the functions of this `chain-complex` instance.

```
(check-chain-complex-p *Cone-PHI*)
```

The system is not able to find the proof of the event generated by this macro in the first attempt. Namely, the event consists of the 49 chain complex definitional axioms, 40 of which are automatically proved by ACL2, and the rest, the trickiest ones, need some auxiliary lemmas suggested by the failed proof.

Finally, in order to make the instantiation of the cone construction for concrete chain complex morphisms easier, we have used the *generic instantiation tool* [23].

In the above development we have taken advantage of the tools presented throughout this paper. However, the same formalization could be performed from scratch, but not without difficulty, as we will see as follows.

First of all, to define the generic chain complex morphism ϕ, the `encapsulate` mechanism should be employed. The definition of this generic object involves 19 function symbols, which define the operations of the chain complex morphism, the corresponding 19 witnesses and 84 axioms which ensures that the 19 function symbols fulfill the properties which characterize the chain complex morphism operations.

Afterwards, from the function symbols of the generic chain complex morphism, we introduce the operations which define the chain complex associated with the cone construction; as we said previously, this means 9 new definitions.

Eventually, we could state the 49 events which claim that the 9 operations introduced in the previous step satisfy the axioms which characterize the chain complex operations. The non trivial events are the same as before, so; in 9 of them some auxiliary lemmas are necessary.

In this way, we can prove that given a chain complex morphism the cone construction produces a chain complex; however, in spite of being able to carry out this task, this way of working is, from our point of view, worse than the one presented using our tools. First of all, in the from scratch approach, there is a considerable amount of definitions and theorems, both in the definition of the generic chain complex morphism and in the certification of the correctness of

Table 1 Comparison between the different approaches

	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 34 auxiliary lemmas
half-way	19 function symbols 19 witnesses 84 axioms	9 definitions 1 `chain-complex`	1 macro call 34 auxiliary lemmas
hierarchical	1 macro call	9 definitions 1 `chain-complex`	1 macro call 34 auxiliary lemmas

the cone construction; so, it is likely that some of them can be forgotten causing unexpected problems. In addition, the functions which represent the operations of the structure are not gathered in any way, then, the structure is not explicitly given as is done when working with pencil and paper or using other theorem provers such as Coq or Isabelle.

Table 1 shows a comparison between the two approaches (the one which uses our tools will be called hierarchical) considering the cone construction. In addition, we also consider a half-way method presented in [14] where the macros in charge of certifying that an object satisfies the axioms which characterize an algebraic structure were defined, but not the functionality to generate generic instances of concrete structures.

As can be noticed, there are some figures which are the same in all the cases (the number of definitions of the cone construction and the auxiliary lemmas), this is due to the fact that the result that we are proving is always the same; so, ACL2 always needs help in the same places. However, the use of the tools presented in this paper means a great improvement with respect to the other approaches. First, the amount of definitions and theorems is considerably reduced; then, both the number of lines of our development and the chance of forgetting some results decrease. Related to the previous fact, it is also worth noting that the developments are more readable thanks to the use of macros, an important issue when we are documenting our work. Besides, the use of the hierarchical methodology presented in this paper makes the work of the user easier because he only has to focus on the difficult parts of the proofs.

As a final remark, we can say that the certification of the cone construction is interesting not only from the point of view of the formalization of a theoretical result but also in the context of program verification. Kenzo [10] 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 neither theoretical or computational means. Therefore, a wide project was launched several years ago to prove the correctness of Kenzo using different theorem proving tools, see [2, 9, 21]. The use of our hierarchy makes the formalization of Kenzo algorithms, as the cone construction, easier; for instance, we can compare our ACL2 development with the Coq formalization of the cone construction presented in [8]. The benefit of using our methodology is twofold: on the one hand, several parts of the proof are automated by ACL2 and the user only has to focus on the difficult parts; whereas in the Coq formalization all the steps must be given by the user;

and, on the other hand, since both ACL2 and Kenzo are Common Lisp systems, the gap between the ACL2 formalization and the Kenzo code is much smaller than the one between Coq and Kenzo.

6 Conclusions and Further work

In this paper, we have presented an ACL2 infrastructure to deal with algebraic structures and morphisms between them. Namely, we have provided several tools which make the handle of this kind of objects easier. As a result, an ACL2 hierarchy of the most common algebraic structures has been provided; a task, that as far as we are aware, had not been undertaken up to now for this system.

The feasibility of using our framework has been illustrated with the example of the homology group of two abelian group morphisms satisfying the nilpotency condition. This example involves several common constructions in Universal Algebra such as subalgebras, morphisms or quotients.

In addition, we have presented the benefits of using our approach when dealing with complex mathematical structures instead of working from scratch. In this kind of problems the development effort is considerably reduced using our tools.

With the acquired experience, the method presented in this paper could be extrapolated to other algebraic structures, for instance to the case of Tarski Kleene Algebras which was previously studied in Isabelle [11]. It is also appealing the idea of formalizing the generic theory of Universal Algebra as was previously done in Coq, see [6,28].

In addition, as we have seen in Section 3, the definition of morphism between structures always follows the same pattern; so, it would be desirable a tool able to automatize, at least part of, the process to generate the tools related to morphisms between structures.

Another interesting research line is the use of our tools to certify critical fragments of the Kenzo system, as we have done for the cone construction. Some previous ACL2 formalizations about first order fragments of Kenzo were presented in [1, 24,15]. However, an environment like the one presented in this paper is desirable when we are dealing with the mathematical structures included in Kenzo.

References

1. Andrés, M., Lambán, L., Rubio, J., Ruiz-Reina, J.L.: Formalizing Simplicial Topology in ACL2. In: Proceedings of ACL2 Workshop 2007, pp. 34–39 (2007)
2. Aransay, J., Ballarin, C., Rubio, J.: A mechanized proof of the Basic Perturbation Lemma. Journal of Automated Reasoning **40**(4), 271–292 (2008)
3. Bailey, A.: The machine-checked literate formalisation of algebra in type theory. Ph.D. thesis, Manchester University (1999)
4. Bishop, E.A.: Foundations of constructive analysis. McGraw-Hill Publishing Company, Ltd. (1967)
5. Brock, B.: `defstructure` for ACL2 version 2.0. Tech. rep., Computational Logic, Inc. (1997)
6. Capretta, V.: Universal Algebra in Type Theory. In: Proceedings 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), *Lecture Notes in Computer Science*, vol. 1690, pp. 131–148 (1999)
7. Denecke, K., Wismath, S.L.: Universal Algebra and Applications in Theoretical Computer Science. Chapman Hall/CRC (2002)

8. Domínguez, C., Rubio, J.: Computing in Coq with Infinite Algebraic Data Structures. In: Proceedings 17th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2010), *Lectures Notes in Artificial Intelligence*, vol. 6167, pp. 204–218. Springer-Verlag (2010)
9. Domínguez, C., Rubio, J.: Effective Homology of Bicomplexes, formalized in Coq. Theoretical Computer Science **412**, 962–970 (2011)
10. Dousson, X., Rubio, J., Sergeraert, F., Siret, Y.: The Kenzo program. Institut Fourier, Grenoble (1998). URL `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`
11. Foster, S., Struth, G., Weber, T.: Automated Engineering of Relational and Algebraic Methods in Isabelle/HOL. In: Proceedings of 12th International Conference Relational and Algebraic Methods in Computer Science (RAMICS'2011), *Lecture Notes in Computer Science*, vol. 6663, pp. 52–67 (2011)
12. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2009), *Lecture Notes in Computer Science*, vol. 5674, pp. 327–342 (2009)
13. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in Coq. Journal of Symbolic Computation **34**(4), 271–286 (2002)
14. Heras, J.: Mathematical Knowledge Management in Algebraic Topology, chap. An ACL2 infrastructure to formalize Kenzo Higher Order constructors, pp. 293–312. Ph.D. thesis, University of La Rioja (2011). `http://www.unirioja.es/servicios/sp/tesis/22488.shtml`
15. Heras, J., Pascual, V., Rubio, J.: Proving with ACL2 the correctness of simplicial sets in the Kenzo system. In: Proceedings of the 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'2010), *Lectures Notes in Computer Science*, vol. 6564, pp. 37–51 (2011)
16. Jackson, P.: Enhancing the Nuprl proof-development system and applying it to computational abstract algebra. Ph.D. thesis, Cornell University (1995)
17. Kammller, F.: Modular Structures as Dependent Types in Isabelle. In: International Workshop, Types for Proofs and Programs (TYPES'98), *Lecture Notes in Computer Science*, vol. 1657, pp. 121–133 (1999)
18. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
19. Kaufmann, M., Moore, J S.: Structured Theory Development for a Mechanized Logic. Journal of Automated Reasoning **26**(2), 161–203 (2001)
20. Kaufmann, M., Moore, J S.: ACL2 version 4.3 (2011). URL `http://www.cs.utexas.edu/users/moore/acl2/`
21. Lambán, L., Martín-Mateos, F.J., Rubio, J., Ruiz-Reina, J.L.: Applying ACL2 to the Formalization of Algebraic Topology: Simplicial Polynomials. In: Proceedings Interactive Theorem Proving (ITP'2011), *Lecture Notes in Computer Science*, vol. 6898, pp. 200–215 (2011)
22. Lambán, L., Pascual, V., Rubio, J.: An object-oriented interpretation of the EAT system. Applicable Algebra in Engineering, Communication and Computing **14**, 187–215 (2003)
23. Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M., Ruiz-Reina, J.: A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. In: Proceedings of the third international ACL2 workshop and its applications, pp. 188–201 (2002)
24. Martín-Mateos, F.J., Rubio, J., Ruiz-Reina, J.L.: ACL2 verification of simplicial degeneracy programs in the Kenzo system. In: Proceedings 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2009), *Lectures Notes in Computer Science*, vol. 5625, pp. 106–121. Springer-Verlag (2009)
25. Maunder, C.: Algebraic Topology. Dover (1996)
26. Rubio, J., Sergeraert, F.: Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs. University of Genova (2006). URL `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Genova-Lecture-Notes.pdf`
27. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Education (1999)
28. Spitters, B., van der Weegen, E.: Type Classes for Mathematics in Type Theory. Mathematical Structures in Computer Science **21**, 795–825 (2011)
29. Weibel, C.A.: An introduction to homological algebra, *Cambridge studies in advanced mathematics*, vol. 38. Cambridge University Press (1994)