Proving with ACL2 the correctness of simplicial sets in the Kenzo system.*

Jónathan Heras, Vico Pascual, and Julio Rubio

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, julio.rubio}@unirioja.es

Abstract. Kenzo is a Common Lisp system devoted to Algebraic Topology. Although Kenzo uses higher-order functional programming intensively, we show in this paper how the theorem prover ACL2 can be used to prove the correctness of first order fragments of Kenzo. More concretely, we report on the verification in ACL2 of the implementation of simplicial sets. By means of a generic instantiation mechanism, we achieve the reduction of the proving effort for each family of simplicial sets, letting ACL2 automate the routine parts of the proofs.

1 Introduction

Kenzo [4] is a computer algebra system devoted to Algebraic Topology and Homological Algebra calculations. It was created by Sergeraert following his ideas about *effective homology* [13], and has been successful in the sense that Kenzo has been capable of computing previously unknown homology groups. This implies that increasing user's trust in the system is relevant. To this aim, a wide project to apply formal methods in the study of Kenzo was launched several years ago.

One feature of Kenzo is it uses higher order functional programming to handle spaces of infinite dimension. Thus, the first attempts to apply theorem proving assistants in the analysis of Kenzo were oriented towards higher order logic tools. Concretely, the Isabelle/HOL proof assistant was used to verify a very important algorithm in Homological Algebra: the Basic Perturbation Lemma (see [2]). Let us note, however, that this formalization was related to *algorithms* and not to the real *programs* implemented in Kenzo. The problem of extracting programs from the Isabelle/HOL proofs has been dealt with in [3], but even there the programs are generated in ML, far from Kenzo. Because Kenzo is programmed in Common Lisp.

In this paper, we report on a verification on some fragments of the real Kenzo code, by means of the ACL2 theorem prover [7]. ACL2 works with first order logic. Hence, in a first step, Kenzo first order fragment are dealt with. A

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Community FP7, STREP project ForMath.

previous work on this area was published in [10], where the programs to deal with degeneracies in Kenzo were proven correct by means of ACL2.

To understand another source of first order fragments of Kenzo, let us explain the way of working with the system. As a first step, the user constructs some initial spaces (under the form of *simplicial sets* [11]) by means of some built-in Kenzo functions; then, in a second step, he constructs new spaces by applying topological constructions (as Cartesian products, loop spaces, and so on); as a third, and final, step, the user asks Kenzo for computing the homology groups of the spaces. The important point for our discussion is that only steps 2 and 3 need higher-order functional programming. The first step, the construction of *constant* spaces, can be modeled in a first order logic. Thus, this paper is devoted to an ACL2 infrastructure allowing us to prove the correctness of Kenzo programs for constructing these constant simplicial sets.

We started that work by examining each family of constant simplicial sets in Kenzo (spheres, Moore spaces, Eilenberg-MacLane spaces, and so on). By doing the first ACL2 proofs, it became quickly clear that all the proofs match a common pattern. So, we carefully analyzed which is the common part for all the constant spaces, and which is particular for each family of simplicial sets. This led us to Theorem 3 in Subsection 3.2. The proof of this theorem in ACL2 allows us to use the generic instantiation tool by Martín-Mateos et al. [9], to produce ACL2 proofs for concrete families of constant simplicial sets. This has been done, up to now, for the spheres and for the simplicial sets coming from simplicial complexes. It is worth noting that from 4 definitions and 4 theorems the instantiation tool generates (and instantiates) 15 definitions and 375 theorems. The benefits in terms of proof effort are considerable. These two issues are the main contributions of this paper.

The unique difference between actual code and ACL2 verified code is the transformation of loops into ACL2 recursive functions. Since this transformation is very well-known and quite safe, we consider that the verified code is as close as possible to the real Kenzo programs.

The methodological approach has been imported from [10]. EAT [12] was the predecessor of Kenzo. The EAT system is also based on Sergeraert's ideas, but its Common Lisp implementation was closer to the mathematical theory. This means a poorer performance (since in Kenzo the algorithms have been optimized), but also that the ACL2 verification of EAT programs is easier. Thus, the main idea is to prove first in ACL2 the correctness of EAT programs, and then, by a domain transformation, to translate the proofs to Kenzo programs. To this aim, an intermediary model, based on binary numbers, is employed (this was introduced in [10], too).

The complete ACL2 code of our formalization can be found at [5].

The organization of the paper is as follows. Section 2 is devoted to mathematical preliminaries and to their concrete materialization in the EAT and Kenzo systems. The main theorems to factorize the proofs which are common to all the spaces are presented in Section 3 (both in the EAT and in the Kenzo models).

Technical issues about the ACL2 proofs are dealt with in Section 4. The paper ends with conclusions, future work and the bibliography.

2 Simplicial models

In this section, we present the minimal mathematical preliminaries to understand the rest of the paper, and we explain how the elementary data structures are encoded in both systems EAT and Kenzo.

2.1 A mathematical model

The following definition is the most important one in this work (see [11] for the context and further details).

Definition 1 A *simplicial set* K, is a union $K = \bigcup_{q\geq 0} K^q$, where the K^q are disjoints sets, together with functions:

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

subject to the relations:

$$\begin{array}{llcll} (1)\ \partial_i^{q-1}\partial_j^q & = & \partial_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (2)\ \eta_i^{q+1}\eta_j^q & = & \eta_j^{q+1}\eta_{i-1}^q & \text{if} & i > j, \\ (3)\ \partial_i^{q+1}\eta_j^q & = & \eta_{j-1}^{q-1}\partial_i^q & \text{if} & i < j, \\ (4)\ \partial_i^{q+1}\eta_i^q & = & identity & = & \partial_{i+1}^{q+1}\eta_i^q, \\ (5)\ \partial_i^{q+1}\eta_j^q & = & \eta_j^{q-1}\partial_{i-1}^q & \text{if} & i > j+1, \end{array}$$

The functions ∂_i^q and η_i^q are called *face* and *degeneracy* maps, respectively.

The elements of K^q are called *q-simplexes*. A q-simplex x is *degenerate* if $x = \eta_i^{q-1} y$ for some simplex y, and for some degeneracy map η_i^{q-1}; otherwise x is *non degenerate*.

In the rest of the paper a non degenerate simplex will be called *geometric* simplex, to stress that only these simplexes really have a geometric meaning; the degenerate simplexes can be understood as *formal* artifacts introduced for technical (combinatorial) reasons. This becomes clear in the following discussion.

The next essential result, which follows from the commuting properties of degeneracy maps in Definition 1, was modeled and proved by means of the ACL2 theorem prover in [1].

Proposition 1 Let K be a simplicial set. Any n-simplex $x \in K^n$ can be expressed in a unique way as a (possibly) iterated degeneracy of a non-degenerate simplex y in the following way:

$$x = \eta_{j_k} \ldots \eta_{j_1} y$$

with $y \in K^r$, $k = n - r \geq 0$, and $0 \leq j_1 < \cdots < j_k < n$.

In the previous statement the super-indexes in the degeneracy maps have been skipped, since they can be inferred from the context. It is a usual practice and will be freely used in the sequel, both for degeneracy and for face maps.

This proposition allows us to encode all the elements (simplexes) of *any* simplicial set in a generic way, by means of a structure called *abstract simplex*. More concretely, an *abstract simplex* is a pair $(dgop \quad gmsm)$ consisting of a sequence of degeneracy maps $dgop$ (which will be called a *degeneracy operator*) and a geometric simplex $gmsm$. The indexes in a degeneracy operator $dgop$ must be in strictly decreasing order. For instance, if σ is a non-degenerate simplex, and σ' is the degenerate simplex $\eta_1\eta_2\sigma$, the corresponding abstract simplexes are respectively $(\emptyset \quad \sigma)$ and $(\eta_3\eta_1 \quad \sigma)$, as $\eta_1\eta_2 = \eta_3\eta_1$, due to equality (2) in Definition 1. Of course, the nature of geometric simplexes depends on the concrete simplicial set we are dealing with, but the notion of abstract simplex allows us a generic handling of all the elements in our proofs.

Equation (2) in Definition 1 allows one to apply a degeneracy map η_i over a degeneracy operator $dgop$ to obtain a new degeneracy operator. Let us consider, for example, η_4 and the degeneracy operator $\eta_5\eta_4\eta_1$; then $\eta_4\eta_5\eta_4\eta_1 = \eta_6\eta_4\eta_4\eta_1 = \eta_6\eta_5\eta_4\eta_1$. We will use the notation $\eta_i \circ dgop$ for the resulting degeneracy operator; in our example: $\eta_4 \circ (\eta_5\eta_4\eta_1) = \eta_6\eta_5\eta_4\eta_1$.

We can also try to apply a face map ∂_i over a degeneracy operator $dgop$. But now, there are two cases, according to whether the indexes i or $i-1$ appear in $dgop$. If they do not appear, then there is a face that *survives* in the process (for instance: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). Otherwise, the cancellation equation (4) from Definition 1 applies and the result of the process is simply another degeneracy operator (example: $\partial_4\eta_5\eta_3\eta_0 = \eta_4\partial_4\eta_3\eta_0 = \eta_4\eta_0$). We will denote by $\partial_i \circ dgop$ the output degeneracy operator, in both cases (in our examples: $\partial_4 \circ (\eta_5\eta_2\eta_0) = \eta_4\eta_2\eta_0$ and $\partial_4 \circ (\eta_5\eta_3\eta_0) = \eta_4\eta_0$).

With these notational conventions, the behaviour of face and degeneracy maps over abstract simplexes is characterized as follows:

$$\eta_i^q(dgop \quad gmsm) := (\eta_i^q \circ dgop \quad gmsm)$$
$$\partial_i^q(dgop \quad gmsm) := \begin{cases} (\partial_i^q \circ dgop \quad gmsm) & \text{if } \eta_i \in dgop \vee \eta_{i-1} \in dgop \\ (\partial_i^q \circ dgop \quad \partial_k^r gmsm) & \text{otherwise;} \end{cases}$$

where

- $r = q - \{\text{number of degeneracies in } dgop\}$ and
- $k = i - \{\text{number of degeneracies in } dgop \text{ with index lower than } i\}$[1].

Note that the degeneracy map expressed in terms of abstract simplexes only affects the degeneracy operator of the abstract simplex; therefore degeneracy maps can be implemented independently from the simplicial set. On the contrary,

[1] In fact, we are still abusing the notation here, since the face of a geometric simplex is an abstract simplex and, sometimes, a degenerate one; this implies that to get a correct representation of $(\partial_i^q \circ dgop \quad \partial_k^r gmsm)$ as an abstract simplex $(dgop' \quad gmsm')$ we should compose the degeneracy operator $\partial_i^q \circ dgop$ with that coming from $\partial_k^r gmsm$.

face maps can depend on the simplicial set because, when $\eta_i \notin dgop$ and $\eta_{i-1} \notin dgop$, the application of a face map ∂_i *arrives* until the geometric simplex, and, this, of course, requires some knowledge from the concrete simplicial set where the computation is carried out.

This observation will be very important in the sequel, since it indicates which parts of the proofs could be automatized (those independent from the concrete simplicial sets), and which parts must be explicitly provided by the user.

Furthermore, it is necessary to characterize the pattern of the admissible abstract simplexes for a given simplicial set, since it will allow us to determine over which elements the proofs will be carried out. The following property gives such a characterization.

Proposition 2 Let K be a simplicial set and $absm = (dgop \;\; gmsm)$ be a pair where $dgop$ is a degeneracy operator and $gmsm$ is a geometric simplex of K. Then, $absm$ is an abstract simplex of K in dimension q if and only if the following properties are satisfied:

1. $gmsm \in K^r$, for some natural number $r \leq q$;
2. the length of the sequence of degeneracies $dgop$ is $q - r$;
3. the index of the first degeneracy in $dgop$ is lower than q.

Each concrete representation for degeneracy operators defines a different model to encode elements of simplicial sets. In the following two subsections we will explain the EAT and the Kenzo models, respectively.

2.2 The EAT model

An abstract simplex, $absm$, is represented internally in the EAT system by a Lisp object: $(dgop \;\; gmsm)$ where $dgop$ is a strictly decreasing integer list (a *degeneracy list*) which represents a sequence of degeneracy maps, and $gmsm$ is a geometric simplex (whose type is left unspecified). For example, if we retake the examples introduced in the previous subsection $(\emptyset \;\; \sigma)$ and $(\eta_3\eta_1 \;\; \sigma)$, the corresponding EAT objects are respectively $(nil \;\; \sigma)$ and $((3 \;\; 1) \;\; \sigma)$, where nil stands for the empty list in Lisp.

Now, we can implement an invariant function in Lisp, which is a predicate indicating when a Lisp pair is an abstract simplex in EAT. The invariant translates the conditions from Proposition 2:

1. $gmsm$ is an element of K^r (this information depends on K and must be implemented for each simplicial set);
2. the length of the $dgop$ list is equal to $q - r$;
3. the first element of $dgop$ is lower than q.

With respect to the EAT representation of a simplicial set, it is based on considering a simplicial set as a tuple of functional objects. Since the degeneracy maps do not depend on the simplicial set, only two functional slots are needed: one for recovering faces (on *geometric* simplexes), and other with information about the encoding of geometric simplexes.

As mentioned previously, we want to focus on the simplicial sets of the Kenzo system. The EAT model has been used as a simplified formal model to reduce the gap between the mathematical structures and their Kenzo representations.

2.3 The Kenzo model

Both EAT and Kenzo systems are based on the same Sergeraert's ideas, but the performance of the EAT system is much poorer than that of Kenzo. One of the reasons why Kenzo performs better than EAT is because of a smart encoding of degeneracy operators. Since generating and composing degeneracy lists are operations which appear in an exponential way in most of Kenzo calculations (through the Eilenberg-Zilber theorem [11]), it is clear that having a better way for storing and processing degeneracy operators is very important. But, no reward comes without a corresponding price, and the Kenzo algorithms are somehow obscured, in comparison to the clean and comprehensible EAT approach.

An abstract simplex, *absm*, is represented internally in the Kenzo system by a Lisp object: (*dgop* *gmsm*) where *dgop* is a non-negative integer coding a strictly decreasing integer list and *gmsm* is a geometric simplex. The strictly decreasing integer list represents a sequence of η operators and is coded as a *unique* integer. Let us explain this with an example: the degeneracy list (3 1) can equivalently be seen as the binary list (0 1 0 1) in which 1 is in position i if the number i is in the degeneracy list, and 0 otherwise. This list, interpreted as a binary number in the reverse order, defines the natural number 10. Thus, Kenzo encodes the degeneracy list (3 1) as the natural number 10. The empty list is encoded by the number 0. Then, the abstract simplexes ($\emptyset$ σ) and ($\eta_3\eta_1$ σ) are implemented in the Kenzo system as (0 σ) and (10 σ), respectively.

With this representation, we will say that an index is an *active bit* in a natural number representing a degeneracy operator if the index appears in the degeneracy operator.

With this representation and notation, the invariant function for abstract simplexes in Kenzo can be defined according to:

1. *gmsm* is an element of K^r;
2. the number of active bits in *dgop* is equal to $q - r$;
3. $dgop < 2^q$.

As Kenzo encodes degeneracy lists as integers, the face and degeneracy maps can be implemented using very efficient Common Lisp primitives dealing with binary numbers (such as *logxor*, *ash*, *logand*, *logbitp* and so on). This is one of the reasons why Kenzo dramatically improves the performance of its predecessor EAT. Nevertheless, these efficient operators have a more obscure semantics than their counterparts in EAT.

The Kenzo representation of a simplicial set follows the same pattern that the EAT one, that is, a simplicial set is a tuple of functional λ-expressions.

In order to establish an infrastructure to prove the objects handled in Kenzo as simplicial sets are *really* simplicial sets (in other words, they satisfy the equations in Definition 1), the following strategy has been followed (inspired from that of [10]). First, the correctness of the EAT representation will be proven. In a second step, a proof of the correctness of the domain transformations between the EAT and Kenzo representations will be built. Then, it will become easy to prove a property about a Kenzo operator by first proving the property about the EAT one (which is usually much simpler) and then translating it to Kenzo, by means of the domain transformations theorems. These tasks have been fulfilled by means of the ACL2 system, as reported in the next section.

3 Schema of the proof

3.1 Proving that EAT objects are Simplicial Sets

The first task consists in proving that the face and degeneracy operators are well defined. Let $absm$ be an abstract simplex belonging to K^q; then the face and degeneracy EAT operators must satisfy: (i) $\eta_i^q absm \in K^{q+1}$; (ii) $\partial_i^q absm \in K^{q-1}$. As the definition of the degeneracy maps over abstract simplexes is independent from the simplicial set, so is the proof of its correctness. On the contrary, the face map invariance must be proven for each particular object.

Then, as a second task, the properties stated in Definition 1 must be accomplished by the face and degeneracy maps.

As no additional information from the particular object is needed, some important equalities can be obtained for every simplicial set:

Theorem 1 Let $absm$ be an abstract simplex. Then:

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

Then all properties of Definition 1 are proven without using a particular simplicial set except relation (1). On the contrary, we must require that the particular simplicial set satisfies some properties in order to obtain the proof of the first property of Definition 1. These required conditions have been characterized by means of the following result.

Theorem 2 Let $\mathcal{E}$ be an EAT object implementing a simplicial set. If for every natural number $q \geq 2$ and for every geometric simplex $gmsm$ in dimension q the following properties hold:

1. $\forall i, j \in \mathbb{N} : i < j \leq q \Longrightarrow \partial_i^{q-1}(\partial_j^q gmsm) = \partial_{j-1}^{q-1}(\partial_i^q gmsm)$,
2. $\forall i \in \mathbb{N}$, $i \leq q$: $\partial_i^q gmsm$ is a simplex of $\mathcal{E}$ in dimension $q-1$,

then:

$$\mathcal{E} \text{ is a simplicial set.}$$

This theorem has been proven in ACL2, by using in particular the `cmp-d-ls` EAT function. This function takes as arguments a natural number i and a degeneracy list (that is, a strictly decreasing list of natural numbers); the function has two outputs (compare with the discussion about the two cases in the formula 2.1):

- a new degeneracy list, obtained by systematically applying equations (3), (4) and (5) of Definition 1, starting with ∂_i, and
- an index which *survives* in the previous process, or the symbol *nil* in the case where the equation (4) in Definition 1 is applied and the face map is cancelled.

For instance, with the inputs 4 and (5 2 0) the results of `cmp-d-ls` are (4 2 0) and 2 (recall the process: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). With the inputs 4 and (5 3 0), the outputs are (4 0) and *nil* (since $\partial_4\eta_5\eta_3\eta_0 = \eta_4\partial_4\eta_3\eta_0 = \eta_4\eta_0$).

This function will play an important role when tackling the same problem in Kenzo. We will say that this function implements a *face operator* (over degeneracy operators).

Thanks to Theorem 2, certifying that an EAT object is a simplicial set can be reduced to prove some basic properties of the particular object, the rest of the proof can be generated automatically by ACL2, as will be detailed in Subsection 4.2. The same schema will be applied to the Kenzo model in the next subsection.

3.2 Proving that Kenzo objects are Simplicial Sets

In [10] the correctness of Kenzo degeneracy maps was proven. Thus, we must focus here on the correctness of face maps. The most important function to this aim is called in Kenzo `1dlop-dgop`, equivalent to the previously evoked EAT function `cmp-d-ls`. The arguments and outputs are, of course, equivalent in both functions, but recall that in Kenzo a degeneracy operator is encoded as a natural number.

The proof that the function `1dlop-dgop` is equivalent to `cmp-d-ls` is not simple, for two main reasons. On the one hand, the Kenzo function and the EAT one deal with different representations of degeneracy operators. On the other hand, the Kenzo function implements an algorithm which is not intuitive and is quite different from the algorithm of the EAT version, which is closely related to the mathematical definitions. A suitable strategy, used in [10] for the degeneracy operator, to attack the proof consists in considering an intermediary representation of degeneracy operators based on binary lists (that is, lists of bits), as explained at the second paragraph of Subsection 2.3.

The plan has consisted in defining a function `1dlop-dgop-binary` implementing the application of the face operator over a degeneracy list represented as a

binary list, by means of an algorithm directly inspired from that of Kenzo. Thus, the equivalence between EAT and Kenzo face maps has been proven in two steps. We have proven the equivalence between the Kenzo function `1dlop-dgop` and the binary function `1dlop-dgop-binary`. Subsequently, it has also been proven that the binary version function and the EAT function are equivalent.

Schematically, let $\mathcal{D}_g^L$ be the set of strictly decreasing lists of natural numbers, $\mathcal{D}_g^B$ be the set of binary lists and $\mathbb{N}$ the set of natural numbers. The proof consists in verifying the commutativity of the following diagram (in which the names of the transformation functions have been omitted):

$$\begin{array}{ccccc}
\mathbb{N} \times \mathcal{D}_g^L & \rightleftarrows & \mathbb{N} \times \mathcal{D}_g^B & \rightleftarrows & \mathbb{N} \times \mathbb{N} \\
\Big\downarrow{\scriptstyle cmp-d-ls} & & \Big\downarrow{\scriptstyle 1dlop-dgop-binary} & & \Big\downarrow{\scriptstyle 1dlop-dgop} \\
\mathcal{D}_g^L \times (\mathbb{N} \cup \{nil\}) & \rightleftarrows & \mathcal{D}_g^B \times (\mathbb{N} \cup \{nil\}) & \rightleftarrows & \mathbb{N} \times (\mathbb{N} \cup \{nil\})
\end{array}$$

These functions, that implement the face operators in the different representations, receive as input two arguments (a natural number representing the index of the face map and a degeneracy operator encoded in the respective representation) and have as output a pair composed of a degeneracy operator (in the same representation of the input one) and either a natural number or the value `nil` (as explained in detail in the case of `cmp-d-ls` in the previous subsection).

Thus the commutativity of the diagram ensures the equivalence modulo the change of representation between the EAT and Kenzo models.

More concretely, both the degeneracy operator correctness and the properties included in Theorem 2, can be translated to the Kenzo system thanks to the domain transformation theorems, producing a similar structural theorem in ACL2 for the Kenzo model, which accurately corresponds with the next statement.

Theorem 3 Let $\mathcal{K}$ be a Kenzo object implementing a simplicial set, satisfying for every natural number $q \geq 2$ and for every geometric simplex $gmsm$ in dimension q the following properties:

1. $\forall i, j \in \mathbb{N} : i < j \leq q \Longrightarrow \partial_i^{q-1}(\partial_j^q gmsm) = \partial_{j-1}^{q-1}(\partial_i^q gmsm)$,
2. $\forall i \in \mathbb{N}$, $i \leq q$: $\partial_i^q gmsm$ is a simplex of $\mathcal{K}$ in dimension $q-1$,

then:

$\mathcal{K}$ is a simplicial set.

4 ACL2 technical issues

In this section we deal with three technical issues in ACL2: (1) how to prove the correctness of face operators implemented in Kenzo, (2) the definition of a generic simplicial set theory in ACL2 which can be used to certify that the so-called simplicial sets of the Kenzo system are really... simplicial sets, and (3) the instantiation of this generic tool to concrete examples of families of simplicial sets actually programmed in Kenzo.

4.1 Correctness of face and degeneracy operators

Since the degeneracy operator was studied in [10], we are going to focus on the face operator.

The left side in Figure 1 contains the *real* Common Lisp code of Kenzo for the application of a face map over a degeneracy operator. That definition receives as inputs two naturals numbers `1dlop` and `dgop` (the second one to be interpreted as a degeneracy operator) and returns two values: a natural number (encoding a degeneracy operator) and an index (observe the occurrence of the `values` primitive). The algorithm takes advantage of the encoding of degeneracy operators as numbers, by using very efficient Common Lisp (and ACL2) primitives. For example, to know if `1dlop` is an active bit of `dgop` it uses the function `(logbitp 1dlop dgop)`. Or for computing the *xor* of two numbers it uses the `logxor` operator. Thanks to this way of programming, the function does not need an iterative processing, but just a conditional distinction of cases. It is to be compared with the corresponding EAT function `cmp-d-ls` whose linear time algorithm closely follows the natural mathematical iteration (recall once more the process: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). It is easy to understand the benefits of the Kenzo approach from the performance point of view. It should also be clear that the correctness of the Kenzo function is not evident (contrary to its EAT counterpart), and then an ACL2 verification is a highly valuable objective.

```
(defun 1dlop-dgop (1dlop dgop)
  (progn
    (when (logbitp 1dlop dgop)
      (let ((share (ash -1 1dlop)))
        (declare (fixnum share))
        (return-from 1dlop-dgop
          (values
           (logxor
            (logand share (ash dgop -1))
            (logandc1 share dgop))
           nil))))
    (when (and (plusp 1dlop)
               (logbitp (1- 1dlop) dgop))
      (let ((share (ash -1 1dlop)))
        (declare (fixnum share))
        (setf share (ash share -1))
        (return-from 1dlop-dgop
          (values
           (logxor
            (logand share (ash dgop -1))
            (logandc1 share dgop))
           nil))))
    (let ((share (ash -1 1dlop)))
      (declare (fixnum share))
      (let ((right (logandc1 share dgop)))
        (declare (fixnum right))
        (values
         (logxor
            right
            (logand share (ash dgop -1)))
         (- 1dlop (logcount right)))))))
```

```
(defun 1dlop-dgop-dgop (1dlop dgop)
  (if (and (natp 1dlop) (natp dgop))
      (cond ((logbitp 1dlop dgop)
             (logxor
              (logand (ash -1 1dlop)
                      (ash dgop -1))
              (logandc1 (ash -1 1dlop)
                        dgop)))
            ((and (plusp 1dlop)
                  (logbitp (- 1dlop 1) dgop))
             (logxor
              (logand (ash (ash -1 1dlop) -1)
                      (ash dgop -1))
              (logandc1 (ash (ash -1 1dlop) -1)
                        dgop)))
            (t (logxor
                (logandc1 (ash -1 1dlop) dgop)
                (logand (ash -1 1dlop)
                        (ash dgop -1)))))
    nil))

(defun 1dlop-dgop-indx (1dlop dgop)
  (if (or (logbitp 1dlop dgop)
          (and (plusp 1dlop)
               (logbitp (- 1dlop 1) dgop)))
      nil
    (- 1dlop
       (logcount (logandc1 (ash -1 1dlop) dgop)
                 ))))
```

Fig. 1. 1dlop-dgop definition in Kenzo and in ACL2

The right box of Figure 1 contains our ACL2 definition. The function `1dlop-dgop` is separated into two functions (a different, but equivalent, alternative consists in using the `mv` ACL2 macro which returns two or more values). However, this is the only important difference between the two sides of Figure 1, since all the binary operations of Common Lisp (`logxor` and the like) are present in ACL2. Thus, the two ACL2 programs make up an accurate version of the Kenzo one. But, of course, the challenge of proving the equivalence with the (ACL2 verified) EAT version remains. Following the guidelines given in [10], and with some effort, our methodology was used with success.

Once the correctness of the face and degeneracy maps over degeneracy operators has been proven, the task of certifying properties like Theorem 1 can be carried out. First proving them using the EAT formal specification and later on translating the properties to Kenzo by means of the domain transformation theorems. Figure 2 shows our ACL2 version of Kenzo definitions of the face and degeneracy maps for simplicial sets over abstract simplexes. The name of the functions (with the prefix `imp-`) is reminiscent of the *imp-construction* introduced in [8] to explain, in an algebraic specification framework, the way in which EAT manipulates its objects. Here it is used, at a syntactical level, to get a signature closer to mathematical Definition 1 (with three arguments: an index `n`, a dimension `q` and a simplex), and then to hide the irrelevant information to the operational functions (both the ambient simplicial set and the dimension are irrelevant for the degeneracy function, but only the dimension is irrelevant for the face map). More importantly, the imp-construction allows us to organize simplicial sets as indexed families (the parameter `ss` being the "index" of the space in the family), factoring out proofs for each element of the family (the families we are thinking of having already been mentioned: spheres, Moore spaces, simplicial complexes, and so on).

The `face-kenzo` function of Figure 2 uses the previously introduced functions `1dlop-dgop-dgop` and `1dlop-dgop-indx`, and the function `face` which contains the actual Kenzo definition for faces of geometric simplexes in a simplicial set belonging to a concrete family of spaces. The `degeneracy-kenzo` function uses `dgop-ext-int` to changing of domain from lists (of natural numbers) to natural numbers; then it applies the Kenzo function `dgop*dgop` to compute the composition of degeneracy lists (the ACL2 reification and verification of these Kenzo functions were dealt with in [10]). Let us remark that the ACL2 verified code is the Kenzo code with the sole transformation depicted in Figure 1.

Each function presented in Figure 2 has an equivalent for the EAT representation. Figure 3 uses them to state the theorem `eat-property-3`, which reflects accurately the equation (3) of Definition 1. That theorem must be proven in ACL2 from scratch, by using only the EAT models. Then the domain transformation theorems are applied, and the theorem `kenzo-property-3` of Figure 3 is obtained for free in ACL2. The same schema applies for the rest of properties in Definition 1.

```
(defun face-kenzo (ss d dgop gmsm)
  (if (1dlop-dgop-dgop d dgop)
      (list (1dlop-dgop-dgop d dgop) gmsm)
    (list (1dlop-dgop-dgop d dgop) (face ss (1dlop-dgop-indx d dgop) gmsm))))

(defun imp-face-kenzo (ss d q absm)
  (declare (ignore q)) (face-kenzo ss d (car absm) (cadr absm)))

(defun degeneracy-kenzo (d dgop gmsm)
  (list (dgop*dgop (dgop-ext-int (list d)) dgop) gmsm))

(defun imp-degeneracy-kenzo (ss d q absm)
  (declare (ignore ss q)) (degeneracy-kenzo d (car absm) (cadr absm)))
```

Fig. 2. ACL2 definition of Kenzo operators over abstract simplexes

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

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

Fig. 3. A Kenzo property from an EAT property

4.2 A generic simplicial set theory

The strength of Theorem 3 relies on the few preconditions needed in order to prove that a Kenzo object is a simplicial set. It is worth providing an ACL2 tool such that, when a user proves the preconditions in ACL2, the system generates automatically the complete proof.

To this aim, a generic instantiation tool [9] has been used. This tool provides a way to develop a generic theory and to instantiate the definitions and theorems of the theory for different implementations, in our case different Kenzo objects.

We must specify our simplicial sets generic theory, by means of an ACL2 tool called `encapsulate`. The signatures of the encapsulated functions are as follows:

- `(face * * *)`: to compute the face of a geometric simplex,
- `(dimension *)`: to compute the dimension of a simplex,
- `(canonical *)`: to determine if an object is a simplex in canonical form,
- `(member-ss * *)`: to know if the second argument is a simplex of the first one, a simplicial set.

To finish our generic model we have to assume the properties of Figure 4 (these properties correspond with hypothesis of Theorem 3) and to prove them with respect to a *witness*, to ensure that the consistency of the logical world is kept.

```
(defthm faceoface
  (implies (and (natp i) (natp j) (< i j) (canonical gmsm))
           (equal (face ss i (face ss j gmsm)) (face ss (- j 1) (face ss i gmsm)))))

(defthm face-dimension
 (implies (and (canonical gmsm) (natp i) (< i (dimension gmsm)))
          (equal (dimension (face ss i gmsm)) (1- (dimension gmsm)))))

(defthm face-member
 (implies (and (canonical gmsm) (member-ss ss gmsm) (natp i) (< i (dimension gmsm)))
          (member-ss ss (face ss i gmsm))))

(defthm natp-dimension
  (implies (canonical gmsm)
           (natp (dimension gmsm))))
```

Fig. 4. Assumed axioms

Once the generic theory has been built, producing a book[2] (for further reference, let us name this book by `generic-kenzo-properties-imp`), if a user gives instances for the previous definitions and proves for them the theorems in Figure 4, ACL2 produces a certification ensuring that all the properties of a simplicial set hold, achieving a proof of Theorem 3 for these instances. Let us note the importance of the automatic generation of the proof by means of some data: from 4 definitions and 4 generic theorems the instantiation tool generates (and instantiates) 15 definitions and 375 theorems, in addition to 77 definitions and 601 theorems which are included from other books. In this way, the hard task of proving that a Kenzo object is a simplicial set from scratch can be relaxed to introduce 4 definitions and to prove 4 theorems.

4.3 Obtaining ACL2 correctness certifications for concrete Kenzo simplicial sets families

We have applied the previous infrastructure to two families of simplicial sets in Kenzo: spheres (indexed by a natural number n, with $n > 0$) and simplicial sets coming from finite simplicial complexes (here, each space in the family is determined by a list of maximal simplexes).

If our intention is simply to prove the correctness of finite spaces like the previous ones, one strategy could be to verify the Kenzo function which is used to generate this kind of simplicial sets, called `build-finite-ss`. Nevertheless, our aim is also to provide proofs for infinite dimensional spaces which are offered by Kenzo to the user to initiate computations. Examples are Eilenberg-MacLane spaces (see [11]) and the universal simplicial set, usually denoted by Δ (this

[2] *Book* refers in the ACL2 jargon to a file containing definitions and statements that have been certified as admissible by the system.

particular space already played an important role in the ACL2 proof of Proposition 1 in [1]). To reach this objective, our approach is more general, as it can be applied to *any* simplicial set, regardless of its dimension.

We now explain how the generic tool is instantiated in the particular case of spheres. Spheres are produced in Kenzo by invoking the function `sphere` over a positive natural number `n`. The constructed object contains, in particular, a slot with a λ-term computing the faces of each simplex in the given sphere.

Consider that we want to write an ACL2 book `ss-sphere.lisp` which generates a proof of the fact that spheres with this Kenzo implementation are simplicial sets, through the functions and properties of the generic theory. To this aim, we include the following: `(include-book "generic-kenzo-properties-imp")`. This event generates `definstance-*simplicial-set-kenzo*`, a macro which will be used to instantiate the events from the generic book. It is now needed to define the counterparts of the generic functions. For instance, the counterpart of the function `face` will be called `face-sphere`, and will be, essentially, the λ-term previously evoked. Later on, the statements presented in the previous subsection in Fig. 4 must be also proven. For instance, the following theorem must be proven.

```
(defthm faceoface-sphere
 (implies (and (natp i) (natp j) (< i j) (canonical-sphere gmsm))
          (equal (face-sphere n i (face-sphere n j gmsm))
                 (face-sphere n (+ -1 j) (face-sphere n i gmsm)))))
```

Finally we instantiate all the events from `*simplicial-set-kenzo*`, simply by this macro call:

```
(definstance-*simplicial-set-kenzo*
 ((face face-sphere) (canonical canonical-sphere)
  (dimension dimension-sphere) (member-ss member-ss-sphere))
 "-sphere")
```

At this moment, new instantiated definitions and theorems are available in the ACL2 logical world, proving that Kenzo spheres satisfy all the conditions of Definition 1.

5 Conclusions and future work

A framework to prove the correctness of simplicial sets as implemented in the Kenzo system has been presented. As examples of application we have given a complete correctness proof of the implementation in Kenzo of spheres and of simplicial sets coming from simplicial complexes (modulo a safe translation of Kenzo programs to ACL2 syntax) has been done. By means of the same generic theory the correctness of other Kenzo simplicial sets can be proved.

Some parts of the future work are quite natural. With the acquire experience, the presented methodology could be extrapolated to other algebraic Kenzo data structures. So, this work can be considered a solid step towards our objective of verifying in ACL2 first order fragments of the Kenzo computer algebra system. Nevertheless, it is not evident how ACL2 could be used to certify the correctness

of constructors which generate new spaces from another ones because, as was explained in the Introduction, higher-order functional programming is involved.

In a different line, we want to integrate ACL2 certification capabilities in our user interface *fKenzo* [6]. The idea is to interact in a same friendly front-end with Kenzo and ACL2. For instance, and closely related to the contributions presented in this paper, the user could give information to construct a new simplicial set with Kenzo. In addition, he could provide minimal clues to ACL2 explaining why his construction is sensible (technically, he should afford the system with the ACL2 hypothesis of Theorem 3); then ACL2 would produce a complete proof of the correctness of the construction. This kind of interaction between computer algebra and theorem provers would be very valuable, but severe difficulties related to finding common representation models are yet to be overcome.

References

1. M. Andrés, L. Lambán, J. Rubio, and J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. *Proceedings ACL2 Workshop 2007. University of Austin*, pages 34–39, 2007.
2. J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4):271–292, 2008.
3. 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.
4. X. Dousson, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo`.
5. J. Heras. ACL2 verification of Kenzo simplicial sets, 2010. `http://www.unirioja.es/cu/joheras/ss-tool.html`.
6. J. Heras, V. Pascual, and J. Rubio. Using Open Mathematical Documents to interface Computer Algebra and Proof Assistant systems. *Lecture Notes in Artificial Intelligence*, 5625:467–473, 2009.
7. M. Kaufmann and J. S. Moore. ACL2 Home Page. `http://www.cs.utexas.edu/users/moore/acl2/`.
8. L. Lambán, V. Pascual, and J. Rubio. An object-oriented interpretation of the EAT system. *Applicable Algebra in Engineering, Communication and Computing*, 14(3):187–215, 2003.
9. 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. *Proceedings of the Third ACL2 Workshop. University of Grenoble, France*, pages 188–203, 2002.
10. F. J. Martín-Mateos, J. Rubio, and J. L. Ruiz-Reina. ACL2 verification of simplicial degeneracy programs in the Kenzo system. *Lecture Notes in Computer Science*, 5625:106–121, 2009.
11. J. P. May. *Simplicial objects in Algebraic Topology*, volume 11 of *Van Nostrand Mathematical Studies*. 1967.
12. J. Rubio, F. Sergeraert, and Y. Siret. EAT: Symbolic Software for Effective Homology Computation. Institut Fourier, Grenoble, 1990. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/#Eat`.
13. F. Sergeraert. The computability problem in Algebraic Topology. *Advances in Mathematics*, 104:1–29, 1994.