Constructive Algebraic Topology

Julio Rubio – Francis Sergeraert

[redacted]

The question of computability in topology
is a delicate one.

G. Whitehead [53]

1 Introduction.

1.1 What does "constructive" mean?

It would not be reasonable to "recall" here what the *constructive* point of view is when a mathematical theory is considered; examining the modern reference [52] will show you the subject is rich. A simplified version, here sufficient, consists in requiring that any construction process must lead to, or more simply must *be*, an *algorithm* constructing the result when coherent data are provided; in other words, it is a matter of *computability*. The theory can then be strongly modified.

The following example is striking: in ordinary mathematics, if an increasing real sequence (x_n) is bounded, this sequence has a limit ℓ; from the constructive point of view, a real number is an algorithm associating to the natural number i a rational approximation p_i/q_i satisfying $|p_i/q_i - p_j/q_j| < 2^{-i}$ if $i < j$. Then, it is possible to *explicitly* construct an increasing rational sequence (x_n), explicitly bounded, satisfying the following property: for every real number ℓ, you can *explicitly* construct a number N and a positive number ε such that $n \geq N \Rightarrow |x_n - \ell| > \varepsilon$; in other words, the classical result about the bounded increasing real sequences becomes strongly false. See the entry *Specker sequence* in the index of [52].

1.2 A new research field.

The evolution of Algebraic Topology with respect to the computability problem is strange and this long introduction is devoted to clarifying a complex situation. Several previous tentative versions of this paper have been processed by referees and commentators in a rather amazing way; see [42]. This work concerns an actual *research* field; the first writings presented many defects for two reasons: the subject is new and all must be invented about what a good writing style could be; on the other hand, experience shows "standard" culture in computer science is relatively poor among topologists and this favours numerous misunderstandings. We understand the referees required to give an opinion on this work are a little puzzled, but this does not excuse some pathological behaviours: the computability questions, theoretical or concrete, put numerous colleagues into a strange state; we must recall that, whatever the situation is, it is better to behave responsably and, why not, friendly.

But the subject is not the ethology of our profession, it is mathematics. Imagine a mathematical world where Arithmetic would be as you know it, with an exception: if a *random* naturel number (say $< 10^{10}$) is given, you would be in general unable to determine its prime divisor set, that is, the first classical invariant for a natural number; probably you would consider such a situation is rather strange; instead of working on the large roots of the ζ function, maybe you would find more urgent to design an algorithm to compute these divisor sets?

This is precisely the situation in Algebraic Topology. On one hand, a marvelous work has been done in the last fifty years, an idea of which could be given by the sections of [28]. Well, but let us suppose a *random* small 1-reduced[1] simplicial set (small could mean less than 50 simplices) is given; please, could you determine the first homotopy groups of it? Noone knows today how to solve this problem. Do not object this is an exercise theoretically easy, in fact tedious and without real interest; this is false: the *theoretical* complexity of the solutions proposed elsewhere, see below, their beauty also, refute such an appreciation. The goal of the present paper is on the contrary to give a conceptually simple solution; but the key concepts, namely the notions of *locally effective* object and of object *with effective homology*, are *new* and obviously at the origin of our difficulties with the previous referees.

1.3 Finiteness of homotopy groups.

It is elementary to prove the homology groups of finite polyhedrons are computable. The problem begins to become serious with homotopy groups. The first progress in this matter is the general finiteness result obtained by Jean-Pierre Serre [44]: the homotopy groups of finite simply connected polyhedrons are of finite type; more generally, the classical "reasonable" constructions of algebraic topology give new spaces with invariants of finite type at least if the standard hypotheses about simple connectivity are satisfied. Once the usual spectral sequences are available, the result is obtained in a natural way, a so natural way that it can seem clear it is a simple exercise to extend Serre's results to obtain the corresponding results of computability. In fact the subject is a little more complicated.

[1] Only one vertex, no edge.

1.4 Computability of homotopy groups.

Two difficulties, very different from each other, are met to transform the spectral sequences into algorithms. These difficulties, detailed later in the paper, are so serious that Edgar Brown [11], proving the first and, for a long time, the only actual result about the computability of homotopy groups, avoids using spectral sequences! Furthermore, Brown declared in his introduction his method has only a theoretical interest, because much too complicated to be concretely applied. Such an appreciation is still valid forty years later, even with the most powerful computers you could imagine. The question of a computation method which on the contrary could be used in a significant number of cases remained open.

The paper of Edgar Brown is divided in two parts; the first one uses the Postnikov tower method to process the case of simply connected simplicial sets with *finite* homology groups; this means each homology group is really finite, not of finite type: this first method does not work for the 2-sphere. The first part is relatively easy and is essentially a nice consequence of Serre's result about the equivalence between finiteness of homology groups and finiteness of homotopy groups. But how to process for example the case of the 2-sphere?

The second part of Brown's paper undertakes a complex approximation work to extend the result of the first part to simplicial sets with homology groups of finite *type*, not necessarily finite. Actual programming of such a method, using an algorithm of very dynamic style, would be a high level programming work. No details about programming techniques were included in Brown's paper; the implicit hypothesis was the following: if a computing plan can be defined which could be "obediently" followed by an arbitrary (!?) person, a program can be written implementing it. The computing program of Brown consists in writing definitions of finite simplicial sets, free $\mathbb{Z}$-modules of finite type, computing kernel and image of various homomorphisms, constructing new simplicial sets, and so on. When Brown's paper was proposed to[2] the Annals of Mathematics, writing such a machine program was theoretically possible, but the programming languages of this time were much too poor for an actual work. Furthermore simple calculations of time complexity prove such a writing work would not have given any serious application...

1.5 Through a desert.

And the subject was abandonned for thirty years. During this time, an enormous and wonderful work was undertaken in algebraic topology in many fields, but the problem of designing algorithms able to compute homological, homotopical, K-theoretical... invariants for a significant number of *arbitrary* data was not really considered.

[2]And accepted by.

1.6 Homology of loop spaces.

A consequence of Serre's discovery about the essential role of *loop spaces* soon led Frank Adams to define the cobar construction [1]; after all this construction is nothing but an algorithm allowing one to compute the homology groups of a loop space, when a model of the initial space with an Alexander-Whitney diagonal (natural coalgebra structure) is known. If this construction could be *iterated*, a solution for most of the computability problems in algebraic topology would be available, but the cobar construction *does not carry* a natural coalgebra structure and the problem of iterating the cobar construction then became an essential obstacle.

1.7 The second cobar construction.

Twenty-four years passed before Hans Baues finds a solution for the second cobar construction, based on a nice geometric interpretation of the first cobar construction [8, 9]. But this second cobar construction cannot be iterated either! Iterating the cobar construction undoubtedly seemed a hard problem.

1.8 How to understand David Anick.

A little later, David Anick obtained negative results about computability in algebraic topology [4]: for example computing homology groups of ΩX, where X is a finite simply connected simplicial set, is at least NP-hard[3]. Such a result is usually understood as follows: the considered problem is so complex that no algorithm can be used for a large number of cases, so that trying to find such an algorithm in order to use it is wasting time. But this appreciation is erroneous for at least three different reasons.

On one hand, the comparison problem between the NP and P-complexities is today *open*. The statement of this problem is outstandingly simple and its practical importance is enormous: most of the (secret) coding methods would become dangerous if ever the complexities P and NP were proved equal; in fact $P = NP$ would imply the usual codes can be more or less easily broken! Every mathematician having understood the problem considers it is the most important[4] one in the mathematical landscape of today. It is generally conjectured that $P \neq NP$, but if $P = NP$, Anick's objection[5] disappears. And this situation on the contrary gives further reasons to study algorithms computing homology of loop spaces; after all, the topological framework, rather original with respect to what is usual in computer science, could give new insights into the problem P vs NP.

[3] The reference [24] gives the basic informations about NP-complexity (non-deterministic polynomial time complexity).

[4] Important does not necessarily imply difficult.

[5] To our knowledge, David Anick never *objected* anything on this subject; this is only a shorthand for the more precise expression *the objection you could state according to Anick's result.*

On the other hand, saying an algorithm with non-polynomial time complexity has no interest is a little ridiculous. Many undefined or unknown "constants" take place in these questions of time estimates, which often remove any interest from these appreciations. Numerous polynomial algorithms have never been implemented because evidence shows time saving would be negligible or even negative for actual problems. Conversely the algorithms determining Groebner basis for polynomial ideals have a hyper-exponential complexity; however an enormous work has been devoted to them; is it an error? Anyway even if theory and/or experience would show an algorithm is without any practical use, this algorithm is a *mathematical* object a priori as interesting as any other[6]: for example the problem $P =$ or $\neq NP$ is nothing but a *mathematical* question about the most classical *exponential* algorithms; the problem is open, which means these algorithms have deep *mathematical* unknown properties. As to the practical point of view, we give also in this paper a few interesting results about the twisted Eilenberg-Zilber theorem which have been *experimentally* discovered thanks to our hyper-exponential algorithms.

Furthermore the negative result of Anick artificially favours one parameter of the problem and does not consider another one, in fact more important from a practical point of view. Let X be a finite simply connected simplicial set and n an integer. Anick proved computing $H_n\Omega X$ is NP-hard *with respect to the pair* (X, n); on the contrary if you fix n, it is not hard to prove that, if you use the right algorithms to reduce integer matrices (see for example [6]), then the Adams algorithm has polynomial time *with respect to* X. This paper[7] gives the same result for $\pi_n X$: for n fixed, these groups can be computed in polynomial time with respect to X; these algorithms therefore satisfy the sacrosanct rule and can be officially declared interesting to be studied. And so we are closer to the original goal of algebraic topology: classifying *arbitrary* spaces. Of course these considerations do not reduce the interest of Anick's nice results which in fact should be reinterpreted as follows: in studying the *complete* calculation of $\pi_n X$, instead of considering arbitrary big values for n, you should bound the n parameter and consider $\pi_n X$ mainly as a function of X[8].

1.9 Schön's inductive limits.

The beginning of the story, up to 1985, is therefore mainly Serre, Adams, Baues and Anick. From this time, several people considered again the general computability problem in algebraic topology. See the papers by Smirnov [46], Schön [40], Justin Smith [47] and the second author of the present paper [41].

Schön's solution consists in replacing the usual five lemma by an effective one. The traditional five lemma gives a short exact sequence $0 \to A \to X \to C \to 0$

[6] By the way, Adams' cobar construction is an algorithm with *exponential* time complexity; no interest there?

[7] And also [46, 40, 47], see later.

[8] The best later results due to David Anick concern on the contrary the large values of n [5]! No contradiction here: no algorithm computing $\pi_n X$ in [5].

where the $\mathbb{Z}$-modules A and C are known and X is the $\mathbb{Z}$-module to be determined. Nothing is usually given for the solution of the possible extension problem when A and C have torsion. Various considerations sometimes allow to solve it if the situation is not too complicated, but when large intricate sets of five lemmas are involved, for example when processing a spectral sequence, no algorithm is known to conclude[9]. Schön replaces the $\mathbb{Z}$-modules by strongly structured inductive limits of $\mathbb{Z}$-modules — let us call them *Schön modules* — and then proves an *effective* five lemma in this situation: an algorithm working on the given Schön modules A and C *constructs* the unknown one X; this is Schön's main tool, with which he transforms every classical exact or spectral sequence into an effective method computing the unknown object even when complicated torsions take place. The approximation process of the inductive limit by terms of the defining sequence is an elegant, tricky and powerful extension of Edgar Brown's method.

It is not at all obvious how to implement on a theoretical or concrete machine the Schön modules. It is not common in programming to install on a computer an infinite inductive system of $\mathbb{Z}$-modules, moreover in a *dynamic* framework: *during* runtime, new large sets of Schön modules must be constructed, which will construct other ones and so on; and an enormous work must be undertaken before seeing the nice results of Schön transformed into concrete programs; no indications about programming techniques are given in [40]. In fact, the most modern techniques of functional programming are needed to program Schön algorithms and at this time these techniques are not very known among the topologists. The field so opened by Schön is fascinating and studying it should be a major subject for the next years.

1.10 Operads.

The papers by Smirnov [46] and Justin Smith [47] are very different from Schön's one. Both use essentially the same tool: if X is a simplicial set, using the geometric nature of X, it is possible to considerably enrich the structure of the chain complex $C_*(X)$ up to installing a "coalgebra" structure with respect to some particular algebraic operad. If a "reasonable" geometric construction GX is made starting from X, for example the loop space construction, then the information available on the "coalgebra" $C_*(X)$ is sufficient to construct a new chain complex $C_*(GX)$ having the right homology type and, much more important, to install on it an analogous structure: the chain complex $C_*(GX)$ then has the right *homotopy* type; so that the process can be iterated; a solution for the essential problem of iterating the cobar construction is so obtained. The coalgebra structures of Smirnov and Smith can be considered as a large and *complete* extension of the Steenrod cohomological operations. This short summary gives a weak idea of the complexity of their method.

[9]The Bockstein spectral sequence is *not* an answer: the problem is so better structured, but now transferred to its differentials.

1.11 Schön vs Smirnov and Smith.

How to compare Schön's solution on one hand, Smirnov and Smith' solution on the other hand ? The last ones are relatively sophisticated; the operads to be used and the "coalgebras" with respect to them are *algebraic* objects which carry extremely rich structures: such a structure *is* a homotopy type; in other words these structures are a perfect solution for the problem of algebraic topology; topology is entirely transformed into a subdomain of algebra. On the contrary, once you can handle on your machine the Schön modules, his method is rather elementary and its complexity comes only from the large set of objects to put together before seeing a concrete result. But Schön, Smirnov and Smith have not yet considered at this time an actual implementation work, so that the readers of their papers are left a little unsatisfied. But each thing in its proper time: their work is essential and opens large interesting new fields.

Another comparison must be done; Smirnov and Smith' solutions *do not need* functional programming. In a sense their solution is the canonical one if you want to use only standard programming. Roughly speaking, things are as follows: many different constructions can be undertaken if you start of a simplicial set X with the chain complex C_*X. Then, when C_*X is provided with the coalgebra structures defined by Smirnov and Smith, you have a *complete* data base allowing you to answer any homological further question coming from a geometric construction. It is a question of *resolution*: the situation is analogous to the one which is met when you install the bar-construction for a group G: this *unique* object contains an answer for *any* particular homological question concerning a G-fibration: a resolution is nothing but an *initial* object in an appropriate category.

The gigantic size of Smirnov and Smith' operads and coalgebras seems to be a serious obstacle for a concrete use; but these sophisticated objects can be coded in a functional way! A wonderful work field is in front of us here, and it is much too early to appreciate the concrete efficiency of their methods; anyway, the strongly structured style of their organization is a sufficient reason not to give up their point of view.

1.12 Our method.

Our method is very different, from any point of view. The idea consists in working as close as possible to the original scheme due to Serre. So that the question is the following: how to process in order to transform the classical spectral sequences into effective methods, without using complicated objects as Schön modules? Two essential tools are used, which perfectly complete each other. These tools are curiously known for a long time: the first one (perturbation lemma) is known since 1962, and the second one (functional programming), strictly speaking, since 1936! The unique original idea here consists in putting together these essential tools; the result is soon obtained.

1.12.1 The perturbation lemma.

The *homological perturbation lemma* is an elementary but powerful algorithm which can be used in the following situation: let h be an *explicit* chain equivalence between two chain complexes $h : C_1 \to C_2$; let us suppose a perturbation δ_1 of the differential d_1 of C_1 is also given, defining a new chain complex C'_1, with the same underlying graded module, but the new differential $d_1 + \delta_1$. Then, if some smallness (nilpotency) hypotheses are satisfied, it is possible to perturb the whole object h to obtain a new chain equivalence $h' : C'_1 \to C'_2$; it is an implicit function theorem: the relations satisfied by h are strong and determine how the whole object must evolve to follow the evolution of one of its components. The prototype example is the case where C_1 is the chain complex of a non-twisted product, and the perturbation δ_1 comes from a *torsion* of the product, defining a non trivial fibration; then C'_1 is the chain complex of the total space of this fibration. If C_2 describes the homology of the non-twisted product,then the new complex C'_2 describes the homology of the twisted total space [45].

1.12.2 Functional programming.

The homological perturbation lemma is known for a long time and took its definitive form thanks to Shih Weishu [45] and Ronnie Brown [13]. The previous use which is the closest to ours is due to Victor Gugenheim [26]; we shall detail in due time the point reached by Gugenheim and the reason why his work did not give a solution for iterating the cobar construction. We shall explain how *functional programming* gives the ideal tool to extend Gugenheim's work up to a theoretical *and practical* solution for the *iterated* cobar construction.

Note also a spectral sequence is nothing but the by-product of the homological perturbation lemma, if functional programming is not available.

It is difficult to integrate this computer science[10] tool into a work of topology, and also to persuade colleagues that a solution of a purely topological problem, namely how to iterate the cobar construction, can be obtained in this way. This point is the most original part of the present paper. The sections 3 to 8 of the paper are devoted to this technique, finally organized around an appropriate terminology.

We have previously explained Schön's solution also needs functional programming, though the author does not give any explanation about. On one hand our functional technique is significantly simpler: it is only a question of macro-generation; this means runtime is precisely split into two steps: during the first one a (complex) work of automatic program writing is undertaken, but during

[10] Let us recall that theoretical computer science is a subdomain of mathematics; more precisely the negative answer to Hilbert's conjecture (Entscheidungsproblem) comes from the following inverse inclusion relations: on one hand, "computer science" $\subset$ "mathematics"; on the other hand, "mathematics" $\subset$ "computer science". Turing *invented* computer science in general and in particular discovered these relations; it was not easy for the 1954 Fields Medal Jury to realize the case of Turing had to be seriously considered; the fantastic importance of his work, concrete as well as theoretical, will have been recognized only many years later. Alan Turing committed suicid in June 1954, see [27].

the second step, only ordinary calculations are done. On the contrary in Schön's computing plan, functional programming must constantly be used up to the end of calculation. On the other hand, our technique is certainly the closest to the traditional organisation of algebraic topology, so close that some renowned but in this case superficial commentators [42] did not hesitate to publicly declare there is nothing new here; an amusing story.

Finally the relationship between our method and those of Smirnov, Justin Smith and Schön can be quickly described as follows. Without functional programming you must use the terrible resolutions of Smirnov and Justin Smith; if you use the functional programming tool, more elementary solutions can be designed; the first one is due to Schön, which is not so simple; but, if you systematically use functional programming, you can go much further and design a solution extremely close to traditional algebraic topology.

1.12.3 Locally effective objects.

A locally effective object is a set of algorithms modelizing in a *locally effective* way a mathematical object such as a chain complex or a simplicial set. The "locally effective" qualifier means these algorithms are in general unable to give you any *global* information about the underlying object; for example it is not possible in general to decide whether the object is trivial (null or empty) or not. The available algorithms allow the user to get useful informations about some component or other of the object. The underlying object could be a presumably enormous simplicial set; no information is available about the real size of the simplicial set; but if someone presents *any* simplex of this simplicial set, an algorithm can compute its faces.

You have met locally effective objects for a long time. When you use a pocket calculator to compute the product of two integers, you use a locally effective version of the set $\mathbb{Z}$ of integers: your calculator is unable to give you any *global* information about $\mathbb{Z}$; but if you enter two *particular* integers m and n, it is able to compute the product $m \times n$; these integers are particular but also *arbitrary*[11]. We shall see it is possible to install and use locally effective versions of the various enormous objects which must be used to extend Gugenheim's work, and in this way to iterate the cobar construction.

The mandatory technique to implement these algorithm sets, whatever the point of view you are interested in, theoretical or concrete, is *functional programming*, that is, the wonderful tool invented essentially by Gödel, Church and Turing to invalidate Hilbert's conjecture about the existence of a universal algorithm solving any mathematical problem (Entscheidungsproblem). Church's version of functional programming, known as λ-calculus, led to one of the major programming languages of today, namely *Common Lisp*, result of a long and hard work, see [49]. The authors used this programming language to *concretely* implement their theoretical method and to obtain in several particular cases unknown homological

[11] Except for memory size considerations.

groups. The interest of this program is detailed later in this section.

1.12.4 Objects with effective homology.

When the notion of locally effective object is available, it is elementary to define what an *object with effective homology* is, and to use it. It is conceptually a mixed object: on one hand a *locally effective* component codes the underlying and possibly infinite object such as a simplicial set or a chain complex; on the other hand an *effective* chain complex describes the homology groups of the other object; the last component is a chain complex of free $\mathbb{Z}$-modules of finite type, which is *explicitly* chain equivalent with the chain complex canonically associated to the locally effective object: a (strong) chain equivalence is included in the object. If you are interested in the homology groups, you can use the effective component: because it is effective, global information is available and it is elementary to compute the homology groups.

More important, if you use one or several objects A_i with effective homology to construct a new locally effective object B, then the information available in the A_i's often allows you to complete B in order to make of it again an object with effective homology. The process is *stable* and therefore can be iterated; in this way, it is possible to iterate the cobar construction.

1.12.5 The present paper.

The status of the present paper with respect to [41][12] must be given. Strictly speaking, nothing new here; all the main ideas were described in [41]. However, two auxiliary tools are available here, which make considerably easier the exposition, which should make also much easier the understanding.

The first tool is the EAT[13] program [39], providing us with a concrete support to our considerations about functional programming in algebraic topology. These difficult questions, out of scope of "standard" programming culture, are the most easily understood in front of a terminal screen, with the appropriate demonstration. A few people understood our work thanks to the demonstration logfile of [42]. But in the framework of a traditional paper, the exposition work is extraordinarily difficult: the role of the time factor is essential in these questions and practically impossible to explain in an ordinary text. The authors spent much work in the first sections of the present paper making these questions as comprehensible as possible; these didactical but in fact necessary sections could not be included in [41]: the EAT program plays an important role in this connection, and was not available during the writing of [41].

The second tool is the homological perturbation lemma. Its role was not yet understood by the author when [41] was written and you can observe [41] does not quote any paper about it. Again this tool makes much easier the exposition,

[12]Kindly published in Advances in Mathematics.
[13]EAT = Effective Algebraic Topology.

so that we can easily include for example a complete re-proof of Edgar Brown's theorem about the computability of homotopy groups. Please compare to the original proof by Edgar Brown to precisely appreciate the powerfulness of our methods; note also the considerably larger scope of our result: it can be directly applied to the simply connected simplicial sets *with effective homology*; the finite simplicial sets of Edgar Brown are a microscopic subclass. The same large class is also covered by Rolf Schön's result [40], but again compare the simplicity of our organization with respect to Schön's one. Similarly, before thinking that our solution is too easy to attach to it much interest, please read the solutions due to Smirnov [46] and Justin Smith [47]; their papers are interesting, and furthermore you will then be able to judge will full knowledge of the facts.

1.13 The EAT program

.

The EAT (Effective Algebraic Topology) program [39] implements the ideas of this paper about constructive algebraic topology, only in the particular but essential case of iterated loop spaces. It is a 5,000 lines Common Lisp program, which was written by both authors of the present paper, without meeting particular difficulties; only a little patience was necessary.

1.13.1 Computing homology groups.

It is the first time such programs are written, and a large set of technical choices have had to be done. The following story gives a good idea of the importance of technical and also theoretical options: we were interested by the group $H_7\Omega^2\text{Moore}(\mathbb{Z}_2, 4)$ because a loop space specialist explained a $\mathbb{Z}_4$ should be there. In fact this group is $\mathbb{Z}_2$; the first computation for this group spent three days with the good result; a few days later the first author found a *theoretical* bug in the program. We were obliged to replace a program segment by another one, which is significantly more complicated, so that we were surprised to see the result then obtained in one day; moreover the result was the same[14]! We are still unable to explain why the right program, which seems much more complicated than the first one, gives a quicker result; probably the good algorithm in fact implies hidden strong simplifications in intermediate results. A few months later the program was entirely rewritten to integrate numerous improvements which became obvious according to the experiences done. With the new version the same group is computed in 10 minutes. Many further improvements remain to be installed in the program.

The topologists are usually interested in results they are able to compute more quickly than the program. And examples of this sort are numerous. For instance, the paper [10] allows to easily calculate the homology groups $H_*\Omega^2S^3$, but our program has been unable to compute the group $H_8\Omega^2S^3$: it then deads because of

[14] An explanation is given at the end of Section 14.3.

lack of memory[15]. In 20 minutes, the EAT program computes the (right) H_7. But wait a moment, the comparison will soon be inversed.

Similarly, most of the observers declare not to be interested by the groups of the form $H_*\Omega^n\Sigma^n X$: the classical papers [34] and [17] in principle allow to deduce these groups from the homology of X. But experience shows this is not so easy: our program computes for example $H_5\Omega^3\mathrm{Moore}(\mathbb{Z}_2, 4)$ in a few hours; no expert has yet been able to give us the right result, and however the best ones have been questionned; if this group had a real interest they would certainly find the good result, but in such a case our program could be useful for a validity control, for the final result or also for intermediate results as other experiences showed it.

But as soon as a group $H_n\Omega^p X$ must be computed where X is not a suspension, it seems noone can give an answer, even in simple situations. Let us quote the recent paper by Carlsson and Milgram [14, p.545]:

> ... To go further one should study similar models for double loop spaces, and more generally for iterated loop spaces. In principle this is direct. Assume X has no i-cells for $1 \leq i \leq n$ then we can iterate the Adams-Hilton construction of Section 5 and obtain a cell complex which represents $\Omega^n X$. However, the question of determining the boundaries of the cells is very difficult as we already saw with Adams' solution of the problem in the special case that X is a simplicial complex with $sl_1(X)$ collapsed to a point. It is possible to extend Adams' analysis to $\Omega^2 X$, but as we will see there will be severe difficulties with extending it to higher loop spaces except in the case where $X = \Sigma^n Y$.

Let us explain a little bit more explicitly. The solution for ΩX is due to Adams; a (sophisticated) geometrical solution can also be given for $\Omega^2 X$ based on permutohedra, called Zilchgons in [14] (see also [9]). But no such solution was previously known for $\Omega^n X$ if $n > 2$, unless $X = \Sigma^n Y$. Instead of qualifying the difficulties as *severe*, maybe it would have been a little more precise in a mathematical text to explain the problem was simply open? In fact the papers [46], [47], [40] and [41] (none is quoted in [14]) give a complete solution for this problem and many others. The fourth one, the subject of the present paper, is so far the only one which has been concretely applied, thanks to the EAT program.

Let us consider the following example which contains a few severe dificulties. Firstly you attach a 3-disk to ΩS^3 with a map of degree 2 to obtain $X = \Omega S^3 \cup_2 D^3$. Then X is 1-reduced; what about determining the homology groups of ΩX? Those of X are obvious: $\mathbb{Z}, 0, \mathbb{Z}_2, 0, \mathbb{Z}, 0, \mathbb{Z}, \ldots$ It is easy to prove a cellular model for X exists with one cell in even dimensions and a further cell in dimension 3. Most think in this situation it is sufficient to apply Adams-Hilton (or equivalently Eilenberg-Moore); the graded module underlying the cobar construction is obvious but some severe problems are met to establish the right differential. Several loop space specialists have been questioned and so far none succeeded in giving

[15] This group is in fact $\mathbb{Z}_2 \oplus \mathbb{Z}_{30}$, see [10].

an *algorithm* computing the desired homology groups. It is elementary with the EAT program to construct a version *with effective homology* of ΩX, and then the effective component can give the wished homology groups; of course time complexity is hyper-exponential but the EAT program computes $H_n \Omega X$ for $n \leq 7$ in twenty minutes (one hour with a PC-Pentium 100 Mhz). More precisely, computing $H_7(\Omega(\Omega S^3 \cup_2 D^3))$ is exactly as easy with the EAT program as computing $H_7 \Omega^2 S^3$: the added D^3 somewhere in the construction does not significantly change the computing plan.

1.13.2 More important.

Our experience shows the interest of such a program is not at all the computation of some homology group or other[16]. Three more important reasons can be given to devote some work to these programs.

- Locating an actual difficulty: a *complete* program computing some object allows its user to analyse where the actual problem is; it it sufficient for such an analyse to observe during long calculations what program segments are most requested. From this point of view the striking fact brought to light by our work is the following: the most important gap in algebraic topology today is in the twisted Eilenberg-Zilber theorem [12][17]. Our programs spent a negligible time through Eilenberg-Moore, almost all the work is devoted to compute the twisted Eilenberg-Zilber data. With the exception of the fantastic paper by Szczarba [50], which unfortunately looks today rather like a blind alley, no significant work has yet been devoted to the twisted Eilenberg-Zilber theorem; it is an error[18].
- Discovering experimental facts. As explained just before, the twisted Eilenberg-Zilber theorem works today as a mysterious black box. Our programs allowed us to *experimentally* discover striking properties of this mysterious object; some of them are now proved, others remain unproved. A section is devoted to these questions in the paper. They concern the deep nature of coalgebra structures, not at all the value of some homology group.
- Theoretical computing tool. Our programs allow their users to handle the traditional objects of algebraic topology: simplicial sets (finite or not), chain complexes (of finite type or not), algebra and coalgebra structures, suspensions, wedges, pushout (topological or algebraic), and so on, like with a

[16]Jean-Pierre Serre got his Fields Medal in 1954 not for having computed a few sphere homotopy groups: forty years later, noone knows an actual application; the interest of Serre's work was in the important[(1)] *new methods* which were necessary to reach them.

[(1)]At least in algebraic topology.

[17]Let us note it was proved by Edgar Brown, the only person interested by constructive algebraic topology during the birth of algebraic topology.

[18]A referee transmitted a previous version of the present paper to another one, in particular because he knew nothing about the twisted Eilenberg-Zilber theorem; we (sincerely) congratulate him for his honesty.

pocket calculator. Our programs are also a convenient tool to investigate various properties of our usual objects. For example several complicated formulas for important operators were so easily determined.

We must also remark a fourth type of use is important. The EAT program is an excellent (mandatory?) didactical tool: most of people having so far understood our work have attended some day an EAT demonstration. The hard work in this paper is more or less to *simulate* a demo; not easy.

1.14 Conclusion.

The readers of this long introduction will have probably understood their authors are amazed a so little work has so far been done in *constructive* algebraic topology. It is true both subjects, algebraic topology and computability, are still among the most difficult ones today and navigating in both simultaneously is not so easy. But the authors witness it is fascinating. We hope the present paper will help other colleagues to enter this nice field[19].

1.15 Structure of the paper.

Section 2 roughly describes our general organization of Constructive Algebraic Topology. The main obstacle is around functional programming, and Sections 3 to 8 are entirely devoted to this question. Section 3 presents, thanks to elementary Lisp examples, what functional programming is. Section 4 uses the technique so introduced to explain how it is then possible to handle infinite objects, such as infinite simplicial sets, on a necessarily finite machine. Section 5 uses the examples of Section 4 to explain the difference between a usual "mathematical" definition and a *constructive* one. In a sense, Sections 3, 4 and 5 constitute a whole, based on elementary Lisp examples.

Sections 6 and 7 introduce the EAT program, which will be used later to illustrate with numerous examples the various *constructive* results of the paper; main theorems have a statement beginning by *"An algorithm A can be constructed. . . "*; many essential algorithms of this sort in fact *are* already constructed, and seeing them working is certainly the ideal tool for understanding. Section 6 gives a general presentation, and Section 7 focuses on loop spaces. Finally Section 8 defines a new mathematical language, around the notions of *effective* and *locally effective* object (chain complex, simplicial set, etc.). Section 9 reconsiders the vague definitions of Section 2, taking account of the precise mathematical language now available. The results of the paper are now *stated*, they must also be proved; this is much easier.

[19]This introduction would not be complete without quoting Jean-Pierre Serre declaring in official circumstances that this work contains no theorem (sic); constructive algebraic topology seems a little more difficult than Taniyama-Weil. See also Peter May's opinion at [42]. These friendly colleagues have not yet explained the reasons why Smirnov [46], Schön [40] and Justin Smith [47] spent so much work proving their non-theorems.

The main tool is the *Basic Perturbation Lemma*, a detailed proof of which is the subject of Section 10. Mixing this "lemma" with the traditional organization of spectral sequences, we obtain our various results all stated in Section 9; the classical spectral sequences, Serre and Eilenberg-Moore, are now constructive tools. Sections 11 to 21 detail the process.

Several sections from Section 10 to 21 end with a subsection entitled *EAT implementation*, using what is available in the EAT program to help understanding as far as possible; frequently, in these implementation sections, forward references must be used: the EAT program is a whole, so that it can be impossible to illustrate Section s without using what will be explained in Section $s+s'$; some redundancies cannot be avoided in such a context; another consequence is that the illustration of Section $s+s'$ can be almost entirely given in Section s.

Finally, a few appendices give some complements.

2 General organization.

We present in this section, without precise definitions from the algorithmic point of view, our general organization of constructive algebraic topology.

Every considered chain complex is a chain complex of *free* $\mathbb{Z}$-modules, not necessarily of finite type.

Definition 1 — A *reduction* is a 5-tuple $(\widehat{C}, C, f, g, h)$:

$$\begin{array}{ccc} \widehat{C} & \xrightarrow{h} & \widehat{C} \\ {\scriptstyle f}\downarrow\uparrow{\scriptstyle g} & & \\ C & & \end{array}$$

where $\widehat{C}$ and C are chain complexes, f and g are chain complex morphisms, h is a homotopy operator of degree 1; these data must satisfy the following relations:

1) $fg = 1_C$;
2) $fh = 0$;
3) $hg = 0$;
4) $hh = 0$;
5) $1_{\widehat{C}} - gf = hd + dh$.

The morphisms f and g and the homotopy operator h describe the (big) chain complex $\widehat{C}$ as a direct sum of the (small) chain complex C and an acyclic direct summand. More precisely, our reduction gives an explicit decomposition $\widehat{C} = \ker(f) \oplus \mathrm{im}(g)$; the various relations satisfied by h express that $h|\mathrm{im}(g) \equiv 0$ and $h|\ker(f) : \ker(f) \to \ker(f)$ is a homotopy contraction of $\ker(f)$. It is frequent but not mandatory the big chain complex $\widehat{C}$ is not of finite type, *even if implemented on our machine* ("implemented" in short), and on the contrary, the small chain complex C is of finite type; in such a case, the homology of C is computable, the homology of $\widehat{C}$ is not, but the reduction is after all a chain equivalence between $\widehat{C}$ and C, so that the small complex C can be considered as *describing* the homology of $\widehat{C}$.

Other authors call such an object a contraction or a retraction. We think such a terminology is not appropriate, because the claimed contraction does not concern topological objects, but only algebraic ones; in general the chain complexes C and $\widehat{C}$ do not carry a simplicial structure. So that the classical algebraic name *reduction* is the right one.

Definition 2 — A *homotopy equivalence* between two chain complexes C_1 and C_2 is a pair of reductions:

$$\begin{array}{ccc} & \hat{C} & \\ \rho_1 \swarrow & & \searrow \rho_2 \\ C_1 & & C_2 \end{array}$$

If C_1 and C_2 are *free* $\mathbb{Z}$-chain complexes, a usual chain equivalence between them can be organized in this way. Frequently the chain complexes C_1 and $\hat{C}$ are not of finite type and on the contrary the chain complex C_2 is of finite type; its homology is computable, so that C_2 can then be understood as a description of the homology of C_1. The chain complex $\hat{C}$ is only an intermediate object.

Note we already contradict our theory about what a right terminology should be. Our "homotopy equivalence" is strictly algebraic, and would be better called something like a *strong chain equivalence*, but it is a little lengthy and we prefer "homotopy equivalence".

Definition 3 — An *object with effective homology* is a 4-tuple (X, C, EC, ε) where:

1) X is an object;
2) C is the chain complex canonically associated to X;
3) EC is a chain complex of finite type (EC = effective chain complex);
4) ε is a homotopy equivalence between C and EC.

For example X could be a simplicial set (and $C = C_*X$), or a group (and $C = C_*(BX)$), or even a chain complex (and $C = X$). Usually, the homology groups of a simplicial set X are simply the homology groups of $C_*(X)$, but we will frequently meet situations where X is an *implemented* infinite simplicial set; a machine program can then *construct* the associated chain complex $C_*(X)$, but its homology groups are not computable. A simplicial set with effective homology contains also a chain complex EC of finite type having the right homology groups and a homotopy equivalence between EC and $C_*(X)$. If you are interested by the homology groups of X, you can compute the homology groups of EC. The *explicit* homotopy equivalence ε will be the main tool allowing us to transform the usual spectral sequences into actual algorithms computing homology groups of new objects.

Meta-theorem 4 — *If a "classical" exact or spectral sequence "describes" the homology groups of a new object X constructed from given objects $X_1, \ldots, X_n$, then a general algorithm can work on implemented objects with effective homology $(X_1, C_1, EC_1, \varepsilon_1), \ldots, (X_n, C_n, EC_n, \varepsilon_n)$ to construct a corresponding object with effective homology (X, C, EC, ε).*

So that if you are interested by the homology groups of X, it is sufficient to compute the homology groups of EC, that are computable. More important, if you intend to use X to construct a new object Y, an algorithm can use the version with effective homology of X to obtain such a version for Y: the process can be *iterated*, and in this way we can obtain for example the homology groups of an iterated loop space of a sufficiently connected space X, *even if* the initial space X is not a suspension. The *severe* difficulties of Carlsson and Milgram [28, p.545] are overcome. Furthermore our solution works even if X is highly infinite, but with effective homology.

Theorem 5 — *A general algorithm Ω_{EH} can work on a 1-reduced simplicial set with effective homology X_{EH} to construct a version with effective homology $(\Omega X)_{EH}$ of its loop space.*

Corollary 6 — *A general algorithm Ω^n_{EH} can work on an n-reduced simplicial set with effective homology X_{EH} to construct a version with effective homology $(\Omega^n X)_{EH}$ of its n-th loop space.*

See Definition 8 for the notion of n-reduced simplicial set; it is a strong form of n-connectedness.

Our work needs now two important steps. The only really new consists in explaining how it is possible to implement on a *finite* machine an object with effective homology, several components of which are frequently "highly" infinite from a classical point of view; as explained in the introduction, it is a matter of functional programming. The second step then consists in implementing versions *with effective homology* of the classical exact and spectral sequences; the main tool is the *basic homological perturbation lemma* and this step is quite elementary; however interesting new problems are so opened.

3 Functional programming in Lisp.

This section contains a few cultural indications about the most important theoretical machine models. The main tool in this paper is *functional programming* available on a machine equipped with a *functional* programming language, the Lisp language here. Some simple tutorial examples are given to explain what the very nature of functional programming is.

3.1 Theoretical machines.

There are numerous theoretical machine models. Historically, the first machine model is the *recursive machine*, which was designed mainly by Hilbert, Ackermann and Gödel. For the recursive machine, a *program* is a (general) *recursive function* $f : \mathbb{N}^d \to \mathbb{N}$ or more precisely its definition with respect to the elementary recursive function constructors; see for example [51, chap. 2 and 3]. The domain of a recursive function $f : \mathbb{N}^d \to \mathbb{N}$ is not necessarily the whole set $\mathbb{N}^d$; in other words, a program properly works only for some data, otherwise the program does not terminate. There does not exist a general algorithm able to guess whether an element of $\mathbb{N}^d$ is in the domain of f (Post's theorem). There exist a recursive function $f : \mathbb{N} \to \mathbb{N}$ and an element $n \in \mathbb{N}$ satisfying the following properties:

1) The "program" computing $f(n)$ does not terminate;

2) There does not exist a proof of 1).

This is essentially incompleteness Gödel's theorem. Thanks to Novikov's theorem linking Gödel's theorem to groups of finite presentation [36], a consequence of these results is the following: there does not exist a general algorithm able to decide whether a finite simplicial complex is simply connected; furthermore there exists a finite simplicial complex satisfying the following properties:

1) it is not simply connected;

2) there does not exist a proof of 1).

There exists an explicit *universal* recursive function $u : \mathbb{N}^2 \to \mathbb{N}$; in other words for every recursive function $f : \mathbb{N}^d \to \mathbb{N}$, an explicit integer n_f can be given such that $f(\overline{n}) = u(n_f, \chi(\overline{n}))$ for every $\overline{n} \in \mathbb{N}^d$, the function $\chi : \mathbb{N}^d \to \mathbb{N}$ being a simple explicit coding $\chi : \mathbb{N}^d \to \mathbb{N}$. Any computer is nothing but such a universal function.

We shall not use in this paper the framework of recursive functions. The most classical machine model is the Turing machine [2, chap. 1], which is close to ordinary computer hardware; the classical computability theorems quoted above for recursive functions have a natural translation for the Turing machine. For example an ordinary computer is a *universal* Turing machine. We shall not use the Turing machine model either.

Another machine model is the λ-calculus. The classical reference is [16] but a much more convenient reference to quickly understand the main notions of λ-calculus is [37]. Again the classical computability results have a translation in the λ-calculus framework; in fact all known machine models are equivalent and Church's thesis [51, chap. 7] consists in believing that it will not be possible to design a *stricly* more powerful machine. The λ-calculus machine is at the origin of an important programming language, the Lisp language, which is the *new* essential tool used in this paper to solve theoretical and practical computability problems in algebraic topology.

3.2 The Lisp machine.

It is not reasonable to give here a *mathematical* definition of what is a Lisp machine, that is, a computer equipped with the Lisp programming language; the reference [21] is such a *mathematical* definition (900p.); see also [49]. The only part of [21] which is essential for us and relatively new with respect to standard programming culture is the section *About Scope and Extent* [21, pp 38-47]; this section explains the notion of *lexical closure*, important for us; the extraordinary **Y**-combinator of λ-calculus is also described there: readers thinking that computer science is elementary are advised to take a glance at this object, but we do not use this combinator.

The Nanolisp machine [42] is in a sense the smallest Lisp machine theoretically equivalent to the classical Lisp machine; but Nanolisp does not contain the very numerous technical functions that are helpful only for concrete efficient work. In particular the Nanolisp machine uses the notion of *lexical closure*; the guide about closures, available at [42], is probably the most convenient text to understand this notion and how it is possible to implement it; but reading of this text should not be necessary to understand the present paper.

The lexical closure technique in Lisp programming is a powerful tool allowing the user to easily write functions which *at run-time* will create new functions which *later at run-time* will create other new functions and so on; the problem of identifier scope is then difficult and is very elegantly solved in Lisp, thanks to the notion of lexical closure. Such a tool is not available in usual programming languages such as Pascal, C, Ada, etc.; but two (reasonably efficient) arbitrary machines are equivalent, so that it is possible for an experienced programmer to implement this technique with the help of his favourite language; he should then understand the tricky underlying list technique which is explained for example in [42] and [3, sect. 3.10]. Instead of explaining the essential ingredients of this technique, it will be sufficient here to explain how it can be practically used. The didactical examples given in this section are so natural that the reader will probably meet some difficulty to believe there is here something *new* and essential; he will be convinced when he will succeed in creating a C or Pascal environment allowing him to write equivalent programs.

3.3 First steps in Lisp.

A Lisp expression is a parenthezed list such that `(+ 2 2)` where the first element is an operator (prefix convention). In this text, an example is displayed as follows:

```
> (+ 2 2)  ==>
4
>
```

The '`>`' symbol is the *Lisp prompt*, displayed on the terminal screen by Lisp to explain Lisp is awaiting for the following expression to be *evaluated.* The Lisp prompt depends on the used Lisp implementation; all the examples of this paper are given using the freeware Allegro[20] Common Lisp [22], but translation into another Common Lisp is always obvious. The sign '==>' ending the first line, in fact *not visible* on the screen, means the user typed the Return key to ask Lisp about the result of evaluation, which is here the integer 4. Then Lisp displays the waiting prompt for the following expression and so on.

The assignment operator is the predefined `setf` Lisp operator:

```
> (setf x 3)  ==>
3
> (setf y 4)  ==>
4
> (+ x y)  ==>
7
```

3.4 First steps in functional programming.

If a Lisp user[21] wants to construct a function which adds 123 to any integer, he can work in this way:

```
> (setf add123
    (f-of (n) (+ n 123)))  ==>
#<function 1 #xD9B260>
> (? add123 23)  ==>
146
> (? add123 100)  ==>
223
```

The external form of a result such as `#<function 1 #xD9B260>` depends on the used Lisp implementation; in this case the component '1' of the result recalls the number of variables of the constructed function and the number following the characters '`#x`' is a machine address which will not be displayed in this paper. The first `setf` statement can be read as follows: assign to the symbol `add123` the function able to work with `n` and returning the result of `(+ n 123)`. In particular the special symbol `f-of`[22] means "*build the function able to work on... returning...*"; such a meaning is directly inherited from λ-calculus. Then this function can be

[20]This version of Common Lisp is very convenient if you have a Windows-95 platform: a complete hypertext online documentation is available and specialized editors are provided; the only difference with the commercial version is a limited memory, quite sufficient for the examples of this paper. The commercial version provides also a convenient tutorial book. Non-payed advertisement.

[21]The more or less complicated Lisp examples of this paper have been stored in a public file `examples.lisp` which can be obtained at [42]; this Lisp file contains all the Lisp instructions:
`(setf ... ...)`
showed here; so that the interested user can (Lisp-)load this file and then verify the objects so constructed and assigned to symbols properly work.

[22]In fact Common Lisp normally needs here `(setf add123 #'(lambda ...))`; the strange special combination `#'(lambda ...)` asks Lisp to construct the appropriate *lexical closure*; but it is possible, using the following *macro-definition*:

questionned by the '?' operator, giving the symbol locating the function and also the necessary object(s) on which the function must work[23].

Let us now suppose we want to use a general function able to construct any additionner function:

```
> (setf make-adder
    (f-of (constant)
      (f-of (n)
        (+ n constant))))  ==>
#<function 1 ...>
> (setf add5 (? make-adder 5))  ==>
#<closure 1 ...>
> (? add5 6)  ==>
11
> (setf add7 (? make-adder 7))  ==>
#<closure 1 ...>
> (? add7 8)  ==>
15
> (? add5 8)  ==>
13
```

You see the symbol `CONSTANT` has now two different values which can be asked at any time, according to you reach it through `add5` or `add7`; in fact an arbitrary number of values for this symbol could be installed and used in such a Lisp environment; the resulting problem of identifier scope is solved thanks to the lexical closures: note the values of `add7` and `add5` are *closures*, not functions.

A general composition operator can be written to compose any two functions $\mathbb{N} \to \mathbb{N}$:

```
> (setf compose
    (f-of (g f)
      (f-of (n)
        (? g (? f n)))))  ==>
#<function 2 ...>
> (setf add12 (? compose add5 add7))  ==>
#<closure 1 ...>
> (? add12 23)  ==>
35
```

You can read: assign to the symbol `compose` the function working on `g` and `f`, returning the function working on `n` which returns $g(f(n))$; then assign to the symbol `add12` the result of the function `compose` working on `add5` and `add7`. Like in mathematics, in this functional framework, a function is an ordinary object

---

`(defmacro f-of (&rest rest) '(function (lambda ,@rest)))`
to exempt the user from writing these signs `#'`; such a macro-definition is provided at the beginning of the file `examples.lisp`.

[23]In fact the predefined Lisp function doing this work is the `funcall` function, but it is easy to make `?` and `funcall` equivalent:
`(setf (symbol-function '?)  #'funcall)`
and this is done at the beginning of the file `examples.lisp`.

which can work on functions to create other functions which will create again other functions and so on.

4 Functional programming and infinite simplicial sets.

We explain in this section how the functional programming techniques can be used to code "highly" infinite simplicial sets on a (necessarily finite) machine. This is illustrated in this section by a Lisp implementation of the infinite standard simplex Δ^∞. In the same way we will be able later to install on our machine the Kan model GX of a simplicial set X assumed present in our machine memory. If X is a (finite or not) simplicial set, the Kan model GX for the loop-space ΩX is a highly infinite simplicial set where the set GX_n of n-simplices is a free (non-commutative) group generated by some $(n+1)$-simplices of X; if X is itself a Kan model of a loop-space, the set of *generators* in GX_n is itself highly infinite, and so on; but it will be proved later that the homology groups of these iterated loop spaces can be sometimes computed, using these necessarily finite implementations of Kan models and some further data.

Defining a simplicial set X consists in giving for each dimension n a simplex set X_n and the various face and degenaracy operators [23, 32, 30]. Let us recall X is a family $\{X_n, \partial_i^n, \eta_i^n\}$ where X_n is defined for $n \in \mathbb{N}$ and is the set of n-simplices, the operator $\partial_i^n : X_n \to X_{n-1}$ (face operator) is defined for $n \geq 1$ and $0 \leq i \leq n$ and the operator $\eta_i^n : X_n \to X_{n+1}$ (degeneracy operator) is defined for $n \geq 0$ and $0 \leq i \leq n$. Furthermore, these operators must satisfy a few commuting relations, see [23, 32, 30]. We will give in this section some didactical examples of simplicial sets, Lisp-implemented.

4.1 The components of a machine simplicial set.

Because we are working on a Lisp machine, any simplex will be a *Lisp object*. If X is a simplicial set, the set X_n of the n-simplices of X can be infinite and the only solution to code such a possibly infinite set is to consider it as a *type*, that is, a function able to work on any (Lisp) object and answering whether the object is an element of the type or not. For example, let us suppose we intend to install the infinite simplex Δ^∞. Then a simplex is an incresing list (finite sequence) of natural numbers. To prepare the construction of Δ^∞, we define the appropriate type:

```
> (setf delta-infinity-type
    (f-of (object)
      (and (listp object)
           (every #'integerp object)
           (apply #'<= (append '(0) object)))))  ==>
#<function 1 ...>
> (? delta-infinity-type 'symbol)
```

```
NIL
> (? delta-infinity-type '(0 2 3 4 symbol))
NIL
> (? delta-infinity-type '(-4 0 2 3 4))
NIL
> (? delta-infinity-type '(0 2 3 4 3))
NIL
> (? delta-infinity-type '(0 2 3 4 4))
T
```

The function `delta-infinity-type` successively verifies whether the object is a list, then whether each member is an integer, then whether the list is an increasing sequence of null or positive integers. The logical "true" object in Lisp is the symbol "`T`" and the logical "false" object is the symbol "`NIL`".

It is necessary to have a function able to determine the simplex dimension:

```
> (setf delta-infinity-dmn
    (f-of (smp) (1- (length smp))))  ==>
#<function 1 ...>
> (delta-infinity-dmn '(0 2 4 6 8))  ==>
4
```

The predefined Lisp function `1-` subtracts 1 from its argument, and the `length` function returns the length of its argument list. The following point is important about the general coherence of our functions: the user is supposed to call a function such as `delta-infinity-dmn` for an object *which actually is* a simplex of Δ^∞, this is the reason why the argument is named `smp` (for simplex). A safer function would be:

```
> (setf safe-delta-infinity-dmn
    (f-of (smp)
      (unless (? delta-infinity-type smp)
        (error "The argument of safe-delta-infinity-dmn~@
                is not a simplex of delta-infinity."))
      (1- (length smp))))  ==>
#<function 1 ...>
> (? safe-delta-infinity-dmn '(0 2 4 6 8))  ==>
4
> (? safe-delta-infinity-dmn '(0 4 2 6 8))  ==>
;; Error: The argument of safe-delta-infinity-dmn
;;   is not a simplex of delta-infinity.
```

But these safety strategies are not considered in this paper: the programmer is assumed secure (!?). The type functions are only used to define the underlying simplex set through its characteristic function, able to work for any machine object.

It will be frequently necessary to decide whether two simplices are equal or not, so that an *equality* function is needed for each machine simplicial set; in particular such an equality function will allow us to easily construct *quotient* simplicial sets, by modifying only the equality function. For Δ^∞, we have to compare two integer lists and the predefined function `equal` does this work:

```
> (setf delta-infinity-equal #'equal)
#<function 2 ...>
> (? delta-infinity-equal '(1 2 2) '(1 2 3))
NIL
> (? delta-infinity-equal '(1 2 3) '(1 2 3))
T
```

The simplicial set $\overline{\Delta}^\infty$, the quotient of Δ^∞ by the relation which identifies any two vertices is important in Kan theory. It can be coded as Δ^∞ with a unique difference for the equality function:

```
> (setf delta-infinity-bar-equal
    (f-of (smp1 smp2)
      (or (equal smp1 smp2)
          (= 1 (length smp1) (length smp2)))))
#<function 2 ...>
> (? delta-infinity-bar-equal '(1 2 2) '(1 2 3))
NIL
> (? delta-infinity-bar-equal '(2) '(3))
T
> (? delta-infinity-equal '(2) '(3))
NIL
```

The equality relation between $\overline{\Delta}^\infty$-simplices is only a little weaker than for Δ^∞: if two simplices have dimension 0 (i.e. their list length is 1), they are certainly equal.

The following ingredient needed to define a simplicial set is the face operator. The ∂_i^n operator of Δ^∞ consists in removing the i-th element of the simplex $(i_0 \ldots i_n)$. Another function does this work:

```
> (setf delta-infinity-del
    (f-of (i smp)
      (append (subseq smp 0 i) (subseq smp (1+ i)))))
#<function 2 ...>
> (? delta-infinity-del 2 '(3 4 5 6 7))
(3 4 6 7)
```

The first call to the `subseq` predefined Lisp function extracts the subsequence between the indices 0 (inclusive) and i (exclusive), and the second call extracts the subsequence starting at the index $i+1$ up to the end because the second argument is omitted; finally these two subsequences are appended. In other words the i-th element is removed, and the needed i-face operator is so defined; it is not necessary to give the dimension integer, because in this organization it is implicitly included in the simplex. In the same way, the degeneracy operator, consisting in repeating the i-th element of the simplex, can be defined as follows:

```
> (setf delta-infinity-deg
    (f-of (i smp)
      (append (subseq smp 0 (1+ i)) (subseq smp i))))
#<function 2 ...>
> (? delta-infinity-deg 2 '(0 1 2 3 4))
(0 1 2 2 3 4)
```

4.2 A machine simplicial set is an ss instance.

The various functions already defined *are* a complete definition of Δ^∞. The Lisp language allows the user to put together the components of this definition thanks to an appropriate `structure`, the type of which is defined by the predefined operator `defstruct` (that is, "define structure"). Let us begin as in a Lisp tutorial with a very simple example:

```
> (defstruct point x y z)  ==>
POINT
> (setf a1 (make-point :x -2 :y 3 :z -4))
#S(POINT X -2 Y 3 Z -4)
> (setf a2 (make-point :x 1 :y 2 :z 2))
#S(POINT X 1 Y 2 Z 2)
> (setf add-points
    (f-of (a b)
      (make-point :x (+ (point-x a) (point-x b))
                  :y (+ (point-y a) (point-y b))
                  :z (+ (point-z a) (point-z b)))))
#<function 2 ...>
> (setf a3 (? add-points a1 a2))
#S(POINT X -1 Y 5 Z -2)
```

The first instruction `(defstruct ...)` tells Lisp a new *structure* type is defined, namely the `point` structure type; an *instance* of this type, or a `point` instance, has three *slots* of name `x`, `y` and `z`. Automatically constructed Lisp functions (constructed when `defstruct` works) allow the user to construct instances, to copy them, to read a particular slot of some instance and also to modify (update) some particular slot.

The automatically defined `make-point` function can then be used to construct a `point` instance; we do not want to explain the strange (necessary) colon which is in front of `x` in the `(make-point :x ...)` statement: it is a question of *keyword parameter* (see [49, 21]). Then it is possible to define functions working on and/or making points, for example a function adding two points and returning their sum; the slots of an instance can be reached (read) through functions also automatically constructed by Lisp, the name of which is made of the name of the structure and the name of the slot; for example to obtain the `x`-slot of a `point` instance, you may use the automatically defined function `point-x`.

So that it is possible to collect the functions defining the simplicial structure of Δ^∞ or $\overline{\Delta}^\infty$ in instances of an appropriate structure type:

```
> (defstruct ss type dmn equal del deg)  ==>
SS
> (setf delta-infinity
    (make-ss :type  delta-infinity-type
             :dmn   delta-infinity-dmn
             :equal delta-infinity-equal
             :del   delta-infinity-del
             :deg   delta-infinity-deg))  ==>
```

```
#S(SS TYPE #<function 1 #xD90E9C> DMN #<function 1 ...> ...)
> (setf delta-infinity-bar
    (make-ss :type  delta-infinity-type
             :dmn   delta-infinity-dmn
             :equal delta-infinity-bar-equal
             :del   delta-infinity-del
             :deg   delta-infinity-deg))  ==>
#S(SS TYPE ...)
```

We choose the name `ss` (simplicial set) for the new structure type, and an instance of this type has five slots of name `type`, `dmn`, `equal`, `del` and `deg`. Each slot of an `ss` instance will be a function able to do some work with respect to the underlying simplicial set. Once the simplicial set is so installed in the Lisp machine, it can be used through the functional slots:

```
> (? (ss-equal delta-infinity) '(0) '(3))  ==>
NIL
> (? (ss-equal delta-infinity-bar) '(0) '(3))  ==>
T
> (? (ss-del delta-infinity) 2 '(3 4 5 6 7))  ==>
(3 4 6 7)
```

4.3 Constructing a new simplicial set from others.

The tutorial example of points in $\mathbb{R}^3$ shows how it is possible to define functions working on given points, making another point. In the same way it is possible to define functions working on simplicial sets and making other simplicial sets. For example let us consider the *product* functor for simplicial sets. A simplex of the product simplicial set is a pair of simplices, which will be coded as a list with two elements. A few elementary predefined Lisp functions allow the user to construct and handle such lists:

```
> (setf l (list 3 4))  ==>
(3 4)
> (first l)  ==>
3
> (second l)  ==>
4
```

The `list` function constructs a list with an arbitrary number of elements, and the functions `first` and `second` extract the first and second components. Let us now prepare the product construction of two arbitrary simplicial sets:

```
> (setf type-product
    (f-of (ss1 ss2)
      (f-of (obj)
        (and (listp obj)
             (= 2 (length obj))
             (? (ss-type ss1) (first obj))
```

```
             (? (ss-type ss2) (second obj))
             (= (? (ss-dmn ss1) (first obj))
                (? (ss-dmn ss2) (second obj)))))))  ==>
#<function 2 ...>
> (setf dmn-product
    (f-of (ss1 ss2)
      (f-of (smp)
        (? (ss-dmn ss1) (first smp)))))  ==>
#<function 2 ...>
> (setf equal-product
    (f-of (ss1 ss2)
      (f-of (smp1 smp2)
        (and (? (ss-equal ss1) (first smp1) (first smp2))
             (? (ss-equal ss2) (second smp1) (second smp2))))))  ==>
#<function 2 ...>
> (setf del-product
    (f-of (ss1 ss2)
      (f-of (i smp)
        (list (? (ss-del ss1) i (first smp))
              (? (ss-del ss2) i (second smp))))))  ==>
#<function 2 ...>
> (setf deg-product
    (f-of (ss1 ss2)
      (f-of (i smp)
        (list (? (ss-deg ss1) i (first smp))
              (? (ss-deg ss2) i (second smp))))))  ==>
#<function 2 ...>
```

For example, given two simplicial sets `ss1` and `ss2`, the function `type-product` *constructs* a new type function, verifying whether the argument transmitted is a list of length two, whether the first (second) component is a simplex of `ss1` (`ss2`), and whether the dimensions of both components are equal. The other function constructors are analogous. As in the very simple examples of section 3 it is a matter of producing new functions depending on other functions already constructed and this is the reason why *functional programming* is definitively necessary.

We can see this mechanism working properly as follows:

```
> (setf d-db-dmn
    (? dmn-product delta-infinity delta-infinity-bar))  ==>
#<closure 1 ...>
> (? d-db-dmn '((1 3 5 7) (2 4 6 8)))  ==>
3
> (setf d-db-equal
    (? equal-product delta-infinity delta-infinity-bar))  ==>
#<closure 2 ...>
> (? d-db-equal '((1 2 2) (3 4 4)) '((1 2 2) (3 4 5)))  ==>
NIL
> (? d-db-equal '((4) (2)) '((5) (2)))  ==>
NIL
> (? d-db-equal '((2) (4)) '((2) (5)))  ==>
T
```

We can construct in the same way the other components of the simplicial product $\Delta^\infty \times \overline{\Delta}^\infty$, and collect them in an `ss` instance, but it is much better to write a function doing this work by itself:

```
> (setf ss-product
    (f-of (ss1 ss2)
      (make-ss :type  (? type-product ss1 ss2)
               :dmn   (? dmn-product ss1 ss2)
               :equal (? equal-product ss1 ss2)
               :del   (? del-product ss1 ss2)
               :deg   (? deg-product ss1 ss2))))
#<function 2 ...>
> (setf d-db (? ss-product delta-infinity delta-infinity-bar))  ==>
#S(SS TYPE #<closure 1 ...> ...)
> (? (ss-type d-db) '((1 2 3) (5 7)))  ==>
NIL
```

The last object which is proposed as a possible simplex of $\Delta^\infty \times \overline{\Delta}^\infty$ obtains a negative answer: the dimensions of the components are different. And you can as easily continue:

```
> (setf d-db-d (? ss-product d-db delta-infinity))  ==>
#S(SS TYPE #<closure 1 ...> ...)
> (? (ss-type d-db-d) '(((1 2) (3 4)) (5 6)))  ==>
T
> (? (ss-type d-db-d) '((1 2) ((3 4) (5 6))))  ==>
NIL
> (? (ss-type d-db-d) '((1 2) (3 4) (5 6)))  ==>
NIL
```

In this necessarily very precise framework, the product operator for simplicial sets is not associative! This is a frequent source of unavoidable technical difficulties; they can be overcome thanks to tricky technicalities, but this is not the subject of the present paper.

5 A mathematical definition is not necessarily a constructive definition.

5.1 A negative result.

It seems so easy in the previous section to construct machine objects that are equivalent to possibly infinite simplicial sets that a careless observer could believe the computability problem in algebraic topology is solved. This is erroneous, even from a *theoretical* point of view. Let us call a machine object such as `delta-infinity` a *locally effective simplicial set*; this terminology will be justified at the end of this section.

Theorem 7 — *There does not exist any general algorithm able to work on an arbitrary locally effective simplicial set, determining whether the underlying simplicial set is empty or not.*

The precision *"general"* for the non-existing algorithm is in principle redundant but useful to avoid a frequent misunderstanding. In fact it is possible to define relatively large classes of locally effective simplicial sets such that the emptyness problem on the contrary has an algorithmic solution, but such a framework is so artificial that this is without any actual interest.

PROOF. This theorem is in fact a simple corollary of Post's theorem about the halting problem: given an arbitrary program P working in discrete time (some state at time 0, the following state at time 1 and so on, until a possible terminal state, or otherwise always continuing working), you can associate to P the discrete simplicial set ${}^{ss}P$ where a Lisp object ω is a simplex if and only if ω is a pair of integers (d,t) where $d \in \mathbb{N}$ is its dimension and t is an integer such that the program P is halted at time t; in other words ${}^{ss}P_d = \{d\} \times [h, +\infty[$ if the program P halts at time h, else ${}^{ss}P = \emptyset$. The `type` function for ${}^{ss}P$ working on the pair (d,t) must simulate the work of P between times 0 and t to decide whether P is halted at time t; it is a classical "exercise" about a programming language to write a program able to simulate the work of the underlying machine when another arbitrary program is given in the same language (or another one). The faces and degeneracy operators of ${}^{ss}P$ are the simple maps $(d,t) \mapsto (d-1,t)$ or $(d+1,t)$, so that the non-degenerate simplices have dimension 0. In this way it is possible to associate to the program P the simplicial set ${}^{ss}P$; now determining whether ${}^{ss}P$ is empty or not is equivalent to the halting problem for P: will the program P terminate? But the latter problem does not have any algorithmic solution (Post's theorem). ∎

No general program can determine whether a (locally effective) machine simplicial set is empty or not; therefore no general program is able to compute the homology groups of a machine simplicial set, otherwise the emptyness problem would have an algorithmic solution. In the same way no general program can determine whether the homology of a machine simplicial set is of finite type, or null above some dimension, and so on. A positive solution for a general computability problem needs more information in the data. The aim of this paper consists in proving that a few more informations on the contrary allow us to give a positive solution for the main computability problems, mainly in situations where the underlying machine simplicial set is highly infinite.

5.2 A further slot for the ss instances.

Let us redefine the `ss` structure type of the previous section, by addition of a new slot named `basis`:

```
> (defstruct ss type dmn equal basis del deg)  ==>
SS
```

The machine object modelizing the infinite simplex Δ^∞ must be also redefined:

```
> (setf delta-infinity
```

```
    (make-ss :type  delta-infinity-type
             :dmn   delta-infinity-dmn
             :equal delta-infinity-equal
             :basis :locally-effective
             :del   delta-infinity-del
             :deg   delta-infinity-deg))  ==>
#S(SS ...)
```

The keyword (that is, a symbol beginning with a colon) `:locally-effective` given as the value of the `basis` slot means this simplicial set is only *locally effective*, so that the information usually asked for in this slot is not available. On the contrary, if the simplicial set is *effective*, then this slot must contain a function giving for each dimension the (necessarily finite) list of simplices of this dimension. For example let us install in this way the finite simplex Δ^5; firstly:

```
> (setf delta-5-type
    (f-of (object)
      (and (listp object)
           (every #'integerp object)
           (apply #'<= (append '(0) object '(5))))))  ==>
#<function 1 ...>
> (? delta-infinity-type '(2 3 6))  ==>
T
> (? delta-5-type '(2 3 6))  ==>
NIL
> (? delta-5-type '(2 3 3 5))  ==>
T
```

will allow us to define the needed new simplex type: now the vertices must be in $\{0, 1, \ldots, 5\}$. The slots `dmn`, `equal`, `del` and `deg` for Δ^5 can be simply copied from Δ^∞. For the `basis` slot, we must write a function giving all the simplices of dimension `dmn` in Δ^5:

```
> (setf delta-5-basis
    (f-of (dmn)
      (let ((result '()))
        (if (zerop dmn)
            (dotimes (i 6)
              (push (list i) result))
            (dolist (smp-1 (? delta-5-basis (1- dmn)))
              (dotimes (i (1+ (first smp-1)))
                (push (cons i smp-1) result))))
           result)))  ==>
#<function 1 ...>
> (? delta-5-basis 0)  ==>
((5) (4) (3) (2) (1) (0))
> (? delta-5-basis 1)  ==>
((0 0) (1 1) (0 1) (2 2) (1 2) (0 2) (3 3) (2 3) (1 3) (0 3) (4 4)
 (3 4) (2 4) (1 4) (0 4) (5 5) (4 5) (3 5) (2 5) (1 5) (0 5))
```

We are now ready to construct the *effective* simplicial set `delta-5` $= \Delta^5$:

```
> (setf delta-5
    (make-ss :type  delta-5-type
             :dmn   delta-infinity-dmn
             :equal delta-infinity-equal
             :basis delta-5-basis
             :del   delta-infinity-del
             :deg   delta-infinity-deg))  ==>
#S(SS TYPE ...)
> (? (ss-type delta-infinity) '(2 3 6))  ==>
T
> (? (ss-type delta-5) '(2 3 6))  ==>
NIL
> (? (ss-type delta-5) '(2 3 3 5))  ==>
T
```

Obvious similarities between Δ^∞ and Δ^5 have been used in the implementation of Δ^5, essentially because Δ^5 is a simplicial subset of Δ^∞. Compare the previous implementation of Δ^5 with the following:

```
> (setf locally-effective-delta-5
    (make-ss :type  delta-5-type
             :dmn   delta-infinity-dmn
             :equal delta-infinity-equal
             :basis :locally-effective
             :del   delta-infinity-del
             :deg   delta-infinity-deg))
#S(SS ...)
```

The only difference is in the `basis` slot: here this slot does not contain a functional object, in principle computing a simplex set; in `locally-effective-delta-5`, this slot contains only the keyword `:locally-effective`, so that it is no longer possible to get basis information:

```
>(? (ss-basis locally-effective-delta-5) 4)
;; Error: The function :LOCALLY-EFFECTIVE is undefined.
```

Lisp detects a coherence error: the keyword *is not* a functional object. In this new version of Δ^5, we are not able to determine the simplex list in some dimension: the `basis` slot is essentially missing. An observer cleverly studying this problem could object a careful examination of the slots `type`, `dmn` and `equal` in fact allows in this case to determine such a basis; this observer is right: in a sense the `basis` slot is redundant, but such a redundancy is valid only, in some cases, for *clever* observers, not for a machine. Precisely Theorem 7 states that no (general) *algorithm* is able to deduce from the slots `type`, `dmn` and `equal` a `basis` information, even if some finiteness property is given.

5.3 Effective and locally effective objects.

Still more precisely, from a *mathematical* point of view, the `basis` slot is always redundant: the slots `type`, `dmn` and `equal`, if they are coherent, *mathematically* define the simplex set in each dimension; the key point is the following: in general, this definition *is not constructive*. The definition of the simplicial set becomes constructive when the `basis` slot is added. Such a complementary information is possible only if the underlying simplicial set is finite in each dimension. Otherwise the only remaining possibility is to artificially fill in the `basis` slot with the keyword `:locally-effective`, a message explaining no basis information is in fact available.

For example the locally effective implementation of Δ^∞ is a complete *mathematical* definition, but an *incomplete constructive one*. Our work in the rest of this paper will consist in defining a few supplementary informations to be added to such a locally effective implementation of an infinite simplicial set, allowing the user to *effectively* determine the homology groups of this simplicial set: the homology groups will become reachable; but the underlying simplicial set will definitively remain locally effective. The main infinite simplicial sets we have in mind are Kan models of iterated loop spaces; in this way we do obtain a simple and elegant solution of the old *Adam's problem*: how to iterate the Cobar construction; compare [40, 46, 47].

We find convenient this terminology: "*effectiveness*" vs "*local effectiveness*". An *effective* object is essentially entirely known, at least up to a given finite dimension (or degree): for a simplicial set, the *complete* set of simplices in some dimension can be enumerated, the faces of these simplices can also be computed, giving the (necessarily finite) boundary matrix for this dimension, and the same for the following dimension; an elementary program can then determine the corresponding homology group. But this is possible only for finite simplicial sets (in each dimension). Infinite simplicial sets can also be *locally* installed; this means some functions are provided so that if a user is interested in some face of *some* (any!) simplex, it can be computed: *local* information is available. On the contrary, global information is not, and it is even not possible in general to determine whether the underlying simplicial set is void or not. We shall explain in the rest of this paper that carefully combining effective objects and locally effective objects allows us to easily solve the general computability problem for simply connected algebraic topology; the method is so powerful that concrete programs can be written and used to obtain new significant results.

For simplicial sets, a terminology like "simplex-wise effective simplicial set" (effective for *some* — in fact for any! — simplex of the underlying simplicial set, but not globally effective) would be more precise; but the same situation occurs in relatively different frameworks so that we feel more convenient a "generic" terminology. In the other sections we shall mainly handle locally effective simplicial sets, locally effective chain complexes ("generator-wise" effective), and effective chain complexes. An effective chain complex is a free $\mathbb{Z}$-chain complex, of finite type in each dimension. It is elementary to compute its homology groups. A locally effective chain complex does not satisfy any finiteness condition, but if *some* (any!) generator is given, a functional slot can compute the boundary of this generator; no global information is available so that it is not possible in general to determine whether some chain group is null or not.

The chain complex canonically associated to a locally effective simplicial set is a locally effective one, but it is sometimes possible to have a chain equivalence between this locally effective chain complex with another *effective* one, so that the homology groups of the last one are isomorphic to the homology groups of the locally effective simplicial set. The 4-tuples (X, C_*X, EC_*, h) where X is a locally effective simplicial set, C_*X is the canonically associated free $\mathbb{Z}$-chain complex, EC_* is some *effective* chain complex and finally h is a chain equivalence between C_*X and EC_* will be our main ingredients.

6 The EAT program.

The Lisp example programs given in the previous sections, to illustrate the notions we are interested by, are very weak; a real use of such programs would quickly become quite awkward. But Lisp (precisely Common Lisp) is powerful and allows its user to design much more convenient and efficient programs. The subject of this paper is not to study Lisp technique; however, it will be easier for the other sections to be helped by the *actual* program EAT (Effective Algebraic Topology) freely distributed by the authors at [39]. EAT is a prototype program, a first try to implement the particular results of this paper devoted to homology of iterated loop spaces. In particular it contains general functions handling effective and locally effective chain complexes, and also effective and locally effective simplicial sets. The examples constructed using EAT are specially convenient to explain the very nature of the present work and this section gives the necessary introduction. Many technicalities are not detailed here. The initialization work undertaken by the EAT program is relatively complex (see [42]) and is not considered either; we assume the EAT Lisp environment is installed.

The reader will probably be surprised to see so many technical (and quite elementary) questions examined in this section, in a paper the aim of which is strictly mathematical: how to compute homology groups and homotopy groups known (thanks to Jean-Pierre Serre) of finite type; in particular how to iterate Adam's cobar construction, the last problem being (thanks to Jean-Pierre Serre) the essential one. Experience cruelly shows it is extremely hard to understand our results without a machine demonstration, so that we must in this paper give information as close as possible to what can be shown in a machine demo. Without the keyboard, the program and the monitor screen, it is a little lengthy, but conversely the level of detail used here should make relatively easy the reading of this section.

6.1 EAT and the simplicial sets.

The first step consists in considering the previous examples (Δ^∞, $\overline{\Delta}^\infty$, Δ^5, etc.) and to reconstruct them in the EAT framework. The EAT function `build-ss` (build simplicial set) allows the user to construct a simplicial set like Δ^∞:

```
> (setf delta-infinity
    (build-ss
      :bsp '(0)
      :eqs #'equal
      :sbs :locally-effective
      :gdl #'(lambda (i dmn gsm)
               (asm empty-list
                    (append (subseq gsm 0 i) (subseq gsm (1+ i)))))
      :org '(delta-infinity)))  ==>
[SS-4]
```

The EAT documentation explains you construct in this way a simplicial set:

1. The base point (`bsp`) is the `(0)` object, to be understood as the simplex spanned by the 0-vertex, therefore a 0-simplex;
2. Comparing two *non-degenerate* simplices will use the predefined Lisp function `equal`;
3. The `sbs` (simplicial basis) slot is the keyword `:locally-effective` so that it will not be possible to get any global information;
4. The `gdl` (geometric ∂) slot explains how to compute a face of a *non-degenerate* simplex; three arguments are needed, the face-index (`i`), the dimension (`dmn`) of the simplex and the simplex itself (`gsm`)[24];
5. Finally the `org` slot contains a small (text) information about the origin of the simplicial set.

The output '`[SS-4]`' means the result is the simplicial set #4 (to be considered as a number plate); no detail is given in the output about the internal structure of this object. Each time the function `build-ss` is used, EAT gives to the new simplicial set the first free number.

A simplex can be degenerate or not, and many simplex calculations are in fact only degeneracy calculations. To take advantage of this situation, the EAT implementation of simplices uses a relatively sophisticated technique. The user firstly decides a coding for the *non-degenerate* simplices frequently located in the EAT program through the symbol `gsm` ("geometric" simplex); for example for Δ^∞, a non-degenerate simplex is a strictly increasing integer list, to be understood as the simplex spanning the corresponding vertices. Then an *arbitrary* simplex is

[24]Note we do not use anymore the "`(f-of ...)`" style; the perfectly equivalent standard Lisp form `#'(lambda ...)` is now preferred

an `asm` object, a structure with two slots, which can be constructed by the `asm` function, needing two arguments; the first one is a strictly decreasing integer list representing a composite degeneracy operator so that the empty-list means "no degeneracy at all"; the second argument is the coding for the (unique) underlying non-degenerate simplex. For example look at the `gdl` slot of Δ^∞: a face of a non degenerate simplex of $\Delta^\infty$is non degenerate too, so that the `asm` object constructed by the `gdl` slot has an empty list as multiple degeneracy operator, but this face *must* be implemented as an `asm` object.

It is important here not to be misled by the terminology similarity between the '`asm`' and '`gsm`' notions: an `asm` object is an instance of the `asm` structure type with two fields, called '`dop`' (degeneracy operator) and `gsm` (geometric simplex). On the contrary, a `gsm` object can be any Lisp object, according to the decision of the user about the coding of the non-degenerate simplices of its simplicial set.

In this way, a (really) degenerate simplex has a unique coding as an `asm` object; for example $\eta_2\eta_0\sigma_{135}$, the 4-simplex of Δ^∞ which would be coded as `(1 1 3 3 5)` in the previous section, obtained by degenerating two times the non-degenerate 2-simplex σ_{135} (spanning the vertices 1, 3 and 5) will be here constructed and displayed as follows:

```
> (asm '(2 0) '(1 3 5))  ==>
<ASM 2-0 (1 3 5)>
```

On the contrary, every non-degenerate simplex can be coded in two different ways. For example the above 2-simplex σ_{135} of Δ^∞ can be coded as the list `(1 3 5)` *as a "geometric" simplex* (the first solution) or (second solution) as the `asm` object constructed as follows:

```
> (asm empty-list '(1 3 5))  ==>
<ASM * (1 3 5)>
```

The '`*`' in the result represents the void list, and the '`2-0`' in the previous result represents the composite degeneracy $\eta_2\eta_0$. In particular handling non-degenerate simplices needs much care about the necessary representation. Usually, if a function works only on non-degenerate simplices, the "geometric" representation is assumed; the functions handling possibly degenerate simplices use the "arbitrary" representation as an `asm` object, even if the considered simplex is non-degenerate.

Now we can understand the esoteric function installed in the `gdl` slot for Δ^∞. The EAT function `build-ss` needs in the `gdl` slot a function working on a face-index, a dimension and a "geometric" (non-degenerate) simplex of this dimension; this function must compute the right face of this simplex, which could be a degenerate simplex, so that the arbitrary format (`asm` object) is necessary to express the result, even if the result is non-degenerate, and this is just the case here. Note also a small difference with the organization of the previous section: now a simplex does not contain its dimension, so that when some work is asked for a simplex, its dimension must be also given. The reasons for this organization, less convenient from a theoretical point of view, will not be discussed here; it is a matter of memory efficiency. Note again that in the Δ^∞ example, it is in fact possible to deduce the dimension of a "geometric" simplex from its coding, but even in such a case the (right) dimension must be provided as argument for the function to be installed in the `gdl` slot.

The EAT strategy uses the well known fact that the structure of a simplicial set is entirely determined by the non-degenerate simplices and their faces (which in general may be degenerate).

Normally, the various slots of the structure implementing Δ^∞ are not directly used; the EAT program provides a relatively large set of functions allowing the user to easily ask some result or other. For example the `gdl` slot could be directly used to compute a face:

```
> (funcall (ss-gdl delta-infinity) 2 4 '(1 3 5 7 9))  ==>
<ASM * (1 3 7 9)>
```

but it is a little simpler and more readable to use the EAT predefined `gdl` function; such a function like many others, must firstly quote the simplicial set where the computation is done, the other arguments are the arguments necessary for the corresponding functional slot:

```
> (gdl delta-infinity 2 4 '(1 3 5 7 9))  ==>
<ASM * (1 3 7 9)>
```

The letter 'g' of `gdl` recalls this function may work only on *geometric* simplices, in other words on non-degenerate simplices correctly coded. Otherwise an error is generated (even if the simplex is non-degenerate):

```
> (gdl delta-infinity 2 4 (asm empty-list '(1 3 5 7 9)))  ==>
;; Error: Argument <ASM * (1 3 5 7 9)> to sequence function
;; is not a sequence.
```

Another EAT predefined function, the `adl` function, is provided for this work:

```
> (adl delta-infinity 2 4 (asm empty-list '(1 3 5 7 9)))  ==>
<ASM * (1 3 7 9)>
> (adl delta-infinity 2 4 '(1 3 5 7 9))  ==>
;; Error: Argument (1 3 5 7 9) was not a structure in ADL.
> (adl delta-infinity 2 4 (asm '(0) '(1 3 5 7)))  ==>
<ASM 0 (1 5 7)>
> (adl delta-infinity 1 4 (asm '(0) '(1 3 5 7)))  ==>
<ASM * (1 3 5 7)>
```

The `adl` function computes correctly $\partial_2\sigma_{13579} = \sigma_{1379}$; on the contrary the `adl` function may not work on the "geometric" coding of the same simplex. But the `adl` function may also compute $\partial_2\eta_0\sigma_{1357} = \eta_0\sigma_{157}$ and $\partial_1\eta_0\sigma_{1357} = \sigma_{1357}$: the `adl` function knows the commutation relations between face and degeneracy operators; in the last example, the geometric simplex `(1 3 5 7)` in fact was not really involved in the computation; it was simply a use of the commutation relation: $\partial_1\eta_0 = \mathrm{id}$.

The `sbs` field may not really be used, an error will necessarily occur:

```
> (sbs delta-infinity 2)  ==>
;; Error: The simplicial set [SS-4] is locally-effective.
```

because the simplicial set is only locally effective.

Two non-degenerate simplices can be compared by the `eqs` function, *without giving* the dimension of the simplices to be compared; in particular these simplices are assumed to have *the same dimension*. A consequence is the following: the equality function must be able to work without dimension information, a fact which can have some influence when the coding of non-degenerate simplices is chosen. The (possibly) degenerate simplices can be compared with the `eqas` function. A few examples:

```
> (eqs delta-infinity '(1 3 5) '(1 3 5))  ==>
T
> (eqas delta-infinity (asm empty-list '(1 3 5))
                       (asm empty-list '(1 3 5)))  ==>
T
> (eqs delta-infinity '(1 3 5)
                      (asm empty-list '(1 3 5)))  ==>
NIL
> (eqas delta-infinity '(1 3 5) '(1 3 5))  ==>
;; Error: Argument (1 3 5) was not a structure in EQAS.
```

The third result is rather strange and the reader would probably prefer an error result: it should be illegal to use the `eqs` function to compare both versions (`gsm` and `asm`) of σ_{135}; yes, it is illegal, but the EAT function `eqs` only asks Lisp to compare both objects with the Lisp function `equal`, installed in the `eqs` slot of Δ^∞; this Lisp function may compare two arbitrary Lisp objects, so that the comparison works and returns a negative result: the `asm` and `gsm` objects are (Lisp-) different! We recall EAT in general does not control the coherence of what you ask for and can possibly work in a perfectly absurd way if the data are not correct. It is obvious a more complete version of these programs should allow the user to turn on and off a safety switch; if on, then the EAT functions would verify the coherence of data; if off, the program would assume the data are coherent and would immediately work; the safe mode would be safer but also slower; for long and difficult calculations, the second mode would be necessary to save time, but coherence would be under the user's responsability; here only the “off" version is available.

On the contrary an error occurs in the fourth example, for Lisp is unable to extract the slots of `(1 3 5)`: the `eqas` function should find two `asm` objects.

The reader should be puzzled by the absence of a `type` slot, able to determine whether a Lisp object is a simplex of the underlying simplicial set. Nothing prevents from completing the program with such a slot, but the rest of this paper will prove this type function, useful to understand the *mathematical* structure of the program, in fact is without use in the planned computations! So that, for ending more quickly the EAT program, these type slots have simply be omitted. With this organization the `ss` instance `delta-infinity` *is not* a mathematical definition of Δ^∞; this object does not contain any information about simplex sets, so that any simplicial set which is a simplicial complex, with vertices among Lisp objects, the simplices being naturally coded as Lisp lists, is correctly coded as a *locally* effective simplicial set by the same `ss` instance!

A simplicial set $\overline{\Delta}^\infty$ (all the vertices are identified) can be constructed in the same way; the only difference is in the equality function: if the simplices to be compared are of dimension 0, then they certainly are equal:

```
> (setf delta-infinity-bar
    (build-ss
     :bsp '(0)
     :eqs #'(lambda (gsm1 gsm2)
              (or (= 1 (length gsm1))
                  (equal gsm1 gsm2)))
     :sbs :locally-effective
     :gdl (ss-gdl delta-infinity)
     :org '(delta-infinity-bar)))  ==>
[SS-5]
```

Taking account of the hypotheses about the simplices to be compared, if the first one has dimension 0, then the second simplex has also dimension 0 and they are equal: there is only one simplex in dimension 0. Otherwise the simplices, some integer lists, can be compared by the standard `equal` Lisp function. The `gdl` slot can be copied from Δ^∞; this works well, even in dimension 1:

```
> (setf 0-face (gdl delta-infinity-bar 0 1 '(5 6)))  ==>
<ASM * (6)>
> (setf 1-face (gdl delta-infinity-bar 1 1 '(5 6)))  ==>
<ASM * (5)>
> (eqas delta-infinity-bar 0-face 1-face)  ==>
T
> (eqas delta-infinity 0-face 1-face)  ==>
NIL
```

The 0-simplices carried respectively by the 5 and 6 vertices are equal in $\overline{\Delta}^\infty$, but different in Δ^∞.

The standard 5-simplex Δ^5 can be implemented in the same way; because Δ^5 is finite, an effective simplicial set can be installed:

```
> (setf delta-5
    (build-ss
      :bsp '(0)
      :eqs #'equal
      :sbs #'(lambda (dmn)
               (delta-inj dmn 5))
```

```
      :gdl (ss-gdl delta-infinity)
      :org '(delta-5))) ==>
[SS-6]
```

The unique difference with Δ^∞ is the `sbs` slot; it is now a function working on one integer argument and returning the list of *non-degenerate* (`gsm`) simplices in this dimension:

```
> (sbs delta-5 0) ==>
((0) (1) (2) (3) (4) (5))
> (sbs delta-5 1)
((0 1) (0 2) (1 2) (0 3) (1 3) (2 3) (0 4)
 (1 4) (2 4) (3 4) (0 5) (1 5) (2 5) (3 5) (4 5))
> (sbs delta-5 5)
((0 1 2 3 4 5))
> (sbs delta-5 1000)
NIL
```

If you are interested by a list including the degenerate simplices (an `asm` object list), you can use the `sbs-d` function:

```
> (sbs-d delta-5 1) ==>
(<ASM 0 (0)> <ASM 0 (1)> <ASM 0 (2)> <ASM 0 (3)>
 <ASM 0 (4)> <ASM 0 (5)> <ASM * (0 1)> <ASM * (0 2)>
 <ASM * (1 2)> <ASM * (0 3)> <ASM * (1 3)> ...)
```

The `sbs` (simplicial basis) slot for Δ^5 uses the EAT predefined `delta-inj` function computing the ordered injections between two objects of the abstract Δ category [30]:

```
> (delta-inj 2 4)
((0 1 2) (0 1 3) (0 2 3) (1 2 3) (0 1 4)
 (0 2 4) (1 2 4) (0 3 4) (1 3 4) (2 3 4)) ==>
```

Each injection is represented as the image list. Note also that the Lisp object `delta-5` *is* a mathematical definition of Δ^5: the `sbs` slot gives a complete description of the simplex sets.

6.2 EAT and the chain complexes.

The EAT program allows the user to construct chain complexes in an analogous way, but we do not want to burden the reader with the various technical (but necessary) tools available in the EAT program. For understanding the rest of this paper, it will be sufficient to see how the chain complexes associated to simplicial sets can be automatically obtained, and then used, for example to compute homology groups.

The EAT `ss-cc` function constructs the (normalized) chain complex canonically associated to a simplicial set. The following simple example, starting from the previous implementation of Δ^5, explains the process:

```
> (setf cc-delta-5 (ss-cc delta-5))  ==>
[CC-7]
```

The `ss-cc` function has worked on the `ss` instance Δ^5 and the result, the chain complex #7 is assigned to the symbol `cc-delta-5`. A `cc` (chain complex) object has several slots; the main ones can be used as follows.

Because Δ^5 is an *effective* simplicial set, the associated chain complex is also effective, that is, global information is available. Each chain group is a free $\mathbb{Z}$-module and its basis can be obtained by the `cbs` (chain complex basis) function:

```
> (cbs cc-delta-5 1)
((0 1) (0 2) (1 2) (0 3) (1 3) (2 3) (0 4)
 (1 4) (2 4) (3 4) (0 5) (1 5) (2 5) (3 5) (4 5))
> (cbs cc-delta-5 1000)
NIL
> (cbs cc-delta-5 -1)
NIL
```

The basis of $C_1\Delta^5$ is nothing but the list of nondegenerate 1-simplices of Δ^5. There is no such simplex in dimension 1000, and also in dimension -1. The boundary of a generator can be computed by the EAT predefined `d-?` function:

```
> (d-? cc-delta-5 2 '(1 2 3))  ==>
---------------------------------------------------{CMB 1}
<MNM 1 * (1 2)>
<MNM -1 * (1 3)>
<MNM 1 * (2 3)>
----------------------------------------------------------
```

Here the boundary of the generator σ_{123} of $C_2\Delta^5$ is asked: the arguments of the `d-?` function are:

1. The chain complex where the calculation is asked;
2. The dimension of the generator;
3. The generator itself.

The result is a `cmb` object, that is a linear combination of generators; the degree of the combination is displayed in the top right-hand corner (1 here) and, between the dashed lines, a list of "monomials" is enumerated; each monomial contains a coefficient (1 or -1 here) and the corresponding generator; the '$*$' sign recalls the "product": coefficient times generator. A combination can be made by the user as follows:

```
> (setf cmb-sample
        (cmb 1 100 '(1 2) 10 '(1 3) 1 '(2 3)))  ==>
---------------------------------------------------{CMB 1}
<MNM 1 * (2 3)>
<MNM 10 * (1 3)>
<MNM 100 * (1 2)>
----------------------------------------------------------
```

The degree of the combination to be constructed must firstly be given; then, for each term, the coefficient and the generator is inserted. The boundary of a combination is computed with the `d-???` function:

```
> (d-??? cc-delta-5 cmb-sample)  ==>
---------------------------------------------------{CMB 0}
<MNM -110 * (1)>
<MNM 11 * (3)>
<MNM 99 * (2)>
----------------------------------------------------------
```

Now the degree must not be given: the `cmb` object in fact contains this degree, but of course the chain complex where the calculation occurs must be indicated. It is easy to verify the rule $d^2 = 0$ for a particular case:

```
> (d-??? cc-delta-5
         (d-??? cc-delta-5 (cmb 5 1000 '(0 1 2 3 4 5))))  ==>
---------------------------------------------------{CMB 3}
----------------------------------------------------------
```

No “monomial" is indicated, and the resulting combination is therefore null.

The homology groups of an *effective* chain complex can be computed by the `cc-homology` function. For example let us compute $H_0 C_* \Delta^5$:

```
> (cc-homology cc-delta-5 0)  ==>
Computing boundary-matrix in dimension 0.
Rank of the source-module : 6.

;; Clock -> 16h 53m 45s.
Computing the boundary of the generator 1 (dimension 0) :
(0)
End of computing.

;; Clock -> 16h 53m 45s.
Computing the boundary of the generator 2 (dimension 0) :
(1)
End of computing.
[...Lines deleted...]
;; Clock -> 16h 53m 46s.
Computing the boundary of the generator 6 (dimension 0) :
(5)
End of computing.

Computing boundary-matrix in dimension 1.
Rank of the source-module : 15.

;; Clock -> 16h 53m 47s.
Computing the boundary of the generator 1 (dimension 1) :
(0 1)
End of computing.
[...Lines deleted...]
;; Clock -> 16h 53m 51s.
```

```
Computing the boundary of the generator 15 (dimension 1) :
(4 5)
End of computing.

Homology in dimension 0 :
Component Z

---done---
```

Because such a computation can be long (several days sometimes), a little information is displayed for the intermediate steps to allow the user to have some idea about the progress of its calculation. The EAT program must firstly compute the basis of three chain groups and the corresponding boundary matrices; here the C_{-1} group is null, but this is not visible in the output. The rank of the source modules is given and then the beginning of the computation of the boundary of each generator is signalled. Finally the result is displayed as a list of components ($\mathbb{Z}$ or $\mathbb{Z}/n\mathbb{Z}$). No component at all means the homology group is null:

```
> (cc-homology cc-delta-5 2)  ==>
Computing boundary-matrix in dimension 2.
Rank of the source-module : 20.

;; Clock -> 17h 15m 41s.
Computing the boundary of the generator 1 (dimension 2) :
(0 1 2)
End of computing.
[...Lines deleted...]
;; Clock -> 17h 15m 47s.
Computing the boundary of the generator 20 (dimension 2) :
(3 4 5)
End of computing.

Computing boundary-matrix in dimension 3.
Rank of the source-module : 15.

;; Clock -> 17h 15m 47s.
Computing the boundary of the generator 1 (dimension 3) :
(0 1 2 3)
End of computing.
[...Lines deleted...]
;; Clock -> 17h 15m 52s.
Computing the boundary of the generator 15 (dimension 3) :
(2 3 4 5)
End of computing.

Homology in dimension 2 :

---done---
```

The simplicial set Δ^5 is contractible and kernel and image in $C_2\Delta^5$ are equal.

The reader probably wishes here an example with torsion in homology. Let X the wedge of the 3-sphere and the Moore space Moore($\mathbb{Z}/3\mathbb{Z}, 3$):

```
> (setf X (wedge (sphere 3) (moore 3 3)))  ==>
[SS-11]
> (cc-homology (ss-cc X) 3)  ==>
Computing boundary-matrix in dimension 3.
Rank of the source-module : 2.

;; Clock -> 17h 24m 25s.
Computing the boundary of the generator 1 (dimension 3) :
<W-GSM 1 <S3>>
End of computing.

;; Clock -> 17h 24m 25s.
Computing the boundary of the generator 2 (dimension 3) :
<W-GSM 2 <M3>>
End of computing.

Computing boundary-matrix in dimension 4.
Rank of the source-module : 1.

;; Clock -> 17h 24m 26s.
Computing the boundary of the generator 1 (dimension 4) :
<W-GSM 2 <MM4>>
End of computing.

Homology in dimension 3 :

Component Z/3Z
Component Z

---done---
```

There are only three nondegenerate simplices for X in dimensions 3 and 4. The faces and the boundary of the only 4-simplex are:

```
> (dotimes (index 5)
     (print (gdl X index 4 (w-gsm 2 '<MM4>))))  ==>
<ASM * <W-GSM 2 <M3>>>
<ASM 2-1-0 <W-GSM 0 W-BSP>>
<ASM * <W-GSM 2 <M3>>>
<ASM 2-1-0 <W-GSM 0 W-BSP>>
<ASM * <W-GSM 2 <M3>>>
NIL
> (d-? (ss-cc X) 4 (w-gsm 2 '<MM4>))  ==>
--------------------------------------------------{CMB 3}
<MNM 3 * <W-GSM 2 <M3>>>
----------------------------------------------------------
```

In the first result, the index runs from 0 (inclusive) to 5 (exclusive), and for each value, the corresponding face is displayed; the terminal `NIL` has no meaning, it is a by-product of the `dotimes` process. In the second result, the boundary of the 4-simplex of the Moore space is three times the 3-simplex of the same space.

The EAT program can also construct the chain complex canonically associated to a locally effective simplicial set, but the chain complex result is then also locally effective:

```
> (setf cc-delta-infinity (ss-cc delta-infinity))  ==>
[CC-10]
> (cbs cc-delta-infinity 3)  ==>
;; Error: The chain complex [CC-10] is locally-effective.
```

No global information is available for the chain complex $C_*\Delta^\infty$, and it is therefore not possible to get a basis of $C_3\Delta^\infty$. Trying to compute a homology group of $C_*\Delta^\infty$ fails:

```
> (cc-homology cc-delta-infinity 2)  ==>
;; Error: CC-MAT cannot work
;;        with a LOCALLY-EFFECTIVE chain complex.
```

The `CC-MAT` function is called by `cc-homology` to compute the boundary matrices; this is not possible if the chain complex is locally effective. But this does not prevent the user to undertake “local" computations:

```
> (d-? cc-delta-infinity 3 '(12 456 678 2222222))  ==>
-------------------------------------------------{CMB 2}
<MNM -1 * (12 456 678)>
<MNM 1 * (12 456 2222222)>
<MNM -1 * (12 678 2222222)>
<MNM 1 * (456 678 2222222)>
-------------------------------------------------------
```

We have here the *essential tool* which will allow us to organize our solution for the problem of constructive algebraic topology: of course an infinite object cannot be installed in a finite machine, but a partial implementation can be done, which will be just sufficient for our work. The new *mathematical* notion of locally effective object will exempt us from the complicated machineries of Schön [40], Smirnov [46] and Justin Smith [47]; we will work in this way closely to usual algebraic topology.

6.3 EAT and the chain complex morphisms.

After the *objects* constructed and handled by EAT, we must now consider the *morphisms* between them, that is, the chain complex morphisms. The `mrp` (<u>m</u>o<u>r</u><u>p</u>hism) EAT objects are more generally morphisms of graded $\mathbb{Z}$-modules, not necessarily compatible with differentials. For example the homotopy operators are (can be implemented as) `mrp` objects in the EAT environment.

We illustrate what is possible with the EAT program about morphisms by the following exercise: the chain complex $C_*\Delta^\infty$ is locally effective and its homology groups cannot be directly computable. But it is well known this complex is acyclic; we construct a contracting homotopy h and verify the relation $\mathrm{id} - hd - dh = 0$ for particular cases.

After all, the differential of a chain complex can be considered as a “morphism" and, in the EAT organization, it is actually a `mrp` object which can be extracted from the chain complex by the `cc-d` function:

```
> (setf d (cc-d cc-delta-infinity))  ==>
[MRP-10]
```

This differential morphism has been assigned to the symbol `d`. It is possible to make this morphism work on some generator; compare both following statements:

```
> (? d 3 '(4 5 6 7))  ==>
---------------------------------------------------{CMB 2}
<MNM -1 * (4 5 6)>
<MNM 1 * (4 5 7)>
<MNM -1 * (4 6 7)>
<MNM 1 * (5 6 7)>
----------------------------------------------------------
> (d-? cc-delta-infinity 3 '(4 5 6 7))  ==>
---------------------------------------------------{CMB 2}
<MNM -1 * (4 5 6)>
<MNM 1 * (4 5 7)>
<MNM -1 * (4 6 7)>
<MNM 1 * (5 6 7)>
----------------------------------------------------------
```

Both statements are equivalent, and in fact EAT converts the second one into the first one[25]. When a chain complex is build by EAT, the differential is constructed as a morphism with source and target this chain complex; source and target of a morphism can be obtained by the functions `mrp-src` and `mrp-trg`. The circularity of this organization can be made visible:

```
> cc-delta-infinity  ==>
[CC-10]
> (mrp-src (cc-d cc-delta-infinity))  ==>
[CC-10]
> (mrp-src (cc-d cc-delta-infinity))  ==>
[CC-10]
> d  ==>
[MRP-10]
> (cc-d (mrp-src d))  ==>
[MRP-10]
> (cc-d (mrp-trg d))  ==>
[MRP-10]
```

A morphism can also work on combinations:

```
> (??? d (cmb 2 10 '(1 2 3) 100 '(2 3 4)))  ==>
---------------------------------------------------{CMB 1}
<MNM -10 * (1 3)>
<MNM 10 * (1 2)>
<MNM 100 * (3 4)>
<MNM -100 * (2 4)>
<MNM 110 * (2 3)>
```

[25] the '?' function in the EAT program may be applied only on the `mrp` instances, not on any functional object like in the previous sections.

```
---------------------------------------------------------
> (d-??? cc-delta-infinity (cmb 2 10 '(1 2 3) 100 '(2 3 4)))  ==>
--------------------------------------------------{CMB 1}
<MNM -10 * (1 3)>
<MNM 10 * (1 2)>
<MNM 100 * (3 4)>
<MNM -100 * (2 4)>
<MNM 110 * (2 3)>
---------------------------------------------------------
```

The usual contracting homotopy for Δ^∞ is the operator "cone with respect to the 0 vertex"; it can be constructed in the EAT framework as follows:

```
> (setf h
        (build-mrp
           :src cc-delta-infinity
           :trg cc-delta-infinity
           :dgr +1
           :f #'(lambda (dgr gnr)
                   (if (= 0 (first gnr))
                      (cmb (1+ dgr))
                      (cmb (1+ dgr) 1 (cons 0 gnr))))
           :org '(delta-infinity-contraction)))  ==>
[MRP-11]
```

A morphism is constructed with the slots `src` (source), `trg` (target), `dgr` (degree), `f` (the Lisp function giving the image of a generator as a combination) and `org` (a comment). Here the `f` slot contains the function returning the null combination if the generator has the 0 vertex as first element, or otherwise the 0-cone with respect to the generator, presented as a combination; a combination `(cmb 3)` is the null combination of degree 3. Our homotopy operator can be tried:

```
> (? h 3 '(4 5 6 7))  ==>
--------------------------------------------------{CMB 4}
<MNM 1 * (0 4 5 6 7)>
---------------------------------------------------------
> (? h 3 '(0 5 6 7))  ==>
--------------------------------------------------{CMB 4}
---------------------------------------------------------
```

Morphisms can be composed, added, subtracted:

```
> (setf dh (cmp-mrp d h))  ==>
[MRP-12]
> (setf hd (cmp-mrp h d))  ==>
[MRP-13]
> (setf dh+hd (add-mrp-to-mrp dh hd))  ==>
[MRP-14]
> (setf id-cdelta (id-mrp cc-delta-infinity))  ==>
[MRP-15]
> (setf should-be-null
        (sbt-mrp-from-mrp dh+hd id-cdelta))  ==>
[MRP-16]
```

When reading the above Lisp lines, it is important to know the text `dh+hd` is nothing but *one* symbol without any addition asked; it is only a notation. Now we can verify the relation $\mathrm{id} = dh + hd$ for particular cases:

```
> (? should-be-null 3 '(4 5 6 7))  ==>
--------------------------------------------------{CMB 3}
----------------------------------------------------------
> (? should-be-null 3 '(0 5 6 7))  ==>
--------------------------------------------------{CMB 3}
----------------------------------------------------------
```

6.4 Conclusion.

We hope the reader begins to understand the general organization of the EAT program. The traditional mathematical objects of algebraic topology are implemented as structures where the important slots are Lisp functions. The ordinary constructions of new mathematical objects from others already available is implemented as constructions of new structure instances, where the functional slots can be constructed in using the functional slots of the given objects; it is a simple question of *functional programming.* And in some particular cases, these functional slots can be used to obtain some intermediate "local" result.

7 EAT and the loop spaces.

What about the loop space construction in the EAT program? A combinatorial version of the loop space construction must firstly be given. Kan [29] gave such a version, able to work on reduced simplicial sets.

Definition 8 — A simplicial set $X = \{X_n\}$ is *reduced* if the vertex set X_0 has only one element. More generally, if n is a natural number, the simplicial set X is *n-reduced* if it is reduced and if, for every $0 < k \leq n$, the simplex set X_k does not have any non-degenerate simplex.

The unique vertex of a reduced simplicial set is usually considered as the *base point* of the simplicial set; if X is n-reduced, it is n-connected ($\pi_k X = 0$ for $0 \leq k \leq n$), and, conversely, any n-connected simplicial set has in its homotopy type an n-reduced version.

A tricky process, due to Kan [29], constructs a combinatorial version GX of the loop space $\Omega|X|$, for every *reduced* simplicial set X. Here, $|X|$ is the *realization* of the simplicial set X, that is, the *topological* space canonically associated to the (combinatorial) simplicial set X; see [23, 32, 30]. In this paper, because we are only interested in homotopy types, we simply call GX "the" loop space of X.

The loop space GX is a simplicial set and we must firstly define its simplex sets $\{GX_n\}_{n\in\mathbb{N}}$. For n a positive integer, let us denote by $X_n^* \subset X_n$ the subset of the n-simplices that are not 0-degenerate: $X_n^* = X_n - \eta_0 X_{n-1}$. Then GX_n is defined as the free *non-commutative* group generated by X_{n+1}^*; this is the set of words with respect to the alphabet X_{n+1}^*, where each letter has also an exponent, a non-null integer, positive or negative, this set being divided by the equivalence relation generated by $\sigma^m\sigma^{-m} = 1$ for every letter $\sigma \in X_{n+1}^*$ and every integer m; the group law is defined by concatenation. If the generator set is non-empty, the non-commutative freely generated group is highly infinite.

Face and degeneracy operators remain to be defined for GX. It is convenient to denote $\tau(\sigma)$ the generator of the free group GX_n corresponding to the simplex $\sigma \in X_{n+1}^*$. Then the face (resp. degeneracy) operators will be group homomorphisms $\partial_i : GX_n \to GX_{n-1}$ (resp. $\eta_i : GX_n \to GX_{n+1}$); in other words, the simplicial set GX will be defined as a *simplicial group* (see [32, 30]). Now, because GX_n is a free group, it is sufficient to define the face and degeneracy operators for the generators $\tau(\sigma) \in GX_n$, for every simplex $\sigma \in X_{n+1}^*$:

$$\begin{array}{rcll}
\partial_i\tau(\sigma) & = & \tau(\partial_{i+1}\sigma), & \text{if } 1 \leq i \leq n; \\
\partial_0\tau(\sigma) & = & \tau(\partial_1\sigma)\tau(\partial_0\sigma)^{-1}; & \\
\eta_i\tau(\sigma) & = & \tau(\eta_{i+1}\sigma), & \text{if } 0 \leq i \leq n.
\end{array}$$

Several choices are possible; for example it is possible to replace the second formula by the following:

$$\partial_0\tau(\sigma) = \tau(\partial_0\sigma)^{-1}\tau(\partial_1\sigma);$$

but this implies a corresponding different choice for the action (right or left?) of the structural groups in the simplicial fibrations considered later. In the same way the choice of the "special" index 0 (again for the second formula) can be replaced by n with a corresponding change in the result. The choices given here are those that have been done in the EAT program.

The EAT `loop-space` function works on a reduced simplicial set X and returns a *locally-effective* simplicial set GX. For example, let us use this function to construct the loop space $G\overline{\Delta}^5$ of the standard 5-simplex where all the vertices are identified:

```
> (setf delta-bar-5
      (build-ss
         :bsp '(0)
         :eqs #'(lambda (gsm1 gsm2)
                  (if (= 1 (length gsm1))
                     t
                     (equal gsm1 gsm2)))
         :sbs #'(lambda (dmn)
                  (if (zerop dmn)
                     '((0))
                     (delta-inj dmn 5)))
         :gdl #'(lambda (i dmn gsm)
                  (asm empty-list
                       (append (subseq gsm 0 i)
```

```
                                    (subseq gsm (1+ i)))))
         :org '(delta-bar-5)))
[SS-4]
> (setf g-delta-bar-5 (loop-space delta-bar-5))
[SS-5]
```

We assume a new Lisp session is restarted, with the EAT program loaded. The `ss` object `delta-bar-5` is constructed to modelize $\overline{\Delta}^5$; a simplex is an increasing list of integers between 0 and 5. In particular a 0-simplex is a *list* of one integer and any two such simplices in fact are the same: only one vertex in $\overline{\Delta}^5$. Then the predefined EAT function `loop-space` can be invoked to construct the loop-space $G\overline{\Delta}^5$, another `ss` object located through the symbol `g-delta-bar-5`. For example let us look at its base point:

```
> (ss-bsp g-delta-bar-5)  ==>
<<LOOP *>>
```

which is the trivial loop. Let us try now to look at the non-degenerate 3-simplices of $G\overline{\Delta}^5$:

```
>  (sbs g-delta-bar-5 3)  ==>
;; Error: The simplicial set [SS-5] is locally-effective.
```

This combinatorial version of the loop-space of $\overline{\Delta}^5$ is highly infinite; there are 252 (degenerate or not) simplices of dimension 4 in $\overline{\Delta}^5$ and 126 among them are not 0-degenerate; so that the set of 3-simplices in $G\overline{\Delta}^5$ is a free non-commutative group with 126 generators. But this situation does not prevent us from undertaking *local* computations, a "local" computation being understood as using only a finite number of simplices.

For example let us denote again by σ_{02345} the simplex of $\overline{\Delta}^5$ spanning the vertices 0, 2, 3, 4 and 5[26]. Then $\tau(\sigma_{02345})^2$ is a simplex of $G\overline{\Delta}^5$ and we should be able to compute its faces. We must firstly *code* this simplex; the predefined `loop3` function allows the EAT user to do it:

```
> (setf tau-s02345-squared
        (loop3 empty-list '(0 2 3 4 5) 2))  ==>
<<LOOP (<PWR * (0 2 3 4 5) ** 2>)>>
```

The loop constructor is called `loop3` because each information unit needs three arguments:

1. a composite degeneracy operator;
2. a non-degenerate (`gsm`) simplex of the original simplicial set;

[26]More precisely the simplex of $\overline{\Delta}^5$ coming from the simplex spanning the vertices 0, 2, 3, 4 and 5 of Δ^5 divided by the relation identifying its vertices.

3. a power indicator (a non-null integer).

Here the original simplex σ_{02345} is non-degenerate so that the first argument is the empty list. The second argument is the original simplex σ_{02345} denoted by an integer list; finally the third argument is the power 2. The result is a list between brackets, prefixed by the indication `LOOP`, the list being constituted of "powers" (again a bracketed list prefixed by the indication `PWR`). Look at the following example, a little more complicated:

```
> (setf more-complicated-loop
     (loop3 '(2 1) '(0 2 3 4 5) 1
            '(3 1) '(0 2 3 4 5) -2
            '(3 2) '(0 2 3 4 5) 1))  ==>
<<LOOP (<PWR 2-1 (0 2 3 4 5) ** 1>
        <PWR 3-1 (0 2 3 4 5) ** -2>
        <PWR 3-2 (0 2 3 4 5) ** 1>)>>
```

Here the loop $\tau(\eta_2\eta_1\sigma_{02345})^1\tau(\eta_3\eta_1\sigma_{02345})^{-2}\tau(\eta_3\eta_2\sigma_{02345})^1$ is constructed. The interpretation of the statement and the result should be clear. Because there is no intersection between the three composite degeneracy operators, this loop is non-degenerate and the object now located by the symbol `more-complicated-loop` is a *legal* `gsm` simplex for $G\overline{\Delta}^5$. On the contrary the following one is illegal:

```
> (setf erroneous-loop
     (loop3 '(2 1) '(0 2 3 4 5) 1
            '(3 1) '(0 2 3 4 5) -2))  ==>
<<LOOP (<PWR 2-1 (0 2 3 4 5) ** 1>
        <PWR 3-1 (0 2 3 4 5) ** -2>)>>
```

This loop is an illegal `gsm` object in $G\overline{\Delta}^5$; because of the definition of degeneracy operators in $G\overline{\Delta}^5$, we have the relation: $\eta_0(\tau(\eta_1\sigma_{02345})^1\tau(\eta_2\sigma_{02345})^{-2}) = \tau(\eta_2\eta_1\sigma_{02345})^1\tau(\eta_3\eta_1\sigma_{02345})^{-2}$. In other words this loop is degenerate, mainly because the number 1 is in the intersection of the degeneracy families. The predefined function `normalize-loop` is able to examine this question; it needs the dimension of the loop considered and the claimed `gsm` loop itself; the result is expressed as a correct `asm` object:

```
> (normalize-loop 5 erroneous-loop)  ==>
<ASM 0 <<LOOP (<PWR 1 '(0 2 3 4 5) ** 1>
               <PWR 2 '(0 2 3 4 5) ** -2>)>>>
> (normalize-loop 5 more-complicated-loop)  ==>
<ASM * <<LOOP (<PWR 2-1 '(0 2 3 4 5) ** 1>
               <PWR 3-1 '(0 2 3 4 5) ** -2>
               <PWR 3-2 '(0 2 3 4 5) ** 1>)>>>
```

You see the `normalize-loop` function detects the degeneracy property of the wrongly coded loop; on the contrary the other loop was really non-degenerate.

The `ss` object located by `g-delta-bar-5` contains the necessary information to compute simplex faces:

```
> (gdl g-delta-bar-5 1 3 tau-s02345-squared)  ==>
<ASM * <<LOOP (<PWR * (0 2 4 5) ** 2>)>>>
```

The formula $\partial_1 \tau(\sigma) = \tau(\partial_2 \sigma)$ is applied. For the 0-face instead of the 1-face:

```
> (gdl g-delta-bar-5 0 3 tau-s02345-squared)  ==>
<ASM * <<LOOP (<PWR * (0 3 4 5) ** 1>
               <PWR * (2 3 4 5) ** -1>
               <PWR * (0 3 4 5) ** 1>
               <PWR * (2 3 4 5) ** -1>)>>>
```

You see the program knows ∂_0 is a free group homomorpphism. The same computations for the more complicated loop:

```
> (gdl g-delta-bar-5 1 3 more-complicated-loop)  ==>
<ASM * <<LOOP (<PWR 1 (0 2 3 4 5) ** 1>
               <PWR 2 (0 2 3 4 5) ** -1>)>>>
> (gdl g-delta-bar-5 0 3 more-complicated-loop)  ==>
<ASM * <<LOOP (<PWR 1 (0 2 3 4 5) ** 1>
               <PWR 2 (0 2 3 4 5) ** -2>
               <PWR 2-1 (0 3 4 5) ** 1>
               <PWR 2-1 (2 3 4 5) ** -1>)>>>
```

The reader can easily verify the results. The function `loop-space` uses the functional components, essentially the face operator `gdl`, of the original simplicial set to construct the corresponding functional components of the loop space; as previously it is a question of writing functions working on functional arguments and able to construct a functional result.

Because $G\overline{\Delta}^5$ is a (locally effective) `ss` object, the `ss-cc` function can compute the corresponding chain complex:

```
> (setf cc-g-delta-bar-5
     (ss-cc g-delta-bar-5))  ==>
[CC-7]
```

which can be *locally* used:

```
> (d-? cc-g-delta-bar-5 3 tau-s02345-squared)  ==>
----------------------------------------------------{CMB 2}
<MNM -1 * <<LOOP (<PWR * (0 2 3 4) ** 2>)>>>
<MNM 1 * <<LOOP (<PWR * (0 2 3 5) ** 2>)>>>
<MNM -1 * <<LOOP (<PWR * (0 2 4 5) ** 2>)>>>
<MNM 1 * <<LOOP (<PWR * (0 3 4 5) ** 1>
                 <PWR * (2 3 4 5) ** -1>
                 <PWR * (0 3 4 5) ** 1>
                 <PWR * (2 3 4 5) ** -1>)>>>
----------------------------------------------------
```

and it is possible to verify the relation $d \circ d = 0$ for particular cases :

```
> (d-??? cc-g-delta-bar-5
     (d-? cc-g-delta-bar-5 3 tau-s02345-squared))  ==>
----------------------------------------------------{CMB 1}
-----------------------------------------------------------
```

Only a finite number of simplices were concerned in this calculation. But computing the homology groups of this chain complex needs *global* information and, because the complex is locally effective:

```
> (cc-cbs cc-g-delta-bar-5)  ==>
:locally-effective
```

this is not possible:

```
> (cc-homology cc-g-delta-bar-5 3)  ==>
;; Error: CC-MAT cannot work with a LOCALLY-EFFECTIVE chain complex.
```

It is also possible to iterate the loop-space construction. For example, to construct a second loop space, we must start with a 1-reduced simplicial set. Let us consider the simplicial set $\overline{\Delta}^{5,3}$ defined as the infinite simplex divided by its 2-skeleton:

```
> (setf delta-bar-five-3
     (build-ss :bsp '(0)
               :eqs #'(lambda (gsm1 gsm2)
                          (if (= 1 (length gsm1))
                             t
                             (equal gsm1 gsm2)))
               :sbs #'(lambda (dmn)
                          (case dmn
                             (0 '((0)))
                             ((1 2) empty-list)
                             (otherwise (delta-inj dmn 5))))
               :gdl #'(lambda (i dmn gsm)
                          (case dmn
                             (0 (error "A face of a 0-dimensional simplex~@
                                        does not make sense."))
                             ((1 2) (error "No non-degenerate 1-simplex~@
                                            in delta-bar-five-3."))
                             (3 (asm '(1 0) '(0)))
                             (otherwise
                                (asm empty-list
                                     (append (subseq gsm 0 i)
                                             (subseq gsm (1+ i)))))))
              :org '(delta-bar-infinity-2)))  ==>
[SS-6]
```

The only significant difference with $\overline{\Delta}^5$ is that any face of a 3-simplex is the 2-dimensional degeneracy of the base point. The chain-complex of the second loop-space $C_* G^2 \overline{\Delta}^{5,3}$ is:

```
> (setf cc-g2-delta-bar-five-3
     (ss-cc (loop-space (loop-space delta-bar-five-3))))  ==>
[CC-8]
```

We can consider the simplex $\tau(\tau(\sigma_{01234})\tau(\sigma_{02345})^{-1})^2$:

```
> (setf double-loop
    (loop3 empty-list
           (loop3 empty-list '(0 1 2 3 4) 1
                  empty-list '(0 2 3 4 5) -1)
           2))  ==>
<<LOOP (<PWR * <<LOOP (<PWR * (0 1 2 3 4) ** 1>
                       <PWR * (0 2 3 4 5) ** -1>)>> ** 2>)>>
```

and compute its boundary in $C_*G^2\overline{\Delta}^{5,3}$:

```
> (d-? cc-g2-delta-bar-five-3 2 double-loop)  ==>
---------------------------------------------------{CMB 1}
<MNM 1 * <<LOOP (<PWR * <<LOOP (<PWR * (0 1 2 3) ** 1>...
<MNM -1 * <<LOOP (<PWR * <<LOOP (<PWR * (0 1 2 4) ** 1>...
<MNM 1 * <<LOOP (<PWR * <<LOOP (<PWR * (0 1 3 4) ** 1>...
----------------------------------------------------------
```

(only a small part of the result is copied here) and also verify $d \circ d = 0$ for this double loop:

```
> (d-??? cc-g2-delta-bar-five-3
     (d-? cc-g2-delta-bar-five-3 2 double-loop))  ==>
---------------------------------------------------{CMB 0}
----------------------------------------------------------
```

but it is not possible to directly compute the homology groups $H_*G^2\overline{\Delta}^{5,3}$:

```
> (cc-homology cc-g2-delta-bar-five-3 2)  ==>
;; Error: CC-MAT cannot work with a LOCALLY-EFFECTIVE chain complex.
```

Much more work is needed to be able to use these methods to compute such homology groups. This is the subject of the rest of this paper: we must add much information to an object like `cc-g2-delta-bar-five-3` and transform it into an *object with effective homology*.

8 A tentative appropriate mathematical language.

The main results of this paper concern the construction of some algorithms satisfying interesting properties. For example our main result about iterating the cobar construction has the following form.

Theorem 9 — *Let $\mathcal{SS}_n$ the type of n-reduced locally effective simplicial sets with effective homology; an algorithm λ can be constructed:*

$$\lambda : \mathcal{SS}_1 \to \mathcal{SS}_0$$

such that if $X \in \mathcal{SS}_1$, then $\lambda(X)$ is a version with effective homology of GX, the loop space of X. In particular, $\lambda(\mathcal{SS}_n) \subset \mathcal{SS}_{n-1}$ if n is a positive integer.

Many things are underlying in such a statement and probably the reader would prefer a more precise one. But such a precise statement would be very long and would certainly not help understanding. We must now explain what must actually be understood: this is the aim of the present section.

8.1 Types.

A *type* is nothing but the machine version of the mathematical notion of *set* (collection). Usually, in order to avoid the classical paradoxes of set theory, the notion of *class* is preferred for *all* the simplicial sets, but in our environment this does not matter because the objects are *machine objects*, taken among the (countable) *set* $\mathcal{U}$ of all the machine objects. However, the classical Russel paradox appears again under a slightly different form; for example the type $\mathcal{SS}$ is even not *locally effective*: in other words no algorithm can determine whether an arbitrary machine object is an element of $\mathcal{SS}$ and this is the reason why the notion of *type* is preferred. We shall explain later an object of type $\mathcal{SS}$ is a structure instance with a few components satisfying some properties; but in general it is impossible to verify these properties for an arbitrary object. So that an algorithm λ as above is in principle used only for objects X provided with a proof that really $X \in \mathcal{SS}$. On the contrary the various types considered in the previous sections, defining for example the simplex set of some simplicial set, were assumed to be locally effective, and in fact were defined by an algorithm implementing the corresponding characteristic function.

8.2 The theoretical Lisp machine.

It is possible to modelize the Lisp machine as follows. The set $\mathcal{U}$ (<u>u</u>niverse) is the countable set of machine objects. The set $\mathcal{E}$ is the countable set of <u>e</u>nvironments. The <u>e</u>valuator ε is a function $\varepsilon : \mathcal{E} \times \mathcal{U} \to (\mathcal{E} \times \mathcal{U}) \coprod \{?\}$. A mathematical definition of the Lisp machine such as the thick book [49] is nothing but a definition of the *mathematical* objects $\mathcal{U}$, $\mathcal{E}$ and ε. The following indications are here sufficient.

1. An *environment* $E \in \mathcal{E}$ is a set of bindings from symbols to arbitrary objects, usually called the *values* of these symbols when evaluation starts. An environment is strongly structured to implement the *lexical closure* mechanism, see [42, Nanolisp] or [3, section 3.10], but we will not consider these complex

notions here, giving us the possibility of using *functional programming*; we consider the numerous shown examples should be sufficient.

2. An object $\omega \in \mathcal{U}$ is some "machine object" such as a number, a string character, a function,...

3. If E is an environment and ω a machine object, the value $(E', \omega') = \varepsilon(E, \omega)$ is a pair where the first component E' is the new environment after evaluation (some bindings can be modified or added), and the object ω' is simply called the object *returned* by the evaluator working on ω in the environment E; still more simply, ω' is the *result* of the evaluation of ω, the underlying environment being implicit.

4. Sometimes the evaluator work does not terminate, for example if there is an infinite loop in the program. In such a case the mathematical translation is $\varepsilon(E, \omega) = ?$. In general, according to the Post theorem, we are unable to *guess* such a result.

Let us look at the simple example:

```
> (setf 2-x-list
     #'(lambda (list)
          (append list list)))  ==>
#<function 1 ...>
> (funcall 2-x-list '(a b c))  ==>
(A B C A B C)
```

The first statement *returns* a functional object *and* modifies the current environment: a new binding from the symbol `2-x-list` towards the function just created is installed. The second statement makes this functional object work on the list `(a b c)` through the binding just installed.

Another way to obtain the same result, more usual in Common Lisp, is the following:

```
> (defun 2-x-list (list)
     (append list list))  ==>
2-X-LIST
> (2-x-list '(a b c))  ==>
(A B C A B C)
```

The first statement (`defun` = <u>def</u>ine <u>fun</u>ction) assigns a functional object to the *functional* slot of the symbol `2-x-list`; in this way the symbol can be directly used at the very beginning of the list `(2-x-list ...)` without using `funcall`, which is compulsory if the functional object is assigned to the *value* slot of the symbol; see [49] for details. We understand the reader is a little irritated at considering so technical considerations, but they cannot be avoided: in the first sections it was didactically more appropriate to consider only the value slots of symbols; for actual programming, using the functional slots is much more convenient and because we intend to base exposition on the *actual* EAT program, we are obliged to speak of these esoteric notions.

A stranger but equivalent method is the following:

```
> ((lambda (list)
     (append list list))
   '(a b c))  ==>
(A B C A B C)
```

A unique statement makes the functional object and immediately uses it. The first method, using a binding installed in the environment is the usual one, but it is easy to proof any set of statements can always be put together in a unique statement, with the same result. In this way we can theoretically ignore the role of the environments and modelize our machine as a simpler function $\varepsilon : \mathcal{U} \to \mathcal{U} \coprod \{?\}$. We will no longer consider the questions of environment.

Most of our computability results have the following form:

Proposition 10 — *An algorithm α can be constructed satisfying the following property: if the object ω_1 (resp. ω_2) satisfies the property P_1 (resp. P_2), then the object $\alpha(\omega_1, \omega_2)$ satisfies the property P.*

The number of arguments (here 2) which the function α may work on is usually constant but sometimes may be also variable. For example the previous Lisp example could be formalized as follows:

Proposition 11 — *An algorithm α can be constructed such that, if λ is a list, then the result $\alpha(\lambda)$ is the list λ concatenated with itself.*

A more serious example of such a result, concerning the loop space examples of section 7 can be now stated in this way:

Proposition 12 — *An algorithm λ can be constructed such that, if X is an effective or locally effective reduced simplicial set, then $\lambda(X)$ is a locally effective version of the loop space GX.*

Such a statement needs a preliminary work. On one hand the object types *effective simplicial sets* and *locally effective simplicial sets* must be defined. On the other hand there must be explained the reason why producing the result object GX amounts to essentially constructing a few functional slots from various ingredients, in particular the functional slots of X; in other words it is essentially a question of functional programming. The situation is even a little better than in traditional mathematics: a *complete* formal construction is often available as an *actual* Lisp program[27].

[27] A referee of a previous version of this paper did not hesitate to judge: “The paper's claim to be *algorithmic* is a joke" (sic); however the necessary information was given to reach the *complete* actual program, a program already demonstrated in numerous mathematical departments, a program where no error has yet been discovered.

8.3 The ss object type.

Definition 13 — An `ss` machine object $X \in \mathcal{U}$ is a 4-tuple $X = (\tau, \chi, \beta, \partial)$ where:

1. The τ component is a function $\tau : \mathbb{N} \times \mathcal{U} \to \{\mathtt{nil}, \mathtt{t}\}$ describing the sets of simplices of X: a machine object σ is a non-degenerate n-simplex of X if and only if $\tau(n, \sigma) = \mathtt{t}$.[28]

2. The χ component is a function $\chi : \mathcal{U}_\chi \to \{\mathtt{nil}, \mathtt{t}\}$; the source set $\mathcal{U}_\chi$ is the set of pairs (σ_1, σ_2) where both components are simplices having the same dimension in X[29]; the result is the symbol `t` if and only if the simplices σ_1 and σ_2 are *equal*: one simplex can be modelized as different machine objects.

3. The β component can be the keyword `:locally-effective` or a function $\beta : \mathbb{N} \to \mathcal{U}$; in the last case, the value $\beta(n)$ is the (necessarily finite) list of non-degenerate simplices in dimension n (only one representant for every "mathematical" simplex): the simplicial set is therefore finite in each dimension. In the first case, this information is unavailable and the `ss` object X is then called *locally effective* and can be "highly" infinite; in the second case, the `ss` object X is called *effective*.

4. The ∂ component is a function $\partial : \mathcal{U}_\partial \to \mathcal{U}$; the source $\mathcal{U}_\partial$ is the set of triples (i, n, σ) where n is a positive integer, the integer i satisfies $0 \leq i \leq n$ and σ is an n-simplex of X. The image $\partial(i, n, \sigma) = (\eta, \sigma')$ is a pair; the first component η is a multiple degeneracy operator (possibly the identity: no degeneracy at all) and σ' is some nondegenerate simplex; this pair is the canonical expression of the corresponding face: $\partial_i(\sigma) = \eta\sigma'$.

5. Finally these components must verify the obvious coherence conditions to define a simplicial set; for example the ∂ component must satisfy the classical commutation relations; if two simplices are equal, the faces of same index must be equal too, and so on.

Nothing new in this definition with respect to what was explained in Section 6: the chosen language is only closer to the traditional mathematical one. If the Lisp machine is precisely defined as a mathematical object like in [49], a definition like above allows one to state that some object X *is* an effective simplicial set or a locally effective simplicial set: the object X is an element of the *mathematical* set $\mathcal{U}$ satisfying some properties with respect to the Lisp evaluator ε.

[28] We recall this slot is in fact never used in the calculations we are interested in, so that it is omitted in the EAT program.

[29] *Several* dimensions are possible for the same pair, but the result must not depend on the considered dimension.

8.4 The cc object type.

Definition 14 — We can define in the same way the `cc` object type. A `cc` object C is a 4-tuple $C = (\tau, \chi, \beta, d)$ where:

1. The τ component is a function $\tau : \mathbb{N} \times \mathcal{U} \to \{\mathtt{nil}, \mathtt{t}\}$ describing the sets of generators of the chain complex C: a chain group C_n is a free $\mathbb{Z}$-module provided with an explicit basis, namely the set of machine objects x satisfying the condition $\tau(n, x) = \mathtt{t}$.[30]

2. The χ component is a function $\chi : \mathcal{U}_\chi \to \{\mathtt{nil}, \mathtt{t}\}$; the source set $\mathcal{U}_\chi$ is the set of pairs (x_1, x_2) where both components are generators having the same degree in C[31]; the result is the symbol `t` if and only if the generators x_1 and x_2 are *equal*: one generator can be modelized as different machine objects.

3. The β component can be the keyword `:locally-effective` or a function $\beta : \mathbb{N} \to \mathcal{U}$; in the last case, the value $\beta(n)$ is the (necessarily finite) list of generators in degree n (only one representant for every "mathematical" generator): the chain complex is therefore of finite type in each dimension. In the first case, this information is unavailable and the `cc` object C is then called *locally effective* and can be "highly" infinite; in the second case, the `cc` object C is called *effective*.

4. The d component is a function $d : \mathcal{U}_d \to \mathcal{U}$; the source $\mathcal{U}_d$ is the set of pairs (n, x) where n is an integer and x is a generator of C_n. The image $d(n, x) = \sum \alpha_i x_i$ is a finite $\mathbb{Z}$-combination of generators of C_{n-1} expressed in an appropriate way.

5. Finally these components must verify the obvious coherence conditions to define a chain complex; for example $d \circ d = 0$; if two generators are equal, their boundaries must be equal too, and so on.

We hope the reader now understands analogous definitions may be given for the classical types of structures used in algebraic topology. For example we shall frequently use some *coalgebras*; in this framework, a coalgebra will be a `cc` object completed with a further component Δ implementing in an obvious way the comultiplication; the function Δ must be able to work on pairs (n, x) where x is a generator of C_n, returning a description of the object $\Delta(x)$, the coproduct of the generator x.

8.5 Morphism objects.

Let us suppose two `cc` objects C_1 and C_2 are installed in our machine.

[30] Cf footnote[28].
[31] Cf footnote[29].

Definition 15 — A *morphism object* $f : C_1 \to C_2$ is a function $f : \mathcal{U}_f \to \mathcal{U}$ where $\mathcal{U}_f = \mathcal{U}_{d_1}$, that is, the set of pairs (n, x) where x is a generator of $C_{1,n}$. Then $f(x, n) = \sum \alpha_i y_i$ is a finite $\mathbb{Z}$-combination of generators of $C_{2,n}$. Of course a few coherence conditions must be satisfied; for example $d_2 \circ f = f \circ d_1$; if two generators are equal, their images by f must be equal too, and so on.

It is easy to define in the same way homotopy operators between chain complexes, coalgebra morphisms, and so on.

8.6 Functor objects.

The classical "reasonable" functors can be implemented. A typical example follows.

Proposition 16 — *An object $\otimes$ can be constructed. This object is a function:*

$$\otimes : \mathcal{CC} \times \mathcal{CC} \to \mathcal{CC}$$

able to work on pairs (C_1, C_2) and returning a new `cc` *object C modelizing the tensor product of chain complexes $C_1 \otimes C_2$.*

PROOF. A generator of C is implemented as a pair of generators of C_1 and C_2; the generator types of C_1 and C_2 therefore allow us to define the corresponding generator type for the chain complex C. The same for the other components of C: constructing them amounts to defining new functions using the functional slots of C_1 and C_2; it is a simple matter of *functional programming*. ■

So that understanding our main theorem 9 amounts to defining the notions of simplicial sets *with effective homology*; it is the subject of the next section.

9 The general organization revisited.

9.1 Basic definitions.

The notions of *effective* and *locally effective* objects (simplicial sets and chain complexes mainly) are now precisely defined. We may reconsider the definitions of Section 2 in the light of these definitions.

Definition 17 — (cf. Definition 1) A *reduction* is a 5-tuple $\rho = (\widehat{C}, C, f, g, h)$:

$$\begin{array}{c} \widehat{C} \xrightarrow{h} \widehat{C} \\ f \downarrow\uparrow g \quad\;\; \\ C \quad\;\; \end{array}$$

where $\widehat{C}$ and C are locally effective chain complexes, f and g are chain complex morphisms, h is a homotopy operator of degree 1; these data must satisfy the following relations:

1) $fg = 1_C$;
2) $fh = 0$;
3) $hg = 0$;
4) $hh = 0$;
5) $1_{\widehat{C}} - gf = hd + dh$.

The reduction ρ is called a (LE, E) reduction if the big chain complex $\widehat{C}$ is locally effective and the small one C is *effective*. Otherwise the reduction is at least of type (LE, LE). If the reduction is of type (LE, E), then the homology groups of the small chain complex C are computable, so that the reduction can be understood as a *complete* description of the homology of the big chain complex $\widehat{C}$: giving a homology class of $\widehat{C}$ amounts to giving a cycle of C, and a representant in $\widehat{C}$ can be reached through the g-component of the reduction ρ. If a cycle of $\widehat{C}$ is given, an algorithm can determine whether this cycle is a boundary, and if so a preimage by the differential can be computed.

Definition 18 — A *homotopy equivalence* between two (locally effective) chain complexes C_1 and C_2 is a pair of reductions:

$$\begin{array}{ccc} & \hat{C} & \\ \rho_1 \swarrow & & \searrow \rho_2 \\ C_1 & & C_2 \end{array}$$

where in general the chain complex $\widehat{C}$ is assumed only locally effective.

A homotopy equivalence is of type (LE, LE, E) if the bottom right-hand chain complex C_2 is effective. Others combinations can be considered but in this paper only the (LE, LE, LE) and (LE, LE, E) situations happen. Again a (LE, LE, E) homotopy equivalence can be considered as a complete descritpion of the homology of the bottom left-hand chain complex C_1.

Definition 19 — An *object with effective homology* is a 4-tuple (X, C, EC, ε) where:

1) X is a locally effective object;
2) C is the locally effective chain complex canonically associated to X;
3) EC is an effective chain complex;
4) ε is a (LE, LE, E) homotopy equivalence between C and EC.

In this paper, the object X is a simplicial set or a chain complex (in this case $C = X$), sometimes with additional structure; for example a simplicial set could be a simplicial group, a chain complex could be a differential coalgebra, etc.

The main results we obtain in this paper are the constructive versions of the classical spectral sequences, due to Serre and Eilenberg-Moore. The following object types will be used:

1. The type of n-reduced simplicial sets with effective homology is denoted by $\mathcal{SS}^n_{EH}$; a simplicial set is n-reduced if it has one vertex and no other non-degenerate simplex in dimension $\leq n$; it is in particular n-connected;

2. The type of locally effective simplicial groups is denoted by $\mathcal{SG}$;

3. The type of n-reduced simplicial groups with effective homology is denoted by $\mathcal{SG}^n_{EH}$

4. A (simplicial) *fibration* is a 5-tuple $\Phi = (B, F, G, \tau, E)$ where F and B are locally effective — *from now on this qualifier is implicit* — simplicial sets, G is a simplicial group acting on F, τ is a torsion operator $\tau : B \to G$ defining a fibration of $E = B \times_\tau F$ over the base space B with fiber$F$. See details in Section 13. If $F = G$ and if the action of G on itself is the canonical one, then the fibration is *principal.* The type of fibrations is denoted by $\mathcal{F}$.

9.2 Results.

Theorem 20 — *An algorithm* $\mathbf{SERRE}_{EH}$ *can be constructed:*

$$\mathbf{SERRE}_{EH} : [\mathcal{F} \times \mathcal{SS}^1_{EH} \times \mathcal{SS}^0_{EH}]_\chi \longrightarrow \mathcal{SS}^0_{EH}$$

where $[\ldots]_\chi$ *is the set of* coherent *triples* (Φ, B_{EH}, F_{EH})*, that is those triples such that the underlying simplicial set of* B_{EH} *is the base space of* Φ *and the underlying simplicial set of* F_{EH} *is the fiber space. The output of the algorithm working on coherent data is a version with effective homology of the total space, in other words a (complete) description of its* $\mathbb{Z}$*-homology is available.*

This is the version with effective homology of the Serre spectral sequence.

Theorem 21 — *An algorithm* $\mathbf{ELMR}_{EH}$ *can be constructed:*

$$\mathbf{ELMR}_{EH} : [\mathcal{F} \times \mathcal{SS}^1_{EH} \times \mathcal{SS}^0_{EH}]_\chi \longrightarrow \mathcal{SS}^0_{EH}$$

where $[\ldots]_\chi$ *is the set of* coherent *triples* (Φ, B_{EH}, E_{EH})*, that is those triples such that the underlying simplicial set of* B_{EH} *is the base space of* Φ *and the underlying simplicial set of* E_{EH} *is the total space. The output of the algorithm working on coherent data is a version with effective homology of the fiber space.*

Just as for the ordinary Eilenberg-Moore spectral sequence, a more general statement can be proved for the total space of induced fibrations.

Combining this with Theorem 9, we obtain as a corollary:

Theorem 22 — *An algorithm* $\mathbf{LPSP}_{EH}$ *can be constructed:*

$$\mathbf{LPSP}_{EH} : [\mathbb{N} \times \mathcal{SS}^0_{EH}]_\chi \longrightarrow \mathcal{SS}^0_{EH}$$

where $[\ldots]_\chi$ *is the set of pairs* (n, X_{EH}) *such that the underlying simplicial set* X *is at least* n*-reduced; the output is* $(\Omega^n X)_{EH}$*, a version with effective homology of tne* n*-th loop space* $\Omega^n X$ *(more precisely of its Kan version). The result is void if* $n = 0$*, (strictly) contains the usual Adams* Cobar *construction for* $n = 1$*, and is a solution of the problem of iterating the Cobar construction if* $n > 1$*.*

Please, compare the solution here proposed with the other ones ([8], [9], [14], [46], [40], [47]). When all these solutions will be understood, then you will be competent to decide if the present work is mathematics or negligible [42] technical programming of available algorithms.

The statement of the previous theorem explains the algorithm $\mathbf{LPSP}_{EH}$ *can* be constructed; in fact this algorithm *is* constructed, see Section 9.3; it is the EAT program to be considered as a prototype application of the methods developped in this paper. Futhermore, when this paper is prepared, the present solution is the only one which has led to concrete programming work.

The other ordinary Eilenberg-Moore spectral sequence "computes" the homology of the base space.

Theorem 23 — *An algorithm* $\mathbf{ELMR}'_{EH}$ *can be constructed:*

$$\mathbf{ELMR}'_{EH} : [\mathcal{F} \times \mathcal{SS}^0_{EH} \times \mathcal{SS}^0_{EH} \times \mathcal{SG}^0_{EH}]_\chi \longrightarrow \mathcal{SS}^1_{EH}$$

where $[\ldots]_\chi$ *is the set of coherent 4-tuples* $(\Phi, F_{EH}, E_{EH}, G_{EH})$ *and the result a version with effective homology of the base space.*

Combining this result with the classical results about $K(\pi, 1)$, we obtain the ideal result about Eilenberg-MacLane spaces.

Theorem 24 — *An algorithm* $\mathbf{CARTAN}_{EH}$ *can be constructed:*

$$\mathbf{CARTAN}_{EH} : \mathcal{FAG} \times \mathbb{N} \longrightarrow \mathcal{SS}_{EH}$$

where $\mathcal{FAG}$ *is the type of abelian groups of finite type and the output of the algorithm working on the pair* (π, n) *is* $K(\pi, n)_{EH}$*, a version with effective homology of* $K(\pi, n)$*.*

How to compare this result with famous Cartan's paper [15]? On one hand Cartan gave a simple algorithm which very quickly gives the *ordinary* homology groups of $K(\pi, n)$. Our algorithm certainly cannot be used for $n = 20$ for example, because of time and space complexity. On the contrary, it is elementary to use Cartan's algorithm to compute the $\mathbb{Z}$-homology of $K(\pi, 20)$, and the algorithm will be quite efficient. But without further informations, you are unable to use the *ordinary* $\mathbb{Z}$-homology of $K(\pi, n)$ to construct the Postnikov or Whitehead tower of a simply connected space X. The ordinary $\mathbb{Z}$-homologies of a base space and a fiber space do not determine in general the $\mathbb{Z}$-homology of the total space of a given fibration. The missing informations are precisely inside $K(\pi, n)_{EH}$, our version with effective homology of $K(\pi, n)$. In this way we obtain the computability of the homotopy groups of simply connected spaces.

Theorem 25 — *An algorithm* $\mathbf{WHTH}_{EH}$ *can be constructed:*

$$\mathbf{WHTH}_{EH} : \mathcal{SS}^1_{EH} \times \mathbb{N} \longrightarrow \mathcal{SS}^1_{EH}$$

where the output of the algorithm working on the pair (X, n) *is* X^n_{EH}*, a version with effective homology of the* n*-th floor of the Whitehead tower, that is, the space* X *with the* π_i *canceled for* $i < n$*.*

In fact the $(n+1)$-th floor is the total space of a fibration the base space of which is the n-th floor and the fiber space is $K(H_n X_n, n-1)$. Because the *effective* homology of X_n is known, you can construct this fibration; because *effective* homologies of base and fibre spaces are available, using Theorem 20, you can determine the effective homology of X_{n+1} and iterate. Again compare with the other solutions [11] and [40]: we have a conceptually simple solution to determine the first homotopy groups of an arbitrary 1-reduced simplicial set with effective homology; the computations already done for iterated loop spaces indicate a significant concrete work is feasible.

Corollary 26 — *An algorithm* **HMT-GRP** *can be constructed:*

$$\textbf{HMT-GRP} : \mathcal{SS}^1_{EH} \times \mathbb{N} \longrightarrow \mathcal{FAG}$$

computing the homotopy groups of 1-reduced simplicial sets with effective homology.

An analogous work can be undertaken for the Postnikov tower to determine the Postnikov "invariants".

9.3 EAT implementation.

9.3.1 Reductions

The EAT program defines and uses a `rdc` (reduction) structure type allowing the user to conveniently handle reductions. For example let us consider the Eilenberg-Zilber reduction, detailed in Section 14. The Eilenberg-Zilber theorem is a process constructing a reduction $\rho = (f, g, h)$ for every pair (X, Y) of simplicial sets; the chain map f (resp. g, h) is the *Alexander-Whitney* (resp. *Eilenberg-MacLane*, *Shih*) operator. The EAT program provides a function `eilenberg-zilber` to construct such a reduction. Let us apply this function in the case $X = Y = \Delta^\infty$, two copies of the simplex spanned by the natural numbers, see page 34; this simplicial set was then constructed from scratch but it is in fact predefined in the EAT program and can be reached through the symbol `*delta*`:

```
> *delta*  ==>
[SS-0]
> (gdl *delta* 2 4 '(0 1 2 3 4))  ==>
<ASM * (0 1 3 4)>
```

The reduction ρ corresponding to two copies of Δ^∞ can be constructed as follows:

```
> (setf dd-rdc (eilenberg-zilber *delta* *delta*))  ==>
[RDC-0]
> (setf cpr-cc (rdc-tcc dd-rdc))
[CC-10]
> (setf tpr-cc (rdc-bcc dd-rdc))
[CC-9]
> (setf aw (rdc-f dd-rdc))  ==>
[MRP-11]
> (setf eml (rdc-g dd-rdc))  ==>
[MRP-12]
> (setf shih (rdc-h dd-rdc))  ==>
[MRP-13]
```

This particular Eilenberg-Zilber reduction is the object `[RDC-0]` with five main slots:

1. `tcc` (top chain complex) = `[CC-10]`, that is, $C_*(\Delta^\infty \times \Delta^\infty)$;
2. `bcc` (bottom chain complex) = `[CC-9]`, that is, $C_*(\Delta^\infty) \otimes C_*(\Delta^\infty)$;
3. `f`, the Alexander-Whitney map $C_*(\Delta^\infty \times \Delta^\infty) \xrightarrow{f} C_*(\Delta^\infty) \otimes C_*(\Delta^\infty)$;
4. `g`, the Eilenberg-MacLane map $C_*(\Delta^\infty) \otimes C_*(\Delta^\infty) \xrightarrow{g} C_*(\Delta^\infty \times \Delta^\infty)$;
5. `h`, a homotopy between $1_{[\mathtt{CC-10}]}$ and $g \circ f$.

The traditional definitions of these maps can be verified:

```
> (? aw 2 (cpr nil '(0 1 2) nil '(100 101 102)))  ==>
-----------------------------------------{CMB 2}
<MNM 1 * <TPR (0 1 2) (102)>>
<MNM 1 * <TPR (0 1) (101 102)>>
<MNM 1 * <TPR (0) (100 101 102)>>
------------------------------------------------

> (? eml 3 (tpr 1 '(0 1) 2 '(100 101 102)))  ==>
-----------------------------------------{CMB 3}
<MNM 1 * <CPR 2-1 (0 1) 0 (100 101 102)>>
<MNM -1 * <CPR 2-0 (0 1) 1 (100 101 102)>>
<MNM 1 * <CPR 1-0 (0 1) 2 (100 101 102)>>
------------------------------------------------

> (? shih 1 (cpr nil '(0 1) nil '(100 101)))
-----------------------------------------{CMB 2}
<MNM -1 * <CPR 0 (0 1) 1 (100 101)>>
------------------------------------------------
```

A (geometric) simplex of the cartesian product of two $\mathcal{SS}$ objects is constructed by the function `cpr` (cartesian product), using as arguments:

1. A multi-degeneracy operator as an integer list, `nil` meaning an empty list, that is, no degeneracy;
2. A geometric simplex of the first factor;
3. Another multi-degeneracy operator;
4. A geometric simplex of the second factor.

For example, the generator on which `aw` works above is the triangle of $\Delta^\infty \times \Delta^\infty$ whose projections are respectively `(0 1 2)` and `(100 101 102)` (non-degenerate); the image of the Shih map, working on the diagonal of a square $\alpha_1 \times \alpha_2$, the projection α_1 (resp. α_2) being the 1-simplex `(0 1)` (resp. `(100 101)`), is the top left-hand triangle of this square; the first projection of this triangle is $\eta_0\alpha_1$ and the second one is $\eta_1\alpha_2$.

In the same way, to construct a generator of the tensor product of two chain complexes, you use the `tpr` function, needing also four arguments:

1. The degree of the generator of the first factor, for example `1`;
2. A generator of the first factor `(0 1)`;
3. The degree of the second factor, `2`;
4. A generator of the second factor `(100 101 102)`.

and in this way the generator `(tpr 1 '(0 1) 2 '(100 101 102))`, representing essentially a prism, product of an edge by a triangle, can be used for the Eilenberg-MacLane map, returning the decomposition of the prism into three 3-simplices. Note that when a tensor product is *returned*, the degrees of factors are not displayed, look for example at the result of the Alexander-Whitney map.

9.3.2 Homotopy equivalences.

A homotopy equivalence is simply a pair of reductions; the `heq` structure type is defined in the EAT program and a `heq` object has nine main slots, the bottom left-hand chain complex (`lbcc`), the top one (`tcc`), the bottom right-hand one (`rbcc`), the left f (`lf`), the left g (`lg`) and the left h (`lh`), and also the components of the right reduction (`rf`, `rg` and `rh`). Let us consider the following exercise: construct the version with effective homology of $\Omega^2 S^3$, extract the homotopy equivalence, compute the 3-basis of the right bottom chain complex, and verify the composition ${}^{\ell}f \circ {}^{r}g$ is compatible with the differentials.

Constructing the version with effective homology of $\Omega^2 S^3$ needs the `loop-space-eh` function, see details in Section 9.3.3.

```
> (setf o2s3eh (loop-space-eh (ess-sseh (sphere 3)) 2))  ==>
[SS-EH 8]
```

The last argument 2 for `loop-space-eh` asks for the *second* loop space. The result is the object `[SS-EH 8]`, an *oeh* object, the main slot of which is the `heq` slot which can be extracted:

```
> (setf o2s3eh-heq (oeh-heq o2s3eh))  ==>
[HEQ-10]
```

Then the slots of this `heq` object we are interested in can be also extracted:

```
> (setf rbcc (heq-rbcc o2s3eh-heq))  ==>
[CC-79]
> (setf lbcc (heq-lbcc o2s3eh-heq))  ==>
[CC-80]
> (setf rbcc-d (cc-d rbcc))  ==>
[MRP-274]
> (setf lbcc-d (cc-d lbcc))  ==>
[MRP-281]
> (setf rg (heq-rg o2s3eh-heq))  ==>
[MRP-276]
> (setf lf (heq-lf o2s3eh-heq))  ==>
[MRP-323]
```

The basis (in degree 3) of the bottom left-hand chain complex is given by the `cbs` function, and we extract from it the first generator:

```
> (setf basis (cbs rbcc 3))  ==>
(<<ALOOP (<GGNR 0 GNR-Z> ...
[... Lines deleted ...]
> (setf generator (first basis))  ==>
<<ALOOP (<GGNR 0 GNR-Z>
         <GGNR 4 <<ALOOP (<GGNR 0 GNR-Z>
                          <GGNR 3 <S3>>
                          <GGNR 3 <S3>>)>>>)>>
```

Both necessary combinations of morphisms and differentials are to be computed:

```
> (setf comp-1 (cmp-mrp lbcc-d (cmp-mrp lf rg)))  ==>
[MRP-329]
> (setf comp-2 (cmp-mrp (cmp-mrp lf rg) rbcc-d))  ==>
[MRP-331]
```

Applying them to our generator gives two combinations:

```
> (setf cmb-1 (? comp-1 3 generator))  ==>
---------------------------------------------------{CMB 2}
<MNM -2 * <<LOOP ( <PWR 1 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>
```

```
                  <PWR 2 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>)>>>
<MNM 2 * <<LOOP ( <PWR 2 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>
                 <PWR 1 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>)>>>
-----------------------------------------------------------
> (setf cmb-2 (? comp-2 3 generator))  ==>
----------------------------------------------------{CMB 2}
<MNM -2 * <<LOOP ( <PWR 1 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>
                  <PWR 2 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>)>>>
<MNM 2 * <<LOOP ( <PWR 2 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>
                 <PWR 1 <<LOOP ( <PWR * <S3> ** 1>)>> ** 1>)>>>
-----------------------------------------------------------
```

and the simplest way to compare them is to ask for a subtraction:

```
> (sbt-cmb-from-cmb lbcc cmb-1 cmb-2)  ==>
----------------------------------------------------{CMB 2}
-----------------------------------------------------------
```

9.3.3 Objects with effective homology.

We reconsider $\Omega^2 S^3$, this time constructed step by step (a new EAT session is restarted). We construct the 3-sphere and transform it into a (trivial) object with effective homology, the function `ess-sseh` (effective simplicial set to simplicial set with effective homology) does this work; the simplicial set is finite so that the homotopy equivalence is trivial, but anyway a normal form containing such a homotopy equivalence must be firstly constructed:

```
> (setf s3 (sphere 3))  ==>
[SS-4]
> (setf s3eh (ess-sseh s3))  ==>
[SS-EH 0]
```

Then the EAT function `loop-space-eh` can be applied twice, which constructs versions with effective homology of the first and second loop space:

```
> (setf os3eh (loop-space-eh s3eh))  ==>
[SS-EH 4]
> (setf o2s3eh (loop-space-eh os3eh))  ==>
[SS-EH 8]
```

The last object contains *the* chain complex of the second loop-space which is locally effective:

```
> (oeh-cc o2s3eh)  ==>
[CC-80]
> (cbs (oeh-cc o2s3eh) 3)  ==>
;; Error: The chain complex [CC-80] is locally-effective.
```

It is the chain complex #80 and because it is locally effective, asking for its basis in degree 3 generates an error. But the same object with effective homology contains also an *effective* chain complex, the basis of which in some degree may be determined:

```
> (oeh-ecc o2s3eh)  ==>
[CC-79]
> (cbs (oeh-ecc o2s3eh) 3)  ==>
(<<ALOOP (<GGNR 0 GNR-Z> <GGNR 4 #>)>>
 <<ALOOP (<GGNR 0 GNR-Z> <GGNR 2 #> <GGNR 2 #> <GGNR 2 #>)>>)
> (cc-homology (oeh-ecc o2s3eh) 3)
Computing boundary-matrix in dimension 3.
[... Lines deleted ...]
Homology in dimension 3 :
Component Z/2Z
---done---
```

The basis is a set of *algebraic loops*, that is some elements of our *bi*-Cobar construction coming from C_*S^3. The result obtained can be easily verified, for example by using the results of the paper [10]. But let us decide now to paste a three disk to the first loop space before constructing the second loop space, using an attachment map of degree 2 from S^2 to ΩS^3; the space denoted by $\Omega S^3 \cup_2 D^3$ is to be constructed. The EAT program may build this space as a simplicial set with effective homology; firstly the “fundamental" simplex of ΩS^3 is located through the symbol **fund-simp** and also the null 2-simplex through the symbol **null-simp**:

```
> (setf fund-simp (asm nil (loop3 nil '<S3> 1)))  ==>
<ASM * <<LOOP ( <PWR * <S3> ** 1>)>>>
> (setf null-simp (null-asm-loop 2))  ==>
<ASM 1-0 <<LOOP *>>>
```

Then we can paste a three simplex to our loop space *with effective homology*, asking for a new simplex having as faces of index 0 and 2 the fundamental simplex, as faces of index 1 and 3 the base point:

```
> (setf dos3eh
      (disk-paste-eh os3eh 3
          (list fund-simp null-simp fund-simp null-simp) :new '<D3>))  ==>
[SS-EH 9]
```

The space so obtained is also with effective homology, so that for example the second homology group can be verified being the right one:

```
> (homology dos3eh 2)  ==>
Computing boundary-matrix in dimension 2.
[... Lines deleted ...]
Homology in dimension 2 :
Component Z/2Z
---done---
```

Our simplicial set `dos3eh` is again a simplicial set with effective homology, so that its loop space (with...) can be constructed:

```
> (setf odos3eh (loop-space-eh dos3eh))  ==>
[SS-EH 13]
```

and the first homology groups of $\Omega(\Omega S^3 \cup_2 D^3)$ are reachable, for example:

```
> (homology odos3eh 5)  ==>
Computing boundary-matrix in dimension 5.
Rank of the source-module : 14.
[... Lines deleted ...]
End of computing.
Computing boundary-matrix in dimension 6.
Rank of the source-module : 26.
[... Lines deleted ...]
End of computing.

Homology in dimension 5 :

Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z

---done---
```

that is, $H_5(\Omega(\Omega S^3 \cup_2 D^3)) = \mathbb{Z}_2^6 \oplus \mathbb{Z}$; the severe [14, p.545] difficulties in computing the right differential have been overcome.

10 The Basic Perturbation Lemma [45] [13].

Let $\rho = (\widehat{C}, C, f, g, h)$ be a reduction. This reduction can be considered as a description of the homology of the "big" chain complex $\widehat{C}$ through the "small" one C. Frequently, taking account of the ordinary vertical diagram for a reduction (see Definition 1 or 17), we name the big chain complex $\widehat{C}$ the *top* chain complex; in the same way, C is called the *bottom* chain complex. The chain complex $\widehat{C}$ could be *locally effective*, hence its homology is unreachable, whereas C could be *effective*, with computable homology groups.

The *Basic Perturbation Lemma* works as follows: if the differential of the big chain complex $\widehat{C}$ is *softly* modified, then coherent modifications can be applied to the other components of the reduction in such a way a new reduction is obtained; the homology of the *new* big chain complex is again described through the homology of the new small one. The basic perturbation lemma is a little like an implicit function theorem; Appendix 23 shows it is in fact a simple generalization of the well-known fact that a triangular matrix is equivalent to a diagonal one.

This "lemma" is the heart of our algorithmic versions of the main spectral sequences (Serre, Eilenberg-Moore, etc.). The ordinary versions of these spectral sequences are nothing but what remains available when functional programming is not.

Definition 27 — A *perturbation* of the differential d of a chain complex C is an operator $\delta : C \to C$ of degree -1 such that the sum $d + \delta$ is again a differential: the relation $(d+\delta) \circ (d+\delta) = 0$ is satisfied.

Definition 28 — A perturbation $\widehat{\delta}$ of the differential $\widehat{d}$ of the top chain complex $\widehat{C}$ of a reduction $\rho = (\widehat{C}, C, f, g, h)$ satisfies the condition of *local nilpotency* if for every x in $\widehat{C}$, there exists an integer n satisfying the relation $(h \circ \widehat{\delta})^n(x) = 0$; this relation is equivalent to the following: for every x in $\widehat{C}$, there exists an integer n satisfying the relation $(\widehat{\delta} \circ h)^n(x) = 0$.

Theorem 29 — **(Basic Perturbation Lemma [45] [13])** *Let $\mathcal{R}$ be the type of reductions. An algorithm can be constructed:*

$$\mathbf{bpl} : [\mathcal{R} \times \mathcal{P}]_\chi \longrightarrow \mathcal{R}$$

where $[\ldots]_\chi$ is the set of coherent pairs $(\rho = (\widehat{C}, C, f, g, h), \widehat{\delta})$ (that is, $\widehat{\delta}$ is a perturbation of the differential $\widehat{d}$ of $\widehat{C}$ satisfying the condition of local nilpotency*); the output is a reduction $\rho' = (\widehat{C'}, C', f', g', h')$ where the new top chain complex $\widehat{C'}$ is the old one $\widehat{C}$ provided with the new differential $\widehat{d'} = \widehat{d} + \widehat{\delta}$; in particular the new bottom chain complex C' is the old one with a new differential $d + \delta$, where d is the old differential of C and δ is a perturbation determined by the algorithm* **bpl***; the same for the new maps f', g' and h'.*

It is important to note that the graded modules underlying the top and bottom chain complexes are let unchanged in the perturbation process; only the various maps are perturbed.

Taking account of the importance of this "lemma", which would be better called the *Fundamental Theorem of Homological Algebra*, we give a demonstration which is nothing but a detailed rewriting of [13] (see also [7]).

PROOF. Because of the local nilpotency condition, the following series have, for each element which they work on, only a finite number of non-null terms and their sums are defined:

$$\phi = \sum_{i=0}^{\infty} (-1)^i (h\widehat{\delta})^i; \qquad \psi = \sum_{i=0}^{\infty} (-1)^i (\widehat{\delta} h)^i.$$

The operators ϕ and ψ have degree 0 and trivially satisfy a few relations; *these relations are the only ones that are from now on utilized*:

$\phi h = h\psi$;
$\widehat{\delta}\phi = \psi\widehat{\delta}$;
$\phi = 1 - h\widehat{\delta}\phi = 1 - \phi h\widehat{\delta} = 1 - h\psi\widehat{\delta}$;
$\psi = 1 - \widehat{\delta}h\psi = 1 - \psi\widehat{\delta}h = 1 - \widehat{\delta}\phi h.$

The reduction $\rho' = (\widehat{C'}, C', f', g', h')$ to be constructed is then simply defined by:

$\widehat{d'} = \widehat{d} + \widehat{\delta}$, differential of $\widehat{C}'$;
$d' = d + \delta$ is the differential of C' where $\delta = f\widehat{\delta}\phi g = f\psi\widehat{\delta}g$;
$f' = f\psi$;
$g' = \phi g$;
$h' = \phi h = h\psi$.

Lemma 30 — *Let C be a chain complex with the differential d and let h be an operator on C of degree $+1$, satisfying the relations:*

$hh = 0$;
$hdh = h$.

Then $D = dh + hd$ is a projector which splits the chain complex C into the direct sum of chain complexes $\ker(D) \oplus \operatorname{im}(D)$ where the second one is acyclic. In other words, $(C, \ker D, 1 - D, 1, h)$ is a reduction.

Proof. The operator D is a projector, because of the computation: $D^2 = (dh + hd)^2 = dhdh + hdhd = dh + hd = D$ (because $hh = 0$ and $dd = 0$). Furthermore D and therefore also $1 - D$ are chain complex morphisms : $d(dh + hd) = dhd = (dh + hd)d$ (because $dd = 0$). The lemma is proved. ■

In the theorem, the operator h does satisfy these relations with respect to $\widehat{d}$, because $hh = 0$ is explicitly required among the reduction properties and $h\widehat{d}h = (1 - \widehat{d}h - gf)h = h$ (because $hh = 0$ et $fh = 0$). The projection $D = \widehat{d}h + h\widehat{d}$ is also the difference $1 - gf$, and therefore the complementary projection $1 - D$ is the composition gf.

The new homotopy operator h' has been defined by $h' = \phi h = h\psi$. Firstly, we naturally obtain from the definition of h' the definitions of f', g' et δ.

The new operator h' satisfies also the relations $h'h' = 0$ and $h'\widehat{d'}h' = h'$. In fact $h'h' = \phi hh\psi = 0$ and $h'\widehat{d'}h' = \phi h(\widehat{d}+\widehat{\delta})h\psi = \phi h\widehat{d}h\psi + \phi h\widehat{\delta}h\psi = \phi h\psi + \phi h(1-\psi) = \phi h = h'$ (because $\widehat{\delta}h\psi = 1 - \psi$).

We then obtain from the lemma the fact that $D' = \widehat{d'}h' + h'\widehat{d'}$ is a projector; let us denote by $\pi = gf$ the complementary projector of D and $\pi' = 1 - D'$ the complementary projector of D'.

We already know the relations $hh = h'h' = 0$. Furthermore $hh' = hh\psi = 0$ et $h'h = \phi hh = 0$. In fact any composition of an operator of type h with an operator of type π is null. Firstly $\pi h = (1 - \widehat{d}h - h\widehat{d})h = h - h\widehat{d}h = h - h = 0$ and $h\pi =$

$h(1-\widehat{d}h-h\widehat{d}) = h - h\widehat{d}h = h - h = 0$. Next $\pi h' = \pi h\psi = 0$ and $h'\pi = \phi h\pi = 0$. Then $\pi' h' = h'\pi' = 0$ is proved like $\pi h = h\pi = 0$. Finally $\pi' h = (1-\widehat{d'}h' - h'\widehat{d'})h$; but $h'h = 0$ and $\widehat{d'} = \widehat{d}+\widehat{\delta}$, therefore $\pi' h = h - \phi h(\widehat{d}+\widehat{\delta})h = h - \phi h\widehat{d}h - \phi h\widehat{\delta}h = h - \phi h - (1-\phi)h = 0$ (because $h\widehat{d}h = h$ and $\phi h\widehat{\delta} = 1-\phi$). In the same way $h\pi' = h(1-\widehat{d'}h'-h'\widehat{d'}) = h - h(\widehat{d}+\widehat{\delta})h\psi = h - h\widehat{d}h\psi - h\widehat{\delta}h\psi = h - h\psi - h(1-\psi) = 0$.

Let us now consider the compositions $\pi\pi'\pi$ and $\pi'\pi\pi'$. Firstly $\pi\pi'\pi = \pi(1-\widehat{d'}h' - h'\widehat{d'})\pi = \pi^2 = \pi$, because $\pi h' = h'\pi = 0$. In the same way $\pi'\pi\pi' = \pi'(1-\widehat{d}h - h\widehat{d})\pi' = \pi'^2 = \pi'$. Therefore the operators π and π' are inverse morphisms between the images of π' and π ; they are only homomorphisms *of graded modules*, in general non compatible with the natural differentials of the respective images. But the image of π has a bijective mapping towards the small graded module C through f and g, so that a composition provides an isomorphism of graded modules between C and the image of π' which allows us to install a new differential on C deduced from the differential of $\mathrm{im}(\pi')$, restriction of $\widehat{d'} = \widehat{d}+\widehat{\delta}$.

Firstly let us note that $h'g = \phi hg = 0$, and that $fh' = fh\psi = 0$. Taking account of what was explained in the previous paragraph, it is natural to define $g' = \pi' g = (1-\widehat{d'}h' - h'\widehat{d'})g = g - \phi h\widehat{d}g - \phi h\widehat{\delta}g = -\phi hgd + (1-\phi h\widehat{\delta})g = \phi g$. Then the "projection" f' will be the composition of the actual projection π' with the composition $f\pi$. But $f\pi = f(1-\widehat{d}h - h\widehat{d}) = f - f\widehat{d}h - fh\widehat{d} = f - dfh - fh\widehat{d} = f$ and we obtain $f' = f\pi\pi' = f\pi' = f(1-\widehat{d'}h' - h'\widehat{d'}) = f - f\widehat{d}h\psi - f\widehat{\delta}h\psi = -\widehat{d}fh\psi + f(1-\widehat{\delta}h\psi) = f\psi$. We have obtained the announced formulas for the desired reduction components f' and g'.

The new differential to be installed on the graded module underlying C remains to be determined. We naturally compute: $d+\delta = f\pi(\widehat{d}+\widehat{\delta})\pi' g = f(\widehat{d}+\widehat{\delta})\pi g = f\widehat{d}\pi' g + f\widehat{\delta}\phi g = f\widehat{d}(1-\widehat{d'}h' - h'\widehat{d'})g + f\widehat{\delta}\phi g = f\widehat{d}g - f\widehat{d}\widehat{d'}h'g - dfh'\widehat{d'}g + f\widehat{\delta}\phi g = f\widehat{d}g + f\widehat{\delta}\phi g = d + f\widehat{\delta}\phi g = d + f\psi\widehat{\delta}g$; we must therefore choose $\delta = f\widehat{\delta}\phi g = f\psi\widehat{\delta}g$.

The perturbation lemma is proved. ■

EAT implementation.

The algorithm **bpl** is the function `rdc-bpl` of the EAT program [42]. We think the power of Common-Lisp is well illustrated by the text of this Lisp function (about 25 lines), more powerful than the classical spectral sequences. The programming lines very precisely follow the mathematical definitions of the components of the wished reduction, see Appendix 24. Whatever the spectral sequence to be transformed into an actual algorithm, we use the *same* Lisp function `rdc-bpl`.

We illustrate how the `rdc-bpl` function can be used by our implementation of the twisted Eilenberg-Zilber theorem, the demonstration of which by Shih Weishu is precisely at the origin of the perturbation lemma. The predefined EAT object located through the symbol `*deltab*` implements the simplicial set $\overline{\Delta}^{\infty}$, the quotient of the infinite simplex Δ^{∞}, spanned by the natural numbers, by its 0-skeleton:

```
> *deltab*  ==>
```

```
[SS-2]
> (eqs *deltab* '(0 3) '(0 4))  ==>
NIL
> (eqs *deltab* '(3) '(4))  ==>
T
```

You see the edges (0 3) and (0 4) are different, but the vertices (3) and (4) are the same; in fact only one vertex in $\overline{\Delta}^{\infty}$. The function `loop-space` can construct its loop-space:

```
> (setf *gdeltab* (loop-space *deltab*))  ==>
[SS-4]
> (gdl *gdeltab* 0 2 (loop3 nil '(0 1 2 3) 2))
<ASM * <<LOOP ( <PWR * (0 2 3) ** 1>
                <PWR * (1 2 3) ** -1>
                <PWR * (0 2 3) ** 1>
                <PWR * (1 2 3) ** -1>)>>>
```

You recognize the Kan formula for the expression of $\partial_0\tau(\sigma_{0123})^2$, see Section 7. The `eilenberg-zilber` function constructs the Eilenberg-Zilber reduction corresponding to the non-twisted product $G\overline{\Delta}^{\infty} \times \overline{\Delta}^{\infty}$:

```
> (setf initial-rdc
        (eilenberg-zilber *gdeltab* *deltab*))  ==>
[RDC-0]
```

Let us look at the differential of the top chain complex; we see it is non-twisted:

```
> (setf tcc-d (cc-d (rdc-tcc initial-rdc)))  ==>
[MRP-10]
> (? tcc-d 3 (cpr '(2 1 0) (loop3 )
                 nil '(0 1 2 3)))  ==>
----------------------------------------{CMB 2}
<MNM -1 * <CPR 1-0 <<LOOP *>> * (0 1 2)>>
<MNM 1 * <CPR 1-0 <<LOOP *>> * (0 1 3)>>
<MNM -1 * <CPR 1-0 <<LOOP *>> * (0 2 3)>>
<MNM 1 * <CPR 1-0 <<LOOP *>> * (1 2 3)>>
-----------------------------------------------
```

To observe the non-torsion, we have computed the boundary of the simplex whose vertical projection is the simplex (0 1 2 3), and horizontal projection is the base point of the loop space. Note how this simplex in the product is constructed: the `cpr` function uses the necessary multi-degeneracy (2 1 0) (to be understood as $\eta_2\eta_1\eta_0$), the null loop (`loop3` without any arguments), `nil` (no degeneracy for the second factor) and the simplex (0 1 2 3).

The simplicial set which is the co-universal fibration is constructed by the function `cpr-tau`:

```
> (setf twisted-product (cpr-tau *deltab*))  ==>
[SS-8]
```

This function uses only a (reduced) simplicial set X and constructs the co-universal total space $GX \times_\tau X = \Omega X \times_\tau X$. The torsion can be observed in computing the 0-face of the same simplex:

```
> (gdl twisted-product 0 3 (cpr '(2 1 0) (loop3)
                                nil '(0 1 2 3)))  ==>
<ASM * <CPR * <<LOOP ( <PWR * (0 1 2 3) ** 1>)>> * (1 2 3)>>
```

or in computing the associated chain complex, the corresponding differential and the boundary of the same simplex:

```
> (setf twisted-tcc (ss-cc twisted-product))  ==>
[CC-11]
> (setf twisted-tcc-d (cc-d twisted-tcc))  ==>
[MRP-14]
> (? twisted-tcc-d 3 (cpr '(2 1 0) (loop3 )
                          nil '(0 1 2 3)))  ==>
---------------------------------------------------{CMB 2}
<MNM -1 * <CPR 1-0 <<LOOP *>> * (0 1 2)>>
<MNM 1 * <CPR 1-0 <<LOOP *>> * (0 1 3)>>
<MNM -1 * <CPR 1-0 <<LOOP *>> * (0 2 3)>>
<MNM 1 * <CPR * <<LOOP ( <PWR * (0 1 2 3) ** 1>)>> * (1 2 3)>>
----------------------------------------------------------
```

We are now exactly in a situation where the Basic Perturbation Lemma can be applied; the perturbation is the difference between both differentials:

```
> (setf delta (sbt-mrp-from-mrp tcc-d twisted-tcc-d))  ==>
[MRP-15]
```

and we can pass to the function `rdc-bpl` the initial reduction and the perturbation to obtain the new reduction:

```
> (setf twisted-rdc (rdc-bpl initial-rdc delta))
[RDC-1]
```

Let us compare for example the value of the bottom differentials for the generator $* \otimes \sigma_{0123}$:

```
> (d-? (rdc-bcc initial-rdc) 3 (tpr 0 (loop3) 3 '(0 1 2 3)))  ==>
---------------------------------------------------{CMB 2}
<MNM -1 * <TPR <<LOOP *>> (0 1 2)>>
<MNM 1 * <TPR <<LOOP *>> (0 1 3)>>
<MNM -1 * <TPR <<LOOP *>> (0 2 3)>>
<MNM 1 * <TPR <<LOOP *>> (1 2 3)>>
----------------------------------------------------------
> (d-? (rdc-bcc twisted-rdc) 3 (tpr 0 (loop3) 3 '(0 1 2 3)))  ==>
---------------------------------------------------{CMB 2}
<MNM -1 * <TPR <<LOOP *>> (0 2 3)>>
<MNM 1 * <TPR <<LOOP *>> (0 1 3)>>
```

```
<MNM -1 * <TPR <<LOOP *>> (0 1 2)>>
<MNM 1 * <TPR <<LOOP ( <PWR * (0 1 2 3) ** 1>
                       <PWR 2-1 (1 3) ** 1>)>> (3)>>
<MNM 1 * <TPR <<LOOP ( <PWR 2 (0 1 2) ** 1>
                       <PWR 1 (1 2 3) ** 1>
                       <PWR 2-1 (2 3) ** 1>)>> (3)>>
[... Lines deleted ...]
------------------------------------------------------------
```

We have found the quite complicated Shih differential for the simplex $* \otimes \sigma_{0123}$ of $G\overline{\Delta}^{\infty} \otimes_t \overline{\Delta}^{\infty}$. See Section 14.

11 Bicomplexes and cones.

A convenient method to understand how the ordinary spectral sequences work consists in examining the case of *bicomplexes*; see for example [31, Section XI-6]. We use the same method here, and this section is devoted to the version *with effective homology* of the bicomplex spectral sequence.

11.1 Bicomplexes.

Definition 31 — A *locally effective bicomplex* B is a system:

$$B = \{B_{p,q}, d'_{p,q}, d''_{p,q}\}_{p,q \in \mathbb{Z}}$$

where every $B_{p,q}$ is a locally effective (free) $\mathbb{Z}$-module, every $d'_{p,q}$ is a morphism $d'_{p,q} : B_{p,q} \to B_{p-1,q}$, every $d''_{p,q}$ is a morphism $d''_{p,q} : B_{p,q} \to B_{p,q-1}$; these morphisms satisfy the relations $d'_{p,q} d'_{p+1,q} = 0$ (the horizontals are chain complexes), $d''_{p,q} d''_{p,q+1} = 0$ (the verticals are chain complexes too) and $d'_{p+1,q} d''_{p+1,q+1} + d''_{p,q+1} d'_{p+1,q+1} = 0$ (the squares are anti-commutative) for any integer pair (p,q). The bicomplex is *first quadrant* if $B_{p,q} = 0$ when $p < 0$ or $q < 0$. The bicomplex is *effective* if every chain group $B_{p,q}$ is effective.

Only the *first quadrant* bicomplexes are considered in this section; obvious generalizations can possibly be stated and proved for other situations. If a bicomplex is locally effective, an algorithm is available to decide whether a machine object a is a distinguished generator of $B_{p,q}$; in general, $B_{p,q}$ is not of finite type, no global information is available about it, but if a is any distinguished generator of $B_{p,q}$, a provided algorithm can compute the images of a by $d'_{p,q}$ and $d''_{p,q}$. If B is an effective bicomplex, a further algorithm can work for every integer pair (p,q) to compute the necessarily finite distinguished basis of $B_{p,q}$.

Definition 32 — Let B be a locally effective bicomplex $B = \{B_{p,q}, d'_{p,q}, d''_{p,q}\}$. The *totalization* of the bicomplex B is the chain complex $T(B) = \{T_n(B), d_n\}_{n \in \mathbb{Z}}$ where:

- $T_n(B) = \oplus_{p+q=n} B_{p,q}$;
- $d_n \mid_{B_{p,q}} = d'_{p,q} \oplus d''_{p,q}$ if $p + q = n$.

If B is a locally effective bicomplex, its totalization is also locally efective. Some precautions must be taken to avoid generator collisions: the *same* machine object a could be a generator of *several different* $B_{p,q}$'s; the solution consists in defining for each generator a of $B_{p,q}$ a generator (a, p, q) for $T(B)_{p+q}$. Methods of this sort must constantly been used, are always obvious and will be no longer detailed.

If B is effective and *first quadrant*, its totalization is also effective. This is false for a second quadrant bicomplex, unless some vanishing conditions are satisfied by the components.

A bicomplex can also be presented as a (horizontal) chain complex of (vertical) chain complexes. Let us suppose that, for every integer p, a chain complex $B_{p,*}$ is given and also a chain complex morphism $b_p : B_{p,*} \to B_{p-1,*}$; the system of chain complexes is in turn a complex of chain complexes if the condition $b_{p-1} b_p = 0$ holds for every p. Such a system can be transformed into a bicomplex, using the *suspension* trick; an element of $B_{p,q}$, normally of degree q must now have the modified degree $p + q$; this is done by applying the suspension operator S^p to $B_{p,*}$ which lets $B_{p,*}$ unchanged, except that the degree function decides now the degree of $B_{p,q}$ is $p + q$. The Koszul convention then explains the right definition for the new differential is $dS^p x = (-1)^p S^p dx$: in other words the sign of the differential is changed for all the *odd* columns. So that the $d'_{p,q}$ of the bicomplex to be constructed is simply $b_{p,q}$, and the $d''_{p,q}$ is $(-1)^p d_q(B_{p,*})$. The vanishing conditions for the operators of the bicomplex are then satisfied.

Our version *with effective homology* of the spectral sequence of a bicomplex concerns the case where every column $(B_{p,*})_{EH}$ is a chain complex with effective homology. A general algorithm takes $((B_{p,*})_{EH}, b_p)_{p \in \mathbb{Z}}$ as input and computes a version $T(B)_{EH}$ of the totalization $T(B)$ also with effective homology. The result is much stronger than the ordinary one obtained by the classical spectral sequence: our process is stable and the obtained version for the totalization $T(B)_{EH}$ has exactly the same structure as the chain complex components $(B_{p,*})_{EH}$. In particular if you are interested by the homology groups of $T(B)$, the effective component of $T(B)_{EH}$ gives the result, no mysterious and unreachable differential $d^r_{p,q}$ to be determined (cf. [33, pp. 6 and 28]), no extension problems to be solved at abutment; furthermore if $T(B)$ is used as a component of a new construction, then the version $T(B)_{EH}$ can probably be used to obtain again a version with effective homology of the object newly constructed, and so on.

Theorem 33 — *An algorithm can be constructed:*

$$\mathbf{TOTAL}_{EH} : \mathcal{B}_{EH} \longrightarrow \mathcal{CC}_{EH}$$

where $\mathcal{B}_{EH}$ is the type of first quadrant *bicomplexes defined as a complex $B_{EH} =$*

$((B_{p,*})_{EH}, b_p)_{p\in\mathbb{Z}}$, and the output $\mathbf{TOTAL}_{EH}(B_{EH})$ is a version with effective homology $T(B)_{EH}$ of the totalization of the underlying bicomplex B.

More precisely, for *every* column index p, a chain complex with effective homology $(B_{p,*})_{EH} = (B_p, EB_p, \varepsilon_p)$ is given, that is, a triple with a locally effective chain complex, an effective one and a homotopy equivalence between them; also a morphism $b_p : B_p \to B_{p-1}$ between *underlying* chain complexes is also given; the condition $b_{p-1}b_p = 0$ is satisfied. Then the algorithm outputs a triple $(T(B), ET(B), T(\varepsilon))$ with the ordinary totalization $T(B)$, some *effective* chain complex $ET(B)$ and some homotopy equivalence $T(\varepsilon)$ between them.

PROOF. Each homotopy equivalence ε_p is a pair (ρ_p^1, ρ_p^2) of reductions. The reduction ρ_p^1 is a 5-tuple $\rho_p^1 = (\widehat{B}_p, B_p, f_p^1, g_p^1, h_p^1)$; in the same way the reduction ρ_p^2 is a 5-tuple $\rho_p^2 = (\widehat{B}_p, EB_p, f_p^2, g_p^2, h_p^2)$ where the chain complex EB_p is effective. The (horizontal) morphisms b_p are only between the bottom right-hand complexes B_p, but we can also naturally define:

- $\widehat{b}_p = g_p^1 \circ b_p \circ f_p^1 \; : \; \widehat{B}_p \to \widehat{B}_{p-1}$;
- $Eb_p = f_p^2 \circ g_p^1 \circ b_p \circ f_p^1 \circ g_p^2 \; : \; EB_p \to EB_{p-1}$.

The first definition is correct and actually defines a top bicomplex $\widehat{B} = (\widehat{B}_p, \widehat{b}_p)_{p\in\mathbb{Z}}$: the necessary relation $\widehat{b}_{p-1} \circ \widehat{b}_p = 0$ holds, essentially because $f_{p-1}^1 \circ g_{p-1}^1 = 1_{B_{p-1}}$. But the second definition *does not* define a bicomplex $(EB_p, Eb_p)_{p\in\mathbb{Z}}$, because this time, to prove the necessary relation $Eb_{p-1} \circ Eb_p = 0$, we should use the hoped-for relation $g_{p-1}^2 \circ f_{p-1}^2 = 1_{\widehat{B}_p}$, but this relation is false in general; the composition $g_{p-1}^2 \circ f_{p-1}^2$ is only *homotopic* to the identity, certainly not equal to, except when the reduction ρ_p^2 is trivial; this is the essential obstacle which is *partially* overcome by the bicomplex spectral sequence. Here, using the perturbation lemma, we obtain an easy *complete* solution.

But let us firstly examine what can be done on the left-hand part. It is in fact easy to verify that the entire collection of left-hand reductions $(\rho_p^1)_p$ can be put together to produce again a reduction $T(\rho^1) = (T(\widehat{B}), T(B), T(f^1), T(g^1), T(h^1))$. Essentially the reduction $T(\rho^1)$ is the simple direct sum of all the "column" reductions ρ_p^1, no accident happens. The last step consists in determining a *correct* bottom right-hand chain complex $T(EB)$ and a reduction between $T(\widehat{B})$ and $T(EB)$. The notation $T(\ldots)$ is a litlle mistaking because at this time no bicomplex EB is defined.

The tentative incorrect definition above for the bicomplex EB can be roughly simplified if we decide to annihilate the problematic horizontal differentials. Let us denote by EB' the bicomplex obtained from the columns EB_p, with *null* horizontal differentials; this time the bicomplex EB' is correctly defined. In the same way we can consider the bicomplex $\widehat{B'}$ obtained from $\widehat{B}$ by deciding to remove the horizontal differentials $\widehat{b}_p$, more precisely to replace them by null operators. Then $\widehat{B'}$ is a new bicomplex with the same graded module as $\widehat{B}$, but with a different differential. We are in a situation where the basic perturbation lemma can be applied.

The initial reduction is $(T(\widehat{B'}), T(EB'), T(\rho'^2))$; it is a simple direct sum of "column" reductions. The perturbation δ to be applied to the top chain complex is given by the horizontal differentials $\delta = \oplus_p \widehat{b}_p$. We can use the basic perturbation lemma if the nilpotency condition is proved; the perturbation lowers the horizontal p-degree , and the homotopy component $h = \oplus_p h^2_p$ of the initial reduction lets it unchanged (such a homotopy lives inside a column), so that the iteration process $(h\delta)^n$ eventually goes for each element towards the columns with negative index, which were assumed null (first quadrant condition): the nilpotency condition is satisfied.

The basic perturbation lemma produces a new reduction $\rho^2 = (T(\widehat{B}), T(EB), T(f^2), T(g^2), T(h^2))$. In this expression, the notation $T(\widehat{B})$ is correct, but the other notations $T(\ldots)$ are not. More precisely, for example, $T(EB)$ is not the totalization of a <u>bi</u>complex, but the totalization of a <u>multi</u>complex; this means that if you detail $T(EB)$ with the usual bigraduation $T(EB) = \oplus EB_{p,q}$, then the differentials after perturbation coming from $EB_{p,q}$ are $d^0_{p,q} = Ed_{p,q} : EB_{p,q} \to EB_{p,q-1}$ (the differential coming from the initial column), $d^1_{p,q} = Eb_{p,q}$ (the tentative horizontal differential), $d^2_{p,q} = f^2_{p-2,q+1}\widehat{b}_{p-1,q+1}h^2_{p-1,q}\widehat{b}_{p,q}g^2_{p,q}$ (a necessary correction), and so on; you must run the following path for each component $d^r_{p,q} : EB_{p,q} \to EB_{p-r,q+r-1}$; firstly climb through $g^2_{p,q}$ into $\widehat{B}_{p,q}$, then follow a stairs path in the bicomplex $\widehat{B}$ where the horizontal components are some $\widehat{b}_{p',q'}$ and the vertical ones are some $h^2_{p',q'}$, reach the component $\widehat{B}_{p-r,q+r-1}$, finally get down to $EB_{p-r,q+r-1}$ through $f^2_{p-r,q+r-1}$; here the differential component $d^r_{p,q}$ is *entirely* obtained from the initial data. These considerations are direct consequences of the explicit formulas giving the components of the perturbed reduction. Analogous descriptions can be obtained for the components f^2, g^2 and h^2 of the final reduction between $T(\widehat{B})$ and $T(EB)$. ■

11.2 The cone construction.

A particular case is important, when the bicomplex contains only two columns, two chain complexes $B_1 = C$ and $B_0 = C'$, the other columns being null. In such a case, the horizontal differential is a unique chain complex morphism $b : C \to C'$, and the totalization is known as the *cone* $C'' = \mathrm{Cone}(b)$; it is a chain complex where the chain group C''_n is defined as $C''_n = C_{n-1} \oplus C'_n$ and the differential $d''_n : C''_n \to C''_{n-1}$ is defined by the matrix formula:

$$d''_n = \begin{bmatrix} -d_{n-1} & 0 \\ b_{n-1} & d'_n \end{bmatrix} = \begin{bmatrix} -d & 0 \\ b & d' \end{bmatrix}$$

where in the second expression the indices are not showed. Let us suppose the chain complexes C and C' are with effective homology. This means four reductions are provided, $\rho_1 = (\widehat{C}, C, f_1, g_1, h_1)$, $\rho_2 = (\widehat{C}, EC, f_2, g_2, h_2)$, $\rho'_1 = (\widehat{C'}, C', f'_1, g'_1, h'_1)$

and $\rho'_2 = (\widehat{C'}, EC', f'_2, g'_2, h'_2)$. We would like to compute the effective homology of the cone $C'' = \mathrm{Cone}(b)$. A morphism $\widehat{b} : \widehat{C} \to \widehat{C'}$ is naturally defined by $\widehat{b} = g'_1 b f_1$, which allows us to define a "top cone" $\widehat{C''}$. In the same way, a morphism $Eb : EC \to EC'$ is defined by $Eb = f'_2 \widehat{b} g_2$ defining a bottom right-hand cone EC'': because the "bicomplex" has only two columns, the "accident" about the bottom right-hand bicomplex does not happen, so that the perturbation lemma is now useless to guess the right differential at this place. Now combining the components of the four provided reductions, it is easy to construct a new pair of reductions $\rho''_1 = (\widehat{C''}, C'', f''_1, g''_1, h''_1)$ and $\rho''_2 = (\widehat{C''}, EC'', f''_2, g''_2, h''_2)$; the left-hand reduction is simply the direct sum of the reductions ρ_1 and ρ'_1:

$$f''_1 = \begin{bmatrix} f_1 & 0 \\ 0 & f'_1 \end{bmatrix} \; ; \; g''_1 = \begin{bmatrix} g_1 & 0 \\ 0 & g'_1 \end{bmatrix} \; ; \; h''_1 = \begin{bmatrix} -h_1 & 0 \\ 0 & h'_1 \end{bmatrix}$$

but the right-hand one is a little more complicated:

$$f''_2 = \begin{bmatrix} f_2 & 0 \\ f'_2 \widehat{b} h_2 & f'_2 \end{bmatrix} \; ; \; g''_2 = \begin{bmatrix} g_2 & 0 \\ -h'_2 \widehat{b} g_2 & g'_2 \end{bmatrix} \; ; \; h''_2 = \begin{bmatrix} -h_2 & 0 \\ h'_2 \widehat{b} h_2 & h'_2 \end{bmatrix}$$

Corollary 34 — *An algorithm can be constructed:*

$$\mathbf{CONE}_{EH} : [\mathcal{CC}_{EH} \times \mathcal{CC}_{EH} \times \mathcal{M}]_{\chi} \longrightarrow \mathcal{CC}_{EH}$$

where $[\ldots]_{\chi}$ is the type of coherent triples (C_{EH}, C'_{EH}, b), two chain complexes with effective homology and a morphism between the underlying *chain complexes. The algorithm $\mathbf{CONE}_{EH}$ then outputs a chain complex with effective homolgy C''_{EH}, the underlying chain complex being the cone of the morphism b.* ■

Suppose now you have a bicomplex with three non-null columns B_0, B_1 and B_2. You can consider its totalization as the cone of a morphism defined on the column B_2 towards the… cone of the morphism between the columns B_1 and B_0. Applying twice the previous corollary, you find a few more complicated formulas to process such a system of three columns. You can iterate and in this way find the right formulas for the reduction of a "big" bicomplex when a reduction of each column is given. You have in this way rediscovered the perturbation lemma.

Proposition 35 — *Let $\rho = (\widehat{C}, C, f, g, h)$ be a reduction. Then both cones* $\mathrm{Cone}(f)$ *and* $\mathrm{Cone}(g)$ *are acyclic; more precisely a reduction to the null chain complex can be defined.*

Proof. The necessary homotopy operators are:

$$h_{\mathrm{Cone}(f)} = \begin{bmatrix} -h & g \\ 0 & 0 \end{bmatrix} \; ; \; h_{\mathrm{Cone}(g)} = \begin{bmatrix} 0 & f \\ 0 & h \end{bmatrix} . \ ■$$

Proposition 36 — *Two reductions $\rho = (C, C', f, g, h)$ and $\rho' = (C', C'', f', g', h')$ can be composed to give the reduction $\rho'' = (C, C'', f'f, gg', h + gh'f)$.* ■

But how to compose two homotopy equivalences? Let $\varepsilon = (\rho_1, \rho_2)$ and $\varepsilon' = (\rho'_1, \rho'_2)$ be two homotopy equivalences between C, C' and C'':

$$\underbrace{C \overset{\rho_1}{\Leftarrow} \widehat{C} \overset{\rho_2}{\Rightarrow} C'}_{\varepsilon} \underbrace{\overset{\rho'_1}{\Leftarrow} \widehat{C'} \overset{\rho'_2}{\Rightarrow}}_{\varepsilon'} C''$$

We would like to compose them. Then we can put together $\widehat{C}$, C' and $\widehat{C'}$ with the help of f_2 and f'_1 to organize them as a "bicone" $BC = \mathrm{Bicone}(f_2, f'_1)$. This object BC can be considered as the cone of f_2 from $\widehat{C}$ to the cone of f'_1; because the last cone is contractible, we can construct a reduction $\rho''_1 : C \Leftarrow BC$. But in a symmetric way we can also construct a reduction $\rho''_2 : BC \Rightarrow C''$. The pair (ρ''_1, ρ''_2) is the wished homotopy equivalence. ■

Proposition 37 — *An algorithm can be constructed:*

$$\textbf{HE-COMP} : [\mathcal{HE} \times \mathcal{HE}]_\chi \longrightarrow \mathcal{HE}$$

where $[\ldots]_\chi$ is the type of pairs of homotopy equivalences which could be composed, the output being the composition. ■

11.3 EAT implementation.

The algorithms $\mathbf{TOTAL}_{EH}$ and $\mathbf{CONE}_{EH}$ are not implemented for the general case in the EAT program, but a particular case of the first one is an essential component allowing us to obtain versions with effective homology of iterated loop spaces. We illustrate this point with the following example: let us take again the simplicial set with effective homology `odos3eh` implementing $\Omega(\Omega S^3 \cup_2 D^3)$ in the section 9, and let us inspect it:

```
> odos3eh
[SS-EH 13]
> (oeh-heq (oeh 13))
[HEQ-16]
> (heq-lrdc (heq 16))
[RDC-41]
> (rdc-org (rdc 41))
(CMP-RDC [RDC-40] [RDC-38])
> (rdc-org (rdc 38))
(BPL [HEQ-14] [MRP-420] NEW-LRDC)
> (heq-org (heq 14))
(PRE-COTOR-2HEQ [HEQ-12] [HEQ-13])
> (mrp-org (mrp 420))
(COBAR-HOR-D [COM-11] [COM-12] [CC-117])
```

You can read:

1. Our implementation of $\Omega(\Omega S^3 \cup_2 D^3)$ is the object with effective homology `[SS-EH 13]`;
2. The `HEQ` slot of this object is the homotopy equivalence `[HEQ-16]`;
3. The `LRDC` (left reduction) slot of this homotopy equivalence is the reduction `[RDC-41]`;
4. This reduction is the composition of the reductions `[RDC-40]` and `[RDC-38]`;
5. The last reduction has been obtained by the Basic Perturbation Lemma (`BPL`) working with the homotopy equivalence `[HEQ-14]` and the perturbation `[MRP-420]`;
6. The homotopy equivalence `[HEQ-14]` is a "Pre-Cotor" of homotopy equivalences: the prefix "Pre" means no horizontal differential is present; two homotopy equivalences ε' = `[HEQ-12]` and ε = `[HEQ-13]` are given and the homotopy equivalence `[HEQ-14]` is nothing but:
$$\bigoplus_{n=0}^{\infty} \varepsilon' \otimes \varepsilon^{\otimes n}$$
that is, a homotopy equivalence between bicomplexes, each "column" being the homotopy equivalence $\varepsilon' \otimes \varepsilon^{\otimes n}$;
7. The perturbation `[MRP-420]` to be applied in the perturbation lemma is the missing horizontal differential in the bottom left-hand bicomplex of the previous homotopy equivalence; it is produced by the function `COBAR-HOR-D`.

Detailed explanations will be given later, see Sections 12, 17 and 18. Any object produced by the EAT program can be examined in this way to elucidate when and why it has been constructed.

12 Tensor products.

Processing tensor products of chain complexes with effective homology is as elementary as usual. We only state the useful results with minimal indications about proofs.

Proposition 38 — *An algorithm can be constructed:*

$$\textbf{TPR-2CC}_{EH} : \mathcal{CC}_{EH} \times \mathcal{CC}_{EH} \longrightarrow \mathcal{CC}_{EH}$$

which may work on pairs (C_1, C_2) of chain complexes with effective homology, returning a version with effective homology of the tensor product of underlying chain complexes.

PROOF. A homotopy equivalence is a pair of reductions, and it is sufficent to know how to compute the tensor product of two reductions $\rho_1 = (\widehat{C}_1, C_1, f_1, g_1, h_1)$ and $\rho_2 = (\widehat{C}_2, C_2, f_2, g_2, h_2)$. A tensor product $\rho = \rho_1 \otimes \rho_2$ can be defined by $\rho = (\widehat{C}_1 \otimes \widehat{C}_2, C_1 \otimes C_2, f_1 \otimes f_2, g_1 \otimes g_2, h_1 \otimes 1_{\widehat{C}_2} + g_1 f_1 \otimes h_2)$, without forgetting the Koszul convention for the last component. ■

Corollary 39 — *An algorithm can be constructed:*

$$\mathbf{TPR\text{-}N}_{EH} : \mathcal{CC}_{EH} \times \mathbb{N} \longrightarrow \mathcal{CC}_{EH}$$

computing some tensor power of a given chain complex with effective homology. ■

Corollary 40 — *If every column of a first quadrant bicomplex is a tensor product of chain complexes with effective homology, a (general) algorithm can construct a version with effective homology of its totalization.*

PROOF. Combine Theorem 33 and Corollary 38. ■

EAT implementation.

The algorithms of this section are not implemented in the EAT program because not necessary for our particular planned application, the iterated loop spaces. But they are used in a hidden way. The following illustration anticipates other sections, but this should not really hamper the understanding of what concerns the current section. Let us consider again the EAT example of Section 11, page 80, concerning some parts of the construction of a version with effective homology of the simplicial set $X = \Omega Y$ where $Y = \Omega S^3 \cup_2 D^3$. The main ingredients are two homotopy equivalences ε' and ε. The first one is a reduction of a twisted tensor product $C_*(\Omega Y) \otimes_t C_*(Y)$ to the unit chain complex (only one $\mathbb{Z}$ in degree 0); such a reduction can be constructed because the total space of the co-universal fibration $\Omega Y \to \Omega Y \times_\tau Y \to Y$ is contractible. So that:

$$\varepsilon' = (C_*(\Omega Y) \otimes_t C_*(Y) = C_*(\Omega Y) \otimes_t C_*(Y) \Rightarrow \mathbb{Z}).$$

(the left reduction is trivial). The other homotopy equivalence ε is simply the main ingredient of the previously computed version with effective homology of the simplicial set Y:

$$\varepsilon = (C_*Y \Leftarrow \widehat{C}_Y \Rightarrow EC_Y).$$

The complex $C_*(Y)$ has an Alexander-Whitney coalgebra structure, and the first complex $C_*(\Omega Y) \otimes_t C_*(Y)$ is a $C_*(Y)$ right comodule. This allows us to construct a bicomplex $\mathrm{Cobar}^{C_*(Y)}(C_*(\Omega Y) \otimes_t C_*(Y), \mathbb{Z})$; it is a bicomplex $B_{*,*}$ where the p-th column is the tensor product:

$$B_{p,*} = (C_*(\Omega Y) \otimes_t C_*(Y)) \otimes \overline{C}_*(Y)^{\otimes n},$$

the complex $\overline{C}_*(Y)$ being the "augmentation ideal", that is, the 0-degree component is removed. We are almost in the situation of Corollary 40; every column is a tensor product of chain complexes with effective homologies described by ε and ε'; the only difference is that $B_{*,*}$ is a second quadrant bicomplex. So that it is possible to implement this Cobar construction as a unique chain complex; if you remove the horizontal differentials coming from the coalgebra and comodule coproducts, you can even implement the Cobar so simplified (we call it a *Pre-Cobar*) as a chain complex with effective homology, the homotopy equivalence of which being nothing but the object `[HEQ-14]` showed in the previous section; the bottom right-hand chain complex is of finite type because Y is 1-reduced. Let us denote by M the comodule $C_*(\Omega Y) \otimes_t C_*(Y)$; the object `[HEQ-14]` looks like the diagram:

$$\text{Pre-Cobar}^{C_*Y}(M, \mathbb{Z}) \Leftarrow \text{Pre-Cobar}^{\widehat{C}_Y}(M, \mathbb{Z}) \Rightarrow \text{Pre-Cobar}^{EC_Y}(\mathbb{Z}, \mathbb{Z}).$$

All these Pre-Cobar's are defined, though $\widehat{C}_Y$ and EC_Y *are not* coalgebras: no horizontal differential is required.

Now the removed horizontal differential (the object `[MRP-420]`) is the *perturbation* which we must take account of, which has been constructed by the EAT function `COBAR-HOR-D`. Then the perturbation lemma can work with the Pre-Cobar and the perturbation to construct a new homotopy equivalence where this time the bottom left-hand chain complex is the right Cobar, not the Pre-Cobar: the horizontal differential is now present. We obtain the diagram

$$\text{Cobar}^{C_*Y}(M, \mathbb{Z}) \Leftarrow \text{“Cobar”}^{\widehat{C}_Y}(M, \mathbb{Z}) \Rightarrow \text{“Cobar”}^{EC_Y}(\mathbb{Z}, \mathbb{Z}).$$

with a true Cobar at the bottom left-hand side, but pseudo-Cobar's for the top chain complex and also the bottom right-hand one: the pseudo-Cobar structure has been transmitted from the true one at the bottom left-hand side to the others by the basic perturbation lemma. The perturbed bottom right-hand chain complex "Cobar"$^{EC_Y}(\mathbb{Z}, \mathbb{Z})$ is then a multi-complex of finite type which computes the homology of this Cobar; this is the reason why we call this homotopy equivalence a *Cotor*; and the Pre-Cobar with its effective homology is a *Pre-Cotor*. The left reduction of the Cotor homotopy equivalence is the object `[RDC-38]` seen in the previous section; it must next be combined with another reduction `[RDC-40]`, the "missing link"; these considerations will be detailed in Section 15.

13 Fibrations.

Let us detail the short Definition 9.1 for the fibration type. In this paper, a fibration is a *twisted cartesian product*. We need three simplicial sets, the fiber space F, the base space B and the structural group G; the latter is a simplicial set where G_n, the set of n simplices, carries a group structure; furthermore every face and degeneracy operator is a group homomorphism; we denote by e_n the neutral element of G_n. This simplicial group acts over the fiber space F, that is a simplicial map $F \times G \to F$ is given satisfying the traditional coherence properties for a right action.

A *torsion* operator $\tau : B \to G$ is given, describing how the cartesian product $F \times B$ must be *twisted* to obtain the total space $E = F \times_\tau B$. The torsion operator is a collection of maps $\tau_n : B_n \to G_{n-1}$ $(n \geq 1)$; then the set of n-simplices E_n is the same as the one $(F \times B)_n$ of the non-twisted product, but the 0-face operator is modified; in the non-twisted product, the 0-face is $\partial_0(f, b) = (\partial_0 f, \partial_0 b)$; taking account of the torsion operator, the 0-face operator in the twisted product $F \times_\tau B$ is $\partial_0(f, b) = (\partial_0 f . \tau(b), \partial_0 b)$. The torsion operator τ must satisfy a small set of relations in such a way the new ∂_0 so defined is again coherent with respect the other face and degeneracy operators:

$$\begin{array}{rcl} \partial_0 \tau(b) & = & \tau(\partial_1 b) \tau(\partial_0 b)^{-1}, \\ \partial_i \tau(b) & = & \tau(\partial_{i+1} b) \text{ if } i > 0, \\ \eta_i \tau(b) & = & \tau(\eta_{i+1} b) \text{ and} \\ \tau(\eta_0 b) & = & e_*. \end{array}$$

For example, if B is a reduced simplicial set, the Kan loop space model $F = GB$ can be constructed, see Section 7; it is a simplicial group and GB_n is the non-commutative free group generated by a subset of B_{n+1}, namely the set of non-0-degenerate $(n+1)$-simplices; if b is such a simplex of B, the corresponding generator of GB was denoted by $\tau(b)$ in Section 7. This "operator" τ can then after all be interpreted as a torsion operator, and this is correct because the coherence properties are satisfied; in fact Kan defined the group structure in GB (see Section 7) in such a way this is true! So we obtain the canonical fibration $GB \hookrightarrow GB \times_\tau B \to B$, the total space of which is contractible, see Section 17.

13.1 EAT implementation.

The fibration type in fact is not currently available in the EAT program. The particular case of the co-universal fibration:

$$GX \hookrightarrow GX \times_\tau X \to X$$

for an arbitrary 1-reduced locally effective simplicial set X is implemented thanks to the function `loop-space` which constructs GX from X and the function `cpr-tau` (twisted cartesian product) which constructs $GX \times_\tau X$ from X. See Section 10 where these functions have been used to illustrate the basic perturbation lemma, exactly in the situation which gives the twisted Eilenber-Zilber theorem.

Let us illustrate the universal nature of the `cpr-tau` function with the construction of the co-universal fibration for the 3-sphere. The three sphere is constructed, then the function cpr-tau is used to construct the total space $GS^3 \times_\tau S^3$ and the cpr-2ss function is also used to construct the non-twisted cartesian product:

```
> (setf s3 (sphere 3))
[SS-4]
> (setf gs3-times-tau-s3 (cpr-tau s3))
[SS-7]
> (setf gs3-times-s3 (cpr-2ss (loop-space s3) s3))
[SS-9]
```

Then the 0-face operator is successively applied to $(*, \sigma)$, $(*, \sigma)_\tau$ (the subscript τ indicates the simplex must be considered in the twisted product, but it is in fact the same machine object), $(\eta_0\tau(\sigma), \sigma)_\tau$ and $(\eta_0\tau(\sigma)^{-1}, \sigma)_\tau$, the simplex σ (coded as <S3>) being the unique 3-simplex of S^3, and $\tau(\sigma)$ (coded as (loop3 nil '<S3> 1)) being the corresponding "fundamental" simplex of GS^3:

```
> (gdl gs3-times-s3 0 3 (cpr '(2 1 0) (loop3) nil '<S3>))
<ASM 1-0 <CPR * <<LOOP *>> * *>>
> (gdl gs3-times-tau-s3 0 3 (cpr '(2 1 0) (loop3) nil '<S3>))
<ASM * <CPR * <<LOOP ( <PWR * <S3> ** 1>)>> 1-0 *>>
> (gdl gs3-times-tau-s3 0 3 (cpr '(0) (loop3 nil '<S3> 1) nil '<S3>))
<ASM * <CPR * <<LOOP ( <PWR * <S3> ** 2>)>> 1-0 *>>
> (gdl gs3-times-tau-s3 0 3 (cpr '(0) (loop3 nil '<S3> -1) nil '<S3>))
<ASM 1-0 <CPR * <<LOOP *>> * *>>
```

14 The twisted Eilenberg-Zilber algorithm.

One of the simplest topological (bi-) functors is the cartesian product functor. Thanks to the Eilenberg-Zilber theorem, computing the homology groups of a product is easy when the homology groups of each factor are given. The same in constructive algebraic topology, but the situation is interesting: it allows us to describe in a still very simple situation how it is necessary to proceed. In particular the Künneth theorem is no longer the key point.

In our organization, just after the ordinary Eilenberg-Zilber theorem, we find the *twisted* Eilenberg-Zilber theorem, due to Edgar Brown [12], who gave a demonstration based on acyclic carriers; the subsequent demonstration given by Shih Weishu [45] introduced the basic perturbation lemma; we repeat here Shih's proof.

From now on, except where otherwise stated, all the chain complexes canonically associated to a simplicial set are *normalized*, that is, only the non-degenerate simplices are generators.

14.1 The Eilenberg-Zilber theorem.

If X and Y are two simplicial sets, the *cartesian product* $X \times Y$ is naturally defined by $(X \times Y)_n = X_n \times Y_n$, and the face and degeneracy operators are the products of the corresponding operators of each factor simplicial set. If $\sigma \in X_n$ and

$\tau \in Y_n$ are two n-simplices, the notation (σ, τ) must be preferred to the tempting notation $\sigma \times \tau$: the pair notation (σ, τ) has the advantage to clearly mean this is the n-simplex whose first (resp. second) *projection* is σ (resp. τ). The "product" $\sigma \times \tau$, even if both simplices have not the same dimension, should normally denote the element of $C_*(X \times Y)$ which is the Eilenberg-MacLane image of the element $\sigma \otimes \tau \in C_*X \otimes C_*Y$, that is, the geometrical decomposition in simplices of the geometrical product of σ and τ.

Theorem 41 — *An algorithm can be constructed:*

$$\mathbf{EZ} : \mathcal{SS} \times \mathcal{SS} \longrightarrow \mathcal{RDC}$$

computing for any pair (X, Y) *of simplicial sets a reduction*

$$\mathbf{EZ}(X, Y) : C_*(X \times Y) \Rightarrow C_*(X) \otimes C_*(Y). \blacksquare$$

It is the Eilenberg-Zilber theorem, which is true as stated above only if the *normalized* chain complexes are considered. It is frequently presented as a consequence of the theorem of acyclic models [48], which is not very explicit; however this method can be made effective [38]. It is simpler to use the effective formulas for the Eilenberg-Zilber reduction $EZ(X, Y) = (f, g, h)$ known as the Alexander-Whitney (f), Eilenberg-MacLane (g) and Shih (h) operators. They come from the the recursive definition of these operators (see [18] and [19], or [45]).

The Eilenberg-MacLane and Shih operators have an essential "exponential" nature. It is not a question of method of computation, it is a question of very nature: the number of *different terms* produced by the Eilenberg-MacLane operator working on a tensor product of bi-degree (p, q) is the binomial coefficient $\binom{p+q}{p}$. So that any algorithm going through such a formula is necessarily of exponential complexity. Furthermore this formula is unique [35], and the difficulty localized here is therefore quite essential. In a sense, "classical" algebraic topology, typically the work around Steenrod operations, consists in avoiding the definitively exponential complexity of the Eilenberg-MacLane formula in order to be able to reach high dimensions; this paper on the contrary focuses on *arbitrary* spaces in low dimensions (something like < 12) where much interesting work is also to be done. A consequence of these considerations is that our computing methods will certainly not lead to high sphere homotopy groups; we are processing the *orthogonal* problem: we are not concerned by high dimensional invariants of *known* objects, we are only interested by the first invariants of *random* objects.

Corollary 42 — *An algorithm can be constructed:*

$$\mathbf{EZ}_{EH} : \mathcal{SS}_{EH} \times \mathcal{SS}_{EH} \longrightarrow \mathcal{SS}_{EH};$$

The output of the algorithm working on two simplicial sets with effective homology is a version with effective homology of the cartesian product of underlying simplicial sets.

PROOF. Let $(X, C_*(X), EC_X, \varepsilon_X)$ and $(Y, C_*(Y), EC_Y, \varepsilon_Y)$ two simplicial sets with effective homology. Eilenberg and Zilber give a homotopy equivalence $\varepsilon_1 : C_*(X \times Y) = C_*(X \times Y) \Rightarrow C_*(X) \otimes C_*(Y)$ (the left reduction is trivial); Corollary 39 gives also a homotopy equivalence ε_2 between $C_*(X) \otimes C_*(Y)$ and $EC_X \otimes EC_Y$. Composing these homotopy equivalences (Proposition 37), we obtain the wished homotopy equivalence between $C_*(X \times Y)$ and the *effective* chain complex $EC_X \otimes EC_Y$. ■

The Künneth theorem is not used; it allows you to *guess* the homology groups of $EC_X \otimes EC_Y$ if you know the homology groups of factors, but we are not concerned by this question: the chain complexes EC_X and EC_Y are effective, so that $EC_X \otimes EC_Y$ is also effective, and this is sufficient. We are on the contrary essentially interested by an *explicit* homotopy equivalence between $C_*(X \times Y)$ and $EC_X \otimes EC_Y$, and the explicit definition of the Eilenberg-Zilber reduction is the key point.

14.2 EAT implementation.

The bifunctor *cartesian product* for simplicial set is implemented, and also the bifunctor *tensor product* of chain complexes. The Eilenberg-Zilber algorithm 41 is implemented in EAT, but not its version 42 with effective homology.

Let us look at what is available. Let us construct a simplicial version of the real projective plane $X = P^2\mathbb{R}$:

```
> (setf p2r (moore 1 2))  ==>
[SS-4]
> (dotimes (i 3) (print (sbs p2r i)))  ==>
(*)
(<M1>)
(<MM2>)
NIL
```

The call `(moore 1 2)` constructs the Moore space Moore$(\mathbb{Z}_2, 1)$, that is the real projective plane. We see it has only one vertex `(*)`, one edge `(<M1>)` and one triangle (`<MM2>`). What about the faces of the triangle?

```
> (dotimes (i 3) (print (gdl p2r i 2 '<MM2>)))  ==>
<ASM * <M1>>
<ASM 0 *>
<ASM * <M1>>
NIL
```

The faces of index 0 and 2 are the unique edge, and the face 1 is the 0-degeneracy of the base point. The chain complex:

```
> (setf ccp2r (ss-cc p2r))  ==>
[CC-7]
> (d-? ccp2r 2 '<MM2>)  ==>
------------------------------{CMB 1}
<MNM 2 * <M1>>
--------------------------------------
```

has the right homology, because the boundary between dimensions 2 and 1 is the multiplication by 2. The cartesian square $Y = X \times X$ can be constructed:

```
> (setf p2r2 (cpr-2ss p2r p2r))  ==>
[SS-5]
> (dotimes (i 5) (print (sbs p2r2 i)))  ==>
(<CPR * * * *>)
(<CPR 0 * * <M1>> <CPR * <M1> 0 *> <CPR * <M1> * <M1>>)
(<CPR 1-0 * * <MM2>> <CPR 0 <M1> 1 <M1>> <CPR 1 <M1> 0 <M1>>
 <CPR 0 <M1> * <MM2>> <CPR 1 <M1> * <MM2>> <CPR * <MM2> 1-0 *>
 <CPR * <MM2> 0 <M1>> <CPR * <MM2> 1 <M1>> <CPR * <MM2> * <MM2>>)
(<CPR 1-0 <M1> 2 <MM2>> <CPR 2-0 <M1> 1 <MM2>> <CPR 2-1 <M1> 0 <MM2>>
 <CPR 0 <MM2> 2-1 <M1>> <CPR 1 <MM2> 2-0 <M1>> <CPR 2 <MM2> 1-0 <M1>>
 <CPR 0 <MM2> 1 <MM2>> <CPR 0 <MM2> 2 <MM2>> <CPR 1 <MM2> 0 <MM2>>
 <CPR 1 <MM2> 2 <MM2>> ...)
(<CPR 1-0 <MM2> 3-2 <MM2>> <CPR 2-0 <MM2> 3-1 <MM2>> <CPR 3-0 <MM2> 2-1 <MM2>>
 <CPR 2-1 <MM2> 3-0 <MM2>> <CPR 3-1 <MM2> 2-0 <MM2>> <CPR 3-2 <MM2> 1-0 <MM2>>)
NIL
> (length (sbs p2r2 3))  ==>
12
```

There are many simplices, in particular 12 3-simplices. Each simplex in the cartesian product is a `<CPR ...>` object with four slots, a multi-degeneracy operator for the first component, the non-degenerate simplex corresponding to the first component, and the same for the second component; a '`*`' for the multi-degeneracy means no degeneracy at all. So, the object `<CPR 2-1 <M1> 0 <MM2»` must be read $(\eta_2\eta_1\texttt{M1}, \eta_0\texttt{MM2})$, it is a 3-simplex; the object `<CPR * <MM2> 1 <M1»` is the simplex $(\texttt{MM2}, \eta_1\texttt{M1})$ and the mysterious object `<CPR * * * *>` is the base point of Y, each projection on X being the non-degenerate base point of X: the stars in position 1 and 3 mean no degeneracy, and the stars in position 2 and 4 denote the base point of X.

It is an effective simplicial set, so that its chain complex and its homology groups can be computed:

```
> (cc-homology-gen ccp2r2 3)  ==>
Computing boundary-matrix in dimension 3.
Rank of the source-module : 12.
[... Lines deleted ...]
Computing boundary-matrix in dimension 4.
Rank of the source-module : 6.
[... Lines deleted ...]
Homology in dimension 3 :
Component Z/2Z
Generator :
1     *  <CPR 0 <MM2> 2-1 <M1>>
1     *  <CPR 2 <MM2> 1-0 <M1>>
1     *  <CPR 1 <MM2> 2 <MM2>>
1     *  <CPR 2 <MM2> 0 <MM2>>
-1    *  <CPR 2 <MM2> 1 <MM2>>
---done---
```

The function `cc-homology-gen` is analogous to `cc-homology`, but a generator for each component of the homology is also given. Verify the indicated generator is a cycle.

The Eilenberg-Zilber reduction between $C_*(Y)$ and $C_*(X) \otimes C_*(X)$ can be also computed:

```
> (setf rdc (eilenberg-zilber p2r p2r))  ==>
[RDC-0]
> (dotimes (i 5) (print (cbs (rdc-tcc rdc) i)))  ==>
(<CPR * * * *>)
[... Lines deleted ...]
 <CPR 2-1 <MM2> 3-0 <MM2>> <CPR 3-1 <MM2> 2-0 <MM2>> <CPR 3-2 <MM2> 1-0 <MM2>>)
NIL
> (dotimes (i 5) (print (cbs (rdc-bcc rdc) i)))  ==>
(<TPR * *>)
(<TPR <M1> *> <TPR * <M1>>)
(<TPR <MM2> *> <TPR <M1> <M1>> <TPR * <MM2>>)
(<TPR <MM2> <M1>> <TPR <M1> <MM2>>)
(<TPR <MM2> <MM2>>)
NIL
```

The bottom chain complex has less generators, because it corresponds to a bi-simplicial triangulation of $(P^2\mathbb{R})^2$; of course the third homology group should be the same:

```
> (cc-homology-gen (rdc-bcc rdc) 3)  ==>
Computing boundary-matrix in dimension 3.
Rank of the source-module : 2.
[... Lines deleted ...]
Homology in dimension 3 :
Component Z/2Z
Generator :
1     *  <TPR <MM2> <M1>>
1     *  <TPR <M1> <MM2>>
---done---
```

and the Eilenberg-MacLane map will give six 3-simplices corresponding to these 2 prisms, organized as another generator of the same homology:

```
> (??? (rdc-g rdc) (cmb 3 1 (tpr 2 '<MM2> 1 '<M1>)
                          1 (tpr 1 '<M1> 2 '<MM2>)))  ==>
-------------------------------{CMB 3}
<MNM 1 * <CPR 2-1 <M1> 0 <MM2>>>
<MNM -1 * <CPR 2-0 <M1> 1 <MM2>>>
<MNM 1 * <CPR 1-0 <M1> 2 <MM2>>>
<MNM 1 * <CPR 2 <MM2> 1-0 <M1>>>
<MNM -1 * <CPR 1 <MM2> 2-0 <M1>>>
<MNM 1 * <CPR 0 <MM2> 2-1 <M1>>>
--------------------------------------
```

Both generators are certainly homologous. Exercise: use the EAT program to find a 4-chain the boundary of which is the difference between these generators. The (unique) solution is:

```
------------------------------{CMB 4}
<MNM -1 * <CPR 3-1 <MM2> 2-0 <MM2>>>
<MNM 1 * <CPR 2-1 <MM2> 3-0 <MM2>>>
--------------------------------------
```

The label *Constructive* Algebraic Topology is appropriate.

14.3 The twisted Eilenberg-Zilber theorem.

Theorem 43 — *An algorithm can be constructed:*

$$\mathbf{TEZ} : \mathcal{F} \longrightarrow \mathcal{R}.$$

If Φ is a fibration, the algorithm **TEZ** *working on the input $\Phi = (B, F, G, \tau, E)$ constructs a reduction ρ from the chain complex of the total space $C_*(E) = C_*(F \times_\tau B)$, the big complex, on a "twisted tensor product" $C_*(F) \otimes_t C_*(B)$, the small complex, also constructed by* **TEZ**.

Proof. The ordinary (non-twisted) Eilenber-Zilber theorem gives a reduction between the non-twisted cartesian and tensor products, the torsion operator being null. But we must take account of the torsion τ; this torsion does not change the underlying top graded module, only the differential is modified: the 0-face operator is twisted. The basic perturbation lemma may be applied if the nilpotency condition is satisfied.

If (f, b) is a simplex of E, the component b has a unique form $b = \eta b'$ where b' is non-degenerate and η is a multy-degeneracy operator; if b is non-degenerate then $b' = b$ and η is the identity, no degeneracy at all. Following Serre, the filtration degree of (f, b) is the dimension of b', the "base dimension". The Shih homotopy operator of Eilenberg-Zilber is natural, and when it works on (f, b) it is equal to the one which is defined on $F \times b'$, just above the simplex b'; therefore the Shih operator does not increase the filtration degree.

On the contrary the perturbation $\widehat{\delta}(f, b) = (\partial_0 f . \tau(b), \partial_0 b) - (\partial_0 f, \partial_0 b)$ has a filtration degree smaller than the filtration degree of (f, b). If b is non-degenerate, it is obvious. If b is degenerate and if the η in the expression $b = \eta b'$ does not contain a η_0, then $\partial_0 \eta b' = \eta' \partial_0 b'$, because of the commuting relation $\partial_0 \eta_i = \eta_{i-1} \partial_0$ if $i > 0$; the filtration degree of $(f', \partial_0 b)$ is again less than the one of (f, b). Finally, if the multi-degeneracy operator η contains a η_0, then $\tau(b)$ is trivial and the perturbation is null.

The basic perturbation lemma is then applied and produces the wished reduction. ■

The following technical proposition is the key point allowing one to use the twisted Eilenberg-Zilber theorem to obtain versions with effective homology of Serre and Eilenberg-Moore spectral sequences.

Proposition 44 — *Let $\Phi = (B, F, G, \tau, E)$ be a fibration. Let $\rho : C_*(F \times B) \Rightarrow C_*(F) \otimes C_*(B)$ (resp. $\rho' : C_*(F \times_\tau B) \Rightarrow C_*(F) \otimes_t C_*(B)$) be the non-twisted (resp. twisted) reduction given by the Eilenberg-Zilber (resp. twisted Eilenberg-Zilber) theorem. Let d (resp. d') the differential of $C_*(F) \otimes C_*(B)$ (resp. $C_*(F) \otimes_t C_*(B)$) and let $\delta = d' - d$ the bottom differential perturbation computed by the twisted Eilenberg-Zilber theorem. Then, if B is 1-reduced, the bottom perturbation δ decreases the filtration degree at least by 2.*

That is, if b (resp. f) is a p-simplex (resp. q-simplex) of B (resp. F), then:

$$\delta(f \otimes b) = \sum_{r=2}^{p} \delta_r(f \otimes b)$$

where $\delta_r(f \otimes b) \in C_{q+r-1}(F) \otimes C_{p-r}(B)$. Note it is not possible to coherently choose one of both possible notations $(f \otimes b)$ and $(f \otimes_t b)$: in fact $\delta = d' - d$ and d (resp. d') is to be applied to $(f \otimes b)$ (resp. $(f \otimes_t b)$).

Proof. Let $\rho = (AW, EML, SH)$ the ordinary Eilenberg-Zilber reduction between $C_*(F \times B)$ and $C_*(F) \otimes C_*(B)$. If $\widehat{\delta} = \widehat{d}' - \widehat{d}$ is the top perturbation, the explicit formula for the bottom perturbation in the proof of Theorem 29 gives:

$$\delta(f \otimes b) = (AW \circ (\sum_{i=0}^{\infty} (-1)^i (\widehat{\delta} \circ SH)^i) \circ \widehat{\delta} \circ EML)(f \otimes b).$$

We have observed in the previous proof the top perturbation $\widehat{\delta}$ decreasess the filtration degree at least by 1; furthermore, the Shih operator does not increase this filtration degree; therefore, the components with $i \geq 1$ in the expression just above satisfy the wished condition. The main work concerns only the $i = 0$ component.

The Eilenberg-MacLane operator working on $f \otimes b$ (f a non-degenerate q-simplex of F, b a non-degenerate p-simplex of B) produces a set of terms of the form $\pm(\eta f, \eta' b)$ for some multi-degeneracy operators η and η'. If η' contains a η_0, then the corresponding torsion is null and there is no perturbation. We can organize the other terms as follows: $\pm(\eta f, \eta' \eta'' b)$ where η contains a η_0, η'' is a composition of consecutive degeneracies $\eta'' = \eta_k \eta_{k-1} \dots \eta_2 \eta_1 = \eta_1^k$, and η' is another composition $\eta' = \eta_{i_\ell} \dots \eta_{i_1}$ with $i_1 \geq k+2$ and $k + \ell = q$; the integer $k+1$ is the first missing index in the degeneracies of the second component. We have then the expression:

$$(\widehat{\delta} \circ EML)(f \otimes b) = \sum \pm[(\partial_0 \eta f. \tau(\eta' \eta'' b), \eta'_{-1} \eta''_{-1} \partial_0 b) - (\partial_0 \eta f, \eta'_{-1} \eta''_{-1} \partial_0 b)].$$

In the expression above, a term η'_{-1} denotes the multi-degeneracy operator η' where all the indices have been replaced by the same minus one; in particular $\eta''_{-1} = \eta_{k-1} \dots \eta_0$. There remains to apply the Alexander-Whitney operator:

$$AW(f', b') = \sum_{j=0}^{p+q-1} \partial_{j+1}^{p+q-1-j} f' \otimes \partial_0^j b'.$$

If $j > k$, then there is at least two operators ∂_0 which remain alive in the right component; this comes from the relation $\partial_0^j \eta_{k-1} \dots \eta_0 = \partial_0^{j-k}$. In such a case, the term becomes something like $\pm(\dots, \eta''' \partial_0^m b)$ with $m \geq 2$, and the result is obtained.

If $j \leq k$, the torsion modifier $\tau(\eta'\eta''b)$ becomes by Alexander-whitney $\tau(\partial_{j+2}^{p+q-1-j}\eta'\eta''b)$, because the face index is increased by one when entered inside the τ argument. On one hand the inequality $p+q-1-j \geq p+q-1-k = p-1+\ell$ is satisfied; on the other hand all the indices $i_\ell, \dots, i_1$ are greater than $k+1 \geq j+1$, so that the following relation is satisfied:

$$\partial_{j+2}^{p+q-1-j}\eta'\eta'' = \partial_{j+2}^{p-1+k-j}\eta''.$$

But we have also the relation:

$$\partial_{j+2}^{p-1+k-j}\eta_k \dots \eta_1 b = \eta_j \dots \eta_1 \partial_2^{p-1} b;$$

finally, the p-simplex b gives a 1-simplex $\partial_2^{p-1}b$, necessarily degenerate because the base space B is 1-reduced; the corresponding tosion is trivial and the associated bottom perturbation is null. ■

The previous demonstration is heavier than the original ones [12, 45] (see also [25]). In these demonstrations, a more convenient demonstration is based on the notion of *twisting cochain*, giving an interesting conceptual structure for the twisted tensor product $C_*(X) \otimes_t C_*(Y)$. Once this structure is installed, a proof of Proposition 44 is direct. In fact some further properties are needed to obtain such a cochain, which were *not satisfied* in the first version of the EAT program (cf. the "story" of Section 1.13.1), and this is the reason why our program contained a "theoretical" bug. But the more general demonstration given just above proves our program was in fact correct, and the right results then obtained are so explained.

15 The SERRE$_{EH}$ algorithm.

Let $F \hookrightarrow E \to B$ a simplicial fibration. The Serre spectral sequence gives a set of relations between the homology groups of F, E and B. In some particular cases, this spectral sequence gives a method allowing you to deduce the homology groups of one of the components, E for example, when the homology groups of the others (B and F) are given. But in the general case, the Serre spectral sequence *is not* an algorithm; see for example [33, pp 6 and 28] for detailed explanations on this question. Using both our main tools, namely functional programming and the basic perturbation lemma, we construct in this section an algorithmic version of the Serre spectral sequence giving from a base space and a fiber space with effective homology a total space with effective homology.

Theorem 45 — *An algorithm* $\mathbf{SERRE}_{EH}$ *can be constructed:*

$$\mathbf{SERRE}_{EH} : [\mathcal{F} \times \mathcal{SS}^1_{EH} \times \mathcal{SS}^0_{EH}]_\chi \longrightarrow \mathcal{SS}^0_{EH}$$

where $[\ldots]_\chi$ *is the set of* coherent *triples* (Φ, B_{EH}, F_{EH})*, that is those triples such that the underlying simplicial set of* B_{EH} *is the 1-reduced base space of the fibration* Φ *and the underlying reduced simplicial set of* F_{EH} *is the fiber space of the same fibration. The output of the algorithm working on coherent data is a version with effective homology* E_{EH} *of the total space* E.

In other words, you can *compute* the homology groups of the total space E, no mysterious unreachable differential, no extension problem at abutment; see [33, pp 6 and 28]. More important, if the total space E is one of the elements of a new "reasonable" construction, the object E_{EH} can again be used to obtain a version with effective homology of the new constructed object, and so on.

PROOF. If the first component of the given object (Φ, B_{EH}, F_{EH}) is the fibration $\Phi = (B, F, G, \tau, E)$, we must construct a homotopy equivalence ε between $C_*(E) = C_*(F \times_\tau B)$ and some effective chain complex. We construct ε as the composition of two homotopy equivalences ε' and ε''.

The first one is produced by the twisted Eilenberg-Zilber theorem:

$$\varepsilon' = \{C_*(F \times_\tau B) = C_*(F \times_\tau B) \Rightarrow C_*(F) \otimes_t C_*(B)\}$$

where the left reduction is trivial. When ε' and in particular $C_*(F) \otimes_t C_*(B)$ are constructed, then we can construct the second necessary homotopy equivalence ε'', by applying the basic perturbation lemma to the difference between $C_*(F) \otimes_t C_*(B)$ and $C_*(F) \otimes C_*(B)$. Two homotopy equivalences are available:

$$\begin{aligned} \varepsilon_F &= \{C_*(F) \Leftarrow \widehat{C}_F \Rightarrow EC_F\} \\ \varepsilon_B &= \{C_*(B) \Leftarrow \widehat{C}_B \Rightarrow EC_B\} \end{aligned}$$

and we can construct their (non-twisted) tensor product (Proposition 38):

$$\varepsilon_{FB} = \{C_*(F) \otimes C_*(B) \Leftarrow \widehat{C}_F \otimes \widehat{C}_B \Rightarrow EC_F \otimes EC_B.\}$$

A *filtration degree* is defined on the three tensor products according to the degree with respect the second factor $C_*(B)$, $\widehat{C}_B$ or EC_B. Let us introduce on the bottom left-hand chain complex of this homotopy equivalence the necessary perturbation to obtain the twisted tensor product $C_*(F) \otimes_t C_*(B)$; the base space B is 1-reduced and according to Proposition 44, this perturbation decreases the filtration degree at least by 2.

The left reduction of ε_{FB} describes the bottom left-hand chain complex as a subcomplex of the top chain complex $\widehat{C}_F \otimes \widehat{C}_B$, so that the perturbation can be transferred to this top chain complex with the same property about the filtration degree. The homotopical component of the right reduction of ε'' increases the filtration degree at most by one. The basic perturbation lemma can therefore be applied to the right reduction and the perturbation obtained for the top chain complex and the result is obtained.

15.1 EAT implementation.

This version with effective homology of the Serre spectral sequence is not yet implemented in the EAT program. Because we were obviously interested by situations where it is possible with our methods to reach still unknown homology groups, we focused our work on a more complex spectral sequence, the Eilenberg-Moore spectral sequence, see the following sections. It is extremely easy to implement our version with effective homology of the Serre spectral sequence, and if ever you are actually interested by a computation needing it, please contact us.

16 The ELMR$_{EH}$ algorithm.

17 The LPSP$_{EH}$ algorithm.

18 The practical organization of LPSP$_{EH}$ in the EAT program.

19 Results obtained by LPSP$_{EH}$

20 The ELMR$'_{EH}$ algorithm.

21 The CARTAN$_{EH}$ algorithm.

22 The WHTH$_{EH}$ algorithm.

23 App. The root of the Basic Perturbation Lemma.

24 App. Programming the Basic Perturbation Lemma.

References

[1] J. Frank Adams. *On the Cobar construction.* Proceedings of the National Academy of Science of the U.S.A., 1956, vol. 42, pp 409-412.

[2] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[3] John Allen. *Anatomy of Lisp.* McGraw-Hill, 1978.

[4] David J. Anick. *The computation of rational homotopy groups is #P-hard.* In *Computers in geometry and topology*, Martin Tangora ed., Lecture Notes in Pure and Applied mathematics, vol. 114; Decker, New-York, 1989.

[5] David Anick. *Differential algebras in topology.* Research Notes in Mathematics (Boston, Mass.). vol. 3, 1993.

[6] Achim Bachem, Ravindran Kannan. *Applications of polynomial Smith normal form calculations.* In *Numerische Methoden bei graphentheoretischen und*

kombinatorischen Problemen, Bd. 2, Tag. Oberwolfach 1978, ISNM Vol. 46, 9-21 (1979).

[7] D. W. Barnes, L. A. Lambe. *Fixed point approach to homological perturbation theory.* Proceedings of the American Mathematical Society, 1991, vol. 112, pp 881-892.

[8] Hans J. Baues. *Geometry of loop spaces and the cobar construction.* Memoirs of the American Mathematical Society, 1980, vol. 230.

[9] Hans J. Baues. *The double bar and cobar constructions.* Compositio Mathematica, 1981, vol. 43, pp 331-341.

[10] William Browder. *Homology operations and loop spaces.* Illinois Journal of Mathematics, 1960, vol. 4, pp 347-357.

[11] Edgar H. Brown Jr.. *Finite computability of Postnikov complexes.* Annals of Mathematics, 1957, vol. 65, pp 1-20.

[12] Edgar H. Brown Jr.. *Twisted tensor products, I.* Annals of Mathematics, 1959, vol. 69, pp 223-246.

[13] Ronnie Brown. *The twisted Eilenberg-Zilber theorem.* Celebrazioni Arch. Secolo XX, Simp. Top., 1967, pp 34-37.

[14] Gunnar Carlsson and R. James Milgram. *Stable homotopy and iterated loop spaces.* in [28], pp 505-583.

[15] Henri Cartan. *Algèbres d'Eilenberg-MacLane.* in *Œuvres*, Springer, 1979.

[16] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[17] Frederick R. Cohen, Thomas J. Lada and J. Peter May. *The homology of iterated loop spaces.* Lecture Notes in Mathematics, vol. 533, Springer-Verlag, 1976.

[18] Samuel Eilenberg, Saunders MacLane. *On the groups $H(\pi, n)$, I.* Annals of Mathematics, 1953, vol. 58, pp 55-106.

[19] Samuel Eilenberg, Saunders MacLane. *On the groups $H(\pi, n)$, II.* Annals of Mathematics, 1954, vol. 60, pp 49-139.

[20] Samuel Eilenberg, John C. Moore. *Homology and fibrations, I. Coalgebras, cotensor product and its derived functors.* Commentarii Mathematici Helvetici, 1966, vol. 40, pp 199-236.

[21] Franz Inc.. *Common-Lisp, The Reference.* Addison-Wesley, 1988.

[22] `http://www.franz.com`

[23] Peter Gabriel, Michel Zisman. *Calculus of fractions and homotopy theory.* Ergebnisse der Mathematik, vol. 35, Springer-Verlag, 1967.

[24] Michael R. Garey, David S. Johnson. *Computers and intractability, a guide to the theory of NP-completeness.* W. H. Freeman and Company, New-York, 1979.

[25] V.K.A.M. Gugenheim. *On the chain-complex of a fibration.* Illinois Journal of Mathematics, 1972, vol. 16, pp 398-414.

[26] V.K.A.M. Gugenheim. *On a perturbation theory for the homology of the loop space.* Journal of Pure and Applied Algebra, 1982, vol. 25, pp 197-205.

[27] Andrew Hodges. *Alan Turing: the enigma of intelligence.* Burnett Books Limited, 1983.

[28] *Handbook of Algebraic Topology* (Edited by I.M. James). North-Holland (1995).

[29] Daniel M. Kan. *A combinatorial definition of homotopy groups.* Commentarii Mathematici Helvetici, 1958, vol. 67, pp 282-312.

[30] Saunders MacLane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[31] Saunders MacLane. *Homology.* Springer-Verlag, 1975.

[32] J. Peter May. *Simplicial objects in algebraic topology.* Van Nostrand, 1967.

[33] John McCleary. *User's guide to spectral sequences.* Publish or Perish, Wilmington DE, 1985.

[34] R. James Milgram. *Iterated loop spaces.* Annals of Mathematics, 1966, vol. 84, pp 386-403.

[35] Alain Prouté. *Sur la transformation d'Eilenberg-MacLane.* Comptes-Rendus de l'Académie des Sciences de Paris, 1983, vol. 297, pp 193-194.

[36] Michael O. Rabin. *Recursive unsolvability of group theoretic problems.* Annals of Mathematics, 1957, vol. 67, pp 172-194.

[37] G. Revesz. *Lambda Calculus, combinators and functional programming.* Cambridge University Press, 1988.

[38] Julio Rubio, Francis Sergeraert. *Supports Acycliques et Algorithmique*, in *Algorithmique, Topologie et Géométrie Algébriques*, Astérisque, 1990, vol.192.

[39] Julio Rubio, Francis Sergeraert, Yvon Siret. *The EAT program.* `ftp://www-fourier.ujf-grenoble.fr/~ftp/pub/EAT`

[40] Rolf Schön. *Effective algebraic topology.* Memoirs of the American Mathematical Society, 1991, vol. 451.

[41] Francis Sergeraert. *The computability problem in algebraic topology.* Advances in Mathematics, 1994, vol. 104, pp 1-29.

[42] Francis Sergeraert.
`http://www-fourier.ujf-grenoble.fr/~sergerar/`

[43] Jean-Pierre Serre. *Homologie singulière des espaces fibrés. Applications.* Annals of Mathematics, 1951, vol. 54, pp. 425-505.

[44] Jean-Pierre Serre. *Groupes d'homotopie et classes de groupes abéliens.* Annals of Mathematics, 1953, vol. 58, pp. 258-294.

[45] Weishu Shih. *Homologie des espaces fibrés.* Publications Mathématiques de l'I.H.E.S., 1962, vol. 13.

[46] V.A. Smirnov. *On the chain complex of an iterated loop space.* Mathematics of the USSR, Izvestiya, 1990, vol. 35, pp 445-455.

[47] Justin R. Smith. *Iterating the cobar construction.* Memoirs of the American Mathematical Society, 1994, vol. 524.

[48] Edwin H. Spanier. *Algebraic Topology.* McGraw Hill, 1966.

[49] Guy L. Steele Jr.. *Common Lisp, the language.* Digital Press, 1990.

[50] R. H. Szczarba. *The homology of twisted cartesian products.* Transactions of the American Mathematical Society, 1961, vol. 100, pp 197-216.

[51] George J. Tourlakis. *Computability.* Prentice-Hall, 1984.

[52] A.S. Troelstra, D. van Dalen. *Constructivism in mathematics, an introduction.* North-Holland, 1988.

[53] George W. Whitehead. *Fifty years of homotopy theory.* Bulletin of the American Mathematical Society, 1983, vol. 8, pp. 1-29.