ACL2 verification of Simplicial Complexes programs for the Kenzo system

Jónathan Heras and Vico Pascual

Departamento de Matemáticas y Computación, Universidad de La Rioja,
Edificio Vives, Luis de Ulloa s/n, E-26004 Logroño (La Rioja, Spain).
{jonathan.heras, vico.pascual}@unirioja.es

Abstract. Kenzo is a Computer Algebra System devoted to Algebraic Topology, and written in Common Lisp programming language. In spite of being easier the notion of simplicial complex than the notion of simplicial set, the second one is included in Kenzo but not the first one. In this paper, we give the programs which allow us to work with simplicial complexes in the Kenzo system, besides a complete automated proof of the correctness of our programs is provided. The proof is carried out using ACL2, a system for proving properties of programs written in (a subset of) Common Lisp.

1 Introduction

The notion of simplicial complex is the most elementary method to settle a connection between common "general" topology and homological algebra. The "sensible" spaces can be triangulated, at least up to homotopy, and instead of using the notion of topological space, too "abstract", only the spaces having the homotopy type of a CW-complex (see [5]) are considered, and all these spaces in turn have the homotopy type of a simplicial complex. So that a lazy algebraic topologist can decide every space is a simplicial complex.

But many common constructions in topology are difficult to make explicit in the framework of simplicial complexes. It soon became clear in the forties the tricky and elegant notion of simplicial set is much better. The reference [8] certainly remains the basic reference in this subject.

The Kenzo system [3] is a Common Lisp program, developed by F. Sergeraert and devoted to Algebraic Topology. Kenzo works with the main mathematical structures used in Simplicial Algebraic Topology, namely it is able to work with simplicial sets, however the notion of simplicial complex is not included.

In addition, this system was written mainly as a research tool and has got relevant results which have not been confirmed nor refuted by any other means. Then, the question of Kenzo reliability (beyond testing) arose in a natural way. Several works (see [1], [2] and [7]) have focussed on studying the correctness of first order *fragments* of Kenzo with the ACL2 theorem prover [4]. The full verification of the Kenzo system is not possible with ACL2, since the ACL2 logic is first-order but Kenzo uses intensively higher order functional programming.

This paper is devoted to the description of new tools which integrate the notion of simplicial complexes into the Kenzo system. Besides, a certification of the correctness of these programs using the ACL2 theorem prover is provided.

The organization of the rest of the paper is as follows. In Section 2, we introduce the mathematical concepts used in this paper, which are basic notions of Algebraic Topology. In Section 3, a brief introduction to the ACL2 system is given. A description of the main features of the new programs and an example of their use are explained in Section 4. Section 5 is devoted to the description of the ACL2 proof of correctness of the programs presented in Section 4. The paper ends with a section of conclusions and further work.

2 Mathematical definitions

The following definitions about some basic notions of Algebraic Topology, can be found, for instance, in [6].

Definition 1. *A* simplicial complex *C is a finite collection of finite sets of natural numbers, closed under the operation of taking subsets. The elements of the simplicial complex are called* simplices*, which are strictly increasing lists. The* facets *of a simplicial complex are the maximal simplices, which determine the simplicial complex.*

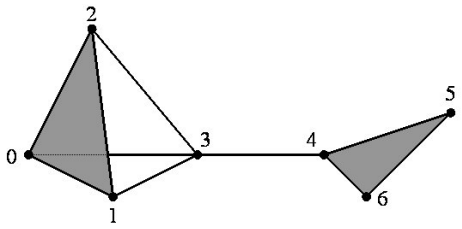

Fig. 1. Butterfly Simplicial Complex

Example 1. The small simplicial complex drawn in Figure 1 is mathematically defined as the object

$$C = \left\{ \begin{array}{l} (0),(1),(2),(3),(4),(5),(6), \\ (0,1),(0,2),(0,3),(1,2),(1,3),(2,3),(3,4),(4,5),(4,6),(5,6), \\ (0,1,2),(4,5,6) \end{array} \right\}$$

In this case, the facets of this simplicial complex are: $\{13, 34, 03, 23, 012, 456\}$, where 012 is a shorthand $(0, 1, 2)$.

Definition 2. *A* Simplicial Set K *is a union* $K = \bigcup_{q \geq 0} K^q$*, where the* K^q *are disjoints sets, together with functions:*

$$\begin{array}{l} \partial_i^q : K^q \to K^{q-1}, \quad q > 0, \quad i = 0, \ldots, q, \\ \eta_i^q : K^q \to K^{q+1}, \quad q \geq 0, \quad i = 0, \ldots, q, \end{array}$$

subject to the relations

$$\begin{array}{lcccl} \partial_i^{q-1}\partial_j^q & = & \partial_{j-1}^{q-1}\partial_i^q & \textit{if} & i < j, \\ \eta_i^{q+1}\eta_j^q & = & \eta_j^{q+1}\eta_{i-1}^q & \textit{if} & i > j, \\ \partial_i^{q+1}\eta_j^q & = & \eta_{j-1}^{q-1}\partial_i^q & \textit{if} & i < j, \\ \partial_i^{q+1}\eta_i^q & = & \textit{identity} & = & \partial_{i+1}^{q+1}\eta_i^q, \\ \partial_i^{q+1}\eta_j^q & = & \eta_j^{q-1}\partial_{i-1}^q & \textit{if} & i > j+1. \end{array}$$

The functions ∂ *and* η *are respectively the* face *and* degeneracy operators.

Definition 3. *Let* C *a simplicial complex. Then the* simplicial set $K(C)$ canonically associated *with* C *is defined as follows. The set* $K^n(C)$ *of* n*-simplices of* C *is the set made of the simplices of cardinality* $n+1$*. In addition, let a simplex* $\{v_0, \ldots, v_q\}$ *the* face *and* degeneracy *operators are defined as follows:*

$$\begin{array}{l} \partial_i(\{v_0, \ldots, v_i, \ldots, v_q\}) = \{v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_q\} \\ \eta_i(\{v_0, \ldots, v_i, \ldots, v_q\}) = \quad \{v_0, \ldots, v_i, v_i, \ldots, v_q\} \end{array}$$

There is here an amusing bug of terminology: the notion of simplicial set, due to Sam Eilenberg, is more complex than the notion of simplicial... complex.

3 An introduction to ACL2

ACL2 [4] stands "for A Computational Logic for an Applicative Common Lisp". ACL2 is a programming language, a logic and a theorem prover. Thus, the system constitutes an environment in which algorithms can be defined and executed, and their properties can be formally specified and proved with the assistance of a mechanical theorem prover.

As a programming language, it is an extension of an applicative subset of Common Lisp[1] [10]. The logic considers every function defined in the programming language as a first-order function in the mathematical sense. For that reason, the programming language is restricted to the applicative subset of Common Lisp. This means, for example, that there is no side-effects, no global variables, no destructive updates and no higher-order programming. Even with these restrictions, there is a close connection between ACL2 and Common Lisp: ACL2 primitives that are also Common Lisp primitives behave exactly in the same way, and this means that, in general, ACL2 programs can be executed in any compliant Common Lisp.

[1] In this paper, we will assume familiarity with Common Lisp.

The ACL2 logic is a first-order logic, in which formulas are written in prefix notation; they are quantifier-free and the variables in it are implicitly universally quantified. The logic includes axioms for propositional logic (with connectives `implies`, `and`,...), equality (`equal`) and those describing the behavior of a subset of primitive Common Lisp functions. Rules of inference include those for propositional logic, equality and instantiation of variables. The logic also provides a principle of *proof by induction* that allows to prove a conjecture splitting it into cases and inductively assuming some instances of the conjecture that are smaller with respect to some wellfounded measure.

An interesting feature of ACL2 is that the same language is used to define programs and to specify properties of those programs. Every time a function is defined with `defun`, in addition to define a program, it is also introduced as an axiom in the logic (whenever it is proved to terminate for every input). Theorems and lemmas are stated in ACL2 by the `defthm` command, and this command also starts a proof attempt in the ACL2 theorem prover.

The main proof techniques used by ACL2 in a proof attempt are simplification and induction. The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, in a deeper sense the system is interactive: very often non-trivial proofs are not found by the system in a first attempt and then it is needed to guide the prover by adding lemmas, suggested by a preconceived hand proof or by inspection of failed proofs. These lemmas are then used as rewrite rules in subsequent proof attempts. This kind of interaction with the system is called "The Method" by its authors.

4 New Kenzo programs

The programs we have developed (with about 150 lines) allow us on the one hand the generation of a simplicial complex from its facets and on the other hand, from a simplicial complex it is possible to build the simplicial set canonically associated with it. Besides, the Kenzo program permits the computation of the homology groups of the simplicial sets define with our program.

4.1 A description of the programs

In this subsection we explain the essential part of these programs, describing the functions with the same format as in the Kenzo documentation [3].

The first step has been to implement the functions which generate a simplicial complex from its facets. The description of the main function in charge of this task is showed here:

simplicial-complex-generator ls

From a list of simplices `ls`, which represents the facets of a simplicial complex, generates the associated simplicial complex.

The second programs, generates from a simplicial complex a simplicial set instance of the Kenzo system class. The description of the main functions is:

simplicial-complex-bspn ***simplicial-complex***
Returns the smaller simplex of *simplicial-complex*.

simplicial-complex-basis ***simplicial-complex***
From the simplicial complex *simplicial-complex*, builds a basis function for the simplicial set canonically associated with *simplicial-complex*. In degree n, the elements of the basis are the simplices of cardinality $n+1$.

simplicial-complex-face
Builds the face function for simplicial complexes. Note that the face function of the simplicial set canonically associated with a simplicial complex is independent from the simplicial complex.

ss-from-sc ***simplicial-complex***
Build the simplicial set canonically associated with a simplicial complex *simplicial-complex*, using the basic functions above.

To provide a better understanding of these new tools, an elementary example of their use is showed in the next subsection.

4.2 A case study: the torus ($S^1 \times S^1$)

As explained in the previous section, the new programs allow us to build the simplicial set canonically associated with a simplicial complex generated from its facets. Thanks to the already programs of the Kenzo system the homology groups of these spaces can be computed. In this subsection a simple example of this computation is presented. In this case, the homology groups are well known and can be obtained without using a computer.

With these programs it is possible to obtain the homology groups of the torus from a triangulation of this space, namely the presented in Figure 2.

The facets of the torus are the triangles of Figure 2, then the torus simplicial complex $S^1 \times S^1$ is generated in Kenzo in the following way:

```
> (setf torus (simplicial-complex-generator
'((0 1 3) (0 1 5) (0 2 4)
  (0 2 6) (0 3 6) (0 4 5)
  (1 2 5) (1 2 6) (1 3 4)
  (1 4 6) (2 3 4) (2 3 5)
  (3 5 6) (4 5 6)))) ✠
((0 1 3) (0 1) (0 1 5) (0 2 4) (0 2) ...)
```

A Kenzo display must be read as follows. The initial '>' is the Lisp prompt of this Common Lisp implementation. The user types out a Lisp statement, here `(setf torus (simplicial-complex-generator ...))` and the maltese cross ✠ (in fact not visible on the user screen) marks in this text the end of the Lisp statement, just to help the reader: the right number of closing parentheses is reached. The Return key then asks Lisp to *evaluate* the Lisp statement. Here the torus is constructed by the function `simplicial-complex-generator`, taking account of the argument `((0 1 3) (0 1 5) (0 2 4) ... )`, and this torus is *assigned* to the

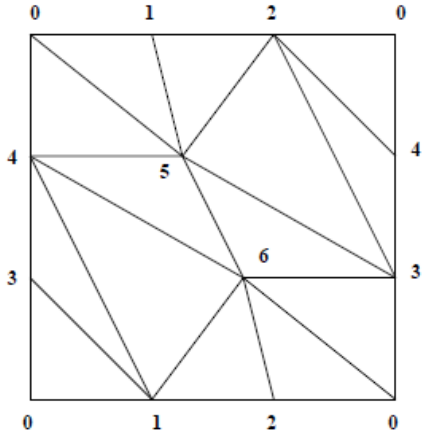

Fig. 2. Torus triangulation

Lisp symbol `torus` for later use. Also evaluating a Lisp statement *returns* an object, the result of the evaluation, in this case the Lisp object implementing the torus.

Subsequently, the simplicial set canonically associated with the torus simplicial complex can be built in Kenzo in the following way:

```
> (setf torus-ss (ss-from-sc torus)) ✠
[K1 Simplicial-Set]
```

Finally, the homology groups of this space can be computed:

```
> (homology torus-ss 0 3) ✠
Homology in dimension 0:
Component Z
Homology in dimension 1:
Component Z
Component Z
Homology in dimension 2:
Component Z
```

To be interpreted as stating $H_0(torus) = \mathbb{Z}$, $H_1(torus) = \mathbb{Z} \oplus \mathbb{Z}$ and $H_2(torus) = \mathbb{Z}$.

5 From Kenzo programs to ACL2

As explained in Section 4, two different programs have been developed, a program to generate simplicial complexes from their facets and another one which builds a simplicial set from a simplicial complex. In this section the certification of the correctness of these programs by means of the ACL2 system is explained.

5.1 simplicial-complex-generator generates simplicial complexes

The first task we have undertaken has consisted of certifying that the function in charge of generating a simplicial complex from a list of simplices really generates a simplicial complex.

Firstly, the functions implemented for the Kenzo system must be converted into ACL2 functions. Fortunately, in this case the functions developed for the Kenzo system can also be introduced in ACL2 without changes. So, to certify that the Kenzo programs are correct, which means that from a "sensible" input the output is a simplicial complex, two main theorems must be proved:

```
(defthm simplicial-complex-generator-theorem-1
 (implies (list-of-simplex-p ls)
          (list-of-simplex-p (simplicial-complex-generator ls))))

(defthm simplicial-complex-generator-theorem-2
 (implies (and (simplex-p s1)
               (simplex-p s2)
               (list-of-simplex-p ls)
               (member-equal s1 (simplicial-complex-generator ls))
               (subset-p s2 s1))
          (member-equal s2 (simplicial-complex-generator ls))))
```

The description of the functions and the meaning of the theorems is showed here:

simplex-p ***simplex*** *Function*
Returns `t` if *simplex* is an increasing list of naturals, otherwise `nil`.

list-of-simplex-p ***list-of-simplices*** *Function*
Returns `t` if *list-of-simplices* is a list of simplices, otherwise `nil`.

member-equal ***simplex list-of-simplices*** *Function*
Returns `t` if *simplex* is an element of *list-of-simplices*, otherwise `nil`.

subset-p ***simplex1 simplex2*** *Function*
Returns `t` if *simplex1* is a subset of *simplex2*, otherwise `nil`.

simplicial-complex-generator ***list-of-simplices*** *Function*
Generates the simplicial complex which has as facets *list-of-simplices*.

simplicial-complex-generator-theorem-1 *Theorem*
This theorem asserts that if the function `simplicial-complex-generator` is applied over a list of simplices a new list of simplices is generated.

simplicial-complex-generator-theorem-2 *Theorem*
This theorem validates that if a simplex belongs to a simplicial complex, then all the subsets of that simplex also belong to the simplicial complex.

We now describe how we state the main theorems about the correctness of the function which generates a simplicial complex from its facets.

The algorithm `simplicial-complex-generator` can be decomposed in three steps: firstly, the function generates recursively the simplicial complex associated with each one of the simplices of the input list of simplices; subsequently the program generates a list of simplices containing all the obtained simplices; finally, the elements that are duplicated are removed from the output list of simplices.

The ACL2 proof has followed the same schema used for implementing that function. First of all, two theorems in the same line that the two above presented, are proved to certify that the output of the function which generates a simplicial complex from a simplex is really a simplicial complex. Subsequently, the same results are proved for the function which gathers all the simplicial complexes obtained from the recursive generation of simplicial complex from a list of simplices. Finally, the looked for theorems are obtained removing the duplicate elements.

In this way, we can claim that the function `simplicial-complex-generator` really generates a simplicial complex from a list of simplices.

5.2 ss-from-sc builds simplicial sets

In order to finish the certification of our programs, it is necessary to provide a proof which verifies the correctness of the function `ss-from-sc`. This proof consists of verifying that taking as input a simplicial complex this functions returns a representation of a simplicial set.

This is a tricky task, the main reason is that the output of the function `ss-from-sc` is an instance of a CLOS class where the slots of the class determine the simplicial set, and this kind of objects can not be represented in the ACL2 system.

Instead of using this Kenzo representation for determining simplicial sets, the following specification for a simplicial set *representation* is provided:

```
inv:         nat U    -> bool
face:        nat nat U -> U
degeneracy: nat nat U -> U
```

The degeneracy operator is always defined in the same way in the Kenzo system because it is independent from the simplicial set; however, the face and inv (which is the invariant of the underlying set) operators are defined from the slots of the simplicial set instance.

If these face and degeneracy operators are well-defined, which means that let $x \in K^q$ then $\partial_i^q x \in K^{q-1}$ with $q > 0$ and $i = 0, \ldots, q$ and $\eta_i^q x \in K^{q+1}$ with $q \geq 0$ and $i = 0, \ldots, q$ (these questions are validated by means of the inv function); and they satisfy the commuting relations of Definition 2; then these operators determines a simplicial set.

The methodology followed here to prove the desired result is the same that was presented in [7]. The main idea of that methodology consists of using EAT [9] (the predecessor of Kenzo) as the main component of the specification of the intended properties. The EAT approach for representing the functions of simplicial sets is clean and comprehensible due to the natural way of representing these functions, namely the degeneracy lists are encoded as decreasing lists of natural numbers (more details about this codification can be found in [7]). On the contrary, the Kenzo system codifies the degeneracy lists as natural numbers, a way of working which improves dramatically the performance of the system

but the algorithms in the Kenzo system become obscured. Thus, EAT, which is logically simpler (i.e, easier to be verified) but less efficient than Kenzo, acts as a mathematical model and then our Kenzo program for building simplicial sets from simplicial groups is formally verified against it.

Due to the big jump between the EAT and the Kenzo codifications, the technique presented in [7] makes explicit an intermediary representation based on binary lists in order to prove the equivalence between EAT and Kenzo functions through the intermediary representation. Our proofs also use this technique; thus, the first step of our work has consisted of verifying the following equivalences module the change of representation between the different operators of each codification:

face-eat	$\equiv$	*face-binary*	$\equiv$	face-kenzo
degeneracy-eat	$\equiv$	degeneracy-binary	$\equiv$	degeneracy-kenzo
inv-eat	$\Leftrightarrow$	inv-binary	$\Leftrightarrow$	inv-kenzo

Subsequently, the second question to be solved was the verification of the simplicial set properties for the functions which codifies the face, the degeneracy and the invariant operators; that is the well defining of the face and degeneracy operator and the commuting relations of this operators state in Definition 2. For instance, the third commuting relation ($\eta_{j-1}^{q-1}\partial_i^q = \partial_i^{q+1}\eta_j^q$ when $i < j$) is expressed with the face operators as follows:

```
(defthm property-3-eat
  (implies (and (< i j)
                (inv-eat sc q absm))
           (equal (degeneracy-eat sc (1- j) (1- q)
                                  (face-eat sc i q absm))
                  (face-eat sc i (1+ q)
                            (degeneracy-eat sc j q absm)
                  ))))
```

Finally, gathering the equivalence between the operators module the change of representation and the properties verified with the EAT codification the looked for properties of our Kenzo operators can be obtained. For instance, the same property showed for the EAT representation is defined in terms of Kenzo operators as follows:

```
(defthm property-3-kenzo
  (implies (and (< i j)
                (inv-kenzo sc q absm))
           (equal (degeneracy-kenzo sc (1- j) (1- q)
                                    (face-kenzo sc i q absm))
                  (face-kenzo sc i (1+ q)
                              (degeneracy-kenzo sc j q absm)
                  ))))
```

From these theorems, we can assert that taking as input a simplicial complex the function `ss-from-sc` returns a representation of a simplicial set.

6 Conclusions and Further Work

In this paper, we have presented some programs that improve the functionality of Kenzo, generating simplicial complexes from its facets and building the simplicial sets canonically associated with simplicial complexes. In spite of not being too complex programs, the great advantage with respect to the Kenzo system is the certification of the correctness of our algorithms using the ACL2 theorem prover.

As future work, two different tasks can be undertaken. On the one hand, the development of new algorithms for the Kenzo system. On the other hand, the verification of already implemented algorithms of Kenzo. Besides, in the same way explained in this paper these two lines can be gathered into one to certify new algorithms for the Kenzo system.

References

1. Andres M., Lamban L. and Rubio J., *Executing in Common Lisp, Proving in ACL2*, In *Proceedings of Calculemus 2007*, Lecture Notes in Artificial Intelligence, 4573 (2007) 1-12.
2. Andres M., Lamban L., Rubio J. and Ruiz-Reina J. L., *Formalizing Simplicial Topology in ACL2,*, In *Proceedings of ACL2 Workshop 2007*, (2007) 34–39.
3. Dousson X., Sergeraert F., Siret Y., *The Kenzo program*, Institut Fourier, Grenoble, 1999. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.
4. Kaufmann, M., Manolios P. and Moore, J., *Computer-Aided Reasoning: An Approach. Kluwer Academic Press*, Boston, 2000.
5. Lundell A. T. and Weingram S., *The topology of CW complexes*, Van Nostrand, 1969.
6. Mac Lane S., *Homology*, Springer, 1994.
7. Martín-Mateos F.J., Rubio J. and Ruiz-Reina J.L., *ACL2 verification of simplicial degeneracy programs in the Kenzo system*, In *Proceedings of Calculemus 2009*, Lecture Notes in Computer Science, 5625 (2009) 106–121.
8. May J.P., *Simplicial Objects in Algebraic Topology*, Van Nostrand, 1967.
9. Rubio J., Sergeraert F., Siret Y., *EAT: Symbolic Software for Effective Homology Computation*, Institut Fourier, 1997. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/EAT-program.zip`.
10. Steele G. L. Jr., *Common Lisp The Language*, 2nd edition, Digital Press, 1990.