CONTRIBUCIONES CIENTÍFICAS
EN HONOR DE MIRIAN ANDRÉS GÓMEZ
(Laureano Lambán, Ana Romero y Julio Rubio, editores),
Servicio de Publicaciones, Universidad de La Rioja,
Logroño, Spain, 2010.

FORMALIZING SIMPLICIAL TOPOLOGY IN ISABELLE/HOL AND COQ

JESÚS ARANSAY-AZOFRA AND CÉSAR DOMÍNGUEZ

A nuestra compañera y amiga Mirian

RESUMEN. En este trabajo estudiamos uno de los últimos trabajos de Mirian, de título *Formalizing Simplicial Topology in ACL2* [2], que fue presentado por ella en el congreso anual más relevante sobre ACL2, celebrado en Austin (Texas) en 2007. Aquí explicamos los principales resultados incluidos en ese trabajo, que fueron formalizados por medio de ACL2. También realizamos una formalización de dichos resultados en dos asistentes de demostración diferentes, Isabelle/HOL y Coq. El artículo ha sido escrito de tal forma que debería servir también como introducción a algunas de las características básicas de dichos sistemas.

ABSTRACT. In this paper we study one of Mirian's last works entitled *Formalizing Simplicial Topology in ACL2* [2] which was presented and defended by her in the main meeting about ACL2 held in Austin (Texas) in 2007. We explain the main results included in that work which were formalized using the ACL2 theorem prover. A formalization of these results in two different proof assistants, namely Isabelle/HOL and Coq, is presented. The paper is written in such a way that it could serve also as an introduction to some of the basic features of these systems.

1. INTRODUCTION

One of the objectives of our research group consists in using formal methods to analyze the Kenzo system [5]. This is a Common Lisp program designed by F. Sergeraert, implementing his ideas on *Constructive Algebraic Topology* [11]. One of these formal methods consists in using theorem provers in order to increase the reliability of the system. This was the main topic of Mirian. In particular she was specialized in the use of the ACL2 theorem prover [6]. ACL2 is an extension of a sub-language of Common Lisp and also an environment to produce proofs of properties of programs. As Kenzo has been implemented in Common Lisp, the idea of using ACL2 to verify the *actual* Kenzo programs is quite appealing. However, some of the features erased from Common Lisp in ACL2 are widely used in the implementation of Kenzo. From a programming point of view, Kenzo is essentially based on two matters [11]: higher-order functional programming and

2000 *Mathematics Subject Classification.* 68N30, 57T99.
Key words and phrases. Formalized mathematics, Simplicial Topology, Isabelle/HOL, Coq.
Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01.

symbolic manipulation of data structures as lists or trees of numbers or symbols. The logic of ACL2 is first-order and does not allow using higher-order functionals. Nevertheless, it is very well suited to work with data structures as lists. Thus, it is possible to look for essential first-order fragments of Kenzo and to reprogram and verify them in ACL2. This was one of Mirian's goals which originated different works (as for instance [1, 2]).

There exist other proof assistants which can be taken into account to this aim. Two of them used in our group are Isabelle [10] and Coq [4]. Both tools admit higher-order constructs but, since they have not direct relation with Common Lisp, the theories and proofs that can be built in these tools are quite distant from the Kenzo code. Then, although it is not possible to work with the real code, these assistants can be used to model and verify the algorithms programmed in it. For instance, Isabelle has been useful to implement a formal proof of one of the fundamental algorithms included in Kenzo, as is the *Basic Perturbation Lemma* [3].

It is clear then that these different tools can collaborate in order to reach the objective of increasing the reliability of Kenzo tackling the problem from different points of view. Besides, a comparison of the possibilities and styles of the systems in this particular area could be possible.

In this paper we study one of Mirian's last works. This work was entitled *Formalizing Simplicial Topology in ACL2* [2] and was presented and defended by Mirian in the main meeting about ACL2 held in Austin (Texas) in 2007. Simplicial Topology is an essential mathematical tool in Kenzo and the paper presents an approach to formally analyze concepts and algorithms in this area using ACL2. This work is summarized in the next section. We also include a manuscript which details the proof of the main lemma being formalized. Taking her work as a guideline, we verify the results in [2] using Isabelle, in Section 3, and Coq, in Section 4. Both formalizations point out essential features of the systems and illustrate a comparison between them. The paper ends with a section of conclusions and further work.

2. Simplicial Topology in ACL2

Simplicial Topology [9] is a theory where *abstract* topological spaces are replaced by *combinatorial* artifacts, such as *simplicial sets* or *simplicial complexes*. Then, topological spaces are recovered by means of a notion of *(geometrical) realization* of simplicial structures. The idea behind Simplicial Topology is that *algebraic* invariants associated to topological spaces are read (*i.e.*, computed) in an easier way from that combinatorial objects. In the following we will focus on the notion of simplicial set, since it is the one usually employed in Kenzo.

Definition 1. A *simplicial set* K consists of a graded set $\{K_q\}_{q\in\mathbb{N}}$ and, for each pair of integers (i, q) with $0 \leq i \leq q$, *face* and *degeneracy* maps, $\partial_i \colon K_q \to K_{q-1}$

and $\eta_i \colon K_q \to K_{q+1}$ satisfying the following identities:

$$\partial_i \partial_j = \partial_{j-1} \partial_i \quad \text{if } i < j \tag{1}$$

$$\eta_i \eta_j = \eta_{j+1} \eta_i \quad \text{if } i \leq j \tag{2}$$

$$\partial_i \eta_j = \eta_{j-1} \partial_i \quad \text{if } i < j \tag{3}$$

$$\partial_i \eta_j = id \quad \text{if } i = j \text{ or } i = j+1 \tag{4}$$

$$\partial_i \eta_j = \eta_j \partial_{i-1} \quad \text{if } i > j+1 \tag{5}$$

The elements of K_q are called *q-simplices.* A q-simplex x is *degenerated* if $x = \eta_i y$ with $y \in K_{q-1}$, $0 \leq i < q$; otherwise x is called *non-degenerated.*

The following result states that every degenerated simplex admits a representation in the form of a non-degenerated simplex together with a collection of degeneracies. This is the main theorem that Mirian deals with in [2].

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

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

with $y \in K_r$, $k = n - r > 0$, $0 \leq j_1 < \ldots < j_k < n$.

There exists an important example of simplicial set called *universal* simplicial set Δ [9]. This simplicial set contains the minimal number of identifications from the equalities in Definition 1. That is to say, any theorem proved on Δ, by using only these identities, will be also true for any other simplicial set K. The simplicial set Δ admits a representation in the form of lists with suitable face and degeneracy operations. Thus, it is possible to formalize fragments of the theory of Simplicial Topology in ACL2, which is specially well suited for working with these type of structures. This representation was the one used in [2] (that we resemble in Sections 3 and 4 for our development).

Following the previous idea, a q-simplex of Δ is formalized in ACL2 as a list of length $q+1$. Then, a non-degenerated simplex is a list where any two consecutive elements are different (for instance, $(0, 3, 6)$). The face operator ∂_i (`(del-nth i)` in ACL2 notation) deletes the i-th element of a given list, and the degeneracy operator η_i (`(deg i l)` in ACL2) repeats the i-th element of a given list.

From the previous definitions over Δ, Theorem 1 can be now seen in the following light: every list (*i.e.*, any simplex in Δ) admits a unique representation as a pair of lists, the first one being a (strictly increasing) list of natural numbers (called *degeneracy list*, which represents the set of degenerations $\eta_{j_1}, \ldots, \eta_{j_k}$ in Theorem 1) and the second one a list (called *non-degenerated simplex*) which does not contain two consecutive equal elements (y in Theorem 1). Such a pair is representing the simplex obtained by repeatedly applying the degeneracy operators `deg` using as indexes those of the degeneracy list. For instance, a given list $(k, t, r, r, t, t, t, l, m)$ admits a representation by a pair of lists $(3, 5, 6)$, (k, t, r, t, l, m), where the first list contains the indexes of the positions of the second list that must be duplicated. The ACL2 function which obtains from a degeneracy list `dl` and a non-degenerated simplex `l` the corresponding simplex is named `(degenerate dl l)`. With this notations and definitions, the previous theorem can be stated, in the case of Δ, in the following way:

Theorem 2. *Any simplex l in Δ admits a* unique *representation as a pair of lists (dl, l') with*

$$l = degenerate(dl, l')$$

where l' is a list without two equal consecutive elements and dl a strictly increasing degeneracy list.

The proof of this theorem was divided in [2] in two parts: existence and uniqueness of the pair of lists. Proving the existence requires the definition of a witnesses (dl, l') from l. It is obtained in a natural way using a function `generate(l)` which builds the degeneracy list and the non-degenerated simplex associated to l. Then, the existence theorem was stated in ACL2 in the following way:

```
(defthm existence
  (let ((gen (generate l)))
    (and (canonical gen)
         (equal (degenerate (car gen)(cdr gen)) l))))
```

where `canonical` encodes the required properties in Theorem 2. This theorem was proved using standard interaction with ACL2 by providing proper lemmas and suitable induction schemes.

The uniqueness was enunciated in the following way:

```
(defthm uniqueness
  (implies
     (and (canonical p1) (canonical p2)
          (equal (degenerate (car p1)(cdr p1)) l)
          (equal (degenerate (car p2)(cdr p2)) l))
     (equal p1 p2)))
```

This theorem is easy to achieve using the following lemma:

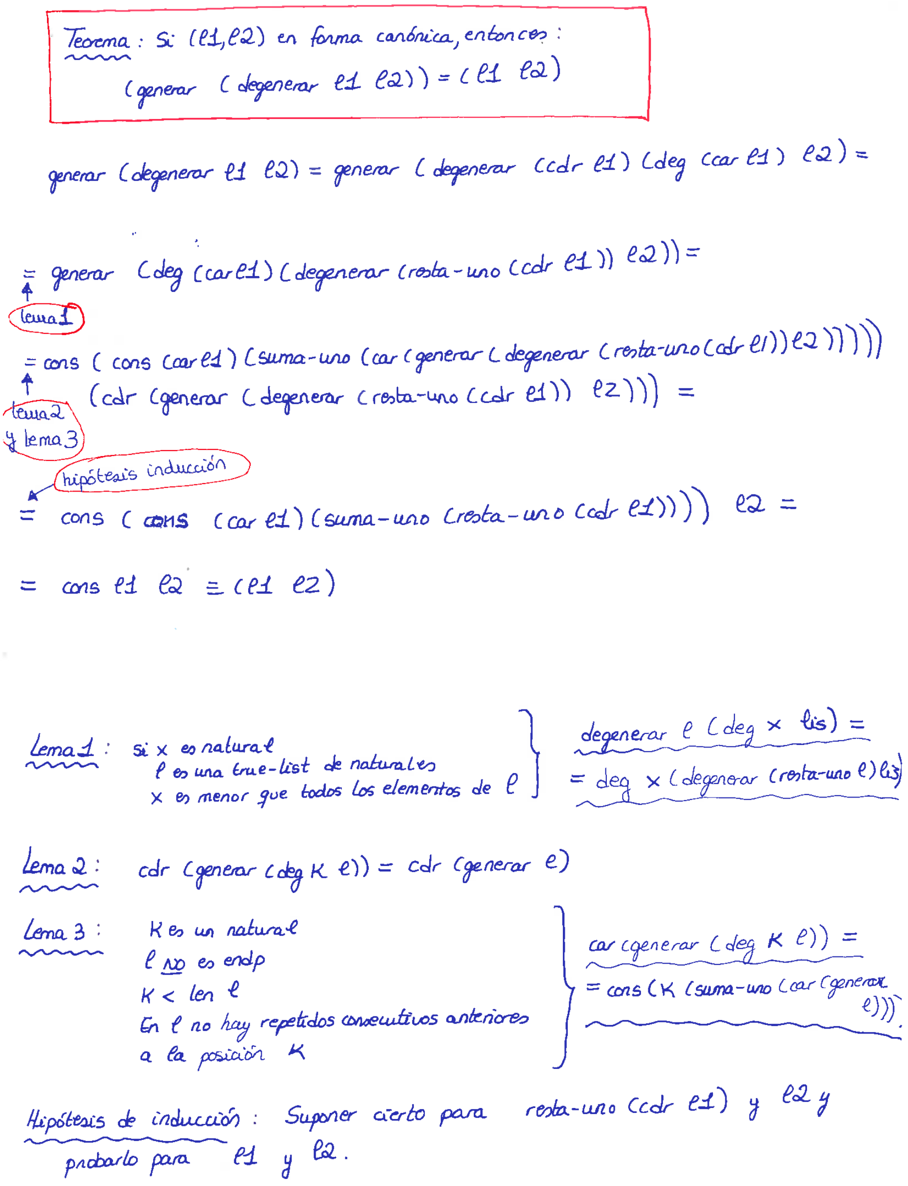
```
(defthm uniqueness-main-lemma
  (implies (canonical (cons l1 l2))
           (equal (generate (degenerate l1 l2))
                  (cons l1 l2))))
```

A sketch of the proof of this lemma can be found in Mirian's manuscript in Figure 1. Her proof is by induction over the length of the first list and uses three auxiliary lemmas.

In the following sections it is explained how the previous definitions and results can be implemented in other different proof assistants as Isabelle/HOL and Coq.

3. Formalization in Isabelle

Isabelle [10] is a generic theorem prover designed for interactive reasoning in a variety of formal theories. It provides currently useful proof procedures for Constructive Type Theory, various first-order logics, Zermelo-Fraenkel set theory and higher-order logic (the latter, usually known as Isabelle/HOL).



Teorema: Si (l1, l2) en forma canónica, entonces:

(generar (degenerar l1 l2)) = (l1 l2)

generar (degenerar l1 l2) = generar (degenerar (cdr l1) (deg (car l1) l2) =

= generar (deg (car l1) (degenerar (resta-uno (cdr l1)) l2)) =

lema 1

= cons (cons (car l1) (suma-uno (car (generar (degenerar (resta-uno (cdr l1)) l2)))))
(cdr (generar (degenerar (resta-uno (cdr l1)) l2))) =

lema 2 y lema 3

hipótesis inducción

= cons (cons (car l1) (suma-uno (resta-uno (cdr l1)))) l2 =

= cons l1 l2 = (l1 l2)

Lema 1: si x es natural
l es una true-list de naturales
x es menor que todos los elementos de l

degenerar l (deg x lis) =
= deg x (degenerar (resta-uno l) lis)

Lema 2: cdr (generar (deg K l)) = cdr (generar l)

Lema 3: K es un natural
l NO es endp
K < len l
En l no hay repetidos consecutivos anteriores a la posición K

car (generar (deg K l)) =
= cons (K (suma-uno (car (generar l))))

Hipótesis de inducción: Suponer cierto para resta-uno (cdr l1) y l2 y probarlo para l1 y l2.

Figure 1. Mirian's manuscript.

Since its first releases (*e.g.*, Isabelle-86) it has been widely used in the formalization of various topics. For instance, in Mathematics, it has been used to formalize the proof of the prime number theorem and the formalization of the Fundamental Theorem of Algebra is now work in progress. The features of the system convert it also in a rather convenient tool for the verification of properties of programming languages (simplified versions of Java and C++ have been fully specified and properties of them verified), hardware specification and testing, or verification of security protocols.

The system has evolved along the years with some remarkable improvements. One of that is Isar, a human-like language on top of Isabelle. In Isar, theorems are stated in an almost natural language (as terms of a type called *prop*, which stands for "propositions"); the proof of that theorem becomes then a proof state (with one or more subgoals). Then the user makes use of proof methods that act over the proof state, trying to solve each single subgoal. The variety of proof methods ranges from decision procedures to methods that are parameterized by rules (or previously proved theorems) and that are resolved against any subgoal. When every subgoal emerging from the original theorem has been proved, the result is accepted by the system. Actually, the system accepts proofs that, starting from the statement, split it into simpler subgoals, but also proofs that, starting from basic facts, successively build up the theorem (in a manner that closely resembles proofs carried out *by hand*). Therefore, the system cannot be considered as an automatic theorem prover, since it requires human interaction to carry out proofs (but supposedly several tedious steps are automatically completed).

In our formalization we have used Isabelle/HOL because of its expressive power and the high degree of automation that it offers in the Isabelle system. Higher-order logic is an implementation of the Simple Theory of Types introduced by Church (also known as simply typed lambda calculus, where neither polymorphism nor dependent types occur).

This section pretends to present a complete development in Isabelle, based on the results presented in Section 2, but is also intended to be a tutorial to a small part of the syntax and style of the system. Let us start by introducing some of the definitions that are required in our formalization.

We first introduce the following type abbreviation that will be used in our development:

```
types 'a deg_pair = "nat list × 'a list"
```

In the previous expression, $'a$ represents a type variable (which can be later replaced by any possible Isabelle type, including basic types, *nat, int* or also functional types themselves). Then, *deg_pair* will work as a type abbreviation, representing a product type, where the first component contains a list of natural numbers and the second one a list of elements of identical type $'a$.

The previous expression also serves us for the purpose of illustrating that the Isabelle implementation of higher-order logic also admits, among others, *product* types. The system is also specially well-equipped to admit data types defined by induction, such as *nat* or *list*.

Let us now introduce a function *generate* over the previous type *'a deg_pair*:[1]

```
fun generate :: "'a list => 'a deg_pair"
  where
  generate_Nil: "generate [] = ([], [])"
  | generate_Cons_1: "generate (a # []) = ([], (a # []))"
  | generate_Cons_2:
  "generate (a # b # l) = (if a = b
                     then (0 # add_one(fst (generate (b # l))),
                            snd (generate (b # l)))
                     else (add_one(fst (generate (b # l))),
                             a # snd (generate (b # l))))"
```

Let us explain some details concerning the previous function definition.

- The function has been defined by means of the *fun* command, which makes internal use of a *fun* package [7] facilitating the definition of functions. Essentially, this facility will be in charge of proving that the previous function *generate* is total and terminating. A function is *total* if it is defined for every possible input. This is specially relevant in our logical setting, since HOL is a logic of total functions. We can observe that the previous function *generate* is total, since it has been defined for lists of length 0 (the empty list, in the case *generate_Nil*), lists of length 1 (in the case *generate_Cons_1*) and lists of length greater than 1 (in the case *generate_Cons_2*).[2] Thus, it is defined for every list.
- A second concern that is also solved by means of the *fun* package is *function termination*. A function is terminating iff the evaluation of every possible input will finish after a finite number of steps. An example of a non-terminating function is $f\colon \mathbb{N} \to \mathbb{N}$ where $f(n) = f(n) + 1$. Accepting the definition of f would guide us to inconsistencies, since from its definition it follows that $0 = 1$. In the case of function *generate*, the proof of its termination could be expressed as follows (this proof is internally carried out by the system, prior to accepting the function definition). If the list is empty, the result of applying *generate* to that list (see the case *generate_Nil* above) will be a pair of empty lists (and thus it terminates). If the list contains a single element a, the result of applying *generate* (see the case *generate_Cons_1*) will be an empty list of degeneracies, and the same list of simplices (and again in this case, it terminates). If the list has two or more elements, the result of applying *generate* to that list (see *generate_Cons_2*) will be the conditional expression shown above. If the two first elements are equal, a degeneracy has been found and is moved to the first list. Otherwise,

[1]The function *add_one* is an auxiliary function that, given a list of natural numbers, increases each of them in one unit. We will present it later.

[2]In Isabelle syntax, [] stands for the empty list, and # for the *append* operation.

the element is appended to the list of simplices. The function recursively calls itself, but always with a shorter list, and thus it terminates (after a finite number of steps, in this case equal to the length of the list).

- The two previous steps have been related to proving that the function *generate* satisfies the minimal requirements to be accepted by the system. Actually, the goal of introducing the function *generate* in the system is to prove some properties of it (in our case, the kind of properties that we presented in Section 2). It can be observed that the function has been *recursively* defined (the case *generate_Cons_2*) over the inductive data type of lists. Thus, a good strategy to prove properties of such function will be by using induction. Once again, the *fun* package will be in charge of automatically introducing an induction rule (named *generate.induct*) that should help us to prove properties of this function. This induction rule is to be added to some other well-known induction rules already available in the system, such as structural induction on lists, or induction on the length of lists. The automatically generated induction rule for this function looks as follows:

```
⟦P []; ⋀a. P [a];
⋀a b l.⟦a = b ⟹ P (b # l)⟧⟹ P (a # b # l);
⋀a b l.⟦a ≠ b ⟹ P (b # l)⟧⟹ P (a # b # l)⟧ ⟹ P a
```

  The previous induction rule, apart from the special syntax of the system, should be read as follows: "If a given property P holds for the empty list (base case), and it also holds for lists with a single element (another base case), and the property is such that for every two elements a and b with $a = b$ and every list l, if the property holds for the list $b\#l$ we know how to prove the property for $a\#b\#l$, and for a and b with $a \neq b$, and any list l, if the property holds for $b\#l$ we also know to prove P for $a\#b\#l$, then we know that P holds for every list l". Usually, this induction rule will be the most adequate one for proving properties of the function *generate*, but any other rule matching with the property to be proved can be applied. Actually, the previous induction rule can be also seen as a particular case of structural induction on lists with a proper case distinction (illustrating the fact that the same theorem can be proven in different ways, or at least by means of different rules).
- Finally, and concerning the definition of the function, we had to provide a type definition of *generate* (otherwise the system would infer the most general type, which is not always the one we pretend), which in this case consists in a functional type mapping a list over a variable type $'a$ to the type of pairs of degeneracy lists over $'a$. Then, the behavior of *generate* over any possible list must be defined (in our case, splitting the definition into three possible cases).

Even if most of the recursive functions that can be thought of are accepted by the *fun* mechanism shown, there are some other ways to introduce function definitions in Isabelle/HOL that we will not consider in this paper [7].

After introducing the concrete syntax of the system, we can now clarify the expected behaviour of function *generate*. For any given list of elements, the function *generate* computes a pair of lists. The first one, of natural numbers, contains the positions in which the original list has two consecutive equal elements (the list of degeneracies in the original list). The second one is a list with all the elements of the original list, but without two consecutive equal elements (a *non-degenerated* list).[3]

The Isabelle definition of the function *add_one* should be now self-explanatory. We also take the chance to introduce another auxiliary function, *red_one*, which decreases in one unit every element of a given list and will be used later:

```
fun add_one :: "nat list => nat list"
  where
  add_one_Nil: "add_one [] = []"
  | add_one_Cons: "add_one (a # l) = (a + 1) # add_one l"

fun red_one:: "nat list => nat list"
  where
  red_one_Nil: "red_one [] = []"
  |red_one_Cons: "red_one (a # l) = (a - 1) # red_one l"
```

The next step in our development would consist in proving some properties of the introduced function. In order to state such properties, one can indistinctly use different commands in the Isabelle syntax (such as *theorem, lemma* or *corollary*).

```
lemma  "hd (snd (generate (a # l))) = a"
  by (induct l rule: generate.induct, auto)
```

The first line in the previous code script states the property that we would like to prove. In this case, that whenever we apply the function *generate* to a non-empty list (every non-empty list can be expressed in the form $a\#l$ for a given element a and a list l), the second list produced, which should contain the elements of the original list without equal consecutive elements, will have the element a as first element (or *hd*). The second line in the previous script contains a proof of such result (indeed, a proof accepted by the system, otherwise the system would return a failure).

If we look closely to the proof, it can be observed that it is carried out by induction on the list l, applying the induction rule shown above (*generate.induct*). Applying such rule produces four different goals (cases $l = []$, $l = [b]$, $l = b\#c\#m$ with $b = c$ and $l = b\#c\#m$ with $b \neq c$). The *auto* tactical is capable of fulfilling

[3]Of course, that is the purpose of our definition, but we must prove such properties.

the details of the proofs of that four cases, and thus the result is accepted as a new *lemma* for the system.

Actually, we could explicitly provide such details giving place to a longer proof (as in mathematics, there is no unique proof for a given result). Unfortunately, not every proof can be completed in a single line, and we will see examples of more elaborated proofs later. Nevertheless, the previous proof should be helpful for the interested reader to estimate the kind of details that the system is capable of verifying automatically.

The next required definition in our development is the one of the *deg* operator. We use again a recursive definition (in this case, the recursion is defined over the natural numbers) by means of the *fun* package. The cases that are considered in the definition are whether the natural number is equal to 0, or otherwise, is the successor (*Suc*) of some n:

```
fun deg :: "nat => 'a list => 'a list"
  where
  deg_0: "deg 0 (a # l) = a # a # l"
  | deg_Suc: "deg (Suc n) (a # l) = a # deg n l"
```

The previous function is now applied to introduce the definition of the *degenerate* function. The function *degenerate* will be our candidate to be proved to be the inverse of the previous function *generate*.

```
fun degenerate :: "'a deg_pair => 'a list"
  where
  degenerate_Nil: "degenerate ([], m) = m"
  |degenerate_Cons: "degenerate (i # l, m) = degenerate (l, (deg i m))"
```

The function *degenerate* is applied to a pair of a list of natural numbers and a list of elements of any type, and it returns a new list in which the repetitions of the elements pointed out by the natural numbers of the first list are applied to the second list.

The previous functions *generate* and *degenerate* are to be proved now that they define a bijection. This bijection is defined between two sets: One is the set of every list of elements over an (arbitrary) type $'a$. The second set is the one of *canonical* pairs of lists (over the same type $'a$). This property of canonicity of pairs of lists includes various predicates. First, the first list of the pair, the one containing natural numbers, has to be strictly increasing (or decreasing, both cases would be valid). Second, the second list cannot include equal consecutive elements (*i.e* it should be a non-degenerated list). Finally, each natural number in the first list has to be less than the length of the list over which it will be applied (in order to be a meaningful degeneracy when applied to such list). Let us introduce the Isabelle definitions of such predicates:

```
fun str_incr:: "nat list => bool"
  where
  str_incr_Nil: "str_incr [] = True"
  | str_incr_Cons_1: "str_incr (a # []) = True"
  | str_incr_Cons_2:
    "str_incr (a # b # l) = ((a < b) ∧ (str_incr (b # l)))"

fun no_consec_repet :: "'a list => bool"
  where
  no_consec_repet_Nil: "no_consec_repet [] = True"
  | no_consec_repet_Cons_1: "no_consec_repet (a # []) = True"
  | no_consec_repet_Cons_2:
  "no_consec_repet (a # b # l) = ((a ≠ b) ∧ no_consec_repet (b # l))"

fun deg_list :: "'a deg_pair => bool"
  where
  deg_list_Nil: "deg_list ([], l) = True"
  |deg_list_Cons:
   "deg_list (a # b, l) = ((a < length l) ∧ (deg_list (b, deg a l)))"
```

Now, we join the three previous predicates (it can be observed that the return value of each of them is of type *bool*, the type of boolean values) in a single definition named *canonical*, by means of the following command:

```
definition canonical :: "'a deg_pair => bool"
  where canonical_def: "canonical dg =
  (str_incr (fst dg) ∧ no_consec_repet (snd dg) ∧ deg_list dg)"
```

The previous predicate *canonical* also serves us for the purpose of illustrating that not every definition in the system needs to be recursively stated (by means of the *fun* package).

With the previous definitions (and some auxiliary lemmas about them), we are ready to state (and to prove) the following statement:

```
lemma existence:
  "canonical (generate l) ∧ degenerate (generate l) = l"
```

The statement includes two different facts. First, that the result of applying the function *generate* over any given list l satisfies the *canonical* definition. Secondly, and regarding the bijection, the result of applying function *generate* to any list l and then function *degenerate* is equal to that list.

There is still a second interpretation of the previous statement (and that is the reason why we named it *existence*). For any given list l, *there exists* a canonical representation of it, *generate* l (and this canonical representation is such that when applying *degenerate* to it is equal to the original list l).

The proof of the previous result in Isabelle is implemented by induction over the list l, and consumed *ca.* 100 lines of code.

The second part of our development consists in proving that for any *canonical* pair of lists $l = (l1, l2)$, the result of applying function *degenerate* and then *generate* is equal to l. This result will also give us the key to prove that, for every list l there exists a *unique* pair of lists *(l1, l2)* such that they are canonical and $l =$ *degenerate (l1, l2)*.

The statement of the result is as follows:

```
lemma uniqueness_main_lemma:
  assumes "canonical (l1, l2)"
  shows "generate (degenerate (l1, l2)) = (l1, l2)"
```

The previous result, together with the previous lemma *existence*, complete the proof of the set of lists and the set of canonical pairs of lists being bijective.

The previous lemma, though, contains a little subtlety. If we try to prove it by *induction over the structure* of $l1$, the two following cases show up.

First, the case where $l1$ is the empty list. Its proof is completed in the following way. It can be proved that degenerating any list (for instance, $l2$) by an empty list of natural numbers ($l1 = []$) produces the same list (in this case, $l2$). Then, the result of generating a list where no consecutive elements are equal ($l2$) is a pair of lists, where the list of degeneracies is empty, and the list containing no consecutive equal elements is that list ($l2$).

The second case considers $l1$ not empty, so let us assume that $l1 = a\#ld$. In order to apply the induction hypothesis, we should be capable of proving that the pair $(ld, l2)$ is canonical, for which we only can count on the premise $canonical(l1, l2)$, but in general that is not true.

A different approach has to be think of. The idea (thanks to L. Lambán and Mirian) consisted in using induction *on the length* of $l1$. Therefore, the induction hypothesis will be (in Isabelle syntax) as follows:

```
∀l.length l < length l1 ⟶ canonical (l,l2)
          ⟶ generate (degenerate (l,l2))=(l,l2)
```

Now, we apply case distinction over the structure of $l1$. The proof of the case $l1 = []$ is completed as explained above. The case where $l1 = a\#ld$ requires to detect under which conditions it is possible to extract the first degeneration (*deg a*), obtaining thus a case where the induction hypothesis can be applied.

This condition relies on a predicate named *fst_repet* which, given a list, returns the position of the list in which the first pair of consecutive equal elements appear (if there is no such pair, it returns the length of the list).

This function is defined as follows:

```
fun fst_repet_i :: "'a list => nat => nat"
  where
  fst_repet_i_Nil: "fst_repet_i [] i = i"
  |fst_repet_i_Cons_1: "fst_repet_i (a # []) i = Suc i"
  |fst_repet_i_Cons_2: "fst_repet_i (a # b # c) i =
  (if a = b then (Suc i) else (fst_repet_i (b # c) (Suc i)))"
```

definition `fst_repet :: "'a list => nat"`
  where `fst_repet_def: "fst_repet l = fst_repet_i l 0"`

The previous function, *fst_repet*, helps us in the following way. Let us show a stepwise proof of the lemma *uniqueness_main_lemma* in Isabelle, that closely follows the one in Mirian's manuscript.

We first unfold the definition of $l1$, which is equal to $a\#ld$:

have `"generate (degenerate (l1, l2)) = generate (degenerate (a # ld, l2))"`

Then, we can extract the first degeneration, a, from the *degenerate* function, by making use of the result named in Mirian's manuscript *lema 1*, obtaining the following expression:[4]

also have `"... = generate (deg a (degenerate (red_one ld, l2)))"`

Our next step will consist in moving the degeneracy *deg a* out of the *generate* operator. This step requires two previous results named *lema 2* and *lema 3* in Mirian's manuscript. The result labeled as *lema 3* is the one demanding the use of the function *fst_repet*. In order to move the degeneracy a out of the *generate* function, we need to prove first that a is less than the first repetition occurring in *degenerate* (*red_one* $ld, l2$). After proving that premise, the following equality holds:

```
also have "... = (a # add_one (fst(generate(degenerate(red_one ld, l2)))),
                  snd(generate(degenerate(red_one ld, l2))))"
```

Now we are ready to introduce the induction hypothesis. The length of *red_one* ld (which is equal to the length of ld), is less than the length of $l1$. Additionally, the pair (*red_one* $ld, l2$) is canonical. Thus, the induction hypothesis is applicable to (*red_one* $ld, l2$), and the following equality holds:

also have `"... = (a # add_one (fst (red_one ld, l2)), snd (red_one ld, l2)"`

The functions *add_one* and *red_one* are inverse of each other, so this permits to prove the following equality:

also have `"... = (a # ld, l2)"`

Finally, and taking into account that $l1 = a\#ld$, the proof is completed.

[4]The abbreviation "..." stands for the previous expression in the chain of rewriting steps.

With the previous results of existence and uniqueness (that complete the proof of the bijection), proving the unicity of representation becomes a corollary which complete proof we have included below:

```
lemma uniqueness:
  assumes can_1: "canonical (d1, l1)"
  and can_2: "canonical (d2, l2)"
  and deg_1_eq_l: "degenerate (d1, l1) = l"
  and deg_2_eq_l: "degenerate (d2, l2) = l"
  shows "(d1, l1) = (d2, l2)"
proof -
  have "(d1, l1) = generate (degenerate (d1, l1))"
    using uniqueness_main_lemma [OF can_1] ..
  also have "... = generate (degenerate (d2, l2))"
    unfolding deg_1_eq_l
    unfolding sym [OF deg_2_eq_l] ..
  also have "... = (d2, l2)"
    using sym [OF uniqueness_main_lemma [OF can_2]] ..
  finally show ?thesis .
qed
```

4. Formalization in Coq

Coq [4] is an interactive proof assistant based on a theory called Calculus of Inductive Constructions, that is a lambda calculus with a rich type system. Coq provides a specification language named Gallina. Terms of Gallina can represent formal specifications, programs, and proofs of programs complying with their specifications. Using the so-called *Curry-Howard isomorphism*, specifications, programs and proofs are formalized in the same language. The system also supplies a language of tactics that allows the user to guide the proof process. Coq is widely used for building machine-checked mathematical proofs and machine-certified programs.

In this section we present a formalization in Coq of the results included in Section 2. Besides, as in the previous Isabelle development, it is used to illustrate the syntax and style of the tool and to explain some of its characteristics.

There exists libraries which contains some basic structures developed in Coq. An example of such a library is `List` which is *required* (*i.e.*, included) in our development. Such a library contains the inductive definition of the type of lists (which is parameterized by the type of elements of the list):

```
Inductive list (A:Type): Type
  nil : list A
| cons : A -> list A -> list A.
```

The inductive definitions contain the introduction rules that define the type or *constructors*. In this case a list of elements on a type is `nil` or it is built through the inclusion of an element on a list. Coq automatically adds properties to the context that allow defining functions and proving theorems by structural induction on the inductive definitions. For instance, the following property, which allows proofs by structural induction on lists, is included (the constructor `cons` is denoted with the infix notation "`::`"):

```
list_ind : forall (A : Type) (P : list A -> Prop),
  P nil -> (forall (a : A) (l : list A), P l -> P (a :: l)) ->
  forall l : list A, P l
```

In this last property *dependent types* are used. In Coq it is possible to build a type that depends on values of another type. Besides, types are included as terms in the system, *i.e.*, types are first-class values. So, it is also possible to build types that depends on types. This is a very powerful type system which allows defining precise specifications.

In order to represent the simplices of Δ in Coq a type, `A: Type`, with a decidable equality, `eq_dec: forall x y: A,` {`x = y`} + {`x <> y`}, is introduced. Simplices are then formalized as the lists over this type: `ListA:=list A`.

The requirement of the decidable equality could need a more detailed explanation. The equality on a type in Coq is defined through an inductive type:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : eq x x.
```

which includes only one constructor corresponding to reflexivity (`eq` can be represented with the infix notation "`=`"). So, this equality is not a boolean relation like in the Isabelle case. An equality test on the type `A` can be obtained with the function `eq_dec`. Given two elements `x` and `y` in `A`, `eq_dec x y` allows us to check whether {`x = y`} or {`x <> y`} is true. The type of this test is obtained by means of another inductive type `sumbool`:

```
Inductive sumbool (A B : Prop) : Set :=
     left : A -> sumbool A B
   | right : B -> sumbool A B.
```

(`sumbool A B` will be represented by {`A`} + {`B`}). Then, the two options which play the role of *true* and *false* can be retrieved by pattern matching on `eq_dec x y` by means of its two possible cases `left p` and `right q` with `p:x = y` or `q:x <> y`. In that case, we also obtain that `p` or `q` are proofs of `x = y` or `x <> y`, respectively. There exists in Coq an inductive definition of disjunction of propositions `or`. Its definition is the same that `sumbool` above except that its type is `Prop` instead of `Set`. Nevertheless, `or` is not useful in this case because definitions by case analysis on proofs of propositions are not allowed in Coq. The reason of this restriction comes from the intended distinction between the predefined types `Set` and `Prop`, for program types or specifications and logical propositions, respectively. In Coq when we use the sort `Set`, programs are considered as different even if they have the same type. For `Prop`, the details of the proofs that prove a proposition are not important and we are just interested in the existence of the proof. This motivates the *proof irrelevance* on propositions which means that different proofs of a proposition cannot be distinguished. Based on this difference, Coq can automatically extract executable programs from specifications. Roughly speaking, once a program is verified, the extraction mechanism obtains the code erasing the parts with type `Prop`.

Using the previous data types, we can define two recursive functions: `generatel` and `generateN` which will be collected using pairs in a single function `generate`. The first one obtains from a simplex the associated non-degenerated simplex without consecutive repeated elements and the second one the degeneracy list with the positions of the repetitions, represented as a list of natural numbers (`ListN:= list nat`):

```
Fixpoint generatel (l : ListA) : ListA :=
 match l with
 | nil => l
 | a :: nil => l
 | a :: (b :: l2) as l1 =>
     if eq_dec a b then generatel l1 else a::generatel l1
 end.

Fixpoint generateN (l : ListA) : ListN :=
 match l with
 | nil => nil
 | a :: nil => nil
 | a :: (b :: _) as l1 =>
     if eq_dec a b then 0::add_one (generateN l1)
     else add_one (generateN l1)
 end.
```

Again, some comments can be done on the previous definitions. The `Fixpoint` command allows definitions by structural induction on an argument. That is the case of both definitions above defined on the inductive structure of the list. In order to be accepted by the system, definitions have to satisfy some syntactical constraints. Coq imposes an exhaustive pattern matching on the argument structure ensuring that the function is total, *i.e.*, defined over all possible arguments. Also, this argument must decrease along the recursive calls which ensures that the `Fixpoint` definition always terminates. This is obtained imposing that the argument of the recursive call has to be a pattern variable obtained from a case analysis of the formal argument of the function (called *guardedness condition*). In this case, the recursive call `generatel` is applied to the pattern variable `l1` (which renames `b::l2`) obtained from the case analysis, which obviously decreases the argument `l`. This new variable is mandatory because `b::l2` does not fulfill the previous condition since it is not directly obtained as a pattern variable. (At this point, it is equivalent to replace `l2` in `b::l2` by a *placeholder* `b::_` since this term is not used in the definition. This feature is included in the `generateN` definition.) The `if c then b1 else b2` structure also corresponds with a case analysis in which the inductive type of `c` has exactly two constructors. This is the case of `eq_dec a b` as explained before.

The degeneration function of a simplex by a natural number can be defined by pattern matching on a pair of a natural number and the list representing the simplex:

```
Fixpoint deg(i:nat)(l:ListA){struct l}: ListA:=
 match i, l with
    | _, nil => nil
    | O, x :: l' => x::x::l'
    | S n, x :: l' => x::deg n l'
 end.
```

In this recursive function two different inductive types appear as arguments. In that case, it is necessary to tell the system which argument must decrease along the recursive calls. This is made explicit by the {`struct l`} annotation which assigns `l` as decreasing argument. It should be observed that in this definition it is also possible to use as decreasing argument the one in `nat`. This decision has an influence in the subsequent Coq development. In particular, expressions will be simplified using matching by the chosen argument. For instance, in our case `(deg n (x::l))` will be simplified (obtaining a function with a simple pattern matching on the natural number argument `n`), but `(deg (S n) l)` will remain unchanged. For this reason, proofs about this function should rely on induction on the chosen argument. In this case, the argument in `ListA` is chosen considering that the list structure could be more relevant than the natural number in our development.

With these structures it is possible to prove some preliminary lemmas. For instance, the second equality in Definition 1 for our representation of the universal simplicial set can be stated and proved in the following way.

```
Lemma deg_permut: forall (a b:nat)(l:ListA), a<=b -> b<(length l)
  -> deg a (deg b l) = deg (S b)(deg a l).
  Proof.
   double induction a b.
     intro l; case l; simpl; trivial.
     intros n H l; case l; case n; simpl; trivial.
     intros n b0 l H; inversion H.
     intros n H n0 H0 l H1 H2; induction l.
       inversion H2.
       simpl; rewrite H0; auto with arith.
  Qed.
```

In this lemma, the tactic-style of Coq can be appreciated. Its proof is based on double induction on two natural numbers. This origins four subgoals. Three of them are proved by case analysis or induction on the list and the fourth one by contradiction with the hypotheses.

The previous degeneration function of a simplex by a natural number is easily extended in order to define the `degenerate` function of a simplex by a list of natural numbers:

```
Fixpoint degenerate(ln:ListN)(l:ListA):ListA:=
 match ln with
    | nil  => l
    | a::ln' => (degenerate ln' (deg a l))
 end.
```

Then, a witness for the existence part of Theorem 2 is given by the pair of lists built by the generate function. So, by means of the degeneration of the generated lists the initial list is recovered:

```
Lemma existence_degenerate: forall l:ListA,
  (degenerate (generate l)) = l.
```

The proof of this lemma can be obtained by induction on the structure of the list using some auxiliary lemmas.

In order to complete the existence lemma is necessary to prove some canonical properties on both generated lists. These properties are the following. First, generateN generates a strictly increasing list (property denoted by str_incr), second, generatel generates a list without two equal consecutive elements (denoted by no_consc_rep) and finally generateN generates a *correct* degeneracy list with respect to the list generated by generatel, which means that each natural number of the degeneracy list is less than the length of the simplex which degenerates. All these properties are defined in a similar way using inductive types and collected in a predicate named canonical. For instance, the last one is defined as follows:

```
Inductive deg_list: ListN -> ListA -> Prop:=
    deg_list0: forall l:ListA, deg_list nil l
  | deg_list1: forall (a:nat)(l:ListA)(ln:ListN), a<length l ->
             (deg_list ln (deg a l)) -> (deg_list (a::ln) l).
```

With the previous ingredients it is possible to obtain the existence lemma:

```
Lemma existence: forall l:ListA,
 canonical (generate l) /\ (degenerate (generate l)) = l.
```

where the required canonical properties are proved using again induction on the list.

The statement and proof of this lemma has consumed 492 Coq code lines (including the definitions and previous lemmas).

The proof of the uniqueness part of Theorem 2 is based on the following main lemma which is described in the manuscript in Figure 1:

```
Lemma uniqueness_main_lemma: forall (l1:ListN)(l2:ListA),
 canonical (l1, l2) -> (generate (degenerate (l1, l2))) = (l1, l2).
```

The proof of this lemma is also detailed in that manuscript. It can be obtained by induction on the *length* of the first list and using a chain of three rewriting lemmas. Structural induction is not possible in this case. If we try to apply this type of induction, in the cons case l1 is assumed to be a::l. Then, in order to apply the induction hypothesis is necessary to prove canonical (l, l2) from canonical (a::l, l2) which is not true in general. But, it is possible to prove canonical (red_one(l), l2) with red_one a function which reduces in one each element of the list. Since length(red_one(l)) < length(a::l), induction on the length of the list can be useful.

In general, in Coq it is possible to reason by induction on relations that are well-founded. That is the case of the list length relation. A proof that this relation is well-founded is included by P. Castéran and E. Contejean in the Coq users contributions [8]. They also prove the following lemma which formalizes the induction on the length of a list (see [8], for a detailed description):

```
Lemma length_list_ind:
     forall (P:list nat -> Prop),
        (forall (l2:list nat), (forall (l1:list nat),
         length l1 < length l2 -> P (l1)) -> P (l2))
           -> forall (l2:list nat), P l2.
```

Besides, the three auxiliary rewriting lemmas are defined. The first one:

```
Lemma Lema1: forall a ln l, str_incr (a::ln) -> deg_list (a::ln) l
 -> degenerate ln (deg a l) = deg a (degenerate (red_one ln) l).
```

is proved by induction on the list `ln` of natural numbers, and the second one:

```
Lemma Lema2: forall l a, a < length l ->
 generatel(deg a l) = generatel l.
```

by induction on the list `l`. The third Lemma:

```
Lemma Lema3: forall l a, a < fst_repet l ->
 generateN(deg a l) = a::(add_one(generateN l)).
```

introduces a new function `fst_repet` which obtains the position of the first pair of consecutive elements repeated in a list (in the case that there are no consecutive pair of elements repeated then it obtains the length of the list). This function is defined in the following way:

```
Fixpoint fst_repet (l : ListA) : nat :=
 match l with
 | nil => 0
 | a :: nil => 1
 | a :: (b :: _) as l1 => if eq_dec a b then 1 else S(fst_repet l1)
 end.
```

This third lemma is also proved by induction on the list `l`.

Finally, the uniqueness lemma can be trivially obtained by rewriting the previous uniqueness main lemma:

```
Lemma uniqueness:  forall (l1 l2: ListNxListA)(l: ListA),
 canonical l1 /\ canonical  l2 -> (degenerate l1) = l /\
 (degenerate l2) = l -> l1 = l2
  Proof.
   intros ln1 ln2 l1 l2 l H H0;elim H;elim H0;intros H1 H2 H3 H4.
   rewrite <-(uniqueness_main_lemma' H3);trivial.
   rewrite <-(uniqueness_main_lemma' H4);trivial.
   rewrite H1;rewrite H2;trivial.
  Qed.
```

This second part requires 433 lines of Coq code.

5. Conclusions and Further work

This work has presented the formalization of a result in Simplicial Topology in two different theorem provers. Actually, the aim of this work was not to increase the confidence in such a result, since its proof had been already formalized in another theorem prover, ACL2, but to illustrate the *look and feel* of Isabelle/HOL and Coq by means of an example.

The capacities of both systems are far beyond from the ones that we have presented here. The logics underneath them permit quite more elaborated formalizations, and they both pose a wide users' community increasing their features and also the amount of results being formalized.

In this setting, our work should be useful as an introduction to the basics of their type systems, the definition of functions, the proving strategies and the amount of effort being required for a complete formalization.

In addition to this, the formalization presented in this paper should serve also to illustrate that, even for a rather intuitive result, as the one stated in Theorem 2, which simply says that every list admits a representation by means of two lists, one containing the repetitions of that list and another one without repetitions, a formal proof (a pencil & paper one or by means of a theorem prover) can sometimes become rather tricky (and requires a certain amount of previous results and additional definitions).

Another conclusion is that, very commonly, results in Mathematics do admit very different interpretations. In our case, Theorem 2 can be understood as a way to simplify the representation of lists. But the proof of the result demanded from us the definition of a new function *generate*, as well as proving that this function was the inverse of the introduced *degenerate* function. Along our development, we had to prove that both functions defined a bijection between lists and the set of *canonical* pairs of lists. From a different point of view, Theorem 2 could be also seen as a compression algorithm. Actually, extracting the degeneracies out of a list is not very far from the well-known compression algorithm *run-length*.

The work presented here may continue in different lines. For instance, it could be formally proved that the simplicial set Δ and its representation by means of lists really fulfill the identities in Definition 1. Some of that identities have been already proved in our work. Moreover, it could be explored the relation of Δ with respect to the Category of simplicial sets. To this respect, Δ is not an initial object, but it is known to be the simplicial set containing the minimal set of identifications required to define a simplicial set. Yet another idea is to formalize the identities verified by any simplicial set as a rewriting system, and then try to build up proofs of theorems for simplicial sets as normalization processes. This proposal was also explored in Mirian's work [2], getting a different proof of Theorem 2, and its possibilities are still being used in the ACL2 system for proving more elaborated results.

The goal of certifying Kenzo still appears far from our reach. Actually, the system has been widely tested, and the results known have been proven always correct (it is worth noting that some of the results obtained with Kenzo have not been obtained by any other means). Along these years, and thanks, among others, to Mirian's research, different parts of the system have been formalized in different provers, some crucial algorithms implemented and certified, and some of its computations mechanically verified. Thus, the system has shown to be a great laboratory to measure and test the strengths and weaknesses of proof assistants and theorem provers.

References

[1] M. Andrés, L. Lambán, J. Rubio. Executing in Common Lisp, proving in ACL2. In *Proceedings of Calculemus 2007*, M. Kauers et al. (eds.). Lecture Notes in Computer Science **4573**, pp. 1–12. Springer, Hagenberg, Austria, 2007.

[2] M. Andrés, L. Lambán, J. Rubio, J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. In *Seventh International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 34-39. Austin, Texas (Estados Unidos), 2007.

[3] J. Aransay, C. Ballarin, J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning* **40**, 271–292, 2008.

[4] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science, vol. 25. Springer, Berlin, 2004.

[5] X. Dousson, F. Sergeraert, Y. Siret. *The Kenzo program.* 1999. Available at `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.

[6] M. Kaufmann, P. Manolios, J. Strother-Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, 2000.

[7] A. Krauss. *Automating recursive definitions and termination proofs in higher order logic.* Ph.D. Thesis, Technische Universität München, 2009.

[8] LogiCal project. *The Coq Proof Assistant.* 2009. `http://coq.inria.fr/`.

[9] J. P. May. *Simplicial objects in Algebraic Topology.* Van Nostrand Mathematical Studies, vol. 11. D. Van Nostrand Co., Princeton, 1967.

[10] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL: A proof assistant for higher order logic.* Lecture Notes in Computer Science, vol. 2283. Springer, Berlin, 2002.

[11] J. Rubio, F. Sergeraert. Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques* **126**, 389–412, 2002.

Departamento de Matemáticas y Computación, Universidad de La Rioja, Spain
E-mail address: `jesus-maria.aransay@unirioja.es`

Departamento de Matemáticas y Computación, Universidad de La Rioja, Spain
E-mail address: `cesar.dominguez@unirioja.es`