ACL2 Verification of Simplicial Degeneracy Programs in the Kenzo System

Francisco-Jesus Martín-Mateos[1], Julio Rubio[2], and Jose-Luis Ruiz-Reina[1]

[1] Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
{fjesus,jruiz}@us.es
[2] Dept. of Mathematics and Computation, University of La Rioja
Edificio Vives, Luis de Ulloa s/n. 26004 Logroño, Spain
julio.rubio@unirioja.es

Abstract. Kenzo is a Computer Algebra system devoted to Algebraic Topology, and written in the Common Lisp programming language. It is a descendant of a previous system called EAT (for Effective Algebraic Topology). Kenzo shows a much better performance than EAT due, among other reasons, to a smart encoding of degeneracy lists as integers. In this paper, we give a complete automated proof of the correctness of this encoding used in Kenzo. The proof is carried out using ACL2, a system for proving properties of programs written in (a subset of) Common Lisp. The most interesting idea, from a methodological point of view, is our use of EAT to build a model on which the verification is carried out. Thus, EAT, which is logically simpler but less efficient than Kenzo, acts as a mathematical model and then Kenzo is formally verified against it.

1 Introduction

The Kenzo system [8] is a Common Lisp program, developed by F. Sergeraert and devoted to Algebraic Topology. It was written mainly as a research tool and has got relevant results which have not been confirmed nor refuted by any other means. Being a compact program (around 16000 lines of Common Lisp, implementing complicated algorithms), the question of Kenzo reliability (beyond testing) came up in a natural way.

Several approaches based on Formal Methods have been used to undertake this problem, ranging from the Algebraic Specification of its data structures ([12], [7], and recently computer aided with Coq [6]) to the application of Proof Assistants to study the correctness of *algorithms* implemented in Kenzo. In this second line, the most important contributions have been the Isabelle/HOL proof of the Basic Perturbation Lemma [3] and the project by Coquand and Spiwack which is based on Constructive Type Theory and Coq [5]. As it is well-know, Coq

* This work has been supported by Ministerio de Educación y Ciencia, project MTM2006-06513.

proofs carry their corresponding programs, and also some work has been done to produce running code from Isabelle/HOL proofs in this context [4]. Nevertheless, the extracted programs are not comparable with the real Kenzo system, both from the efficiency and the programming languages points of view (OCaML or ML code instead of Common Lisp).

Due to this drawback of the approaches based on Isabelle and Coq, a new research line was launched, focused on the ACL2 theorem prover. ACL2 is oriented to prove properties of Common Lisp programs, and thus it could seem, at first sight, very promising to verify Kenzo. Nevertheless, since the ACL2 logic is first-order, the *full* verification of Kenzo is not possible, since it uses intensively higher order functional programming (to encode, in particular, topological spaces of infinite dimension). This observation, however, does not close the possibility of verifying first order *fragments* of Kenzo with ACL2. Some preliminary works in this line have been published in [1] and [2]. It is worth noting that in those papers we undertake the problem of verifying some Common Lisp programs about simplicial topology (in particular, algebraic manipulation and simplicial properties of Kenzo algorithms), but that no *actual* Kenzo fragment was studied.

In this paper we present for the first time the verification of a Kenzo fragment within the ACL2 theorem prover. The verified fragment is small in number of lines, but it is central to the efficiency got by Kenzo. This is compared to the predecessor of Kenzo, another Common Lisp system called EAT [15], based on the same Sergeraert's ideas, but whose performance was much poorer than that of Kenzo. One of the reasons why Kenzo performs better than EAT is because of a smart encoding of *degeneracy lists*. These combinatorial objects are usually presented in the Simplicial Topology literature as decreasing lists of natural numbers, and so they were encoded in EAT. On the contrary, in Kenzo degeneracy lists are encoded as natural numbers. Since to generate and compose degeneracy lists are operations which appear in an *exponential* manner in most Kenzo calculations (through the Eilenberg-Zilber theorem [14]), it is clear that the benefits of having a better way for storing and processing degeneracy lists is very important. But, on the negative side, the algorithms are somehow *obscured* in Kenzo, with respect to the clean and comprehensible approach in EAT. Therefore, to prove the correctness of the implementation of degeneracy algorithms in Kenzo seems to be a good test-bed to apply computer-aided formal methods.

A complete ACL2 proof of the correctness of the degeneracy programs in Kenzo is described in this paper. The main methodological contribution of the proof is, in our opinion, using EAT to build a model with respect to the verification is carried out. Thus, EAT, which is logically simpler (i.e., easier to be verified) but less efficient than Kenzo, acts as a mathematical model and then Kenzo is formally verified against it.

The organization of the rest of the paper is as follows. In Section 2, we introduce briefly both Simplicial Topology and the role of degeneracy operators in it. In Section 3, we give a brief introduction to the ACL2 system. Even if a *first order* fragment of Kenzo (and EAT) has been chosen, the Kenzo functions cannot be directly defined in ACL2 (due to Common Lisp features, like loops

or destructive updates, which are not available in ACL2). Thus, in Section 4 we explain how to obtain actual ACL2 functions from Kenzo and EAT degeneracy programs, in a safe and reliable way. Sections 5 and 6 are devoted to the description of the ACL2 proof of correctness and other important properties. Finally we comment some conclusions and point out possible further work.

Due to the lack of space, we will not give here details about the proofs obtained and some function definitions will be omitted. The interested reader may consult [13], where the complete development is available.

2 The Role of Degeneracy Operators in Simplicial Topology

Simplicial Topology [14] is a subarea of Topology devoted to replace topological spaces by combinatorial models, in order to ease their study. The simplest combinatorial model of a topological space is a *simplicial complex*. Let V be a set together with a partial order $<$ on it. A *n-simplex* is a list $[v_0, v_1, \ldots, v_n]$ where $v_0 < v_1 < \ldots < v_n$ are elements of V. For each index i we consider the *i-face* operator ∂_i that given a n-simplex constructs a $(n-1)$-simplex deleting the element at position i. A *simplicial complex* K (over $(V, <)$) is a set of simplices closed with respect to the face operators.

Each n-simplex can be *realized* as an affine geometrical simplex (for instance, a 0-simplex is realized as a point, a 1-simplex as a segment, a 2-simplex as a triangle, a 3-simplex as a tetrahedron and so on). Thus, simplicial complexes are models for *triangulated spaces*, which are a class of topological spaces sufficiently large to develop much of the general and algebraic topology. Nevertheless, simplicial complexes have a severe drawback: one needs many simplices to model relatively simple spaces. For instance, to model a sphere with a tetrahedron we need 4 vertices, 6 edges and 4 triangles. Since the topological notions are quite flexible, we could use a much more efficient way of representing a sphere: by means of a triangle where all the edges and vertices are collapsed to just one point. The problem with this new representation is the "dimension jump": there is one element of dimension 2 (the triangle) and one element of dimension 0 (the point), and then this set of simplices is not closed with respect to the face operators.

The solution to this problem is to move from simplicial *complexes* to simplicial *sets*. In addition to the face operators, new operators of *degeneracy* are considered. These operators create "artificial" simplexes (with no geometrical meaning) but allowing "jumping" among dimensions. To give an idea of this sophisticated instrument let us comment briefly on how a simplicial complex can be viewed as a simplicial set. The trick is to accept simplexes that are ordered but not necessarily strictly ordered; that is, repeated elements are allowed. Then for each index i with $0 \leq i \leq n$, we define the *i-degeneracy* operator η_i that given a n-simplex constructs a $(n+1)$-simplex repeating the element at position i.

Based on this idea, we define a *simplicial set* as a graded set $\{K_q\}_{q \in \mathbb{N}}$ of *abstract* simplexes (i.e. not necessarily lists of elements) with the i-face and i-degeneracy operators, satisfying the following *simplicial identities* (see [14] for details):

$$
\begin{array}{ll}
\forall i < j & \partial_i \partial_j = \partial_{j-1} \partial_i \\
\forall i \leq j & \eta_i \eta_j = \eta_{j+1} \eta_i \\
\forall i < j & \partial_i \eta_j = \eta_{j-1} \partial_i \\
\forall i, j & \partial_i \eta_i = Id = \partial_{j+1} \eta_j \\
\forall i > j+1 & \partial_i \eta_j = \eta_j \partial_{i-1}
\end{array} \qquad (1)
$$

A simplicial set represents a topological space in a much less expensive manner than a simplicial complex. For instance, a sphere of dimension n can be represented with just two non-degenerate simplices: one in dimension n and other in dimension 0 (geometrically, all the faces on the affine n-simplex are collapsed over a unique point, producing a topological sphere; think in a segment where the two extremes are identified, producing a circle, a 1-sphere).

A simplex is *degenerate* if it is obtained as the application of some operator η_i. It could be proved that given a simplex x there exists a unique non-degenerate simplex y and a unique strictly decreasing list of natural numbers $[i_0, i_1, \ldots, i_n]$ such that $\eta_{i_0} \eta_{i_1} \ldots \eta_{i_n}(y) = x$. (This fundamental result of Simplicial Topology has been proved in ACL2 as documented in [2]). We call this list of indices $[i_0, i_1, \ldots, i_n]$ a *degeneracy list* and we say that x is obtained applying the degeneracy list $[i_0, i_1, \ldots, i_n]$ to y.

In general, the application of degeneracy lists to simplexes is a very common operation in Kenzo, even for degenerate simplexes. Let us note that the application of a degeneracy list $[i_0, \ldots, i_n]$ to an degenerate simplex x, that is the result of applying another degeneracy list $[j_0, \ldots, j_m]$ to a non-degenerate simplex y, is the result of applying the composition of the two degeneracy lists, $[i_0, \ldots, i_n] \circ [j_0, \ldots, j_m]$, to y. The *composition* of two degeneracy lists is defined as the composition of the degeneracy operators: $[i_0, \ldots, i_n] \circ [j_0, \ldots, j_m] = \eta_{i_0} \ldots \eta_{i_n} \eta_{j_0} \ldots \eta_{j_m}$; repeatedly applying equation (1) above, this could be transformed again into a degeneracy list. The implementation in Kenzo of this composition operation is central in the system as a whole. For example, the composition of the degeneracy lists $[3, 1]$ and $[5, 3, 0]$ is $\eta_3 \eta_1 \eta_5 \eta_3 \eta_0$, and applying repeatedly the equation $\eta_i \eta_j = \eta_{j+1} \eta_i$, when $i \leq j$, we successively obtain $\eta_3 \eta_6 \eta_1 \eta_3 \eta_0$, $\eta_3 \eta_6 \eta_4 \eta_1 \eta_0$, $\eta_7 \eta_3 \eta_4 \eta_1 \eta_0$ and finally $\eta_7 \eta_5 \eta_3 \eta_1 \eta_0$, that is, the degeneracy list $[7, 5, 3, 1, 0]$.

The strategy Sergeraert devised was to interpret a degeneracy list $[i_0, \ldots, i_n]$ as a binary representation of an integer. He stores the degeneracies as integers (with the corresponding memory saving) and implements the composition of degeneracy lists by using very efficient Common Lisp primitives dealing with binary numbers (like `logxor`, `ash`, and so on). This is one of the reasons why Kenzo improves dramatically the performance of its predecessor EAT. Nevertheless, this efficient composition operator called `dgop*dgop` in Kenzo has a more obscure semantics than its corresponding in EAT, called `cmp-ls-ls`. This paper is devoted to describe the certification in ACL2 of the correctness of `dgop*dgop`, using `cmp-ls-ls` as a formal specification, and then proving additional properties like equation (1) of simplicial sets or associativity of `dgop*dgop`.

3 An Introduction to the ACL2 System

ACL2 ([10],[11]) stands for "A Computational Logic for an Applicative Common Lisp". Roughly speaking, ACL2 is a programming language, a logic and a theorem prover. Thus, the system constitutes an environment in which algorithms can be defined and executed, and their properties can be formally specified and proved with the assistance of a mechanical theorem prover.

As a programming language, it is an extension of an applicative subset of Common Lisp[1] [16]. The logic considers every function defined in the programming language as a first-order function in the mathematical sense. For that reason, the programming language is restricted to the applicative subset of Common Lisp. This means, for example, that there are no side-effects, no global variables, no destructive updates and no higher-order features. Even with these restrictions, there is a close connection between ACL2 and Common Lisp: ACL2 primitives that are also Common Lisp primitives behave exactly in the same way, and this means that, in general, ACL2 programs can be executed in any compliant Common Lisp.

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

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

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

4 From Kenzo and EAT to ACL2

Before giving the ACL2 definition of the composition of degeneracy lists (and the statements of the theorems we have proved), let us present the Kenzo code for

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

that operation. As we have said before, Kenzo deals with degeneracy lists using a smart encoding. Basically, every degeneracy list can be seen as the natural number whose binary notation represents the characteristic function of the set of elements of the list. Let us explain this with an example: the degeneracy list $[5, 3, 0]$ can equivalently be seen as the binary list $[1, 0, 0, 1, 0, 1]$ in which 1 is in position i if the number i is in the degeneracy list, 0 otherwise. This list, seen as a binary number in the reverse order, is the natural number 41. Thus, Kenzo encodes the above degeneracy list as 41.

Let us now explain how Kenzo implements composition of degeneracy lists. This is better understood if we think first in the binary representation. Let us consider the composition of the degeneracy lists $[3, 1]$ and $[5, 3, 0]$. Applying repeatedly the equation $\eta_i \eta_j = \eta_{j+1} \eta_i$, when $i \leq j$, we obtain $[7, 5, 3, 1, 0]$. Using binary notation, this means that the composition of $[0, 1, 0, 1]$ and $[1, 0, 0, 1, 0, 1]$ is $[1, 1, 0, 1, 0, 1, 0, 1]$. In general (although it is not obvious), composition between two degeneracy lists in binary notation can be described as sequentially replacing the 0's in the first list by the successive elements of the second list, until one of the lists is exhausted; and then completing the result with the remaining elements of the other list.

As we have said before, Kenzo does not directly use the binary notation: it uses the natural number that this binary notation represents. Common Lisp logical operations on numbers, like `logxor` and `ash`, are used to reflect the corresponding manipulations on binary lists. The following is *the real* Common Lisp code of Kenzo for composition of degeneracy lists[2]:

```
(defun dgop*dgop (dgop1 dgop2)
  (declare (type fixnum dgop1 dgop2))
  (let ((dgop 0) (bmark 0))
    (declare (fixnum dgop bmark))
    (loop (when (zerop dgop1)
            (return-from dgop*dgop (logxor dgop (ash dgop2 bmark))))
          (when (zerop dgop2)
            (return-from dgop*dgop (logxor dgop (ash dgop1 bmark))))
          (cond ((evenp dgop1)
                 (when (oddp dgop2) (incf dgop (2-exp bmark)))
                 (setf dgop2 (ash dgop2 -1)))
                (t (incf dgop (2-exp bmark))))
          (setf dgop1 (ash dgop1 -1))
          (incf bmark))))
```

This definition receives as input two fixnum natural numbers `dgop1` and `dgop2` (encoding two degeneracy lists) and executes a loop that uses two local variables `dgop` and `bmark` storing respectively the (partially computed) result, and the number of elements of `dgop` already scanned. When one of the degeneracy lists is exhausted, it stops and returns the concatenation of `dgop` and the remaining elements of the other list. Otherwise, it updates the two local variables (according to the values of the first elements of `dgop1` and `dgop2`) and executes again the body of the loop, removing the first element of `dgop1`, and eventually the first element of `dgop2`.

[2] In the following, to distinguish ACL2 code from general Common Lisp code, we will use italics for the latter.

Since the function `dgop*dgop` deals with natural numbers, we emphasize again that logical operators are used to treat them as binary lists. For example, computing `(logxor dgop (ash dgop2 bmark))` is equivalent to "concatenate" `dgop` and `dgop2` (since `bmark` is the length of `dgop`). Or, for example, `(ash dgop1 -1)` is equivalent to remove "the first element" of `dgop1`. These logical operators on fixnum numbers are usually computed in Common Lisp very efficiently, and this is one of the reasons why Kenzo performs much better than EAT. On the negative side, the formal verification of `dgop*dgop` seems a hard task. In the rest of this section, we present a definition of `dgop*dgop` in ACL2 (trying to keep as close as possible to its original Common Lisp definition) and we state the theorem we want to prove in order to increase our confidence in the way Kenzo deals with degeneracy lists.

4.1 Definition of `dgop*dgop` in ACL2

Since the ACL2 programming language is a subset of Common Lisp, the definition of `dgop*dgop` in ACL2, based on the above Common Lisp code, is quite direct. Nevertheless, due to the applicative nature of ACL2, there are some things that have to be defined in a different (but equivalent) way. In particular, the only way to iterate in ACL2 is by means of recursion. Thus, we use an auxiliary recursive definition implementing the internal loop, trying to be as faithful as possible to the original version. Also, since destructive updates are not allowed in ACL2, we consider the local variables `dgop` and `bmark` as extra input parameters. Finally, since ACL2 functions have to be total, we have to define a result just in case the inputs were not of the intended type (`(type fixnum dgop1 dgop2)`). Taking all these considerations into account, the following is the ACL2 definition of the loop[3]:

```
(defun dgop*dgop-loop (dgop1 dgop2 dgop bmark)
  (if (and (natp dgop1) (natp dgop2))
      (cond ((zerop dgop1) (logxor dgop (ash dgop2 bmark)))
            ((zerop dgop2) (logxor dgop (ash dgop1 bmark)))
            ((evenp dgop1)
             (dgop*dgop-loop (ash dgop1 -1) (ash dgop2 -1)
                             (if (oddp dgop2)
                                 (+ dgop (ash 1 bmark))
                                 dgop)
                             (+ bmark 1)))
            (t (dgop*dgop-loop (ash dgop1 -1) dgop2
                               (+ dgop (ash 1 bmark)) (+ bmark 1))))
      0))
```

Finally, the ACL2 definition of `dgop*dgop` is a call to the above auxiliary function, with suitable initial zero values for `dgop` and `bmark`:

```
(defun dgop*dgop (dgop1 dgop2)
  (dgop*dgop-loop dgop1 dgop2 0 0))
```

We claim that the ACL2 version is faithful with the original Kenzo definition, since we have tried to keep it as similar as possible. As we have said, the fact that

[3] `(2-exp n)` returns 2^n, the same as `(ash 1 n)`; we will comment more on this in the conclusions.

ACL2 is a subset of Common Lisp makes this translation almost direct. Anyway, we strengthened our claim by an intensive testing. Since both definitions can be executed on any compliant Common Lisp, it was very easy to (successfully) test that they return the same result for all pairs of inputs n and m, with $n, m \leq 10000$.

4.2 Stating the Correctness Property of `dgop*dgop`

We now describe how we state the main theorem about the correctness of the above ACL2 definition. It is clear that we would like to prove that the function computes, using the natural number encoding, the composition of two degeneracy lists. Degeneracy lists have been defined in Section 2 as strictly decreasing lists of natural numbers.

Therefore, the first thing we have to define in ACL2 is the composition of degeneracy lists, represented as strictly decreasing lists. That will be our "specification" of the intended behavior of any implementation of composition of degeneracy lists. Note that, in principle, the computation carried out by `dgop*dgop` has nothing to do with the definition given in section 2. While the original definition is based on successive applications of degeneracy operators onto a degeneracy list, the function `dgop*dgop` makes some kind of "merge" between the binary representation of degeneracy lists. As we have said before, the EAT system (the Kenzo predecessor) used strictly decreasing lists of natural numbers to represent degeneracy lists. Thus, it seems a good idea to prove the equivalence (modulo the change of representation) of the Kenzo function with the corresponding EAT function.

In EAT, the composition of degeneracy lists is defined as an iterative application of the equation $\eta_i \eta_j = \eta_{j+1} \eta_i$, when $i \leq j$. The following is the real code for the EAT definition of composition. Note that the auxiliary function `cmp-s-ls` implements the application of a degeneracy operator to a degeneracy list; this function is iteratively used by the main function `cmp-ls-ls` to define composition:

```
(defun cmp-s-ls (s ls)
  (declare (type fixnum+ s) (type list ls))
  (do ((p ls (cdr p))
       (rsl (list ) (cons (1+ (car p)) rsl)))
      ((endp p) (nreverse (cons s rsl)))
    (declare (type list p rsl))
    (when (> s (car p)) (return (nreconc (cons s rsl) p)))))

(defun cmp-ls-ls (ls1 ls2)
  (declare (type list ls1 ls2))
  (do ((p (reverse ls1) (cdr p))
       (rsl ls2 (cmp-s-ls (car p) rsl)))
      ((endp p) rsl)
    (declare (type list p rsl))))
```

We have defined ACL2 versions of these functions, trying to keep as faithful as possible with the original code. Analogously to the previous subsection, a `do` loop has to be replaced by auxiliary recursive functions. These are our ACL2 definitions for composition of degeneracy lists:

```
(defun cmp-s-ls-do (s p rsl)
  (cond ((endp p) (reverse (cons s rsl)))
        ((> s (car p)) (nreconc (cons s rsl) p))
        (t (cmp-s-ls-do s (cdr p) (cons (1+ (car p)) rsl)))))

(defun cmp-s-ls (s ls)
  (cmp-s-ls-do s ls nil))

(defun cmp-ls-ls-do (p rsl)
  (cond ((endp p) rsl)
        (t (cmp-ls-ls-do (cdr p) (cmp-s-ls (car p) rsl)))))

(defun cmp-ls-ls (ls1 ls2)
  (cmp-ls-ls-do (reverse ls1) ls2))
```

Again, the translation from the real Common Lisp code of EAT to the ACL2 version is quite straightforward. But in order to strengthen even more our confidence in this "model", we did intensive testing, checking that they compute the same results for 100000 inputs randomly generated.

We now have to define functions relating the encoding used by Kenzo and the representation of degeneracy list used by EAT. First, the function `dgop-ext-int` transforms a degeneracy list represented as a strictly decreasing list of natural numbers (checked by the function `dgl-p`) to its corresponding representation as a natural number. Note the use of logical arithmetic operators:

```
(defun dgop-ext-int (ext-dgop)
  (if (dgl-p ext-dgop)
      (if (endp ext-dgop)
          0
          (logxor (ash 1 (car ext-dgop))
                  (dgop-ext-int (cdr ext-dgop))))
      0))
```

We also define the function `dgop-int-ext`, its inverse. For that, we use an auxiliary recursive definition that simulates a `do` loop, with the input variables `rslt` and `bmark`, that work as extra parameters for storing respectively the result (partially) computed and the number of binary digits analyzed. The main function simply calls this auxiliary definition with suitable initial values for the extra parameters. This is our ACL2 definition:

```
(defun dgop-int-ext-do (dgop rslt bmark)
  (if (natp dgop)
     (if (zerop dgop)
        rslt
        (if (oddp dgop)
            (dgop-int-ext-do (ash dgop -1) (cons bmark rslt) (1+ bmark))
            (dgop-int-ext-do (ash dgop -1) rslt (1+ bmark))))
     nil))

(defun dgop-int-ext (dgop)
  (if (natp dgop)
      (dgop-int-ext-acc dgop nil 0)
      nil))
```

It should be emphasized that these definitions are defined trying to be as close as possible to the corresponding Kenzo definitions of these operations (although due to the lack of space we do not include here this part of the Kenzo code).

We have now defined all the functions that we need for stating the correctness property of `dgop*dgop`. This property expresses that for every pair of degeneracy

lists represented as strictly decreasing lists of natural numbers, the result of computing `dgop*dgop` on their corresponding encoding as natural numbers is equal to the encoding of the result of the composition carried out by the EAT system. The following is the corresponding ACL2 theorem stating that property:

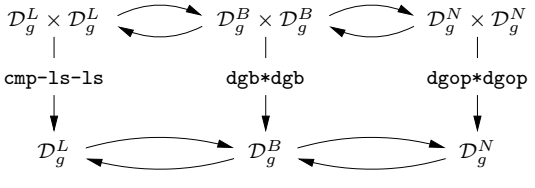
```
(defthm dgop*dgop-cmp-ls-ls
  (implies (and (natp dgn1) (natp dgn2))
           (equal (dgop*dgop dgn1 dgn2)
                  (dgop-ext-int (cmp-ls-ls (dgop-int-ext dgn1)
                                           (dgop-int-ext dgn2))))))
```

In the next section, we will explain how we carried out a mechanical proof of this theorem in ACL2.

5 The Proof: Transforming the Domain

The ACL2 proof of the above result is not simple, mainly for two reasons. Firstly, the functions `dgop*dgop` (Kenzo) and `cmp-ls-ls` (EAT) deal with different representations of degeneracy lists. Secondly, the Kenzo function implements an algorithm which is not intuitive and quite different from the algorithm of the EAT version, which is closely related to the mathematical definition. A suitable strategy to attack the proof is to try to solve the above two questions separately. Thus, it seems natural to consider an intermediate representation of degeneracy lists based on the binary lists described at the beginning of Section 4.

Our plan will be to define a function `dgb*dgb` implementing composition of degeneracy lists represented as binary lists, following the same algorithm than `dgop*dgop`, except for the use of this intermediate representation. This will allow us to prove the equivalence of `dgop*dgop` and `dgb*dgb` dealing only with the encoding aspects. After that, we will prove the equivalence of `dgb*dgb` and `cmp-ls-ls`, focusing only on the algorithmic aspects of the Kenzo definition. Schematically, if $\mathcal{D}_g^L$ denotes the set of strictly decreasing lists of natural numbers, $\mathcal{D}_g^B$ the set of binary lists and $\mathcal{D}_g^N$ the set of natural numbers, we will prove the commutativity of the following diagram (in which, for the sake of clarity, we have omitted the names for the encoding and decoding functions between the different representations):



The rest of this section will be devoted to explain our proof. We will describe separately the properties concerning each of the three representations (or domains) considered, and finally we will show how we compose all these results to achieve the desired theorem.

5.1 The Domain $\mathcal{D}_g^L$

The tail-recursive definitions of functions `cmp-s-ls` and `cmp-ls-ls` that we adopted (since we wanted to keep as close as possible to the EAT version) are not the best option for reasoning in ACL2. Therefore, we proved that these functions verify the following simple recursive schemata, which are much more suitable for the induction heuristics of the ACL2 prover:

```
(defthm cmp-s-ls-recursive
  (equal (cmp-s-ls s ls)
         (cond ((endp ls) (list s))
               ((> s (car ls)) (cons s ls))
               (t (cons (1+ (car ls)) (cmp-s-ls s (cdr ls)))))))

(defthm cmp-ls-ls-recursive
  (equal (cmp-ls-ls ls1 ls2)
         (cond ((endp ls1) ls2)
               (t (cmp-s-ls (car ls1) (cmp-ls-ls (cdr ls1) ls2))))))
```

If we use these alternative recursive schemata, instead of the original versions, it turns out that some properties of `cmp-ls-ls` (for example, its associativity) can be proved very easily in ACL2.

5.2 The Domain $\mathcal{D}_g^B$

The following is the definition of the functions `dgl->dgb` and `dgb->dgl` implementing the change of representation between the domains $\mathcal{D}_g^L$ and $\mathcal{D}_g^B$:

```
(defun dgb-pos (n)
  (cond ((zp n) '(1))
        (t (cons 0 (dgb-pos (- n 1))))))

(defun dgb-app (dgb1 dgb2)
  (cond ((endp dgb1) dgb2)
        ((endp dgb2) dgb1)
        (t (cons (car dgb1) (dgb-app (cdr dgb1) (cdr dgb2))))))

(defun dgl->dgb (dgl)
  (cond ((endp dgl) nil)
        (t (dgb-app (dgl->dgb (cdr dgl)) (dgb-pos (car dgl))))))

(defun 1+ls (lst)
  (cond ((endp lst) nil)
        (t (cons (1+ (car lst)) (1+ls (cdr lst))))))

(defun dgb->dgl (dgb)
  (cond ((endp dgb) nil)
        ((eql (car dgb) 0) (1+ls (dgb->dgl (cdr dgb))))
        (t (append (1+ls (dgb->dgl (cdr dgb))) (list 0)))))
```

These functions are bijections between $\mathcal{D}_g^L$ and $\mathcal{D}_g^B$, and therefore they are indeed a change of representation between two different encodings. The following theorems proved in ACL2 establish that fact (where `dgl-p` and `dgb-p` are respectively functions checking membership to $\mathcal{D}_g^L$ and $\mathcal{D}_g^B$):

```
(defthm dgb->dgl-dgl->dgb
  (implies (dgl-p dgl)
           (equal (dgb->dgl (dgl->dgb dgl)) dgl)))

(defthm dgl->dgb-dgb->dgl
  (implies (dgb-p dgb)
           (equal (dgl->dgb (dgb->dgl dgb)) dgb)))
```

Now we define the composition of degeneracy lists in $\mathcal{D}_g^B$, following the same algorithmic procedure than in the Kenzo version. Recall from section 4 that the result of composing two binary lists $d^B_{g\,1} \circ d^B_{g\,2}$, is obtained by sequentially replacing the 0's in the first list by the successive elements of the second list, until one of the two lists is exhausted; and then completing the result with the remaining elements of the other list. This is precisely what the recursive function `dgb*dgb` does:

```
(defun dgb*dgb (dgb1 dgb2)
  (if (and (dgb-p dgb1) (dgb-p dgb2))
      (cond ((endp dgb1) dgb2)
            ((endp dgb2) dgb1)
            ((eql (car dgb1) 0)
             (cons (car dgb2) (dgb*dgb (cdr dgb1) (cdr dgb2))))
            (t (cons 1 (dgb*dgb (cdr dgb1) dgb2))))
      nil))
```

As expected, the following theorem can be proved, establishing the equivalence of the function `dgb*dgb` in $\mathcal{D}_g^B$ and the function `cmp-ls-ls`, based on the EAT version:

```
(defthm dgb*dgb-cmp-ls-ls
  (implies (and (dgb-p dgb1) (dgb-p dgb2))
       (equal (dgb*dgb dgb1 dgb2)
          (dgl->dgb (cmp-ls-ls (dgb->dgl dgb1)
                               (dgb->dgl dgb2))))))
```

5.3 The Domain $\mathcal{D}_g^N$

The following is the definition of the functions `dgb->dgn` and `dgn->dgb` implementing the change of representation between the domains $\mathcal{D}_g^B$ and $\mathcal{D}_g^N$. Recall that this is simply done by considering the elements of $\mathcal{D}_g^B$ as the reverse of the binary notation of a natural number:

```
(defun dgb->dgn (dgb)
  (cond ((endp dgb) 0)
    (t (+ (car dgb) (ash (dgb->dgn (cdr dgb)) 1)))))

(defun dgn->dgb (dgn)
  (cond ((zp dgn) nil)
    (t (cons (if (evenp dgn) 0 1) (dgn->dgb (ash dgn -1))))))
```

As in the previous subsection, it can be proved that these functions define a change of representation between two different encodings. That is, they are bijections between $\mathcal{D}_g^B$ and $\mathcal{D}_g^N$, as established by the following theorems:

```
(defthm dgb->dgn-dgn->dgb
  (implies (natp dgn)
           (equal (dgb->dgn (dgn->dgb dgn)) dgn))))

(defthm dgn->dgb-dgb->dgn
  (implies (dgb-p dgb)
           (equal (dgn->dgb (dgb->dgn dgb)) dgb)))
```

The following theorem proves the equivalence between the functions `dgb*dgb` and `dgop*dgop` (modulo the change of representation):

```
(defthm dgop*dgop-dgb*dgb
  (implies (and (natp dgn1) (natp dgn2))
       (equal (dgop*dgop dgn1 dgn2)
          (dgb->dgn (dgb*dgb (dgn->dgb dgn1)
                     (dgn->dgb dgn2))))))
```

5.4 Correctness of Kenzo Degeneracy Lists Composition

Having proved the equivalences of both `dgop*dgop` and `cmp-ls-ls` with the intermediate function `dgb*dgb`, the final step is to prove that the functions `dgop-int-ext` and `dgop-ext-int` are equivalent to the composition of the corresponding transformations between the domains $\mathcal{D}_g^L$, $\mathcal{D}_g^B$ and $\mathcal{D}_g^N$. That is:

```
(defthm dgop-int-ext-dgb->dgl-dgn->dgb
  (implies (natp dgn)
           (equal (dgop-int-ext dgn) (dgb->dgl (dgn->dgb dgn)))))

(defthm dgop-ext-int-dgb->dgn-dgl->dgb
  (implies (dgl-p dgl)
           (equal (dgop-ext-int dgl) (dgb->dgn (dgl->dgb dgl)))))
```

Now we only have to glue together the different pieces, using these last properties and the equivalences of the previous subsections, and finally obtaining the main correctness property we wanted to prove:

```
(defthm dgop*dgop-cmp-ls-ls
  (implies (and (natp dgn1) (natp dgn2))
           (equal (dgop*dgop dgn1 dgn2)
                  (dgop-ext-int (cmp-ls-ls (dgop-int-ext dgn1)
                       (dgop-int-ext dgn2))))))
```

That is, we have established the correctness of the function `dgop*dgop` (based on the Kenzo code) with respect to the specification defined by the function `cmp-ls-ls` (based on the EAT code).

6 Translating Properties from $\mathcal{D}_g^L$ to $\mathcal{D}_g^N$

Once we have proved the main correctness theorem, it is easy to prove a property about `dgop*dgop` by first proving the property about `cmp-ls-ls` (which is usually much simpler) and then translating it to `dgop*dgop`, by means of the above theorem. Let us illustrate this with an example.

One of the properties assumed as an axiom in the definition of simplicial set is the following equation between degeneracy operators: $\eta_i\eta_j = \eta_{j+1}\eta_i, \forall i \leq j$. That is, for every pair of natural numbers $i \leq j$ and every degeneracy list d_g, we have $\eta_i(\eta_j(d_g)) = \eta_{j+1}(\eta_i(d_g))$. With respect to the composition of degeneracy lists, the property is stated as follows:

$$\forall i, j \in \mathbb{N}, \forall d_g \in \mathcal{D}_g : \; i \leq j \rightarrow [i] \circ ([j] \circ d_g) = [j+1] \circ ([i] \circ d_g)$$

This property should be true for any implementation of the composition operation. In particular, that is the case for the function `cmp-ls-ls`, as shown in

the theorem below. ACL2 can prove this property immediately, using the simpler recursive schemata presented in subsection 5.1:

```
(defthm cmp-ls-ls-property
  (implies (<= i j)
           (equal (cmp-ls-ls (list i) (cmp-ls-ls (list j) dg))
                  (cmp-ls-ls (list (+ 1 j)) (cmp-ls-ls (list i) dg)))))
```

Now, this allows us to prove in a quite straightforward manner the corresponding version of this theorem for `dgop*dgop`. It is an easy consequence of the above theorem, the theorem `dgop*dgop-cmp-ls-ls` of the previous section, and the relations between the functions `dgop-int-ext` and `dgop-ext-int` and the transformations between the domains $\mathcal{D}_g^L$, $\mathcal{D}_g^B$ and $\mathcal{D}_g^N$. This results in the following:

```
(defthm dgop*dgop-property
  (implies (and (natp dgop) (natp i) (natp j) (<= i j))
           (equal (dgop*dgop (dgop-ext-int (list (+ 1 j)))
                             (dgop*dgop (dgop-ext-int (list i)) dgop))
                  (dgop*dgop (dgop-ext-int (list i))
                             (dgop*dgop (dgop-ext-int (list j)) dgop)))))
```

In a similar way, we have also proved that the function `dgop*dgop` is associative, from the same property for `cmp-ls-ls`, whose proof is very simple.

7 Conclusions and Further Work

In this paper we have described an ACL2 proof of the correctness of a first-order fragment of the Kenzo system. Concretely, the Kenzo programs dealing with degeneracy lists have been certified. Although the verified fragment is short in number of lines, it is important for efficiency reasons in Kenzo.

As for the proof effort, we followed "The Method" described in [10] and outlined in Section 3. We recall that although every proof attempt of the system is fully automatic, the system can be seen as interactive, since the appropriate lemmas has to be previously proved in order to obtain the proof. Thus, and following "The Method", when a proof attempt of a result failed, we inspected its output to discover which lemmas were needed to lead the prover to a successful proof. Most of the resulting proofs were carried out by induction and simplification. In most cases, the heuristics of the prover were able to automatically find a suitable induction scheme. Only in a few cases, we needed to supply an explicit induction scheme. All our interaction with the prover resulted in a collection of 27 definitions and 112 theorems. It is interesting to point out that we had no preconceived proof in mind, and that all we did was to follow the suggestions from the failed proof attempts. We urge the interested reader to consult the complete development in [13].

From a practical point of view, a library of results about the logical arithmetic operands was very useful. This library contains results previously proved by other ACL2 users and comes with the ACL2 distribution. Thus, we think this is a good example of reusability.

It is also worth pointing out the methodology devised to formally verify a system written in Common Lisp. Obviously, we are not directly verifying the actual

code, due to limitations of the ACL2 programming language. But since ACL2 is a subset of Common Lisp, a "model" very closely related to the original code can be defined. And since ACL2 functions can be executed in any compliant Common Lisp, we could do intensive testing to strengthen even more the assumption that our model is faithful. Another remarkable point is our use of a previous version of Kenzo, called EAT, as a main component of the specification of the intended properties. This also increases the trust in the correctness of the methods appearing in both EAT and Kenzo.

In this paper, we have not dealt with efficiency issues. In fact, there are two technical details in the verified ACL2 function which make it less efficient than its Kenzo counterpart. The first one is that the ACL2 function `dgop*dgop-loop` has an explicit test in its body, checking that its first two arguments are natural numbers. This is needed to ensure termination of the function on *all possible inputs*, as required by the principle of definition of the ACL2 logic ([10]). That explicit condition has a negative impact on the efficiency of the ACL2 algorithm, since it is checked in *every* recursive call. The other technical detail that affects efficiency has to do with how 2^n is computed by Kenzo: at initialization, a lookup table is built, with the powers of two until the biggest fixnum; after that, every time a power of two is needed, the function `(2-exp n)` used by Kenzo simply retrieves the value from position `n` of the table. In contrast, our ACL2 function computes `(ash 1 n)`, which is an equivalent, but less efficient method. Although in a first stage we have not dealt with this issues, both technical details can be solved in ACL2, using the `defexec` and `stobj` features, respectively (see the users manual in [11] for details). We plan to introduce these improvements in our ACL2 code and formally verify them, in order to obtain a certified algorithm, comparable in efficiency with the Kenzo algorithm.

The work presented here is a first approach on using ACL2 with the purpose of certifying fragments of an already implemented system as Kenzo (that is, we do not want to reimplement the system, but to certify the existing code). This case study shows the benefits of the fact that both systems (Kenzo and ACL2) deal with the same programming language. Nevertheless, further research has to be done to test how ACL2 will behave with two important issues not addressed here. First, the mathematical theory underlying most Kenzo computations (algebraic topology) is more complex than the needed by this example. Second, Kenzo intensively uses higher order programming, not allowed in ACL2.

Thus, our future work will follow two lines of research: first, we intend to formalize in ACL2 some results of algebraic topology, which will allow us to tackle more difficult algorithms, such as the one extracted from the Eilenberg-Zilber theorem, where the combinatorial explosion of simplicial degeneracy *shuffles* appears [14]. Or for example to verify the Kenzo builders (to construct spheres, Moore spaces, projective spaces, ...) which are used as primitives in the *reKenzo* graphical user interface [9]. Other line of research will be to study how we can model the higher-order features used by the Common Lisp Kenzo code, in a first-order Common Lisp ACL2 code; after that, we will be able to compare this approach with the alternative of using higher-order theorem provers like Coq, PVS or HOL.

Acknowledgements

In memoriam of Mirian Andrés, our colleague and, much more important, our friend.

References

1. Andrés, M., Lambán, L., Rubio, J.: Executing in Common Lisp, Proving in ACL2. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/ CALCULEMUS 2007. LNCS, vol. 4573, pp. 1–12. Springer, Heidelberg (2007)
2. Andrés, M., Lambán, L., Rubio, J., Ruiz-Reina, J.L.: Formalizing Simplicial Topology in ACL2. In: ACL2 Workshop 2007, University of Austin, pp. 34–39 (2007)
3. Aransay, J., Ballarin, C., Rubio, J.: A Mechanized Proof of the Basic Perturbation Lemma. Journal of Automated Reasoning 40, 271–292 (2008)
4. Aransay, J., Ballarin, C., Rubio, J.: Extracting Computer Algebra Programs from Statements. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2005. LNCS, vol. 3643, pp. 159–168. Springer, Heidelberg (2005)
5. Coquand, T., Spiwack, A.: Towards Constructive Homological Algebra in Type Theory. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/ CALCULEMUS 2007. LNCS, vol. 4573, pp. 40–54. Springer, Heidelberg (2007)
6. Domínguez, C.: Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS, vol. 5144, pp. 270–284. Springer, Heidelberg (2008)
7. Domínguez, C., Lambán, L., Rubio, J.: Object Oriented Institutions to Specify Symbolic Computation Systems. Rairo - Theoretical Informatics and Applications 41, 191–214 (2007)
8. Dousson, X., Rubio, J., Sergeraert, F., Siret, Y.: The Kenzo Program, Institut Fourier (1999), `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`
9. Heras, J., Pascual, V., Rubio, J.: Mediated Access to Symbolic Computation Systems. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS, vol. 5144, pp. 446–461. Springer, Heidelberg (2008)
10. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (2000)
11. Kaufmann, M., Moore, J.S.: ACL2 Home Page, `http://www.cs.utexas.edu/users/moore/acl2`
12. Lambán, L., Pascual, V., Rubio, J.: An Object-Oriented Interpretation of the EAT System. Applicable Algebra in Engineering, Communication and Computing 14, 187–215 (2003)
13. Martín–Mateos, F.J., Ruiz–Reina, J.L., Rubio, J.: ACL2 verification of simplicial degeneracy programs in the Kenzo system, `http://www.cs.us.es/~fmartin/acl2/kenzo`
14. May, J.P.: Simplicial Objects in Algebraic Topology. Van Nostrand (1967)
15. Rubio, J., Sergeraert, F., Siret, Y.: EAT: Symbolic Software for Effective Homology Computation, Institut Fourier (1997), `ftp://ftp-fourier.ujf-grenoble.fr/pub/EAT`
16. Steele Jr., G.L.: Common Lisp The Language, 2nd edn. Digital Press (1990)