Gestión mecanizada del conocimiento matemático en Topología Algebraica

Jónathan Heras Vicente

Memoria presentada para la obtención del grado de Doctor

Directores: Dra. Dña. Vico Pascual Martínez-Losa
Dr. D. Julio Rubio García

Universidad de La Rioja
Departamento de Matemáticas y Computación

Logroño, marzo de 2011

Mathematical Knowledge Management in Algebraic Topology

Jónathan Heras Vicente

Dissertation submitted for the degree
of Doctor of Philosophy

Supervisors: Dr. Vico Pascual Martínez-Losa
Dr. Julio Rubio García

Universidad de La Rioja
Departamento de Matemáticas y Computación

Logroño, March 2011

This work has been partially supported by an FPI grant from the Comunidad Autónoma de La Rioja and projects MTM2009-13842-C02-01 and MTM2010-09087-E from the Spanish Ministerio de Educación y Ciencia, European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath), project API08-08, contract COLABORA 2007/16 and grants ATUR10/18, ATUR09/14, ABOV10-51, ABOV09-50 from the Universidad de La Rioja.

Acknowledgments

To Vico and Julio
To my parents

By means of these lines I would like to thank my supervisors, Vico Pascual and Julio Rubio; without them it is clear that this work would not have been possible. Thank you for hearing me all the times I need it, for your wise ideas, your understanding, patience and confidence. Words are not enough to express all my gratitude.

I also wish to thank the support of all the members of my research group, Jesús María Aransay, César Domínguez, Francisco García, Arturo Jaime, Laureano Lambán, Eloy Mata, Juan José Olarte, Beatriz Pérez, María Poza, Ángel Luis Rubio and Eduardo Sáenz de Cabezón, who have welcomed me as one of them from the very first moment. I must name in particular Ana Romero, who heartened me in the hard times. I cannot forget Mirian Andrés, colleague and friend who went too soon and only knew the beginnings of this work.

I am grateful too to the Department of Mathematics and Computer Science of the University of La Rioja, for the technical support, and all their members, for the kindness that I have always received from all of them. I would like to name specially Luis Javier Hernández and M$^{\text{a}}$ Teresa Rivas, who helped me in my first steps in the research world, and also Clara Jiménez, who suffered me during this time.

Last but not least, thanks to my family and friends, who always trusted me, even in the moments when I did not. And a special thanks to my parents who, among other things, helped me in the bad times.

To all of you, thanks.

Contents

Introduction

The interest in Mechanized Mathematics appeared at the beginning of Computer Science, or even before: see the work of Alan Turing [Tur36], one of the fathers of Computer Science. Leaving scientific applications of numerical analysis (which gave rise to the first computers) far behind, from the fifties Symbolic Computation started to be developed with the aim of reproducing the mathematicians way of working, with the help of computers. Symbolic Computation has two branches: Computer Algebra and Mechanized Reasoning based on computational logic. For the early history of Computer Algebra, see the reference [DST93]. Related to both Mechanized Reasoning and proof assistant tools we can quote the influential GPS (General Problem Solver) of Newell and Simon [NSS59], whose first version appeared around 1955, linked to the birth of Artificial Intelligence.

These two lines of symbolic manipulation (Computer Algebra and Mechanized Reasoning) underwent a parallel development in the next 40 years, without too much relation between them. However, the scene radically changed 15 years ago, when the Calculemus initiative [Cal], funded by European projects, set out the challenge of integrating both Computer Algebra (computation) and Mechanized Reasoning (deduction) systems. In the same line, the MAP (Mathematics, Algorithms and Proofs) network [MAP], founded in 2003, works in the foundations of that integration mainly based on *constructive mathematics*.

Moreover, at the same time, with the success of Internet technologies, several options appeared to make feasible (symbolic) Mathematics on the Net. Perhaps, the most solid contribution was the one of the initiative IAMC (Internet Accessible Mathematical Computation) [IAM]. From this point on, several architectures (such as "buses" [CH97], [MATd], "brokers" [Mata], [Matb], generic servers [Matc], or based on web services [DSW05], [SSW04])) and design patterns [Dus06] have been proposed, culminating in both the MoNET (Mathematics on the Net) paradigm [Mon] and the Mathematics on semantic web [CDDP04]. The specification of XML standards, namely MathML [A$^+$08] and OpenMath [Con04], has been instrumental in all the quoted contributions, since they provide communication protocols to transfer mathematical knowledge. A successful initiative, but independent of these standards, is SAGE [Ste]; an alternative to currently dominant paradigms.

From the confluence of these lines (the one related to the integration of computation and deduction, and the one focused on the Mathematics on the Net), a community devoted to Mathematical Knowledge Management (MKM) [MKM] emerged in 2003. Its final goal is the development of integral assistants for Mathematics including computation, deduction, internet access and powerful user interfaces able to make the daily work of mathematical researchers easier, as well as influencing the strategies in mathematical teaching. Of course, this is a quite ambitious goal which retakes some of the classic Artificial Intelligence topics.

After this general introduction about the scope of this thesis, let us present now our application context: Algebraic Topology. This mathematical subject studies topological spaces by means of algebraic means, in particular through algebraic invariants (groups or rings, usually). This allows one to study interesting properties about topological spaces by means of statements about groups which are often easier to prove.

However, in spite of being an abstract mathematical subject, Algebraic Topology methods can be implemented in software systems and then applied to different contexts such as coding theory [Woo89], robotics [Mac03] or digital image analysis [GDMRSP05, GDR05] (in this last case, in particular in the study of medical images [SGF03]).

We can say that the work presented in this memoir tries to particularize the MKM work to the Algebraic Topology scope.

In this work, the Kenzo program [DRSS98], a Computer Algebra system devoted to research in Algebraic Topology, will be instrumental. This system has been mainly developed by Francis Sergeraert and is implemented in Common Lisp [Gra96]. The Kenzo system implements the Effective Homology method appeared in the eighties trying to make available real algorithms for the computation of homology and homotopy groups (two of the most important invariants in Algebraic Topology). Introduced by Francis Sergeraert in [Ser87] and [Ser94], the present state of this technique is described in [RS97] and [RS06].

From the year 2000 the work of the Programming and Symbolic Computation Team of University of La Rioja, supervised by Julio Rubio, has been, in some way, parallel to the one outlined for the general discipline. After a first stage where efforts were devoted both to increase and improve the algorithms and programs of Kenzo, a new research line was launched, without leaving the previous one, dedicated to apply Formal Methods of Software Engineering to the Kenzo system. Related to this research, relevant results were achieved about algebraic specification of Kenzo and, in general, of object oriented software systems (see, for instance, [LPR03] and [DLR07]). Deepening in this line, the Isabelle/HOL proof assistant was used to verify in [ABR08] a very important algorithm in Homological Algebra: the Basic Perturbation Lemma. In the same line, we can find the work of [DR11] where the Effective Homology of the Bicomplexes (another important result in Homological Algebra) was formalized in Coq, or the proof of the Normalization Theorem [LMMRRR10] in the ACL2 Theorem Prover. Regarding Algebraic Topology on the Net, a first approach was published in [APRR05], where a (partial) remote access to the Kenzo system was achieved using CORBA [Gro].

In this memoir the three previous research lines (*development of algorithms*, *formal-*

ization with proof assistant tools and *implementation of user interfaces* for Algebraic Topology) converge. To be more concrete, the goal of this work has consisted in developing an integral assistant for research in Algebraic Topology. The "integral" adjective means that the assistant not only provides a graphical interface for using the Kenzo kernel, but also guides the user in his interaction with the system, and, as far as possible, produces certificates about the correctness of the computations performed. As a by-product, several subgoals have been achieved. First of all, we have improved the usability and the accessibility of Kenzo, trying to increase in this way the number of users who can take profit of Kenzo. Moreover, we have also enhanced the computation capabilities of the Kenzo system providing, on the one hand, new Kenzo modules and, on the other hand, a connection with the GAP Computer Algebra system [GAP]. In addition, the goal of integrating computation (Kenzo) and deduction (ACL2 [KM]) is reached in this work by means of the formalization of Kenzo programs in the ACL2 Theorem Prover.

The rest of this memoir is organized as follows. The first chapter includes some preliminary notions and results that will be used in the rest of the work. Basic notions about Homological Algebra, Simplicial Topology and Effective Homology are presented in the first section of this chapter. The second part introduces the Kenzo Computer Algebra system, as well as some of its more important features. Finally, the ACL2 Theorem Prover is briefly presented.

After this first chapter, the memoir is split into three different parts. Chapters 2 and 3 are devoted to improve the usability and the accessibility of the Kenzo system. The computational capability of Kenzo is increased in Chapters 4 and 5 by means of the connection with other systems and the development of new Kenzo modules verified in ACL2. Chapter 6 is fully devoted to prove the correctness of some Kenzo programs by means of ACL2.

Chapter 2 presents an environment, called Kenzo framework, which provides a mediated access to the Kenzo system, constraining the Kenzo functionality, but providing guidance to the user in his navigation on the system. The architecture, the components and the execution flow of the Kenzo framework are described in this chapter. Moreover, at the end of the chapter, two ongoing works related to increase the computation capabilities of the Kenzo framework by means of ideas about remote and distributed computations are presented.

Once we have presented the Kenzo framework, we want to be able to increase the capabilities of the system by means of new Kenzo functionalities or the connection with other systems such as Computer Algebra systems or Theorem Prover tools but without modifying the kernel source code. Moreover, we realize that the interface of the Kenzo framework is not desirable for a human user, since it is based on XML [B^+08]; then, a more suitable way of interacting with the Kenzo framework should be provided. Chapter 3 is devoted to present how these problems have been handled. The first part of this chapter is focused on introducing a plug-in framework which allows us to extend the Kenzo framework without modifying the source code. A customizable graphical user interface which makes the interaction with the Kenzo framework easier is presented in the second part of this chapter. This graphical user interface improves the usability and the accessibility of the Kenzo system. The whole system, that is to say, the combination of the two frameworks and the front-end, is called *fKenzo*, an acronym for friendly Kenzo. The next two chapters focus on extending *fKenzo* by means of the connection with other systems and new Kenzo functionalities respectively.

Chapter 4 describes the integration of several systems in *fKenzo*. First of all, the Computer Algebra system GAP [GAP] is linked to *fKenzo*, allowing the computation of group homology. Moreover, the GAP system is not only used individually but also some of its programs are composed with the Kenzo system in order to obtain new tools, which increase the computational capabilities of *fKenzo*; this part of the memoir is an enhancement to a previous work [RER09]. The second part of this chapter is devoted to integrate the ACL2 Theorem Prover in *fKenzo*, achieving in this way the integration of computation and deduction in the same system. Likewise that in the GAP case, the ACL2 system is not only used individually but it is also combined with Kenzo to increase the reliability of the new programs of our system. From this chapter on, ACL2 will play a key role in our work.

Chapter 5 carries out a contribution to the algorithms and programs of the Kenzo system by means of new Kenzo modules which in turn extend the *fKenzo* system. First of all, a new Kenzo module in charge of working with simplicial complexes is presented. This module about simplicial complexes is the foundation to develop a setting to analyze monochromatic digital images. Namely, a new module which can be used to study digital images is explained in the second part of this chapter. Finally, the necessary algorithms and programs to build the effective homology of the pushout of simplicial sets are presented in the last section of the chapter. As an application of the pushout, we compute with Kenzo the first homotopy groups of the suspension of the classifying space of $SL_2(\mathbb{Z})$, the group of 2×2 matrices with determinant 1 over $\mathbb{Z}$, with the group operations of ordinary matrix multiplication and matrix inversion. Moreover, this chapter is not only devoted to present the new Kenzo modules but also to increase the reliability of the programs implemented in that modules by means of the ACL2 Theorem Prover.

Chapter 6 is devoted to prove the correctness of some programs implemented in the Kenzo system by means of ACL2. In the first part of this chapter, a set of ACL2 tools which will be fundamental onwards are presented. The second part is focussed on proving the correctness of the Kenzo simplicial set constructors which do not involve other spaces. Subsequently, an infrastructure for modeling mathematical structures in ACL2 is introduced. Eventually, a methodology to verify the correctness of the construction of Kenzo spaces from other ones applying topological constructors is presented in the last section of this chapter.

The memoir ends with a chapter which includes conclusions and further work, a glossary and the bibliography.

Chapter 1

Preliminaries

In this chapter we introduce some preliminaries that we will use in the rest of this memoir. The first section is devoted to the main mathematical notions employed in this work. The Kenzo system, a Common Lisp program developed by Francis Sergeraert and some coworkers devoted to perform computations in Algebraic Topology, is presented in Section 1.2. Finally, the *deduction* machinery employed in this memoir, the ACL2 system, is briefly introduced in Section 1.3.

1.1 Mathematical preliminaries

Algebraic Topology is a vast and complex subject, in particular mixing Algebra and (combinatorial) Topology. Algebraic Topology consists in trying to use as much as possible "algebraic" methods to attack topological problems. For instance, one can define some special groups associated with a topological space, in a way that respects the relation of homeomorphism of spaces. This allows us to study properties about topological spaces by means of statements about groups, which are often easier to prove.

1.1.1 Homological Algebra

The following basic definitions can be found, for instance, in [Mac63].

Definition 1.1. Let R be a ring with a unit element $1 \neq 0$. A *left R-module M* is an additive abelian group together with a map $p : R \times M \to M$, denoted by $p(r, m) \equiv rm$, such that for every $r, r' \in R$ and $m, m' \in M$

$$\begin{aligned}(r + r')m &= rm + r'm\\ r(m + m') &= rm + rm'\\ (rr')m &= r(r'm)\\ 1m &= m\end{aligned}$$

A similar definition is given for a *right R-module.*

If $R = \mathbb{Z}$ (the integer ring), a $\mathbb{Z}$-module M is simply an abelian group. The map $p : \mathbb{Z} \times M \to M$ is given by

$$p(n,m) = \begin{cases} m + \overset{n}{\cdots} + m & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ (-m) + \overset{-n}{\cdots} + (-m) & \text{if } n < 0 \end{cases}$$

Definition 1.2. Let R be a ring and M and N be R-modules. An *R-module morphism* $\alpha : M \to N$ is a function from M to N such that for every $m, m' \in M$ and $r \in R$

$$\begin{aligned} \alpha(m + m') &= \alpha(m) + \alpha(m') \\ \alpha(rm) &= r\alpha(m) \\ \alpha(0_M) &= 0_N \end{aligned}$$

Definition 1.3. Given a ring R, a *graded module* M is a family of left R-modules $(M_n)_{n\in\mathbb{Z}}$.

Definition 1.4. Given a pair of graded modules M and M', a *graded module morphism* f of degree k between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ such that $f_n : M_n \to M'_{n+k}$ for all $n \in \mathbb{Z}$.

Definition 1.5. Given a graded module M, a *differential* $(d_n)_{n\in\mathbb{Z}}$ is a family of module endomorphisms of M of degree -1 such that $d_{n-1} \circ d_n = 0$ for all $n \in \mathbb{Z}$.

From the previous definitions, the notion of chain complex can be defined. Chain complexes are the central notion in Homological Algebra and can be used as an algebraic means to study properties of topological spaces in several dimensions.

Definition 1.6. A *chain complex* C_* is a family of pairs $(C_n, d_n)_{n\in\mathbb{Z}}$ where $(C_n)_{n\in\mathbb{Z}}$ is a graded module and $(d_n)_{n\in\mathbb{Z}}$ is a differential of C_*.

The module C_n is called the module of *n-chains.* The image $B_n = \operatorname{Im} d_{n+1} \subseteq C_n$ is the (sub)module of *n-boundaries.* The kernel $Z_n = \operatorname{Ker} d_n \subseteq C_n$ is the (sub)module of *n-cycles.*

In many situations the ring R is the integer ring, $R = \mathbb{Z}$. In this case, a chain complex C_* is given by a graded abelian group $(C_n)_{n\in\mathbb{Z}}$ and a graded group morphism of degree -1, $(d_n : C_n \to C_{n-1})_{n\in\mathbb{Z}}$, satisfying $d_{n-1} \circ d_n = 0$ for all n. From now on in this memoir, we will work with $R = \mathbb{Z}$.

Let us present some examples of chain complexes.

Example 1.7.

- The *unit chain complex* has a unique non null component, namely a $\mathbb{Z}$-module in degree 0 generated by a unique generator and the differential is the null map.

- A chain complex to model the *circle* is defined as follows. This chain complex has two non null components, namely a $\mathbb{Z}$-module in degree 0 generated by a unique generator and a $\mathbb{Z}$-module in degree 1 generated by another generator; and the differential is the null map.
- The *diabolo* chain complex has associated three chain groups:
  - C_0, the free $\mathbb{Z}$-module on the set $\{s_0, s_1, s_2, s_3, s_4, s_5\}$.
  - C_1, the free $\mathbb{Z}$-module on the set $\{s_{01}, s_{02}, s_{12}, s_{23}, s_{34}, s_{35}, s_{45}\}$.
  - C_2, the free $\mathbb{Z}$-module on the set $\{s_{345}\}$.

  and the differential is provided by:
  - $d_0(s_i) = 0$,
  - $d_1(s_{ij}) = s_j - s_i$,
  - $d_2(s_{ijk}) = s_{jk} - s_{ik} + s_{ij}$.

  and it is extended by linearity to the combinations $c = \sum_{i=1}^{m} \lambda_i x_i \in C_n$ where $\lambda_i \in \mathbb{Z}$ and $x_i \in C_n$.

We can construct chain complexes from other ones, applying constructors such as the direct sum or the tensor product.

Definition 1.8. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes. The *direct sum* of C_* and D_* is the chain complex $C_* \oplus D_* = (M_n, d_n)_{n\in\mathbb{Z}}$ such that, $M_n = (C_n, D_n)$ and the differential map is defined on the generators (x, y) with $x \in C_n$ and $y \in D_n$ by $d_n((x, y)) = (d_{C_n}(x), d_{D_n}(y))$ for all $n \in \mathbb{Z}$.

Definition 1.9. Let M be a right R-module, and N a left R-module. The *tensor product* $M \otimes_R N$ is the abelian group generated by the symbols $m \otimes n$ for every $m \in M$ and $n \in N$, subject to the relations

$$\begin{aligned}(m + m') \otimes n &= m \otimes n + m' \otimes n\\ m \otimes (n + n') &= m \otimes n + m \otimes n'\\ mr \otimes n &= m \otimes rn\end{aligned}$$

for all $r \in R$, $m, m' \in M$, and $n, n' \in N$.

If $R = \mathbb{Z}$ (the integer ring), then M and N are abelian groups and their tensor product will be denoted simply by $M \otimes N$.

Definition 1.10. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes of right and left $\mathbb{Z}$-modules respectively. The *tensor product* $C_* \otimes D_*$ is the chain complex of $\mathbb{Z}$-modules $C_* \otimes D_* = ((C_* \otimes D_*)_n, d_n)_{n\in\mathbb{Z}}$ with

$$(C_* \otimes D_*)_n = \bigoplus_{p+q=n} (C_p \otimes D_q)$$

where the differential map is defined on the generators $x \otimes y$ with $x \in C_p$ and $y \in D_q$, according to the Koszul rule for the signs, by

$$d_n(x \otimes y) = d_{C_p}(x) \otimes y + (-1)^p x \otimes d_{D_q}(y)$$

Let us present now, one of the most important invariants used in Homological Algebra. Given a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$, the identities $d_{n-1} \circ d_n = 0$ mean the inclusion relations $B_n \subseteq Z_n$: every boundary is a cycle (the converse in general is not true). Thus the next definition makes sense.

Definition 1.11. Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex of R-modules. For each degree $n \in \mathbb{Z}$, the *n-homology module* of C_* is defined as the quotient module

$$H_n(C_*) = \frac{Z_n}{B_n}$$

It is worth noting that the homology groups of a space X are the ones of its associated chain complex $C_*(X)$; the way of constructing the chain complex associated with a space X is explained, for instance, in [Mau96]. In an intuitive sense, homology groups measure "n-dimensional holes" in topological spaces. H_0 measures the number of connected components of a space. The homology groups H_n measure higher dimensional connectedness. For instance, the n-sphere, S^n, has exactly one n-dimensional hole and no m-dimensional holes if $m \neq n$.

Moreover, it is worth noting that homology groups are an *invariant*, see [Mau96]. That is to say, if two topological spaces are homeomorphic, then their homology groups are isomorphic.

Let us finish this section with some additional definitions related to chain complexes.

Definition 1.12. A chain complex $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ is *acyclic* if $H_n(C_*) = 0$ for all n, that is to say, if $Z_n = B_n$ for every $n \in \mathbb{Z}$.

Definition 1.13. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be two chain complexes, a *chain complex morphism* between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ of degree 0 between $(C_n)_{n\in\mathbb{Z}}$ and $(D_n)_{n\in\mathbb{Z}}$ such that $d'_n \circ f_n = f_{n-1} \circ d_n$ for each $n \in \mathbb{Z}$.

Definition 1.14. Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex. A chain complex $D_* = (D_n\ ,\ d'_n)_{n\in\mathbb{Z}}$ is a *chain subcomplex* of C_* if

- D_n is a submodule of C_n, for all $n \in \mathbb{Z}$
- $d'_n = d_n \mid_{D_*}$

The condition $d'_n = d_n \mid_{D_*}$ means that the boundary operator of the chain subcomplex is just the differential operator of the larger chain complex restricted to its domain. We denote $D_* \subset C_*$ if D_* is a chain subcomplex of C_*.

Definition 1.15. A short exact sequence is a sequence of modules:

$$0 \leftarrow C'' \overset{j}{\leftarrow} C \overset{i}{\leftarrow} C' \leftarrow 0$$

which is exact, that is, the map i is injective, the map j is surjective and Im $i =$ Ker j.

From now on in this memoir, we will work with *non-negative chain complexes*, that is to say, $(C_n)_{n\in\mathbb{Z}}$ such that $C_n = 0$ if $n < 0$. A non-negative chain complex C_* will be denoted by $C_* = (C_n)_{n\in\mathbb{N}}$. Moreover, the chain complexes we work with are supposed to be free.

Definition 1.16. A chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ of $\mathbb{Z}$-modules is said to be *free* if C_n is a free $\mathbb{Z}$-module (a $\mathbb{Z}$-module which admits a basis) for each $n \in \mathbb{N}$.

1.1.2 Simplicial Topology

1.1.2.1 Simplicial Sets

Simplicial sets were first introduced by Eilenberg and Zilber [EZ50], who called them *semi-simplicial complexes.* They can be used to express some topological properties of spaces by means of combinatorial notions. A good reference for the definitions and results of this section is [May67].

Definition 1.17. A *simplicial set* K, is a union $K = \bigcup_{q\geq 0} K^q$, where the K^q are disjoints sets, together with functions:

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

subject to the relations:

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

The ∂_i^q and η_i^q are called *face* and *degeneracy* operators respectively.

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

An example of a simplicial set, that can be useful to clarify some notions, is the *standard simplicial set of dimension* m, $\Delta[m]$.

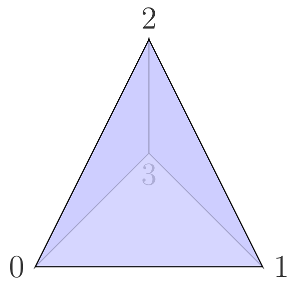


Figure 1.1: Non degenerate simplexes of the standard simplicial set $\Delta[3]$

Definition 1.18. For $m \geq 0$, the *standard simplicial set of dimension* m, $\Delta[m]$, is a simplicial set built as follows. An n-simplex of $\Delta[m]$ is any $(n+1)$-tuple $(a_0, \ldots, a_n)$ of integers such that $0 \leq a_0 \leq \cdots \leq a_n \leq m$, and the face and degeneracy operators are defined as

$$\begin{aligned}\partial_i(a_0, \ldots, a_n) &= (a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)\\ \eta_i(a_0, \ldots, a_n) &= (a_0, \ldots, a_{i-1}, a_i, a_i, a_{i+1} \ldots, a_n)\end{aligned}$$

In Definition 1.18 the super-indexes in the degeneracy and face maps have been skipped, since they can be inferred from the context. It is a usual practice and will be freely used in the sequel, both for degeneracy and for face maps.

Example 1.19. Figure 1.19 shows the non degenerate simplexes of the standard simplicial set $\Delta[3]$:

- 0-simplexes (vertices): (0), (1), (2), (3);
- non degenerate 1-simplexes (edges): (0 1), (0 2), (0 3), (1 2), (1 3), (2 3);
- non degenerate 2-simplexes (triangles): (0 1 2), (0 1 3), (0 2 3), (1 2 3); and
- non degenerate 3-simplex (filled tetrahedra): (0 1 2 3).

It is worth noting that there is not any non degenerate n-simplex with $n > 3$.

Once we have presented the non degenerate simplexes, let us introduce the behavior of the face and degeneracy maps. For instance, if we apply the face and degeneracy maps over the 2-simplex (0 1 2) (for the rest of simplexes is analogous) we will obtain:

$$\partial_i((0\ 1\ 2)) = \begin{cases} (1\ 2) & \text{if } i = 0\\ (0\ 2) & \text{if } i = 1\\ (0\ 1) & \text{if } i = 2 \end{cases}$$

$$\eta_i((0\ 1\ 2)) = \begin{cases} (0\ 0\ 1\ 2) & \text{if } i = 0\\ (0\ 1\ 1\ 2) & \text{if } i = 1\\ (0\ 1\ 2\ 2) & \text{if } i = 2 \end{cases}$$

Let us note that the face operator applied over the 2-simplex (0 1 2) produces simplexes with geometrical meaning (that are the three edges of the simplex (0 1 2)). On the contrary, the simplexes obtained from applying the degeneracy operator do not have any geometrical meaning.

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

The next essential result, which follows from the commuting properties of degeneracy maps in the definition of simplicial sets, provides a way to represent any simplex of a simplicial set in a unique manner.

Proposition 1.20. Let K be a simplicial set. Any n-simplex $x \in K^n$ can be expressed in a unique way as a (possibly) iterated degeneracy of a non degenerate simplex y in the following way:

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

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

This proposition allows us to encode all the elements (simplexes) of *any* simplicial set in a generic way, by means of a structure called *abstract simplex*. More concretely, an *abstract simplex* is a pair ($dgop$ $gmsm$) consisting of a sequence of degeneracy maps $dgop$ (which will be called a *degeneracy operator*) and a geometric simplex $gmsm$. The indexes in a degeneracy operator $dgop$ must be in a strictly decreasing order. For instance, if σ is a non degenerate simplex, and σ' is the degenerate simplex $\eta_1\eta_2\sigma$, the corresponding abstract simplexes are respectively $(\emptyset \ \ \sigma)$ and $(\eta_3\eta_1 \ \ \sigma)$, as $\eta_1\eta_2 = \eta_3\eta_1$, due to equality (2) in Definition 1.17.

In a similar way that we defined chain subcomplex we can define the notion of simplicial subcomplex.

Definition 1.21. Let $K = (K^n, \partial_i, \eta_i)_{n\in\mathbb{Z}}$ be a simplicial set. $L = (L^n)_{n\in\mathbb{Z}}$ is a *simplicial subcomplex* of K if

- $L^n \subset K^n$ for all $n \in \mathbb{Z}$
- $(L^n, \partial_i|_{L^n}, \eta_i|_{L^n})_{n\in\mathbb{Z}}$ is a simplicial set

Let us show now, other examples of simplicial sets that are (simplicial) models of well-known topological spaces. The following list comes from the simplicial sets that will appear frequently in the rest of the memoir.

Definition 1.22. For $m \geq 1$, the *sphere of dimension m*, S^m, is a simplicial set built as follows. There are just two geometric simplexes: a 0-simplex, let us denote it by $\star$, and an m-simplex, let us denote it by sm. The faces of sm are the degeneracies of $\star$; that is to say $\partial_i(sm) = \eta_{m-1}\eta_{m-2}\dots\eta_0\star$ for all $0 \leq i \leq m$.

Definition 1.23. Let $m_1, \dots, m_n$ natural numbers such that $m_i \geq 1$ for all $1 \leq i \leq n$, the *wedge of spheres of dimensions* $m_1, \dots, m_n$, $S^{m_1} \vee \dots \vee S^{m_n}$, is a simplicial set built as follows. It has the following geometric simplexes: in dimension 0 a 0-simplex, let us denote it by $\star$, and in dimension p as many simplexes as the number of m_j, $1 \leq j \leq n$, such that $m_j = p$. The faces of every p-simplex are the degeneracies of $\star$; that is to say, let x be a p-simplex, then $\partial_i(x) = \eta_{p-1}\eta_{p-2}\dots\eta_0\star$ for all $0 \leq i \leq p$.

Definition 1.24. For $n > 0$, $p > 1$ and $n > 2p - 4$, the *Moore space of dimensions* p, n, $M(\mathbb{Z}/p\mathbb{Z}, n)$, is a simplicial set built as follows. There are just three geometric simplexes: a 0-simplex, let us denote it by $\star$, an n-simplex, let us denote it by Mn, and an $n+1$-simplex, let us denote it by Mn'. The faces of Mn are the degeneracies of $\star$; moreover, p of the faces of Mn' are identified with Mn and the rest are the degeneracies of $\star$.

Definition 1.25. The *real projective plane*, $P^\infty\mathbb{R}$, is a simplicial set built as follows. In dimension n, this simplicial set has only one geometric simplex, namely the integer n. The faces of this non degenerate simplex n are given by the following formulas: $\partial_0 n = \partial_n n = n - 1$ and for $i \neq 0$ and $i \neq n$, $\partial_i n = \eta_{i-1}(n-2)$.

The *real projective plane n-dimensional*, $P^n\mathbb{R}$, is a simplicial set analogous to the model of $P^\infty\mathbb{R}$ but without simplexes in dimensions $m \geq n$.

We can also construct simplicial models of truncated real projective planes. Let $n > 1$, $P^\infty\mathbb{R}/P^{n-1}\mathbb{R}$ is a simplicial set analogous to the model of $P^\infty\mathbb{R}$ but without simplexes in dimensions $1 \leq m < n$. The faces of the n-simplex n are the degeneracies of the 0-simplex 0.

Let $n > 1$ and $l \geq n$, $P^l\mathbb{R}/P^{n-1}\mathbb{R}$ is a simplicial set analogous to the model of $P^l\mathbb{R}$ but without simplexes in dimensions $1 \leq m < n$.

The four above definitions are related to simplicial models of source topological spaces. Moreover, we also have models for topological constructors that are applied to some spaces to obtain new ones.

Definition 1.26. Given two simplicial sets K and L, the *Cartesian product* $K \times L$ is a simplicial set with n-simplexes:

$$(K \times L)^n = K^n \times L^n$$

and for all $(x, y) \in K^n \times L^n$ the face and degeneracy operators are defined as follows:

$$\begin{aligned} \partial_i(x, y) &= (\partial_i x, \partial_i y) \quad \text{for } 0 \leq i \leq n \\ \eta_i(x, y) &= (\eta_i x, \eta_i y) \quad \text{for } 0 \leq i \leq n \end{aligned}$$

Let K be a simplicial set and $\star \in K_0$ a chosen 0-simplex (called the *base point*). We will also denote by $\star$ the degenerate simplexes $\eta_{n-1}\dots\eta_0\star \in K_n$ for every n.

Definition 1.27. A simplicial set K is said to be *reduced* (or 0-*reduced*) if K has only one 0-simplex. Given $m \geq 1$, K is *m-reduced* if it is reduced and there are not any non degenerate simplex $\forall n, 0 < n \leq m$.

Definition 1.28. Given a reduced simplicial set X with 0-simplex k_0, the *suspension* $\Sigma(X)$ is a simplicial set defined as follows. $\Sigma(X)^0 = b_0$; $\Sigma(X)^n =$ consists of all symbols $(i, x), i \geq 1, x \in X^{n-i}$, modulo the identification $(i, k_n) = \eta_{n+i}\eta_{n+i-1} \dots \eta_n b_0 = b_{n+i}$, where $k_n = \eta_n \eta_{n-1} \dots \eta_0 k_0$. The face and degeneracy operators on $\Sigma(X)$ are defined by:

$$\begin{aligned}
\eta_i \dots \eta_0(j, x) &= (i + j, x)\\
\eta_{i+1}(1, x) &= (1, \eta_i x)\\
\partial_0(1, x) &= b_n, x \in X^n\\
\partial_1(1, x) &= b_0, x \in X^0\\
\partial_{i+1}(1, x) &= (1, \partial_i x), x \in X^n, n > 0
\end{aligned}$$

We define inductively $\Sigma^n(X) = \Sigma(\Sigma^{n-1}(X))$ for all $n \geq 1$, $\Sigma^0(X) = X$.

If we impose some conditions on the sets of simplicial sets we will obtain new kinds of simplicial objects.

Definition 1.29. An *(abelian) simplicial group* G is a simplicial set where each G^n is an (abelian, respectively) group and the face and degeneracy operators are group morphisms.

Let us present some examples of *simplicial groups*. For instance, the simplicial version of the loop space of a connected space X, that is to say the space of continuous functions from the circle S^1 to X, is given as follows.

Definition 1.30. Given a reduced simplicial set X, the *loop space* simplicial version of X, $G(X)$, is a simplicial group defined as follows: $G^n(X)$ is the free group generated by X^{n+1} with the relations $\{\eta_0 x; x \in X^n\}$. If e_n is the identity element of $G^n(X)$, we have the canonical application $\tau : X^{n+1} \to G^n(X)$ defined by:

$$\tau(y) = \begin{cases} e_n & \text{if it exists } x \in X^n \text{ such that } y = \eta_0 x,\\ y & \text{otherwise.}\end{cases}$$

If ∂, η are the face and degeneracy operators of X, we define the face and degeneracy operators, denoted by $\overline{\partial}, \overline{\eta}$, over the generators of $G^n(X)$ as follows:

$$\begin{aligned}
\overline{\partial_0}\tau(x) &= [\tau(\partial_0 x)]^{-1}\tau(\partial_1 x)\\
\overline{\partial_i}\tau(x) &= \tau(\partial_{i+1} x) \text{ if } i > 0\\
\overline{\eta_i}\tau(x) &= \tau(\eta_{i+1} x) \text{ if } i \geq 0
\end{aligned}$$

In the sequel, we will denote $G(X)$ as $\Omega(X)$. We define inductively $\Omega^n(X) = \Omega(\Omega^{n-1}(X))$ for all $n \geq 1$, $\Omega^0(X) = X$.

An important example of abelian simplicial groups is the model for an Eilenberg MacLane space.

Definition 1.31. An Eilenberg MacLane space of type (π, n) is a simplicial group K with base point $e_0 \in K^0$ such that $\pi_n(K) = \pi$ and $\pi_i(K) = 0$ if $i \neq n$ (where $\pi_n(K)$ is the *n-th homotopy group*, that is, the set of homotopy classes of continuous maps $f : S^n \to K$ that map a chosen base point $a \in S^n$ to the base point e_0, a more detailed description and results about homotopy groups can be found in [Hat02]). The simplicial group K is called a $K(\pi, n)$ if it is an Eilenberg MacLane space of type (π, n) and in addition it is minimal (see [May67]).

In order to construct the spaces $K(\pi, n)$'s several methods can be used, although the results are necessarily isomorphic [May67]. Let us consider the following one.

Let π be an abelian group. First of all, we construct an abelian simplicial group $K = K(\pi, 0)$ given by $K^n = \pi$ for all $n \geq 0$, and with face and degeneracy operators $\partial_i : K^n = \pi \to K^{n-1} = \pi$ and $\eta_i : K^n = \pi \to K^{n+1} = \pi$, $0 \leq i \leq n$, equal to the identity map of the group π. This abelian simplicial group is the Eilenberg MacLane space of type $(\pi, 0)$. For $n \geq 0$, we build recursively $K(\pi, n)$ by means of the classifying space constructor.

Definition 1.32. Let G be an abelian simplicial group. The *classifying space* of G, written $B(G)$, is the abelian simplicial group built as follows. The n-simplexes of $B(G)$ are the elements of the Cartesian product:

$$B(G)^n = G^{n-1} \times G^{n-2} \times \cdots \times G^0.$$

In this way $B(G)^0$ is the group which has only one element that we denote by $[\,]$. For $n \geq 1$, an element of $B(G)^n$ has the form $[g_{n-1}, \ldots, g_0]$ with $g_i \in G^i$. The face and degeneracy operators are given by

$$\begin{aligned}
\eta_0[\,] &= [e_0] \\
\partial_i[g_0] &= [\,], \quad i = 0, 1 \\
\partial_0[g_{n-1}, \ldots, g_0] &= [g_{n-2}, \ldots, g_0] \\
\partial_i[g_{n-1}, \ldots, g_0] &= [\partial_{i-1} g_{n-1}, \ldots, \partial_1 g_{n-i+1}, \partial_0 g_{n-i} + g_{n-i-1}, g_{n-i-2}, \ldots, g_0], \quad 0 < i \leq n \\
\eta_0[g_{n-1}, \ldots, g_0] &= [e_n, g_{n-1}, \ldots, g_0] \\
\eta_i[g_{n-1}, \ldots, g_0] &= [\eta_{i-1} g_{n-1}, \ldots, \eta_0 g_{n-i}, e_{n-i}, g_{n-i-1}, \ldots, g_0], \quad 0 < i \leq n
\end{aligned}$$

where e_n denotes the null element of the abelian group G^n.

We define inductively $B^n(G) = B(B^{n-1}(G))$ for all $n \geq 1$, $B^0(G) = G$.

Theorem 1.33. [May67] Let π be an abelian group and $K(\pi, 0)$ as explained before. Then $B^n(K)$ is a $K(\pi, n)$.

1.1.2.2 Link between Homological Algebra and Simplicial Topology

Up to now, we have given a brief introduction to Simplicial Topology and Homological Algebra; let us present now the link between these two subjects that will allow us to compute the homology groups of simplicial sets.

Definition 1.34. Let K be a simplicial set, we define the *chain complex associated with* K, $C_*(K) = (C_n(K), d_n)_{n\in\mathbb{N}}$, in the following way:

- $C_n(K) = \mathbb{Z}[K^n]$ is the free $\mathbb{Z}$-module generated by K^n. Therefore an n-chain $c \in C_n(K)$ is a combination $c = \sum_{i=1}^m \lambda_i x_i$ with $\lambda_i \in \mathbb{Z}$ and $x_i \in K^n$ for $1 \leq i \leq m$;
- the differential map $d_n : C_n(K) \to C_{n-1}(K)$ is given by

$$d_n(x) = \sum_{i=0}^{n} (-1)^i \partial_i(x) \text{ for } x \in K^n$$

  and it is extended by linearity to the combinations $c = \sum_{i=1}^m \lambda_i x_i \in C_n(K)$.

The following statement is an immediate consequence of the previous definition.

Proposition 1.35. Let X, Y simplicial sets, where X is a simplicial subcomplex of Y, then $C_*(X)$ is a chain subcomplex of $C_*(Y)$.

From the link between simplicial sets and chain complexes we can define the homology groups of a simplicial set as follows.

Definition 1.36. Given a simplicial set K, the *n-homology group* of K, $H_n(K)$, is the n-homology group of the chain complex $C_*(K)$:

$$H_n(K) = H_n(C_*(K))$$

To sum up, when we want to study properties of a compact topological space which admits a triangulation, we can proceed as follows. We can associate a simplicial model K with a topological space X which admits a triangulation. Subsequently, the chain complex $C_*(K)$ associated with K can be constructed. Afterwards, properties of this chain complex are computed; for instance, homology groups. Eventually, we can interpret the properties of the chain complex as properties of the topological space X.

1.1.3 Effective Homology

As we have seen, a central problem in our context consists in computing *homology groups* of topological spaces. By definition, the homology group of a space X is the one of its associated chain complex $C_*(X)$. If $C_*(X)$ can be described as a graded free abelian group with a *finite* number of generators at each degree; then, computing each homology group can be translated to a problem of diagonalizing certain integer matrices (see [Veb31]). So, we can assert that homology groups are computable in this finite type case.

However, things are more interesting when a space X is not of finite type (in the previously invoked sense) but it is known that its homology groups *are* of finite type. Then, it is natural to study if these homology groups are computable. The *effective homology method*, introduced in [Ser87] and [Ser94], provides a framework where this computability question can be handled.

In this subsection, we present some definitions and fundamental results about the effective homology method. More details can be found in [RS02] and [RS06].

In the context of effective homology, we can distinguish two different kinds of objects: *effective* and *locally effective* objects. In particular, we are going to focus on *effective* and *locally effective* chain complexes.

Definition 1.37. An *effective chain complex* is a *free* chain complex of $\mathbb{Z}$-modules $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ where each group C_n is finitely generated and:

- a provided algorithm returns a (distinguished) $\mathbb{Z}$-basis in each degree n, and
- a provided algorithm returns the differential maps d_n.

If a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ is effective, the differential maps $d_n : C_n \to C_{n-1}$ can be expressed as finite integer matrices, and then it is possible to know *everything* about C_*: we can compute the subgroups $\operatorname{Ker} d_n$ and $\operatorname{Im} d_{n+1}$, we can determine whether an n-chain $c \in C_n$ is a cycle or a boundary, and in the last case, we can obtain $z \in C_{n+1}$ such that $c = d_{n+1}(z)$. In particular an elementary algorithm computes its homology groups using, for example, the Smith Normal Form technique (for details, see [Veb31]).

Definition 1.38. A *locally effective chain complex* is a *free* chain complex of $\mathbb{Z}$-modules $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ where each group C_n can have infinite nature, but there exists algorithms such that $\forall x \in C_n$, we can compute $d_n(x)$.

In this case, no *global* information is available. For example, it is not possible in general to compute the subgroups $\operatorname{Ker} d_n$ and $\operatorname{Im} d_{n+1}$, which can have infinite nature.

In general, we can talk of *locally effective objects* when only "local" computations are possible. For instance, we can consider a locally effective simplicial set; the set of n-simplexes can be infinite, but we can compute the faces of any specific n-simplex.

The effective homology technique consists in combining locally effective objects with effective chain complexes. In this way, we will be able to compute homology groups of locally effective objects.

The following notion is one of the fundamental notions in the effective homology method, since it will allow us to obtain homology groups of locally effective chain complexes in some situations.

Definition 1.39. A *reduction* ρ (also called *contraction*) between two chain complexes C_* and D_*, denoted in this memoir by $\rho : C_* \Rrightarrow D_*$, is a triple $\rho = (f, g, h)$

$$\begin{tikzcd} C_* \arrow[loop left, "h"] \arrow[r, bend left, "f"] & D_* \arrow[l, bend left, "g"] \end{tikzcd}$$

where f and g are chain complex morphisms, h is a graded group morphism of degree $+1$, and the following relations are satisfied:

1) $f \circ g = \mathrm{Id}_{D_*}$;
2) $d_C \circ h + h \circ d_C = \mathrm{Id}_{C_*} - g \circ f$;
3) $f \circ h = 0; \quad h \circ g = 0; \quad h \circ h = 0.$

The importance of reductions lies in the following fact. Let $C_* \Rrightarrow D_*$ be a reduction, then C_* is the direct sum of D_* and an acyclic chain complex; therefore the graded homology groups $H_*(C_*)$ and $H_*(D_*)$ are canonically isomorphic.

Very frequently, the *small* chain complex D_* is effective; so, we can compute its homology groups by means of elementary operations with integer matrices. On the other hand, in many situations the *big* chain complex C_* is locally effective and therefore its homology groups cannot directly be determined. However, if we know a reduction from C_* over D_* and D_* is effective, then we are able to compute the homology groups of C_* by means of those of D_*.

Given a chain complex C_*, a *trivial reduction* $\rho = (f, g, h) : C_* \Rrightarrow C_*$ can be constructed, where f and g are identity maps and $h = 0$.

As we see in the next proposition, the composition of two reductions can be easily constructed.

Proposition 1.40. Let $\rho = (f, g, h) : C_* \Rrightarrow D_*$ and $\rho' = (f', g', h') : D_* \Rrightarrow E_*$ be two reductions. Another reduction $\rho'' = (f'', g'', h'') : C_* \Rrightarrow E_*$ is defined by:

$$\begin{aligned} f'' &= f' \circ f \\ g'' &= g \circ g' \\ h'' &= h + g \circ h' \circ f \end{aligned}$$

Another important notion that provides a connection between locally effective chain complexes and effective chain complexes is the notion of equivalence.

Definition 1.41. A *strong chain equivalence* (from now on, *equivalence*) ε between two chain complexes C_* and D_*, denoted by $\varepsilon : C_* \Lleftarrow\!\!\Rrightarrow D_*$, is a triple (B_*, ρ_1, ρ_2) where B_* is a chain complex, and ρ_1 and ρ_2 are reductions from B_* over C_* and D_* respectively:

$$\begin{tikzcd} & B_* \arrow[dl, Rrightarrow, "\rho_1"'] \arrow[dr, Rrightarrow, "\rho_2"] & \\ C_* & & D_* \end{tikzcd}$$

Very frequently, D_* is effective; so, we can compute its homology groups by means of elementary operations with integer matrices. On the other hand, in many situations both C_* and B_* are locally effective and therefore their homology groups cannot directly be determined. However, if we know a reduction from B_* over C_*, a reduction from B_* over D_*, and D_* is effective, then we are able to compute the homology groups of C_* by means of those of D_*.

Once we have introduced the notion of equivalence, it is possible to give the definition of *object with effective homology*, which is the fundamental idea of the effective homology technique. These objects will allow us to compute homology groups of locally effective objects by means of effective chain complexes.

Definition 1.42. An *object with effective homology* X is a quadruple $(X, C_*(X), HC_*, \varepsilon)$ where:

- X is a locally effective object;
- $C_*(X)$ is a (locally effective) chain complex associated with X, that allows us to study the homological nature of X;
- HC_* is an effective chain complex;
- ε is an equivalence $\varepsilon : C_*(X) \Lleftarrow\!\!\Rrightarrow HC_*$.

Then, the graded homology groups $H_*(X)$ and $H_*(HC_*)$ are canonically isomorphic, then we are able to compute the homology groups of X by means of those of HC_*.

The main problem now is the following one: given a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{N}}$, is it possible to determine its effective homology? We must distinguish three cases:

- First of all, if a chain complex C_* is by chance effective, then we can choose the trivial effective homology: ε is the equivalence $C_* \Lleftarrow C_* \Rrightarrow C_*$, where the two components ρ_1 and ρ_2 are both the trivial reduction on C_*.
- In some cases, some theoretical results are available providing an equivalence between some chain complex C_* and an *effective* chain complex. Typically, the Eilenberg MacLane space $K(\mathbb{Z}, 1)$ has the homotopy type of the circle S^1 and a reduction $C_*(K(\mathbb{Z}, 1)) \Rrightarrow C_*(S^1)$ can be built.
- The most important case: let $X_1, \ldots, X_n$ be objects with effective homology and Φ a constructor that produces a new space $X = \Phi(X_1, \ldots, X_n)$ (for example, the Cartesian product of two simplicial sets, the classifying space of a simplicial group, etc). In *natural* "reasonable" situations, there exists an effective homology version of Φ that allows us to deduce a version with effective homology of X, the result of the construction, from versions with effective homology of the arguments $X_1, \ldots, X_n$. For instance, given two simplicial sets K and L with effective homology, then the Cartesian product $K \times L$ is an object with effective homology too; this is obtained by means of the Eilenberg-Zilber Theorem, see [RS06].

Two of the most basic (in the sense of fundamental) results in the effective homology method are the two *perturbation lemmas.* The main idea of both lemmas is that given a reduction, if we *perturb* one of the chain complexes then it is possible to perturb the other one so that we obtain a new reduction between the *perturbed* chain complexes. The first theorem (the Easy Perturbation Lemma) is very easy, but it can be useful. The Basic Perturbation Lemma is not trivial at all. It was discovered by Shih Weishu [Shi62], although the abstract modern form was given by Ronnie Brown [Bro67].

Definition 1.43. Let $C_* = (C_n, d_n)_{n\in\mathbb{N}}$ be a chain complex. A *perturbation* δ of the differential d is a collection of group morphisms $\delta = \{\delta_n : C_n \to C_{n-1}\}_{n\in\mathbb{N}}$ such that the sum $d + \delta$ is also a differential, that is to say, $(d + \delta) \circ (d + \delta) = 0$.

The perturbation δ produces a new chain complex $C'_* = (C_n, d_n + \delta_n)_{n\in\mathbb{N}}$; it is the *perturbed* chain complex.

Theorem 1.44 (Easy Perturbation Lemma, EPL)**.** Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{N}}$ be two chain complexes, $\rho = (f, g, h) : C_* \Rrightarrow D_*$ a reduction, and δ_D a perturbation of d_D. Then a new reduction $\rho' = (f', g', h') : C'_* \Rrightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from C_* by replacing the old differential d_C by a perturbed differential $d_C + g \circ \delta_D \circ f$;

2) the new chain complex D'_* is obtained from the chain complex D_* only by replacing the old differential d_D by $d_D + \delta_D$;

3) $f' = f$;

4) $g' = g$;

5) $h' = h$.

The perturbation δ_D of the *small* chain complex D_* is naturally transferred (using the reduction ρ) to the *big* chain complex C_*, obtaining in this way a new reduction ρ' between the perturbed chain complexes. On the other hand, if we consider a perturbation d_C of the top chain complex C_*, in general it is not possible to perturb the small chain complex D_* so that there exists a reduction between the perturbed chain complexes. As we will see, we need an additional hypothesis.

Theorem 1.45 (Basic Perturbation Lemma, BPL)**.** [Bro67] Let us consider a reduction $\rho = (f, g, h) : C_* \Rrightarrow D_*$ between two chain complexes $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{N}}$, and δ_C a perturbation of d_C. Furthermore, the composite function $h \circ \delta_C$ is assumed *locally nilpotent*, in other words, given $x \in C_*$ there exists $m \in \mathbb{N}$ such that $(h \circ \delta_C)^m(x) = 0$. Then a new reduction $\rho' = (f', g', h') : C'_* \Rrightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from the chain complex C_* by replacing the old differential d_C by $d_C + \delta_C$;

2) the new chain complex D'_* is obtained from D_* by replacing the old differential d_D by $d_D + \delta_D$, with $\delta_D = f \circ \delta_C \circ \phi \circ g = f \circ \psi \circ \delta_C \circ g$;

3) $f' = f \circ \psi = f \circ (\mathrm{Id}_{C_*} - \delta_C \circ \phi \circ h)$;

4) $g' = \phi \circ g$;

5) $h' = \phi \circ h = h \circ \psi$;

with the operators ϕ and ψ defined by

$$\phi = \sum_{i=0}^{\infty} (-1)^i (h \circ \delta_C)^i$$
$$\psi = \sum_{i=0}^{\infty} (-1)^i (\delta_C \circ h)^i = \mathrm{Id}_{C_*} - \delta_C \circ \phi \circ h,$$

the convergence of these series being ensured by the local nilpotency of the compositions $h \circ \delta_C$ and $\delta_C \circ h$.

It is worth noting that the effective homology method is not only a theoretical method but it has also been implemented in a software system called *Kenzo* [DRSS98]. In this system, the BPL is a central result and has been intensively used.

1.2 The Kenzo system

Kenzo is a 16000 lines program written in Common Lisp [Gra96], devoted to Symbolic Computation in Algebraic Topology. It was developed by Francis Sergeraert and some co-workers, and is www-available (see [DRSS98] for documentation and details). It works with the main mathematical structures used in Simplicial Algebraic Topology, [HW67], (chain complexes, differential graded algebras, simplicial sets, morphisms between these objects, reductions and so on) and has obtained some results (for example, homology groups of iterated loop spaces of a loop space modified by a cell attachment, see [Ser92]) which have not been confirmed nor refuted by any other means.

The fundamental idea of the Kenzo system is the notion of *object with effective homology* combined with *functional programming*. By using functional programming, some Algebraic Topology concepts are encoded in the form of algorithms. In addition, not only *known* algorithms were implemented but also new methods were developed to *transform* the main "tools" of Algebraic Topology, mainly the spectral sequences (which are not *algorithmic* in the traditional organization), into actual *computing* methods.

The computation process in the Kenzo system requires a great amount of algebraic structures and equivalences to be built, and thus a lot of resources.

This section is devoted to present some important features of Kenzo that will be relevant onward.

1.2.1 Kenzo mathematical structures

The data structures of the Kenzo system are organized in two layers. As algebraically modeled in [LPR03, DLR07], the first layer is composed of algebraic data structures (chain complexes, simplicial sets, and so on) and the second one of standard data structures (lists, trees, and so on) which are representing *elements* of data from the first layer.

Here we give an overview of how mathematical structures (the first layer) are represented in the Kenzo system, using object oriented (*CLOS*) and functional programming features simultaneously.

Most often, an object of some *type* in Mathematics is a structure with several components, frequently of functional nature. Let us consider the case of a user who wants to handle groups; this simple particular case is sufficient to understand how CLOS gives the right tools to process mathematical structures.

We can define a GRP class whose instances correspond to concrete groups as follows.

```
> (DEFCLASS GRP ()
    ((elements :type list :initarg :elements :reader elements)
     (mult :type function :initarg :mult :reader mult1)
     (inv :type function :initarg :inv :reader inv1)
     (nullel :type function :initarg :nullel :reader nullel)
     (cmpr :type function :initarg :cmpr :reader cmpr1))) ✠
#<STANDARD-CLASS GRP>
```

In this organization, a group is made of five slots which represent a list of the elements of the group (`elements`), the product of two elements of the group (`mult`), the inverse of an element of the group (`inv`), the identity element of the group (`nullel`) and a comparison test (`cmpr`) between the elements of the group. It is worth noting that `mult`, `inv`, `nullel` and `cmpr` slots are functional slots.

The most important difference between the mathematical definition and its implementation is that no axiomatic information appears in CLOS. Kenzo, being a system for computing, does not need any information about the properties of the operations, only their behavior is relevant. As a consequence of this, abelian groups can be implemented in Common Lisp similarly to groups, even when their mathematical definitions differ.

Let us construct now the group $\mathbb{Z}/2\mathbb{Z}$; that is to say, the cyclic group of dimension 2. In our context we define the comparison, the product and the inverse functions with

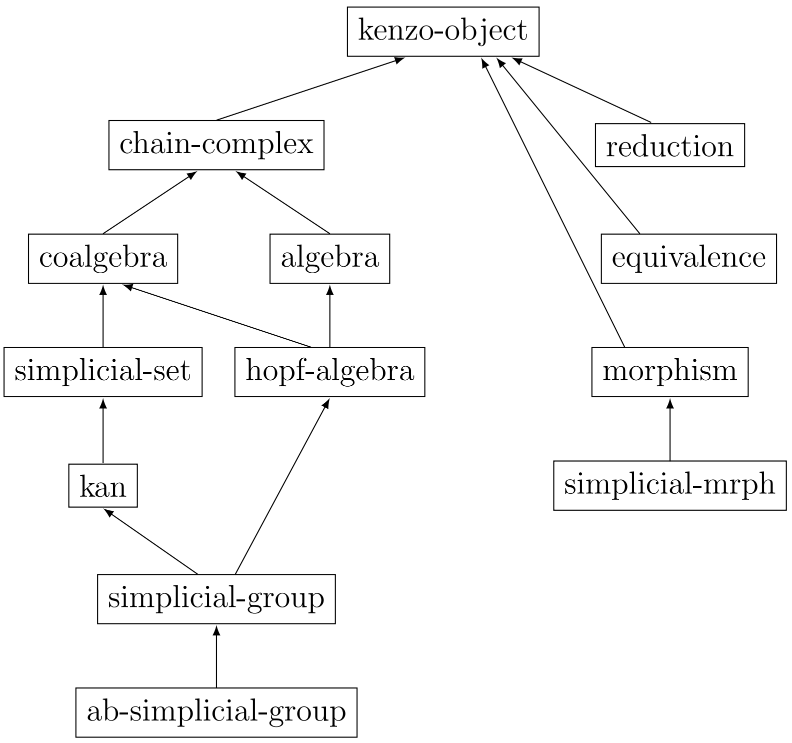


Figure 1.2: Kenzo class diagram

the help of the `mod` function, a predefined Lisp function.

```
> (MAKE-INSTANCE 'GRP
    :elements '(0 1)
    :mult #'(lambda (g1 g2) (mod (+ g1 g2) 2))
    :inv #'(lambda (g) (mod (- 2 g) 2))
    :nullel #'(lambda () 0)
    :cmpr #'(lambda (g1 g2) (= 0 (mod (- g1 g2) 2)))) ✠
#<GROUP @ #x26f4bc12>
```

In this way, we can define mathematical structures in Common Lisp using object oriented (*CLOS*) and functional programming features. The interested reader can consult [Ser01] where a detailed explanation about how defining mathematical structures in Common Lisp is presented.

The previously explained organization is moved to the Kenzo context in order to define the main mathematical structures used in Simplicial Algebraic Topology, [HW67]. Figure 1.2 shows the Kenzo class diagram where each class corresponds to the respective mathematical structure.

The lefthand part of the class diagram is made of the main mathematical objects that are used in combinatorial Algebraic Topology. As we said previously, a *chain complex* is a graded differential module; an *algebra* is a chain complex with a compatible multiplicative structure, the same for a *coalgebra* but with comultiplicative structure. If a multiplicative and a comultiplicative structures are added and if they are compatible with each other in a natural sense, then it is a *Hopf algebra*, and so on. The righthand part of the class diagram is made of the operations over the mathematical structures of the lefthand. It is worth noting that all the mathematical structures in the Kenzo system are graded structures.

The following class definition corresponds to the simplest algebraic structure implemented in Kenzo, free chain complexes:

```
(DEFCLASS CHAIN-COMPLEX ()
   ((cmpr  :type cmprf :initarg :cmpr  :reader cmpr1)
    (basis :type basis :initarg :basis :reader basis1)
    ;; BaSe GeNerator
    (bsgn :type gnrt :initarg :bsgn :reader bsgn)
    ;; DiFFeRential
    (dffr :type morphism :initarg :dffr :reader dffr1)
    ;; GRound MoDule
    (grmd :type chain-complex :initarg :grmd :reader grmd)
    ;; EFfective HoMology
    (efhm :type homotopy-equivalence :initarg :efhm :reader efhm)
    ;; IDentification NuMber
    (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
    ;; ORiGiN
    (orgn :type list :initarg :orgn :reader orgn)))
```

The relevant slots are `cmpr`, a function coding the equality between the generators of the chain complex; `basis`, the function defining the basis of each group of n-chains, or the keyword `:locally-effective` if the chain complex is not effective; `dffr`, the differential morphism, which is an instance of the class `MORPHISM`; `efhm`, which stores information about the effective homology of the chain complex; and `orgn`, is used to keep record of information about the object.

The class `CHAIN-COMPLEX` is extended by inheritance with new slots, obtaining more elaborate structures. For instance, extending it with an `aprd` ($\underline{\text{a}}$lgebra $\underline{\text{pr}}$o$\underline{\text{d}}$uct) slot, we obtain the `ALGEBRA` class. Multiple inheritance is also available; for example, the class `SIMPLICIAL-GROUP` is obtained by inheritance from the classes `KAN` and `HOPF-ALGEBRA`.

It is worth emphasizing here that simplicial sets have also been implemented as a subclass of `CHAIN-COMPLEX`. To be precise, the class `SIMPLICIAL-SET` inherits from the class `COALGEBRA`, which is a direct subclass of `CHAIN-COMPLEX`, with a slot `cprd` (the $\underline{\text{c}}$o$\underline{\text{pr}}$o$\underline{\text{d}}$uct). The class `SIMPLICIAL-SET` has then one slot of its own: `face`, a Lisp function computing any face of a simplex of the simplicial set. The `basis` is in this case (when working with effective objects) a function associating to each dimension n the list of non degenerate n-simplexes, and the differential map of the associated chain complex is given by the alternate sum of the faces, where the degenerate simplexes are cancelled.

1.2.2 Kenzo way of working

In Kenzo there is one objective: compute groups associated with topological spaces. This main objective can be broken in two actions: (1) constructing spaces, and (2) computing groups. Note that the first task is necessary to carry out the second one.

When a user has decided to construct a space in Kenzo, he should decide which type he wants to build: a simplicial set, a simplicial group and so on; namely, an object of one of the types of the lefthand part of the class diagram of Figure 1.2. Therefore, the user has to construct an instance of one of those classes.

As this task can be quite difficult for a non expert user, Kenzo provides useful functions to create interesting objects of regular usage, which belong to four types: chain complexes, simplicial sets, simplicial groups and abelian simplicial groups.

These functions can be split in two different kinds: (1) functions to construct initial spaces and, (2) functions to construct spaces from other ones applying topological constructors.

The following elements, gathered by the types of the constructed object, represent the main spaces that can be constructed in Kenzo from scratch (that is to say, which belong to the first kind):

- Chain Complexes:
  - *Unit chain complex*: the `zcc` function, with no arguments, constructs the unit chain complex, see Example 1.7.
  - *Circle*: the `circle` function, with no arguments, constructs the circle chain complex, see Example 1.7.
- Simplicial Sets:
  - *Standard simplicial set*: the `delta` function, with a natural number n as argument, constructs $\Delta[n]$, see Definition 1.18.
  - *Sphere*: the `sphere` function, with a natural number n as argument, constructs S^n, see Definition 1.22.
  - *Sphere Wedge*: the `sphere-wedge` function, with a sequence of natural numbers $n1, \ldots, nk$ as arguments, constructs $S^{n1} \vee \ldots \vee S^{nk}$, see Definition 1.23.
  - *Moore space*: the `moore` function, with two natural numbers n and p as arguments, constructs $M(\mathbb{Z}/n\mathbb{Z}, p)$, see Definition 1.24.
  - *Projective space*: the `r-proj-space` function has two optional arguments k and l. If neither k and l are provided, then the function constructs $P^\infty\mathbb{R}$. If k is provided but l is not, then the function constructs $P^\infty\mathbb{R}/P^{k-1}\mathbb{R}$. If both k and l are provided, then the function constructs $P^l\mathbb{R}/P^{k-1}\mathbb{R}$. In the latter case, if $k = 1$, then the function constructs $P^l\mathbb{R}$; see Definition 1.25.

  - *Finite simplicial set*: the `build-finite-ss` function, with a list of lists as argument, constructs a simplicial set, see [DRSS98].

- Abelian Simplicial Group:
  - *Eilenberg MacLane space* type $(\mathbb{Z}, n)$: the `k-z` function, with a natural number n as argument, constructs $K(\mathbb{Z}, n)$, see Definition 1.31.
  - *Eilenberg MacLane space* type $(\mathbb{Z}/2\mathbb{Z}, n)$: the `k-z2` function, with a natural number n as argument, constructs $K(\mathbb{Z}/2\mathbb{Z}, n)$, see Definition 1.31.

On the contrary, the following elements, gathered by types, represent the main spaces that can be constructed in Kenzo from other spaces applying topological constructors (that is to say, which belong to the second kind):

- Chain Complexes:
  - *Tensor Product*: the `tnsr-prdc` function, with two chain complexes C_*, D_* as arguments, constructs the tensor product $C_* \otimes D_*$, see Definition 1.10.
- Simplicial Sets:
  - *Cartesian product*: the `crts-prdc` function, with two simplicial sets K, L as arguments, constructs the Cartesian product $K \times L$, see Definition 1.26.
  - *Suspension*: the `suspension` function, with a simplicial set X and a natural number n as arguments, constructs the suspension $\Sigma^n(X)$, see Definition 1.28.
- Simplicial Group:
  - *Loop space*: the `loop-space` function, with a simplicial set X and a natural number n as arguments, constructs the loop space $\Omega^n(X)$, see Definition 1.30.
  - *Classifying space*: the `classifying` function, with a simplicial group X, constructs the classifying space $B(X)$, see Definition 1.32.

Eventually, once we have constructed some spaces in our Kenzo session, the Kenzo user can perform computations. Namely, the `homology` function with a chain complex X (or an instance of one of its subclasses: simplicial set, simplicial group and so on) and a natural number n as arguments computes $H_n(X)$.

To sum up, a simplification of the way of working with Kenzo is as follows. As a first step, the user constructs some initial spaces by means of some built-in Kenzo functions (as spheres, Moore spaces, Eilenberg MacLane spaces and so on); then, in a second step, he constructs new spaces by applying topological constructions (as Cartesian products, loop spaces, and so on); as a third, and final, step, the user asks Kenzo for computing the homology groups of the spaces. Let us remark that this kind of interaction does not fully cover all the Kenzo capabilities, but it is just a simplification.

1.2.3 Kenzo in action

Let us show a didactic example to illustrate the interaction with the Kenzo program. The homology group $H_5(\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4)))$ is "in principle" reachable thanks to old methods, see [CM95], but experience shows even the most skilful topologists meet some difficulties to determine it, see [RS02]. With the Kenzo program, you construct the Moore space $M(\mathbb{Z}/2\mathbb{Z},4)$ in the following way:

```
> (setf m4 (moore 2 4)) ✠
[K1 Simplicial-Set]
```

A Kenzo display must be read as follows. The initial `>` is the Lisp prompt of this Common Lisp implementation. The user types out a Lisp statement, here `(setf m4 (moore 2 4))` and the maltese cross ✠ (in fact not visible on the user screen) marks in this text the end of the Lisp statement, just to help the reader: the right number of closing parentheses is reached. The Return key then asks Lisp to *evaluate* the Lisp statement. Here the Moore space $M(\mathbb{Z}/2\mathbb{Z},4)$ is constructed by the Kenzo function `moore`, taking into account of the arguments 2 and 4, and this Moore space is *assigned* to the Lisp symbol `m4` for later use. Also evaluating a Lisp statement *returns* an object, the result of the evaluation, in this case the Lisp object implementing the Moore space, displayed as `[K1 Simplicial-Set]`, that is, the Kenzo object #1, a `Simplicial-Set`. The internal structure of this object, made of a rich set of data, in particular many functional components, is not displayed. The identification number printed by Kenzo allows the user to recover the whole object by means of a function called simply `k` (for instance, the evaluation of `(k 1)` returns the Moore space $M(\mathbb{Z}/2\mathbb{Z},4)$, in our running example). In addition, another function (called `orgn` and which is one of the slots of the `Chain-Complex` class) allows the user to obtain the origin of the object (i.e. from which function and with which arguments has been produced), and thus the printed information is enough to get a complete control of the different objects built with Kenzo.

It is then possible to construct the third loop space of the Moore space, $\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4))$, as a simplicial *group*.

```
> (setf o3m4 (loop-space m4 3)) ✠
[K30 Simplicial-Group]
```

The combinatorial version of the loop space is *highly* infinite: it is a combinatorial version of the space of *continuos* maps $S^3 \to M(\mathbb{Z}/2\mathbb{Z},4)$, but functionally encoded as a small set of functions in a `Simplicial-Group` object.

Eventually, the user can compute the fifth homology group of this space.

```
> (homology o3m4 5) ✠
Homology in dimension 5:
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
```

This result must be interpreted as stating $H_5(\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4))) = \mathbb{Z}_2^5$. In this way, Kenzo computes the homology groups of *complicated* spaces. This is due to the Kenzo implementation of the effective homology method which is explained in the following subsection.

1.2.4 Effective Homology in Kenzo

As we have previously said, the central idea of the Kenzo system is the notion of *object with effective homology*. In this subsection, we are going to show how this notion explained in Subsection 1.1.3 is used in Kenzo.

As we stated in Subsection 1.1.3, the main problem is the following one: given an object, determine its effective homology version. Three cases have been distinguished.

First of all, let X be an object; if the chain complex $C_*(X)$ is by chance effective, then we can choose the trivial effective homology: ε is the equivalence $C_*(X) \Lleftarrow C_*(X) \Rrightarrow C_*(X)$, where both reductions are the trivial reduction on C_*. This situation happens, for instance, in the case of the sphere S^3 (we consider a fresh Kenzo session).

```
> (setf s3 (sphere 3)) ✠
[K1 Simplicial-Set]
```

We can ask for the effective homology of S^3 as follows:

```
> (efhm s3) ✠
[K9 Homotopy-Equivalence K1 <= K1 => K1]
```

An homotopy equivalence is automatically constructed by Kenzo where both reductions are the trivial reduction on $C_*(S^3)$ (let us note that the `K1` object not only represents the simplicial set S^3 but also the chain complex $C_*(S^3)$ due to the heritage relationship between simplicial sets and chain complexes). In this case, the effective homology technique does not provide any additional tool to the computation of the homology groups of S^3. On the contrary, we will see the power of this technique when the initial space X is locally effective.

The second feasible situation happened when given a locally effective object X some theoretical result was available providing an equivalence between the chain complex $C_*(X)$ and an *effective* chain complex.

According to Subsubsection 1.1.2.1, a simplicial model of the Eilenberg MacLane space $K(\mathbb{Z}, 1)$ is defined by $K(\mathbb{Z}, 1)_n = \mathbb{Z}^n$; an infinite number of simplexes is required in every dimension $n \geq 1$; that is, we have a locally effective object. This does not prevent such an object from being installed and handled by Kenzo.

```
> (setf kz1 (k-z 1)) ✠
[K10 Abelian-Simplicial-Group]
```

The `k-z` Kenzo function construct the standard Eilenberg MacLane space. In ordinary mathematical notation (as seen in Subsubsection 1.1.2.1), a 3-simplex of `kz1` could be for example $[3, 5, -5]$, denoted by $(3\ 5\ \ -5)$ in Kenzo. The faces of this simplex can be determined as follows.

```
> (dotimes (i 4) (print (face kz1 i 3 '(3 5 -5)))) ✠
<AbSm - (5 -5)>
<AbSm - (8 -5)>
<AbSm 1 (3)>
<AbSm - (3 5)>
nil
```

The faces are computed as explained in Subsubsection 1.1.2.1. Then, *local* computations are possible, so, the object `kz1` is *locally effective*. But no global information is available. For example, if we try to obtain the list of non degenerate simplexes in dimension 3, we obtain an error.

```
> (basis kz1 3) ✠
Error: The object [K10 Abelian-Simplicial-Group] is locally-effective
```

This basis in fact is $\mathbb{Z}^3$, an infinite set whose element list cannot be explicitly stored nor displayed. So, the homology groups of `kz1` cannot be elementarily computed. However, $K(\mathbb{Z}, 1)$ has the homotopy type of the circle S^1 and the Kenzo program knows this fact.

```
> (efhm kz1) ✠
[K31 Homotopy-Equivalence K10 <= K10 => K25]
```

A reduction `K10` $= K(\mathbb{Z}, 1) \Rrightarrow$ `K25` is constructed by Kenzo. What is `K25`?

```
> (orgn (k 25)) ✠
(circle)
```

K25 is the expected object, the circle S^1 which is an effective chain complex; so we can compute its homology groups by means of tradicional methods. Therefore, we can compute the homology groups of the space $K(\mathbb{Z}, 1)$ by means of the effective homology method.

Eventually, the last situation happened when given $X_1, \dots, X_n$ objects with effective homology and Φ a constructor that produced a new space $X = \Phi(X_1, \dots, X_n)$, we wanted to compute the effective homology version of X. Let us present an example.

The Cartesian product of two locally effective simplicial sets produces another locally effective simplicial set.

```
> (setf kz1xkz1 (crts-prdc kz1 kz1)) ✠
[K15 Simplicial-Set]
> (basis kz1xkz1 3) ✠
Error: The object [K15 Simplicial-Set] is locally-effective
```

So, the homology groups of kz1xkz1 cannot be elementarily computed. However, Kenzo is able to construct an equivalence between this object and an effective chain complex.

```
> (efhm kz1xkz1) ✠
[K63 Homotopy-Equivalence K15 <= K53 => K43]
```

An equivalence K15 $= K(\mathbb{Z}, 1) \times K(\mathbb{Z}, 1) \twoheadleftarrow$ K53 $\twoheadrightarrow$ K43 is constructed by Kenzo. What is K43?

```
> (orgn (k 43)) ✠
(TNSR-PRDC (circle) (circle))
```

The object K43 is the tensor product of two circles $C_*(S^1) \otimes C_*(S^1)$ (the reduction is obtained from the Eilenberg-Zilber Theorem, see [RS06]), an effective chain complex; so we can compute its homology groups by means of tradicional methods. Therefore, we can compute the homology groups of the space $K(\mathbb{Z}, 1) \times K(\mathbb{Z}, 1)$ by means of the effective homology method.

It is worth noting that a Kenzo user does not need to explicitly construct the equivalence to compute the homology groups of a locally effective object. This task is automatically performed by the Kenzo system which constructs the necessary objects without any additional help.

For instance, let us consider a fresh Kenzo session where we have constructed the space $\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4))$, the example of the previous subsection:

```
> (setf m4 (moore 2 4)) ✠
[K1 Simplicial-Set]
> (setf o3m4 (loop-space m4 3)) ✠
[K30 Simplicial-Group]
```

At this moment, we can check the number of objects constructed in Kenzo (this information is stored in a global variable called `*idnm-counter*`).

```
> *idnm-counter* ✠
41
```

Subsequently, after computing the third homology, we ask again the number of objects constructed in Kenzo and we obtain the following result.

```
> (homology o3m4 3) ✠
Homology in dimension 3 :
Component Z/4Z
Component Z/2Z
> *idnm-counter* ✠
404
```

This means that Kenzo has constructed 363 intermediary objects in order to compute the homology groups of $\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4))$; namely, in order to construct an equivalence between the locally effective object $\Omega^3(M(\mathbb{Z}/2\mathbb{Z},4))$ and an effective chain complex.

1.2.5 Memoization in Kenzo

The Kenzo program is certainly a functional system. It is frequent that several thousands of functions are present in memory, each one being dynamically defined from other ones, which in turn are defined from other ones, and so on. In this quite original situation, the same calculations are frequently asked again. To avoid repeating these calculations, it is better to store the results and to systematically examine for each calculation whether the result is already available (*memoization* strategy).

As a consequence, the state of a space evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a homology group, depends on the concrete state of the space involved in the calculation (in the more explicit case, to re-calculate the homology group of a space could be negligible in time, even if in the first occasion this was very time consuming).

Let us shown an example, of the Kenzo memoization. We want to compute the fifth homology group of the space $\Omega^3(S^4 \times S^4)$ in a fresh Kenzo session and see how much time this computation takes (using the `time` function); therefore, we proceed as usual.

```
> (setf s4 (sphere 4)) ✠
[K1 Simplicial-Set]
> (setf s4xs4 (crts-prdc s4 s4)) ✠
[K6 Simplicial-Set]
> (setf o3s4xs4 (loop-space s4xs4 3)) ✠
[K35 Simplicial-Group]
> (time (homology o3s4xs4 5)) ✠
;; some lines skipped
; real time  1,139,750 msec (00:18:59.750)
```

The first time that we compute $H_5(\Omega^3(S^4 \times S^4))$, Kenzo takes almost 20 minutes to obtain the result. However, if we ask again for the same computation:

```
> (time (homology o3s4xs4 5)) ✠
;; some lines skipped
; real time  262,484 msec (00:04:22.484)
```

in this case Kenzo only needs 4 minutes. It is worth noting that Kenzo does not store the final result (that is to say, the group $H_5(\Omega^3(S^4 \times S^4))$) but intermediary computations related to the differential of the generators of the space. Then, when a Kenzo user asks a computation previously computed, Kenzo does not simply look up and returned it, but it uses some previously stored computations to calculate the result faster.

Moreover, it is very important not to have several copies of the same function; otherwise it is impossible for one copy to guess some calculation has already been done by another copy. This is a very important question in Kenzo, so that the following idea has been used. Each Kenzo object has a rigorous definition, stored as a list in the `orgn` slot of the object (`orgn` stands for *origin* of the object). This is the main reason of the top class `kenzo-object`: making this process easier. The actual definition of the `kenzo-object` class is:

```
(DEFCLASS KENZO-OBJECT ()
  ((idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
   (orgn :type list :initarg :orgn :reader orgn))) ✠
```

Then, when any `kenzo-object` is to be considered, its *definition* is constructed and the program firstly looks at `*k-list*` (a list which stores the already constructed `kenzo-object` instances) whether some object corresponding to this definition already exists; if yes, no `kenzo-object` is constructed, the already existing one is simply returned. Look at this small example where we construct the second loop space of S^3, then the first loop space, and then again the second loop space. In fact the initial construction of the second loop space required the first loop space, and examining the identification number `K??` of these objects shows that when the first loop space is later asked for, Kenzo is able to return the already existing one.

```
> (setf s3 (sphere 3)) ✠
[K372 Simplicial-Set]
> (setf o2s3 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (setf os3 (loop-space s3 1)) ✠
[K374 Simplicial-Group]
> (setf o2s3-2 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (eq o2s3 o2s3-2) ✠
T
```

The last statement shows the symbols `o2s3` and `o2s3-2` points to the same machine address. In this way we are sure any `kenzo-object` has no duplicate, so that the memory process for the values of numerous functions cannot miss an already computed result.

1.2.6 Reduction degree

Working with Kenzo, a constructor for the space X has associated a number $g(X) \in \mathbb{Z}$. When we apply an operation O over the space X, a set of rules are used to compute $g(O(X))$. We will call *reduction degree of* X to this number, $g(X)$, which is attached to our concrete representation of the spaces. This number is a lower bound of the simply connectedness degree; that is to say, the homotopy groups of the space X are null at least from 1 to $g(X)$. We list as follows the set of rules for the initial spaces and topological operators which are going to be employed in this memoir.

Tensor product ($X \otimes Y$): $g(X \otimes Y) = min\{g(X), g(Y)\}$.

Suspension ($\Sigma^n(X)$): $g(\Sigma^n(X)) = \begin{cases} g(X)+n & \text{if } g(X) \geq 0 \\ g(X) & \text{if } g(X) < 0 \end{cases}$

Sphere (S^n): $g(S^n) = n-1$.

Moore space ($M(\mathbb{Z}/p\mathbb{Z}, n)$): $g(M(\mathbb{Z}/p\mathbb{Z}, n)) = n-1$.

Standard simplicial set (Δ^n): $g(\Delta^n) = 0$.

Sphere wedge ($S^{n_1} \vee S^{n_2} \vee \ldots \vee S^{n_k}$):

$$g(S^{n_1} \vee S^{n_2} \vee \ldots \vee S^{n_k}) = min\{g(S^{n_1}), g(S^{n_2}), \ldots, g(S^{n_k})\}.$$

Projective space ($P^\infty\mathbb{R}$) : $g(P^\infty\mathbb{R}) = 0$.

Projective space ($P^\infty\mathbb{R}/P^n\mathbb{R}$) : $g(P^\infty\mathbb{R}/P^n\mathbb{R}) = n-1$.

Projective space ($P^l\mathbb{R}$) : $g(P^l\mathbb{R}) = 0$.

Projective space ($P^l\mathbb{R}/P^n\mathbb{R}$) : $g(P^l\mathbb{R}/P^n\mathbb{R}) = n - 1$.

$K(\mathbb{Z}, n)$: $g(K(\mathbb{Z}, n)) = n - 1$.

$K(\mathbb{Z}/2\mathbb{Z}, n)$: $g(K(\mathbb{Z}/2\mathbb{Z}, n)) = n - 1$.

Loop space ($\Omega^n(X)$): $g(\Omega^n(X)) = g(X) - n$.

Cartesian product ($X \times Y$): $g(X \times Y) = min\{g(X), g(Y)\}$.

Classifying space ($B^n(X)$): $g(B^n(X)) = \begin{cases} g(X) + n & \text{if } g(X) \geq 0 \\ g(X) & \text{if } g(X) < 0 \end{cases}$

The reduction degree of a space X provides us information about the capabilities of the Kenzo system to obtain the homology groups of X. When $g(X) < 0$, a Kenzo attempt to compute the homology groups of X will raise in an error; otherwise the Kenzo system can compute the homology groups of X.

For instance, the reduction degree of $\Omega^2 S^2$ is -1; then, let us show what happens if we try to compute the third homology group of this space.

```
> (setf s2 (sphere 2)) ✠
[K1 Simplicial-Set]
> (setf o2s2 (loop-space s2 2)) ✠
[K18 Simplicial-Group]
> (homology o2s2 3) ✠
Error: ‘NIL’ is not of the expected type ‘NUMBER’
[condition type: TYPE-ERROR]
```

As we can see an error is produced; then, the Kenzo system cannot compute the homology groups for $\Omega^2(S^2)$.

1.2.7 Homotopy groups in Kenzo

Up to now, we have been working with one of the most important algebraic invariants in Algebraic Topology: homology groups. We can wonder what happens with the other main algebraic invariant: *homotopy groups*.

The n-homotopy group of a topological space X with a base point x_0 is defined as the set of homotopy classes of continuous maps $f : S^n \to X$ that map a chosen base point $a \in S^n$ to the base point $x_0 \in X$. A more detailed description and results about homotopy groups can be found in [Hat02, May67].

Homotopy groups were defined by Hurewicz in [Hur35] and [Hur36] as a generalization of the fundamental group [Poi95]. It is worth noting that except in special cases, homotopy groups are hard to be computed. For instance, whereas the homology groups of spheres are easily computed, computing their homotopy groups remains as a difficult subject, see [Tod62, Mah67, Rav86].

For the general case, Edgar Brown published in [Bro57] a *theoretical* algorithm for the computation of homotopy groups of simply connected spaces such that their homology groups are of finite type. However Brown himself explained that his "algorithm" has not practical use.

An interesting algorithm based on the effective homology theory was developed by P. Real, see [Rea94]. In that paper, an algorithm that computes the homotopy groups of a 1-reduced simplicial set was explained. Here, we just state the algorithm.

Algorithm 1.46 ([Rea94]).
Input: a 1-reduced simplicial set with effective homology X and a natural number n such that $n \geq 1$.
Output: the n-th homotopy group of the underlying simplicial set X.

This algorithm is based on the Whitehead tower process [Hat02], a method which allows one to reach any homotopy group of a 1-reduced simplicial set.

It is worth noting that Kenzo, in spite of not providing a function called `homotopy`, like in the case of homology groups, implements all the necessary tools to use the algorithm presented in [Rea94]. A detailed explanation about how to use this method in Kenzo was explained in Chapter 21 of the Kenzo documentation [DRSS98]. However, it is worth noting that in the current Kenzo version, homotopy groups of a 1-reduced simplicial set X can only be computed if the first non null homology group of X is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$; this is due to the fact that the algorithm presented in [Rea94] needs the calculation of homology groups of Eilenberg-MacLane spaces $K(G,n)$, and in the original Kenzo distribution only the homology of spaces $K(\mathbb{Z},n)$ and $K(\mathbb{Z}/2\mathbb{Z},n)$ are built-in.

1.3 ACL2

The ACL2 Theorem Prover has been used to verify the correctness of some Kenzo programs, which will be presented in chapters 5 and 6. This section is devoted to provide a brief description of ACL2. In spite of being a glimpse introduction to this system, this description provides enough information to read the sections dedicated to ACL2 topics. A complete description of ACL2 can be found in [KMM00b, KM].

In addition, an interesting ACL2 feature, that will be really important in our developments, is described in this section, too. Some other necessary concepts about ACL2 will be introduced later on for a better understanding of this memoir.

1.3.1 Basics on ACL2

Information in this section has been mainly extracted from [KMM00b].

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

The first version of ACL2 was developed by B. Boyer and J S. Moore in 1989, using the functional programming language Common Lisp. ACL2 is the successor to the Nqthm [BM97] (or Boyer-Moore) logic and proof system and its Pc-Nqthm interactive enhancement. ACL2 was born as response to the problems Nqthm users faced in applying that system to large-scale proof projects. Namely, the main drawback of the Nqthm appears when the logic specification were used not only for reasoning about the modeling system but also for executing. The main difference between ACL2 and Nqthm lies in the chance of executing the models of the ACL2 logic in Common Lisp.

ACL2 and Nqthm have already shown their effectiveness for implementing (non-trivial) large proofs in Mathematics. One of these examples can be found in [Kau00], where the Fundamental Theorem of Calculus is formalized; another development can be found in [BM84], where a mechanical proof of the unsolvability of the halting problem is implemented; other examples of theorems certified with ACL2 and Nqthm are: the Gauss's Law of Quadratic Reciprocity [Rus92], the Church-Rosser theorem for lambda calculus [Sha88], the Gödel's incompleteness theorem [Sha94], and so on.

Besides, ACL2 has been used for a variety of important formal methods projects of industrial and commercial interest, including (for example) the verification of the RTL code which implements elementary floating point operations of the AMD Athlon processor [Rus98] and ROM microcode programs of CAP digital signal processor of Motorola [BKM96].

More references about these and other developments related to ACL2 and Nqthm are www-available at [KM, BM97].

After this brief historical introduction, let us devote some lines to explain the programming language, the logic and the theorem prover of ACL2.

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

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

An interesting feature of ACL2 is that the same language is used to define programs and to specify properties of those programs. Every time a function is defined with `defun`, in addition to define a program, it is also introduced as an axiom in the logic. Theorems and lemmas are stated in ACL2 by the `defthm` command, and this command also starts a proof attempt in the ACL2 theorem prover. In the ACL2 jargon a file containing definitions and statements that have been certified as admissible by the system is called a *book*.

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

In the following subsection, an introduction to an interesting ACL2 feature, that will be critical in our developments, is presented.

1.3.2 ACL2 encapsulates

In this subsection, an introduction to the ACL2 encapsulates is given. Encapsulates are a powerful tool which allow the structural development of theories. The encapsulation facility is much more general than sketched here, see [KMM00b, KMM00a, BGKM91, KM01]. They will be used, in this memoir, in Section 4.3 and onward.

The *encapsulate principle* allows the introduction of function symbols in ACL2, without a complete specification of them, but just assuming some properties which partially define them. To be admissible and to preserve the consistency, there must exist some functions (called "local witness") verifying the properties assumed as axioms. Once this is proved, the local witness definitions can be neglected, and the function symbols introduced with the encapsulate principle are partially specified by means of the assumed properties. Then, we can assume a property described by the formula `F` for the functions `f1`, ..., `fn` if:

- `f1`, ..., `fn` are new function symbols.
- There exist admissible definitions of n functions whose names are `f1`, ..., `fn` such that the formula `F` can be proved for those functions.

If the encapsulate is admissible, the function symbols `f1`, ..., `fn` are added to the ACL2 language and the formula `F` is added as axiom to the logic. It is worth noting that the local witness are only used to ensure the admissibility of the encapsulate, since the axioms associated with their definitions are not kept.

In order to clarify this ACL2 mechanism an example will be presented. Let us suppose that we want to assume the existence of a binary function which is associative and commutative. The ACL2 command in charge of this task is the following one.

```
(encapsulate
    ; the signatures
    (((op * *) => *))

    ; the witnesses
    (local (defun op (a b) (+ a b)))

    ; the axioms
    (defthm op-associative
      (equal (op (op x y) z) (op x (op y z))))

    (defthm op-commutative
      (equal (op x y) (op y x))))
```

The first expression is a list with the arity description of the functions which are introduced with the encapsulate principle (called *signature*). In this case, we indicate that `op` is a function with two arguments which returns a unique value. The local witnesses are defined using `defun`, but they are declared as local by means of `local`. The properties assumed over the encapsulate functions are stated by means of `defthm`. The encapsulate can contain definitions and properties declared as local; however, once the admissibility of the encapsulate has been proved, just the non local definitions and properties are added to the ACL2 logic.

The functions defined by means of an encapsulate *cannot be executed*, since its partial specification can be not enough to deduce the value for every input. However, there exists a great advantage from the verification point of view; due to the fact that we are formalizing and verifying a generic result which can be instantiated for concrete cases later on. This issue will be presented in Subsection 6.1.1.

Chapter 2

A framework for computations in Algebraic Topology

Traditionally, Symbolic Computation systems, and Kenzo is no exception, have been oriented to research. This implies in particular, that development efforts in the area of Computer Algebra systems have been focussed on aspects such as the improvement of the efficiency or the extension of the applications scope. On the contrary, aspects such as the interaction with other systems or the development of friendly user interfaces are pushed into the background (it is worth noting that most of Computer Algebra systems use a command line as user interface). Things are a bit different in the case of widely spread commercial systems such as *Mathematica* or *Maple*, where some attention is also payed to connectivity issues or to special purpose user interfaces. But even in these cases the central focus is on the results of the calculations and not on the interaction with other kind of (software or human) agents.

The situation is, in any sense, similar in the area of interoperability among Symbolic Computation systems (including here both Computer Algebra systems and Proof Assistants tools). In this case, the emphasis has been put in the universality of the middleware (see, for instance, [CH97]). Even if important advances have been achieved, severe problems have appeared, too, such as difficulties in reusing previous proposals and the final obstacle of the speculative existence of a *definitive mathematical interlingua*. The irruption of XML technologies (and, in our context, of MathML [A^+08] and OpenMath [Con04]) has allowed standard knowledge management, but they are located at the *infrastructure* level, depending always on higher-level abstraction devices to put together different systems. Interestingly enough, the initiative SAGE [Ste] producing an integrated environment seems to have no use for XML standards, intercommunication being supported by ad-hoc SAGE mechanisms.

To sum up, in the symbolic computation area, we are always looking for *more powerful* systems (with more computation capacities or with more general expressiveness). However, it is the case that our systems became so powerful, that we can lose some interesting kinds of users or interactions. This situation was encountered in the design and development of *TutorMates* [GL$^+$09]. TutorMates is aimed at linking an educational front-end (based on Java) with the *Maxima* system [Sch09] (a Common Lisp Computer Algebra system specialized in symbolic operations but that also offers numerical capabilities such as arbitrary-precision arithmetic). The purpose of TutorMates was educational, so it was clear that many outputs given by Maxima were unsuitable for final users (students, and teachers, at high school level) depending on the degree and the topic learned in each TutorMates session. To give just an example, an imaginary solution to a quadratic equation has meaning only in certain courses. In this way, a *mediated* access to Maxima was designed. The central concept is an intermediary layer that communicates, by means of an extension of MathML, the front-end and Maxima. This approach is now transferred to the field of Symbolic Computation in Algebraic Topology, where the Kenzo system provides a complete set of calculation tools, which can be considered difficult to use by a non-Common Lisp trained user (typically, an Algebraic Topology student, teacher or researcher).

The most elaborated approach to increase the usability and accessibility of Kenzo was reported in [APRR05]. There, a remote access to Kenzo was devised, using *CORBA* [Gro] technology. An XML extension of MathML played a role there too, but just to give genericity to the connection (avoiding the definition in the CORBA Interface Description Language [Gro] of a different specification for each Kenzo class and datatype). There was no intention of taking profit from the semantics possibilities of MathML. Being useful, this approach ended in a prototype, and its enhancement and maintenance were difficult, due both to the low level characteristics of CORBA and to the pretentious aspiration of providing *full* access to Kenzo functionalities. We could classify the work of [APRR05] in the same line as [CH97] or the initiative [IAM], where the emphasis is put into powerful and generic access to symbolic computation engines.

Now, we have undertaken the task of devising a framework, from now on called Kenzo framework, which provides a *mediated* access to the Kenzo system, constraining the Kenzo functionality, but providing guidance to the user in his navigation on the system.

The rest of this chapter is organized as follows. A brief overview of the Kenzo framework architecture is presented in Section 2.1. Section 2.2 is devoted to provide a detailed description of each one of the Kenzo framework components. Finally, the Kenzo framework execution flow is presented by means of an example in Section 2.3. Two ongoing works devoted to increase the computation capabilities of the Kenzo framework by means of ideas about remote and distributed computations are presented respectively in Section 2.4 and 2.5.

2.1 Framework architecture

When starting the project of developing a framework for Kenzo, several requirements were stated. Some of them were simply natural specifications, others were of a more problematic nature. Those issues are presented here as challenges to be fulfilled. These challenges largely determined the design decisions presented in this section. The most important challenges we faced were:

1. *Functionality.* The system should provide access to the Kenzo capabilities for constructing topological spaces and computing (homology and homotopy) groups.
2. *Extensibility of Kenzo.* The system design should be capable of evolving at the same time as the Kenzo system.
3. *Integration with other systems.* In spite of having Kenzo as main computational kernel, the system should be designed in such a way that it could support different connections to other symbolic manipulation systems (GAP [GAP] in computational algebra, for instance, or ACL2 [KM] from the theorem proving side).
4. *Interaction with different clients.* The system should be designed in such a way that it could support several ways of interaction (graphical user interfaces, web services, web applications, Computer Algebra systems, and so on).
5. *Efficiency.* The framework should be roughly equivalent to Kenzo in time and space efficiency.
6. *Error handling.* Our framework should forbid the user some manipulations raising errors, from both structural and semantical points of view.
7. *Representation of mathematical knowledge.* The representation of the data of our framework should be independent of the modules of the framework, that can be programmed in different programming languages.
8. *Communication of the mathematical knowledge.* The encoded data should be easily transferable both between the different modules of the framework and also outside the framework.

Let us observe that, as it is usual in system design, some decisions aimed to fulfill a concrete requirement could compromise other ones. The most important trade-off in our previous list is between requirements 2, 3, 4 (these three requirements are three different extensibility nuances) and 5. A layered architecture with complex mediators could produce poorer performance. A careless treatment of intermediary documents and files could also imply a great memory waste. The error handling (item 6) is closely related to the functionality included in the system (item 1), since the pure Kenzo system allows some instructions that are not desirable from the point of view of error managing (since they produce runtime errors), then, the system should avoid this kind of situations. Besides, error handling (item 6) could be in a conflict with efficiency, too, because dealing with semantical information at the external layers of an architecture can slow down the system as a whole. In addition, it would be very useful if the chosen encoding for the data, related to requirements 7 and 8, was able to include some knowledge in order to

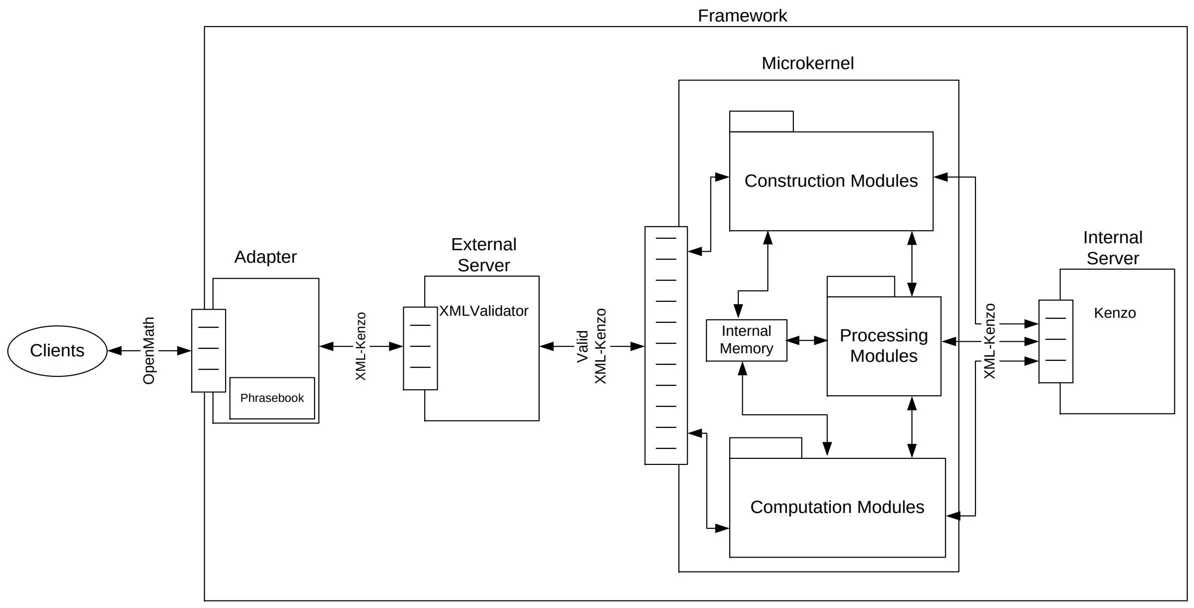


Figure 2.1: Architecture of the Kenzo framework based on the Microkernel pattern

help the management of error handling. We have tried to deal with all these constraints while respecting the requirements guided by already proved methodologies and *patterns*.

The previous discussion led us to choose the *Microkernel architectural pattern* [B$^+$96, B$^+$07] to organize the system and XML to encode the mathematical and systemic knowledge.

The Microkernel pattern gives a global view as a *platform*, in terminology of [B$^+$96], which implements a virtual machine with applications running on top of it, namely a *framework* (in the same terminology). The *Microkernel* architectural pattern applies to software systems that must be able to adapt to changing systems requirements. It separates a minimal functional core from extended functionality and customer-specific parts. This pattern defines five kinds of participating *components*: *internal servers*, *external servers*, *adapters*, *clients* and the *microkernel*.

The *microkernel* is the main component and includes functionality that enables other components running in separate processes to communicate with each other. It is also in charge of maintaining system-wide resources such as files or processes. Core functionality that cannot be implemented within the microkernel is separated in *internal servers*. An *external server* is a component that uses the microkernel for implementing its own view of the underlying application domain. The external server receives requests from client applications using the communication facilities provided by the *adapter*.

A high level perspective of the architecture of our system, based on this pattern, as a whole is shown in Figure 2.1.

Let us present a brief overview of the framework components, a more detailed description of each one of them will be provided in Section 2.2.

First of all, let us give some flavor about the concept of *mediated access* in our framework. It is worth noting that the mediated access is not provided just by one of the framework components but by the whole system.

In our framework the mediated access refers to the knowledge which guides a user to interact correctly with the system avoiding errors, that was one of the challenges of our framework. There are different kinds of knowledge included in our framework, and they categorize the errors managed in our system as follows:

- Related to the construction of spaces:
  - mathematical restrictions:
    - *type restrictions*: a space constructor can only be applied over objects of a concrete type (and, of course, all its subtypes); for instance, the "classifying space" constructor can only be applied over a space which is a simplicial group; and,
    - restrictions of the arguments of space constructors:
      - *independent argument restrictions*: the value of an argument of the constructor must satisfy some properties which are independent from the rest of the arguments of the space constructor; for instance, the "Moore" constructor takes p and n as arguments to construct the Moore space $M(\mathbb{Z}/p\mathbb{Z}, n)$, both p and n are restricted to be natural numbers and p must be higher than 1; those are examples of independent argument restrictions;
      - *functional dependencies*: the value of an argument of the space constructor depends on the value of another one; for instance, in the "Moore" constructor the value of n must be higher or equal than $2p-4$; that is a functional dependency.
  - *Kenzo implementation argument restrictions*: some constrains are imposed by the Kenzo implementation of spaces; for instance, the "Sphere" constructor takes as argument a natural number that Kenzo constrains to be lower than 15.
- Related to the computation of (homology and homotopy) groups:
  - *restriction of the computation dimension*: in the case of computing $H_n(X)$ the value of n must be higher or equal than 0; and in the case of computing $\pi_n(X)$ the value of n must be higher or equal than 1; and,
  - *reduction degree restrictions*: the notion of reduction degree was explained in Subsection 1.2.6. In the case of computing $H_n(X)$ the reduction degree of X must be higher or equal than 0; and in the case of computing $\pi_n(X)$ the reduction degree of X must be higher or equal than 1.

All this knowledge is spread throughout the Kenzo framework in its components. These components are going to be briefly explained in the following paragraphs.

As we have said previously, XML is the chosen technology to encode the data of our framework since it perfectly fulfills requirements 7 and 8, and also let us encode some mathematical knowledge that will be useful to manage error handling. In particular, we have defined an XML language called *XML-Kenzo*. This language is used for data interchange among the different components of the framework.

The XML-Kenzo specification is employed to represent some of the knowledge included in the framework, namely the knowledge which allows us to manage the following restrictions:

1. type restrictions,
2. independent argument restrictions of the space constructors,
3. implementation restrictions of the space constructors arguments, and
4. restriction of the dimension in computations.

On the contrary, the rest of constraints (functional dependencies of the arguments of the constructors and reduction degree restrictions) cannot be represented in XML-Kenzo; so, they have been included in a different way in the framework.

Let us present now the rest of the system components.

Kenzo itself, wrapped with an interface based on XML-Kenzo, is acting as *internal server* and is used as the core to perform computations in our framework. It is worth noting that the functionality available from the internal server is a subset of the Kenzo one. Namely, the functionality that allows us to construct spaces of regular usage and to perform computations of groups is accessible through the Internal Server.

Due to the Microkernel pattern organization new computations and deduction engines can be incorporated as internal servers; solving the third requirement, the integration with other systems.

The main component of this architecture, the microkernel, is responsible for managing all system resources, maintains information about resources and allows access to them in a coordinated and systematic way. The microkernel acting as intermediary layer is based on an XML-Kenzo processor, allowing both a link with Kenzo and including the management of constraints not handled in the XML-Kenzo specification. The functionality exposed by the microkernel is related to the Kenzo way of working, providing, on the one hand, the way of constructing spaces (construction modules) and, on the other hand, the functionality to perform computations (computation modules). Besides, the fifth requirement (efficiency) has been solved at this level by programming a *memoization* strategy, a technique also used in Kenzo, see Subsection 1.2.5. This has required to include an improvement of our own in the Microkernel pattern: an internal memory used to the optimization tasks. As a result, the waiting time is to a great extent similar to that of the original Kenzo system. Moreover, as we have already said, the sixth requirement (error handling) is partially fulfilled at the microkernel level. To be more concrete, the modules of the microkernel are in charge of validating the restrictions that were not included in the XML-Kenzo specification; that is to say, functional dependencies of the arguments of the constructors and reduction degree restrictions. In addition, processing modules provide several enhancements to the Kenzo system, for instance to compute homotopy groups. The functionality of the processing modules is not exposed by the microkernel but it is used by both construction and computation modules.

The external server exports the functionality of the microkernel, that is, the mechanisms to construct spaces and to compute groups. Moreover, the external server is in charge of validating the knowledge included in the XML-Kenzo specification. Namely, the external server is in charge of checking: type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, and restriction of the dimension in computations. Therefore, *requests* arriving to the microkernel always satisfy these restrictions; that is to say, they satisfied the XML-Kenzo specification, this kind of requests are called *valid XML-Kenzo requests*.

Finally, the *adapter* exposes the functionality provided by the external server to clients. However, due to the fact that XML-Kenzo is ad-hoc for our framework is not sensible to use this language to communicate with the outside. In order to grapple with this problem, the OpenMath XML standard [Con04] has been employed. Then, the *adapter* converts the interface provided by the external server, based on XML-Kenzo, into a more suitable interface, based on OpenMath, which can be used by different clients (for instance, graphical user interfaces, web applications or web services) without knowing the internal representation of the data of our framework, then, the fourth aspect requested to our framework, the interaction with different clients, is achieved. Besides, as OpenMath is also an XML language, requirements 7 and 8 are also satisfied at this level. The process to convert from/to OpenMath requests to/from XML-Kenzo requests is tackled by a program included in the adapter called *Phrasebook* [Con04].

The communication between the different components of the system is based on a message style model, that is, the modules of the framework are communicated in a direct and synchronous way.

The main complaint to this framework could be the restriction of the full capabilities of the Kenzo system, since just the main Kenzo functionality is included, however the interaction with it is easier and enriched.

2.2 Framework components

This section is devoted to present a detailed description of each one of the Kenzo framework components.

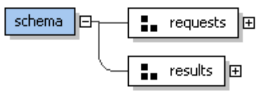


Figure 2.2: Main elements of the XML-Kenzo schema

2.2.1 XML-Kenzo

One of the most important decisions in the development of our framework was the language employed to represent the mathematical data inside the framework. The representation language should be independent of the implementation language used, and the represented data must be easily interchangeable among the different components of the framework, both current components and future extensions. These requirements oriented us towards our solution: using XML technology [B$^+$08]. Once we chose XML to represent data inside our framework, we should decide whether we extended a mathematical XML language, MathML or OpenMath, or if we defined a fresh one. *MathML* provides several facilities to represent data in the web but is not suitable to represent content, see [KSN10] for a survey about this question. *OpenMath* is very useful in the communication with the outside of the framework, but, we have to extend it with the problems associated to this task, since OpenMath is a general purpose standard which cannot be fully adapted to our needs. So, we opted for defining from scratch an XML language suited to our requirements. The new XML language was called *XML-Kenzo*.

It is worth noting that the XML-Kenzo language is employed inside the framework; on the contrary, to communicate the framework with the outside we use OpenMath since it is more suitable for that task.

The specification of XML-Kenzo is based on both Kenzo and mathematical conventions and is provided by means of an *XML schema* definition (XSD), see [E$^+$07]. The formal specification of XML-Kenzo defines the structure of the objects indicating their restrictions and providing valid combinations.

There are two types, *groups* in terminology of XML schema definitions, of elements in the XML-Kenzo specification based on the usual interaction between a client and a software system. To be more concrete, the interaction between a client and a software system consists of the user sending *requests* to the system and the system returning *results* to the client. Therefore, we have defined two types: `requests` and `results`, see Figure 2.2.

Let us focus first on the `requests` XML-Kenzo group. As we explained in Subsection 1.2.2, Kenzo includes functions with two different aims: construct spaces and perform computations. This situation is reflected in the specification of XML-Kenzo, where two elements belong to the `requests` group. Namely, the elements: `operation` to represent the computation functions and `constructor` to represent the different spaces that can be built, see Figure 2.3.

Now, let us concentrate on the `constructor` element. When a user has decided to

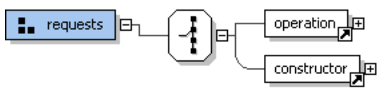


Figure 2.3: `requests` XML-Kenzo group

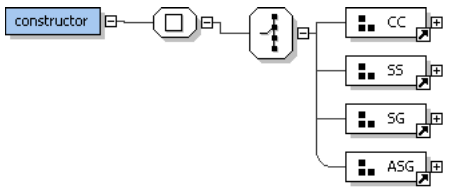


Figure 2.4: Types of spaces in XML-Kenzo

construct a space in Kenzo, he should decide which type he wants to build: a simplicial set, a simplicial group and so on (see all the possible types in Subsection 1.2.1). As was explained in Section 1.2.2, the Kenzo system provides useful functions to create interesting objects of regular usage, which can belong to four types: chain complexes, simplicial sets, simplicial groups and abelian simplicial groups. All these functions of regular usage are represented in our XML-Kenzo specification and have a unique XML-Kenzo representation. So, four types are defined in the XML-Kenzo specification: `CC` (Chain Complex), `SS` (Simplicial Set), `SG` (Simplicial Group) and `ASG` (Abelian Simplicial Group). Then the child of the `constructor` element must be an element of one of these types, see Figure 2.4. The following elements, gathered by types, represent the spaces that can be constructed in our framework (the complete list of spaces that can be constructed and their type can be seen in Figure 2.5):

- The `SS` type contains the elements that represent most of the constructors of Kenzo. Some of them construct spaces from scratch such as `sphere`, `moore-space`, `build-finite-ss`, and so on. Others construct spaces from other ones; for instance, `crts-prdc` which represents the $\underline{C}a\underline{r}\underline{t}e\underline{s}ian$ $\underline{pr}o\underline{d}u\underline{c}t$.

- The `SG` type contains the elements whose arguments are simplicial groups; that is to say, both loop spaces, `loop-space`, and classifying spaces, `classifying-space`.

- `ASG` contains the elements to construct Eilenberg MacLane spaces of type $K(\mathbb{Z}, n)$, `k-z`, and $K(\mathbb{Z}/2\mathbb{Z}, n)$, `k-z2`.

- The `CC` type contains elements that correspond with the constructors defined at the algebraic level but not at the simplicial one. The element `chain-complex` constructs a simple chain complex such as the unit chain complex or the circle. The rest of the elements of this type construct spaces from other ones, for instance `tnsr-prdc` which represents the $\underline{t}e\underline{ns}o\underline{r}$ $\underline{pr}o\underline{d}u\underline{c}t$.

As we said previously, XML-Kenzo can be used to represent knowledge and restrictions about its elements. We commented in Section 2.1 that there are three kinds of

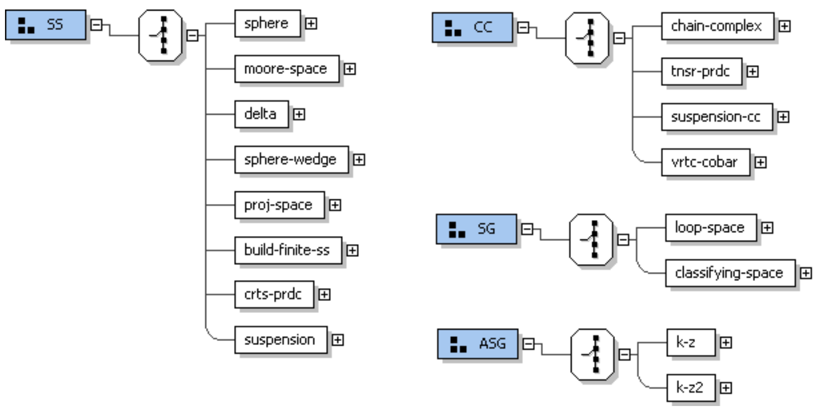


Figure 2.5: Constructor groups in XML-Kenzo

restrictions related to the construction of spaces that are dealt with in the XML-Kenzo specification; namely, type restrictions, independent argument restrictions of the space constructors and implementation restrictions of the space constructors. Let us present how we handle these restrictions in the specification of the XML-Kenzo language.

Let us focus first on type restrictions. These constraints are applied over the constructors of spaces from other ones. As we explained in Subsection 1.2.1, Kenzo is, in its pure mode, an untyped system (or rather, a dynamically typed system), inheriting its power and its weakness from Common Lisp. Thus, for instance, in Kenzo a user could apply a constructor to an object without satisfying its input specification. For instance, the method constructing the classifying space of a simplicial group could be called with an argument which is a simplicial set without a group structure over it. Then, at runtime, Common Lisp would raise an error informing the user of this restriction. This is shown in the following fragment of a Kenzo session.

```
> (setf s4 (sphere 4)) ✠
[K1 Simplicial-Set]
> (classifying-space s4) ✠
Error: No methods applicable for generic function #<STANDARD-GENERIC-FUNCTION
CLASSIFYING-SPACE> with args ([K1 Simplicial-Set]) of classes (SIMPLICIAL-SET)
[condition type: PROGRAM-ERROR]
```

With the first command, we construct the sphere of dimension `4`, a simplicial set. Thus, when in the second command we try to construct the classifying space of a simplicial set, the Common Lisp Object System (*CLOS*) raises an error.

This kind of error is controlled in our framework thanks to the XML-Kenzo specification, since the inputs for the operations between spaces can be only selected among the spaces with suitable characteristics (Figure 2.6 shows the specification of the `classifying-space` element in XML-Kenzo, this element only allows as child an element either from the `SG` or the `ASG` group). This enriches Kenzo with a small (semantical) type system.

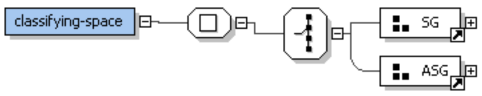


Figure 2.6: `classifying-space` specification in XML-Kenzo

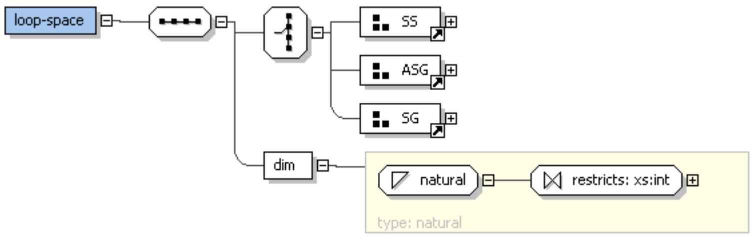


Figure 2.7: `loop-space` specification in XML-Kenzo

It is worth noting that the inheritance relations between Kenzo types, see Section 1.2.1, cannot be directly specified using an XML schema, so, we tackle this situation in the following way. If we have the types A and B, represented in the XML schema as the groups `A` and `B` respectively, where B inherits from A and a function f that can be applied over the objects of the type A, and of course also over the objects of type B, then the element of the XML schema that represents the function f has as child an element that belongs either to the `A` or the `B` group, and this must be included in the schema in an explicit way. For instance, the `loop-space` Kenzo function is applied over a simplicial set, but, as can be seen in Figure 1.2 of Subsection 1.2.1, simplicial groups and abelian simplicial groups are subtypes of simplicial sets, so the `loop-space` element of XML-Kenzo has as child an element belonging either to the `SS`, the `SG` or the `ASG` group. Moreover, it also has a child that represents the dimension of the loop space, as can be seen in Figure 2.7.

Both second and third kinds of restrictions, that are, independent argument restrictions of the space constructors and implementation restrictions of the space constructors, are also coped with in the XML-Kenzo specification. These restrictions are always applied over objects constructed from scratch, such as spheres, Moore spaces, Eilenberg MacLane spaces and so on. The restrictions over the arguments of those functions are translated into the restrictions over the elements encoding them. For instance, spheres only have sense if their dimension is a natural number (an independent argument restriction). In addition, the function that constructs a sphere in Kenzo has as argument a natural number n, such that $0 < n < 15$; this restriction is included in the XML-Kenzo specification of the `sphere` element as is shown in Figure 2.8. This kind of restrictions can be included in the specification of the XML-Kenzo language without any special hindrance.

On the contrary, functional dependencies of the arguments cannot be imposed in the XML-Kenzo specification, since restrictions about the value of an element depending on the value of other elements cannot be defined in XML schemas.

Once we have presented the way of encoding Kenzo constructors in XML-Kenzo, we

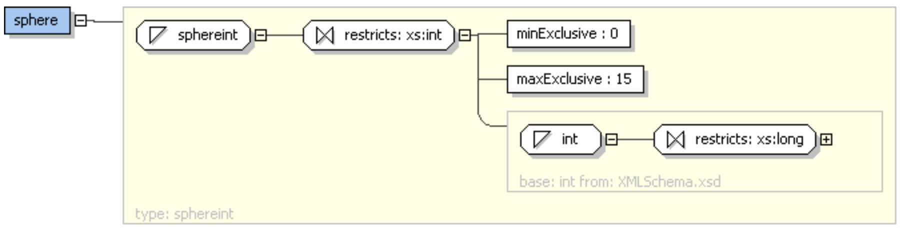


Figure 2.8: `sphere` specification in XML-Kenzo

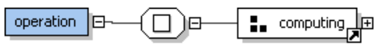


Figure 2.9: `operation` element in XML-Kenzo

can construct different XML-Kenzo objects such as the space $\Omega^3(S^4)$ which is represented as the following XML-Kenzo object:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

It is worth noting that the type of the elements (`requests`, `SG` and `SS` for the `constructor`, `loop-space` and `sphere` elements respectively) is not included in the XML-Kenzo object since it is implicit to the element (an element defined in an XML schema only belongs to one type). From now on, we will call *construction requests* to the XML-Kenzo objects whose root element is `constructor`.

Let us focus now on the other element of the `requests` group: `operation`. The `operation` element represents requests which ask to the system the computation of homology and homotopy groups. The `operation` element has as child one element of the `computing` group, see Figure 2.9. The computation of homology and homotopy groups is represented by means of the elements `homology` and `homotopy` respectively. Both elements belong to the `computing` group, see Figure 2.10. The structure of both `homology` and `homotopy` elements is the same, they have two children, the first one is a space of one of the groups `CC`, `SS`, `SG` or `ASG`; and the second one is an element called `dim`.

As we commented in Section 2.1, we can handle one of the restrictions related to computations, that is the restriction of the dimension in homology and homotopy computations. This is translated in the XML-Kenzo specification into a constraint of the value of the `dim` element of both `homology` and `homotopy` elements. Nevertheless, the other restriction related to computations (the reduction degree restriction of the space) cannot be managed at this level since its value must be computed from the description of the space.

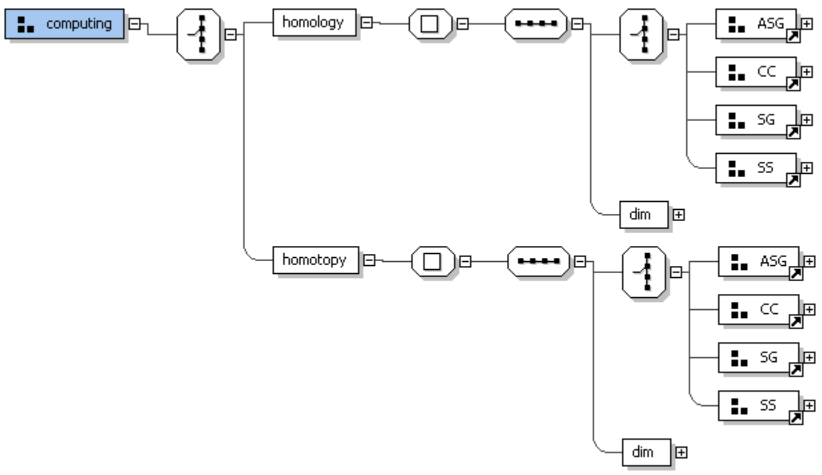


Figure 2.10: `computing` group of XML-Kenzo

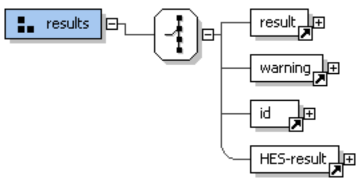


Figure 2.11: Elements of XML-Kenzo `results` group

Once we have presented the way of encoding Kenzo operations in XML-Kenzo, we can construct different XML-Kenzo objects such as the operation $H_5(\Omega^3(S^4))$ which is represented by the following XML-Kenzo object

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  </homology>
</operation>
```

From now on, we will call *computation requests* to the XML-Kenzo objects whose root element is `operation`. We will call *valid XML-Kenzo requests* to the XML-Kenzo objects of `requests` type which satisfy the XML-Kenzo specification. This finishes the description of the elements of the `requests` XML-Kenzo group.

Let us present now the `results` XML-Kenzo group. We have defined four elements belonging to this group: `id`, `warning`, `result` and `HES-result`, see Figure 2.11.

In our framework, objects have associated a unique identification number. To communicate this identification among the modules of the framework we use the `id` element whose value is a natural number (the unique identification number).

```
<id> 1 </id>
```

The `warning` element is used to represent errors returned by the framework and its value is a string with the correspondent error in natural language.

```
<warning> The dimension of the sphere must be a natural number </warning>
```

The `result` element is used to return the result of a computation by means of an undefined number of `components` whose value is a natural number.

```
<result>
   <component> 2 </component>
   <component> 3 </component>
</result>
```

If the above `result` element is returned, that means that the result is $\mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}/3\mathbb{Z}$.

Finally, `HES-result` provides a result, by means of an undefined number of components whose value is a natural number, and an explanation of the reasoning followed to obtain the result, by means of a string.

```
<HES-result>
 <component>0</component>
 <explanation>
    The space was a contractible space since it was the cartesian product of
    two contractible spaces. The homotopy groups of a contractible space are
    always null. Then, the homotopy group of the space in dimension 3 is null.
 </explanation>
</HES-result>
```

To sum up, the XML-Kenzo schema specifies two kind of objects: requests and results. There are two kinds of requests: to construct spaces and to compute groups. Moreover, most of the restrictions of the functions devoted to construct spaces and compute groups are imposed in XML-Kenzo, in this way some knowledge is provided with the XML-Kenzo specification.

As we will see throughout the next subsections, XML-Kenzo played a key role in the development of the rest of the framework components.

The complete specification of the XML-Kenzo language can be seen in [Her11].

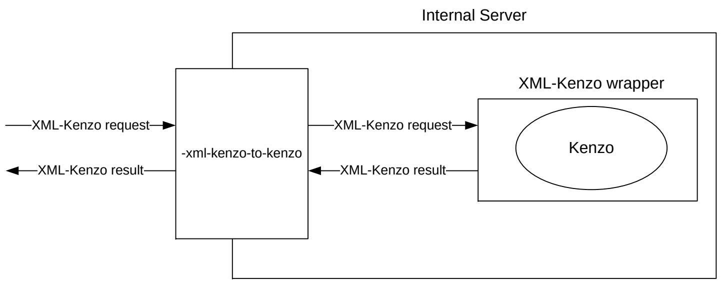


Figure 2.12: Internal Server of the Kenzo framework

2.2.2 Internal Server

The *internal server* is a Common Lisp *component* that provides access to the Kenzo functionality through an XML-Kenzo interface. This module is split in three parts (see Figure 2.12): the full Kenzo system, an XML-Kenzo wrapper for Kenzo, that is a bunch of Common Lisp files that defines a Kenzo interface based on XML-Kenzo, and also the transformation from XML-Kenzo objects to their Kenzo encoding and viceversa; and last but not least, an *interface* which offers a *service* to use Kenzo through the XML-Kenzo wrapper.

We have talked at length about Kenzo, see Section 1.2; so let us focus on both the interface and the XML-Kenzo wrapper. The interface provides a service called `xml-kenzo-to-kenzo` which allows one to access to the functionality exported by the XML-Kenzo wrapper. The XML-Kenzo wrapper provides access to a subset of the Kenzo functionality, namely the functionality specified in the XML-Kenzo language by means of the elements of the `requests` group. Hence, from this interface we can construct the topological spaces of regular usage in Kenzo (such as spheres, Moore spaces, loop spaces and so on), and compute homology and homotopy groups.

The workflow when the `xml-kenzo-to-kenzo` service is invoked is as follows. If the request is a construction request, the internal server transforms the XML-Kenzo request into a Kenzo instruction by means of a function called `xml-kenzo-to-kenzo`. Afterwards, the Kenzo instruction is executed in the Kenzo kernel, and then as a result a Kenzo object is obtained. The result returned by the internal server is the unique identifier of the Kenzo object, the value of the slot `idnm` of a Kenzo instance (see Section 1.2.1). This identification number is returned by means of an `id` XML-Kenzo object. For instance, if the internal server receives the request:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

the instruction `(loop-space (sphere 4) 3)` is executed in the Kenzo kernel. As a result an object of the Simplicial Group class is constructed, and the identifier of that object is returned, in a fresh Kenzo session the result will be:

```
<id>30</id>
```

In the case of computation requests, the internal server transforms the XML-Kenzo request into a Kenzo instruction by means of the `xml-kenzo-to-kenzo` function. Afterwards, the Kenzo instruction is executed in the Kenzo kernel, and as a result a group is obtained. The result returned by the internal server is the group codified in a `result` XML-Kenzo object. For instance, if the internal server receives the request:

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  </homology>
</constructor>
```

the instruction `(homology (loop-space (sphere 4) 3) 5)` is executed in the Kenzo kernel. As a result, the group $\mathbb{Z}/2\mathbb{Z}$ is obtained and returned by the internal server using an XML-Kenzo `results` object whose root is `result`.

```
<result>
  <component>2</component>
</result>
```

The answer returned by the internal server is always an identifier, in the case of a *construction request*, or a group, in the case of a *computation request*. What we want to highlight is that errors never happen since all the dangerous situations are stopped in a more external level of the framework (some of them were dealt with in the specification of XML-Kenzo, others will be handled in the microkernel), as we will see in the following subsections, and therefore, the requests received by the internal server are always safe.

The main disadvantage of the internal server with respect to Kenzo is the restriction of the full capabilities of the Kenzo kernel. Let us note that the restriction is only related to the construction of spaces where, as we explained in Subsection 2.2.1, only the spaces that can be constructed using the Kenzo ad-hoc functions are available; on the contrary, the Kenzo capability to perform computations remains untouched. Nevertheless, thanks to the combination of the internal server and XML-Kenzo the interaction between the modules of the framework, that can be programmed in a language different from Common Lisp, and Kenzo is easier since the communication is performed by means of XML-Kenzo objects and the internal server is in charge of converting them from/to Kenzo instructions.

2.2.3 Microkernel

The *microkernel* is the main component of the framework, it is devoted to the management of resources and includes some of the knowledge of the framework (the rest of knowledge was provided by the XML-Kenzo specification). The microkernel is split in four constituents (see Figure 2.1): the construction modules, the computation modules, the processing modules and the internal memory. The interaction with the microkernel is provided by means of an *interface* based on XML-Kenzo. This *interface* only gives access to the functionality of the construction and computation modules.

2.2.3.1 Internal Memory

The first component of the microkernel that we are going to describe is the *internal memory*. The *internal memory* is a Common Lisp module which stores both the constructed spaces and the computed results. That is to say, the internal memory stores the state of the framework.

The storage of constructed spaces and computed results avoids unnecessary communications between the microkernel and the internal server. Therefore, the efficiency of the framework is improved. In this component, for both constructed spaces and computed results, a *memoization* strategy has been implemented.

Let us focus first on the spaces constructed in the microkernel.

At this moment, the Kenzo framework just interacts with an internal server that is the Kenzo system. However, in the future we want to include different internal servers. This means that objects of very different nature (such as spaces or groups) are going to coexist in the Kenzo framework and, in particular, in the microkernel.

Taking this question into account, we decided to design a class hierarchy with a main class; and specialize this class with different subclasses for the objects coming from the different internal servers.

Therefore, we have defined the main class to represent microkernel objects, `MK-OBJECT`, whose definition is:

```
(DEFCLASS MK-OBJECT ()
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *number-of-objects*) :reader idnm)
     ;; ORiGiN
     (orgn :type string :initarg :orgn :reader orgn)))
```

This class has two slots (which are common for all the microkernel objects):

- `idnm`, an integer being the object identifier for this system. This is generated by the microkernel in a sequential way, each time a new object is created.
- `orgn`, a string containing the XML-Kenzo object that is the *origin* of the space. This comment is unique and is important, because when a module constructs a new `mk-object` instance, it uses the XML-Kenzo string to search in a specific list, `*object-list*`, if the object has not been already built. So, one avoids the duplication of instances of the same object.

The `*object-list*` list stores the state of the microkernel related to the constructed objects.

As we have said, we are going to have different specializations of the `MK-OBJECT` class, but at this moment we just have one devoted to represent Kenzo spaces in the microkernel. Namely, we have defined a subclass of the `MK-OBJECT` class, the class `MK-SPACE-KENZO`, whose definition is:

```
(DEFCLASS MK-SPACE-KENZO (MK-OBJECT)
    (;; REduction DEgree
     (rede :type fixnum :initarg :rede :reader rede)
     ;; Kenzo IDentification NuMber
     (kidnm :type fixnum :initarg :kidnm :reader idnm)
```

This class has two additional slots to the ones of the `MK-OBJECT` class:

- `rede`, an integer giving the reduction degree, see Subsection 1.2.6, of the space.
- `kidnm`, an integer being the object identifier inside the Kenzo system.

To sum up, Kenzo spaces are represented in the microkernel as instances of the `MK-SPACE-KENZO` class. In addition, as objects of the microkernel they are stored in the `*object-list*` list.

Let us explain now the memoization strategy implemented in the internal memory to handle computed results. As we have seen in Subsection 1.2.5, computing some groups in Kenzo can require a substantial amount of time. Kenzo is the kernel of our framework, so this situation also happens in our framework. Therefore, it seems desirable to store the computations of groups somewhere once they have been computed, in order to avoid the re-computation in Kenzo. The microkernel stores the groups associated with a concrete space in the internal memory when a computation is executed for the first time, and if this computation is executed again later, the group is simply looked up and returned, without further execution.

This means that the behavior of the functions which compute the groups depends on whether the asked group for an space has been already computed. Otherwise, the group must be stored after it has been calculated. These two extra tasks are done by two pair of functions that are implemented in the computation modules, namely the *testers* and the *setters*. The testers take as arguments the identifier of the object, that is stored in the `idnm` slot of the `mk-space-kenzo` instance, and the dimension `n` of the group and return the result or `nil` according to whether the `n`-th homology (or homotopy) group of the space whose identifier is `idnm` had already been stored. The setters take as arguments the identifier `idnm` of the object, the dimension `n`, and the result `result` of computing the `n`-th homology (or homotopy) group of the object `idnm` and put the result into the internal memory, where the result can directly look it up. The main procedures, implemented in the computing modules, are called the *getters*, and from the preceding discussion it is seen that there must really be at least two methods for the getters. One method is used when the tester returns `nil`; it is the method which first does the real computation by means of the Kenzo functions and then executes the setter with the computed value. A second method is used when the tester does not return `nil`; it simply returns the stored value.

Up to now, we have explained the implemented strategy but not the way of storing the results. This is an important issue for efficiency reasons and has been tackled by means of efficient data structures, namely *hash tables* and arrays.

To store computed results, we use two hash tables, one for the computations related to homology groups and another one for the computations of homotopy groups. The identification number of each object, slot `idnm` of the `mk-object` class, acts as key into the hash tables. The value associated to each key is an array where the n-th homology (or homotopy) group of the object associated with the identification number is stored in the position n.

Then, if we try to compute several times the fifth homology group of the space $\Omega^3(S^4 \times S^4)$, we can see the profit of using the memoization technique in the microkernel. In the first computation, the microkernel takes almost 22 minutes, but in a further trial the result is returned in just a second since it has been stored in the internal memory.

Therefore, thanks to this memoization strategy implemented in the internal memory the framework efficiency increases.

It is worth noting that the values stored in the hash tables are not kept from one session to another. The reader can wonder the reason to not reuse computations from a previous session storing the results in a persistent way. At least two different approaches can be considered to manage results from other sessions.

- We could keep on using our hash tables to store results, the main improvement would consist of saving the results stored in the hash tables at the end of the session and loading the stored results at the beginning of each session. Advantage: we use hash tables that are efficient data structures. Disadvantage: when the number of results increase, the time to load all the results in memory and the amount of memory used increase, too.

- We could use a database to store in a persistent way the results instead of keeping the results in hash tables. Advantage: efficient persistence storage of results. Disadvantage: since we are working with XML data, our database should be an XML database, and this kind of databases are usually big and the search of results in these databases is less efficient than the search in hash tables.

Thus, additional research is needed to achieve an efficient storage of results to be reused in different sessions. Therefore, at this moment we use the approach previously presented without saving results from one session to another.

The state of the microkernel is obtained gathering both the `*object-list*` list of constructed spaces and the two hash tables of computed results.

2.2.3.2 Processing Modules

The processing modules are three Common Lisp modules that are in charge of providing some enhancements to the construction of spaces and the computation of groups. The functionality of these modules is not available outside the microkernel but it is used by both construction and computation modules.

2.2.3.2.1 Reduction degree module The reduction degree module is in charge of managing the reduction degree of spaces, see Subsection 1.2.6. Kenzo allows the user to construct spaces whose reduction degree is lower than 0 (as the loop space iterated three times of the sphere of dimension 2). In these spaces some operations (for instance, the computation of the set of faces of a simplex) can be achieved without any problems. On the contrary, theoretical results ensure that their homology groups are not of finite type, and then they cannot be computed. In pure Kenzo, the user could ask for a homology group of such space, catching a runtime error, as is shown in the following fragment of a Kenzo session:

```
> (setf o2s2 (loop-space (sphere 2) 3)) ✠
[K18 Simplicial-Group]
> (homology o2s2 1) ✠
Error: ‘NIL’ is not of the expected type ‘NUMBER’
[condition type: TYPE-ERROR]
```

The reduction degree module is used to avoid this kind of errors in our framework. To this aim, the reduction degree module is implemented as a small expert system, computing, in a symbolic way (that is to say, working with the XML-Kenzo description of the spaces, and not with the spaces themselves), the reduction degree of the space. The set of rules that allows the reduction degree module to obtain the reduction degree of a space was given in Subsection 1.2.6, and the computational price of using this small rule-based system is negligible with respect to ordinary computations in Algebraic Topology.

As we have seen in Subsubsection 2.2.3.1, the reduction degree of a space will be assigned to the `rede` slot of the `mk-space-kenzo` instance in the construction of the object.

The relevance of the reduction degree, and hence of this module, is due to the computation modules, since when a computation is demanded, the system monitors if the reduction degree of the space allows the computation of the homology or homotopy group, or whether a warning must be returned as answer. In this way, the dangerous requests related to the reduction degree are always stopped at the microkernel and never arrive to the internal server, avoiding operations which would raise errors.

2.2.3.2.2 Homotopy algorithm module The homotopy algorithm module (from now on, HAM) is in charge of chaining methods in order to compute some homotopy groups. In pure Kenzo, there is no final function allowing the user to compute homotopy groups. Instead, there is a number of complex algorithms, allowing a user to chain them to get some homotopy groups. The HAM is in charge of chaining the different algorithms presented in Kenzo to reach the final objective.

Moreover, Kenzo, in its current version, has limited capabilities to compute homotopy groups (depending on the homology of Eilenberg MacLane spaces that are only partially implemented in Kenzo), so the *chaining* of algorithms cannot be universal. Thus the homotopy algorithm module processes the call for a homotopy group, making some requests to the Kenzo kernel (computing some intermediary homology groups, for instance) before deciding if the computation is possible or not.

Let us present the procedure implemented in HAM. That algorithm and its underlying mathematics were presented in [Rea94].

Algorithm 2.1.

1. Find the lower dimension $0 < r \leq n$, such that the r-th homology group of X is not null.
    (a) If r does not exist (i.e., $H_q(X) = 0$ for all $0 < q \leq n$), $\pi_n(X)$ is null (applying the Hurewicz theorem, see [Whi78]).
    (b) If $r = n$, $\pi_n(X) = \pi_r(X) = H_r(X)$ (applying the Hurewicz theorem, see [Whi78]).
    (c) If $r < n$,
        i. If the r-th homology group is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$ go to step 2.
        ii. Otherwise, return a `warning` XML-Kenzo object informing that the homotopy group is not computable in the current version.

2. Compute the r-th fundamental cohomology class of X, $h_r = [\chi_r] \in H^r(X, \pi_r)$, where $\pi_r = \pi_r(X) = H_r(X)$.

3. Build the fibration over X canonically associated to the above cohomology class:

$$\begin{tikzcd} K(\pi_r, r-1) \arrow[d] \\ T \equiv X' \equiv X^{(r+1)} = X \times_{\tau_{\chi r}} K(\pi_r, r-1) \arrow[d] \\ X \end{tikzcd}$$

4. Get the total space $T = X^{(r+1)}$ associated to the above twisting operator.

   (a) If $r + 1 = n$, $\pi_n(X) = H_n(T)$.

   (b) If $n > r + 1$ and $H_{r+1}(T) = \mathbb{Z}$ or $H_{r+1}(T) = \mathbb{Z}/2\mathbb{Z}$, increase one unit the value of r and go to step 2 but using the space T instead of X.

This is the algorithm implemented in HAM.

2.2.3.2.3 Homotopy expert system As we explained in the previous paragraph, the HAM is able to compute some homotopy groups of some spaces by means of the algorithm presented in [Rea94]. Nevertheless, there are several homotopy groups that are not reachable by the Kenzo framework using the HAM (the implementation of the algorithm is limited to the spaces which first non null homology group is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$). To overcome this gap, we undertook the task of developing a homotopy *expert system* (from now on, HES) from the extensive literature about this topic; taking profit of theoretical knowledge contained in theorems. The knowledge (that is, the theorems about homotopy groups) can be found in different books, for instance, [CM95, Mau96, Hat02].

The HES is a rule-based system [GR05]. Rule-based systems do not represent knowledge in a declarative and static way (as a bunch of things that are true), but they represent knowledge in terms of a bunch of rules that tell what you can conclude in different situations. Rule-based systems have been employed in a wide variety of contexts, such as the discovery of molecular structures [LBFL80], the identification of bacterias which cause severe infections [BS84] or to configure computer systems [McD82].

The structure of a rule-based expert system, see [GDR05], consists of, and the HES is no exception, the following components (see Figure 2.13):

- the *Working memory (the facts)*, representing what we know at any time about the problem we are working at,

- the *Knowledge base (the rules)*, containing the domain specific problem-solving knowledge,

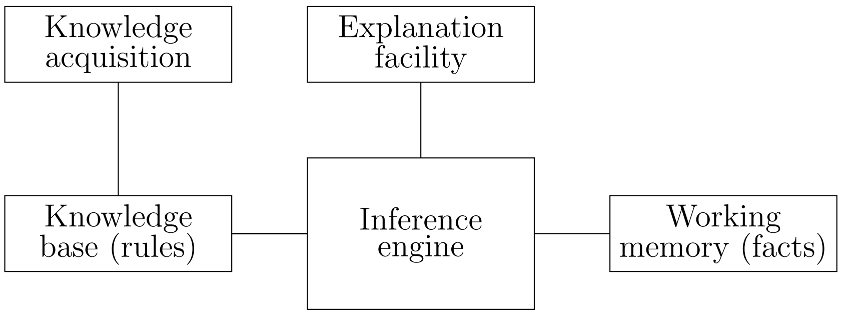


Figure 2.13: Structure of a rule-based expert system

- the *Inference engine*, a general program that activates the knowledge in the knowledge base. This program depending on the facts applies different rules to obtain a conclusion.
- a *Knowledge acquisition* module, allowing one to acquire and edit the knowledge base,
- an *Explanation facility* module, allowing the user to understand how the expert system obtains the results.

Let us present each component for the particular case of the HES.

The working memory represents what we know at any time about the problem we are working at by means of facts. There exist two kinds of facts in the HES: *static* and *dynamic*.

Static facts are properties associated with the spaces. These properties are known at the moment of the construction of the object. Some examples of this kind of facts are:

Fact 1.
$$\forall n \in \mathbb{N}: \quad \Delta^n \text{ is a contractible space.}$$

Fact 2. If $X = A \times B$ where A and B are contractible spaces, then,
$$X \text{ is a contractible space.}$$

Fact 3.
$$\forall n \in \mathbb{N} \text{ and } G \text{ group}: \quad K(G,n) \text{ is an Eilenberg MacLane space of type } (G,n)$$

Fact 4. If $X = B(Y)$ where Y is an Eilenberg MacLane space of type (G,n), then,
$$X \text{ is an Eilenberg MacLane space of type } (G,n+1)$$

At this moment, the static facts determine the spaces that are *contractible spaces*, *spheres*, *Eilenberg MacLane spaces* and *Loop spaces of spheres.* The way of implementing this issue in the microkernel is by means of four subclasses of the `mk-space-kenzo` class, presented in Subsubsection 2.2.3.1; this class hierarchy will be used to determine which rules can be applied.

To store spaces which represent spheres, we use a new subclass called `MK-SPACE-SPHERE`, whose definition is:

```
(DEFCLASS MK-SPACE-SPHERE (MK-SPACE-KENZO)
    (;; DIMension
     (dim :type fixnum :initarg :dim :reader dim))
```

this class has an additional slot, to the ones of the `MK-SPACE-KENZO`, called `dim`, which represents the dimension of the sphere.

To store contractible spaces, we use a new subclass called `MK-SPACE-CONTRACTIBLE`, whose definition is:

```
(DEFCLASS MK-SPACE-CONTRACTIBLE (MK-SPACE-KENZO) ())
```

this class does not have any additional slot to the ones of the `MK-SPACE-KENZO` class.

To store spaces which represents Eilenberg MacLane spaces, we use a new subclass called `MK-SPACE-K-G`, whose definition is:

```
(DEFCLASS MK-SPACE-K-G (MK-SPACE-KENZO)
    (;; ITERation
     (iter :type fixnum :initarg :iter :reader iter))
    (;; GROUP
     (group :type fixnum :initarg :group :reader group))
```

this class has two additional slots to the ones of `MK-SPACE-KENZO`: (1) `iter`, which represents the number of iterations of the Eilenberg MacLane space, and (2) `group`, which represents the group of the Eilenberg MacLane space.

To store spaces which represent loop spaces of spheres, we use a new subclass called `MK-SPACE-LS-SPHERE`, whose definition is:

```
(DEFCLASS MK-SPACE-LS-SPHERE (MK-SPACE-KENZO)
    (;; ITERation
     (iter :type fixnum :initarg :iter :reader iter))
    (;; DIMension Sphere
     (dims :type fixnum :initarg :dims :reader dims))
```

this class has two additional slots to the ones of `MK-SPACE-KENZO`: (1) `iter`, which represents the number of iterations of the Loop space, and (2) `dims`, which represents the dimension of the sphere. When, the `Loop Space` module constructs a `MK-SPACE-LS-SPHERE` instance the value of the `iter` slot is the number of iterations of the Loop Space and the value of `dims` is the value of the `dim` slot of the `MK-SPACE-SPHERE` object component of the Loop Space.

In the future, if we include new static facts which determine other types of spaces, we only need to define a new subclass of the `mk-space-kenzo` class for that concrete kind of spaces.

As we have said, the static facts are properties that are known at the moment of the construction of the objects. On the contrary, *dynamic* facts are properties that are obtained from computations. For instance:

Fact 5.

$$\text{For } n = 4: \quad \pi_n(S^3) \text{ has been computed previously and its value is } \mathbb{Z}/2\mathbb{Z}.$$

Fact 6.

$$\text{For } 0 \leq n < 4: \quad H_n(S^4) \text{ is null.}$$

The storage of dynamic facts was already explained in this memoir, namely in Subsubsection 2.2.3.1 where we talked at length about the storage of results in the internal memory of the microkernel. Then, when a new result is obtained, it is automatically saved in the internal memory to avoid re-computations. The HES has access to this internal memory, and therefore to dynamic facts.

Let us present now, the *knowledge base* of our HES. The rules in the HES represent possible results to take when specified conditions hold on items of the working memory. They are sometimes called condition-action rules, since they are `IF-THEN` rules.

The current knowledge base consists of 23 rules (the complete list can be seen in [Her11]) which are related to *contractible spaces*, *spheres*, *Eilenberg MacLane spaces* and *Loop spaces of spheres.* Let us present here some of them:

Rule 1.

IF X is a contractible space
AND $n \geq 1$
THEN $\pi_n(X) = 0$

Rule 2.

IF X is an Eilenberg MacLane space of type (G, n)
AND $n = r$
THEN $\pi_n(X) = G$

Rule 3.

IF X is the sphere S^2
AND $\pi_n(S^3)$ has been computed previously
THEN $\pi_n(X) = \pi_n(S^3)$

Both Rules 1 and 2 are applied over static facts; however, the application of Rule 3 involves a dynamic fact, namely the knowledge of a previously computed homotopy group. Therefore, some rules can only be applied when some computations have been previously performed.

These rules are used by the *inference engine* of the HES which uses the *forward chaining* method for reasoning [GR05]. This method starts with the available data and uses inference rules to extract more data until a goal is reached. Let us consider an example. We want, for instance, to compute $\pi_3(\Delta^4 \times \Delta^5)$. Then the inference engine proceeds as follows:

1	Δ^4 is a contractible space,	Fact 1
2	Δ^5 is a contractible space,	Fact 1
3	$\Delta^4 \times \Delta^5$ is a contractible space,	Fact 2 (1,2)
4	$3 = n \geq 1$	
$\therefore$	$\pi_3(\Delta^4 \times \Delta^5) = 0$	Rule 1 (3,4)

In the case of having conflicts between some rules, the method which has been used to solve this problem consists of using the rule with highest priority based on our mathematical knowledge. The priority is established by placing the rules in an appropriate order in the knowledge base. This strategy works properly for small expert systems, see [BS84], as our HES. This conflict resolution strategy has worked in our system; however, we are aware of the necessity of a more sophisticated strategy to deal with the conflicts in the HES when the number of rules increases.

The inference engine has been implemented by means of a very powerful tool of Common Lisp: the combination of generic functions and methods [Gra96]. When the class system and the functional organization of Common Lisp are considered, the notions of *generic functions* and *methods* are normally used. A *generic function* is a functional object whose behavior will depend on the class of its arguments; a generic function is defined by a `defgeneric` statement. The code for a generic function corresponding to a particular class of its arguments is a *method* object; each method is defined by a `defmethod` statement. We have a generic function:

```
(DEFGENERIC homotopy-HES (object n))
```

which has assigned concrete methods for the different spaces with special characteristics. Namely, we have four methods for the generic `homotopy` function, one for each one of the subclasses of the `mk-space-kenzo` class, extending their functionality:

```
(DEFMETHOD homotopy-HES  ((contractible mk-space-contractible) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((sphere mk-space-sphere) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((k-g mk-space-k-g) n) ...)
```

```
(DEFMETHOD homotopy-HES  ((ls-s mk-space-ls-sphere) n) ...)
```

Each one of these methods uses the concrete rules of the HES for each one of the spaces with special characteristics to obtain their homotopy groups. As we have said previously, other subclasses can be defined in the future to represent other kinds of spaces; then, if we add rules for that kinds of spaces, we only need to define new methods (one per each new kind of space) to manage the computation of homotopy groups for them.

The body of those methods is a *conditional* statement using the Common Lisp `cond` instruction. Each one of the `cond` options represents a rule of the HES. For instance, the code of the method related to the `mk-space-k-g` objects is as follows:

```
(defmethod homotopy ((k-g mk-space-k-g) n)
  (cond ((equal (iter k-g) n)
         (explanation-facility-module k-g n
                    (format nil "<component>~A</component>" (group k-g))))
        ((not (equal (iter k-g) n))
         (explanation-facility-module k-g n "<component>0</component>"))))
```

Each one of the clauses of the conditional statement represents one rule of the HES. Then, if we want to add new rules to the knowledge base, we only need to include more clauses at the end of the conditional statement. This is the knowledge acquisition mechanism of our HES. We are aware of the necessity of including a different *knowledge acquisition mechanism* in order to provide a way of adding new rules at runtime and without modifying the source code. But, at this moment, this elementary organization has been shown sufficient.

Last but not least, we have an *explanation facility module* that is a mechanism to explain the reasoning which the expert system has followed in order to get a conclusion. This mechanism is provided to the user and is implemented as a generic function.

```
(DEFGENERIC explanation-facility-module (space n result))
```

This generic function takes as argument an object X, a natural number n, and the value of the $\pi_n(X)$ group. For each one of the subclasses of the `mk-space-kenzo` class we have a concrete method which is able to explain the followed reasoning by the HES to obtain the result. With that information, the `explanation-facility-module` function generates a `HES-result` XML-Kenzo object which is the returned result by the HES.

Gathering the functionality of the HES and the HAM, our framework is able to compute some homotopy groups. In addition, we can also combine the HES and the HAM to obtain other homotopy groups as we are going to show in the following paragraph.

2.2.3.2.4 Integration of the HES and the HAM It is worth noting that if we disable either the HES or the HAM in the microkernel, our framework can keep on computing homotopy groups with the other one. Then, they can be considered as independent modules. However, if we enable both modules, the HAM and the HES cooperate to compute homotopy groups of spaces.

On the one hand, the HAM is used as a last appeal tool in the HES. That is to say, if none of the rules of the HES can be applied to obtain a homotopy group, the HES invokes the homotopy algorithm module to check if it is able to obtain some result applying the algorithm.

On the other hand, the homotopy algorithm module is used to obtain some results that are used as dynamic facts of the objects. For instance, if we want to compute $\pi_i(S^3)$ for $i = 4, 5, 6$ the HES does not have any rules which can help. Then, the HES invokes the HAM to perform the computations and stores the results in the internal memory of the microkernel. Subsequently, if we want to compute $\pi_i(S^2)$ for $i = 4, 5, 6$, the HES can use Rule 3 and the previously computed homotopy groups of the sphere S^3 to obtain the homotopy groups of the sphere S^2; in this way the HES and the HAM interact. Of course, the microkernel could have also used the HAM to compute $\pi_i(S^2)$ for $i = 4, 5, 6$; but the option of using the HES system is better because of the complexity of the algorithm implemented in HAM.

2.2.3.3 Construction Modules

Construction modules provide the functionality to construct topological spaces in the microkernel. In particular, they implement the procedures to construct the spaces associated with the elements that are children of the `constructor` element of the XML-Kenzo specification. Moreover, they are in charge of checking the Kenzo restrictions that were not included in the XML-Kenzo specification for the constructors of spaces. That is to say, the functional dependencies over the arguments.

It is worth noting that there are two kinds of functional dependencies over the arguments. On the one hand, if there are at least two arguments and the value of one of them depends on the value of another one, then, we have a functional dependency (for instance, in the "Moore" constructor the value of its argument n must be higher or equal than two times plus four its argument p). On the other hand, if there is just one argument, but with the singularity of being compound, and the value of one of the elements of this compound argument depends on the value of another one, then, we also have a functional dependency (for instance, in the "finite-ss" constructor the value of its argument is a list of elements which must determine a simplicial set; then, this com-

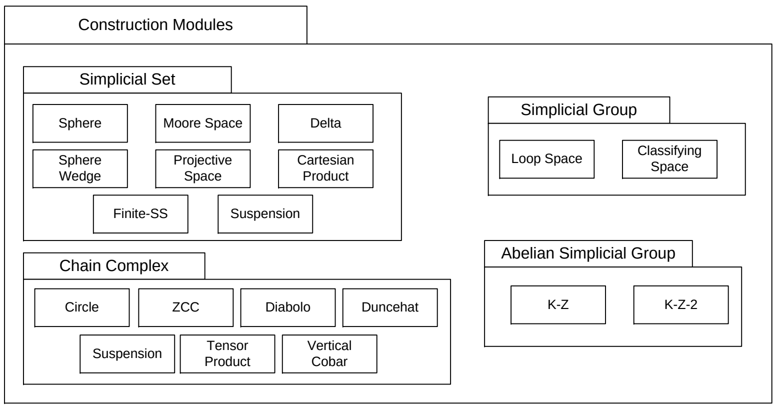


Figure 2.14: Construction modules

pound argument is restricted to the values of its components). Both kinds of functional dependencies are dealt with in the microkernel, namely in the construction modules.

Following the XML-Kenzo specification of the `constructor` element, we have organized construction modules in four blocks, one block per each type of constructor specified in XML-Kenzo (`CC`, `SS`, `SG` and `ASG`). Besides, each one of these blocks has a module for each one of the elements of the respective type of constructor, see Figure 2.14. All these modules are implemented in Common Lisp.

When the microkernel receives a construction request the functionality of the module associated with the child of the `constructor` element is invoked. For instance, if the microkernel receives the request:

```
<constructor>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
</constructor>
```

the functionality of the `Loop Space` module is invoked.

When a construction module is invoked three situations are feasible: (1) a new space is created in the microkernel, (2) the object was previously built and is simply returned or (3) a warning is produced.

The procedure to construct the spaces in the construction modules is very similar in all of them.

1. Check the functional dependencies (this step depends on each particular constructor):

    (a) If the object does not fulfill the restrictions a `warning` XML-Kenzo object is returned.

(b) Otherwise, go to step 2.

2. Search in the `*object-list*` list if the object was built previously:

    (a) If the object was built previously, its identification number, in an `id` XML-Kenzo object, is returned.

    (b) Otherwise, go to step 3.

3. Construct an instance of the `mk-space-kenzo` class where:

    - the value of the `rede` slot is computed by the reduction degree module; see paragraph 2.2.3.2.1,
    - `idnm` is automatically generated,
    - `kidnm` is the value obtained from requesting the internal server the construction of the space, and
    - the XML-Kenzo object received (from the external server) as input is assigned to `orgn`.

4. Push the object in the `*object-list*` list of already created spaces.

5. Return the `idnm` of the object in an `id` XML-Kenzo object.

The above procedure is followed by most of the construction modules of the micro-kernel. However some of them do not construct instances of the `mk-space-kenzo` class but of some of its subclasses (see Paragraph 2.2.3.2.3) in order to use the additional information provided by the subclasses in the computation of homotopy groups (see Paragraph 2.2.3.2.3). However, the values of the slots which come from the `mk-space-kenzo` class of those instances are the same explained in the above procedure. To be more concrete, these construction modules are `Sphere`, `Delta`, `K-Z`, `K-Z-2`, `Cartesian Product`, `Suspension`, `Classifying Space` and `Loop Space`. The procedure implemented in these modules is different since they are the modules which can produce contractible spaces, spheres, Eilenberg MacLane spaces and loop spaces of spheres; that is to say, the spaces whose homotopy groups can be handled in the HES.

The `Sphere` module constructs instances of the `MK-SPACE-SPHERE` subclass, where the value of the `dim` slot is the dimension of the sphere (this dimension is obtained from the XML-Kenzo object).

The `Delta` module constructs instances of the `MK-SPACE-CONTRACTIBLE` subclass.

Both `K-Z` and `K-Z2` modules construct instances of the `MK-SPACE-K-G` subclass where the value of the `iter` slot is the number of iterations of the Eilenberg MacLane space (this number is obtained from the XML-Kenzo object) and the value of the `group` slot is 1 if is `K-Z` the working module and 2 if is `K-Z2`.

The `Cartesian Product` module constructs a `MK-SPACE-CONTRACTIBLE` instance if both components of the cartesian product are `MK-SPACE-CONTRACTIBLE` objects; otherwise, it constructs a `MK-SPACE-KENZO` instance with the general procedure.

The `Suspension` module constructs a `MK-SPACE-SPHERE` instance when the component of the suspension is a sphere, S^n, the value of the `dim` slot of the `MK-SPACE-SPHERE` instance constructed by the `Suspension` module is 1 plus n; otherwise, the `Suspension` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

The `Classifying Space` module constructs a `MK-SPACE-K-G` instance when the component of the Classifying Space is an Eilenberg MacLane space $K(G, n)$, the value of the `iter` slot of the `MK-SPACE-K-G` object constructed by the `Classifying Space` module is 1 plus n and the value of the `group` slot is the representation of G (1 in the case of $K(\mathbb{Z}, n)$ and 2 in the case of $K(\mathbb{Z}/2\mathbb{Z}, n)$); otherwise, if the component of the Classifying space is not an Eilenberg MacLane space, then the `Classifying Space` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

Finally, the `Loop Space` module constructs a `MK-SPACE-K-G` instance when the component of the Loop Space is an Eilenberg MacLane space, $K(G, n)$, the value of the `iter` slot of the `MK-SPACE-K-G` object constructed by the `Loop Space` module is n minus the number of iterations of the loop space and the value of the `group` slot in the `MK-SPACE-K-G` object constructed by the `Loop Space` module is is the representation of G. When the component of the Loop Space is a sphere S^n, the `Loop Space` module constructs a `MK-SPACE-LS-SPHERE` instance, where the value of the `iter` slot is the number of iterations of the Loop Space and the value of `dims` is n. Otherwise, the `Loop Space` module constructs a `MK-SPACE-KENZO` instance with the general procedure.

2.2.3.4 Computation Modules

Computation modules provide the functionality to perform computations through the microkernel. As we have seen in the XML-Kenzo specification, the `operation` element only has a child, either the `homology` or the `homotopy` element. Hence, the system only allows the computation of homology and homotopy groups. This situation is reflected in the computation modules, two Common Lisp modules that compute homology and homotopy groups respectively.

2.2.3.4.1 Homology module The *homology module* implements a procedure in Common Lisp that allows us to compute homology groups through the microkernel. The procedure not only calls the homology function of the Kenzo system but also wraps it to enhance the computation of homology groups in the framework, avoiding computations that would raise errors and optimizing them.

When the microkernel receives a computation request where the child of the `operation` element is `homology` the homology module is invoked. For instance, the request:

```
<operation>
  <homology>
    <loop-space>
        <sphere>4</sphere>
        <dim>3</dim>
    </loop-space>
    <dim>5</dim>
  <homology>
</operation>
```

activates the homology module. The procedure implemented in the homology module is as follows:

1. Extract the space whose homology wants to be computed.
2. Search in the `*object-list*` list if the space was built previously:
   - If the space was built previously, go to step 3.
   - Otherwise, the correspondent construction module builds the space and stores it in the `*object-list*` list.
     - If the construction module returns a warning then return the warning as result in a `warning` XML-Kenzo object.
     - Otherwise, go to step 4.
3. Search in the internal memory if the computation was performed previously; see Subsubsection 2.2.3.1.
   - If the computation was stored then the result by means of a `result` XML-Kenzo object is returned.
   - Otherwise, go to step 4.
4. Monitor the value of the reduction degree of the space, that is stored in the `rede` slot.
   - If the reduction degree is higher or equal than 0, go to step 5.
   - Otherwise, inform with a `warning` XML-Kenzo object that the system cannot compute the homology group since the reduction degree of the space is lower than 0.
5. Ask to the internal server the homology group of the space.
6. Store the result obtained from the internal server in the internal memory; see Subsubsection 2.2.3.1.
7. Return the result by means of a `result` XML-Kenzo object.

The result returned by the homology module when receives the above request is

```
<result>
    <component>3</component>
    <component>2</component>
    <component>1</component>
</result>
```

that must be read as $\mathbb{Z}/3\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}$ that is the fifth homology group of $\Omega^3 S^4$.

2.2.3.4.2 Homotopy module The *homotopy module* implements a procedure in Common Lisp that allows us to compute some homotopy groups through the microkernel in two different ways depending on the space. The homotopy module uses the HES (see Paragraph 2.2.3.2.3) if we want to compute homotopy groups of spheres, contractible spaces, Eilenberg MacLane spaces or loop spaces of spheres; otherwise, the homotopy module invokes the HAM (see Paragraph 2.2.3.2.2).

When the microkernel receives a computation request where the child of the `operation` element is `homotopy` the homotopy module is invoked. For instance, the request:

```
<operation>
  <homotopy>
    <sphere>3</sphere>
    <dim>6</dim>
  </homotopy>
</operation>
```

activates the homotopy module. The procedure implemented in the homotopy module is as follows.

1. Extract the space whose homotopy wants to be computed.
2. Search in the `*object-list*` list if the space was built previously.
    - If the object was built previously, go to step 3.
    - Otherwise, the correspondent construction module builds the space and stores it in the `*object-list*` list.
        - If the construction module returns a warning then the warning is returned as result in a `warning` XML-Kenzo object.
        - Otherwise, go to step 4.
3. Search in the internal memory if the computation was performed previously, see Subsubsection 2.2.3.1.

- If the computation was stored then the result using a `result` XML-Kenzo object is returned.
- Otherwise, go to step 4.

4. Monitor the value of the reduction degree of the space.
   - If the reduction degree is higher or equal than 1, go to step 5.
   - Otherwise, inform with a `warning` XML-Kenzo object that the system cannot compute the homotopy group since the reduction degree of the space is lower than 1.
5. If the space whose homotopy wants to be computed is an instance of the classes: `mk-space-sphere`, `mk-space-contractible`, `mk-space-k-g` or `mk-space-ls-sphere`:
   - Invoke the HES; see Paragraph 2.2.3.2.3.
   - Otherwise, invoke the HAM; see Paragraph 2.2.3.2.2
6. Store the obtained result in the internal memory; see Subsubsection 2.2.3.1.
7. Return the result using a `result` XML-Kenzo object if the result was obtained by the HAM or a `HES-result` XML-Kenzo object if the result was obtained by the HES.

The result returned by the homotopy module when receives the above request is

```
<result>
    <component>12</component>
</result>
```

that must be read as $\mathbb{Z}/12\mathbb{Z}$ that is the sixth homotopy group of S^3.

It is worth noting that the last restriction handled in the framework, that is the reduction degree restriction, is dealt with in the computation modules; namely, in step 4 of the procedures.

To sum up, computation modules (with the help of processing modules) not only use the functionality of Kenzo to obtain homology and homotopy groups, but also execute some test that enhance the original computation capabilities of Kenzo.

2.2.4 External Server

The *external server* is a Common Lisp *component* providing access to all *services* available in the microkernel interface through an XML-Kenzo interface. The external server receives XML-Kenzo requests, analyzes them, invokes the microkernel and sends the results back to the caller.

The analysis performed in the external server consists in validating the XML-Kenzo requests against the specification given in the XML-Kenzo language. This means that constraints imposed in the XML-Kenzo language (type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, and restriction of the dimension in computations) are checked here.

To validate these constraints an XML validator has been implemented. It takes as input an XML object, accesses to the XML-Kenzo schema definition and checks that the XML object is valid against the XML-Kenzo specification. It is worth noting that if we modify the XML-Kenzo schema definition, for instance adding a new kind of constructor, the external server evolves without modifying the source code.

As we have seen previously, other restrictions which were not included in the XML-Kenzo specification are checked at the microkernel level (namely, functional dependencies of the arguments of the constructors and reduction degree restrictions). Besides, as we are working with XML objects as data, they must be *well-formed XML*, but the task of verifying this aspect is responsibility of the *adapter*.

2.2.5 Adapter

The *adapter* is a Common Lisp module that provides access to the *services* available in the external server through an *OpenMath* [Con04] interface.

As we explained in Subsection 2.2.1, the main reason to define the fresh XML-Kenzo language instead of using the standard OpenMath language was the general purpose of this standard which makes its adaptation to our context a bit hard. However, the need of providing a suitable interface, uncoupled of the internal representation used in the framework, for different clients turns the tables in the most external part of the framework where we use OpenMath instead of using XML-Kenzo.

The OpenMath standard defines several mathematical objects of general purpose, such as linear algebra operators or arithmetic functions. However, specific objects, such as the mathematical structures used in Algebraic Topology, have not been declared; but, they can be defined following the guidelines given in [Dav00].

In particular, we have extended the OpenMath standard defining the objects specified in XML-Kenzo. To declare new objects in OpenMath we defined *Content Dictionaries*. A Content Dictionary is the declaration of a collection of objects, their names, descriptions, and rules. Namely, we have defined five Content Dictionaries, one for each one of the groups defined in XML-Kenzo (`CC`, `SS`, `SG`, `ASG` and `Computing`). Each one of these Content Dictionaries contains the declaration of the elements that belong to that group in XML-Kenzo; see Figures 2.5 and 2.10.

The definition of an object in a Content Dictionary includes its name, its description, examples of the object and sometimes its formal properties specified using OpenMath (in which case the equivalent commented property using natural language should also be given). For instance, the definition of the sphere element in OpenMath starts as follows:

```
<CDDefinition>
```

then the name of the object

```
<Name>sphere</Name>
```

the description

```
<Description>
  This symbol is a function with one argument, which should be a natural number
  n between 1 and 14. When applied to n it represents the sphere of dimension n.
</Description>
```

an example (namely S^3)

```
<Example>
  <OMOBJ>
    <OMA>
      <OMS cd="SS" name="sphere"/> <OMI>3</OMI>
    </OMA>
  </OMOBJ>
</Example>
```

the commented property in natural language

```
<CMP>The dimension of the sphere must be a natural number between 1 and 14 </CMP>
```

the formal property using the OpenMath standard

```
<FMP>
  <OMA>
    <OMS cd="logic1" name="and"/>
    <OMA>
      <OMS cd="set1" name="in"/> <OMV name="n"/> <OMS name="N" cd="setname1"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMI>1</OMI> <OMV name="n"/>
    </OMA>
    <OMA>
      <OMS cd="relation1" name="leq"/> <OMV name="n"/> <OMI>14</OMI>
    </OMA>
  </OMA>
</FMP>
```

are given and finally the definition is closed.

```
</CDDefinition>
```

In this way, all the objects represented in the XML-Kenzo language are specified in OpenMath. Moreover, another Content Dictionary called `Aux` to specify the objects belonging to the `results` XML-Kenzo group, such as the elements to return warnings or results, has been defined. The complete definition of the Content Dictionaries can be seen in [Her11].

At first glance, formal properties could seem a suitable way to represent the restrictions imposed in the XML-Kenzo language and this is true for some of them. However some restrictions (namely, type restrictions) can only be given here by means of the commented properties that are just properties in natural language, and this is not enough to our purpose.

Therefore, although OpenMath is not strong enough to represent all the mathematical knowledge that we need in our framework, we can use it in the most external part of our framework to represent our mathematical objects and exchange them with other systems, a task that suits perfectly to OpenMath.

Once we have defined the OpenMath objects, we need a software in charge of converting from XML-Kenzo representation to OpenMath representation and viceversa, this component is called *Phrasebook* and is programmed as a Common Lisp module of the adapter.

To sum up, the adapter receives OpenMath requests, checks that the OpenMath request is *well-formed XML* by means of a XML package of Common Lisp [Inc05], converts the OpenMath request into a well-formed XML-Kenzo request, invokes the external server with the XML-Kenzo request and sends the result received from the external server back to the caller using the OpenMath format.

2.3 Execution Flow

Once we have presented all the components of our framework, let us illustrate the execution flow of the system with a detailed scenario: the computation of the sixth homotopy group of the sphere of dimension 3, $\pi_6(S^3)$, in a fresh session of the Kenzo framework; that is, neither objects were constructed or computations were performed previously. The execution flow of this scenario is depicted in Figure 2.15 with a UML-like sequence diagram.

The OpenMath representation of $\pi_6(S^3)$ is the following one:

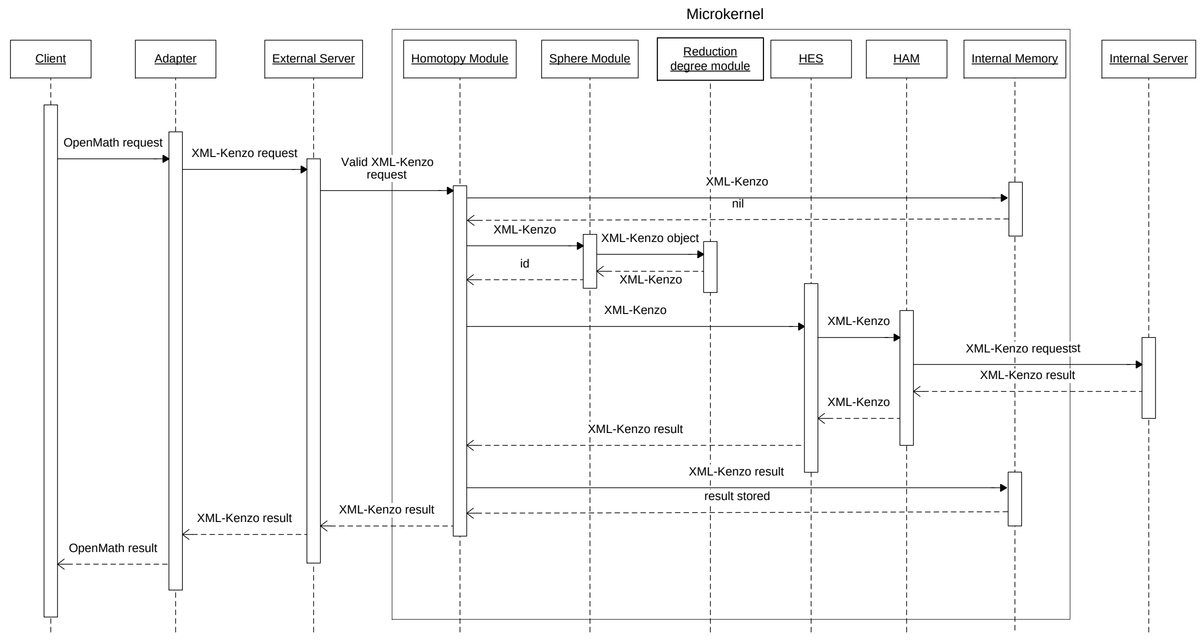


Figure 2.15: Execution flow of the Kenzo framework

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
            <OMS cd="SS" name="sphere"/>
            <OMI>3</OMI>
        </OMA>
        <OMI>6</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath data from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
    <homotopy>
        <sphere>3</sphere>
        <dim>6</dim>
    </homotopy>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification, in this case as the root element is `operation` it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homotopy` element has two children. ✓
3. The first child of the `homotopy` element belongs to one of the groups `CC`, `SS`, `SG` or `ASG`. ✓
4. The value of the `sphere` element is a natural number higher than 0 and lower than 15. ✓
5. The second child of the `homotopy` element is the `dim` element. ✓
6. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, a valid request is sent to the microkernel. In the microkernel the `homotopy` module is invoked. When the `homotopy` module is invoked, the procedure explained in Paragraph 2.2.3.4.2 is executed. First, the `homotopy` module search in `*object-list*` list of the internal memory if the sphere of dimension 3 was constructed previously, as this space was not constructed, the `sphere` module is invoked to construct it, this module in turn calls the reduction degree module to obtain the reduction degree of the sphere. Subsequently, once constructed the `mk-space-sphere`, as the reduction degree of the space S^3 is two, the `homotopy` module can call the HES. The HES tries to apply its rules; however, as all the rules which can be applied to `mk-space-sphere` instances depend on dynamic facts, it cannot apply any rule (since we are working with a fresh session, then dynamic facts have not been computed yet). Therefore, the HES invokes the HAM which performs some intermediary computations and, in this case as the intermediary tests succeed, calls the internal server to compute $\pi_6(S^3)$. The result returned by the internal server is:

```
<result>
    <component>12</component>
</result>
```

This result is stored in the internal memory, to avoid re-computations or to use it as dynamic fact, by the `homotopy` module and sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>12</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

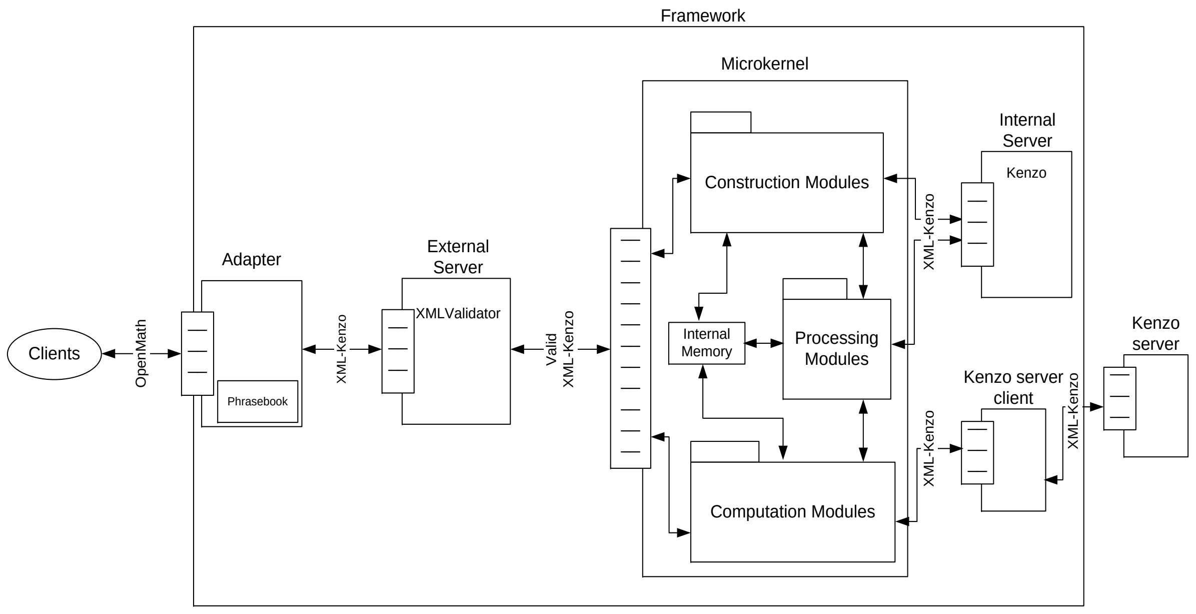


Figure 2.16: Kenzo framework with remote Kenzo computations

2.4 Integration of local and remote Kenzo computations

As we have said several times throughout this memoir, some Kenzo computations can be very time and space consuming (requiring, typically several days of CPU time on powerful dedicated computers). Then, we realized that we could use a dedicated server to perform those heavy computations instead of overloading the Kenzo framework user's computer. This section is an ongoing work, so just some ideas are provided in order to give a flavor of the way of integrating local and remote Kenzo computations.

As a first step to integrate local and remote Kenzo computations, we can consider the framework depicted in Figure 2.16 where the computations are not longer performed locally but in a remote server.

There are two new components in this framework. A powerful and external *Kenzo server* and a *Kenzo server client*.

The external Kenzo server is a dedicated sever which receives XML-Kenzo computation requests and returns XML-Kenzo results. The Kenzo server client is a program which is able to invoke the services which the Kenzo sever offers and receive the results returned by the Kenzo server. It is worth noting that the Kenzo server client can be seen as a new internal server of the Kenzo framework.

In this new organization we are keeping in our framework the internal sever since both construction and processing modules use it. On the contrary, the computation modules are not longer connected with the internal server to perform calculations but with the Kenzo server through the Kenzo server client.

Therefore, we have two different ways of performing computations of groups of spaces in the Kenzo framework. On the one hand, the original way of working, where all the computations are locally performed by means of the Kenzo internal server. On the other hand, the new approach presented in this section where all the computations are performed in an external Kenzo server.

The second mode has obvious drawbacks related to the reliability of Internet connections, to the overhead of management where several concurrent users are allowed, and so on. But the original way of working is not fully satisfactory since interesting Kenzo computations used to be very time and space consuming. Thus a mixed strategy should be convenient: the computation modules of the microkernel of the Kenzo framework should decide if a concrete calculation can be done in the local computer or it deserves to be sent to the Kenzo server. In this case, as it is not sensible to maintain open an Internet connection for several days waiting for the end of a computation, a subscription mechanism, allowing the Kenzo framework to disconnect and to be subscribed to the process of computation in the remote server, should be used.

The difficulties of how to mix local and remote computations have two sources: (1) the knowledge here is not based on well-known theorems since it is context-dependent (for instance, it depends on the computational power of a local computer), and so it should be based on heuristics; and (2) the technical problems to obtain an optimal performance are complicated, due, in particular, to the necessity of maintaining a shared state between two different computers.

With respect to the kind of heuristic knowledge to be managed into the intermediary level, there is some part of it that could be considered obvious: for instance, to ask for a homology group $H_n(X)$ where the degree n is big, should be considered harder than if n is small, and then one could wonder about a limit for n before sending the computation to the remote server. Nevertheless, this simplistic view is to be moderated by some expert knowledge: it is the case that in some kinds of spaces, difficulties decrease when the degree increases. The heuristics should consider each operation individually. For instance, it is true that in the computation of homology groups of iterated loop spaces, difficulties increase with the degree of iteration. Another measure of complexity is related to the number of times a computation needs to call the Eilenberg-Zilber algorithm (see [DRSS98]), where a double exponential complexity bound is reached. Further research is needed to exploit the expert knowledge in the area suitably, in order to devise a systematic heuristic approach to this problem.

In order to understand the nature of the problem of sharing the Kenzo state between two computers it is necessary to consider that there are two kinds of state in our context. Starting from the most simple one, the state of a session can be described by means of the spaces that have been constructed so far. Then, to encode (and recover) such a state, a session file containing a sequence of constructors is enough.

Moreover, there exists another kind of state. As we described in Subsection 1.2.1, a space in Kenzo consists of a number of methods describing its behavior (explaining, for instance, how to compute the faces of its elements). Due to the high complexity of the algorithms involved in Kenzo, a strategy of memoization has been implemented (see Section 1.2.5). As a consequence, the state of a space evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a homology group, depends on the concrete states of every space involved in the calculation (in the more explicit case, to re-calculate a homology group on a space could be negligible in time, even if in the first occasion this was very time consuming). This notion of state of a space is transmitted to the notion of state of a session. We could speak of two states of a session: the one sallow evoked before, that is essentially static and can be recovered by simply re-constructing the spaces; and the other deep state which is dynamic and depends on the computations performed on the spaces.

To analyze the consequences of this Kenzo organization, we should play with some scenarios. Imagine during a local session a very time consuming calculation appears; then we could simply send the sallow state of an object to the remote server, because even if some intermediary calculations have been stored in local memory, they can be re-computed in the remote server (finally, if they are cheap enough to be computed on the local computer, the price of re-computing them in the powerful remote server would be low). Once the calculation is remotely finished, there is no possibility of sending back the deep state of the remote session to the local computer because, usually, the memory used will exhaust the space in the local computer. Thus, it could seem that to transmit the sallow state would be enough. But, in this picture, we are losing the very reason why Kenzo uses the memoization (dynamic programming) style. Indeed, if after obtaining a difficult result (by means of the remote server) we resume the local session and ask for another related difficult calculation, then the remote server will initialize a new session from scratch, being obligated to re-calculate every previous difficult result, perhaps making the continuation of the session impossible. Therefore, in order to take advantages of all the possibilities Kenzo is offering now on powerful scientific servers, we are faced with some kind of state sharing among different computers (the local computers and the server), a problem known as difficult in the field of distributed object-oriented programming.

The same problem above presented appears in the organization of the Kenzo server. At this moment, we are working with the *Broker* architectural pattern, see [B$^+$96, B$^+$07], in order to find a natural organization of the Kenzo server. In a broker based server, there would be several Kenzo kernels and a component called *broker*. The broker component will be responsible for the distribution of requests arriving to the Kenzo server across the different Kenzo kernels, and the returning of results. One of the main advantages of using the *Broker* pattern is the fact that Kenzo kernels not only can work individually but also can collaborate to obtain results. However, to achieve the cooperation of different Kenzo kernels we find the problem of sharing the state among the different kernels presented in the previous paragraph.

Therefore, more work is necessary to find a plausible solution to integrate local and remote Kenzo computations.

2.5 Towards a distributed framework

In the previous section, an ongoing work devoted to increase the computational capabilities of the Kenzo framework through the distribution of computations across local and remote Kenzo kernels was presented. This section, also related to a work in progress, presents an alternative to the Kenzo framework which will take advantage of the collaborative work of different clients by means of a system located not in user's computer but in a shared server. The new framework will be called *server framework*.

The server framework has some common concerns with the Kenzo framework, since both frameworks provide a mediated access to the Kenzo system. There are, however, important differences. The Kenzo framework is installed in the local computer of its clients. On the contrary, the server framework is located in a remote server, and users of this system only need a light client able to communicate with the server framework; due to this fact, we have a collaborative work among different clients of the server framework, since a client can take advantage of the computations performed by other one. However, no reward comes without its corresponding price, and several problems, which do not exist in the Kenzo framework (for instance, the reliability of Internet connections or the management of concurrency problems), appear in the development of the server framework.

The most important challenges that we faced in the development of our server framework were:

1. *Functionality.* The system should provide access to the Kenzo capabilities for computing (homology and homotopy) groups.
2. *Interaction with different clients.* The server framework should be designed in such a way that it could support different kinds of clients (graphical user interfaces, web applications and so on).
3. *Reusability the Kenzo framework components.* As far as possible, we want to reuse the different components implemented in the Kenzo framework, in order to take advantage of the enhancements included in that framework.
4. *Error handling.* The server framework should forbid the user some manipulations raising errors. This question is easily handled due to the reuse of Kenzo framework components.
5. *Decoupling components.* In spite of partly reusing the modules implemented for the Kenzo framework, we are aware that the message passing communication style of the Kenzo framework is not suitable for a server architecture. Then, a kind of asynchronous communication among the components must be provided in order to decouple them.
6. *Storage of results.* The results must be stored in a persistent way to avoid recalculations.

7. *Multithreading.* The server framework should be designed in such a way that it could support the process of several requests at the same time.

8. *Management of concurrency problems.* The control of concurrency issues is one of the most important questions when a system can process several requests at the same time.

9. *Subscription mechanism.* It is not sensible to keep open an internet connection for several days waiting for the end of a computation; then, some reactive mechanism should be implemented, allowing the client to disconnect and to be subscribed in some way to the process of computation in the remote server.

The previous requirements led us, inspired by the work of [MlBR07], to choose both the *Linda model* [Gel85] and the *Shared Repository* architectural pattern [B$^+$96, B$^+$07] to reorganize the Kenzo framework components in a server framework. The communication between the server framework and its clients will be provided by means of *web services* [C$^+$07] based on OpenMath. In addition, a mail subscription mechanism will be implemented to avoid the problem of keeping open a connection during too much time.

The rest of this section is organized as follows. First of all, Subsection 2.5.1 is devoted to introduce the Linda model and the Shared Repository pattern. Our server framework based on the Linda model and the Shared Repository pattern is presented in Subsection 2.5.2.

2.5.1 The Linda model and the Shared Repository pattern

Our server architecture has been inspired by both the *Linda model* [Gel85] and the *Shared Repository* architectural pattern [B$^+$96, B$^+$07].

The *Linda model* is based on *generative communication*, a mechanism for asynchronous communication among processes based on a shared data structure. The asynchronous communication is performed by means of the insertion and extraction of data over the shared space.

The shared space is called *tuple space*, since it contains a set of *tuples* produced by the different processes. As soon as a tuple is inserted in the tuple space, it has an independent existence.

Tuples are represented by means of a list of items, the items are split by means of commas and closed between brackets. Tuples can contain both *actual* and *formal* items. An actual item contains a specific value, like 3 or *foo*. A formal item acts like a placeholder (a character "?" denotes a variable which has to be treated as formal).

Example 2.2. Some examples of tuples are:

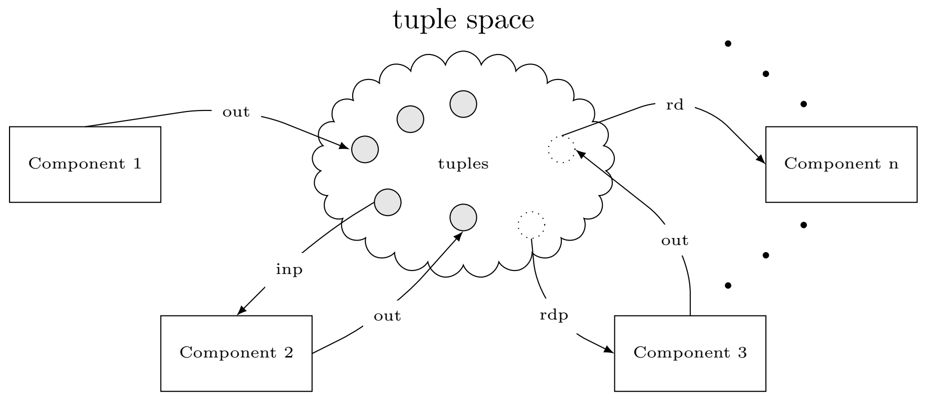


Figure 2.17: Framework based on both Linda model and Shared Repository pattern

$$(Paris, Toronto, ?)$$
$$(Peter, 15)$$
$$(car1, 10, 15.35)$$

In the Linda model, processes access the tuple space using five simple operations:

out: adds a tuple from the process to the tuple space.

in: deletes a tuple from the tuple space and returns it to the process. The process is blocked if the tuple is not available.

rd: returns a copy of one tuple of the tuple space. The process is blocked if the tuple is not available.

inp: a non blocking version of the *in* operation.

rdp: a non blocking version of the *rd* operation.

By means of the last four operations tuples can be retrieved from a tuple space. The arguments of these functions are tuple *templates*, possibly containing formal items (placeholders or wildcards).

The *Shared Repository* pattern is based on a very similar idea to the Linda model. It adds the nuance that a component has no knowledge of both what components have produced the data it uses, and what components will use its outputs.

Figure 2.17 shows a feasible framework which brings together both Linda model and the Shared Repository pattern.

An architecture based on the combination of the Linda model and the Shared Repository pattern solves two of the challenges that were stated at the beginning of this section. On the one hand, the communication among the modules of our framework will be performed through the tuple space; in this way, the different components of our system will

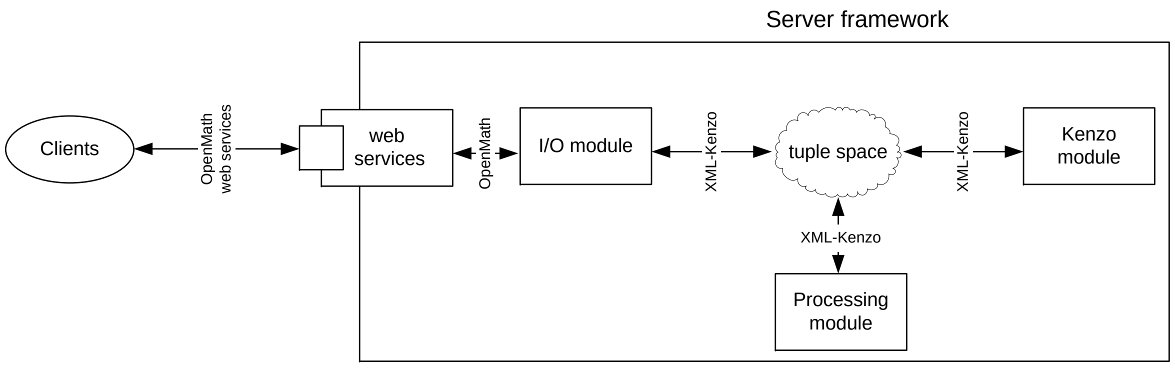


Figure 2.18: Schema of the server framework architecture

be decoupled. On the other hand, the question of storing results is easily solved using the Linda model due to the fact that all the computations can be stored as tuples in the tuple space avoiding re-computations.

2.5.2 A server framework based on the Linda model

A high level perspective of the server framework, based on both the Linda model and the Shared Repository pattern, as a whole is shown in Figure 2.18.

The rest of this subsection is devoted to present the components of this framework, as well as some important issues tackled in its development.

2.5.2.1 Components of the server framework

As we said at the beginning of this section, we decided to reuse, as much as possible, the components implemented in the Kenzo framework in our server framework. This decision was taken due to the fact that we did not want a powerful Kenzo server which can be used just by the Kenzo framework to perform long computations (as in the case presented in Section 2.4), but a server framework which can be employed for different clients taking advantage of the Kenzo enhancements implemented in the Kenzo framework and the collaborative work with other clients.

XML keeps on being the chosen technology to encode the data of our framework. Namely, XML-Kenzo is used for data interchange among the different components of the framework, and OpenMath is the language employed to communicate the server framework with the outside world.

The main innovation of the server framework with respect to the Kenzo framework is the tuple space of the Linda model. In order to use the Linda model in our context, a running implementation of it must be available. There are different Linda model implementations for Java [FHA99], C++ [Slu07], and even one for Common Lisp [Bra96].

Instead of using any of them, we have preferred to develop a fresh Allegro Common Lisp implementation adapted to our very concrete situation since the Common Lisp version presented in [Bra96] is out-of-date and the integration of either Java or C++ versions in our framework could be difficult.

We have developed our implementation of a tuple space holding the following properties: it must be shared (different modules can access it concurrently), persistent (once a tuple is stored in the tuple space, it stays in it until a module deletes it) and should comply with the *ACID* (Atomicity, Consistency, Isolation and Durability) rules.

Two possibilities were considered when designing our Linda model implementation. The first one consists of implementing the Linda model from scratch, without using any kind of external tool (for instance, the tuple space could be installed in computer memory and we could store the data as lists). This option would be very time consuming since it must solve well known problems in the domain of concurrent systems. Additionally, since our data are expressed as XML data, something like *XQuery* or *XPath* [E$^+$07] should be implemented in our case to search tuples inside our system, too.

These problems oriented us towards the second possibility: using already existing technology. Our previous remarks gave us the clue: XML databases [CRZ03] could be a good support in our case.

There are two kinds of XML databases: *native XML databases* and *XML enabled databases*. Native XML databases has XML documents as its fundamental unit of storage; on the contrary, XML enabled databases work with XML objects as data and use as internal model a tradicional (relational or object oriented) database. We chose XML enabled databases because we are going to work with XML-Kenzo objects. In XML enabled databases, each XML object is mapped to a datum of the internal (relational or object oriented) model of the database giving access to all the features and the performance that can be found in the corresponding database manager. These databases include two kind of processes: map XML data to objects or tables, and convert database elements into XML data.

The chosen database was AllegroCache [Aas05]. AllegroCache is an object oriented database of Allegro Common Lisp. One of the main advantages of working with AllegroCache is that the data are always stored persistently but one can work directly with objects as if they were in standard memory. Besides it supports a full transactional model with long and short transactions, meets the classic ACID requirements for a database and maintains referencial integrity of complex data objects. Persistent classes located by AllegroCache are just usual CLOS classes, where the metaclass `persistent-class` is declared.

In order to complete our Linda model implementation, we still have to deal with both tuples and the operations to interact with the tuple space. Since we have decided that tuples rely on XML-Kenzo data but the repository (to be understood as a tuple space) is based on AllegroCache, it is necessary to convert from XML-Kenzo objects to instances of persistent classes stored in the database and viceversa. To this aim, we have

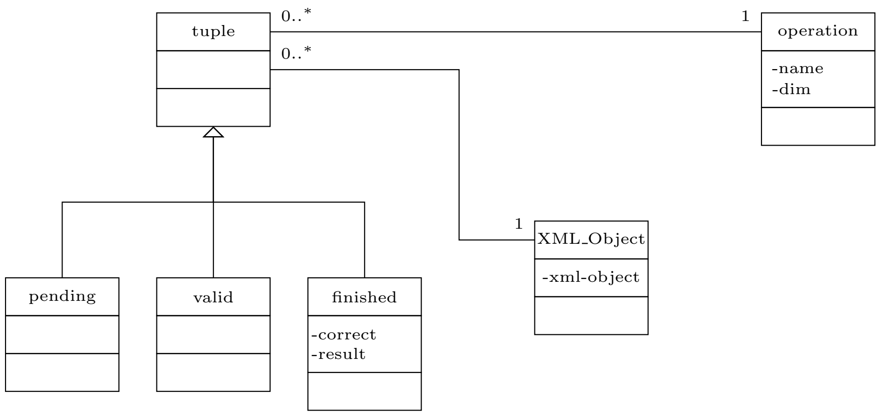


Figure 2.19: Class diagram

devised a class system represented with UML notation in Figure 2.19.

As we have said previously, the server framework is devoted to perform computations; then, the requests which arrive to this framework are computation requests. Each computation request is made up of both a topological space and an operation over it, so two classes, `XML-Object` (with just one slot whose value is the XML-Kenzo representation of the space) and `operation` (with two slots, `name` and `dim`, which provide respectively the name of the operation, homology or homotopy, and the dimension) have been defined to model the two components of a computation request. The `tuple` class binds these two components of a request.

Moreover, in our server framework a tuple can have three different states: (1) the tuple is *pending* of being validated, (2) the tuple has been *validated*, and (3) the system has obtained the result of the computation request associated with the tuple, and, then, the tuple has *finished* its work. Then, a tuple is implemented as an instance of the `pending` subclass in the first case, as a `valid` instance in the second one; and as a `finished` instance in the last one. It is worth noting that instances of the `finished` class have two additional slots: (1) the `correct` slot indicates if some problem was found when processing the tuple by means of a boolean value, and (2) the `result` slot stores a `warning` XML-Kenzo object if some problem was found when processing the tuple or a `result` XML-Kenzo object otherwise.

Finally, as our tuple space is an AllegroCache database, the Linda model operations are implemented in a natural way from the `select`, `insert` and `delete-instance` functions of AllegroCache. For instance, the *inp* operation of the Linda model can be programmed by combining a `select` operation and a `delete-instance` one. Thus, we have built five methods with the following signatures:

$$\begin{array}{ll}
\texttt{writetuple:} & \texttt{tuple} \rightarrow \mathit{Boolean}, \\
\texttt{taketuple:} & \texttt{tuple} \rightarrow \texttt{tuple}, \\
\texttt{readtuple:} & \texttt{tuple} \rightarrow \texttt{tuple}, \\
\texttt{taketuplep:} & \texttt{tuple} \rightarrow \texttt{tuple} \vee \texttt{nil} \text{ and} \\
\texttt{readtuplep:} & \texttt{tuple} \rightarrow \texttt{tuple} \vee \texttt{nil},
\end{array}$$

These are, respectively, our versions for *out*, *in*, *rd*, *inp* and *rdp*.

This finishes the description of our Linda model implementation. The rest of the server framework components are three Common Lisp modules based on the constituents of the Kenzo framework and the interface which allows the communication with the outside. All the modules are organized in two layers: the business logic layer and the persistence layer. The persistence layer is in charge of communicating the module with the tuple space through the operations (`writetuple`, `taketuple` and so on) of our Linda model implementation. The business logic layer always works with XML-Kenzo objects produced by the persistence layer from the tuples of the tuple space.

Then, our modules are as much independent as possible from the chosen technology used to implement the Linda model, since the business logic layers always work with the same representation of objects; then, if we change our implementation of the Linda model, we only need to change the persistence layers; remaining untouched the business logic ones.

Let us explain the three modules.

2.5.2.1.1 The I/O module The *I/O module* business logic layer consists of the Kenzo framework adapter, presented in Subsection 2.2.5. The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `pending` tuples and the reading of `finished` tuples. When, a new OpenMath computation request arrives to the server framework this module is activated and proceeds as follows.

1. Check that the OpenMath request is well formed:
    (a) If the request is not well formed a `warning` OpenMath object is returned.
    (b) Otherwise, go to step 2.
2. Transform the OpenMath object into an XML-Kenzo object.
3. The persistence layer searches among the `finished` tuples of the tuple space if the result for that request was previously computed:
    (a) If the result was stored, the `finished` tuple which stores the result is read by the persistence layer and an XML-Kenzo object result is returned to the business logic layer. Go to step 6.
    (b) Otherwise, go to step 4.

4. The persistence layer constructs a `pending` tuple from the XML-Kenzo request and writes it in the tuple space.

5. When the server framework obtains a result for the computation request, the `finished` tuple which stores the result for that request is read by the persistence layer and an XML-Kenzo object result is returned to the business logic layer.

6. Transform the XML-Kenzo result into an OpenMath result.

7. Return the OpenMath result to the client.

2.5.2.1.2 The Processing module The *Processing module* business logic layer merges the external server, presented in Subsection 2.2.4, and part of the functionality of the microkernel (namely, the knowledge included in that component to manage errors), presented in Subsection 2.2.3. This module is in charge of validating the correctness of requests; namely, all the knowledge included in both the external server and the microkernel to handle errors is gathered in this module. To be more concrete all the restrictions managed in the Kenzo framework (that is to say; type restrictions, independent argument restrictions of the space constructors, implementation restrictions of the space constructors, functional dependencies of the arguments of the constructors, reduction degree restrictions and restriction of the dimension in computations) are handled in the processing module. Let us note that in spite of only working with computation requests, the server framework also has to check the constraints related to the construction of spaces in order to avoid errors.

The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `valid` and `finished` tuples and the reading of `pending` tuples. When, a `pending` tuple is written in the tuple space the processing module is activated and proceeds as follows.

1. The persistence layer takes the `pending` tuple.

2. The persistence layer transforms the `pending` tuple into an XML-Kenzo object which is sent to the business logic layer.

3. The business logic layer checks all the constraints about the XML-Kenzo object:

   (a) the business logic layer indicates to the persistence layer to write a `valid` tuple in the tuple space if all the constraints are fulfilled.

   (b) Otherwise, indicates to the persistence layer to write a `finished` tuple in the tuple space with `nil` as value of the `correct` slot and the warning produced as value of the `result` slot.

4. The persistent layer writes the indicated tuple in the tuple space.

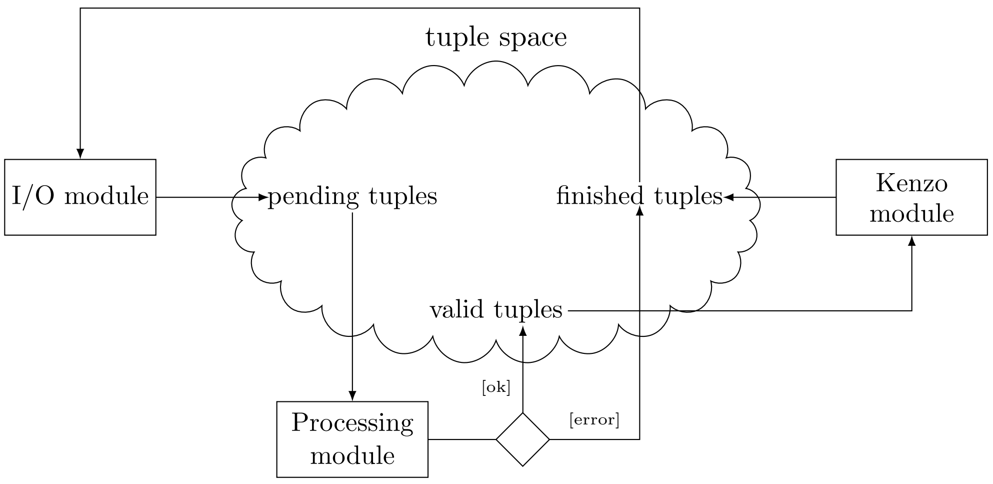


Figure 2.20: Relations among modules and the tuple space

2.5.2.1.3 The Kenzo module The Kenzo module business logic layer is the internal server of the Kenzo framework, presented in Subsection 2.2.2. The persistence layer of this module allows it to interact with the tuple space by means of the insertion of `finished` tuples and the reading of `valid` tuples. When, a `valid` tuple is written in the tuple space the Kenzo module is activated and proceeds as follows.

1. The persistence layer takes the `valid` tuple.
2. The persistence layer transforms the `valid` tuple into an XML-Kenzo object which is sent to the business logic layer.
3. The business logic computes the value of the request and sends an XML-Kenzo result to the persistence layer.
4. The persistence layer constructs a `finished` tuple from the XML-Kenzo result and writes it in the tuple space.

At this moment the server framework has only a Kenzo kernel; however, as we said in Section 2.4, we should use several Kenzo kernels in order to deal with different Kenzo computations. To this aim, we could use the *Broker* pattern [B$^+$96, B$^+$07] which will allow us to distribute different parts of a computation among different Kenzo kernels. Nevertheless, this task remains as further work.

2.5.2.1.4 Relations among the modules and the tuple space The relations among the modules and the tuple space are shown in Figure 2.20.

It is worth noting that `pending` and `valid` tuples are respectively removed from the tuple space by the processing module and the Kenzo module; on the contrary, `finished` tuples remain in the tuple space to avoid re-computations, achieving the challenge of storing results.

2.5.2.1.5 Web Services The previous paragraphs have been devoted to present the different components of the framework and their relations with the tuple space. Let us focus now on other innovation with respect to the Kenzo framework: the use of web services to communicate the server framework with different clients.

In order to offer transparent access to our server computing infrastructure, three *web services* have been developed (briefly, a web service is a program, based on XML, which provides access to an application over the network). These three web services allow the use of the server framework only knowing the OpenMath format of the computation requests (no knowledge about Lisp or Kenzo is needed). So, a developer could build a client (in platforms with support for web services, such as Java, .Net, Lisp and so on) for our web services only knowing the OpenMath syntax, which is described in the OpenMath Content Dictionaries, and, of course, the web services technology of the concrete programming language. We have implemented these web services using the SOAP 1.1 API for Allegro Common Lisp [Inca]. This API allows us to open the access to our server on the internet via the *SOAP* protocol [G$^+$] (a web services protocol). Let us explain the differences among the three web services.

The first web service makes a request to the server framework and waits until the result is returned. This web service can be useful in the implementation of a light client where all the computations are done in the server. However, if the computation asked to the server framework takes too much time, the connection must be kept open all that time.

On the contrary, in the second web service, the connection is not maintained, and therefore it needs not only the OpenMath computation request but also an e-mail address where the result will be returned. In this case, to send mails, we have chosen to use Java joins with the *JavaMail* API [Mic99]. The JavaMail API is a set of abstract APIs modeling a mail system and providing the required technology. In order to do this, we need to communicate our framework (developed in Lisp) with a Java application. There exists different solutions to this problem: we could use the middleware CORBA [Gro], or use the *jLinker* package [Incd] provided by Allegro Common Lisp; nevertheless, these options are too general for our concrete aim. Then, we have used a solution based on one of the *jLinker* ideas. *jLinker* requires two open socket connections between Lisp and Java, one for calls from Lisp to Java, and another one for the inverse communication. In our case we only need one socket for communicating Lisp with Java, because our Lisp system does not require information from our Java system. In Java we have a passive socket waiting for the creation of a connection by means of an active socket (it is the same idea presented in Subsubsection 2.5.2.2 to activate the server framework components). In the Lisp system, whenever a new result is created, the web service creates an active socket which connects with the passive socket of Java and sends both the e-mail address of the client and the result of the computation requested. Finally, the Java program sends an e-mail to the client with the result of the computation asked for the client. However, this web service is not fully satisfactory, since even if the computation asked to the server framework only taking a few microseconds, the client always has to check his e-mail to obtain the result.

As we have said, neither of the two previous web services is fully satisfactory, since they implement two extreme interactions: the first one always keeps open the connection until the result is returned; on the contrary, the second one never does it. Then, to solve this problem a third web service has been implemented mixing the good properties of the others. This web service receives an OpenMath computation request and an e-mail address. When the result is obtained the system checks if the connection with the user is still open, in that case the web service directly returns the result to the user; otherwise it sends the result to the e-mail address. Clients of this web service must decide how much time the connection remains open, an issue which depends on each concrete client.

These web services solve two of the challenges stated at the beginning of this section. On the one hand, web services and OpenMath are general enough to be used for different clients. On the other hand, a subscription mechanism is implemented, then, instead of waiting for the end of a computation, this reactive mechanisms sends an e-mail with the result to the user if the user connection is not longer available.

2.5.2.2 Communication among modules

Up to now, how the different modules are communicated has not been explained. This is one of the most important issues, and has been tackled by means of a reactive mechanism, to be more concrete by means of a *publish-subscribe* machinery.

This mechanism is based on the existence of both *subjects*, which can be modified, and *observers* subscribed to any possible subjects modification. This design has been implemented following the *Publish-Subscriber* pattern [B$^+$96, B$^+$07]. This design pattern, also called *Observer*, helps to keep synchronized the state of cooperating components, providing a mechanism for asynchronous communication.

In this pattern, each subject has associated a component which takes the role of publisher. All the components which depend on changes in the subject are its subscribers (or observers). The publisher component maintains a registry of currently-subscribed components. Moreover, subscribers can be interested only in a kind of changes of the subject, so, this information is also kept by the publisher. Then, whenever the state of a subject changes, its publisher sends a notification to all the subscribers interested in that change.

In our context, we only have one subject that is our tuple space (the AllegroCache database) which has associated a publisher component (a bunch of Common Lisp functions), and the subscribers are the three modules wrapped with a notification mechanism (several Common Lisp functions) which allow them to receive the notifications from the publisher.

To store the subscriptions, an AllegroCache database, called from now on *subscription database*, is associated with the publisher of the tuple space. This database contains instances of the `subscription` class which is defined as follows:

```
(defclass subscription ()
  ((host :type string  :initarg :host :accessor host )
   (port :type integer :initarg :port :accessor port )
   (type-tuples :type symbol :initarg :type :accessor type :index any))
  (:metaclass persistent-class))
```

This class has three slots:

- `host`, a string being the *host* of a subscriber.
- `port`, an integer being the port of a subscriber.
- `type-tuples`, a symbol (`'pending`, `'valid` or `'finished`) indicating the kind of tuples which are interesting for the subscriber. In addition this slot is an *index slot*. Index slots are the way of filtering and ordering the results of a search in AllegroCache.

Moreover we have defined this class as persistent by means of the metaclass `persistent-class`. This implies that every instance of this class will be permanently stored (until a module explicitly deletes it) in the database.

The subscription database always contains, at least, three instances: one per each module of the server framework. To be more concrete, the instance associated with the I/O module includes its host, port and the interesting tuples for this module, that are `finished` tuples:

```
(make-instance 'subscription :host "localhost" :port 8001 :type-tuples 'finished)
```

the instance associated with the processing module includes its host, port and the interesting tuples for this module, that are `pending` tuples:

```
(make-instance 'subscription :host "localhost" :port 8002 :type-tuples 'pending)
```

and, finally, the instance associated with the Kenzo module includes its host, port and the interesting tuples for this module, that are `valid` tuples:

```
(make-instance 'subscription :host "localhost" :port 8003 :type-tuples 'valid)
```

It is worth noting that in this case the host of all the modules is the same. That is to say, currently, all the modules are located in the same computer. However, we could install each module in a different computer, and devote a dedicated computer for each one of the modules.

In the future, if we want to include a new module as subscriber of the tuple space, we only need to define an instance with its host, port and the interesting tuples. In this way, whenever an interesting tuple for the module is written in the tuple space, a notification is sent to the module by the publisher.

Let us explain now the notification mechanism which is based on *sockets* technology and is implemented with the Allegro Common Lisp socket package [Incb]. This package provides all the necessary tools to communicate our modules and the publisher by means of sockets technology.

Each one of the modules of our server framework has a *passive socket* in the port specified in its subscription instance. This passive socket is waiting its activation by means of the following function:

```
(defun observer-<module> ()
  (loop
    (let ((sock (make-socket :connect :passive :local-port <port>)))
      (wait-for-input-available sock)
      (close sock) (validate))))
```

where `<module>` and `<port>` are replaced with the name of the module and the port specified in the subscription database for that module. The rest of the function must be read as follows. We have a loop instruction that repeats always the same process: (1) creates a passive socket, (2) waits until the passive socket is activated, (3) closes the socket, and, finally, (4) does the job associated with the module.

Let us stress in the `observer-<module>` definition the use of the `wait-for-input-available` function. This function waits the connection of an *active socket* with the passive socket `sock`. Therefore, the well-known *busy-waiting* concurrency problem is solved (a good description of the concurrency issues presented in this memoir can be consulted in [Sch97]). *Busy-waiting* happens when a process repeatedly checks if a condition is true (for instance, the availability of a shared resource), the problem is that the processor spends all its time waiting for some condition. A sensible solution to this problem consists of using *semaphores* which is the approach followed with the `wait-for-input-available` function. Note that this function acts as a *binary semaphore*, to be precise as a `P` operation (the `P` operation sleeps the process until the resource controlled by the semaphore becomes available). Then, the module is not repeatedly checking if a new tuple has been written in the tuple space, but it is slept until someone activates it (this activation is produced when an interesting tuple for the module is written in the tuple space).

After the description of the implementation of the notification mechanism from the modules side, let us present now the publisher side. In the publisher we have the following function definition.

```
(defun notify (type-tuple)
  (open-network-database "localhost" 8010)
  (let ((subscrites
         (retrieve-from-index 'subscription 'type-tuple type-tuple :all t)))
    (dolist (temp subscrites)
      (let ((socket (make-socket :remote-host (host temp) :remote-port (port temp))))
        (close socket))))
  (close-database))
```

This function is called when a new tuple is written in the tuple space with the type of the written tuple, `type-tuple`, as argument. The workflow followed by this function when it is activated is as follows: (1) open the subscription database, (2) search all the modules which are subscribed to the writing of a tuple of `type-tuple` type in the tuple space, (3) creates an active socket for each one of the subscribers, and, finally, (4) close the database. It is worth noting that this function is acting as the `V` operation in a binary semaphore (the `V` operation is the inverse of the `P` operation: it makes a resource available). In this way, all the subscribers to the tuples of `type-tuple` type are activated.

Therefore, the tuple space and the modules are communicated achieving the challenge of an asynchronous communication among the modules of the framework.

2.5.2.3 An example of complete computation

Once we have described our server architecture, the communication among the modules and the communication between the server framework and its clients by means of web services, let us illustrate the execution flow of the server framework with a detailed scenario: the computation of the sixth homotopy group of the sphere of dimension 3, $\pi_6(S^3)$, in a *fresh* session of the server framework; that is, computations were not previously performed. The execution flow of this scenario is depicted in Figure 2.21 with a UML-like sequence diagram.

The server has received the OpenMath request by means of anyone of the three web services previously presented. The process of the request is the same for all of them, the only difference is the way of returning the result. The OpenMath representation of $\pi_6(S^3)$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
            <OMS cd="SS" name="sphere"/> <OMI>3</OMI>
        </OMA>
        <OMI>6</OMI>
    </OMA>
</OMOBJ>
```

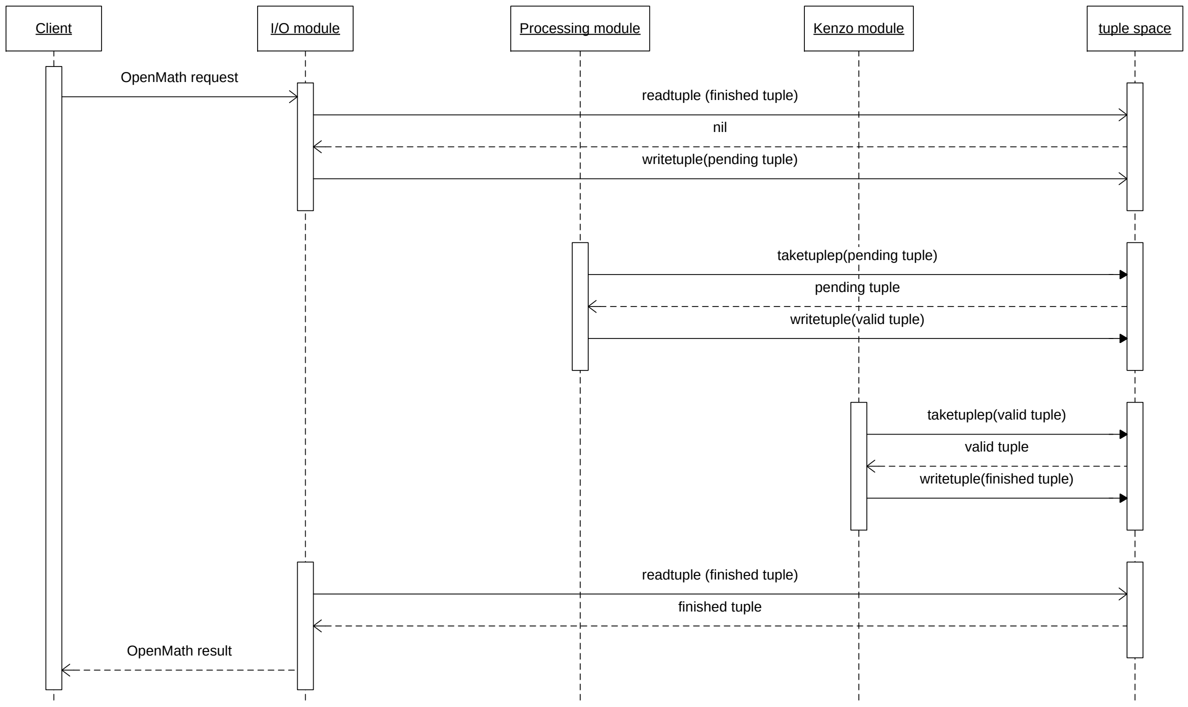


Figure 2.21: Execution flow of the server framework

The I/O module receives the previous OpenMath data from a client through one of the web services. This module checks that the OpenMath instruction is well-formed and converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
  <homotopy>
    <sphere>3</sphere>
    <dim>6</dim>
  </homotopy>
</operation>
```

Subsequently, the I/O module, which has access to the `finished` tuples, asks if a `finished` tuple with the result of the request exists in the tuple space. To this aim, a `finished` template object is created from the XML-Kenzo datum in order to query the tuple space as follows (the wildcard `'?` allows us to define a search template on tuples):

```
> (readtuplep
   (make-instance 'finished
     :operation (make-instance 'operation :name "homotopy" :dim 6)
     :XML_Object (make-instance 'XML_Object :xml-object "<sphere>3</sphere>")
     :correct '? :result '?)) ✠
NIL
```

In this case, `NIL` is returned, the result is not in the database (as we have said, we are working in a fresh session, so, no computation has been performed previously). Then the I/O module writes a `pending` tuple (corresponding to the request) in the tuple space (from now on, we will not include the attribute values of the instances if they are not different from the previous ones):

```
> (writetuple (make-instance 'pending :operation (...) :XML_Object (...))) ✠
T
```

Due to the existence of a new pending tuple the processing module is activated, and it asks for a pending tuple:

```
> (taketuplep (make-instance 'pending :operation '? :XML_Object '?)) ✠
#<PENDING @ #x2143918a>
```

With `(make-instance 'pending :operation '? :XML_Object '?)`, the processing module asks for a general pending tuple. The `taketuplep` method deletes the element of the database and creates an XML-Kenzo object used by the processing module to validate the request. In this case, the request is considered sensible so the processing module writes a valid tuple in the tuple space:

```
> (writetuple (make-instance 'valid :operation (...) :XML_Object (...))) ✠
T
```

Afterwards, the Kenzo module is activated due to the writing of a new valid tuple in the tuple space. Then it asks for the new tuple:

```
> (taketuplep (make-instance 'valid :operation '? :XML_Object '?)) ✠
#<VALID @ #x214be502>
```

and computes the result of the request writing the result as a finished tuple:

```
> (writetuple
    (make-instance 'finished :operation (...) :XML_Object (...)
                             :correct t :result "<component>12</component>")) ✠
T
```

Then the I/O module is activated and asks again for the result of the request.

```
> (readtuplep
    (make-instance 'finished :operation (...) :XML_Object (...)
                             :correct '? :result '?)) ✠
#<FINISHED-PERSISTENT oid: 5022 @ #x214cff02>
```

In this case, the result is not deleted from the tuple space because a `readtuplep` operation is used. Then, the result can be used again, without recomputing it.

Eventually, the I/O module converts the result into its OpenMath representation

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>12</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

2.5.2.4 Concurrency issues

In the previous subsubsection, we have presented an execution scenario where there is only one request in the system. In general, several requests coexist at the same time in the server framework; then, some *concurrency* problems can appear and must be dealt with. Let us remember that the management of concurrency problems was one of the challenges that we faced in the development of the server framework.

It is worth noting that most of the typical concurrency problems are solved thanks to the organization of our framework, based on the Linda model. As the server is based on this model, if a process wants to modify a tuple, it must extract it, modify it and then insert it again in the tuple space (remark that the extraction and the insertion operations are atomic). Therefore different processes keep independent among themselves, avoiding the occurrence of several problems related to concurrency.

Concurrency appears in two different ways in our server architecture. On the one hand, different modules can work at the same time; that is to say, while the Kenzo module is computing a homology group, the processing module can validate the correctness of a request and the I/O module can receive new requests and write them in the tuple space. On the other hand, several processes of the same module can work at the same time; that is to say, the Kenzo module can perform several computations simultaneously.

In the former case (different modules working at the same time), the most important concurrency features which must be considered in our server framework are:

- *Absence of deadlocks.* A deadlock happens when two *processes* are waiting for the other to finish, and thus neither ever does. In our server framework this situation never happens since the execution of each one of the modules is independent from the other ones.

- *Mutual exclusion.* Mutual exclusion occurs when some algorithms avoid the simultaneous modification of a shared resource. In our server, the shared resource is the tuple space; however, each server framework component only access to a kind of tuples. Therefore, in spite of concurrently accessing to the tuple space, the server framework components access to different kinds of tuples; then, inconsistencies are avoided and the good property of mutual exclusion is reached.
- *Absence of starvation.* Starvation describes a situation where a process is unable to access to shared resources and is unable to make progress. This situation is not feasible in our server framework, since the modules are only activated when new resources are included in the tuple space. Moreover, the modules access to the tuple space by means of non blocking operations; then, if a module asks data to the tuple space but there is no datum, then the module backs to sleep. Therefore, our server framework components never starve.

Let us focus now on the latter case (several processes of the same module working at the same time). Every time that a module is activated a new *thread* (in charge of executing the instruction of the correspondent module) is built from the main process. In order to do this, a multiprocessing package of Common Lisp, which provides all the server framework modules with the main tools for working in a concurrent way (management of *threads, locks, queues* and son on), has been used. In this situation, the most important concurrency features which must be considered in our server framework are:

- *Absence of deadlocks.* Each thread of a module is independent from the rest of threads of the module. Then, none waits for the others to finish and, then, deadlocks never happen.
- *Absence of starvation.* A module launches a thread only when the module has received some data to process and this thread is killed when if finishes its task. Therefore, starvation never happens since the thread does not access to the shared resource to receive the data, because they are created with the data that must process.
- *Absence of race condition problems.* Race conditions arise in software systems when separate threads of execution depend on some shared state. It is worth noting that if two processes read (`readtuplep`) the same tuple, two different tuples will be created when each process writes its result, then, the *race condition* problem is avoided.

This short analysis shows that some typical inconsistencies produced by concurrency problems are avoided in our server framework.

Chapter 3

Extensibility and Usability of the Kenzo framework

The previous chapter was mainly devoted to present a framework which provides a mediated access to the Kenzo system, constraining its functionality, but providing guidance to the user in his navigation on the system. However, we cannot consider that the Kenzo framework is enough to fully cover our aims.

On the one hand, we want to be able to increase the capabilities of the system by means of new Kenzo functionalities or the connection with other systems such as Computer Algebra systems or Theorem Prover tools. On the other hand, the XML interface (to be more concrete, the OpenMath interface) is not desirable for a human user; then, a more suitable way of interacting with the Kenzo framework must be provided.

To cope with the extensibility challenge, we have deployed the Kenzo framework as a client of a *plug-in* framework that we have developed. This allows us to add new functionality to the Kenzo framework without accessing to the original source code. Moreover, as a client of both Kenzo and plug-in frameworks, an extensible friendly front-end has been developed. Combining the frameworks and the front-end we are able to improve the usability and the accessibility of the Kenzo system, increasing the number of users who can take profit of the Kenzo computation capabilities. The whole system, that is to say, the merger of the two frameworks and the front-end, is called *fKenzo*, an acronym for f̲riendly K̲e̲n̲z̲o̲ [Her09].

The rest of this chapter is organized as follows. The *plug-in* framework, which the Kenzo framework is a client of, is explained in Section 3.1. The main client of the Kenzo framework, an extensible user interface, is introduced in Section 3.2.

3.1 A plug-in framework

Architectures providing plug-in support are used for building software systems which are extensible and customizable to the particular needs of an individual user. Several interesting plug-in approaches exist. One of the first plug-in platforms was Emacs ("Editor MACroS") [Sta81], whose extensions are written in *elisp* (a Lisp dialect) and can be added at runtime. The Eclipse platform [Ecl03] is certainly the most prominent representative of those plug-in platforms and has driven the idea to its extreme: "Everything is a plug-in". Plug-ins for Eclipse are written in Java. Other examples are OSGi [OSG03], a Java-based technology for developing components, Mozilla [Moz], a web browser with a great amount of extensions, or Gimp [Pec08], a famous image processing software which allows one the addition of new functionality by means of plug-ins.

Kenzo is also a plug-in system: to add new functionality to the Kenzo system, a file with the new functions must be included; remaining untouched the Kenzo core. In general, all the systems implemented in Common Lisp are extensible. Therefore, to extend the functionality of a Common Lisp program, we, usually, only need to load new functions by means of Common Lisp files.

One of the most important challenges that we faced in the development of the Kenzo framework was the management of its extensibility. On the one hand, we wanted that the Kenzo framework could evolve at the same time as Kenzo. On the other hand, the framework should be enough flexible to integrate new tools, such as other Computer Algebra systems and Theorem Provers tools. In addition, we wanted that the original source code of the Kenzo framework remained untouched when the system was extended. Therefore, we decided to include a *plug-in* support in our framework by means of a *plug-in* framework developed by us.

The rest of this section is organized as follows. Subsection 3.1.1 is devoted to present the architecture of the plug-in framework. Explanations about the way of adding new functionality to the Kenzo framework through the plug-in framework are provided in Subsection 3.1.2.

3.1.1 Plug-in framework architecture

Extensibility is very important because an application is never really finished. There are always improvements to include and new features to implement.

With the aim of being able to extend different systems we have developed a plug-in framework. This framework may have systems of very different nature as clients. Therefore, the way of extending a client can be different to the way of extending the rest of them. This issue has been taken into account in the design of the plug-in framework.

The implementation of the plug-in framework has been based on the *plug-in* pattern presented in [MMS03]. This pattern distinguishes two main components: the *plug-ins*

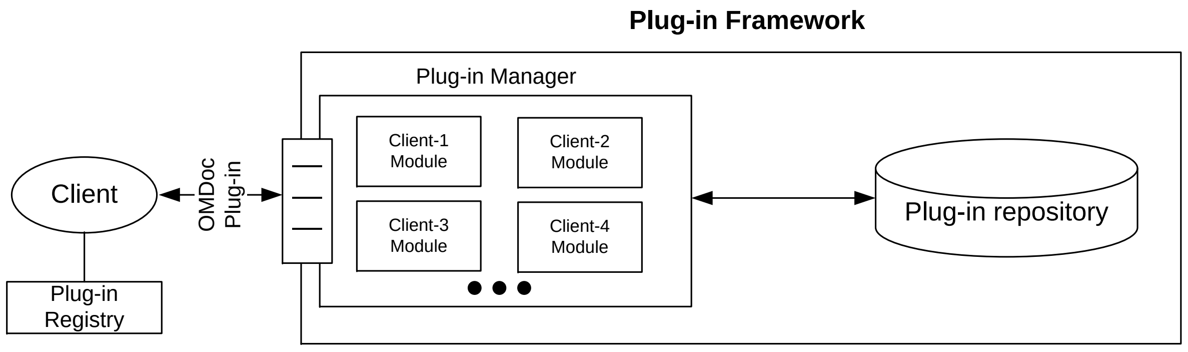


Figure 3.1: Plug-in framework

and the *plug-in manager*. Moreover, we have included two improvements of our own, the *plug-in repository* and the *plug-in registries* (each client of the *plug-in framework* will have associated a *plug-in registry*). A high level perspective of the plug-in framework is depicted in Figure 3.1. Let us explain each one of the constituents of our plug-in framework.

A *plug-in* in our system is an OMDoc document which consists of different resources used to extend a concrete client. Since the clients of the plug-in framework can be very different, the nature of the resources included in them are different, too. However, as it is desirable that the format of all the *plug-ins* (not the format of the resources which obviously depends on each client) was common for the different clients, we have chosen *OMDoc* [Koh06] as the format to encode our *plug-ins*. OMDoc is an open markup language for mathematical documents, and the knowledge encapsulate in them. The OMDoc plug-ins are documents that wrap the necessary resources to extend a concrete client of the plug-in framework.

Essentially, an OMDoc plug-in is an XML document, that stores the metadata about the plug-in (authorship, title, and so on) and wraps the resources (by means of references to other files that, of course, depend on the concrete client) which extend a client of the plug-in framework.

For instance, let us suppose that we have a client, called `client-1`, of our plug-in framework; and we want to extend that client by means of the resources stored in a file called `client-1-resources`. Then, the OMDoc plug-in, called `new-module-client-1`, will contain, in addition to the metadata about the *plug-in*, the following XML fragment (all the OMDoc plug-ins follow the same pattern).

```
<code id="new-module-client-1">
   <data format="client-1"> client-1-resources </data>
</code>
```

The above XML code must be read as follows. The `id` argument of the `code` tag indicates the name of the new module. Inside this tag the different resources to extend `client-1` with the new plug-in are referenced by means of the `data` tag. This tag has as `format` attribute the name of the concrete client, in this case `client-1`, and the value of the `data` tag is the reference to the concrete resource. This information is very useful for the *plug-in manager*.

When the plug-in framework receives as input an OMDoc plug-in, the plug-in manager is invoked. The *plug-in manager* is a Common Lisp program that consists of several modules, one per client of the plug-in framework. As we have said, each one of these modules is related to a concrete client, and then, is implemented depending on the extensibility needs of each one of them. The plug-in manager, depending on the information stored in the plug-in (namely the value of the `format` attribute of the `data` tag), invokes the corresponding client module. Subsequently, this module extends the client with the indicated resources.

The plug-ins and the resources referenced by them are stored in a folder included in the plug-in framework called the *plug-in repository*.

Finally, each client of the plug-in framework has associated a file called *plug-in registry*. Each one of these files stores a list of the *plug-ins* that were added to the corresponding client. When a new plug-in is included in a client, the information about that plug-in is stored in its plug-in registry. Moreover, when a client is started its first task consists in sending the information of the *plug-in registry* to the *plug-in manager* in order to achieve the same state of the last configuration of the client.

If some problem appears during the loading of a plug-in the plug-in framework informs of the error and aborts the loading in order to avoid inconsistencies.

Once we have presented the plug-in framework, let us explain how one of its clients, that is the Kenzo framework, uses it. Another client of this framework is a customizable front-end that will be presented in Section 3.2.

3.1.2 The Kenzo framework as client of the plug-in framework

As it was discussed earlier, one of the most important issues tackled in the design of the Kenzo framework was the deployment of an extensible architecture. A good approach to solve this question consists in having a component-based architecture as base-system, and then equipping it with different components. This is the approach followed in the Kenzo framework. As was explained in the previous chapter, the base-system of the Kenzo framework is based on the *Microkernel* architectural pattern, therefore, we have a component-based architecture. Moreover, to be able to include new improvements and features, the Kenzo framework has been implemented as a client of the plug-in framework.

As we explained previously, all the Kenzo framework components are Common Lisp modules, see Section 2.2, and since Common Lisp programs are designed to be extensible, we only need to load new functions in each component by means of Common Lisp files.

This is the same method followed in Kenzo to extend its functionality.

The plug-in manager of the plug-in framework contains a component that is devoted to load new functionality in the Kenzo framework. This module extends the functionality of the Kenzo framework components with two different aims. On the one hand, to provide access to new Kenzo functionality through the Kenzo framework. On the other hand, to interact with other systems, such as Computer Algebra systems or Theorem Provers tools, by means of the Kenzo framework. Let us present the integration of new functionality in the Kenzo framework.

Let us suppose that we have developed a new module for Kenzo that allows us to construct a kind of spaces that were not included in the Kenzo system, we will see concrete examples in Chapter 5, and, of course, we want to include the new functionality in our framework. Then, we need to extend all the components of the framework by means of the following files:

- a file (let us called it `new-constructor.lisp`) with the functionality to include the new constructor developed for the Kenzo system in the Kenzo component of the internal server,
- a file (let us called it `new-constructor-is.lisp`) with the functionality to expand the interface of the internal server to support the access to the new Kenzo functionality,
- a file (let us called it `new-constructor-m.lisp`) with the functionality for the microkernel to include a new construction module. This functionality follows the pattern explained in Subsubsection 2.2.3.3 for constructions modules. Moreover, this file also expands the microkernel interface to provide access to the functionality of the new construction module,
- the new specification of the XML-Kenzo language including the new constructor. As we have explained previously in Subsection 2.2.4, this will extend the capabilities of the external server. The file containing the XML-Kenzo specification is the `XML-Kenzo.xsd` file, and
- a file (let us called it `new-constructor-a.lisp`) with the functionality which increases the adapter functionality to transform from OpenMath requests to XML-Kenzo requests.

The `new-constructor` plug-in is an OMDoc document which contains some metadata about the document and references the different files in the following way.

```
<code id="new-constructor">
   <data format="Kf/internal-server"> new-constructor.lisp </data>
   <data format="Kf/internal-server"> new-constructor-is.lisp </data>
   <data format="Kf/microkernel"> new-constructor-m.lisp </data>
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/adapter"> new-constructor-a.lisp </data>
</code>
```

Each `data` element has specified in the `format` attribute one of the Kenzo framework components (internal server, microkernel, external server and adapter). The value of a `data` element is a file which extends the Kenzo framework. The indicated component in the `format` attribute of a `data` element is extended by means of the file indicated with the value of the `data` element. For instance in the case of the first `data` element, the functionality of the internal server is extended by means of the `new-constructor.lisp` file.

The plug-in framework receives as input the `new-constructor` plug-in. Subsequently, the Kenzo framework module of the plug-in framework is invoked. This module is split in four constituents, one per Kenzo framework component. The behavior of the constituents in charge of the internal server, the microkernel and the adapter consists in loading the new functionality in the corresponding Kenzo framework component. The constituent for the external server overwrites the XML-Kenzo specification with the new one (no additional interaction is needed to extend the functionality of the external server, since when the XML-Kenzo specification is changed the external server is automatically upgraded).

Finally, the information of the new plug-in is stored in the plug-in registry of the Kenzo framework to store the configuration for further uses.

This was related to the inclusion of new Kenzo functionality in the Kenzo framework; the case of widening the Kenzo framework through the addition of an internal server (a Computer Algebra system or a Theorem Prover tool) is practically analogous. The main difference is the file related to the internal server which, instead of adding new Kenzo functionality to the current internal server, shows how to connect with the new system. Examples of the addition of new internal servers will be presented in Chapter 4.

The features associated to the Kenzo framework owing to its implementation as a client of the plug-in framework are listed below.

- Extensibility: the Kenzo framework can be extended by plug-ins. Each new functionality can be realized as an independent plug-in.
- Flexibility: each unnecessary plug-in can be removed and each necessary plug-in can be loaded at run-time. Therefore the Kenzo framework can be configured in such a way that it has only the needed functionality.
- Storage of configuration: thanks to the registry associated to the Kenzo framework,

the configuration of a session is stored for the future.

- Easy to install: the installation of plug-ins is friendly from the plug-in folder.

3.2 Increasing the usability of the Kenzo framework

The design of a client for our framework was one of the most important issues tackled in our development. The client should not only take advantage of all the enhancements included in the Kenzo framework but also be designed to increase the usability and accessibility of the Kenzo system. In this context, the program designers in Symbolic Computation always meet the same decision problem: two possible organizations[1].

1. Provide a package of procedures in the programming language L, allowing a user of this language to load this package in the standard L-environment, and to use the various functions and procedures provided in this package. The interaction with the procedures is by means of a command line interface.
2. Provide a graphical user interface (GUI) with the usual buttons, menus and other widgets to give to an inexperienced user a direct access to the most simple desired calculations, without having to learn the language L.

The main advantage of the former alternative is that the total freedom given by the language L remains available; however, the technicalities of the language L remain present as well. Moreover, the *prompt* of a command line interface does not usually provide adequate information to the user about the correct command syntax. Besides, since the user has to memorise the syntax and options of each command, it often takes considerable investment in time and effort to become proficient with the program. Therefore, command line interfaces may not be appealing to new or casual users of a software system. The vast majority of Computer Algebra systems fall in this category, including GAP, CoCoA, Macaulay, and Kenzo. On the contrary, graphical interfaces (the second alternative) are easier to use, since instead of relying on commands, the GUI communicates with the user through objects such as menus, dialog boxes and so on. These objects are a means to provide a more intuitive user interface, and supply more information than a simple prompt. However, no reward comes without its corresponding price and GUIs can slow down expert users.

Since the final users of our framework are Algebraic Topology students, teachers or researchers that usually do not have a background in Common Lisp, we opted for implementing a user-friendly front-end allowing a topologist to use the Kenzo program guiding his interaction by means of the Kenzo framework, without being disconcerted by the Lisp technicalities which are unavoidable when using the Kenzo system. This GUI is communicated with the Kenzo framework through the OpenMath interface of the

[1]These are, of course, two extreme positions: many other possibilities can be explored between them.

adapter and can be customized by means of the plug-in framework. The whole system, that is to say the two frameworks and the GUI, is called *fKenzo*, an acronym for friendly Kenzo [Her09]. We have talked at length about the Kenzo framework in Chapter 2 and also about the plug-in framework in Section 3.1, so, let us present now the GUI of the *fKenzo* system.

The *fKenzo* GUI was implemented with the *IDE* of Allegro Common Lisp [Incc]. Various features have been implemented in this GUI. They improve the interaction with the Kenzo system from the user point of view. The main features of the *fKenzo* GUI are listed below.

1. *Easy to install*: the installation process is based on a typical Windows installation.
2. *No external dependencies*: the *fKenzo* GUI does not need any additional installation to work.
3. *Functionality*: the *fKenzo* GUI allows the user to construct topological spaces of regular usage and compute homology and (some) homotopy groups of these spaces.
4. *Error handling*: this GUI is a client of the Kenzo framework; and all the enhancements included in the framework are inherited by the GUI. In this way, the user is guided in his interaction with the system and some errors are avoided.
5. *Consistent metaphors*: an advanced user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured in the GUI.
6. *Interaction styles*: the user can interact with the GUI by means of the mouse and also using keyboard shortcuts.
7. *Mathematical rendering*: the *fKenzo* GUI shows results using standard mathematical notation.
8. *Storage of sessions*: session files include the spaces constructed during a session, and can be saved and loaded in the future. Moreover, these session files can be exported, used to communicate them to other users and rendered in browsers.
9. *Storage of results*: result files storing the results obtained during a session, as in the case of session files, can be saved, exported, used to communicate them to other users and rendered in browsers.
10. *Customizable*: the *fKenzo* GUI can be configured to the needs of its users.
11. *Extensibility*: the *fKenzo* GUI can evolve with the Kenzo framework using the plug-in framework.

The rest of this section is devoted to present the GUI of the *fKenzo* program, trying to cover both the user (Subsections 3.2.1 and 3.2.2) and the developer (Subsections 3.2.3 and 3.2.4) perspectives.

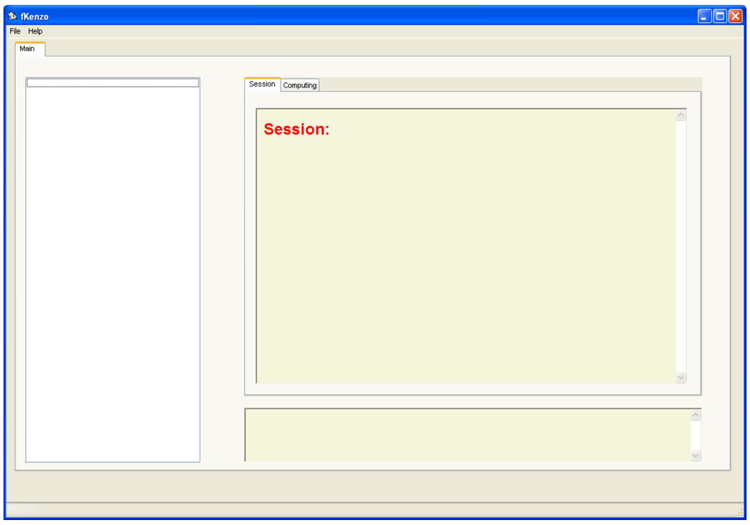


Figure 3.2: Initial screen of *fKenzo*

3.2.1 *fKenzo* GUI: a customizable user interface for the Kenzo framework

To use *fKenzo*, one can go to [Her09] and download the installer. After the installation process, the user can click on the *fKenzo* icon, accessing to an "empty" interface, which is shown in Figure 3.2.

Then, the first task of the user consists in loading some functionality in the *fKenzo* GUI through the different *modules* included in the distribution of *fKenzo*. From the user point of view, a module is a file which loads functionality in the *fKenzo* GUI.

The main toolbar of the *fKenzo* GUI is organized into two menus: *File* and *Help*. The *File* menu has the following options: *Add Module*, *Delete Module*, and *Exit*. The aim of the *Add Module* option consists in loading the functionality of a module. The user can configure the interface by means of five modules: *Chain Complexes*, *Simplicial Sets*, *Simplicial Groups*, *Abelian Simplicial Groups* and *Computing*. Each one of these modules corresponds with one of the XML-Kenzo groups explained in Subsection 2.2.1. In addition, some other *experimental* modules can be installed, such as a possibility of interfacing the GAP Computer Algebra system or the ACL2 Theorem Proving tool; these modules will be presented in Chapter 4.

When one of the "construction modules" (*Chain Complexes*, *Simplicial Sets*, *Simplicial Groups* or *Abelian Simplicial Groups*) is loaded, a new menu appears in the toolbar allowing the user to construct the spaces specified in the XML-Kenzo schema for that type, see Figure 2.5 of Subsection 2.2.1. Moreover, two new options called *Save Session* and *Load Session* are added to the *File* menu. When saving a session a file is produced containing the spaces which have been built in that session. These session files are saved using the OpenMath format and can be rendered in different browsers. These session files can be loaded, from the *Load Session* option, and allow the user to resume a saved session.

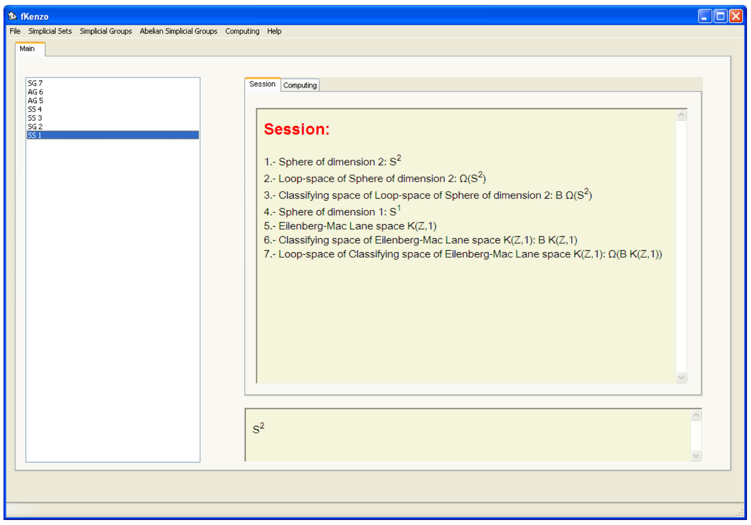
When the *Computing* module is loaded, a new menu appears in the toolbar allowing the user to compute homology and homotopy groups of the constructed spaces. Besides, a new option called *Save Computing* appears in the *File* menu. This option works in a similar way to *Save Session* but instead of saving the spaces built during the current session, it saves the computations also using the OpenMath format. However, these results cannot be reloaded into the *fKenzo* GUI, since they cannot be re-used in further computations. It is worth noting that computations depend on the state of the system, and this state cannot be exported, so we cannot re-use computations performed in a different session. When the user exits *fKenzo*, its configuration is saved for future sessions.

The visual aspect of the *fKenzo* panel is as follows. The "Main" tab contains, at its left side, a list of the spaces constructed in the current session, identified by its type (CC = Chain Complex, SS = Simplicial Set, SG = Simplicial Group, AG = Abelian simplicial Group) and its internal identification number (the `idnm` slot of the `mk-object` instance of the microkernel). When selecting one of the spaces in this list, its standard notation appears at the bottom part of the right side. At the upper part, there are two tabs "Session" (containing a textual description of the constructions made, see an example of session in Figure 3.3) and "Computing" (containing the homology and homotopy groups computed in the session; see an example in Figure 3.4). In both "Session" and "Computing" tabs the results are rendered using again standard mathematical notation.

In the *fKenzo* GUI, focus concentrates on the object (space) of interest, as in Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it (in particular, with respect to computing results). It is separated by means of tabs not only because on the division between spaces and computed results (the system moves from one to the other dynamically, putting the focus on the last user action), but also due to the extensibility of *fKenzo*. To be more precise, our system is capable of evolving to integrate with other systems (computational algebra systems or theorem proving tools), so playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns. Examples of the integration of a Computer Algebra system and a Theorem Prover tool in *fKenzo* will be presented in Chapter 4.

3.2.2 *fKenzo* in action

To illustrate the performance of the *fKenzo* program, let us consider a hypothetical scenario where a graduate course is devoted to *fibrations*, in particular introducing the functors *loop space* Ω and *classifying space* B. In the simplicial framework, [May67] is a good reference for these subjects. In this framework, if X is a connected *space*, its loop space ΩX is a simplicial group, the structural group of a universal fibration $\Omega X \hookrightarrow PX \to X$. Conversely, if G is a simplicial group, the classifying space BG is the base space also of a universal fibration $G \hookrightarrow EG \to BG$. The obvious symmetry



SG 7
AG 6
AG 5
SS 4
SS 3
SG 2
SS 1

Session:

1.- Sphere of dimension 2: S^2
2.- Loop-space of Sphere of dimension 2: $\Omega(S^2)$
3.- Classifying space of Loop-space of Sphere of dimension 2: $B\ \Omega(S^2)$
4.- Sphere of dimension 1: S^1
5.- Eilenberg-Mac Lane space $K(\mathbb{Z},1)$
6.- Classifying space of Eilenberg-Mac Lane space $K(\mathbb{Z},1)$: $B\ K(\mathbb{Z},1)$
7.- Loop-space of Classifying space of Eilenberg-Mac Lane space $K(\mathbb{Z},1)$: $\Omega(B\ K(\mathbb{Z},1))$

S^2

Figure 3.3: Example of session in *fKenzo*

between both situations naturally leads to the question: in an appropriate context, are the functors Ω and B inverse of each other?

For the composition $B\Omega$, let us compare for example the first homology groups of S^2 with the homology groups of $B\Omega S^2$, and the homology groups of ΩS^2 and $\Omega B\Omega S^2$.

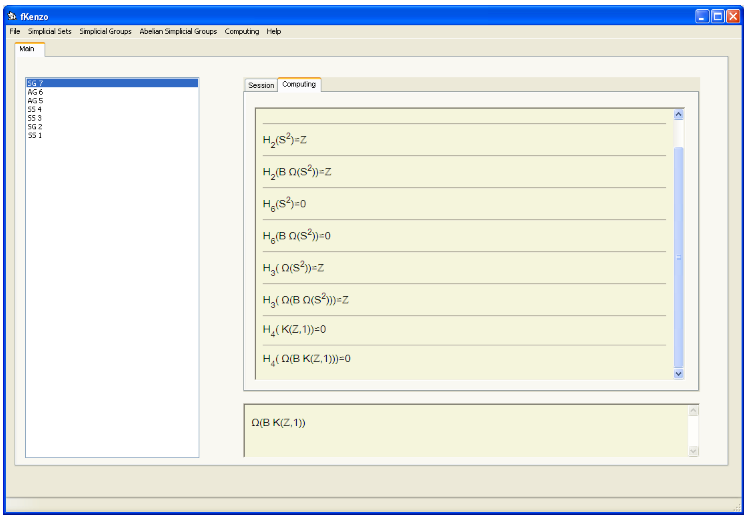
Since we are planning to work with simplicial sets and simplicial groups, we can load the necessary functionality using *File* $\rightarrow$ *Add Module*, and then choosing `Simplicial-Groups.omdoc`. The interface changes including now a new menu called *Simplicial Sets* and another one called *Simplicial Groups*. When selecting *Simplicial Sets* $\rightarrow$ *sphere*, *fKenzo* asks for a natural number, as we saw in the definition of XML-Kenzo, limited to 14. Then (see Figure 3.3) the space denoted by `SS 1` appears in the left side of the screen; when selecting it, the mathematical notation of the space appears in the bottom part of the right side of the panel. The history of the constructed spaces is shown in the "Session" tab.

If we try to construct a classifying space from the *Simplicial Group* menu, *fKenzo* informs us that it needs a simplicial group (thus likely an error is avoided). We can then construct the space ΩS^2. Since ΩS^2 is the only simplicial group in this session, when using *Simplicial Groups* $\rightarrow$ *classifying space*, ΩS^2 is the only available space appearing in the list which *fKenzo* shows. Finally, we can construct the space $\Omega B\Omega S^2$.

When loading the `Computing.omdoc` file, the menu *Computing*, where we can select "homology", becomes available. In this manner we can compute the homology groups of the spaces as can be seen in Figure 3.4.

These results give some plausibility to the relations $B\Omega = id$ and $\Omega B = id$.

The reader could wonder why the simpler case of S^1 has not been considered. Using again *fKenzo* this time a warning is produced; yet the result $B\Omega S^1 \sim S^1$ is true. But the Eilenberg-Moore spectral sequence cannot be used in this case to compute $H_*\Omega S^1$,


fKenzo
File Simplicial Sets Simplicial Groups Abelian Simplicial Groups Computing Help
Main
SG 7
AG 6
AG 5
SS 4
SS 3
SG 2
SS 1
Session Computing

H2(S2)=Z

H2(B Ω(S2))=Z

H6(S2)=0

H6(B Ω(S2))=0

H3(Ω(S2))=Z

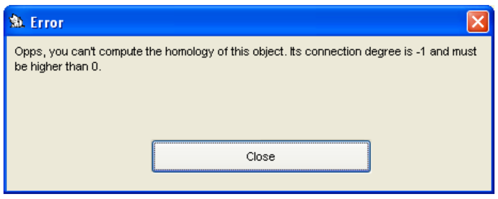
H3(Ω(B Ω(S2)))=Z

H4(K(Z,1))=0

H4(Ω(B K(Z,1)))=0

Ω(B K(Z,1))

Figure 3.4: Example of computations in *fKenzo*



Opps, you can't compute the homology of this object. Its connection degree is -1 and must be higher than 0.

Figure 3.5: Reduction degree error in *fKenzo*

because S^1 is not simply connected, and *fKenzo* checks this point (see Figure 3.5). The symmetry comparison between S^1 and ΩBS^1 fails too, for another reason: the "standard" S^1 is a topological group, but the standard simplicial representation of S^1 cannot be endowed with a structure of simplicial group. This situation is reflected in the construction of classifying spaces in *fKenzo* since the sphere S^1 does not appear in the list which *fKenzo* shows.

This is a good opportunity to introduce the Eilenberg MacLane space $K(\mathbb{Z}, 1)$, the "minimal" Kan model of the circle S^1, a simplicial group. In order to work with Eilenberg MacLane spaces in *fKenzo*, the *Abelian Simplicial Group* module should also be loaded, in this way the space $K(\mathbb{Z}, 1)$ can be built, as can be seen in Figure 3.3. The *fKenzo* comparison between the first homology groups of $K(\mathbb{Z}, 1)$ and $\Omega BK(\mathbb{Z}, 1)$ does give the expected result.

We think this illustrates how *fKenzo* can be used as a research tool, precisely a specialized computer tool, for Algebraic Topology.

Up to now, we have presented the user point of view. In the following subsections, some explanations on the development of the *fKenzo* GUI are given.

3.2.3 Customization of the *fKenzo* GUI

The most important challenge that we have faced in the development of the *fKenzo* GUI was the deployment of an extensible and modular user interface. Modularity has two aims in *fKenzo*. One of them is related to the separation of concerns in the user interface. The second one allows us to design a dynamically extensible GUI, where modules are plugged in.

3.2.3.1 Declarative programming of User Interfaces

In all the graphical user interfaces exist a separation of concerns, hence if we want to extend a GUI we need to extend it at all its levels. With respect to this aspect, our inspiration comes from [HK09], where a proposal for declarative programming of user interfaces was presented. In [HK09], the authors distinguished three constituents in any user interface: *structure*, *functionality* and *layout*.

Structure: Each user interface (*UI*) has a specific hierarchical *structure* which typically consists of basic elements (like text input fields or selection boxes) and composed elements (like dialogs).

Functionality: When a user interacts with a UI, some events are produced and the UI must respond to them. The event handlers are functions associated with events of some widget and that are called whenever such event occurs (for instance clicking over a button).

Layout: The elements of the structure are put in a layout to achieve a visually appealing appearance of the UI. In some approaches layout and structural information were mixed, however in order to obtain clearer and reusable implementations these issues should be distinguished.

The approach presented in [HK09] used the *Curry* language [Han06] to declare all the ingredients. Instead of doing a similar work, but using the Common Lisp language (recall that our GUI is implemented in Common Lisp) to all the ingredients, we have preferred to employ different technologies devoted to each one of the constituents. Let us present each one of these ingredients in our context using a concrete example, that is the window used to construct a sphere in the *fKenzo* GUI, see Figure 3.6.

The *structure* of our GUI is provided by *XUL* [H$^+$00]. XUL, XML User Interface, is Mozilla's XML-based user interface language which lets us build feature rich cross-platform applications defining the structure of all the elements of a UI. Then, a XUL description must be provided in order to define the structure of the elements of our GUI. The main reason to encode the structure of our GUI by means of XUL is the reusability of this language. The XUL code can be used in order to build forms in different environments for different applications without designing new interfaces.

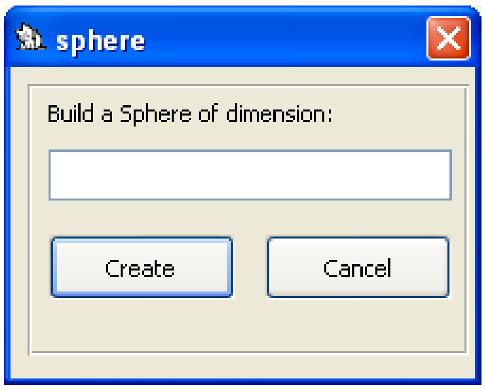


Figure 3.6: Sphere dialog

Let us examine the structure of the window of Figure 3.6. That window is called "sphere" and gathers in a groupbox the following elements: the text "Build a Sphere of dimension:", a textbox to introduce a natural number between 1 and 14, and a row that contains both "Create" and "Cancel" buttons. Moreover, each button has associated an event when its state is changed, namely when the button is clicked. We can specify that structure in the following XUL code:

```
<window name="sphere">
 <groupbox>
     <label value="Build a Sphere of dimension:"/>
     <textbox id="n" type="number" min="1" max="14"/>
     <hbox>
       <button label="Create" name="create" event="on-change"/>
       <button label="Cancel" name="cancel" event="on-change"/>
     </hbox>
 </groupbox>
</window>
```

As we have said previously, we can use the above XUL code in different clients. For instance, the previous XUL is presented in a Mozilla browser [Moz] as shown in Figure 3.7.

As can be thought, providing the XUL description of an element of a GUI can be a tedious task due to the XML nature of XUL. To make this task easier, an interpreter which is able to convert from the Allegro Common Lisp *IDE* forms to their XUL representation, and viceversa, has been developed. This is a more comfortable way of working because we define the forms in the Allegro *IDE* using a graphical interface, and then, the interpreter automatically generates the XUL code.

The *functionality* of the elements of our GUI, that is the set of *event handlers*, has been programmed in Common Lisp keeping the following convention in order to define the names of the *event handlers* functions:

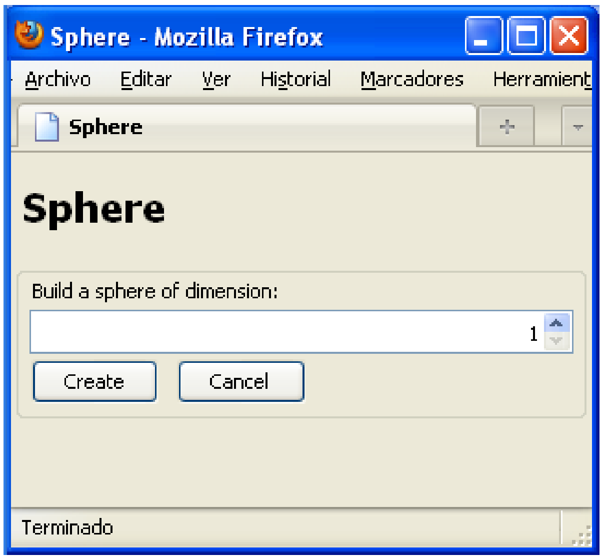

Sphere - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramient
Sphere
Sphere
Build a sphere of dimension:
1
Create
Cancel
Terminado

Figure 3.7: Sphere form in Firefox browser

```
(defun <window-name>-<element-name>-<event> (params)
  ;; event handler code)
```

where `<window-name>`, `<element-name>` and `<event>` must be replaced with the name of the dialog, the name of the element and the event, respectively. For instance, the event associated with the “Create” button of the sphere dialog, see Figure 3.6, is codified as follows:

```
(defun sphere-create-on-change (params)
  ;; event handler code)
```

Finally, the *layout* of the elements of our GUI, that is the visual appearance of the GUI, can be configured by means of a *stylesheet* [K$^+$07]. For instance, if we define the following (fragment of a) stylesheet:

```
<xsl:template name="window">
     <xsl:param name="color">blue</xsl:param>
</xsl:template>

<xsl:template name="groupbox">
     <xsl:param name="color">orange</xsl:param>
</xsl:template>

<xsl:template name="label">
     <xsl:param name="background-color">yellow</xsl:param>
     <xsl:param name="font-color">red</xsl:param>
</xsl:template>
```

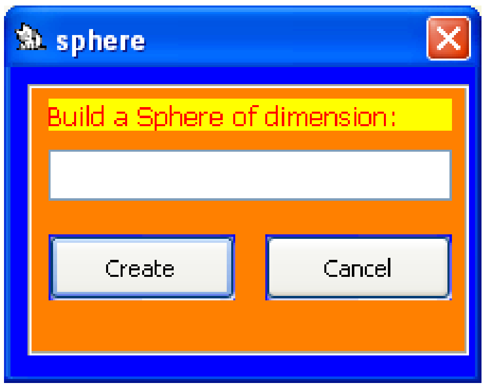


Figure 3.8: Sphere dialog with a stylesheet

the sphere dialog would have the aspect shown in Figure 3.8. That is to say, the stylesheet is used to modify the visual attributes of the elements of the GUI. If we do not provide a stylesheet the default values of the visual aspect attributes are used.

In this way, all the graphical constituents of the interface can be defined.

The next subsubsection is devoted to present how the information related to the different constituents is stored in our modules.

3.2.3.2 *fKenzo* GUI modules

A *fKenzo* GUI module is an OMDoc document that references at least two resources: a file that contains the structure of the graphical elements of the module and another one containing the functionality of those elements. In addition, a *fKenzo* GUI module can reference a file with the layout. Besides, a *fKenzo* GUI module provides some metadata (authorship, title and so on). The following conventions have been followed in the four *fKenzo* GUI construction modules and also in the computation one.

A *fKenzo* GUI module follows the schema presented for the rest of plug-ins of the plug-in framework, see Subsection 3.1.1. For instance, in the case of the `Simplicial Groups` module:

```
<code id="Simplicial Groups">
 <data format="fKenzo/GUI/structure"> simplicial-groups-structure </data>
 <data format="fKenzo/GUI/functionality"> simplicial-groups-functionality </data>
</code>
```

The first reference corresponds to the structure of the graphical constituents of that module. This document is called “<Module>-structure” where “<Module>” is the name of the correspondent module. This file is an OMDoc document. To introduce XUL code in these documents we have used an OMDoc feature called *OpenMath foreign objects* (the `<OMForeign>` tag) which allows us to introduce non-OpenMath XML in OMDoc files. For instance, the module of Simplicial Groups includes a new menu with two menu items: one to construct Loop spaces (which has associated the shortcut "Ctrl+L" and the event `show-loop-space`) and another one to Classifying spaces (which has associated the shortcut "Ctrl+Y" and the event `show-classifying`). In this case the structure of the new menu is stored in the document `simplicial-groups-structure` with the following XUL code.

```
<OMForeign>
  <toolbarbutton type="menu" label="Simplicial Groups">
    <menupopup>
      <menuitem label="Loop Space" acceltext="Ctrl" accesskey="L"
                command="show-loop-space"/>
      <menuitem label="Classifying Space" acceltext="Ctrl" accesskey="Y"
                command="show-classifying"/>
    </menupopup>
  </toolbarbutton>
</OMForeign>
```

The functionality related to a concrete module is encoded in an OMDoc document called "<Module>-functionality" where "<Module>" is the name of the corresponding module. To this aim, we use an OMDoc feature which allows us to introduce code (in our case Common Lisp functions) in OMDoc files by means of the `code` tag. For instance, the functionality of the event called `show-loop-space` associated to the *Loop Space* menu item of the *Simplicial Groups* menu is encoded in the `simplicial-groups-functionality` document as follows.

```
<code id="show-loop-space">
    <metadata>
        <description> The event associated to the Loop Space menuitem </description>
    </metadata>
    <data format="application/fKenzo">
        <![CDATA[ (defun show-loop-space ()
                     ;; code ) ]]>
    </data>
</code>
```

Finally, the resource related to the layout is optional, and in particular the *fKenzo* GUI modules use the default layout, so they never reference any layout file. Anyway, if we want to provide a different appearance for the graphical elements of a module we can define an OMDoc document where we encode the stylesheet, which customizes the visual aspect of the elements of the module, using the same feature employed in the case of structure documents, that is the `<OMForeign>` tag.

The organization presented here allows us to deal with the design of a dynamically extensible GUI, where modules are plugged in. Our front-end becomes *extensible* thanks

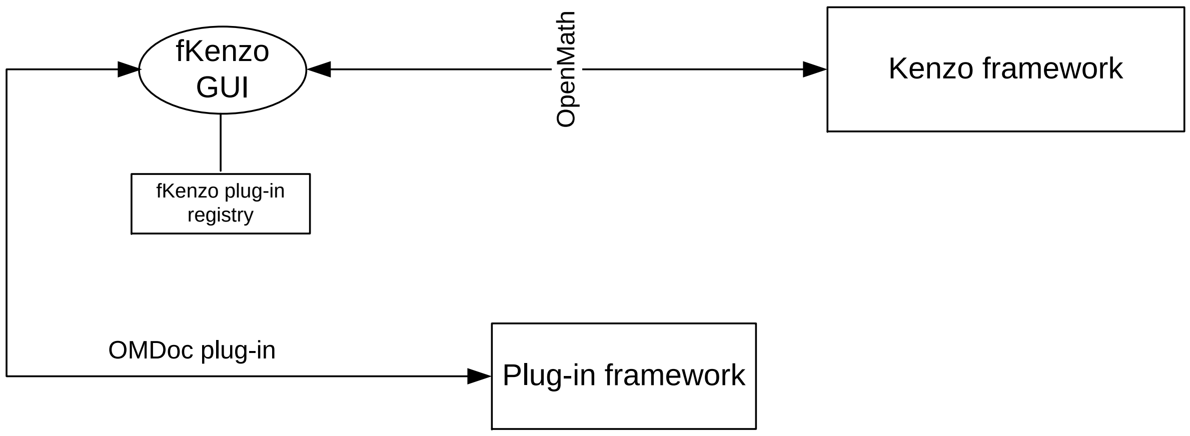


Figure 3.9: Plug-in framework and *fKenzo* GUI

to the plug-in framework since each user interface unit is encoded in a unique OMDoc file, with its inner modular organization: structure, functionality and (optionally) layout.

3.2.3.3 *fKenzo* GUI as client of the plug-in framework

To tackle the extensibility question in our user interface, the *fKenzo* GUI has been designed not only as a client of the Kenzo framework, but also as a client of the plug-in framework presented in Section 3.1. In this way, the *fKenzo* GUI can be extended in an easy way by means of modules, described in the previous subsubsection. In this context, instead of using the term *plug-in* we prefer the term *module* which is more appropriate (an application which is extended by means of modules does not have any functionality, apart from the one which allows us to load modules, if we have not added any module, as the *fKenzo* GUI; on the contrary, an application which is extended by means of plug-ins can work without adding them, as the Kenzo framework). The relations among the *fKenzo* GUI, the Kenzo framework, and the plug-in framework are depicted in Figure 3.9.

The plug-in manager (see Subsection 3.1.1) of the plug-in framework contains a module in charge of extending the *fKenzo* GUI. This module extends the GUI providing access to a concrete part of the functionality of the Kenzo framework by means of the five modules explained for the *fKenzo* GUI (the four construction modules and the computing one).

In particular, the plug-in manager of the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. This module is split in two constituents. The first one is an interpreter in charge of converting from XUL code to Common Lisp code the different graphical components. In addition if a layout file is specified, then the layout properties are applied to the graphical components. The second one associates the functionality of the event handlers to the elements defined in the structure document.

Then, it is enough to produce an OMDoc file with the suitable components, and then it can be interpreted and added in our GUI. It is exactly what happened when in Subsection 3.2.2 we described the way of working with *fKenzo*: using *File* $\rightarrow$ *Add Module* with the `Simplicial-Groups.omdoc` module sends this module to the plug-in framework. Subsequently, the plug-in manager invokes the *fKenzo* module (one of the subcomponents of the plug-in manager) that extends the user interface.

The implementation of the *fKenzo* GUI as a client of the plug-in framework shows the feasibility and usefulness of this framework.

The *fKenzo* GUI features partly owing to the implementation of the user interface as a client of the plug-in framework are listed below.

- Modularity: the GUI is organized in different modules, each one devoted to a concrete concern.
- Extensibility: the GUI can be extended by the modules. Each new functionality can be realized as an independent module.
- Flexibility: each unnecessary module can be removed and each necessary module can be loaded at run-time. Therefore the GUI can be configured in such a way that it has only the needed functionality.
- Easy to install: the installation of modules is friendly (only select a file with the option `Add Module`) from the plug-in folder.
- Internet based update: the GUI supports an update mechanism from the `Help` menu. This allows the GUI to download new modules or updates.
- Storage of configuration: the GUI configuration is automatically saved for further sessions.

3.2.3.4 Extending the Kenzo framework from the *fKenzo* GUI

The previous subsubsections have been devoted to explain how the *fKenzo* GUI can be customized by means of the plug-in framework. Moreover, the plug-in framework can also be employed to increase the functionality of the Kenzo framework as we presented in Subsubsection 3.1.2. Besides, in the same way that we wanted that the Kenzo framework could evolve at the same time as Kenzo, we also hope that the *fKenzo* GUI will be able to evolve at the same time that the Kenzo framework.

Up to now, the *fKenzo* modules that we have presented (the four construction modules and the computation one) to customize the *fKenzo* GUI do not suppose any improvement in the Kenzo framework. However, it is worth noting that the modules for the GUI not only can extend the GUI but also the Kenzo framework.

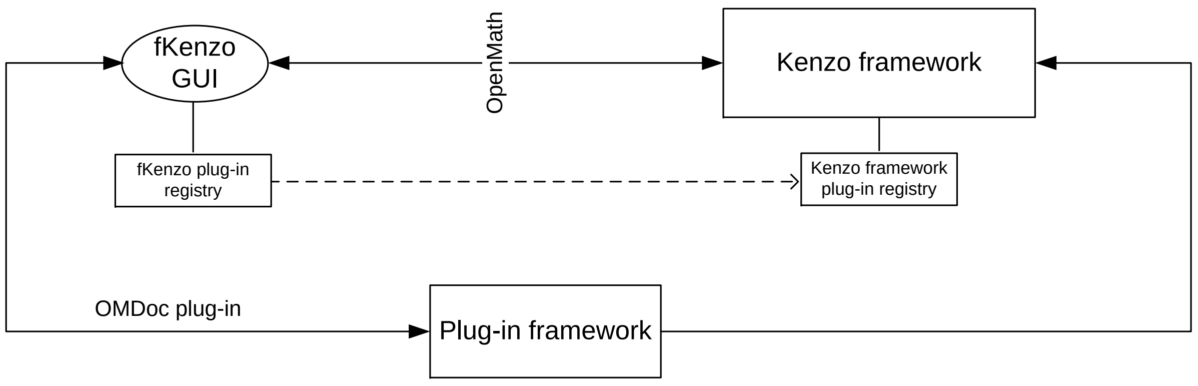


Figure 3.10: Plug-in framework, *fKenzo* GUI and Kenzo framework

Let us retake the example presented in Subsubsection 3.1.2 where a plug-in called `new-constructor` was defined to increase the functionality of the Kenzo framework by means of a new constructor. Now, following the guidelines of Subsubsection 3.2.3.2, we can define three files (structure, functionality and layout) to customize the GUI to interact with the new constructor. Finally, we define a *fKenzo* GUI module which not only references the three files (structure, functionality and layout) to customize the GUI but also the plug-in which adds new functionality to the Kenzo framework.

```
<code id="new-constructor">
 <data format="fKenzo/GUI/structure"> new-constructor-structure </data>
 <data format="fKenzo/GUI/functionality"> new-constructor-functionality </data>
 <data format="fKenzo/GUI/layout"> new-constructor-layout </data>
 <data format="fKenzo/GUI/Kf"> new-constructor-plug-in </data>
</code>
```

When the user selects this new module from the *Add Module* option, the plug-in framework will extend both the *fKenzo* GUI and the Kenzo framework. In Subsubsection 3.2.3.3, we explained that the plug-in framework includes a module in charge of processing the modules related to the *fKenzo* GUI. We said that this module is split in two constituents but we have included a new constituent devoted to invoke the Kenzo framework module of the plug-in framework for the cases explained in this subsubsection. This last constituent is only invoked if the *fKenzo* GUI module references a Kenzo framework plug-in, in that case the Kenzo framework module of the plug-in manager is also called.

In addition, the *fKenzo GUI plug-in registry* and the *Kenzo framework plug-in registry* must be coherent in order to avoid inconsistencies in the whole system.

A high level perspective of the interaction between the two frameworks and the GUI can be seen in Figure 3.10.

This extensibility principle makes very easy to us the incorporation of experimental features to the system, without interfering with the already running modules, as we will see in Chapters 4 and 5.

3.2.4 Interaction design

In the previous subsection we have explained the design decisions that we took to develop an extensible and modular interface, probably the most important feature of the *fKenzo* GUI from the developer perspective. However, other decisions were taken on the *fKenzo* user interface design.

3.2.4.1 Task model

The first idea guiding the construction of a user interface must be the objectives of the interaction. In *fKenzo* there is only one higher-level objective: to compute groups of spaces. This main objective is later on broken in several subobjectives, trying to emulate the way of thinking of a typical Kenzo user. Once this first objective analysis is done, the next step is to design a task model. That is to say, a hierarchical planning of the main actions the user should undertake to get his objectives. This is a previous step before devising the navigation of the user, which will give the concrete guidelines needed to implement the interface.

In our case, the two main actions of the system are: (1) computing groups, and (2) constructing spaces. Note that the second task is necessary to carry out the first one. In turn, the task of constructing spaces can be separated into: (a) constructing new fresh spaces and (b) loading spaces from a previous session. Thus, the notion of session comes on the scene. With respect to the construction of fresh spaces, once the user has decided to go for it, he should decide which type of space he wants to build: simplicial set, simplicial group, and so on. Note that the construction of a space of a type can involve the construction of other space of whether its same type or a different type. This third layer of tasks gives us the module organization of the interface, while computing produces a separated module.

The task design is organized hierarchically by diagrammatic means. See in Figure 3.11 a first decomposition layer depicted by means of a package diagram [Gro09]. Each task (each frame) is linked to auxiliary tasks (giving a horizontal dependency structure). Then, each frame is described in more detail (vertical structure) by making explicit its subtasks graph.

Task modeling provides us with both the high level modular structure and the different steps needed to reach a user subobjective. The concrete actions a user should perform to accomplish the tasks are devised in the control and navigation models.

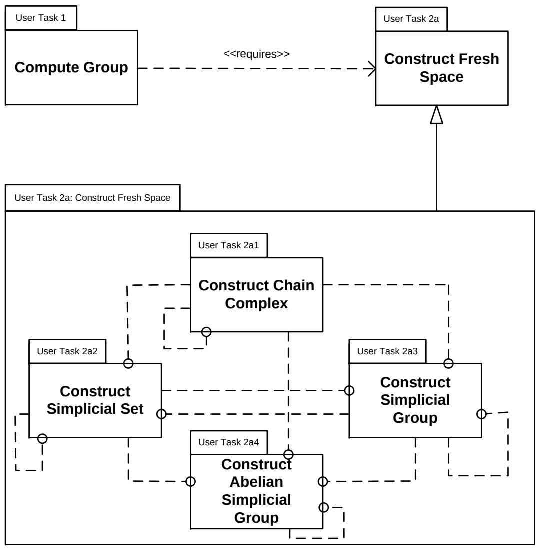


Figure 3.11: Hierarchical decomposition of the "Construct Fresh Space" user task

3.2.4.2 Control and navigation model

The design of the interaction between a user and a computer program involves well-known challenges (use of convenient metaphors, consistency of the control through the whole application, and so on). In order to avoid some frequent drawbacks we have followed the guidelines of the Noesis method (see [DZ07] for the general theory, and [CMDZ06] for the design of reactive systems). In particular, our development has been supported by the Noesis models for control and navigation in user interfaces. These graph-supported models enable an exhaustive traversal study of the interfaces, allowing the analyst to detect errors, disconnected areas, lack of uniformity, and so on, before the programming phase. Figure 3.12 shows the control and navigation submodel describing the construction of a loop space $\Omega^n X$. Different kinds of interactions are graphically represented in Figure 3.12 by different icons. For instance, selecting from a list is depicted with a form icon; directly writing an input is depicted with a pen, and so on. These pictures help the programmer to get a quick overall view of the different controls to be implemented.

Let us observe that this diagrammatic control model is *abstract*, in the sense that nothing is said about the concrete way the transitions should be translated into the user interface. In fact, in the *fKenzo* GUI this model is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through control-keys. The second style has been included thinking of advanced users, who want to use shortcuts to access the facilities of the interface. The adaptation to different kinds of users is one of the principles for design usability in [Nie94], and has been considered, as the rest of principles, in our development.

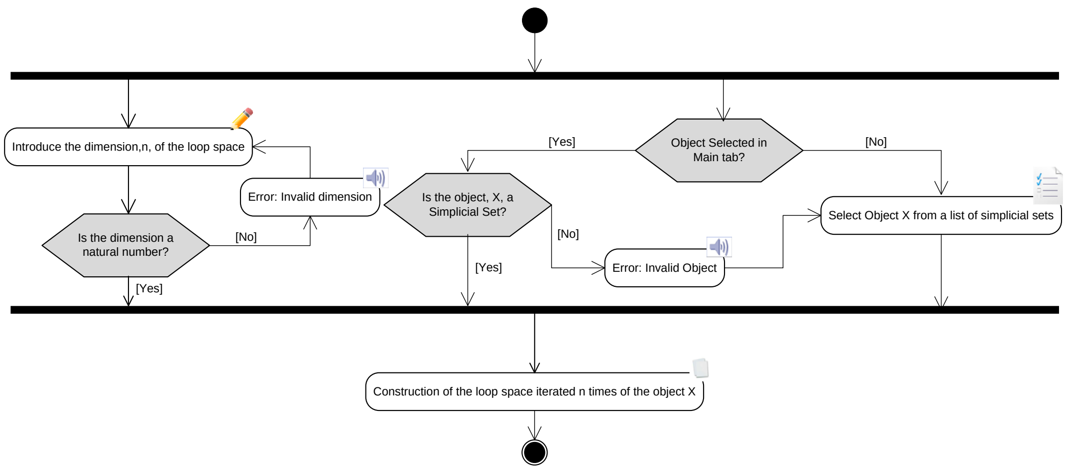

Introduce the dimension,n, of the loop space
Error: Invalid dimension
Is the dimension a natural number?
[No]
[Yes]
Is the object, X, a Simplicial Set?
[Yes]
[No]
Object Selected in Main tab?
[Yes]
[No]
Error: Invalid Object
Select Object X from a list of simplicial sets
Construction of the loop space iterated n times of the object X

Figure 3.12: Control graph for the construction of $\Omega^n X$

3.2.4.3 Challenges in the design of the *fKenzo* GUI

User interface design is a central issue for the usability of a software system. Ideally, the design of a user interface should be done following certain rules, such as those listed in guidelines documents, see for instance [KBN04]. However these guidelines have hundreds of rules, then instead of strictly following those rules the design of a user interface abides by heuristics rules based on common sense. A small set of heuristic principles more suited as the basis for practical design of user interfaces was given in [Nie94]. We have used the nine principles given in [Nie94] for guiding the design decisions of the *fKenzo* GUI.

1. *Visibility of system status.* the *fKenzo* GUI should always keep users informed about what is going on. As we have said previously, the *fKenzo* GUI can be used to construct spaces and compute groups. In the case of the spaces constructed in the *fKenzo* GUI this first principle is achieved, since the *fKenzo* GUI shows a list with the spaces constructed in the current session to the user. Related to the computation of groups, some calculations in Algebraic Topology may need several hours, then to dealt with the *visibility of the fKenzo status*, a message informs the user of this situation when a computation is performed. Besides, the *fKenzo* GUI allows the user to interrupt the current computation and keep on working with his session.

2. *Match between system and the real world.* the *fKenzo* GUI should show results using well-known Algebraic Topology mathematical notation. This second aspect has been solved by means of combining OpenMath and stylesheets. When selecting one of the spaces of the left list of the main *fKenzo* GUI window, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. A *stylesheet* has been defined using *XSLT* [K$^+$07]. This *stylesheet* is in charge of rendering using mathematical notation the object represented with an OpenMath instruction. In both "Session" and "Computing" tabs the results are also rendered using mathematical notation thanks to the same *stylesheet*.

3. *Consistency and standards.* A user of Kenzo feels comfortable with the *fKenzo* GUI; in particular, the typical two steps process (first constructing an space, then computing groups associated to it) is explicitly and graphically captured. Then, the *fKenzo* GUI is *consistent* with respect to Kenzo. It is worthwhile noting that this is the most influential requirement with respect to the visual aspect of our interface. In addition to the menu bar, there are three main parts in the screen: a left part, with a listing of the objects already constructed in the current session, a right panel with several tabs, and a bottom part with the standard mathematical representation of the object selected. Thus, focus concentrates on the object (space) of interest, as in Common Lisp/Kenzo. The central panel takes up most of the place in the interface, because it is the most growing part of it. It is separated by means of tabs not only because on the division among spaces and computing results (the system moves dynamically from one to the other), but also due to the capability of integrating other systems, playing with tabs in the central panel allows us to produce a user sensation of indefinite space and separation of concerns.

4. *Error prevention.* The *fKenzo* GUI should forbid the user the manipulations raising errors. The most important design decision related to this point is the use of the GUI as client of the Kenzo framework. In this way, all the enhancements included in the framework are inherited by the GUI forbidding the user some manipulations raising errors and guiding his interaction with the system.

5. *Recognition rather than recall.* The *fKenzo* GUI should minimize the user's memory load. This design principle is fulfilled thanks to the combination of stylesheets and OpenMath that are used to inform the user about the selected space. Moreover, this principle was important for the design of the dialogs used to construct spaces from other spaces and compute groups. For example, if a space is selected in the screen of the *fKenzo* GUI the dialogs used to construct spaces from other spaces and compute groups take that space by default as input, then the user does not need to select the space in the dialog.

6. *Flexibility and efficiency of use.* The *fKenzo* GUI should provide shortcuts that speed up the interaction for the expert user and also suit the needs of each user. To handle this question, the interaction with the GUI is implemented in two different manners: one by means of the "menu & mouse" style, and the other one through control-keys used as accelerators. Moreover, thanks to the organization of the system by means of modules, a user can load the functionality that he needs.

7. *Minimalist design.* The *fKenzo* GUI should not contain irrelevant information; to that aim, the dialogs showed to the user only contain the key information.

8. *Good error messages.* the *fKenzo* GUI should indicate precisely the problems. Thanks to the use of the GUI as client of the Kenzo framework, the warnings obtained from the framework are used in the GUI. These warnings express in plain language the problem, and constructively suggest a solution.

9. *Help and documentation.* The *fKenzo* GUI should include a good documentation. Even though the *fKenzo* GUI can be used without documentation, help and documentation are provided in the *Help* menu. This help is always available, is focused on the user's tasks, lists concrete steps to be carried out, and contains both information about the use of the *fKenzo* GUI and the underlying mathematical theory.

In summary, design principles have been followed in the *fKenzo* GUI obtaining in this way a usable interface for the Kenzo system through the Kenzo framework.

Chapter 4

Interoperability for Algebraic Topology

When working in Mathematics, a researcher or student uses different sources of information to solve a problem. Typically, he can consult some papers or textbooks, make some computations with a Computer Algebra system, check the results against some known tables or, more rarely, try some conjectures with a Proof Assistant tool. That is to say, mathematical knowledge is dispersed among several sources.

Our aim consists of mechanizing, in some particular cases, the management of these multiple-sources information systems by means of *fKenzo*. Since it would be too pretentious to try to fully solve this problem, we work in a very concrete context. Thematically, we restrict ourselves to (a subset of) Algebraic Topology. With respect to the sources, in order to have a representation wide enough, we have chosen two Computer Algebra systems (Kenzo and GAP), and a Theorem Prover (ACL2).

This aim has some common concerns with the well known SAGE project [Ste] and the Software Composability Science project [F$^+$08], since all of us are trying to join several mathematical software packages. There are, however, important differences. SAGE is an integrated system in which users interact with the SAGE front-end and the Computer Algebra systems are used as back-end servers. The representation of mathematical objects in SAGE is based on an internal representation. On the other hand, the Science project provides a framework that allows services to be both provided and consumed by any Computer Algebra system. The Science project uses OpenMath as representation for mathematical objects.

Our approach combines some of the characteristics of both SAGE and Science projects but also has some significant differences. As in the SAGE initiative, we provide a common front-end to use the different systems that are integrated in the Kenzo framework as internal servers; however, the SAGE front-end is a command line interface, with the problems that this approach presents for a non expert user (we discussed this question in Section 3.2); whereas, in our case we provide a friendly graphical user interface, the *fKenzo* GUI. Moreover, when a user wants to invoke a system from SAGE, he must do it explicitly; on the contrary our front-end hides the details about the system employed at each moment. To communicate the mathematical objects between our system and other ones we use both OpenMath, as in the Science project, and our XML-Kenzo language. In addition, we use some of the programs developed by the Science initiative in our development.

It is worth noting that we do not only want to integrate Computer Algebra systems in our framework (as in the case of SAGE and Science projects), but also Theorem Prover tools.

In addition, our final aim has consisted not only in having several Computer Algebra systems and Theorem Prover tools, and use them individually by means of a common GUI, but also in making them work in a coordinate and collaborative way to obtain new tools and results not reachable if we use severally each system.

In general, the integration of Computer Algebra systems and systems for mechanized reasoning tries to overcome their weak points: efficiency in the case of Theorem Provers and consistency in the case of Computer Algebra systems. There are several possibilities to interface Computer Algebra and Theorem Prover systems, let us cite only three of them: (1) use a Computer Algebra system as a hint engine for a Theorem Prover, (2) use a Computer Algebra system as a proof engine for a Theorem Prover, and (3) prove in the Theorem Prover the correctness of Computer Algebra algorithms.

Both first and second cases involve a certain *degree of trust* of the prover to the Computer Algebra system; several experiments have been performed in these lines, see for instance the interaction between HOL and Maple [HT98] or the communication between Coq and GAP [KKL]. The last track (prove in the Theorem Prover the correctness of Computer Algebra algorithms) allows us to build more reliable and accurate components for a Computer Algebra system, for instance Buchberger's algorithm for computing Gröbner basis (one of the most important algorithms in Computer Algebra) has been formalized in [MPRR10] using the ACL2 theorem prover. In the work presented in this memoir, we have focussed on the third aspect.

The rest of this chapter is mainly organized in two parts devoted to present how the GAP Computer Algebra system and the ACL2 Theorem Prover were integrated in our system as new internal servers.

First of all, the integration and composability of the GAP Computer Algebra system in *fKenzo* is presented. Namely, first things first, to achieve the composability of GAP in our framework we need first its integration; then, Section 4.1 presents the integration of GAP. How Kenzo and GAP work in a coordinate and collaborative way is explained in Section 4.2.

In the second part, some explanations about how ACL2 is integrated in *fKenzo* are given. The integration of ACL2 in the system is shown in Section 4.3. The coordinate way of working of Kenzo, GAP and ACL2 is presented in Section 4.4.

Finally, Section 4.5 is devoted to present a methodology to integrate different systems as internal servers in our system.

4.1 Integration of the GAP Computer Algebra system

The second Computer Algebra system that we have integrated in our framework (the first one was Kenzo) is GAP [GAP] with its HAP package [Ell09]. This decision was taken inspired by the work presented in [RER09] where Kenzo and GAP (and, namely, its HAP package) have been communicated to create new tools. From now on, GAP/HAP refers to the GAP Computer Algebra system where its HAP package has been loaded.

In this first stage of the integration of GAP/HAP in our system, we have provided support for the construction of cyclic groups and the computation of their homology groups.

The rest of this section is organized as follows. Subsection 4.1.1 introduces the basic background about group homology; in Subsection 4.1.2 the GAP Computer Algebra system and its HAP package are presented; Subsection 4.1.3 explains how our framework is extended to include the GAP/HAP functionality. Moreover, an enhancement of the framework to deal with the properties of their objects is presented in Subsection 4.1.4. Finally the way of broadening the *fKenzo* GUI to include the GAP/HAP Computer Algebra system is detailed in Subsection 4.1.5.

4.1.1 Mathematical preliminaries

The following definitions and important results about homology of groups can be found in [Bro82].

Definition 4.1. Let G be a group and $\mathbb{Z}G$ be the *integral group ring* of G (see [Bro82]). A *resolution* F_* for a group G is an acyclic chain complex of $\mathbb{Z}G$-modules

$$\ldots \rightarrow F_2 \xrightarrow{d_2} F_1 \xrightarrow{d_1} F_0 \xrightarrow{\varepsilon} F_{-1} = \mathbb{Z} \rightarrow 0$$

where $F_{-1} = \mathbb{Z}$ is considered a $\mathbb{Z}G$-module with the trivial action and $F_i = 0$ for $i < -1$. The map $\varepsilon : F_0 \rightarrow F_{-1} = \mathbb{Z}$ is called the *augmentation.* If F_i is free for all $i \geq 0$, then F_* is said to be a free resolution.

Given a free resolution F_*, one can consider the chain complex of $\mathbb{Z}$-modules (that is to say, abelian groups) $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ defined by

$$C_n = (\mathbb{Z} \otimes_{\mathbb{Z}G} F_*)_n, \quad n \geq 0$$

(where $\mathbb{Z} \equiv C_*(\mathbb{Z}, 0)$ is the chain complex with only one non-null $\mathbb{Z}G$-module in dimension 0) with differential maps $d_{C_n} : C_n \to C_{n-1}$ induced by $d_n : F_n \to F_{n-1}$.

Although the chain complex of $\mathbb{Z}G$-modules F_* is acyclic, $C_* = \mathbb{Z} \otimes_{\mathbb{Z}G} F_*$ is, in general, not exact and its homology groups are thus not null. An important result in homology of groups claims that the homology groups are independent from the chosen resolution for G.

Theorem 4.2. Let G be a group and F_*, F'_* be two free resolutions of G. Then

$$H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F_*) \cong H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F'_*), \quad for \;\; all \;\; n \in \mathbb{N}.$$

This theorem leads to the following definition.

Definition 4.3. Given a group G, *the homology groups* $H_n(G)$ are defined as

$$H_n(G) = H_n(\mathbb{Z} \otimes_{\mathbb{Z}G} F_*)$$

where F_* is any free resolution for G.

The problem now consists of determining a free resolution F_* for G. For some particular cases, small resolutions can be directly constructed. For instance, let G be the cyclic group of order m with generator t. The resolution F_* for G

$$\ldots \xrightarrow{t-1} \mathbb{Z}G \xrightarrow{N} \mathbb{Z}G \xrightarrow{t-1} \mathbb{Z} \to 0,$$

where N denotes the norm element $1+t+\ldots+t^{m-1}$ of $\mathbb{Z}G$, produces the chain complex of abelian groups

$$\ldots \xrightarrow{0} \mathbb{Z} \xrightarrow{m} \mathbb{Z} \xrightarrow{0} \mathbb{Z} \to 0$$

and therefore

$$H_i(G) = \begin{cases} \mathbb{Z} & \text{if } i = 0 \\ \mathbb{Z}/m\mathbb{Z} & \text{if } i \text{ is odd}, i > 0 \\ 0 & \text{if } i \text{ is even}, i > 0 \end{cases}$$

In general is not easy to obtain a resolution for a group G. As we will see in Subsection 4.1.2, the GAP package HAP has been designed as a tool for constructing resolutions for a wide variety of groups.

4.1.2 GAP and HAP

GAP [GAP] is a Computer Algebra system, well-known for its contributions, in particular, in the area of Computational Group Theory.

HAP [Ell09] is a homological algebra library (developed by Graham Ellis) for use with GAP still under development. The initial focus of this package is on computations related to cohomology groups. A range of finite and infinite groups are handled, with particular emphasis on integral coefficients. It also contains some functions for the integral (co)homology of: Lie rings, Leibniz rings, cat-1-groups an digital topological spaces.

Let us see an example of the use of the GAP/HAP system. To construct the cyclic group of dimension 5 in the GAP system we proceed in the following way:

```
gap> c5:=CyclicGroup(5); ✠
<pc group of size 5 with 1 generators>
```

A GAP display must be read as follows. The initial `gap>` is the prompt of GAP. The user types out a gap statement, here `c5:=CyclicGroup(5);` and the maltese cross ✠ (in fact not visible on the user screen) marks in this text the end of the GAP statement. The Return key then asks GAP to *evaluate* the GAP statement. Here the cyclic group C_5 is constructed by the GAP `CyclicGroup` function (functions in GAP are case sensitive), taking account of the argument 5, and this cyclic group is *assigned* to the symbol `c5` for later use. Also evaluating a GAP statement *returns* an object, the result of the evaluation, in this case the cyclic group of dimension 5, displayed as `<pc group of size 5 with 1 generators>`.

From GAP we can obtain properties of the group; for instance, if we want to know if our group is abelian we proceed as follows:

```
gap> IsAbelian(c5); ✠
true
```

Therefore, the cyclic group C_5 is an abelian group. It is worth noting that this kind of properties (for instance, being abelian, cyclic, solvable and so on) are assigned to the group based on the mathematical definition of the group when it is constructed, this means that GAP does not perform any computation to know if the group satisfies a property but just checks if the object has associated it.

The homology groups of C_5 are computable by means of the HAP `GroupHomology` function (this function has as input two arguments, a group G and an integer n, and the output is $H_n(G)$) in the following way:

```
gap> GroupHomology(c5,0); ✠
[0]
```

The result must be interpreted as stating $H_0(C_5) = \mathbb{Z}$. We can compute several homology groups if we introduce the instructions in a loop; for instance, to compute $H_n(C_5)$ with $0 \leq n \leq 6$:

```
gap> for i in [0..6] do Print(GroupHomology(c5,i), " "); od; ✠
[ 0 ] [ 5 ] [  ] [ 5 ] [  ] [ 5 ] [  ]
```

In this case, the above result must be interpreted as $H_0(C_5) = \mathbb{Z}, H_1(C_5) = \mathbb{Z}/5\mathbb{Z}, H_2(C_5) = 0, H_3(C_5) = \mathbb{Z}/5\mathbb{Z}, H_4(C_5) = 0, H_5(C_5) = \mathbb{Z}/5\mathbb{Z}$ and $H_6(C_5) = 0$ (the expected results as we have seen in the previous subsection).

As we have just seen, HAP can be used to make basic calculations in the homology of finite and infinite groups with the command `GroupHomology`. This command performs two steps to compute the homology groups of a group G: (1) construct a free resolution F_* of G and (2) compute the homology from $\mathbb{Z} \otimes_{\mathbb{Z}G} F_*$ using a version of the Smith Normal Form algorithm [Veb31].

4.1.3 Integration of GAP/HAP

As we claimed in Section 2.1 one of the challenges of the *fKenzo* system was the integration of different tools as new internal servers. This is the first example of this integration.

We present here the integration of some of the functionality of the GAP/HAP system in *fKenzo*, namely the functionality devoted to construct cyclic groups, obtain group properties and compute group homology. The procedure followed here to integrate that functionality (a simple example) is general enough to be applied in the integration of other more interesting GAP/HAP cases without any special hindrance. We have developed a plug-in following the guidelines given in Subsubsection 3.1.2. This new plug-in references the following resources:

```
<code id="gap">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> gap-invoker.lisp </data>
   <data format="Kf/microkernel"> gap-cyclic-m.lisp </data>
   <data format="Kf/microkernel"> gap-homology-m.lisp </data>
   <data format="Kf/adapter"> gap-a.lisp </data>
</code>
```

These resources deserve a detailed explanation that is provided in the following subsubsections.

4.1.3.1 Extending the XML-Kenzo schema

We want to introduce new functionality in our framework which allows us to construct cyclic groups, compute their homology groups and obtain some of their properties by means of GAP/HAP. Therefore, we have extended the XML-Kenzo specification to

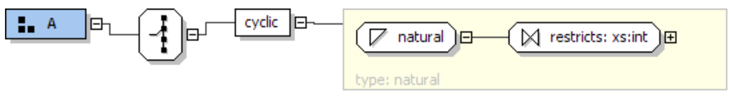


Figure 4.1: `cyclic` element in XML-Kenzo

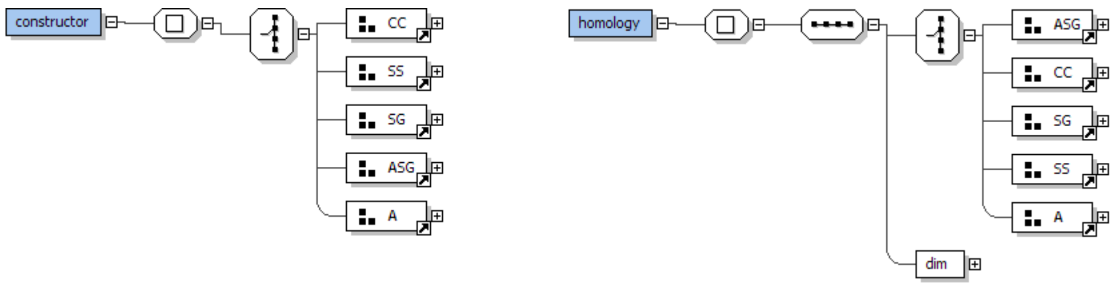


Figure 4.2: `constructor` and `homology` elements

represent the requests and results related to GAP/HAP. In the XML-Kenzo specification, we have defined a new element: `cyclic`, which has one child of natural number type (see Figure 4.1) and belongs to a new type called `A` (the type of GAP abelian groups).

Moreover, the group `A` has been included as a possible child of both `constructor` and `homology` elements, see Figure 4.2. Therefore, the two following requests are valid XML-Kenzo objects which can be processed.

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

```
<operation>
  <homology>
    <cyclic>7</cyclic>
    <dim>3</dim>
  </homology>
</operation>
```

This means that we can request the construction of cyclic groups and the computation of their homology groups in our system.

In addition, a result type element called `gap-id` which is used to return information about the properties of GAP groups has been defined. In particular this element has as value a natural number and has 9 attributes which represent respectively 9 properties (if the group is abelian, cyclic, elementary abelian, nilpotent, perfect, solvable, polycyclic, supersolvable and monomial) of the group by means of boolean values: `true` if the group satisfies the property and `false` otherwise (see Figure 4.3).

As we explained in Subsection 3.1.2, the external server evolves when the

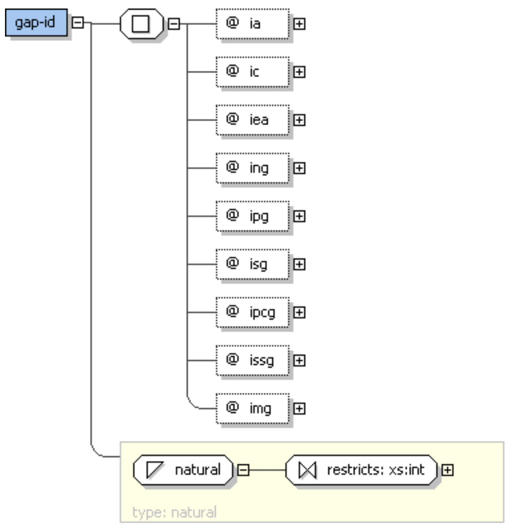


Figure 4.3: `gap-id` element in XML-Kenzo

`XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is upgraded with these new elements, the external server can validate requests such as the above ones.

4.1.3.2 A new internal server: GAP/HAP

Up to now, the Kenzo system was the unique computing kernel of the framework. Now, let us integrate the GAP/HAP system to construct groups and perform group homology computations.

GAP/HAP could be integrated locally as Kenzo was, but the installation of GAP/HAP is not so easy, and will need some interaction of the final user. Since, this option seems to us too uncomfortable, we devised the following organization based on the use of a GAP/HAP remote server.

4.1.3.2.1 A remote server: GAP/HAP + SCSCP Our framework is connected with a GAP/HAP server. The connection with this GAP/HAP server is available by means of SCSCP - the Symbolic Computation Software Composability Protocol [FHK$^+$09]. SCSCP is a remote procedure call framework for computational algebra systems in which both data and protocol instructions are encoded in the OpenMath language [Con04]. This protocol has been successfully used to communicate several Computer Algebra systems as can be seen in [F$^+$08]. GAP has a package, called SCSCP [KL09], which implements this protocol. This package has two main components: a server and a client; we are interested in the server part. The server component can be configured to supply the GAP procedures that can be invoked from different clients (which can be a GAP client with the SCSCP package, one of the SCSCP clients developed for Computer Algebra systems or in general a program with both *socket* support to invoke the GAP services and knowledge about the encoding of SCSCP OpenMath requests). In particular, our GAP/HAP server provides procedures to construct cyclic groups, to get some properties of the cyclic groups and to compute homology groups of cyclic groups.

The GAP/HAP server is available thanks to the SCSCP protocol without any additional development from our side; however, it has been necessary to deploy a client, that will be integrated in our framework. This client, from now on called *gap-invoker*, has been implemented as a Common Lisp program and has the functionality included in the `gap-invoker.lisp` file. This program has two parts: a Phrasebook and a *socket* client. The former component is able to transform from the XML-Kenzo representation to the SCSCP OpenMath one and viceversa. It is worth noting that the Phrasebook included in this component is not the same Phrasebook implemented previously in the adapter (see Subsection 2.2.5), since the OpenMath requests that are sent/received to/from the GAP/HAP server are wrapped with additional information about the location of the server and necessary information for the SCSCP package. An example of this kind of requests is as follows:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Homology"/>
                <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
                <OMI>5</OMI>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

This request has the following parts: the beginning of a SCSCP request (`<?scscp start ?>`), the location of the server (in this case located in the server `esus.unirioja.es` in the port `7500`):

```
<OMATP>
   <OMS cd="scscp1" name="call_id"/>
   <OMSTR>esus.unirioja.es:7500</OMSTR>
</OMATP>
```

the invocation of the GAP/HAP service by means of:

```
<OMS cd="scscp1" name="procedure_call"/>
<OMA>
  <OMS cd="scscp_transient_1" name="Homology"/>
```

In this case the procedure associated with `Homology` is the `GroupHomology` HAP command; the arguments of the procedure `Homology`, in this case the cyclic group and the dimension:

```
<OMA>
  <OMS cd="group1" name="cyclic"/>
  <OMI>5</OMI>
</OMA>
<OMI>5</OMI>
```

and, finally, the end of the SCSCP request (`<?scscp end ?>`).

The *socket* client component of the gap-invoker is a bunch of functions in charge of sending requests and receiving results to/from the GAP/HAP server by means of *sockets* technology.

4.1.3.3 Cyclic groups construction module of the microkernel

The `gap-cyclic-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the microkernel in order to include a new module, called `cyclic-groups`.

When the microkernel receives a construction request where the child of the `constructor` element is `cyclic`, the `cyclic-groups` module of the microkernel is activated. For instance, if the microkernel receives the request:

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

the `cyclic-groups` module is triggered.

When the `cyclic-groups` module is activated two situations are feasible: (1) a new object is created in the microkernel or (2) the object was previously built and its identification is simply returned. In the former case, this module constructs an object which represents a cyclic group. It is worth noting that no warnings are produced by the `cyclic-groups` module since all the restrictions (namely, this constructor only has associated the constraint of the type of its argument, which must be a natural number) about this constructor are handled in the XML-Kenzo specification, and, therefore, they are validated in the external server.

In Subsection 2.2.3, we have presented a representation for microkernel objects (`MK-OBJECT` class), which was specialized for Kenzo spaces (`MK-SPACE-KENZO` class). Now, we need a different specialization for cyclic groups constructed in the microkernel. Instead of defining a specialization just for cyclic groups we have decided to define a general representation for GAP groups, since in the future we could be interested in including support for the construction of other GAP groups. In the same line, in the future we could be interested in including not only groups, but also other GAP objects. As we said in Subsection 2.2.3 to include these new objects (which come from a different internal server) we specialized the `MK-OBJECT` class by means of a subclass, that is the class `MK-GAP`, whose definition is:

```
(DEFCLASS MK-GAP (MK-OBJECT) () )
```

In turn, we specialize this class to represent GAP groups.

```
(DEFCLASS MK-GROUP-GAP (MK-GAP)
     ;; Is Abelian
     (ia :type boolean :initarg :ia :reader ia)
     ;; Is Cyclic
     (ic :type boolean :initarg :ic :reader ic)
     ;; Is Elementary Abelian
     (iea :type boolean :initarg :iea :reader iea)
     ;; Is Nilpotent Group
     (ing :type boolean :initarg :ing :reader ing)
     ;; Is Perfect Group
     (ipg :type boolean :initarg :ipg :reader ipg)
     ;; Is Solvable Group
     (isg :type boolean :initarg :isg :reader isg)
     ;; Is PolyCyclic Group
     (ipcg :type boolean :initarg :ipcg :reader ipcg)
     ;; Is SuperSolvable Group
     (issg :type boolean :initarg :issg :reader issg)
     ;; Is Monomial Group
     (img :type boolean :initarg :img :reader img)
```

This class has nine slots in addition to the `idnm` and `orgn` slots of the `MK-OBJECT` class:

1. `ia`, a boolean, indicates if the group is abelian.
2. `ic`, a boolean, indicates if the group is cyclic.
3. `iea`, a boolean, indicates if the group is elementary abelian.
4. `ing`, a boolean, indicates if the group is nilpotent.
5. `ipg`, a boolean, indicates if the group is perfect.
6. `isg`, a boolean, indicates if the group is solvable.

7. `ipcg`, a boolean, indicates if the group is polycyclic.
8. `issg`, a boolean, indicates if the group is supersolvable.
9. `img`, a boolean, indicates if the group is monomial.

All the information included in the previous nine slots is obtained from GAP/HAP.

The procedure to construct cyclic group instances in the `cyclic-groups` module is very similar to the procedure followed in the construction of spaces presented in Subsubsection 2.2.3.3. However, in this case, this module does not need to check any additional restriction, since all the restrictions are imposed in the XML-Kenzo specification; therefore, all the requests that come from the external server related to cyclic group objects are always safe. The procedure to construct a cyclic group is as follows.

1. Search in the `*object-list*` list if the object was built previously.
   (a) If the object was built previously, return its identification number and the properties about the group in a `gap-id` XML-Kenzo object.
   (b) Otherwise, go to step 2.
2. Construct an instance of the `MK-GROUP-GAP` class where:
   - `idnm` is automatically generated (remember that `idnm` is the object identifier in the microkernel).
   - the XML-Kenzo object received as input is assigned to `orgn`.
   - the information of the rest of the slots is obtained invoking the GAP/HAP internal server.
3. Push the object in the `*object-list*` list of already created objects.
4. Return its identification number and the properties about the group in a `gap-id` XML-Kenzo object.

In this way, cyclic groups are constructed in the microkernel. It is worth noting that in spite of being constructed by different internal servers, all the objects are stored in the `*object-list*` list in the internal memory.

4.1.3.4 Enhancing the homology computation module

The `gap-homology-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the `homology` module of the microkernel in order to include the functionality to compute the homology groups of cyclic groups.

In Paragraph 2.2.3.4.1 the `homology` module of the microkernel was explained. The `homology` module implements a procedure in Common Lisp that allows us to compute homology groups through the microkernel. We have implemented that procedure in such a way that we do not need to overwrite the code of the procedure to allow the microkernel to deal with several internal servers.

Namely, we have used a very powerful tool of Common Lisp: the combination of generic functions and methods [Gra96]. When the class system and the functional organization of Common Lisp are considered, the notions of *generic functions* and *methods* are normally used. A *generic function* is a functional object whose behavior will depend on the class of its arguments; a generic function is defined by a `defgeneric` statement. The code for a generic function corresponding to a particular class of its arguments is a *method* object; each method is defined by a `defmethod` statement. This technique was used in the implementation of the class system of Kenzo [Ser01] and also in the homology computation module of the microkernel.

In particular, we have defined a generic function which corresponds with Step 5 of the procedure explained in Paragraph 2.2.3.4.1:

```
(DEFGENERIC compute-homology (mk-object n))
```

This generic function can have several methods to adapt the generic function to specific cases. In particular, up to now, it had associated just the method:

```
(DEFMETHOD compute-homology ((space mk-space-kenzo) n) ...)
```

This method invokes the Kenzo internal server from an `mk-space-kenzo`.

Now, we have defined a new class of objects; and in this case the system does not use Kenzo as kernel to compute the homology groups, but the GAP/HAP server. Then, the `gap-homology-m.lisp` file defines the method:

```
(DEFMETHOD compute-homology ((group mk-group-gap) n) ...)
```

which allows us to compute homology groups using the GAP/HAP server through the gap-invoker.

It is also worth noting that the instances of the class `mk-group-gap` do not have a `rede` slot which is used in the procedure explained in Paragraph 2.2.3.4.1 to check whether Kenzo could compute the homology groups or not. The way of managing this situation is based on the same idea: use generic functions and methods. Namely, we have defined the generic function which corresponds with the Step 4 of the procedure explained in Paragraph 2.2.3.4.1:

```
(DEFGENERIC check-constraints (mk-object))
```

In the case of dealing with `mk-space-kenzo` instances we have the method:

```
(DEFMETHOD check-constraints ((space mk-space-kenzo))
  (if (>= (rede space) 0) t nil))
```

On the contrary, the method implemented for the case of `mk-group-gap` instances is the following simple method:

```
(DEFMETHOD check-constraints ((group mk-group-gap)) t)
```

In this way, the homology module of the microkernel allows us to use GAP/HAP to perform computations. To sum up, homology groups of spaces are computed with Kenzo and group homology is computed with GAP. In general, we can use different methods to compute the homology groups of different classes of objects without modifying the main procedure.

4.1.3.5 Increasing the functionality of the adapter

Cyclic groups are objects already defined in the OpenMath language, namely in the `groupname1` Content Dictionary. This Content Dictionary is added to the list of Content Dictionaries which can be processed in the adapter. Moreover, we have defined a `gap-id` object, in the `Aux` Content Dictionary, to return the identification and the information related to a group.

In addition, we have extend the Phrasebook by means of new parsers which are able to convert from XML-Kenzo objects related to GAP/HAP to OpenMath objects and viceversa. For instance, the XML-Kenzo request:

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
     <OMS cd="groupname1" name="cyclic"/>
     <OMI>5</OMI>
   </OMA>
</OMOBJ>
```

Therefore, the `gap-a.lisp` file contains the new parsers and a list with both `groupname1` and `Auxiliar` Content Dictionaries in order to raise the functionality of

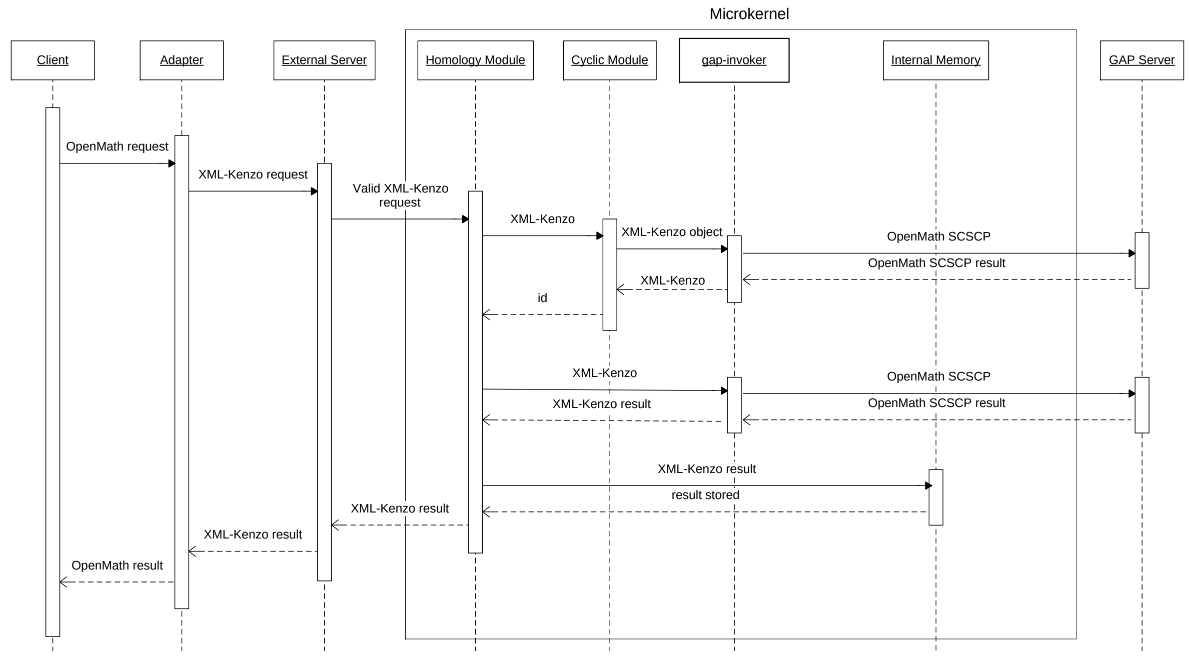


Figure 4.4: Execution flow of a GAP/HAP computation

the adapter to be able to convert from the new OpenMath requests, devoted to GAP operations, to XML-Kenzo requests.

4.1.3.6 Execution flow

To provide a better understanding of the integration of the GAP/HAP system in our framework, let us present an execution scenario where a client asks our framework for computing $H_5(C_5)$ in a fresh session, that is, neither objects were constructed or computations were previously performed. The execution flow of this scenario is depicted in Figure 4.4 with a UML-like sequence diagram.

The OpenMath representation of the request $H_5(C_5)$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homology"/>
        <OMA>
            <OMS cd="groupname1" name="cyclic"/>
            <OMI>5</OMI>
        </OMA>
        <OMI>5</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath request from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
    <homology>
        <cyclic>5</cyclic>
        <dim>5</dim>
    </homology>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification, in this case as the root element is `operation`, then it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homology` element has two children. ✓
3. The first child of the `homology` element belongs to one of the groups `A`, `CC`, `SS`, `SG` or `ASG`. ✓
4. The value of the `cyclic` element is a natural number. ✓
5. The second child of the `homology` element is the `dim` element. ✓
6. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, we have a valid request that is sent to the microkernel. In the microkernel the `homology` module is triggered. When the `homology` module is activated, the procedure explained in Paragraph 2.2.3.4.1 with the extension explained in Subsubsection 4.1.3.4 is executed. First, the `homology` module searches in `*object-list*` list if the cyclic group of dimension 5 was constructed previously; as this object was not constructed, the `cyclic` module is invoked to construct it, this module in turn invokes the GAP/HAP server through the gap-invoker to construct a `mk-group-gap` instance. This instance is stored in the `*object-list*` list to avoid the duplication of elements.

Subsequently, once the object is constructed, the `homology` module sends the XML-Kenzo request to the gap-invoker in order to send a request to the GAP/HAP server to compute the group homology. The request sent to the GAP/HAP server by the gap-invoker is:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Homology"/>
                <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
                <OMI>5</OMI>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

It is worth noting that the above OpenMath request is a bit different from the original one received by the adapter. Namely, it includes SCSCP information.

When the GAP/HAP server receives the above request, it executes the following instruction:

```
gap> GroupHomology(CyclicGroup(5),5); ✠
[5]
```

The result returned by the GAP/HAP server is:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name ="call_id"/>
            <OMSTR> esus.unirioja.es:7500 </OMSTR>
        </OMATP>
        <OMA><OMS cd="scscp1" name ="procedure_completed"/>
            <OMA><OMS cd="list1" name ="list"/><OMI>5</OMI></OMA>
        </OMA>
        </OMATTR>
</OMOBJ>
<?scscp end ?>
```

This result is converted by the gap-invoker into the following XML-Kenzo result.

```
<result>
    <component>5</component>
</result>
```

This result is stored in the internal memory to avoid re-computations by the `homology` module and is sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="ringname" name="Zm"/>
        <OMI>5</OMI>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client. It is worth noting that the OpenMath representation of the result returned by the GAP/HAP server is different from the representation of the result returned by the adapter. This is due to the fact that the GAP/HAP server uses the SCSCP representation; but we consider that is better to keep a consistent representation for all the results returned by our framework for the computation of homology groups.

4.1.4 Properties of objects

As we have explained in the previous subsection, when a (cyclic) group is constructed in our framework not only its identification number is returned but also some properties of that group. This additional information can be helpful, in a client, for instance to allow a student to know some properties of the object.

Then, we realized that in the same way that some properties are associated with groups, we can do the same with spaces. However, there is an important difference, group properties are obtained from the GAP/HAP server, whereas the space properties will be obtained from the knowledge included in the microkernel and not from Kenzo. To include, this improvement in our system, the following small plug-in has been developed.

```
<code id="spaces-properties">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/microkernel"> properties-mk.lisp </data>
   <data format="Kf/adapter"> properties-a.lisp </data>
</code>
```

These resources deserve a detailed explanation that is provided in the following paragraphs.

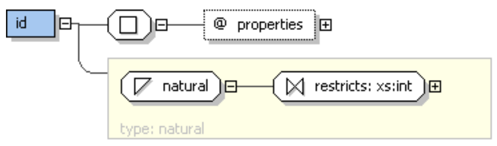


Figure 4.5: `id` element in XML-Kenzo

First of all, we want to modify the `id` XML-Kenzo object to not only store an identification number but also properties about the object which has associated the identifier. Therefore, we have extended the XML-Kenzo specification (XML-Kenzo.xsd file) to admit the new definition of `id` which includes an attribute called `properties`, see Figure 4.5.

The `properties-mk.lisp` file modifies the behavior of the following construction modules: `Sphere`, `Delta`, `K-Z`, `K-Z2`, `Cartesian Product`, `Suspension`, `Classifying Space` and `Loop Space`. That is to say, the modules which are used by the HES, because they provide additional information which can be interesting for a user. Then, the `id` XML-Kenzo objects returned by these modules not only contain the identifiers but also some properties. For instance, when the `Cartesian Product` module is activated and both components of the Cartesian product are contractible spaces, the returned object is:

```
<id properties="The space is contractible because is the cartesian product of two
           contractible spaces"> 1 </id>
```

It is worth noting that the rest of the modules of the microkernel return `id` XML-Kenzo objects whose `properties` attribute is empty.

Finally, the `properties-a.lisp` file includes a new parser in the Phrasebook to be able to handle the new specification of `id` XML-Kenzo objects.

4.1.5 Integration of GAP/HAP in the *fKenzo* GUI

Throughout this section, a plug-in that allows us to include in our system the functionality related to GAP/HAP has been presented. Now, the necessary resources to extend the *fKenzo* GUI to provide support for the new functionality are explained.

In this case we have defined a fresh *fKenzo* module to enhance the GUI with support for the GAP/HAP system. The new OMDoc module references three files: `gap-structure` (that defines the structure of the graphical constituents), `gap-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection.

We have defined three graphical elements, using the XUL specification language, in the `gap-structure` file:

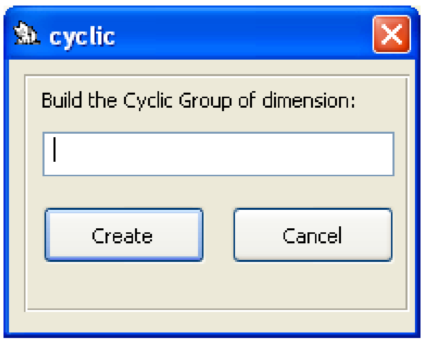


Figure 4.6: Cyclic window

- A menu called `GAP` which contains one option: `Cyclic Group`.
- A window called `Cyclic` (see Figure 4.6). It is worth noting that it is not necessary to specify this window from scratch since it has the same structure that, for instance, the `Sphere` window of the Simplicial Set module. So, we use a generic specification with the concrete values of the new window. In general, when we define a new window, we do it by means of an instance of a generic specification.

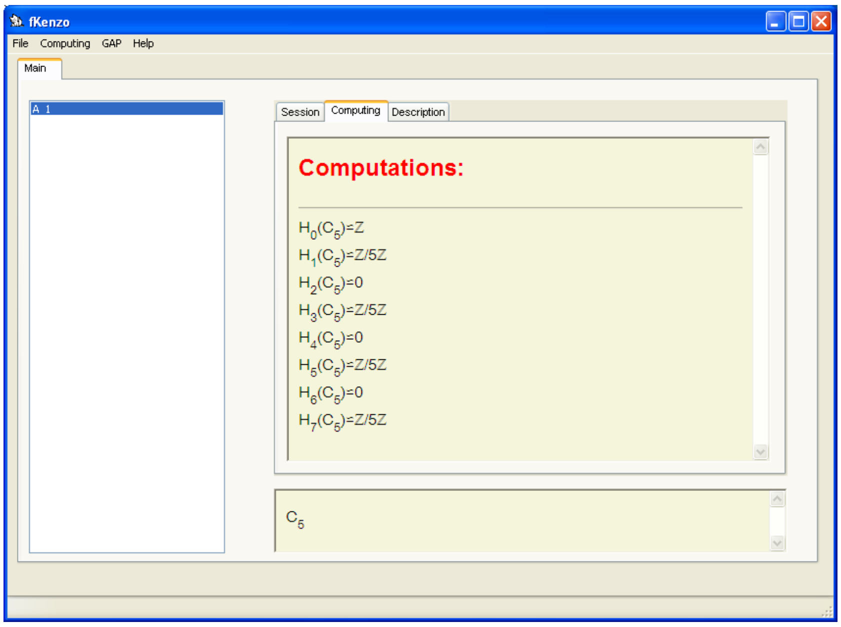
The functionality stored in the `gap-functionality` document related to these components works as follows. A function acting as event handler is associated with the *Cyclic Group* menu option; this function shows the *Cyclic* window (see Figure 4.6). From the *Cyclic* window, the user must introduce the dimension of the cyclic group that must be a natural number. Once the user has introduced the dimension n of the cyclic group, when he presses the `Create` button of the *Cyclic* window, an OpenMath request is generated and sent to the framework.

Then, the cyclic group C_n is built and its identification number in a `gap-id` OpenMath object is returned. Eventually *fKenzo* adds to the constructed objects list (situated in the left side of the main tab of the *fKenzo* GUI) the new group. To identify the (Abelian) groups in *fKenzo* we use the letter A and its identification number.

Finally, if the `Computing` *fKenzo* module is loaded (or has been previously loaded) the user can ask *fKenzo* to compute the homology groups of a group using the *Homology* option of the *Computing* menu, the results are shown, as usual, in the Computing tab. Figure 4.7 shows the computation of the homology groups of C_5.

It is worth noting that from the user point of view the computation of homology groups of both spaces and groups has no differences, since he proceeds in both cases in the same way. Therefore the system used in each moment is transparent to the user, providing access to different software systems in an easy and comfortable way.

In addition to the graphical elements presented about GAP/HAP, a new constituent has been added to the GUI to handle *properties*. Namely, a new tab called *Description* is included in the central panel. This new graphical element is included whenever a



Computations:

$H_0(C_5)$=Z
$H_1(C_5)$=Z/5Z
$H_2(C_5)$=0
$H_3(C_5)$=Z/5Z
$H_4(C_5)$=0
$H_5(C_5)$=Z/5Z
$H_6(C_5)$=0
$H_7(C_5)$=Z/5Z

C_5

Figure 4.7: Homology groups of C_5

construction module or the GAP/HAP module is loaded in *fKenzo*. The functionality associated with this new element is as follows.

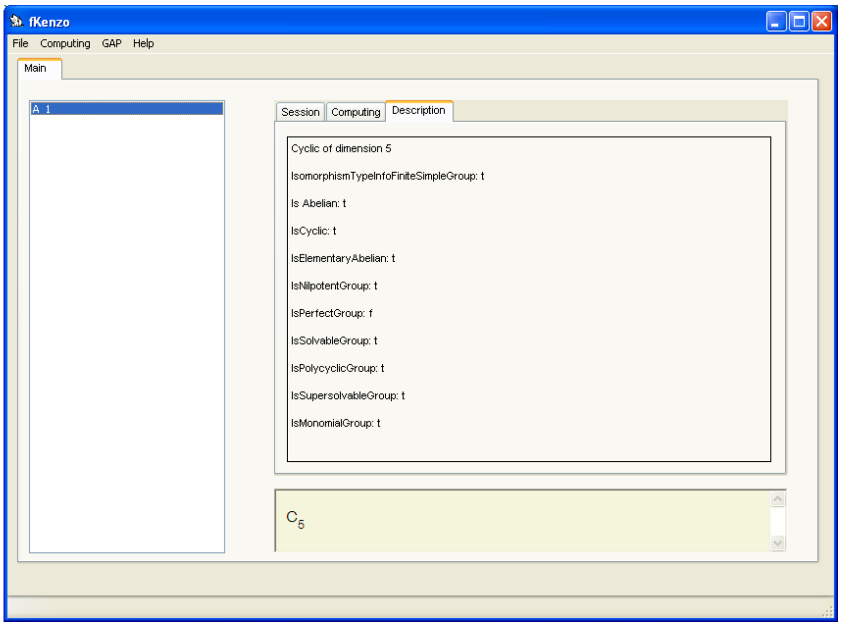
As we explained in Subsection 3.2.1 when an object is selected from the list of constructed spaces, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. This behavior is kept but, in addition, if some additional information is accessible when an object is constructed, then, this information is shown in the Description tab. For instance, when the cyclic group of dimension 5 is selected, the Description tab shows the properties about the group included in the `gap-id` OpenMath object returned, see Figure 4.8. The same happens when an `id` OpenMath object contains some properties of the object.

The next subsubsection is devoted to present how objects are handled in the *fKenzo* GUI.

4.1.5.1 Management of the objects in the *fKenzo* GUI

Let us present how the objects are managed in the *fKenzo* GUI.

The representation of objects in the *fKenzo* GUI has been inspired by the representation of objects in Kenzo. An object of the *fKenzo* GUI is implemented as an instance of a CLOS class (let us remember that the *fKenzo* GUI is implemented in Common Lisp), the class `FKENZO-OBJECT`, whose definition is:



```
Cyclic of dimension 5
IsomorphismTypeInfoFiniteSimpleGroup: t
Is Abelian: t
IsCyclic: t
IsElementaryAbelian: t
IsNilpotentGroup: t
IsPerfectGroup: f
IsSolvableGroup: t
IsPolycyclicGroup: t
IsSupersolvableGroup: t
IsMonomialGroup: t
```

C_5

Figure 4.8: Description tab for GAP groups

```
(DEFCLASS FKENZO-OBJECT ()
     ;; IDentification NuMber
     (idnm :type fixnum :initarg :idnm :reader idnm)
     ;; ORiGiN
     (orgn :type string :initarg :orgn :reader orgn))
     ;; PROPertieS
     (props :type string :initarg :props :reader props)))
```

This class has three slots:

1. `idnm`, an integer, identifier for the object in *fKenzo*. This is the value assigned when a space is constructed.
2. `orgn`, a string containing the OpenMath object that is the *origin* of the object.
3. `props`, a string containing the properties obtained in the construction of the object or the empty string if no property was returned.

The objects are stored in a list called `*FKENZO-OBJECTS*` which is used to generate the shown list to the user in the left side of the *fKenzo* GUI.

Moreover, there is a function, called `fK`, which allows us to get information about the n-th *fKenzo* GUI object. The `fK` function takes as argument a natural number n and returns the n-th *fKenzo* GUI object.

The functionality of the event handler of the list of the left side, when an object is selected from it, works as follows. From the selected object, for instance “SS 3”, extracts its identification number, in this case 3, and obtains the `FKENZO-OBJECT` by means of the `fK` function. Subsequently, it invokes a Common Lisp method [Gra96] associated with that `FKENZO-OBJECT` which shows the mathematical notation of the object in the bottom part of the right side of the *fKenzo* GUI. Moreover, if the value of the `props` slot is not empty, then, the system moves dynamically from the current selected tab to the Description tab showing the properties of the object.

It is worth noting that the approach followed to store the properties of an object, just using a string, is likely too simple, and several improvements can be consider, for instance, replace the `props` string with an association with different classes. However, that remains as further work.

Moreover, due to the fact that objects of very different nature can exists in *fKenzo*, we have specialized the `FKENZO-OBJECT` with two different subclasses. On the one hand, we have defined a new subclass of the `FKENZO-OBJECT` class; to store a name given by the user to identify the object. Namely, the new class, `FKENZO-OBJECT-NAME`, whose definition is:

```
(DEFCLASS FKENZO-OBJECT-NAME (FKENZO-OBJECT)
     ;; NAME
     (name :type string :initarg :name :reader name)
```

This is a subclass of the `FKENZO-OBJECT` class with just one additional slot, `name`, which provides a name for the object. The `FKENZO-OBJECT-NAME` instances will be useful to store objects which do not have a concrete mathematical representation but which include a name to identify them in *fKenzo*, for instance a simplicial set build from a list of its elements.

On the other hand, we have defined a new subclass of the `FKENZO-OBJECT` class; to store the path of a file associated with the object. Namely, the new class, `FKENZO-OBJECT-FILE`, whose definition is:

```
(DEFCLASS FKENZO-OBJECT-FILE (FKENZO-OBJECT)
     ;; FILE
     (file :type string :initarg :file :reader idnm)
```

This is a subclass of the `FKENZO-OBJECT` class with just one additional slot, `file`, which provides the path of a file associated with the object. In turn this class can be specialized depending on the type of the file which is associated with the object.

4.1.5.2 Behavior of the objects in the *fKenzo* GUI

It is worth noting that in spite of belonging to different classes, all the instances constructed in *fKenzo* are stored in the `*FKENZO-OBJECTS*` list, which is used to show the list objects in the left side of the *fKenzo* GUI. However, when we select an object in the left list of the GUI, the behavior of the interface depends on the class associated with the selected object. To deal with this question we have used the *Strategy* pattern [GJ94] which is implemented in Common Lisp by means of generic functions and methods.

In particular, we have defined the generic function to show information in the *fKenzo* GUI about the selected object:

```
(DEFGENERIC show-object (object))
```

This generic function have several methods to adapt it to specific cases. The main method is associated with the `FKENZO-OBJECT` class:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT))
   (show-mathematical-notation (orgn object))
   (if (props object) (show-description (props object))))
```

This method shows the mathematical representation of `object` in the bottom part of the right side of the *fKenzo* GUI by means of the `show-mathematical-notation` function which takes as argument the `orgn` slot of the `FKENZO-OBJECT` instance. Moreover, if some additional information is stored in the `props` slot of the object, then, this information is shown in the Description tab.

When the object associated with the selection of the left list of the GUI belongs to the class `FKENZO-OBJECT-NAME`, the behavior of the GUI is different since we have defined this new method:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-NAME))
   (show-name (name object))
   (if (props object) (show-description (props object))))
```

which instead of showing the mathematical representation of the object shows the name given by the user. In addition, each specialization of the `FKENZO-OBJECT-FILE` will have a concrete method to specify its behavior. In this way, the behavior of the left list of the GUI is modified without touching the main code.

We foresee the definition of new specializations in the future, but we think that approach presented here is general enough to be extrapolated to other cases.

4.2 Interoperability between Kenzo and GAP/HAP

As we said at the beginning of this chapter, the integration of several tools in *fKenzo* was a means and not the end. The final goal of integrating different tools, as internal servers, in the framework (and in *fKenzo*, too) consists in achieving a composability between them to produce results not reachable if the tools worked in an isolated way.

The first case study is the integration of the two Computer Algebra systems included in our system (Kenzo and GAP/HAP). To achieve the goal of the composability of Kenzo and GAP/HAP we have been inspired by the work presented in [RER09]. In that work, Kenzo and GAP/HAP were manually communicated by means of OpenMath objects in order to compute homology of groups of Eilenberg MacLane spaces of type $K(\pi, 1)$. In addition, this integration allowed the authors to develop new tools to compute more algebraic invariants, such as homology groups of $K(\pi, n)$'s, certain 2-types and central extensions, as can be seen in [Rom10].

However, the approach followed in that paper to connect Kenzo and GAP/HAP had some drawbacks that will be explained in the next subsection. Then, we have undertaken the task of improving the cooperation between these systems thanks to our framework.

The interoperability between Kenzo and GAP/HAP is translated into the feasibility of constructing Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, thanks to the combination of Kenzo and GAP/HAP; and subsequently use these spaces as any other space of the system (that means, use them for constructing other spaces and computing their homology and homotopy groups).

The rest of this section is organized as follows. Subsection 4.2.1 is devoted to provide an overview of the method to integrate Kenzo and GAP/HAP used in [RER09]; in addition, the necessary mathematical background is also explained there. The composability of Kenzo and GAP/HAP in our framework is explained in Subsection 4.2.2. Finally, the improvements added to the *fKenzo* GUI are presented in Subsection 4.2.3.

4.2.1 Mathematical preliminaries

In Subsection 4.1.1, we explained that is enough to determine a free resolution F_* of a group G to compute its homology groups. One approach consists in considering the bar resolution $B_* = Bar_*(G)$ (explained, for instance, in [Mac63] and which can be always constructed) whose associated chain complex $\mathbb{Z} \otimes_{\mathbb{Z}G} B_*$ can be viewed as the chain complex of the Eilenberg MacLane space $K(G, 1)$. The homology groups of $K(G, 1)$ are those of the group G and this space has a big structural richness. But it has a serious drawback: its size. If $n > 1$, then $K(G, 1)_n = G^n$. In particular, if $G = \mathbb{Z}$, the space $K(G, 1)$ is infinite. This fact is an important obstacle to use $K(G, 1)$ as a means for computing the homology groups of G.

However, the Effective Homology technique (see Subsection 1.1.3) and the Kenzo program could have a role in the computation of the homology of a group G, since, as we have seen in Section 1.2.2, Kenzo implements Eilenberg MacLane spaces $K(G, n)$ for every n but only for $G = \mathbb{Z}$ and $G = \mathbb{Z}/2\mathbb{Z}$. To our end, we need the Eilenberg MacLane space $K(G, 1)$, for other groups G. The size of this space makes it difficult to calculate the groups in a direct way, but it is possible to operate with this simplicial set making use of the effective homology technique: if we construct the effective homology of $K(G,1)$ then we would be able to compute the homology groups of $K(G,1)$, which are those of G. Furthermore, it should be possible to extend many group theoretic constructions to effective homology constructions of Eilenberg MacLane spaces. We thus introduce the following definition.

Definition 4.4. A group G is a group with effective homology if $K(G,1)$ is a simplicial set with effective homology.

The problem is, given a group G, how can we determine the effective homology of $K(G,1)$? If the group G is finite, the simplicial set $K(G,1)$ is effective too, so that it has trivially effective homology. However, the enormous size of this space makes it difficult to obtain real calculations, and therefore we will try to obtain an equivalence $C_*(K(G,1)) \lll\!\Rightarrow\!\ggg E_*$ where E_* is an effective and (much) smaller chain complex than the initial chain complex.

In [RER09] the algorithm that computes this equivalence from a resolution of G was explained. Here, we just state the algorithm.

Algorithm 4.5 ([RER09])**.**
Input: a group G and a free resolution F_* of finite type.
Output: the effective homology of $K(G,1)$, that is, an equivalence $C_*(K(G,1)) \lll\!\Rightarrow\!\ggg E_*$ where E_* is an effective chain complex.

This algorithm was implemented, by the authors of [RER09], in Common Lisp enhancing the Kenzo system. The free resolution of the group G is obtained from the GAP/HAP system. Particulary to the case where G is a cyclic group, the process to construct the space $K(G,1)$, in Kenzo, can be summed up as follows:

1. Load the necessary packages and files in GAP and Kenzo,
2. build the cyclic group G in GAP,
3. build a resolution of the cyclic group G using the HAP package,
4. export from GAP the resolution into a file using the OpenMath format,
5. import the resolution to Kenzo,
6. build the cyclic group G in Kenzo (thanks to a new Kenzo module developed in [RER09]),
7. assign the resolution to the corresponding cyclic group G in Kenzo,
8. build the space $K(G,1)$ where G is the cyclic group in Kenzo.

This approach has some drawbacks. First of all, the user must install several programs and packages: GAP, its HAP package, the OpenMath package for GAP [SC09], an extension for this OpenMath package developed in [RER09], the Kenzo system and the new module developed in [RER09]. In addition, of course, the user must know how to mix all the ingredients in order to obtain the desired result. Moreover, some of the steps could be performed automatically by a computer program; for instance, the importation/exportation of the resolution from GAP to Kenzo.

Therefore, we undertook the task of integrating this composability of systems in *fKenzo* but overcoming the drawbacks and hiding to the final user the composability details.

4.2.2 Composability of Kenzo and GAP/HAP

We have modified the plug-in used to integrate GAP in our framework to include the functionality related to the construction of Eilenberg MacLane spaces of type $K(G,1)$, where G is a cyclic group. The resources included in that plug-in are:

```
<code id="gap">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> gap-kenzo.lisp </data>
   <data format="Kf/internal-server"> gap-invoker.lisp </data>
   <data format="Kf/microkernel"> gap-cyclic-m.lisp </data>
   <data format="Kf/microkernel"> gap-homology-m.lisp </data>
   <data format="Kf/microkernel"> gap-k-g-1-m.lisp </data>
   <data format="Kf/microkernel"> homotopy-extension-m.lisp </data>
   <data format="Kf/adapter"> gap-a.lisp </data>
</code>
```

It is worth noting that some new resources have been included to the original GAP plug-in (in particular the files `gap-kenzo.lisp`, `gap-k-g-1-m.lisp` and `homotopy-extension-m.lisp`), others have been modified to allow the integration of the new tools (`XML-Kenzo.xsd`, `gap-invoker.lisp` and `gap-a.lisp`) and other files remain untouched.

The modification of the resources and the new resources are going to be explained in the following subsubsections.

4.2.2.1 New elements of the XML-Kenzo schema

We want to introduce a new kind of objects in our system (Eilenberg MacLane spaces of type $K(G,1)$ where G is a group); then, it is necessary to provide a representation for those objects in our framework. Therefore, we have extended the XML-Kenzo specification to admit the new objects related to Eilenberg MacLane spaces of type $K(G,1)$, where G is a group. In the specification, we have defined a new element: `k-g-1`, which

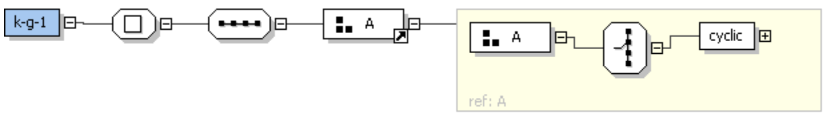


Figure 4.9: `k-g-1` element in XML-Kenzo

has one child, that is, an element, of the group `A` (the type of the GAP abelian groups in our specification, at this moment this group just has the `cyclic` element but we foresee the inclusion of new groups), see Figure 4.9. In addition, the element `k-g-1` belongs to the `ASG` group (the type of Abelian Simplicial Groups); and therefore, this element can be used as any other element of the `ASG` group, that is to say, we can construct requests to compute its homology and homotopy groups and use it for constructing other spaces.

Therefore, the following request is a valid XML-Kenzo object.

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

As we explained in Subsection 3.1.2, the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified the external server can validate requests such as the above one.

4.2.2.2 Enhancements of Kenzo internal server and GAP/HAP server

The Kenzo internal server, the GAP/HAP server and the gap-invoker have been modified in order to integrate the composability of Kenzo and GAP/HAP.

The GAP/HAP server presented in Subsubsection 4.1.3.2.1 supplied services to construct cyclic groups, to obtain properties of them and to compute their homology groups using the GAP Computer Algebra system and its HAP package. Now, we have included a new service which allows the construction of a resolution of a (cyclic) group using the HAP package. Then, a client can invoke this new service. For instance, when the GAP/HAP server receives the following request:

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
              <OMS cd="scscp_transient_1" name="Resolution"/>
              <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

which asks the construction of a resolution for the cyclic group of dimension 5, the GAP server executes the instruction:

```
gap> ResolutionFiniteGroup(CyclicGroup(5)); ✠
Resolution in characteristic 0 for <pc group of size 5 with 1 generators>
```

Subsequently, the GAP server transforms the result to its OpenMath format.

```
<OMOBJ>
 <OMA>
  <OMS cd="resolutions" name="resolution"/>
  <!-- GROUP -->
  <OMA>
   <OMS cd="group1" name="cyclic_group"/>
   <OMI>5</OMI>
  </OMA>
  <!-- More than 1000 lines skipped -->
  ...
</OMOBJ>
```

The OpenMath format of resolutions was explained in [RER09].

The gap-invoker (presented in Subsubsection 4.1.3.2.1), which was able to invoke the GAP/HAP server in order to construct cyclic groups, obtain their properties and compute their homology groups has been upgraded in the `gap-invoker.lisp` file in order to be able to request resolutions to the GAP/HAP server. When this program invokes the GAP/HAP server asking for a resolution of a group, the result returned by this program is the resolution obtained from the GAP/HAP server keeping its OpenMath format. We decided to keep resolutions with their OpenMath format instead of transforming them to an XML-Kenzo representation. The main reason was due to the fact that we have re-used the programs implemented in [RER09] which work with resolutions based on the OpenMath format. Therefore, we considered that converting from an OpenMath resolution to an XML-Kenzo resolution and subsequently performing the inverse transformation to obtain the original OpenMath resolution was unnecessary. Besides, the idea of using XML-Kenzo to check the correctness of the resolutions against some rules (as we have done with the rest of the objects of the system) was rejected because we completely trust in the resolutions returned by the gap-invoker. This means (if connection problems with the GAP/HAP server do not appear in which case the system manages the situation) that the returned resolutions are always safe since they are not manually produced, but automatically generated by a program following the rules which ensure their correctness.

Finally, the functionality of the Kenzo kernel is increased by means of the `gap-kenzo.lisp` file which loads in the Kenzo system the functionality implemented in [RER09] to construct Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group. The functionality implemented in [RER09] includes the functions to construct cyclic groups, to transform an OpenMath resolution into a Common Lisp functional object and, eventually, to construct Eilenberg MacLane spaces of type $K(G, 1)$ in the Kenzo system.

Besides, the functionality of the Kenzo internal server is increased allowing the invocation of the service which allows the construction of new Eilenberg MacLane spaces. Namely, this service takes as argument an XML-Kenzo object which represents the Eilenberg MacLane space of a group G and an OpenMath resolution of G obtained from GAP. When this new service is activated, it extracts the cyclic group encoded in the XML-Kenzo object and constructs a Kenzo object which represents that cyclic group. Subsequently, the OpenMath resolution is codified as a Common Lisp functional object and assigned to the cyclic group. Afterwards, the space $K(G, 1)$, where G is the cyclic group previously constructed in Kenzo, is built. As a result, an instance of the `Abelian-Simplicial-Group` Kenzo class is obtained and its identification number is returned as a result, in an `id` XML-Kenzo object, by the Kenzo internal server.

4.2.2.3 Increasing the functionality of the microkernel

The `gap-k-g-1-m.lisp` file contains the Common Lisp functions which allow the plug-in framework to extend the microkernel in order to include a new construction module, called `k-g-1`, to construct Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group.

When the microkernel receives a construction request where the child of the `constructor` element is `k-g-1`, the `k-g-1` module of the microkernel is activated. For instance, if the microkernel receives the request:

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

the `k-g-1` module is activated.

When the `k-g-1` module is activated two situations are feasible: (1) a new space is created in the microkernel or (2) the object was previously built and its identification number is simply returned.

The procedure to construct Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, in the `k-g-1` module is very similar to the procedure followed in the construction of spaces presented in Subsubsection 2.2.3.3. However, in this case, some additional steps are needed.

1. Search in the `*object-list*` list if the object was built previously.
   (a) If the object was built previously its identification number, in an `id` XML-Kenzo object, is returned.
   (b) Otherwise, go to step 2.
2. Extract the XML-Kenzo object which represents the cyclic group of the `k-g-1` XML-Kenzo object.
3. Activate the gap-invoker to obtain a resolution associated with the XML-Kenzo object which represents the cyclic group obtained in the previous step.
4. Invoke the service of the Kenzo internal server which allow the construction of Eilenberg MacLane space of a group G with the `k-g-1` XML-Kenzo object and the resolution obtained in the previous step as arguments.
5. Construct an instance of the `mk-space-k-g` class (see Subsubsection 2.2.3.1) where:
   - the value of the slot `rede` is 0,
   - `idnm` is automatically generated,
   - `kidnm` is the value returned by the Kenzo internal server in the previous step,
   - the XML-Kenzo object received as input is assigned to `orgn`,
   - the value of the slot `iter` is 1, and
   - the value of the slot `group` is the dimension of the cyclic group.
6. Push the object in the `*object-list*` list of already created spaces.
7. Return the `idnm` of the object in an `id` XML-Kenzo object.

In this way, Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, are constructed in the microkernel. It is worth noting that the above procedure does not need to check any extra constraint for these spaces, since the restriction (namely, the constraint is the correct type of the argument of this constructor) about these Eilenberg MacLane spaces is validated in the external server thanks to the XML-Kenzo specification.

As we have explained previously, Eilenberg MacLane spaces play a key role in the computation of homotopy groups of spaces in our system (see Paragraph 2.2.3.2.2). Then, the `homotopy-extension-m.lisp` file extends the procedure implemented in the HAM (see Paragraph 2.2.3.2.2). The procedure implemented in that module only allowed the computation of homotopy groups if the first non null homology group of the space was $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$; using the new functions implemented in [RER09] and included in the Kenzo internal server, we can compute homotopy groups of spaces whose first non null homology group is a cyclic group using the algorithm implemented in the HAM and the new functionality related to Eilenberg MacLane spaces.

4.2.2.4 Increasing the functionality of the adapter

Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, have been defined in the `ASG` OpenMath Content Dictionary. Then, we have upgraded the functionality of `gap-a.lisp` (which was presented in Subsubsection 4.1.3.5) to raise the functionality of the adapter in order to be able to convert from the new OpenMath requests, devoted to the construction of Eilenberg MacLane spaces of type $K(G, 1)$, where G is a cyclic group, to XML-Kenzo requests. Namely, we have extend the Phrasebook by means of a new parser in charge of that task, then, the following XML-Kenzo request:

```
<constructor>
  <k-g-1>
    <cyclic>5</cyclic>
  </k-g-1>
</constructor>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
     <OMS cd="ASG" name="k-g-1"/>
       <OMA>
         <OMS cd="groupname1" name="cyclic"/>
         <OMI>5</OMI>
       </OMA>
   </OMA>
</OMOBJ>
```

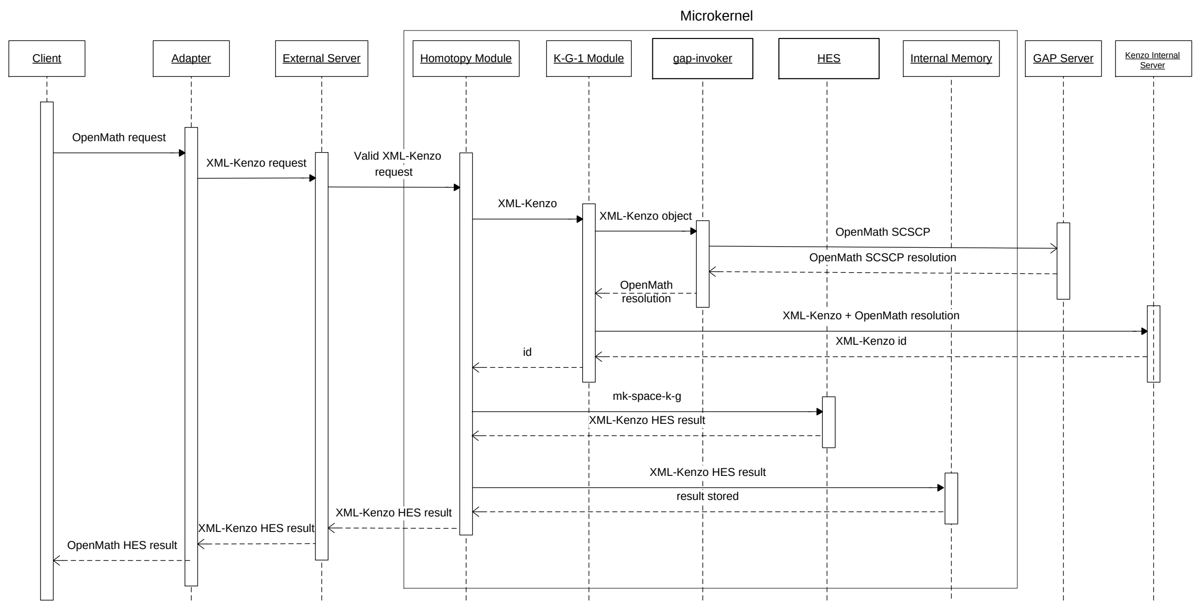


Figure 4.10: Workflow of the computation of $\pi_1(K(C_5,1))$

4.2.2.5 Execution flow

To provide a better understanding of the composability of Kenzo and GAP/HAP, let us present an execution scenario where a client wants to compute $\pi_1(K(C_5,1))$ in a fresh session, that is to say, neither objects were constructed nor computations were performed previously. The execution flow of this scenario is depicted in Figure 4.10 with a UML-like sequence diagram.

The OpenMath representation of the request to compute $\pi_1(K(C_5,1))$ is the following one:

```
<OMOBJ>
    <OMA>
        <OMS cd="Computing" name="Homotopy"/>
        <OMA>
           <OMS cd="ASG" name="k-g-1"/>
           <OMA> <OMS cd="groupname1" name="cyclic"/> <OMI>5</OMI> </OMA>
        </OMA>
        <OMI>1</OMI>
    </OMA>
</OMOBJ>
```

The adapter receives the previous OpenMath request from a client. This module checks that the OpenMath instruction is well-formed and the Phrasebook converts the OpenMath object into the following XML-Kenzo object:

```
<operation>
    <homotopy>
       <k-g-1> <cyclic>5</cyclic> </k-g-1>
       <dim>1</dim>
    </homotopy>
</operation>
```

which is sent to the external server. The external server validates the XML-Kenzo object against the XML-Kenzo specification. In this case as the root element is `operation` it checks that:

1. The child element of `operation` is `homology` or `homotopy`. ✓
2. The `homotopy` element has two children. ✓
3. The first child of the `homotopy` element belongs to one of the groups `CC`, `SS`, `SG` or `ASG`. ✓
4. The `k-g-1` element has one child which belongs to the group `A`. ✓
5. The value of the `cyclic` element of the `k-g-1` element is a natural number. ✓
6. The second child of the `homotopy` element is the `dim` element. ✓
7. The value of the `dim` element is a natural number. ✓

All the tests are passed, so, we have a valid request that is sent to the microkernel. In the microkernel the `homotopy` module is activated. When the `homotopy` module is activated, the procedure explained in Paragraph 2.2.3.4.2 is executed. First, the `homotopy` module searches in `*object-list*` if the space $K(C_5, 1)$ was constructed previously; as this space was not constructed, the `k-g-1` module is invoked to construct it. When the `k-g-1` module is activated, the procedure explained in Subsubsection 4.2.2.3 is executed.

The gap-invoker is activated with the aim of obtaining a resolution from GAP/HAP of the cyclic group C_5 represented with the following XML-Kenzo object.

```
<constructor>
    <cyclic>5</cyclic>
</constructor>
```

From the previous XML-Kenzo object, the gap-invoker constructs the following request, which is sent to the GAP/HAP server.

```
<?scscp start ?>
<OMOBJ>
    <OMATTR>
        <OMATP>
            <OMS cd="scscp1" name="call_id"/>
            <OMSTR>esus.unirioja.es:7500</OMSTR>
        </OMATP>
        <OMA>
            <OMS cd="scscp1" name="procedure_call"/>
            <OMA>
                <OMS cd="scscp_transient_1" name="Resolution"/>
                <OMA>
                    <OMS cd="group1" name="cyclic"/>
                    <OMI>5</OMI>
                </OMA>
            </OMA>
        </OMA>
    </OMATTR>
</OMOBJ>
<?scscp end ?>
```

When the GAP/HAP server receives the above request the following instruction is executed in the GAP/HAP server:

```
gap> ResolutionFiniteGroup(CyclicGroup(5)); ✠
```

The result returned by the GAP/HAP server to the gap-invoker (once we have removed the SCSCP wrapper) is:

```
<OMOBJ>
 <OMA>
  <OMS cd="resolutions" name="resolution"/>
  <!-- GROUP -->
  <OMA>
   <OMS cd="group1" name="cyclic_group"/>
   <OMI>5</OMI>
  </OMA>
  <!--More than 1000 lines skipped-->
  ...
</OMOBJ>
```

This resolution is used by the `k-g-1` module to invoke the Kenzo internal server and construct both in the Kenzo kernel and in the microkernel the space $K(C_5, 1)$. The identifier of the new object is returned to the `homotopy` module.

Afterwards, a `mk-space-k-g` is built in the microkernel as was explained in Subsubsection 4.2.2.3. Subsequently, once the space is constructed in the microkernel, as we are working with a `mk-space-k-g`, the `HES` is activated to compute the homotopy group of the space. The HES uses its rules and returns the result:

```
<result-HES>
    <component>5</component>
    <explanation>
       The space was the Eilenberg MacLane space $K(C_5, 1)$. The homotopy groups
       of an Eilenberg MacLane space $K(G, m)$ are: $\pi_m(K(G, m)) = G$ and
      $\pi_r(K(G, m)) = 0$ if $m \neq r$.
    </explanation>
</result-HES>
```

This result is stored in the internal memory to avoid re-computations by the `homotopy` module and sent to the adapter through the external server. Then, the adapter converts the result into its OpenMath representation:

```
<OMOBJ>
    <OMA>
        <OMS cd="computing" name="result-HES"/>
        <OMI>5</OMI>
        <OMSTR>
          The space was the Eilenberg MacLane space $K(C_5, 1)$. The homotopy groups
          of an Eilenberg MacLane space $K(G, m)$ are: $\pi_m(K(G, m)) = G$ and
         $\pi_r(K(G, m)) = 0$ if $m \neq r$.
        </OMSTR>
    </OMA>
</OMOBJ>
```

and this is the result returned to the client.

4.2.3 Composability of Kenzo and GAP/HAP in the *fKenzo* GUI

This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to provide support for the new functionality presented in the previous subsection.

In this case we have modified the GAP *fKenzo* module, presented in Subsection 4.1.5, to enhance the GUI with support for Eilenberg MacLane spaces of type $K(G, 1)$. Now, the GAP/HAP module references three files: `gap-structure` (that defines the structure of the graphical constituents), `gap-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection (this plug-in is an extension of the presented in Subsection 4.1.3).

We have defined additional graphical elements for the GAP module, using the XUL specification language, in the `gap-structure` file:

- A menu option called `K-G-1` into the menu `Abelian Simplicial Groups`.

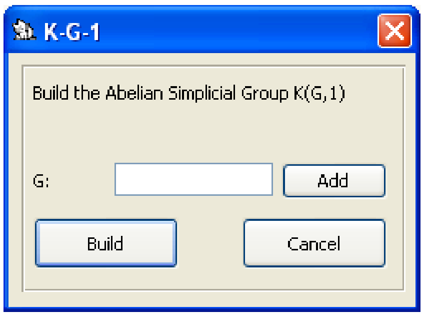


Figure 4.11: `K-G-1` window

- A window called `K-G-1` (see Figure 4.11). As we have explained previously, we do not specify from scratch this window, but we use a generic specification with the desired structure whose attributes take the concrete values of the new window.

In addition to the functionality explained in Subsection 4.1.5, the `gap-functionality` document includes the functionality related to these new components. Namely, a function acting as event handler is associated with the `K-G-1` menu option; this function shows the `K-G-1` window (see Figure 4.11) if a group was constructed previously in the session; otherwise, it shows a message which indicates that a group must be constructed before using this menu option.

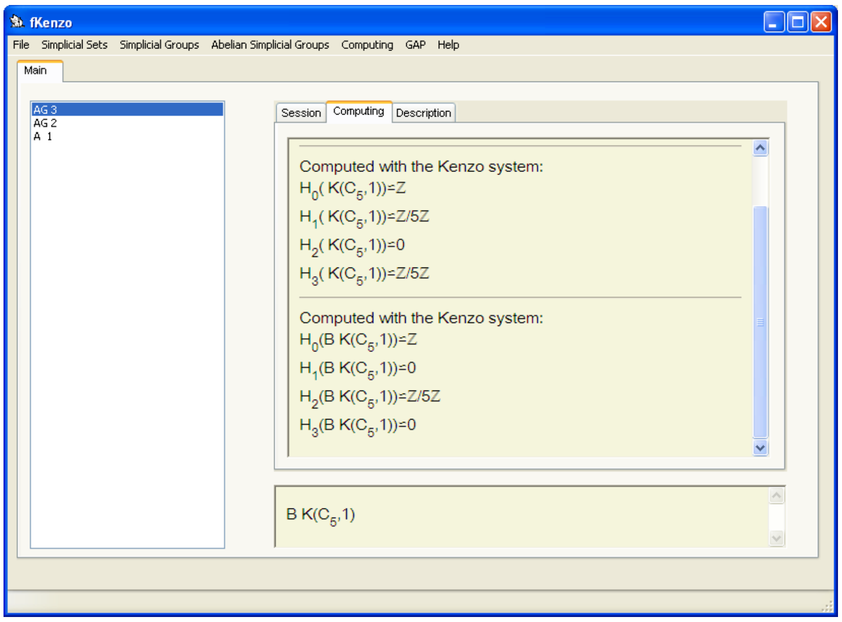
From the `K-G-1` window, the user must select a group from a list with the `Add` button. Once the user has selected a group G, when he presses the `Build` button of the `K-G-1` window, an OpenMath request is generated and sent it to our framework. The Eilenberg MacLane space $K(G, 1)$ and its identification number is returned. Eventually, *fKenzo* adds to the list of constructed objects (situated in the left side of the main tab of the *fKenzo* GUI) the new object.

Then, a *fKenzo* user can use an Eilenberg MacLane space of type $K(G, 1)$ with G a cyclic group as any other space. In particular, he can employ them to construct other spaces (for instance the classifying space of these Eilenberg MacLane spaces, see Figure 4.12) or to compute its homology and homotopy groups, see Figures 4.12 and 4.13.

It is worth noting that from the user point of view he can construct and use the space $K(G, 1)$ as any other space, so he does not know that internally the process to construct this kind of spaces involves the composability of two Computer Algebra systems. Therefore, we are providing an interoperability tool to the user without disconcerting him by the technicalities needed to perform this composability.

In particular, the procedure that the user must follow to achieve the same behavior presented in [RER09] is:

1. Load the GAP *fKenzo* module,
2. build the cyclic group G,

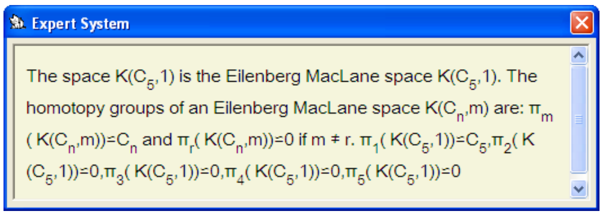

fKenzo
File Simplicial Sets Simplicial Groups Abelian Simplicial Groups Computing GAP Help
Main
AG 3
AG 2
A 1
Session
Computing
Description

Computed with the Kenzo system:
H_0(K(C_5,1))=Z
H_1(K(C_5,1))=Z/5Z
H_2(K(C_5,1))=0
H_3(K(C_5,1))=Z/5Z

Computed with the Kenzo system:
H_0(B K(C_5,1))=Z
H_1(B K(C_5,1))=0
H_2(B K(C_5,1))=Z/5Z
H_3(B K(C_5,1))=0

B K(C_5,1)

Figure 4.12: Homology groups of $K(C_5, 1)$ and $B(K(C_5, 1))$



The space K(C_5,1) is the Eilenberg MacLane space K(C_5,1). The homotopy groups of an Eilenberg MacLane space K(C_n,m) are: π_m(K(C_n,m))=C_n and π_r(K(C_n,m))=0 if m ≠ r. π_1(K(C_5,1))=C_5,π_2(K(C_5,1))=0,π_3(K(C_5,1))=0,π_4(K(C_5,1))=0,π_5(K(C_5,1))=0

Figure 4.13: Explanation facility window for homotopy groups of $K(C_5, 1)$

3. build the space $K(G, 1)$.

As can be seen, this is a much simpler approach than the one presented in [RER09] from the user point of view. However, no reward comes without its corresponding price and some of the constructions developed in [RER09, Rom10] cannot be made available in *fKenzo* (for instance 2-types, since their construction involves a study of its internal structure and a knowledge of the definition of Lisp functions, and such a meticulous study is difficult to integrate in a GUI, at least in an easy and usable way, that is, without giving access to the internal Common Lisp code).

4.3 Integration of the ACL2 Theorem Prover

As we claimed in Section 2.1 one of the challenges of our system was the integration of different kinds of tools; in particular, we were not only interested in integrating tools which allow us to perform computations (such as Computer Algebra systems) but also to certify results (by means of Theorem Proving tools).

To this aim, as a first step, we have taken advantage of the semantical possibilities of OpenMath. Concretely, we have added, in our Content Dictionaries, the properties which the mathematical structures must satisfy. This opens the chance of interfacing OpenMath with different theorem provers. A similar approach (in the sense that it involved both OpenMath and a Theorem Prover), using the proof checkers Lego and Coq, was proposed by Caprotti and Cohen in [CC99]. The approach followed in that paper consisted in checking whether OpenMath expressions were well-typed with Lego and Coq Theorem Provers or not. Our approach is a bit different, on the one hand, we have used the ACL2 theorem prover instead of Coq and, on the other hand, we want to define mathematical structures in Content Dictionaries; then, an interpreter will transform those Content Dictionaries into ACL2 encapsulates (see Subsection 1.3.2) which can be used later on in ACL2. The importance of this case study comes from the fact that we are giving the first steps to store mathematical theories developed in Theorem Provers in OpenMath/OMDoc documents, increasing the portability of those theories to different theorem provers.

The rest of this section is organized as follows. Subsection 4.3.1 is devoted to present how axiomatic information is added to our Content Dictionaries. The transformation from Content Dictionaries to ACL2 encapsulates is explained in Subsection 4.3.2. The integration of ACL2 in our framework and in the *fKenzo* GUI are presented respectively in subsections 4.3.3 and 4.3.4.

4.3.1 Adding axiomatic information to Content Dictionaries

Up to now, we have defined four Content Dictionaries (see Subsection 2.2.5) related to the different kind of objects that can be built in our system (Chain Complexes, Simplicial

Sets, Simplicial Groups and Abelian Simplicial Groups). These Content Dictionaries, which are www-available at [Her11], defined several objects (such as spheres, loop spaces and so on) which instantiate a mathematical structure. However, they did not formally define the mathematical structure. To deal with this question we have proceeded as follows.

The Kenzo mathematical structures (see Figure 1.2 of Subsection 1.2.1) are algebraic structures which properties are axiomatically given and which have associated a signature with the arities of the functions which define an object of that structure. For each one of the Kenzo mathematical structures we have defined an OpenMath Content Dictionary (or extended the ones already defined) to include the formal definition of these mathematical structures.

To this aim we have based on the Small Type System formalism, see [Dav99] for details, which has been designed to give semi-formal signatures to OpenMath symbols. By using this mechanism we have included signatures in the OpenMath objects definition. In addition, we have specified their properties in two different ways (by means of `<FMP>` and `<CMP>` tags) and we have associated an instance example with them.

We are going to focus on the `SS` Content Dictionary which defines the notion of simplicial sets introduced in Definition 1.17; the rest of Content Dictionaries are based on the same ideas.

To define the *simplicial set* structure, we must provide the disjoint sets $\{K^q\}_{q\geq 0}$ and both face and degeneracy operators. The sets $\{K^q\}_{q\geq 0}$ can be seen as a graded set; so, it is possible to consider its characteristic function which, from an element x and a degree g, determines if the element x belongs to the set K^g. To be precise, an *invariant* function can be used in order to encode the characteristic function of the graded set $\{K^q\}_{q\geq 0}$.

Based on the previous way of representation, the following signature, let us called it `SS`, has been defined for simplicial sets.

```
inv  : u   nat       -> bool
face : u   nat   nat -> u
deg  : u   nat   nat -> u
```

where `inv`, `face` and `deg` represent the characteristic function of the underlying set and the face and degeneracy operators respectively, and `u` denotes the Universe, of Lisp objects in this case.

The `SS` signature can be codified using the OpenMath `Signature` element as follows:

```
<Signature name="simplicial-set">
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
            <OMS name="mapsto" cd="sts"/>
            <OMA id="inv">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMS cd="setname2" name="boolean"/>
            </OMA>
            <OMA id="face">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMV name="PositiveInteger"/>
                <OMV name="Element"/>
            </OMA>
            <OMA id="degeneracy">
                <OMS cd="sts" name="mapsto"/>
                <OMV name="Element"/>
                <OMV name="PositiveInteger"/>
                <OMV name="PositiveInteger"/>
                <OMV name="Element"/>
            </OMA>
            <OMV name="Simplicial-Set"/>
        </OMA>
    </OMOBJ>
</Signature>
```

The above OpenMath `Signature` must be read as follows. Each application `OMA` inside the main `mapsto` of the `simplicial-set` signature represents each one of the functions of the `SS` signature. The value of the `id` of the application tag (`<OMA id=" ">`) is the name of the function. The `mapsto` symbol, inside of the application tag, is applied to n variables and/or symbols, the first $n-1$ will be the inputs and the last one the output of the function. The "type" of the inputs and outputs is also included. In this way, we can provide the signature of each Kenzo mathematical structure.

The formal mathematical properties of the simplicial sets are given in the `<FMP>` tags of the *simplicial set* definition. In this case `<FMP>` elements state the properties of invariance of face and degeneracy operators and the relations between them (the five properties included in the definition of simplicial sets). All of them have also been included in natural language by using `<CMP>` elements. For instance, the face operator invariance ($x \in K^q \Rightarrow \partial_i x \in K^{q-1}$) is represented as follows:

```
<CMP> Face operator invariance: x \in K^q => \partial_i x \in K^{q-1} </CMP>
<FMP>
 <OMA>
  <OMS cd="logic1" name="implies"/>
  <OMA> <OMS name="inv"/> <OMV name="x"/> <OMV name="q"/> </OMA>
  <OMA>
    <OMS name="inv"/>
    <OMA> <OMS name="face"/> <OMV name="x"/> <OMV name="i"/> <OMV name="q"/> </OMA>
    <OMA> <OMS cd="arith1" name="minus"/> <OMV name="q"/> <OMI>1</OMI> </OMA>
  </OMA>
 </OMA>
</FMP>
```

Finally, an example of a concrete simplicial set has been included. Namely, the simplicial set with one element belonging to each set K^q and with each face and degeneracy operation of degree q returning the element of degree $q-1$ and $q+1$ respectively has been considered.

```
<Example>
    ...
    <OMBIND>
        <OMS name="face"/>
        <OMBVAR>
            <OMV name="x"/>  <OMV name="i"/> <OMV name="q"/>
        </OMBVAR>
        <OMS cd="list" name="nil"/>
    </OMBIND>
    ....
</Example>
```

In this way, all the Kenzo mathematical structures can be defined by means of OpenMath Content Dictionaries.

4.3.2 From Content Dictionaries to ACL2 encapsulates

Content Dictionaries, defined in the way presented in the previous subsection, open the chance of interfacing OpenMath with theorem provers; namely, in our case, with the ACL2 Theorem Prover (presented in Section 1.3). The main reason to choose ACL2 was the fact that, as Kenzo, it is a Common Lisp program, then, we can use ACL2 to verify real Kenzo code, as we will see from the next chapter.

It is worth noting that our Content Dictionaries include all the necessary information to generate ACL2 encapsulates. Let us remember that ACL2 supports the constrained introduction of new function symbols by means of the encapsulate notion, a detailed description of this ACL2 functionality was presented in Subsection 1.3.2. Briefly, an *encapsulate* allows the introduction of function symbols in ACL2, without a completely specification of them, but just assuming some properties which define them partially. An ACL2 encapsulate consists of a set of function signatures, a set of properties of these functions and a "witness" for each one of the functions, where a witness is an existing function that can be proved to have the required properties (witnesses are provided to avoid the introduction of inconsistencies in ACL2).

From each Content Dictionary specified as the one presented in the previous subsubsection, an ACL2 encapsulate can be generated. From now on, we are going to explain the transformation from one of our Content Dictionaries (in particular the `SS` Content Dictionary) to an ACL2 encapsulate.

First of all, the OpenMath signatures must be transformed to ACL2 signatures. Each application `OMA` inside the main `mapsto` of the simplicial set signature is translated into a function of the encapsulate in the following way. The value of the `id` of the application tag (`<OMA id=" ">`) will be the name of the function, the `mapsto` symbol inside the application tag is converted to `=>` in ACL2. The `mapsto` symbol is applied to n variables and/or symbols, the first $n-1$ will be the inputs and the last one the output. Note that ACL2 is a system without explicit typing, so, although the "type" of the objects has been included in the Content Dictionary they will be translated into asterisks in ACL2. Adding the necessary brackets, the following ACL2 signature is obtained.

```
((inv * *) => *))
(((face * * *) => *)
((degeneracy * * *) => *)
```

The following step consists of transforming the mathematical properties, specified in the Content Dictionary, into ACL2 lemmas. In order to do this, we proceed as follows. First of all, the `<CMP>` tags will be translated into ACL2 comments, expressed with "`;`". To each `<FMP>` tag, a new lemma, `defthm` in ACL2 syntax, with the name `prop-n` where `n` is a variable that indicates the number of the property, must be defined. The `implies`, `and`, `eq` and `minus` OpenMath symbols are translated respectively into the `implies`, `and`, `equal` and "`-`" ACL2 functions (these are some examples of the equivalences established between OpenMath symbols and ACL2 functions). For instance, the following ACL2 lemma about the face operator invariance is obtained from the respective property in the Content Dictionary.

```
; Face operator invariance
(defthm prop-1 (implies (inv x q) (inv (face x i q) (- q 1))))
```

And, finally, from the `<Example>` tags, the witnesses are obtained. Each example in a Content Dictionary will be a local definition in an ACL2 encapsulate. The `OMBIND` symbol indicates the beginning of the definition, `defun` in ACL2, its first argument is a symbol which indicates the name of the function, the second one is an `OMBVAR` element which specifies the name of the parameters and the third one is the body of the function.

If some of the arguments of the function does not appear in its body, they will be ignored to obtain a correct ACL2 function, as we show in the translation of the previous subsubsection.

```
(local (defun face (x i q) (declare (ignore x i q)) nil))
```

Therefore, we have an interpreter which is able to construct ACL2 encapsulates from some concrete Content Dictionaries. This interpreter is a Common Lisp program which takes as input an OMDoc file specified in the way presented in Subsubsection 4.3.1 and constructs an ACL2 encapsulate. All the generated encapsulates can be evaluated in ACL2.

In the same way that we have developed an interpreter which generates ACL2 encapsulates from Content Dictionaries, we can also implement other interpreters which supply suitable code for other Theorem Provers such as Isabelle or Coq.

4.3.3 Integration of ACL2

To integrate the ACL2 Theorem Prover in our framework we have developed a plug-in following the guidelines given in Subsubsection 3.1.2. This new plug-in will allow us to verify ACL2 scripts (an ACL2 script is a file of ACL2 forms, such as definitions or theorems, which is processed in a sequential way) obtaining as result a file with the ACL2 output obtained from the evaluation of the script in ACL2. Moreover, we have included the interpreter presented in the previous subsection as a module of the microkernel to generate ACL2 encapsulates from Content Dictionaries. This plug-in references the following resources:

```
<code id="ACL2">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> ACL2-is.lisp </data>
   <data format="Kf/microkernel"> ACL2-m.lisp </data>
   <data format="Kf/microkernel"> CD-to-ACL2.lisp </data>
   <data format="Kf/adapter"> ACL2-a.lisp </data>
</code>
```

These resources deserve a detailed explanation that is provided in the following subsubsections.

4.3.3.1 Extending the XML-Kenzo schema

As we have just explained, the new plug-in will allow us to execute ACL2 scripts and store the output produced by ACL2 in a file. Then, we need to represent the way of providing the path of both the ACL2 script and the ACL2 output. Therefore, we

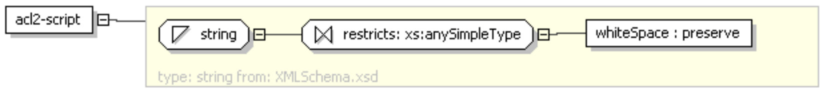


Figure 4.14: acl2-script XML-Kenzo specification

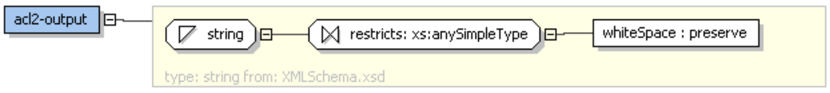


Figure 4.15: acl2-output XML-Kenzo specification

have extended the XML-Kenzo specification (`XML-Kenzo.xsd` file) to this aim. In this specification, we have defined two new elements: a new element of the `requests` group called `acl2-script` (see Figure 4.14), whose value is a string which indicates the path of the ACL2 script, and a new element of the `results` group called `acl2-ouput` (see Figure 4.15) whose value is a string which indicates the path of an ACL2 output file.

Moreover, we want to introduce new functionality in our system which allows us to generate an ACL2 encapsulate from a Content Dictionary. Therefore, we have extended the XML-Kenzo specification to admit this new functionality. In the XML-Kenzo specification, we have defined a new element of the `requests` group called `CD-to-ACL2` (see Figure 4.16), whose value is a string which indicates the path of an OMDoc document; and a new element of the `results` group called `ACL2-encapsulate` (see Figure 4.17) whose value is a string, namely an ACL2 encapsulate.

As we explained in Subsubsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified the external server is able to receive XML-Kenzo objects such as:

```
<acl2-script> acl2-script-path </acl2-script>
```

```
<OMDoc-to-ACL2> OMDoc-path-file </OMDoc-to-ACL2>
```

4.3.3.2 A new internal server: ACL2

To integrate the ACL2 Theorem Prover, we have followed the same methodology explained in the Kenzo case (see Subsection 2.2.2): we have ACL2 wrapped with an

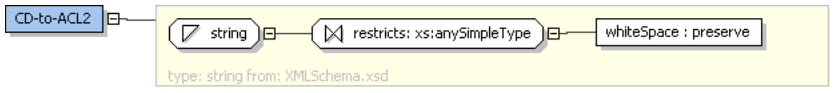


Figure 4.16: `CD-to-ACL2` XML-Kenzo specification

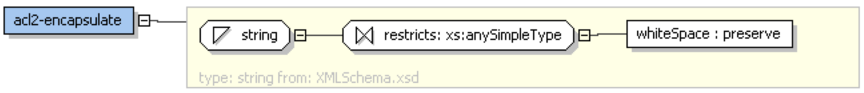


Figure 4.17: `acl2-encapsulate` XML-Kenzo specification

XML-Kenzo interface and the communication between the microkernel and ACL2 is performed by means of XML-Kenzo requests through an external interface which offers the available services of our ACL2 internal server (at this moment the only available service allow the execution of ACL2 scripts). As in the case of the GAP/HAP system we must install ACL2 (the ACL2 installer can be downloaded from [KM]).

The `ACL2-is.lisp` provides all the necessary functionality to integrate the ACL2 internal server; that is to say, ACL2, the wrapper and the interface. The ACL2 interface provides just one service called `execute-acl2-script`. The function associated with this service, also called `execute-acl2-script`, takes as input an `acl2-script` XML-Kenzo object. From the file indicated in the `acl2-script` XML-Kenzo object, the `execute-acl2-script` function extracts the ACL2 code and executes it in ACL2. The `execute-acl2-script` function returns the path of a file where the output generated by ACL2, when executing the script, is stored. This path is returned in an `acl2-output` XML-Kenzo object.

4.3.3.3 New modules of the microkernel

The `ACL2-m.lisp` file defines a new module for the microkernel called `ACL2`. The procedure implemented in this module is just in charge of checking that the path indicated by the `acl2-script` XML-Kenzo object exists in which case the module invokes the ACL2 internal server; if the path does not exist this module informs the user about this situation by means of a `warning` XML-Kenzo object. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the `ACL2` module.

This new module can be considered neither a computation module nor a construction module. This module belongs to a new category of modules related to theorem proving tools called *verification modules*.

Moreover, the `CD-to-ACL2.lisp` defines a new verification module for the microkernel called `CD-to-ACL2`. The procedure implemented in this module is the interpreter which transforms an OMDoc document indicated in an `OMDoc-to-ACL2` XML-Kenzo object to an ACL2 encapsulate. In addition, this file enhances the interface of the microkernel in order to be able to invoke the new module.

4.3.3.4 Increasing the functionality of the adapter

Finally, we have extended the `Aux` Content Dictionary by means of the definition of four new objects: `acl2-script`, `acl2-ouput`, `CD-to-ACL2` and `ACL2-encapsulate`. Therefore, the `ACL2-a.lisp` file contains the functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects, devoted to indicate the path of a file, to XML-Kenzo requests. Namely, we have extend the Phrasebook by means of a new parser in charge of this task. For instance, the XML-Kenzo request:

```
<acl2-script> acl2-script-path </acl2-script>
```

is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
      <OMS cd="Aux" name="acl2-script"/>
      <OMSTR> acl2-script-path </OMSTR>
   </OMA>
</OMOBJ>
```

4.3.4 Integration of ACL2 in the *fKenzo* GUI

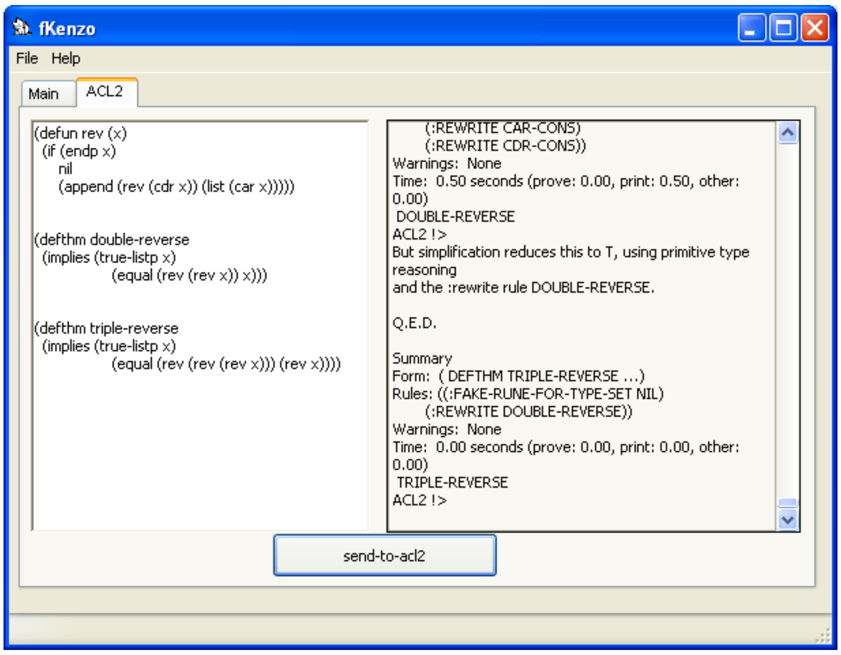
This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to provide access to the new functionality presented in the previous subsection.

In this case we have defined a fresh *fKenzo* module to enhance the GUI with support for the ACL2 system. The new module references three files: `acl2-structure` (that defines the structure of the graphical constituents), `acl2-functionality` (which provides the functionality related to the graphical constituents) and the last plug-in introduced.

We have defined three graphical elements, using the XUL specification language, in the `acl2-structure` file:

- A new tab called `ACL2` which is included in the main panel (see Figure 4.18).
- A menu called `ACL2` which contains two options: `CD to ACL2` and `CD to ACL2 in file`.
- A window called `CD-to-ACL2` (see Figure 4.19).

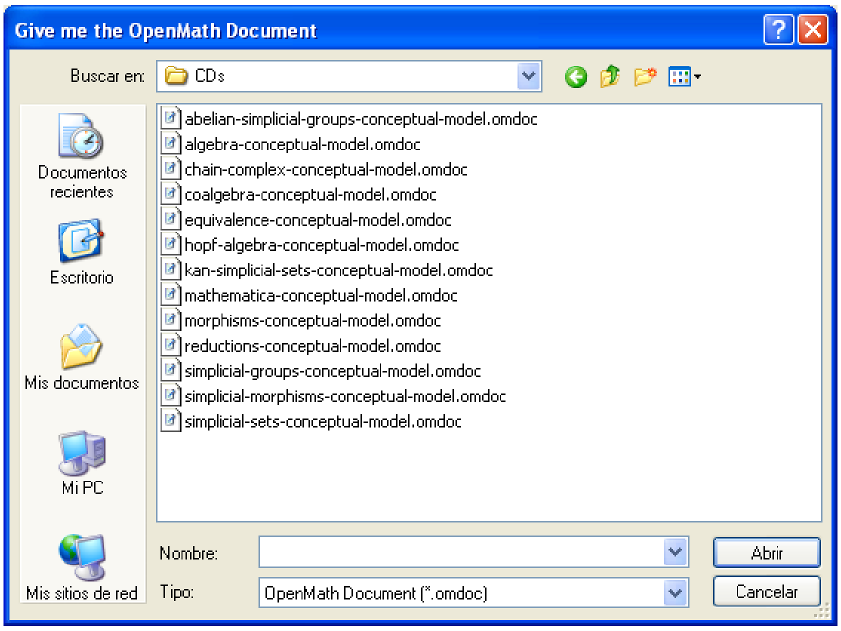
The new tab page contains two areas and one button: the left area will be used to display ACL2 instructions, the button will send the instructions of the left side to ACL2, and finally, the right area will show the ACL2 result obtained from the evaluation of the instructions of the left area. A *fKenzo* user can manually write ACL2 definitions and theorems in the left area and then execute them, but the idea is that the ACL2 scripts are going to be automatically generated by the system and displayed in the left area of the ACL2 tab.



```
(defun rev (x)
 (if (endp x)
    nil
    (append (rev (cdr x)) (list (car x)))))


(defthm double-reverse
 (implies (true-listp x)
          (equal (rev (rev x)) x)))


(defthm triple-reverse
 (implies (true-listp x)
          (equal (rev (rev (rev x))) (rev x))))
```

```
       (:REWRITE CAR-CONS)
       (:REWRITE CDR-CONS))
Warnings:  None
Time:  0.50 seconds (prove: 0.00, print: 0.50, other:
0.00)
 DOUBLE-REVERSE
ACL2 !>
But simplification reduces this to T, using primitive type
reasoning
and the :rewrite rule DOUBLE-REVERSE.

Q.E.D.

Summary
Form:  ( DEFTHM TRIPLE-REVERSE ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
       (:REWRITE DOUBLE-REVERSE))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other:
0.00)
 TRIPLE-REVERSE
ACL2 !>
```

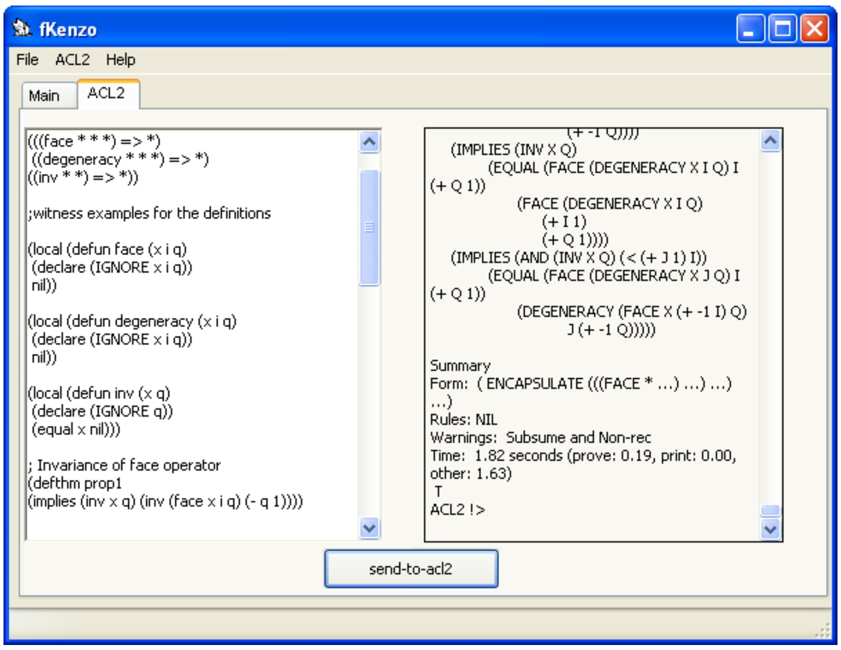
Figure 4.18: The *ACL2* tab with an example



abelian-simplicial-groups-conceptual-model.omdoc
algebra-conceptual-model.omdoc
chain-complex-conceptual-model.omdoc
coalgebra-conceptual-model.omdoc
equivalence-conceptual-model.omdoc
hopf-algebra-conceptual-model.omdoc
kan-simplicial-sets-conceptual-model.omdoc
mathematica-conceptual-model.omdoc
morphisms-conceptual-model.omdoc
reductions-conceptual-model.omdoc
simplicial-groups-conceptual-model.omdoc
simplicial-morphisms-conceptual-model.omdoc
simplicial-sets-conceptual-model.omdoc

Nombre:
Tipo: OpenMath Document (*.omdoc)

Figure 4.19: The CD-to-ACL2 window



```
(((face * * *) => *)
 ((degeneracy * * *) => *)
((inv * *) => *))

;witness examples for the definitions

(local (defun face (x i q)
 (declare (IGNORE x i q))
 nil))

(local (defun degeneracy (x i q)
 (declare (IGNORE x i q))
 nil))

(local (defun inv (x q)
 (declare (IGNORE q))
 (equal x nil)))

; Invariance of face operator
(defthm prop1
(implies (inv x q) (inv (face x i q) (- q 1))))
```

```
                          (+ -1 Q))))
    (IMPLIES (INV X Q)
             (EQUAL (FACE (DEGENERACY X I Q) I
(+ Q 1))
                    (FACE (DEGENERACY X I Q)
                          (+ I 1)
                          (+ Q 1))))
    (IMPLIES (AND (INV X Q) (< (+ J 1) I))
             (EQUAL (FACE (DEGENERACY X J Q) I
(+ Q 1))
                    (DEGENERACY (FACE X (+ -1 I) Q)
                                J (+ -1 Q)))))

Summary
Form:  ( ENCAPSULATE (((FACE * ...) ...) ...)
...)
Rules: NIL
Warnings:  Subsume and Non-rec
Time:  1.82 seconds (prove: 0.19, print: 0.00,
other: 1.63)
 T
ACL2 !>
```

Figure 4.20: The *ACL2* tab with the simplicial sets encapsulate

The functionality stored in the `acl2-functionality` includes the new graphical constituents. On the one hand, the event handler associated with the `send-to-acl2` button is defined in this file. This event handler obtains the ACL2 instructions which have been written in the left side of the ACL2 tab, afterwards it stores them in a temporal file, subsequently an `acl2-script` OpenMath request is created with the path of the temporal file and sent to the framework. The returned result is shown in the right side of the ACL2 tab.

On the other hand, the event handlers associated with the `CD to ACL2` and `CD to ACL2 in file` menu options are also included in the `acl2-functionality` file. The former event handler shows the `CD-to-ACL2` window (see Figure 4.19) which allows the user to choose a Content Dictionary with the description of one of the Kenzo mathematical structures. Once the user has selected a Content Dictionary; the system generates an ACL2 encapsulate which is displayed in the left part of the `ACL2` tab (see Figure 4.20). Then, the user only has to use the functionality associated with the `send-to-acl2` button and the ACL2 output produced when the encapsulate is executed in ACL2 is shown in the right side of the `ACL2` tab (see Figure 4.20). The latter event handler, the one associated with the `CD to ACL2 in file` menu option, instead of writing the encapsulate generated in the left part of the `ACL2` tab, generates a file with the encapsulate and asks a path to the user to save the file.

As can be noticed, our interface to interact with ACL2 is a plain text editor which is obviously less usable than typical ACL2 interfaces (Emacs [Sta81] or ACL2 sedan [DMMV07]). It is worth noting that our goal was not the creation of an ACL2 interface which could compete with regular ACL2 editors. The idea is that the ACL2 scripts are automatically generated by the system and written in the left area of the ACL2 tab (as we have seen for the case of the encapsulates generated from Content Dictionaries) and the user only has to press the `send-to-acl2` button, without typing any additional ACL2 command, to obtain certificates.

4.4 Interoperability between Kenzo, GAP/HAP and ACL2

As we have said several times throughout this chapter, we are interested not only in integrating several tools in our framework and use them individually, but also in making them work together.

The integration of Kenzo and GAP to construct the effective homology version of Eilenberg MacLane spaces of type $K(G, n)$ where G is a cyclic group was presented in Section 4.2. Now, we want to prove the correctness of those programs by means of the ACL2 Theorem Prover. The importance of this verification lies in the fact that Eilenberg MacLane spaces of type $K(G, n)$ where G is a cyclic group are instrumental in the computation of homotopy groups.

We have integrated the already available tools in *fKenzo* to prove the correctness of some Kenzo programs. In particular, we want to verify the correctness of Kenzo statement like the following one using the ACL2 Theorem Prover.

```
> (cyclicgroup 5) ✠
[K1 Abelian-Group]
```

This means that we want to prove in ACL2 that the returned object by the `cyclicgroup` Kenzo function really satisfies the axioms of an abelian group. This is important, for instance, to ensure the following function,

```
> (DEFMETHOD K-G-1 ((group AB-GROUP)) ...) ✠
```

which constructs the Eilenberg MacLane space of its argument, is really applied over a meaningful input (that is to say, an actual abelian group). Let us recall that the construction of the effective homology of that kind of Eilenberg MacLane spaces involves the use of GAP/HAP to obtain a resolution (see Algorithm 4.5). To sum up, we use Kenzo to construct Eilenberg MacLane spaces of cyclic groups; in addition by means of GAP/HAP we are able to construct the effective homology of these Eilenberg MacLane spaces; and, ACL2 increases the reliability of the construction of these Eilenberg MacLane spaces verifying the correctness of their input argument. Then, these three systems are combined producing a powerful and reliable tool.

The rest of this section is organized as follows. Subsection 4.4.1 explains the implementation of cyclic groups in the Kenzo system. Subsection 4.4.2 deals with the proof in ACL2 of the correctness of the implementation of cyclic groups introduced in Subsection 4.4.1. The integration of Kenzo, GAP/HAP and ACL2 in our framework and in the *fKenzo* GUI is presented in Subsections 4.4.3 and 4.4.4 respectively.

4.4.1 Implementation of cyclic groups in Kenzo

The abelian group structure is a mathematical structure which was not included in the original version of Kenzo; but it was included in the development of [RER09] to construct Eilenberg MacLane spaces of abelian groups. This structure was defined, following the same schema that the one used to define mathematical structures in Kenzo (see Subsection 1.2.1), using the two following Common Lisp class definitions:

```
(DEFCLASS GROUP ()
   ((elements :type group-basis :initarg :elements :reader elements)
   (cmpr :type cmprf :initarg :cmpr :reader cmpr1)
   (mult :type function :initarg :mult :reader mult1)
   (inv :type function :initarg :inv :reader inv1)
   (nullel :type gnrt :initarg :nullel :reader nullel)
   (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
   (orgn :type list :initarg :orgn :reader orgn)
   (resolution :type reduction :initarg :resolution :reader resolution)))
```

```
(DEFCLASS AB-GROUP (GROUP) ())
```

It is worth noting that the `AB-GROUP` class, which represents abelian groups, is a subclass, without any additional slot, of the `GROUP` class. The relevant slots (relevant in the sense of being important for our verification process) of this class are `elements`, a list of the elements of the group; `mult`, the function defining the binary operation of two elements of the group; `inv`, the function defining the inverse of the elements of the group and `nullel`, which is the null element of the group.

The `cyclicgroup` function constructs instances of the `AB-GROUP`. In particular, it takes as argument a natural number n and constructs an instance of the `AB-GROUP` which represents the cyclic group C_n. The concrete definition of the `cyclicgroup` function is:

```
(defun cyclicgroup (n)
  (build-ab-group
   :elements (<a-b> 0 (1- n))
   :mult #'(lambda (g1 g2) (mod (+ g1 g2) n))
   :inv #'(lambda (g) (mod (- n g) n))
   :nullel 0
   :orgn `(Cyclic-group of order ,n)))
```

This piece of code must be read as follows. From a natural number n the `cyclicgroup` function constructs an `AB-GROUP` instance where the slot `elements` has as value the ordered list of natural numbers from 0 to $n-1$ (generated from the `<a-b>` function), the slot `mult` is a functional slot which from two elements $g1$ and $g2$ returns the value of $(g1 + g2) \mod n$, the slot `inv` is a functional slot which from an element g returns the value of $(n - g) \mod n$ and 0 is the null element stored in the slot `nullel`. Finally, the `orgn` slot is a comment which provides the "origin" of the object.

The rest of the slots of the `AB-GROUP` instance are either automatically generated, in the case of `idnm`, or a value is assigned after the construction of the object using a GAP resolution, in the case of the `resolution` slot.

4.4.2 The proof in ACL2 about cyclic groups

A group is a mathematical structure which can be defined by means of four functions based on the following signature, called `GRP`:

```
invariant  : u    -> bool
mult       : g  g -> g
inv        : g    -> g
nullel     :      -> g
```

The four functions of this signature are: the `invariant` unary operation (which represents if an element belongs to the group, that is, the characteristic function of the underlying set), the `mult` binary operation (the product), the `inv` unary operation (the inverse) and the `nullel` constant operation (the null element). As concrete axiomatization of a group we have chosen that of [Lip81].

Therefore, if we want to prove in ACL2 that four concrete functions, with the correct arities given by the `GRP` signature, determine an (abelian) group, we need to prove that these functions satisfy the properties of (abelian) groups.

Then, for instance if we want to verify in ACL2 that the Kenzo implementation of the cyclic group C_5 is an abelian group we must proceed as follows. First of all, we need to define the four functions which determine the group (we add the suffix `cn`, where `n` is the dimension of the group, to the function names):

```
(defun invariant-c5 (g) (member g (<0-n> 5))

(defun mult-c5 (g1 g2) (mod (+ g1 g2) 5))

(defun inv-c5 (g) (mod (- 5 g) 5))

(defun nullel-c5 () 0)
```

It is worth noting that the definitions of `mult-c5`, `inv-c5` and `nullel-c5` are exactly the same as the ones used in the `cyclicgroup` function for the slots `mult`, `inv` and `nullel` of the `AB-GROUP` instance in the case of $n = 5$. The `invariant-c5` function is the characteristic function of the group and is directly obtained from the `elements` slot of the `AB-GROUP` instance; the transformation from this slot to our function can be considered safe (we have replaced the `<a-b>` function with the `<0-n>` function which constructs the ordered list of natural numbers from 0 to $n-1$; that is, the `<0-n>` function with argument n has the same behavior that the `<a-b>` function with arguments 0 and n). Then, if we prove that these functions (which have the correct arities specified in the `GRP` signature) determine an abelian group (that is, they satisfy the axioms of abelian groups) we can claim that the object constructed in the Kenzo system determines an abelian group. Then, we need to prove theorems in ACL2 such as the following one:

```
(defthm abelian-c5
    (implies (and (invariant-c5 a) (invariant-c5 b))
             (equal (mult-c5 a b) (mult-c5 b a))))
```

The ACL2 Theorem Prover is able to generate a proof of this kind of theorems without any external help. Therefore, we have a proof of the correctness of the `cyclicgroup` Kenzo function in the case of $n = 5$; and we could proceed in the same way for every concrete case of n. Then, we can say that the `cyclicgroup` Kenzo function taking as input the value 5 produces an abelian group. However, it would be more interesting to have a proof saying that for every natural number n the `cyclicgroup` Kenzo function produces an abelian group.

The `GRP` signature represents a group. In order to handle different groups, in [LPR99] an operation between signatures, called $()_{imp}$ operation was introduced. From a signature for an algebraic structure, a new signature for a family of the above algebraic structures can be defined. In our case, the signature $\texttt{GRP}_{imp}$ is defined as follows:

```
imp-invariant  :  imp_GRP g      -> bool
imp-mult       :  imp_GRP g   g  -> g
imp-inv        :  imp_GRP g      -> g
imp-nullel     :  imp_GRP        -> g
```

where the new sort imp_{GRP} is the sort for groups. This signature, really signatures $()_{imp}$, is the one really used in the Computer Algebra systems because it allows the manipulation of algebraic structures as data.

Therefore, if we want to prove in ACL2 that four concrete functions, with the correct arities given by the $\texttt{GRP}_{imp}$ signature, determine an (abelian) group, we need to prove that these functions satisfy the axioms of (abelian) groups for every element of the imp_{GRP} sort.

In particular, we have the following four definitions for the case of cyclic groups defined in Kenzo:

```
(defun imp-invariant (n g) (member g (<0-n> n))

(defun imp-mult (n g1 g2) (mod (+ g1 g2) n))

(defun imp-inv (n g) (mod (- n g) n))

(defun imp-nullel (n) (declare (ignore n)) 0)
```

which can be considered as the definitions used by the Kenzo systems for the construction of objects which represent cyclic groups. Then, if we prove that these functions determine an abelian group (that is, they satisfy the properties of abelian groups) for every natural number, we can claim that every object constructed in the Kenzo system with the `cyclicgroup` function, taking as argument a natural number, determines an abelian group. Then, we need to prove theorems such as the following one:

```
(defthm imp-abelian
    (implies (and (natp n) (imp-invariant n a) (imp-invariant n b))
             (equal (imp-mult n a b) (imp-mult n b a))))
```

The ACL2 Theorem Prover is able again to generate a proof of those theorems without any external help if we load in ACL2 an arithmetic library. Therefore, we have a proof of the correctness of the `cyclicgroup` Kenzo function for every natural number. That is to say, for every natural number n the `cyclicgroup` Kenzo function constructs an abelian group.

4.4.3 Composability of Kenzo, GAP/HAP and ACL2

In the previous subsubsection we have presented the ACL2 code to verify that the implementation of cyclic groups in the Kenzo system really constructs abelian groups. We have included a new verification module in the microkernel to generate the ACL2 functions and theorems necessary to certify that a concrete group of the Kenzo system is really an (abelian) group. That is to say, given a group G, our framework will generate the four functions (`invariant`, `mult`, `inv` and `nullel`) which define the group and the theorems which ensure that those four functions determine a group.

We have developed a new plug-in which allows us to integrate a code generator of the functions and theorems explained in the previous subsection.

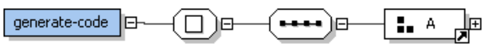


Figure 4.21: generate-code XML-Kenzo specification

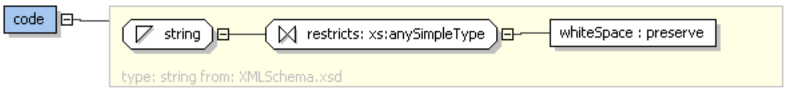


Figure 4.22: code XML-Kenzo specification

```
<code id="certificate">
   <data format="Kf"> GAP </data>
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/microkernel"> code-generator.lisp </data>
   <data format="Kf/adapter"> certificate-a.lisp </data>
</code>
```

It is worth noting that this plug-in needs the GAP plug-in presented in Subsection 4.2.2 to work; then the GAP plug-in is referenced from the `certificate` plug-in. Let us explain now the rest of resources.

We want to introduce new functionality in our system which allows us to obtain the necessary ACL2 functions and theorems to verify that a concrete group of the Kenzo system is really an (abelian) group. Therefore, we have extended the XML-Kenzo specification to admit this new functionality. In the XML-Kenzo specification, we have defined a new element of the `requests` group called `generate-code` (see Figure 4.21), whose value is an element of the `A` group, and a new element of the `results` group called `code` (see Figure 4.22), whose value is a string, namely the functions and theorems associated with a concrete group.

As we explained in Subsection 2.2.4 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the `XML-Kenzo.xsd` file is modified the external server is able to receive XML-Kenzo objects such as:

```
<generate-code> <cyclic> 5 </cyclic> </generate-code>
```

The `code-generator.lisp` defines a new module for the microkernel called `code-generator`. The procedure implemented in this module is a Common Lisp program which generates the functions and theorems associated with the group indicated in the `generate-code` XML-Kenzo object which invokes this module. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the new module.

Finally, we have extended the `Aux` Content Dictionary by means of the definition of two new objects: `generate-code` and `code`. Therefore, the `certificate-a.lisp` file contains the functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects to XML-Kenzo objects. For instance, the above

XML-Kenzo request is generated by the adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="Aux" name="generate-code"/>
      <OMA> <OMS cd="group1" name="cyclic"/> <OMI>5</OMI> </OMA>
   </OMA>
</OMOBJ>
```

It is worth noting that the generation of certificable code is only available for cyclic groups; but we are willing to try the certification of all the objects that can be constructed in our system.

4.4.4 Composability of Kenzo, GAP/HAP and ACL2 in the *fKenzo* GUI

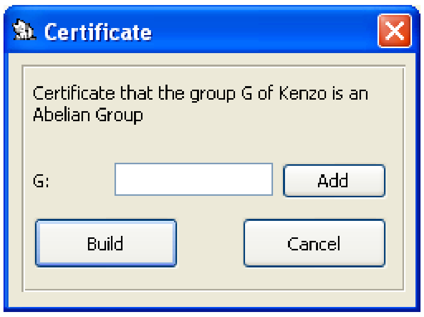
The composability of Kenzo, GAP/HAP and ACL2 in *fKenzo* to provide certificates of the correctness of the implementation of Kenzo cyclic groups does not involve a great development, since most of the ingredients were available from the modules related to GAP/HAP and ACL2.

The new module which provides the tools to compose Kenzo, GAP/HAP and ACL2 in *fKenzo* is called `GAP-Kenzo-ACL2`. This module references the GAP/HAP (Subsection 4.2.3) and ACL2 (Subsection 4.3.4) modules and also three additional files: `gap-kenzo-acl2-structure` (that defines the structure of the new graphical constituents), `gap-kenzo-acl2-functionality` (which provides the functionality related to the graphical constituents), and the plug-in explained in the previous subsection.

When this new module is loaded in *fKenzo*, the graphical elements of the GAP/HAP and ACL2 modules and their functionality are loaded in the system. Besides, two new graphical elements have been defined, using the XUL specification language, in the `gap-kenzo-acl2-structure` file:

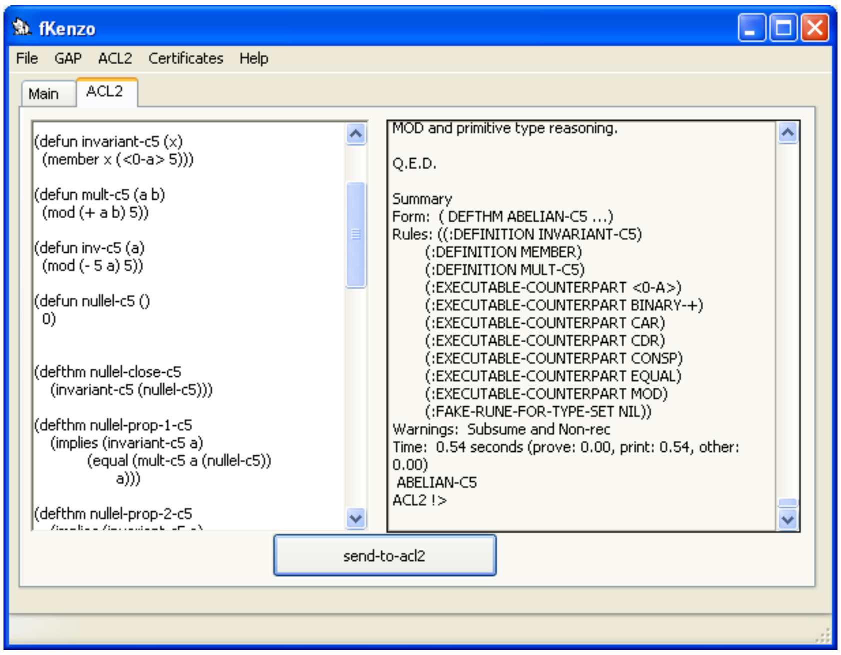
- A menu called `certificates` which contains one option called: `certificate`.
- A window called `certificate` (see Figure 4.23).

The functionality stored in the `gap-kenzo-acl2-functionality` document related to these components works as follows. A function acting as event handler is associated with the `certificate` menu option; this function shows the `certificate` window (see Figure 4.23) if a cyclic group was constructed previously in the session, otherwise the system informs the user about the need of constructing a group.



Certificate that the group G of Kenzo is an Abelian Group

Figure 4.23: `certificate` window



```
(defun invariant-c5 (x)
  (member x (<0-a> 5)))

(defun mult-c5 (a b)
  (mod (+ a b) 5))

(defun inv-c5 (a)
  (mod (- 5 a) 5))

(defun nullel-c5 ()
  0)


(defthm nullel-close-c5
   (invariant-c5 (nullel-c5)))

(defthm nullel-prop-1-c5
   (implies (invariant-c5 a)
            (equal (mult-c5 a (nullel-c5))
                   a)))

(defthm nullel-prop-2-c5
```

```
MOD and primitive type reasoning.

Q.E.D.

Summary
Form:  ( DEFTHM ABELIAN-C5 ...)
Rules: ((:DEFINITION INVARIANT-C5)
        (:DEFINITION MEMBER)
        (:DEFINITION MULT-C5)
        (:EXECUTABLE-COUNTERPART <0-A>)
        (:EXECUTABLE-COUNTERPART BINARY-+)
        (:EXECUTABLE-COUNTERPART CAR)
        (:EXECUTABLE-COUNTERPART CDR)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART MOD)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  Subsume and Non-rec
Time:  0.54 seconds (prove: 0.00, print: 0.54, other:
0.00)
 ABELIAN-C5
ACL2 !>
```

Figure 4.24: Verification of the correctness of the cyclic group C_5

When, the user selects a cyclic group with the `Add` button and subsequently presses the `Build` button the system invokes our framework with a `generate-code` OpenMath object which has as argument the cyclic group. From the request, a `code` object with the definition of the four functions which determine the group (that is the functions `invariant`, `mult`, `inv` and `nullel` described in Subsection 4.4.2 with the corresponding suffix) and the ACL2 theorems which state that these functions satisfy the abelian group properties are generated. The *fKenzo* GUI extracts the code and writes it in the left side of the `ACL2` tab as can be seen in Figure 4.24.

Finally, if the user sends the script to ACL2 with the button `send-to-acl2`, a proof trial is automatically generated and finally the proof of the correctness of the implementation of the cyclic group is obtained in the right side of the `ACL2` tab (see Figure 4.24).

It is worth noting that in this case ACL2 obtains a proof without the help of the user, a situation that does not usually happens. More interesting interactions between

Kenzo and ACL2 will be explained in the next chapters.

This is a simple example of the integration of Kenzo, GAP/HAP and ACL2 where each system has a concrete aim. If we gather the composability of systems presented in this subsection with the one presented in Section 4.2 we can claim that Kenzo, GAP/HAP and ACL2 work together to provide a powerful and reliable tool thanks to the *fKenzo* system.

4.5 Methodology to integrate a new internal server in our system

Throughout this chapter we have presented how to integrate both GAP/HAP (see Section 4.1) and ACL2 (see Section 4.3) in our system. The reader can notice that the same process is followed in both cases, so we can extrapolate a methodology to integrate any Computer Algebra system or Theorem Prover tool in our system.

Let us suppose that we are interested in integrating some functionality of a system called *CAS-or-TP*, it does not really matters if it is a Computer Algebra system or a Theorem Prover since the way of integrating them is analogous.

First of all, we must include the *CAS-or-TP* system as a new internal server of our framework. To that aim, we define a new plug-in which references, at least, the following resources.

```
<code id="CAS-or-TP">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> cas-or-tp-is.lisp </data>
   <data format="Kf/microkernel"> cas-or-tp-m.lisp </data>
   <data format="Kf/adapter"> cas-or-tp-a.lisp </data>
</code>
```

We want to introduce some functionality in our framework which allows us to interact with some of the procedures of the *CAS-or-TP* system. Therefore, we must extend the XML-Kenzo specification to represent the requests and results related to the *CAS-or-TP* system. Then, we define in the XML-Kenzo specification the necessary new elements.

Subsequently, we include the *CAS-or-TP* system as a new internal server following the same method explained in both the Kenzo (see Subsection 2.2.2) and ACL2 (see Subsubsection 4.3.3.2) cases. To be more concrete, we have the *CAS-or-TP* system wrapped with an XML-Kenzo interface and the communication between the microkernel and the *CAS-or-TP* system is performed by means of XML-Kenzo requests through an external interface which offers the available services of our *CAS-or-TP* internal server. This component is included in the file `cas-or-tp-is.lisp`.

The `cas-or-tp-m.lisp` file defines new modules for the microkernel related to the

CAS-or-TP system. The procedures implemented in these modules depend on the functionality which is provided by the *CAS-or-TP* internal server. If several new modules are included in the microkernel, it is better to devote a concrete file per each one of them.

Finally, the `cas-or-tp-a.lisp` file contains the functions (new parsers) to raise the functionality of the adapter to be able to convert from OpenMath objects, devoted to ask requests and return results related to the *CAS-or-TP* internal server, to XML-Kenzo requests.

In addition, if we want to be able to interact with the new functionality by means of the *fKenzo* GUI, we must define a new module which references three files: `cas-or-tp-structure` (that defines the structure of the graphical constituents), `cas-or-tp-functionality` (which provides the functionality related to the new graphical constituents) and the plug-in introduced previously.

In this way, different tools can be integrated in our system as new internal servers.

Chapter 5

New certified Kenzo modules

At this moment the Kenzo system keeps on growing. Several researchers are developing new functionalities for it; for instance, spectral sequences [RRS06], resolutions [RER09], Koszul complexes [RS06] and so on.

An important point, that was already introduced in the previous chapter (see Section 4.4), in the development of new Kenzo modules is the verification of their correctness. The Kenzo system is a research tool that has got some relevant results not confirmed nor refuted by neither theoretical or computational means. Then, the question of Kenzo reliability (beyond testing) naturally arises.

In this chapter three new Kenzo modules (developed by us), their integration in the *fKenzo* system and some remarks about their verification in the ACL2 theorem prover are presented. Section 5.1 is devoted to describe our development related to simplicial complexes, a basic notion in Algebraic Topology which can be used to study digital images. Namely, we can associate a simplicial complex with a digital image and then study properties of the image by means of topological invariants (such as homology groups) of the associated simplicial complex. The framework to study digital images by means of simplicial complexes is explained in Section 5.2. Finally, explanations about the construction of the effective homology of the pushout of simplicial sets with effective homology are given in Section 5.3.

5.1 Simplicial complexes

The most elementary method to settle a connection between common "topology" and Algebraic Topology is based on the usage of simplicial complexes. The notion of topological space is too "abstract" in order to transfer it to machine universe. Simplicial complexes provide a purely combinatorial description of topological spaces which admit a triangulation. The computability of properties, such as homology groups, from a finite simplicial complex associated with a topological space is well-known and, for instance in the case of homology groups the algorithm uses simple linear algebra [Veb31]. Then, an algebraic topologist can identify a compact triangulable topological space with a finite simplicial complex, making computations easier.

Simplicial complexes are not included in the current www-available version of Kenzo. We have undertaken the task of deploying a new certified Kenzo module to work with them. In addition, we have also enhanced the *fKenzo* system to deal with simplicial complexes.

The rest of this section is organized as follows. Subsection 5.1.1 introduces the basic background about simplicial complexes and some algorithms about them; in Subsection 5.1.2 the new Kenzo module about simplicial complexes is presented; Subsection 5.1.3 explains how our framework is extended to include the functionality about simplicial complexes; moreover, the way of widening the *fKenzo* GUI to include simplicial complexes is detailed in Subsection 5.1.4. Finally, some aspects of the verification of the simplicial complexes programs are presented in Subsection 5.1.5.

5.1.1 Mathematical concepts

We briefly provide in this subsection the minimal mathematical background about simplicial complexes. We mainly focus on definitions. Many good textbooks are available for both these definitions and results about them, for instance [Mau96].

Let us start with the basic terminology. Let V be an ordered set, called the *vertex set*. An *(ordered abstract) simplex* over V is any ordered finite subset of V. An *(ordered abstract) n-simplex* over V is a simplex over V whose cardinality is equal to $n+1$. Given a simplex α over V, we call *faces* of α to all the subsets of α.

Definition 5.1. An *(ordered abstract) simplicial complex* over V is a set of simplexes $\mathcal{K}$ over V such that it is closed by taking faces (subsets); that is to say:

$$\forall \alpha \in \mathcal{K},\ if\ \beta \subseteq \alpha \Rightarrow \beta \in \mathcal{K}.$$

Let $\mathcal{K}$ be a simplicial complex. Then the set $S_n(\mathcal{K})$ of n-simplexes of $\mathcal{K}$ is the set made of the simplexes of cardinality $n+1$ of $\mathcal{K}$.

Example 5.2. Let us consider $V = (0, 1, 2, 3, 4, 5, 6)$.

The small simplicial complex drawn in Figure 5.1 is mathematically defined as the object:

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

Note that, because the vertex set is ordered the list of vertices of a simplex is also ordered, which allows us to use a sequence notation $(\dots)$ and not a subset notation

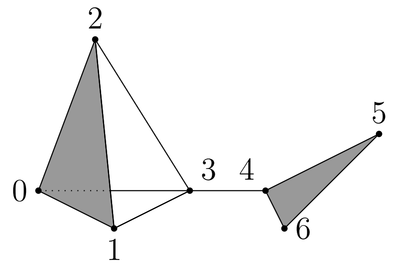


Figure 5.1: Butterfly simplicial complex

$\{\ldots\}$ for a simplex and also for the vertex set V. It is also worth noting that simplicial complexes can be infinite. For instance if $V = \mathbb{N}$ and the simplicial complex $\mathcal{K}$ is $\{(n)\}_{n\in\mathbb{N}} \cup \{(n-1,n)\}_{n\geq 1}$, the simplicial complex obtained can be seen as an infinite bunch of segments.

Definition 5.3. A *facet* of a simplicial complex $\mathcal{K}$ over V is a maximal simplex with respect to the subset relation, $\subseteq$, among the simplexes of $\mathcal{K}$.

Example 5.4. The facets of the small simplicial complex depicted in Figure 5.1 are: $\{(0,3),(1,3),(2,3),(3,4),(0,1,2),(4,5,6)\}$

Let us note that a *finite* simplicial complex can be generated from its facets taking the set union of the power set of each one of their facets. In general, we have the following definition.

Definition 5.5. Let $\mathcal{S}$ be a finite sequence of simplexes, then the set union of the power set of each one of the elements of $\mathcal{S}$ is, trivially, a simplicial complex called the *simplicial complex associated with* $\mathcal{S}$.

It is worth noting that the same simplicial complex can be generated from two different sequences of simplexes; in addition, the minimal sequence of simplexes which generates a finite simplicial complex is the sequence of its facets.

Then, the following algorithm can be defined.

Algorithm 5.6.
Input: a sequence of simplexes $\mathcal{S}$.
Output: the associated simplicial complex with $\mathcal{S}$.

Example 5.7. Let us show the way of generating the simplicial complex depicted in Figure 5.1 from its facets. Table 5.1 shows the faces of facets of the butterfly simplicial complex. If we perform the set union of all the faces, the desired simplicial complex is obtained.

In Subsection 1.1.2, we have defined the notion of simplicial set, a notion more complex than the notion of simplicial complex. Nevertheless, many common constructions

facet	faces
$(1,3)$	$\{\emptyset, (1), (3), (1,3)\}$
$(3,4)$	$\{\emptyset, (3), (4), (3,4)\}$
$(0,3)$	$\{\emptyset, (0), (3), (0,3)\}$
$(2,3)$	$\{\emptyset, (2), (3), (2,3)\}$
$(0,1,2)$	$\{\emptyset, (0), (1), (2), (0,1), (0,2), (1,2), (0,1,2)\}$
$(4,5,6)$	$\{\emptyset, (4), (5), (6), (4,5), (5,6), (4,6), (4,5,6)\}$

Table 5.1: Faces of the facets of the Butterfly simplicial complex

in topology are difficult to make explicit in the framework of simplicial complexes. It soon became clear around 1950 that the notion of simplicial set is much better.

There exists a link between these two notions which will allow us to compute homology groups of a simplicial complex by means of a simplicial set.

Definition 5.8. Let $\mathcal{SC}$ be an (ordered abstract) simplicial complex over V. Then the *simplicial set* $K(\mathcal{SC})$ *canonically associated* with $\mathcal{SC}$ is defined as follows. The set $K^n(\mathcal{SC})$ is $S_n(\mathcal{SC})$, that is, the set made of the simplexes of cardinality $n+1$ of $\mathcal{SC}$. In addition, let $(v_0, \dots, v_q)$ be a q-simplex, then the *face* and *degeneracy* operators of the simplicial set $K(\mathcal{SC})$ are defined as follows:

$$\begin{aligned} \partial_i^q((v_0, \dots, v_i, \dots, v_q)) &= (v_0, \dots, v_{i-1}, v_{i+1}, \dots, v_q), \\ \eta_i^q((v_0, \dots, v_i, \dots, v_q)) &= (v_0, \dots, v_i, v_i, \dots, v_q). \end{aligned}$$

That is, the face operator ∂_i^q removes the vertex in the position i of a q-simplex, and the degeneracy operator η_i^q duplicates the vertex in the position i of a q-simplex.

The proof of the fact that $K(\mathcal{SC})$ is a simplicial set is quite easy.

Then, the above definition provides us the following algorithm.

Algorithm 5.9.
Input: a finite simplicial complex $\mathcal{SC}$.
Output: the simplicial set $K(\mathcal{SC})$ canonically associated with $\mathcal{SC}$.

Definition 5.10. Given a simplicial complex $\mathcal{SC}$, the *n-homology group* of $\mathcal{SC}$, $H_n(\mathcal{SC})$, is the n-homology group of the simplicial set $K(\mathcal{SC})$:

$$H_n(\mathcal{SC}) = H_n(K(\mathcal{SC})).$$

5.1.2 Simplicial complexes in Kenzo

In the current www-available version of Kenzo, the notion of simplicial complex is not included. Then, we have developed a new Common Lisp module to enhance the Kenzo system with this notion. Our programs (with about 150 lines) implement Algorithms 5.6 and 5.9.

The following lines are devoted to explain the essential part of these programs, describing the functions with the same format as in the Kenzo documentation [DRSS98].

First of all, let us note that the vertex set V in our programs is $\mathbb{N}$; besides, we represent an n-simplex as a strictly ordered list of $n+1$ natural numbers that represent the vertices of the simplex. For instance, the 2-simplex with vertices 0, 1 and 3 is represented as the list `(0 1 3)`. Moreover, a finite simplicial complex is represented, in our system, by means of a list of simplexes.

Our first program implements Algorithm 5.6, that is, the functions which generate a simplicial complex from a sequence of simplexes. The description of the main function in charge of this task is shown here:

simplicial-complex-generator ls *[Function]*

From a list of simplexes, `ls`, this function generates the associated simplicial complex, that is to say, another list of simplexes.

The second program implements Algorithm 5.9. It generates the simplicial set canonically associated with a simplicial complex as an instance of the `Simplicial-Set` Kenzo class (see Subsection 1.2.1). The main function is:

ss-from-sc *simplicial-complex* *[Function]*

Build an instance of the `Simplicial-Set` Kenzo class which represents the simplicial set canonically associated with a simplicial complex, *simplicial-complex*, see Definition 5.8, using some auxiliar functions which are necessary to define simplicial sets in Kenzo.

To provide a better understanding of the new tools, an elementary example of their use is presented now. Let us consider the (minimal) triangulation of the torus presented in Figure 5.2.

The facets of the torus of Figure 5.2 are all the triangles (2-simplexes) depicted in that figure. Therefore, in our program the facets of the torus are assigned to the variable `torus-facets` as follows:

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

From these facets, we can construct the torus simplicial complex $S^1 \times S^1$ with our program:

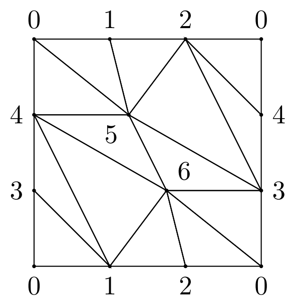


Figure 5.2: Torus triangulation

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

Once we have constructed this simplicial complex, we can build the simplicial set canonically associated with the torus simplicial complex by means of the instruction:

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

Finally, we can determine its homology groups thanks to the Kenzo kernel.

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

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

5.1.3 Integration of simplicial complexes

From the beginning of the development of our framework we strove for a system which could evolve with Kenzo. Therefore, to enhance our system with the functionality related to simplicial complexes we have developed a plug-in following the guidelines given in Subsubsection 3.1.2.

This new plug-in will allow us to construct from a sequence of simplexes $\mathcal{S}$ the simplicial set canonically associated with the simplicial complex defined from $\mathcal{S}$ (applying

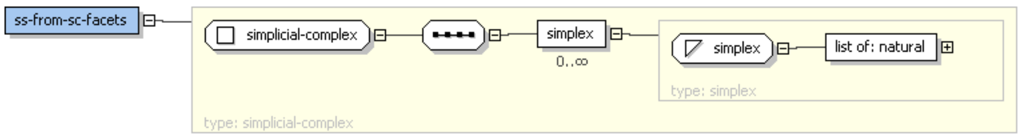


Figure 5.3: ss-from-sc-facets constructor in XML-Kenzo

Algorithms 5.6 and 5.9). The plug-in employed to include the functionality about simplicial complexes references the following resources:

```
<code id="simplicial-complexes">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> simplicial-complexes.lisp </data>
   <data format="Kf/microkernel"> simplicial-complexes-m.lisp </data>
   <data format="Kf/adapter"> simplicial-complexes-a.lisp </data>
</code>
```

As we claimed in Subsubsection 3.1.2, if we want to include new functionality related to Kenzo in our framework, all its components must be broadening. Let us explain each one of the referenced resources.

First of all, as we want to introduce a new kind of objects in our system (the simplicial sets associated with simplicial complexes generated from a sequence of simplexes), it is necessary to provide a representation for that kind of objects in our framework. Therefore, we have extended the XML-Kenzo specification (`XML-Kenzo.xsd` file) to admit the new objects related to simplicial complexes. In this specification, we have defined both the simple type called `simplex` which represents a list of natural numbers and the type called `simplicial-complex` which represents a sequence of `simplex` elements; and a new element: `ss-from-sc-facets` (see Figure 5.3), whose value is an element whose type is `simplicial-complex`. The `ss-from-sc-facets` element is defined as an element of the `SS` type; so, it can be used as any other element of this group.

As we have explained in Subsection 5.1.2 a simplex is implemented in our programs as an ordered list of natural numbers; however, in the XML-Kenzo specification we can only constrain the value of the `simplex` type to be a list of natural numbers. Therefore, the programs included in the microkernel will be in charge of checking the restriction of being ordered; here, we have an example of a functional dependency of a compound argument. As we explained in Subsubsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified to include the new elements, the external server is able to check the restrictions against the XML-Kenzo specification of requests such as:

```
<constructor>
    <ss-from-sc-facets>
        <simplex>0 1 2</simplex>
        <simplex>3 4 5</simplex>
    </ss-from-sc-facets>
</constructor>
```

The `simplicial-complexes.lisp` file includes the functionality explained in Subsection 5.1.2 to extend the Kenzo system. Moreover, this file adds code that enhances the `xml-kenzo-to-kenzo` function of the internal server to process the construction of objects from the `ss-from-sc-facets` XML-Kenzo construction requests. For instance, if the internal server receives the above request, the following instruction is executed in the Kenzo kernel.

```
(ss-from-sc (simplicial-complex-generator '((0 1 2) (3 4 5))))
```

As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned using an `id` XML-Kenzo object.

The `simplicial-complexes-m.lisp` file defines a new construction module for the microkernel called `simplicial-complex`. The procedure implemented in this module follows the guidelines explained in Subsubsection 2.2.3.3. In this case, the procedure implemented in this module checks that the values of `simplex` elements of a `ss-from-sc-facets` request are ordered lists since this constraint cannot be imposed in the XML-Kenzo specification (remember that functional dependencies cannot be defined in the XML-Kenzo specification), and therefore is not checked in the external server. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the new `simplicial-complex` module.

Finally, we have extended the `SS` Content Dictionary by means of the definition of a new object: `ss-from-sc-facets`. Therefore, the `simplicial-complexes-a.lisp` file raises the functionality of the adapter, which is now able to convert from the new OpenMath objects, devoted to simplicial complexes, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the XML-Kenzo request presented previously is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="ss-from-sc-facets"/>
      <OMA>
        <OMS name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>2</OMI>
      </OMA>
      <OMA>
        <OMS name="simplex"/>
          <OMI>3</OMI><OMI>4</OMI><OMI>5</OMI>
      </OMA>
   </OMA>
</OMOBJ>
```

5.1.4 Integration of simplicial complexes in the *fKenzo* GUI

The current subsection is devoted to present the necessary resources to extend the *fKenzo* GUI in order to handle simplicial complexes.

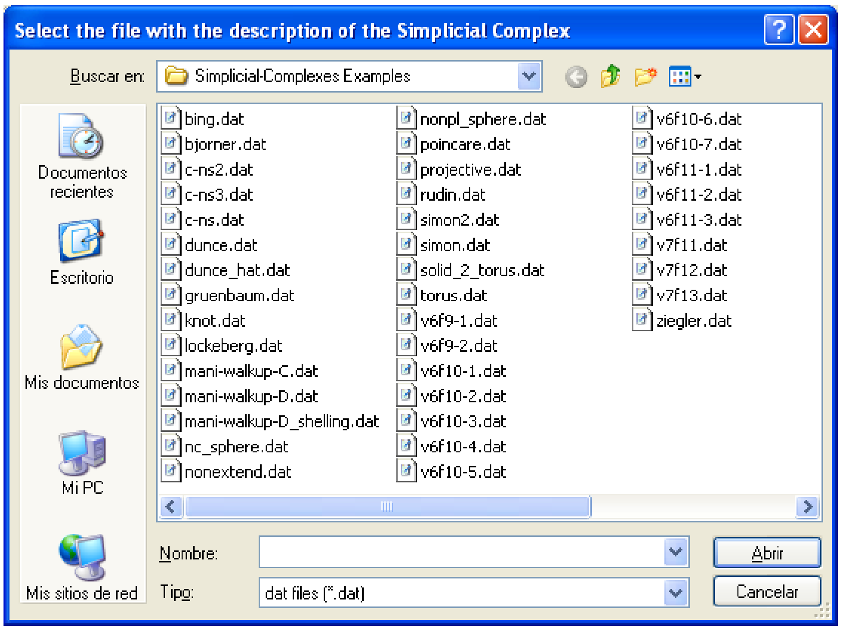
As we presented in Section 3.2 one of the modules which customizes *fKenzo* is the *Simplicial Set* module. This module contains the elements that represent simplicial set constructors of Kenzo: options to construct spaces from scratch (spheres, Moore spaces, finite simplicial sets, and so on) and from other spaces (for instance, cartesian products). We have enhanced this module by means of a new constructor related to simplicial complexes.

The original *Simplicial Set* module referenced two files: `simplicial-sets-structure` (which defined the structure of the graphical constituents of the module) and `simplicial-sets-functionality` (which provided the functionality related to the graphical constituents). Both files have been upgraded in order to tackle the use of simplicial complexes in the *fKenzo* GUI. Moreover, the new *Simplicial Set* module also references the plug-in described in the previous subsection. In this way, when the *Simplicial Set* module is loaded, the whole *fKenzo* system is ready to allow the construction of simplicial sets coming from simplicial complexes.

We have defined three new graphical elements, using the XUL specification language, in the `simplicial-sets-structure` file:

- A new menu option called `Load Simplicial Complex` included in the `Simplicial Sets` menu.
- A window called `Load Simplicial Complex` (see Figure 5.4).
- A window called `SS-from-SC-name` (see Figure 5.5).

The functionality stored in the `simplicial-sets-functionality` document related

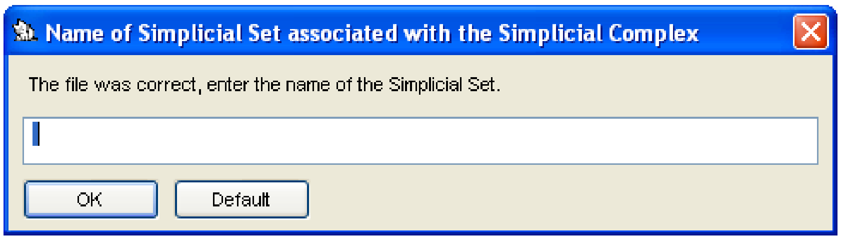


bing.dat
bjorner.dat
c-ns2.dat
c-ns3.dat
c-ns.dat
dunce.dat
dunce_hat.dat
gruenbaum.dat
knot.dat
lockeberg.dat
mani-walkup-C.dat
mani-walkup-D.dat
mani-walkup-D_shelling.dat
nc_sphere.dat
nonextend.dat
nonpl_sphere.dat
poincare.dat
projective.dat
rudin.dat
simon2.dat
simon.dat
solid_2_torus.dat
torus.dat
v6f9-1.dat
v6f9-2.dat
v6f10-1.dat
v6f10-2.dat
v6f10-3.dat
v6f10-4.dat
v6f10-5.dat
v6f10-6.dat
v6f10-7.dat
v6f11-1.dat
v6f11-2.dat
v6f11-3.dat
v7f11.dat
v7f12.dat
v7f13.dat
ziegler.dat

Figure 5.4: `Load Simplicial Complex` window

to these components works as follows. A function acting as event handler is associated with the `Load Simplicial Complex` menu option; this function shows the window `Load Simplicial Complex` (see Figure 5.4) which allows the user to choose a file which contains a list of simplexes. It is worth noting that the user does not introduce each one of the simplexes of the sequence manually: he selects a file with that information (a file which can be either generated by a computer program or build manually with a text editor). Currently, a folder, called *Simplicial-Complexes Examples*, with several examples of simplicial complexes defined from their a sequence of simplexes is included in the distribution of *fKenzo*. At this moment it contains 39 examples such as the torus, the projective space or the duncehat (several of those examples were extracted from [Hac01] keeping its original format, which is the one used in our system).

Once the user has selected a file, the system processes the file to construct an `ss-from-sc-facets` OpenMath request with the sequence of simplexes obtained from the file. Subsequently, the framework constructs the simplicial set associated with the simplicial complex defined from the sequence of simplexes and returns its identification number (an `id` XML-Kenzo object). Afterwards, the system asks a name for the new simplicial set by means of the window `SS-from-SC-name` (see Figure 5.5); if the name given by the user is correct (was not used previously), the system stores it, otherwise it indicates that the name was previously used and asks again a name. Eventually, when the user has given a valid name, the system builds a new `FKENZO-OBJECT-NAME` instance (see Subsubsection 4.1.5.1) and, also, adds the new object to the list of constructed spaces (located at the left side of the main tab of the *fKenzo* GUI). Figure 5.6 shows the control and navigation submodel (with the Noesis notation) describing the construction of the simplicial set canonically associated with a simplicial complex from


Name of Simplicial Set associated with the Simplicial Complex

The file was correct, enter the name of the Simplicial Set.

Figure 5.5: *fKenzo* asking a name for a simplicial set coming from a simplicial complex

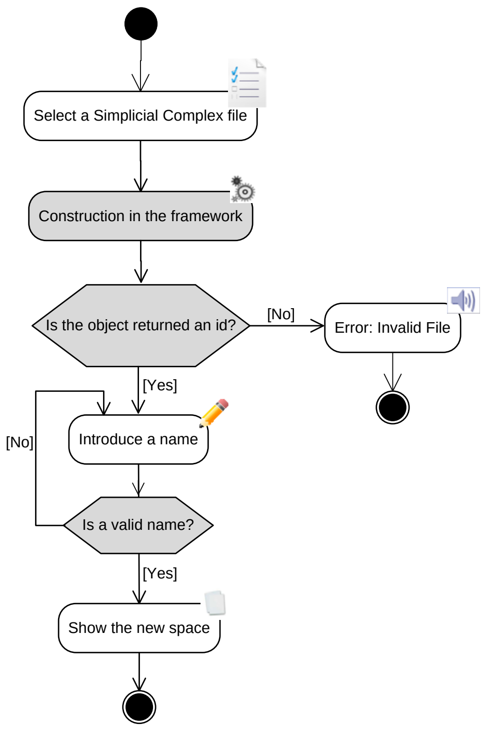

Select a Simplicial Complex file
Construction in the framework
Is the object returned an id?
[No]
Error: Invalid File
[Yes]
[No]
Introduce a name
Is a valid name?
[Yes]
Show the new space

Figure 5.6: Construction of a simplicial set from a simplicial complex in *fKenzo*

the `Load Simplicial Complex` menu option.

5.1.4.1 Execution flow of the *fKenzo* GUI for simplicial complexes

In order to clarify the execution flow followed by *fKenzo*, let us retake the example presented in Subsection 5.1.2, that is, the computation of the first homology groups of the torus. When the Simplicial Set module is loaded in *fKenzo*, the option *Load Simplicial Complex* is available in the *Simplicial Sets* menu. From this option, the user can select the file which contains the facets of the torus; this file is called `torus.dat` and is located at the folder *Simplicial-Complexes Examples*, see Figure 5.4. An extract of the data stored in that file is:

```
#
0 1 3
0 1 5
...
```

which corresponds with the facets of the triangulation of the torus of Figure 5.2. From that file, *fKenzo* constructs the following OpenMath request:

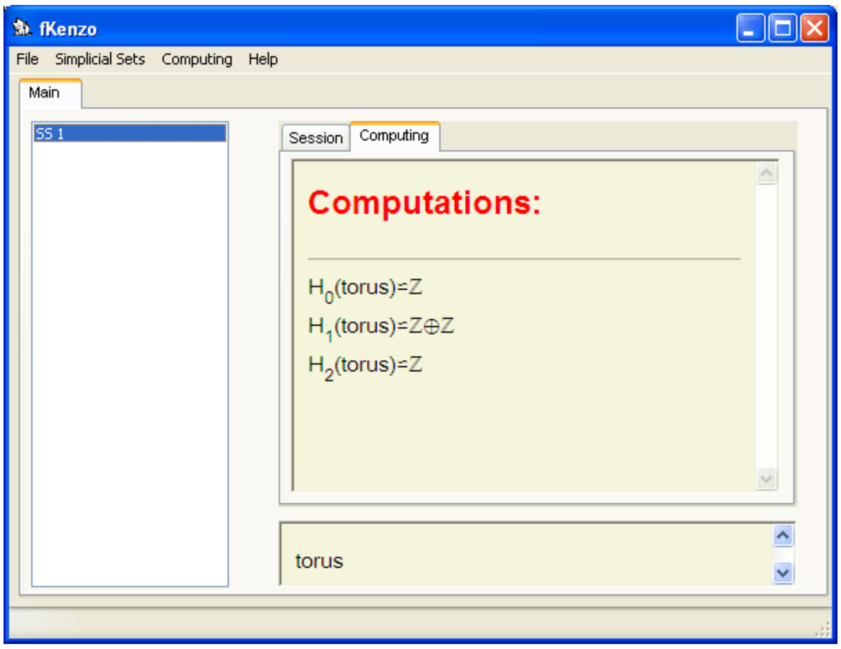
```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="ss-from-sc-facets"/>
      <OMA>
        <OMS cd="SS" name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>3</OMI>
      </OMA>
      <OMA>
        <OMS cd="SS" name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>5</OMI>
      </OMA>
      ...
   </OMA>
</OMOBJ>
```

which is sent to the framework. From this request, the simplicial set associated with the torus simplicial complex is built and its identification number is returned. Then, *fKenzo* asks the user a name for the object (in this case the given name is “torus”). Subsequently, the new object is added to the list of constructed spaces which is shown in the left side of the *fKenzo* GUI. When, the user selects this object, the name that the user provided previously is shown (see Figure 5.7).

Finally, the user can ask *fKenzo* to compute the homology groups of the torus using the `homology` option of the *Computing* menu, the results are shown, as usual, in the Computing tab, see Figure 5.7.

5.1.5 Formalization of Simplicial Complexes in ACL2

As we said at the beginning of this chapter, we are interested not only in developing new tools for the Kenzo system but also in verifying the correctness of these new tools. This subsection is devoted to present the certification of the correctness of our implementation of Algorithm 5.6. The certification of the correctness of the implementation of Algorithm 5.9 belongs to a more general case (the certification of the implementation of Kenzo simplicial sets) and we will cope with it in the next chapter.



Computations:

H_0(torus)=Z

H_1(torus)=Z⊕Z

H_2(torus)=Z

torus

Figure 5.7: Homology groups of the torus

5.1.5.1 Main definitions and properties

As we have just said, we want to formalize in ACL2 the correctness of the `simplicial-complex-generator` function; that is to say, our implementation of Algorithm 5.6. Since both Kenzo and ACL2 are Common Lisp programs we can verify the correctness of the `simplicial-complex-generator` function in ACL2.

First of all, let us present the definition of the `simplicial-complex-generator` function. This program can be decomposed in three steps. First of all, we have defined the `powerset` function which generates the powerset of a simplex.

```
(defun map-cons (x s)
  (if (endp s)
      nil
    (cons (cons x (car s)) (map-cons x (cdr s)))))

(defun powerset (l)
  (if (endp l)
      (list nil)
    (append (powerset (cdr l)) (map-cons (car l) (powerset (cdr l))))))
```

Subsequently, we have defined the `simplicial-complex-generator-aux` function which builds a list gathering the powerset of every simplex of a sequence of simplexes `ls`.

```
(defun simplicial-complex-generator-aux (ls)
  (if (endp ls)
      nil
    (append (powerset (car ls)) (simplicial-complex-generator-aux (cdr ls)))))
```

Eventually, we have implemented the `simplicial-complex-generator` function which removes the duplicates elements (thanks to the ACL2 function `remove-duplicates-equal`) of the list generated by `simplicial-complex-generator-aux`, getting the looking for simplicial complex.

```
(defun simplicial-complex-generator (ls)
  (remove-duplicates-equal (simplicial-complex-generator-aux ls)))
```

This design follows simple recursive schemas, which are suitable for the induction heuristics of the ACL2 theorem prover.

From now on, we define the necessary functions to prove the correctness of our program. First of all, we need some auxiliary functions which define the necessary concepts to prove our theorems. These definitions are based on both Algorithm 5.6 and Definition 5.1. Namely, we need to define the notions of *simplex*, *list of simplexes*, *set of simplexes*, *face* and *member* in ACL2.

As we said in Subsection 5.1.2, a *simplex* in our programs is a strictly ordered list of natural numbers. The `simplex-p` function is a predicate that given a list `list` returns `t` if the list represents a simplex and `nil` otherwise.

```
(defun simplex-p (list)
  (if (endp list)
      (equal list nil)
    (if (endp (cdr list))
        (and (equal (cdr list) nil) (natp (car list)))
      (and (natp (car list)) (natp (cadr list)) (< (car list) (cadr list))
           (simplex-p (cdr list))))))
```

From this notion of simplex, we can easily define the notion of *list of simplexes* by means of the `list-of-simplex-p` predicate which returns `t` if its argument is a list of simplexes and `nil` otherwise:

```
(defun list-of-simplexes-p (ls)
  (if (endp ls)
      (equal ls nil)
    (and (simplex-p (car ls)) (list-of-simplexes-p (cdr ls)))))
```

A *set of simplexes* is a *list of simplexes* without duplicate elements. This is modeled with the `set-of-simplexes-p` predicate (a test function, called `without-duplicates-p`, that checks if a list does not have duplicate elements has been defined as auxiliar function).

```
(defun set-of-simplexes-p (ls)
  (and (list-of-simplexes-p ls) (without-duplicates-p ls)))
```

Finally, to define the relations "be a face of" (between two simplexes) and "be in" (between a simplex and a set of simplexes or between a simplex and a list of simplexes) we have used two already defined ACL2 functions that are `subsetp-equal` and `member-equal` respectively.

Therefore, we have the framework to prove the correctness of our programs. The following theorems state the basic properties that the `simplicial-complex-generator` function must satisfy. First of all, we prove that `simplicial-complex-generator` constructs a simplicial complex. Therefore, we need to prove the following two ACL2 lemmas.

ACL2 Lemma 5.11. Let ls be a list of simplexes, then (`simplicial-complex-generator` ls) builds a set of simplexes.

```
(defthm simplicial-complex-generator-constructs-simplicial-complex-1
  (implies (list-of-simplexes-p ls)
           (set-of-simplexes-p (simplicial-complex-generator ls))))
```

ACL2 Lemma 5.12. Let x be a simplex and ls be a list of simplexes, if x belongs to (`simplicial-complex-generator` ls) and y is a face of x, then y belongs to (`simplicial-complex-generator` ls).

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

Once we have proved these two theorems we can claim that the `simplicial-complex-generator` function constructs an abstract simplicial complex when a list of simplexes is provided as argument.

In addition, we need to prove that the simplicial complex constructed by the `simplicial-complex-generator` function is the one that we are looking for. This means that we need to prove the following lemma.

ACL2 Lemma 5.13. Let `ls` be a list of simplexes and let `s` be an element of the simplicial complex constructed with the `simplicial-complex-generator` function taking as argument `ls`; then, `s` is a face of some of the simplexes of `ls`.

```
(defthm simplicial-complex-generator-correctness
  (implies (and (list-of-simplexes-p ls)
                (member-equal s (simplicial-complex-generator ls))
           (face-of-some-p s ls)))
```

The proof of the above lemmas, in spite of involving some auxiliary results, can be proved without any special hindrance due to the fact that, as we said previously, our programs follow simple inductive schemas that are suitable for the ACL2 heuristics. Then, we have the following theorem.

ACL2 Theorem 5.14. Let ls be a list of simplexes, then (`simplicial-complex-generator` ls) constructs the simplicial complex associated with ls.

The interested reader can consult the complete development in [Her11].

5.1.5.2 Two equivalent programs

The implementation of the `simplicial-complex-generator` function is suitable for the induction heuristics of the ACL2 theorem prover. However, it is an inefficient design, so, it can produce undesirable situations. For instance, if we try to build a simplicial complex from a list of 11613 simplexes, an error message will be shown:

```
> (simplicial-complex-generator ...) ✠
Error: Stack overflow (signal 1000)
[condition type: SYNCHRONOUS-OPERATING-SYSTEM-SIGNAL]
```

This kind of error occurs when too much memory is used on the data structure that stores information about the active computer program.

In order to overcome this drawback, an efficient algorithm called `optimized-simplicial-complex-generator` has been implemented. This new program relies on the *memoization* technique. Let us remember that memoization is used primarily to speed up computer programs. A memoized function "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it.

However, the `optimized-simplicial-complex-generator` program can not be implemented in ACL2 (remember that ACL2 is an applicative subset of Common Lisp). In order to deal with this pitfall we have based on the work presented in [ALR07], where the authors coped with a similar problem, but related to already implemented Kenzo code fragments.

Let us enumerate the characteristics of our situation:

- `simplicial-complex-generator` program is
  - specially designed to be proved;
  - programmed in ACL2 (and, of course, Common Lisp);
  - not efficient;
  - tested;
  - proved in ACL2.
- `optimized-simplicial-complex-generator` program is
  - specially designed to be efficient;
  - written in Common Lisp;
  - efficient;
  - tested;
  - unproved.

In our approach, `simplicial-complex-generator` is *supposed to be equivalent* to `optimized-simplicial-complex-generator`. But we do not pretend to prove this equivalence: this option would lead us to a form of ill-founded recursion. Our aim should be to use the *highly reliable* `simplicial-complex-generator` to perform automated testing of the *efficient* `optimized-simplicial-complex-generator`.

The following toy program will illustrate this idea:

```
(defun automated-testing ()
  (let ((cases (generate-test-cases 100000)))
    (dolist (case cases)
      (if (not (equal-as-sc (simplicial-complex-generator case)
                   (optimized-simplicial-complex-generator case)))
          (report-on-failure case)))))
```

With this intensive testing, it is hoped that `simplicial-complex-generator` accurately models `optimized-simplicial-complex-generator`, and then our strategy could be safely applied.

A really interesting work, in the same line, was presented [G$^+$08], where a method to permit the user of a mathematical logic to write elegant logical definitions while allowing sound and efficient execution was described. Those features afford dual applications: on the one hand, formal proof; on the other hand, execution. In particular, they allow the user to install, in a logically sound way, alternative executable counterparts for logically-defined functions. These alternatives are often much more efficient than the logically equivalent terms they replace. Unfortunately, in order to use the tool presented in [G$^+$08] both programs must be developed in ACL2, which is not our case.

5.2 Applications of simplicial complexes: Digital Images

Algebraic Topology is a complex and abstract mathematical subject; however, some of its techniques can be applied to different contexts such as coding theory [Woo89], data analysis [Her03], robotics [Mac03] or digital image analysis [GDMRSP05, GDR05] (in this last case, in particular in the study of medical images [SGF03]).

Here, we are going to focus on the application of Algebraic Topology to the study of binary digital images. In the Algebraic Topology framework, binary digital images can be studied using simplicial complexes. This section is devoted to present a technique based on simplicial complexes to study binary digital images. In particular, we study monochromatic (black and white) digital images by means of the algorithms related to simplicial complexes explained in Subsection 5.1.1 and new algorithms explained later.

Explanations about how we use simplicial complexes to study digital images are given in the rest of this section. Subsection 5.2.1 presents the technique and the algorithms that we apply to analyze digital images by means of simplicial complexes. The algorithms presented in Subsection 5.2.1 are implemented as an enhancement of the simplicial complex Kenzo module (presented in Subsection 5.1.2) in Subsection 5.2.2 for the cases of 2D and 3D digital images. The way of widening our framework and the *fKenzo* GUI to include digital images is explained in Subsection 5.2.3 and 5.2.4 respectively. Finally, some remarks about the formalization of our algorithms for digital images are provided in Subsection 5.2.5.

5.2.1 The framework to study digital images

Let n be any positive integer. An *n-xel* q in an Euclidean n-space, $\mathbb{R}^n$, is a closed unit n-dimensional (hyper)cube $q \subset \mathbb{R}^n$ whose 2^n vertices have natural coordinates (more precisely, an *n-xel* in $\mathbb{R}^n$ is a cartesian product like $[i_1, i_1+1] \times [i_2, i_2+1] \times \ldots \times [i_n, i_n+1]$). In this memoir, a *pixel* is a 2-xel in $\mathbb{R}^2$. We define an *n-dimensional binary image* or *nD-image*, to be a finite set of n-xels in $\mathbb{R}^n$.

An nD-image $\mathcal{I}$ can, of course, be represented by a finite n-dimensional array of 1's and 0's in which each 1 represents an n-xel in $\mathcal{D}$ and each 0 represents an n-xel that is not in $\mathcal{D}$. Let us focus on the study of nD-images by means of simplicial complexes. Firstly, we present the study for the cases of 2D-images and eventually the general case.

As we have just said, a 2D-image $\mathcal{D}$ can be represented by a finite 2-dimensional array of 1's and 0's in which each 1 represents a pixel in $\mathcal{D}$ and each 0 represents a pixel that is not in $\mathcal{D}$ (in a monochromatic 2D-image $\mathcal{D}$, black pixels are represented by 1's, on the contrary white pixels are represented by 0's).

Let $\mathcal{D}$ be a 2D-image codified as a 2-dimensional array of 1's and 0's. We want to associate a simplicial complex with $\mathcal{D}$. It is worth noting that the most natural and efficient approach to study digital images by topological means consists in using *cubical complexes*, see [KMM04]. However we have preferred to analyse digital images through simplicial complexes, because we can reuse the certified simplicial complex Kenzo module presented in Section 5.1.

From a digital image, there are several ways of constructing a simplicial complex (see [ADFQ03]). The approach that we have followed here consists in obtaining from $\mathcal{D}$ the facets of one of its associated simplicial complexes. Subsequently, applying Algorithm 5.6 (the algorithm which constructs a simplicial complex from a sequence of simplexes), we obtain a simplicial complex associated with $\mathcal{D}$.

The process that we have followed to obtain the facets from a 2D-image $\mathcal{D}$ is as follows. Let $V = (\mathbb{N}, \mathbb{N})$ be the vertex set, that is, a vertex, in this case, is a pair of natural numbers. Let $p = (a, b)$ be the coordinates of a pixel in $\mathcal{D}$ (that is, the position of the pixel in the 2-dimensional array associated with $\mathcal{D}$). From p we can obtain two 2-simplexes that are two facets of the simplicial complex associated with $\mathcal{D}$. Namely, from $p = (a, b)$ we obtain the following facets: the triangles $((a, b), (a+1, b), (a+1, b+1))$ and $((a, b), (a, b+1), (a+1, b+1))$. If we repeat the process for the coordinates of all the pixels in $\mathcal{D}$, we obtain the facets of a simplicial complex associated with $\mathcal{D}$, that will be denoted by $\mathcal{K}_{2D}(\mathcal{D})$.

Therefore, we can define the following algorithm.

Algorithm 5.15.
Input: a 2D-image $\mathcal{D}$ represented by means of a 2-dimensional array of 1's and 0's.
Output: the facets of $\mathcal{K}_{2D}(\mathcal{D})$, a simplicial complex associated with $\mathcal{D}$.

Example 5.16. Consider the 2D-image depicted in the left side of Figure 5.8. This image can be codified by means of the 2-dimensional array: $((1, 0), (0, 1))$, then, the coordinates of the black pixels are $(0, 0)$ and $(1, 1)$. Therefore, applying Algorithm 5.15 we obtain the facets of $\mathcal{K}_{2D}(\mathcal{D})$:

$$(((0,0),(0,1),(1,1)),((0,0),(1,0),(1,1)),((1,1),(1,2),(2,2)),((1,1),(2,1),(2,2))).$$

Once we have the simplicial complex associated with the digital image, we can compute the homology groups of the image from the simplicial complex. As we said previously, several simplicial complexes can be associated with a digital image, but all of them are homeomorphic (see [ADFQ03]); then, we can define the homology groups of a 2D-image as follows:

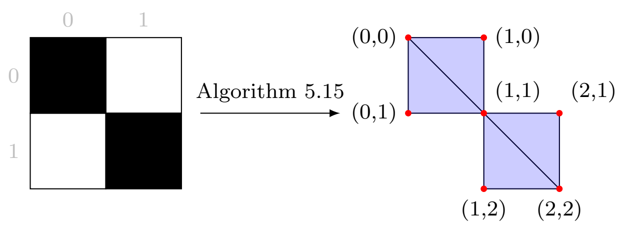


Figure 5.8: On the left, a digital image; on the right, its simplicial complex representation

Definition 5.17. Given a 2D-image $\mathcal{D}$, the *n-homology group* of $\mathcal{D}$, $H_n(\mathcal{D})$ is the n-homology group of the simplicial complex $\mathcal{K}_{2D}(\mathcal{D})$:

$$H_n(\mathcal{D}) = H_n(\mathcal{K}_{2D}(\mathcal{D})).$$

Subsequently, we can interpret properties about the digital image from its homology groups. 2D-images are embedded in $\mathbb{R}^2$ then its homology groups vanish for dimensions greater than 2 and they are torsion-free from dimensions 0 to dimension 1; that is, their homology groups are either null or a direct sum of $\mathbb{Z}$ components in dimensions 0 and 1. The number of $\mathbb{Z}$ components of the homology groups of dimension 0 and 1 measures respectively the number of connected components and the number of holes of the image.

The method presented here for 2D-images can be generalized to nD-images with $n \geq 2$. An nD-image can be represented by a finite n-dimensional array of 1's and 0's in which each 1 represents an n-xel in $\mathcal{D}$ and each 0 represents an n-xel that is not in $\mathcal{D}$.

Let $\mathcal{D}$ be an nD-image, from the coordinates of each n-xel in $\mathcal{D}$ (its position in the n-dimensional array associated with $\mathcal{D}$), we can obtain a triangulation by means of n-simplexes, see [OS03], which are facets of a simplicial complex associated with $\mathcal{D}$. If we repeat the process for the coordinates of all the n-xels in $\mathcal{D}$, we obtain the facets of a simplicial complex associated with $\mathcal{D}$. Then, applying Algorithm 5.6, we can obtain the simplicial complex associated with $\mathcal{D}$. Therefore, the two following algorithms can be defined.

Algorithm 5.18.
Input: the coordinates of an n-xel.
Output: a triangulation of the n-xel by means of n-simplexes.

Algorithm 5.19.
Input: an nD-image $\mathcal{D}$ represented by means of a n-dimensional array of 1's and 0's.
Output: the facets of $\mathcal{K}_{nD}(\mathcal{D})$, a simplicial complex associated with $\mathcal{D}$.

It is worth noting that these two last algorithms are not implemented for the general case, just for 2D-images and 3D-images.

5.2.2 Enhancing the simplicial complex Kenzo module

Algorithms 5.18 and 5.19 explained in Subsection 5.2.1 have been implemented for the cases $n = 2, 3$ as an enhancement for the simplicial complex Kenzo module explained in Subsection 5.1.2. The set of programs that we have developed (with about 600 lines) allows the construction of the facets of the simplicial complexes $\mathcal{K}_{2D}(\mathcal{D}_1)$ and $\mathcal{K}_{3D}(\mathcal{D}_2)$ from a 2D-image, $\mathcal{D}_1$, and a 3D-image, $\mathcal{D}_2$. Moreover, thanks to the simplicial complex module and the Kenzo kernel, we can construct the simplicial complex defined by the list of facets, the simplicial set associated with that simplicial complex, and, finally, compute the homology groups of the simplicial set obtaining properties of the original image.

The rest of this subsection is devoted to present the essential part of the programs which implement algorithms 5.18 and 5.19 for the cases $n = 2, 3$, describing the functions with the same format as in the Kenzo documentation [DRSS98]. Moreover, we will see some examples of the use of these programs.

We have written several functions that allow us to cope with 2D-images and 3D-images; but most of them are auxiliary functions; so we just describe the main ones:

generate-facets-image-2d 2da *[Function]*

This function takes as argument the 2-dimensional array (a 2-dimensional array is represented by means of a list of lists) of 1's and 0's, `2da`, which determines a 2D-image and returns the list of facets of the simplicial complex $\mathcal{K}_{2D}$ associated with the 2D-image (Algorithm 5.15). The facets are returned as a list of simplexes over $V = (\mathbb{N}, \mathbb{N})$.

generate-facets-image-3d 3da *[Function]*

This function takes as argument the 3-dimensional array (a 3-dimensional array is represented by means of a list of lists of lists) of 1's and 0's, `3da`, which determines a 3D-image and returns the list of facets of the simplicial complex $\mathcal{K}_{3D}$ associated with the 3D-image (Algorithm 5.19 case $n = 3$). The facets are returned as a list of simplexes over $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$.

transform-lolol-to-lol lolol *[Function]*

This function transforms a list of simplexes, `lolol`, over $V = (\mathbb{N}, \mathbb{N})$ or $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ to a list of simplexes over $V = \mathbb{N}$. This function is necessary because the programs related to simplicial complexes presented in Subsection 5.1.2 work with simplexes over $V = \mathbb{N}$ and the two above functions return list of simplexes over $V = (\mathbb{N}, \mathbb{N})$ and $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ respectively. Then, we need a transformation between the format of the output of the functions `generate-facets-image-2d` and `generate-facets-image-3d` to the format of the input of the `simplicial-complex-generator` function (defined in Subsection 5.1.2). The `transform-lolol-to-lol` function assigns a unique natural number to each

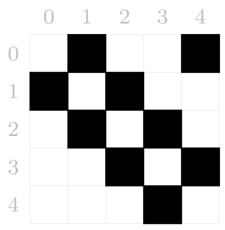


Figure 5.9: Small 2D-image

vertex over $V = (\mathbb{N},\mathbb{N})$ or $V = (\mathbb{N},\mathbb{N},\mathbb{N})$ of the list `lolol` (this function is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ and also from $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$). The natural numbers are assigned based on the lexicographical order of the lists of simplexes `lolol`. For instance, from the list of simplexes $(((0,0),(0,1),(1,1)),((0,0),(1,0),(1,1)))$ over $V = (\mathbb{N},\mathbb{N})$ (where we have the following ordination of the simplexes based on the lexicographical order: $(0,0) < (0,1) < (1,0) < (1,1)$) we obtain the list of simplexes $((0,1,3),(0,2,3))$ over $V = \mathbb{N}$.

To provide a better understanding of the new tools, some examples of their use are presented. Let us consider the 2D-image depicted in Figure 5.9. This image can be represented by the following 2-dimensional array (a list of lists) which is assigned to the variable `small-2d-image`:

```
> (setf small-2d-image '((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0)))
✠
((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0))
```

From the 2-dimensional array of the image we can generate the list of facets over $V = (\mathbb{N},\mathbb{N})$ of $\mathcal{K}_{2D}$(`small-2d-image`):

```
> (setf facets-image (generate-facets-image-2d small-2d-image)) ✠
(((1 0) (2 0) (2 1)) ((1 0) (1 1) (2 1)) ((4 0) (5 0) (5 1)) ((4 0) (4 1) (5 1))
 ((0 1) (1 1) (1 2)) ((0 1) (0 2) (1 2)) ((2 1) (3 1) (3 2)) ((2 1) (2 2) (3 2))
 ((1 2) (2 2) (2 3)) ((1 2) (1 3) (2 3)) ...)
```

Now, we can transform these facets to the suitable format for the `simplicial-complex-generator` function:

```
> (setf facets-image-nat (transform-lolol-to-lol facets-image)) ✠
((0 3 4) (0 1 4) (2 6 7) (2 3 7) (4 8 9) (4 5 9) (7 11 12) (7 8 12) (9 13 14)
 (9 10 14) ...)
```

Then, from these facets we can construct the associated simplicial complex:

```
> (setf image-2d-sc (simplicial-complex-generator facets-image-nat)) ✠
((0 3 4) (3 4) (0 4) (0 3) (4) (3) (0) (0 1 4) (1 4) (0 1) ...)
```

Subsequently, the simplicial set canonically associated with the simplicial complex `image-2d-sc` can be built:

```
> (setf image-2d-ss (ss-from-sc image-2d-sc)) ✠
[K1 Simplicial Set]
```

We obtain as result a `Simplicial-Set` object. Then, we can ask for its homology groups.

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

The results must be interpreted as stating that the image of Figure 5.9 has 2 connected components and 3 holes.

Analogously we can consider an example of a 3D-image, namely, the one depicted in Figure 5.10. This image can be represented by the following 3-dimensional array (a list of lists of lists) which is assigned to the variable `small-3d-image`:

```
> (setf small-3d-image '(((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 0 0 0) (0 0 0 0) (0 0 0 0) (0 0 0 0))) ✠
(((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0)) ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
 ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0)) ((0 0 0 0) (0 0 0 0) (0 0 0 0) (0 0 0 0))))
```

From the 3-dimensional array of an image we can generate the list of facets over $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ of $\mathcal{K}_{3D}$(`small-3d-image`):

```
> (setf facets-image-3d (generate-facets-image-3d small-3d-image)) ✠
(((1 0 0) (2 0 0) (2 0 1) (2 1 1)) ((1 0 0) (2 0 0) (2 1 0) (2 1 1))
 ((1 0 0) (1 0 1) (2 0 1) (2 1 1)) ((1 0 0) (1 0 1) (1 1 1) (2 1 1))
 ((1 0 0) (1 1 0) (2 1 0) (2 1 1)) ((1 0 0) (1 1 0) (1 1 1) (2 1 1))
 ((2 0 0) (3 0 0) (3 0 1) (3 1 1)) ((2 0 0) (3 0 0) (3 1 0) (3 1 1))
 ((2 0 0) (2 0 1) (3 0 1) (3 1 1)) ((2 0 0) (2 0 1) (2 1 1) (3 1 1)) ...)
```

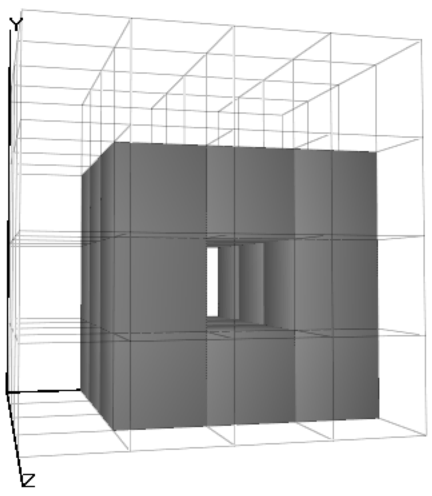


Figure 5.10: 3D cube with a tunnel

Now, we can transform these facets to the suitable format for the `simplicial-complex-generator` function:

```
> (setf facets-image-3d-nat (transform-lolol-to-lol facets-image-3d)) ✠
((0 16 17 21) (0 16 20 21) (0 1 17 21) (0 1 5 21) (0 4 20 21) (0 4 5 21) (16 32 33 37)
 (16 32 36 37) (16 17 33 37) (16 17 21 37) ...)
```

Then, from these facets we can construct the associated simplicial complex:

```
> (setf image-3d-sc (simplicial-complex-generator facets-image-3d-nat)) ✠
((0 16 17 21) (16 17 21) (0 17 21) (0 16 21) (0 16 17) (17 21) (16 21) (16 17)
 (0 21) ...)
```

Subsequently, the simplicial set canonically associated with the simplicial complex `image-3d-sc` can be built:

```
> (setf image-3d-ss (ss-from-sc image-3d-sc)) ✠
[K10 Simplicial Set]
```

We obtain as result a `Simplicial-Set` object. Then, we can ask for its homology groups.

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

The results must be interpreted as stating that the image of Figure 5.10 has 1 connected component, 1 tunnel and 0 cavities.

Up to now, we have presented the programs that, from the 2-dimensional array associated with a 2D-image or the 3-dimensional array associated with a 3D-image, obtain the facets of a simplicial complex associated with them. However, few common 2D-image formats (such as "*jpeg*", "*bmp*", "*png*", "*pbm*" and so on) or 3D-image formats (such as "*byu*", "*jvx*", "*obj*" and so on) encode images as 2-dimensional or 3-dimensional arrays respectively.

Namely, in the case of 2D-images, the only format which codifies a image as a 2 dimensional array is *pbm*. Then, we have implemented, as a Common Lisp program, an interpreter which converts from an image in the *pbm* format to the 2-dimensional array in the format of our programs (that is to say, a list of lists).

The *pbm* images that our program can process are codified in the following way:

- The two characters "P1" which indicate that we work with plain *pbm* files (there are other kinds of *pbm* files; but we only deal with the simpler one).
- The width (w) in pixels of the image and the height (h) in pixels of the image, formatted as ASCII characters in decimal.
- A *raster* of h rows, in order from top to bottom. Each row has w bits. Each bit represents a pixel: 1 is black, 0 is white. The order of the pixels is left to right.

For instance, the image depicted in Figure 5.9 is codified in a *pbm* file as follows.

```
P1
5 5
01001
10100
01010
00101
00010
```

The above code must be read as follows. We have a plain *pbm* file which has a raster of five rows and five columns of pixels. The pixels of the coordinates $(0,1)$, $(1,0)$, $(1,2)$, $(2,1)$, $(2,3)$, $(3,2)$, $(3,4)$, $(4,0)$ and $(4,3)$ are black and the rest of pixels are white.

From a *pbm* file we can easily obtain the 2-dimensional array of the image in the suitable format for our programs. Then, we have the following function.

pbm-to-loc-path pbm-path *[Function]*

> This function takes as argument a *pbm* image stored in the path `pbm-path`, and returns the 2-dimensional array of the image as a list of lists of 1's and 0's.

To provide a better understanding of this new function, an example of its use is presented. Let us consider a *pbm* file called `simple.pbm` that codifies the image depicted in Figure 5.9 as was presented previously.

From that file, we can obtain the 2-dimensional array of the image in the desirable format for our program as follows:

```
> (setf small-2d-image (pbm-to-loc-path "simple.pbm")) ✠
((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0))
```

Now, we can proceed as in the previous examples. Then, to process 2D-images codified in different image formats with our programs, we only need to transform the original codification to the *pbm* format (a task that can be performed with different tools, for instance GIMP [Pec08] allows us to convert an image stored in, practically, any format to the *pbm* one).

To sum up, an interpreter for 2D-images stored in "*pbm*" format has been developed. However, in the case of 3D-images, we have not found any 3D-image format which codifies a 3D-image by means of a 3-dimensional array; and, at this moment, we have not developed an interpreter which converts from 3D-image formats to the suitable input format of our programs; that is, the 3-dimensional array associated with an image. This task remains as further work.

Nevertheless, we have developed and interpreter which converts from a 3-dimensional array, codified as a list of lists of lists, to the "*obj*" 3D format [Tec] in order to be able to show our images in usual 3D-image renders. The information stored in the files which use this format consists of the position of the tetrahedrons of the image; since it works with tetrahedrons instead of working with voxels. Then, we have developed a Common Lisp program which from the 3-dimensional array of a 3D-image returns the 3D-image codified in this format. So, if we save the generated data in an "*obj*" file, we can visualize the image in a 3D viewer.

3da-to-obj 3da *[Function]*

> This function takes as argument a 3-dimensional array, codified as a list of lists of lists, of a 3D-image, and returns the image codified in the "*obj*" format.

For instance, if we invoke the `3da-to-obj` function with the 3-dimensional array declared in `small-3d-image` as argument; the content of the file which allows us to show the image of Figure 5.10 is generated.

```
> (3da-to-obj small-3d-image) ✠
v 0 1 1
v 0 0 1
v 1 0 1
...
f 1 2 3 4
f 8 7 6 5
...
```

To sum up, we have enhanced the Kenzo system with a new module which allows us to study monochromatic 2D-images and 3D-images.

5.2.3 Digital images in our framework

We have enhanced our framework with the functionality related to digital images by means of a new plug-in. This plug-in will allow us to construct the simplicial set canonically associated with a simplicial complex defined by means of a list of facets extracted from the 2-dimensional array of a 2D-image (applying algorithms 5.19 (case $n = 2$), 5.6 and 5.9) and also the simplicial set canonically associated with a simplicial complex defined by means of a list of facets extracted from the 3-dimensional array of a 3D-image (applying algorithms 5.19 (case $n = 3$), 5.6 and 5.9).

This plug-in, which includes the functionality about digital images in our framework, references the following resources which are used by the plug-in framework to extend the functionality of all the components.

```
<code id="digital-images">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> simplicial-complexes.lisp </data>
   <data format="Kf/internal-server"> digital-images.lisp </data>
   <data format="Kf/microkernel"> digital-images-m.lisp </data>
   <data format="Kf/adapter"> digital-images-a.lisp </data>
</code>
```

Let us explain each one of the referenced resources. As we have said previously, the `XML-Kenzo.xsd` file specifies the XML-Kenzo language. We want to introduce two new kind of objects in our system (the simplicial sets associated with simplicial complexes generated from the sequence of facets of 2D-images and 3D-images); then, it is necessary to provide a representation for those objects in our framework.

In the XML-Kenzo specification, we have defined two new simple types: `array-2d`, which represents a 2-dimensional array of 1's and 0's that will be used to provide the

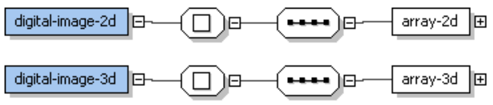


Figure 5.11: Digital images elements in XML-Kenzo

2-dimensional array of a 2D-image, and `array-3d`, which represents a 3-dimensional array of 1's and 0's that will be used to provide the 3-dimensional array of a 3D-image. In addition, two new elements have been defined: `digital-image-2d`, whose value is an element of the type `array-2d`; and, `digital-image-3d`, whose value is an element of the type `array-3d` (see Figure 5.11). Both `digital-image-2d` and `digital-image-3d` elements are defined as elements of the `SS` type because they are used to represent the simplicial sets associated with the simplicial complexes defined from 2D-images and 3D-images. Due to the fact that they are defined as elements of the `SS` type they can be used as any other element of this type.

When upgrading the `XML-Kenzo.xsd` the external server evolves and is able to check the constraints against the XML-Kenzo specification of requests related to digital images such as:

```
<constructor>
    <digital-image-2d>
        <array-2d>
           <line> 1 0 </line>
           <line> 0 1 </line>
        </array-2d>
    </digital-image-2d>
</constructor>
```

The `simplicial-complexes.lisp` file includes the Kenzo functionality about simplicial complexes presented in Subsection 5.1.2, since if we want to use the functionality about digital images in Kenzo, it is necessary to load the functionality about simplicial complexes.

The `digital-images.lisp` file includes the functionality explained in Subsection 5.2.2 to extend the Kenzo system. Moreover, this file includes the functionality that enhances the `xml-kenzo-to-kenzo` function of the internal server to process requests devoted to the construction of objects from the elements `digital-image-2d` and `digital-image-3d`. For instance, if the internal server receives the above XML-Kenzo request. The instruction

```
(ss-from-sc (simplicial-complex-generator (transform-lolol-to-lol
  (generate-facets-image-2d '((1 0) (0 1)))))
```

is executed in the Kenzo kernel. As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned.

The `digital-images-m.lisp` file defines two new construction modules for the microkernel called `digital-images-2d` and `digital-images-3d`. Each one of these modules implements a procedure following the guidelines explained in Subsubsection 2.2.3.3 to construct respectively the simplicial sets associated with 2D-images and 3D-images in the microkernel. Neither of the new modules have to check if the XML-Kenzo requests related to digital images are correct, since all the restrictions about these requests are controlled in the external server thanks to the XML-Kenzo specification (this is due to the fact that we only have the restriction of being an array of 1's and 0's, and this is an independent argument restriction).

Finally, we have extended the `SS` Content Dictionary by means of the definition of two new objects: `digital-images-2d` and `digital-images-3d`. Therefore, the `digital-images-a.lisp` file contains the necessary functions to raise the functionality of the adapter to be able to convert from the new OpenMath requests, devoted to digital images, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the previous XML-Kenzo request is generated by the Adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="digital-image-2d"/>
      <OMA>
        <OMS name="array-2d"/>
        <OMA> <OMS name="line"/> <OMI>1</OMI><OMI>0</OMI> </OMA>
        <OMA> <OMS name="line"/> <OMI>0</OMI><OMI>1</OMI> </OMA>
      </OMA>
   </OMA>
</OMOBJ>
```

5.2.4 Digital images in the ***fKenzo*** GUI

This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to support the interaction with digital images. We have defined a fresh *fKenzo* module to enhance the GUI with support for digital images. The new OMDoc module references three files: `digital-images-structure` (that defines the structure of the graphical constituents), `digital-images-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection.

We have defined six graphical elements, using the XUL specification language, in the `digital-images-structure` file:

- A menu called `Digital Images` which contains two options: `Load Digital Image 2D` and `Load Digital Image 3D`.
- A selection window called `select-2D-image`.

- A selection window called `select-3D-image`.
- A window called `show-2D-image` with an image viewer as sole component.
- A window called `show-3D-image` with a browser as sole component.
- A window called `SS-from-DI-name`.

The functionality stored in the `digital-images-functionality` document related to these components works as follows. A function acting as event handler is associated with the `Load Digital Image 2D` menu option; this function shows the `select-2D-image` window which allows the user to choose an image from a *pbm* file. Currently, a folder, called *Digital Images Examples*, with several examples of *pbm* files is included in the distribution of *fKenzo*.

Once the user has selected a *pbm* image, the system checks if the file is a *pbm* file which can be processed in our system; otherwise it informs the user with a warning message. Subsequently, if the file is valid, it invokes our framework with a `digital-image-2d` OpenMath request with the information obtained from the selected file through the `pbm-to-loc-path` function, which is also included in the `digital-images-functionality` document. Subsequently, the simplicial set associated with the simplicial complex defined from the list of simplexes obtained from the 2-dimensional array of the image is constructed and an identification number is returned. Afterwards, the system asks a name for the new simplicial set by means of the window `SS-from-DI-name`; if the name given by the user is correct (was not used previously), the system stores it, otherwise it indicates that the name was previously used and asks again a name. Eventually, if the result returned by the was an identification number; the system adds to the list of constructed spaces (situated in the left side of the main tab of *fKenzo*) the new object.

Moreover, the `digital-images-functionality` file increases the behavior of the event handler associated with the left list of the *fKenzo* GUI. As we explained in Subsection 3.2.1 when a space is selected from the list of constructed spaces, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. However, if the space selected is a simplicial set constructed from a digital image, the information that appears at the bottom part of the right side of the *fKenzo* GUI is the name given by the user. In addition, the *pbm* file is used to show the image associated with the simplicial set in the window `show-2D-image`. To show the images in the `show-2D-image` window, we use a graphical component which allows us to show 2D-images.

The case of 3D-images is analogous. In this case, examples of 3D digital images are included in the folder *Digital Images Examples 3D*; these images are stored in files with extension "3d", it is our own format to store the 3-dimensional array of an image. Moreover, in this case the image viewer of 3D-images shows the 3D-image in a browser component by means of the JavaView applet [P$^+$02]. To be able to show the 3D-images, we use the `loc-to-obj` function (explained in Subsection 5.2.2) which is included in the `digital-images-functionality` file, and which transforms the 3-dimensional array of an image to its "*obj*" codification (an image format which can be rendered by the JavaView applet).

5.2.4.1 New objects in the *fKenzo* GUI

As we have just said the behavior of the event handler associated with the left list of the *fKenzo* GUI has been modified. In Subsubsection 4.1.5.1 the management of objects in the *fKenzo* GUI and the functionality associated with the left list of the *fKenzo* GUI were explained. To handle the new behavior presented for objects associated with 2D-images and 3D-images we have included the following definitions.

As we commented in Subsubsection 4.1.5.1 we have two subclasses of the `FKENZO-OBJECT` class, `FKENZO-OBJECT-NAME` and `FKENZO-OBJECT-FILE`. Now, we have specialized the `FKENZO-OBJECT-FILE` to represent objects associated with 2D-images and 3D-images. Moreover, the new classes are also an specialization of the `FKENZO-OBJECT-NAME` class. Namely, we have defined the following two classes:

```
(DEFCLASS FKENZO-OBJECT-IMAGE-2D (FKENZO-OBJECT-NAME FKENZO-OBJECT-FILE))
```

```
(DEFCLASS FKENZO-OBJECT-IMAGE-3D (FKENZO-OBJECT-NAME FKENZO-OBJECT-FILE))
```

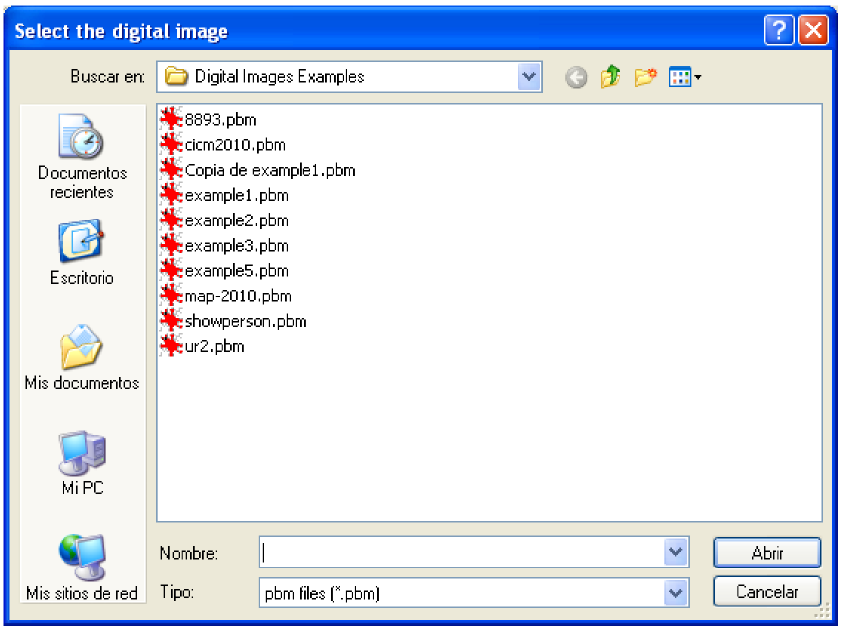
Both `FKENZO-OBJECT-IMAGE-2D` and `FKENZO-OBJECT-IMAGE-3D` are subclasses of both the `FKENZO-OBJECT-NAME` and `FKENZO-OBJECT-FILE` classes without adding any additional slot. An instance of the `FKENZO-OBJECT-IMAGE-2D` class is constructed for simplicial sets associated with 2D-images, the `name` slot is given by the user and the `file` slot is the path of the 2D-image. Analogously for the `FKENZO-OBJECT-IMAGE-3D`.

As we wanted a different behavior for the event handler, associated with the left list of the *fKenzo* GUI for the new objects, to the one presented before, we need some new concrete methods. In particular, we have defined the following two methods:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-IMAGE-2D))
 (show-image-2d (file object))
 (call-next-method))
```

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-IMAGE-3D))
 (show-image-3d (file object))
 (call-next-method))
```

The method associated with the objects of the `FKENZO-OBJECT-IMAGE-2D` class shows the window with the 2D-image stored in the path indicated by the `file` slot; subse-



Buscar en: Digital Images Examples

8893.pbm
cicm2010.pbm
Copia de example1.pbm
example1.pbm
example2.pbm
example3.pbm
example5.pbm
map-2010.pbm
showperson.pbm
ur2.pbm

Tipo: pbm files (*.pbm)

Figure 5.12: Window to load a 2D-image

quently, it calls the method associated with the `FKENZO-OBJECT-NAME` class; and therefore the *fKenzo* GUI shows the name of the object at the bottom part of the right side of the *fKenzo* GUI. Analogously for the objects of the `FKENZO-OBJECT-IMAGE-3D` but showing the 3D-image stored in the path indicated by the `file` slot after the conversion to the *obj* format.

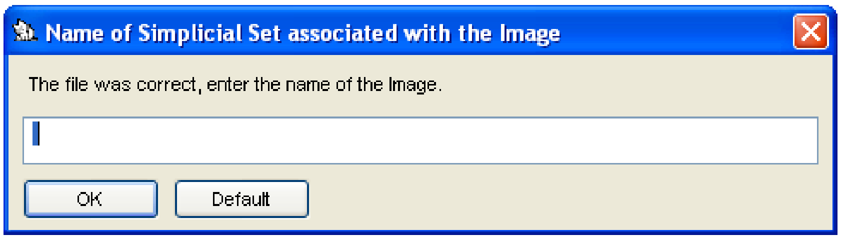
The `call-next-method` function of these methods allow them to refer to a less specific method, namely the one associated with the `FKENZO-OBJECT-NAME` class. This means that if we have selected an object in the left list of the *fKenzo* GUI that corresponds with a `FKENZO-OBJECT-IMAGE-2D` or a `FKENZO-OBJECT-IMAGE-3D` object, the GUI shows the name of the object at the bottom part of the right side of the *fKenzo* GUI.

In this way, the behavior of the left list of the GUI is enhanced without touching the main code.

5.2.4.2 Execution flow of the *fKenzo* GUI for digital images

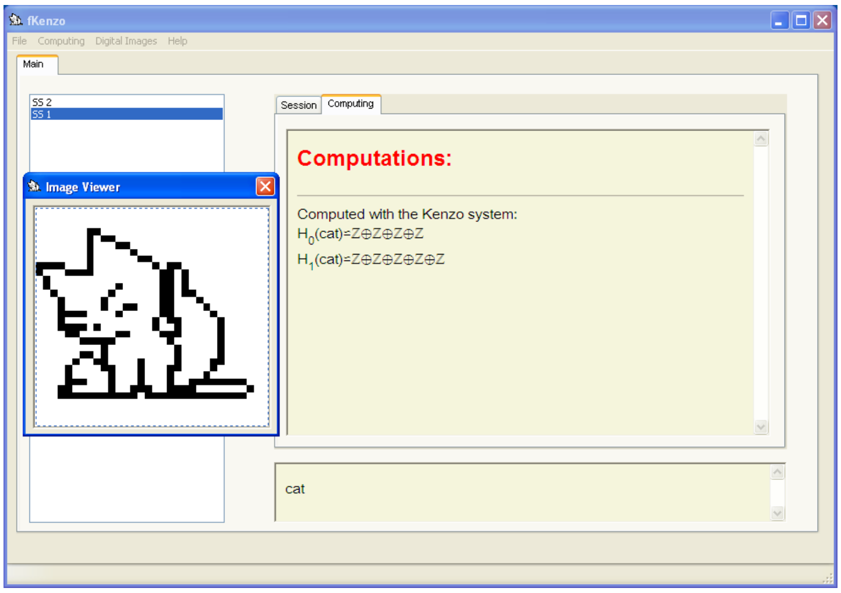
In order to clarify the execution flow followed by *fKenzo*, let us present two examples. When the Digital Images module is loaded in *fKenzo*, a new menu called *Digital Images*, with two options; `Load Digital Image 2D` and `Load Digital Image 3D`, becomes available. From the option `Load Digital Image 2D`, the user can select a *pbm* image from the folder *Digital Images Examples* with the `select-2D-image` window, see Figure 5.12. For instance, the `8893.pbm` file is the image of a small cat.

From the `8893.pbm` file and using the `pbm-to-loc` function, *fKenzo* constructs the OpenMath request which is sent to our framework. From this request the simplicial


Name of Simplicial Set associated with the Image

The file was correct, enter the name of the Image.

Figure 5.13: *fKenzo* asking a name for a simplicial set coming from a digital image



Computations:

Computed with the Kenzo system:
$H_0(cat)=Z\oplus Z\oplus Z\oplus Z$
$H_1(cat)=Z\oplus Z\oplus Z\oplus Z\oplus Z$

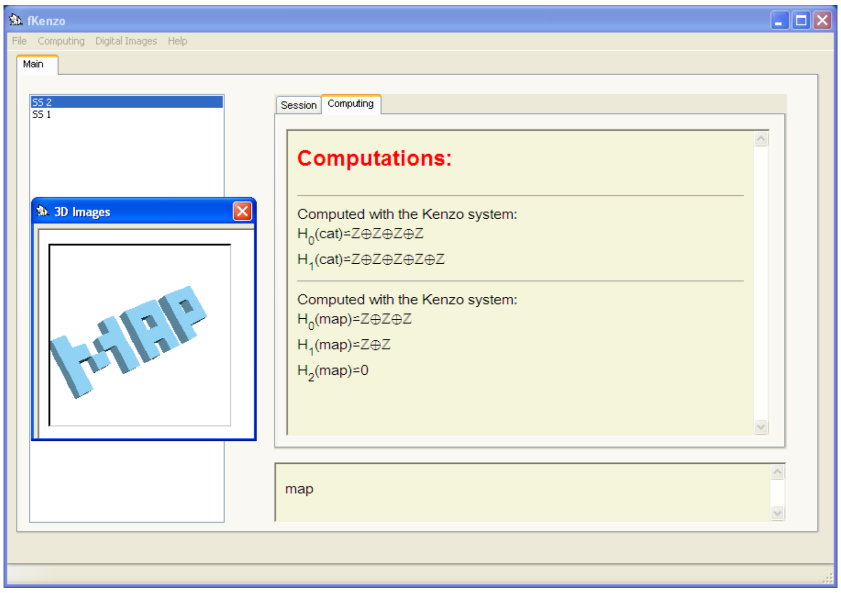
cat

Figure 5.14: Homology groups of the small cat

set associated with the simplicial complex defined from the facets obtained from the coordinates of black pixels of the image is built, and the identification number of the constructed object is returned. Then, *fKenzo* asks the user a name for the object by means of the window `SS-from-DI-name` (see Figure 5.13), in this case the name given is "cat". Subsequently, the new object is added to the list of constructed spaces shown in the left side of the *fKenzo* GUI. When the user selects this object, both the name that the user provided previously and the image are shown (see Figure 5.14).

Finally, the user can ask *fKenzo* to compute the homology groups of the image, and the results are shown, as usual, in the Computing tab. Figure 5.14 shows the computation of the homology groups of the small cat which has 4 connected components and 5 holes. These properties are interpreted from the homology groups of the image.

Analogously for the `Load Digital Image 3D` option. Figure 5.15 shows the computation of homology groups of the letters "MAP" in 3D; this image has 3 connected components, 2 tunnels and 0 cavities. As in the case of 2D-images, these properties are interpreted from the homology groups of the image.



Computations:

Computed with the Kenzo system:
$H_0(\text{cat})=\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}$
$H_1(\text{cat})=\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}$

Computed with the Kenzo system:
$H_0(\text{map})=\mathbb{Z}\oplus\mathbb{Z}\oplus\mathbb{Z}$
$H_1(\text{map})=\mathbb{Z}\oplus\mathbb{Z}$
$H_2(\text{map})=0$

map

Figure 5.15: Homology groups of the letters MAP in 3D

5.2.5 Formalization of Digital Images in ACL2

As we already claim, the verification of programs is an important issue, but specially in the case of the programs for digital images. If we want to use our programs in real life problems (for instance, in the study of medical images), we must be completely sure that the results produced by our programs are correct. Therefore, the formal verification of our programs with a Theorem Prover (in our case, ACL2) is significant.

This subsection is devoted to present the certification of the correctness of Algorithm 5.15 which from a 2D-image constructs a list of simplexes (which represents the facets of the simplicial complex associated with the 2D-image). The task of verifying the correctness of Algorithm 5.19 for the case $n = 3$ is analogous; then, in spite of having both developments, we just focus on the verification of the correctness of Algorithm 5.15.

5.2.5.1 Main definitions and properties

As we just said, we want to formalize in ACL2 the correctness of Algorithm 5.15; namely, our implementation of that algorithm by means of the `generate-facets-image-2d` function. Since both Kenzo and ACL2 are Common Lisp programs we can verify the correctness of that Kenzo function in ACL2.

From now on, we define the necessary functions to establish the correctness of our program. First of all, we need some auxiliary functions which define the necessary concepts to prove our theorems. These definitions are based on both Algorithm 5.15 and the notions for digital images. Namely, we need to define the notion of 2D-image.

As we said in Subsection 5.2.2, a 2D-image $\mathcal{D}$ is represented by means of a finite

2-dimensional array (that is a list of lists) of 1's and 0's where each 1 represents a pixel in $\mathcal{D}$ and each 0 represents a pixel that is not in $\mathcal{D}$. The `2d-imagep` function is a function that checks if its argument is a list of lists of 1's and 0's. This function uses the `list-0-1-p` function that checks if its argument is a list of 1's and 0's.

```
(defun list-0-1-p (list)
  (if (endp list)
      (equal list nil)
    (if (endp (cdr list))
        (and (equal (cdr list) nil)
             (or (equal (car list) 0) (equal (car list) 1)))
      (and (or (equal (car list) 0) (equal (car list) 1))
           (list-0-1-p (cdr list))))))
```

```
(defun 2d-imagep (list)
  (if (endp list)
      (equal list nil)
    (and (list-0-1-p (car list)) (2d-imagep (cdr list)))))
```

Subsequently, we define the `generate-facets-image-2d` function and all its auxiliary functions in ACL2. It is worth noting that the definition of these functions is exactly the same used in the definition of Kenzo functions (see Subsection 5.2.2). Let us show in detail these definitions.

First of all, we define the `list-up-i-j` and `list-down-i-j` functions which are used to generate from a pair of natural numbers (i, j) the simplexes $((i, j), (i+1, j), (i+1, j+1))$ and $((i, j), (i, j+1), (i+1, j+1))$ respectively.

```
(defun list-up-i-j (i j)
  (list (list i j) (list (1+ i) j) (list (1+ i) (1+ j))))
```

```
(defun list-down-i-j (i j)
  (list (list i j) (list i (1+ j)) (list (1+ i) (1+ j))))
```

From the above two functions, we can define a function, called `generate-facets-i-j` which from the pair (i, j) generates the pair of simplexes $(((i, j), (i+1, j), (i+1, j+1)), ((i, j), (i, j+1), (i+1, j+1)))$

```
(defun generate-facets-i-j (i j)
  (list (list-up-i-j i j) (list-down-i-j i j)))
```

Now, we can define the `generate-facets-image-2d` function which generates the simplexes of a list of lists of 0's and 1's `lol`.

```
(defun generate-facets-image-2d (lol)
  (generate-facets-image-aux lol 0))
```

The above function calls the more general function `generate-facets-image-aux` which takes two arguments: a list of lists of 0's and 1's `lol` and a natural number j. The function `generate-facets-image-aux` must be understood as the procedure which generates the simplexes of the list of lists of 0's and 1's `lol` which is the sublist located from position j of another list of lists of 0's and 1's, let us called it `lol-main`.

```
(defun generate-facets-image-aux (lol j)
  (if (endp lol)
      nil
    (append (generate-facets-list (car lol) 0 j)
            (generate-facets-image-aux (cdr lol) (1+ j)))))
```

For each one of the lists of 0's and 1's of `lol` and the position of that list in `lol-main`, the above function invokes the function `generate-facets-list`. The function `generate-facets-list` must be understood as the procedure which generates the simplexes of the list of 0's and 1's `list` which is the sublist located from position i of the list of position j of `lol-main`.

```
(defun generate-facets-list (list i j)
  (if (endp list)
      nil
    (if (equal (car list) 1)
        (append (generate-facets-i-j i j) (generate-facets-list (cdr list) (1+ i) j))
      (generate-facets-list (cdr list) (1+ i) j))))
```

Once we have defined our programs in ACL2 we can prove theorems about them. To be more concrete, we have proved both the correctness a the completeness of our program `generate-facets-image-2d`.

First of all we state the ACL2 theorem which ensures the completeness of `generate-facets-image-2d`.

ACL2 Theorem 5.20. Let `image` be a 2D-image represented by means of a 2 dimensional array, then, $\forall i, j \in \mathbb{N}$ such that the value of the image in position (i, j) of the array is 1, then, the simplexes $((i, j), (i+1, j), (i+1, j+1))$ and $((i, j), (i, j+1), (i+1, j+1))$ are in the list generated by the `generate-facets-image-2d` function taking as input `image`.

To state this theorem in ACL2, we need the ACL2 functions: `(natp n)`, which is a test function returning `t` if `n` is a natural number and `nil` otherwise; `(nth i ls)`, which returns the value of position `i` (a natural number) of the list `ls`; and, `(member-equal x ls)`, which returns `t` if `x` is equal to some of the elements of `ls` (a list).

```
(defthm generate-facets-image-2d-completeness
  (implies (and (2d-imagep image)
                (natp i)
                (natp j)
                (equal (nth i (nth j image)) 1))
           (and (member-equal (list-up-i-j i j) (generate-facets-image-2d image))
                (member-equal (list-down-i-j i j) (generate-facets-image-2d image)))))
```

Once we have proved the completeness of our program, we must prove its correctness. This task is handled by means of the following lemmas.

ACL2 Theorem 5.21. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image`. Then if `simplex` is of the form $((i, j), (i + 1, j), (i + 1, j + 1))$ with i and j natural numbers, then the element $((i, j), (i, j+1), (i+1, j+1))$ is also in the output generated by `generate-facets-image-2d` taking as input `image`.

To state this theorem in ACL2 we need some auxiliary functions. Namely, `member-list-up`, which returns `t` if its input is a list of the form $((i, j), (i + 1, j), (i + 1, j + 1))$ and `nil` otherwise; and `list-down`, which from a list of the form $((i, j), (i + 1, j), (i + 1, j + 1))$ returns the list $((i, j), (i, j + 1), (i + 1, j + 1))$.

```
(defthm generate-facets-image-correctness-1
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image))
                (member-list-up simplex))
           (member-equal (list-down simplex) (generate-facets-image-2d image))))
```

ACL2 Theorem 5.22. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image`. Then if `simplex` is of the form $((i, j), (i, j + 1), (i + 1, j + 1))$ with i and j natural numbers, then the element $((i, j), (i+1, j), (i+1, j+1))$ is also in the output generated by `generate-facets-image-2d` taking as input `image`.

To state this theorem in ACL2 we need some auxiliary functions. Namely, `member-list-down`, which returns `t` if its input is a list of the form $((i, j), (i, j + 1), (i + 1, j + 1))$ and `nil` otherwise; and `list-up`, which from a list of the form $((i, j), (i, j + 1), (i + 1, j + 1))$ returns the list $((i, j), (i + 1, j), (i + 1, j + 1))$.

```
(defthm generate-facets-image-correctness-2
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image))
                (member-list-down simplex))
           (member-equal (list-up simplex) (generate-facets-image-2d image))))
```

ACL2 Theorem 5.23. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image` of the form $((i,j),(i{+}1,j),(i{+}1,j{+}1))$ or $((i,j),(i,j+1),(i+1,j+1))$ with i and j natural numbers. Then, the element of position (i,j) of `image` is 1.

To state this theorem in ACL2 we use some ACL2 functions which have not been used previously: `caar` which returns the first element of the first element of a list and `cadar` which returns the second element of the first element of a list.

```
(defthm generate-facets-image-correctness-3
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image)))
           (equal (nth (caar simplex) (nth (cadar simplex) image)) 1)))
```

Let us present some remarks about the proof of these theorems which state both the completeness and the correctness of the `generate-facets-image-2d` program.

First of all, it is worthwhile noting that the implementation of the `generate-facets-image-2d` function, and its auxiliar ones, follows simple recursive schemas, that are suitable for the induction heuristics of the ACL2 theorem prover.

Let us present now with some details two auxiliary lemmas needed in our development of the proof of the main theorems. As we have seen in the definition of `generate-facets-image-aux`, this function invokes the `generate-facets-list` function with arguments `(car list)`, `0` and `j`. Therefore, it is sensible to think that in the proof of our theorems we are going to need some auxiliary lemmas such as:

```
(thm (implies (and (list-0-1 x) (natp j)
                   (member-equal simplex (generate-facets-list x 0 j)))
              (equal (nth (caar simplex) x) 1))
```

that is to say, a lemma which involves a call to `(generate-facets-list x 0 j)`. However, ACL2 has some problems to find a proof of theorems such as the previous one, since it does not find a good inductive schema for reasoning. On the contrary, for ACL2 is much easier to find a proof of theorems such as:

```
(thm (implies (and (list-0-1 x) (natp i) (natp j)
                   (member-equal simplex (generate-facets-list x i j)))
              (equal (nth (- (caar simplex) i) x) 1))
```

that is to say, lemmas that are generalizations of the previous ones.

Taking this question into account in the development of our proofs, the certification of the completeness and correctness theorems can be done without any special trouble.

In this way, we have proved the completeness and correctness of our implementation of Algorithm 5.15 by means of the program `generate-facets-image-2d`.

In the case of 3D-digital images the development is very similar. The interested reader can consult the complete development in [Her11].

5.3 An algorithm building the pushout of simplicial sets

Many of the usual constructions in Topology are nothing but homotopy pullbacks or homotopy pushouts [Mat76]. Loop spaces, suspensions, mapping cones, wedges or joins, for instance, involve such constructions. In these cases, when the spaces are not of finite type the computation of their homology groups can be considered as a challenging task. A way of dealing with this kind of spaces consists of using the effective homology method explained in Subsection 1.1.3.

In this section we use the effective homology method to design algorithms building the pushout associated with two simplicial morphisms $f : X \to Y$ and $g : X \to Z$, where X, Y and Z are simplicial sets with *effective homology*. In addition the integration of this new tool in *fKenzo* is presented.

The rest of this section is organized as follows. Subsection 5.3.1 introduces the mathematical background about the pushout and some additional concepts about effective homology. The way of constructing the effective homology of the pushout is presented in Subsection 5.3.2. Some examples of the use of the implementation of those algorithms as a new Kenzo module are provided in Subsection 5.3.3. Subsection 5.3.4 deals with the extension of the framework to include the functionality about the pushout; and the way of widening the *fKenzo* GUI to include the pushout is detailed in Subsection 5.3.5. Finally, some remarks about the formalization of the developed algorithms are presented in Subsubsection 5.3.6.

5.3.1 Preliminaries

To explain the effective homology of the pushout is necessary to introduce the notion of pushout and also some additional notions about effective homology which were not presented in Subsection 1.1.3.

5.3.1.1 Pushout notions

First of all, let us introduce the notion of pushout. The following definitions can be found, for instance, in [Mat76, Doe98].

Definition 5.24. Consider two morphisms $f : X \to Y$, $g : X \to Z$ in a category $\mathcal{C}$. A *pushout* of (f, g) is a triple (P, f', g') where

1. P is an object of $\mathcal{C}$,
2. $f' : Y \to P$, $g' : Z \to P$ are morphism of $\mathcal{C}$ such that $f \circ g' = g \circ f'$,

and for every other triple (Q, f'', g'') where

1. Q is an object of $\mathcal{C}$,
2. $f'' : Y \to Q$, $g'' : Z \to Q$ are morphism of $\mathcal{C}$ such that $f \circ g'' = g \circ f''$,

there exists a unique morphism $p : P \to Q$ such that $f'' = p \circ f'$ and $g'' = p \circ g'$ (see the following diagram).

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[ddr, "f''"] & \\
Z \arrow[r, "g'"] \arrow[drr, "g''"'] & P \arrow[dr, dashed, "p"] & \\
 & & Q
\end{tikzcd}$$

Definition 5.25. Let $f, g : X \to Y$ be two morphisms between topological spaces, then a morphism $H : X \times I \to Y$, where I is the unit interval $[0, 1]$, is a homotopy between f and g, denoted by $H : f \sim g$, if $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$.

Definition 5.26. A homotopy commutative diagram:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \\
Z \arrow[r, "g'"] & P
\end{tikzcd}$$

equipped with $H : f' \circ f \sim g' \circ g$, is called a *homotopy pushout* when for any commutative diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f''"] \\
Z \arrow[r, "g''"] & Q
\end{tikzcd}$$

equipped with $G : f'' \circ f \sim g'' \circ g$, the following properties hold:

1. there exists a map $p : P \to Q$ and homotopies $K : f'' \sim p \circ f'$ and $L : p \circ g' \sim g''$ such that the whole diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[ddr, "f''"] & \\
Z \arrow[r, "g'"] \arrow[drr, "g''"'] & P \arrow[dr, dashed, "p"] & \\
& & Q
\end{tikzcd}$$

with all maps and homotopies above is homotopy commutative,

2. if there exists another map $p' : P \to Q$ and homotopies $K' : f'' \sim p' \circ f'$ and $L' : p' \circ g' \sim g''$ such that the diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[ddr, "f''"] & \\
Z \arrow[r, "g'"] \arrow[drr, "g''"'] & P \arrow[dr, dashed, "p'"] & \\
& & Q
\end{tikzcd}$$

is homotopy commutative, then there exists a homotopy $M : p \sim p'$ such that the whole diagram with all maps and homotopies above is homotopy commutative.

From now on, with an abuse of notation, we will call *(homotopy) pushout* of (f, g) to the object P.

There is a "standard" construction of the homotopy pushout of any two maps $f : X \to Y$, $g : X \to Z$ as:

$$P_{(f,g)} \cong (Y \amalg (X \times I) \amalg Z)/ \sim$$

where I is the unit interval $[0, 1]$ and the equivalence relation $\sim$ is defined as follows: for every $x \in X$, $(x \times 0)$ is identified to $f(x) \in Y$ and $(x \times 1)$ is identified to $g(x) \in Z$.

5.3.1.2 Effective homology preliminaries for the pushout

Let us present now some effective homology concepts which are akin to the definition of the effective homology of the pushout. A complete study of the definitions and results presented here can be found in [RS06].

Theorem 5.27 (Direct Sum Equivalence Theorem)**.** Let C_* and D_* be two chain complexes with effective homology. Then the direct sum $C_* \oplus D_*$ is a chain complex with effective homology.

The *cone constructor* is an important construction in Homological Algebra, we present here the definition and the most elementary properties of this construction.

Definition 5.28. Let C_* and D_* be two chain complexes and $\phi : D_* \to C_*$ be a chain complex morphism. Then the cone of ϕ denoted by $Cone(\phi)$ is the chain complex $Cone(\phi) = A_*$ defined as follows. First $A_n := C_{n+1} \oplus D_n$; and the boundary operator is given by the following matrix:

$$d_{A_*} := \begin{bmatrix} d_{C_*} & \phi \\ 0 & -d_{D_*} \end{bmatrix}$$

In the previous definition, if both C_* and D_* are chain complexes with effective homology, then, the following theorems ensure that we can construct an equivalence between $Cone(\phi)$ and an effective chain complex, therefore, $Cone(\phi)$ is also a chain complex with effective homology.

Theorem 5.29 (Cone Reduction Theorem [RS06])**.** Let $\rho = (f, g, h) : C_* \Rightarrow\!\!\!\!\Rightarrow D_*$ and $\rho' = (f', g', h') : C'_* \Rightarrow\!\!\!\!\Rightarrow D'_*$ be two reductions and $\phi : C'_* \to C_*$ a chain complex morphism. Then these data define a canonical reduction:

$$\rho'' : Cone(\phi) \Rightarrow\!\!\!\!\Rightarrow Cone(f\phi g')$$

An extension of the Cone Reduction Theorem is the following result.

Theorem 5.30 (Cone Equivalence Theorem [RS06])**.** Let $\phi : C'_{*,EH} \to C_{*,EH}$ be a chain complex morphism between two chain complexes with effective homology. Then $Cone(\phi)$ is a chain complex with effective homology.

Definition 5.31. An *effective short exact sequence* of chain complexes is a diagram:

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

where i and j are chain complexes morphisms, ρ (retraction) and σ (section) are graded module morphisms satisfying:

- $\rho i = id_{C_*}$;
- $i\rho + \sigma j = id_{B_*}$;
- $j\sigma = id_{A_*}$.

It is an exact sequence in both directions, but to the left it is an exact sequence of chain complexes, and to the right it is only an exact sequence of graded modules.

The following theorem (see [RS06]) states that given an effective short exact sequence, if two of the chain complexes of the short exact sequence are chain complexes with effective homology, then, we can construct the effective homology of the third one.

Theorem 5.32 (SES Theorems)**.** Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then three general algorithms are available:

$$\begin{array}{l} SES_1 : (B_{*,EH}, C_{*,EH}) \mapsto A_{*,EH} \\ SES_2 : (A_{*,EH}, C_{*,EH}) \mapsto B_{*,EH} \\ SES_3 : (A_{*,EH}, B_{*,EH}) \mapsto C_{*,EH} \end{array}$$

producing the effective homology of one chain complex when the effective homology of both others is given.

The following two lemmas are used in the proof of the above theorem and will be important in the development of the effective homology of a pushout.

Lemma 5.33. Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then the effective exact sequence produces a reduction $Cone(i) \Rrightarrow A_*$.

To state the second lemma we need an auxiliary definition.

Definition 5.34. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ be a chain complex. The *suspension functor* applied to C_* is the chain complex $C_*^{[1]} = (M_n, d_n)_{n\in\mathbb{Z}}$ such that, $M_n = C_{n-1}$ and $d_n = -d_{C_{n-1}}$, $\forall n \in \mathbb{Z}$.

Moreover, we can define the effective homology version of the suspension functor.

Theorem 5.35 (Suspension Functor Equivalence Theorem)**.** Let C_* be a chain complex with effective homology. Then $C_*^{[1]}$ is a chain complex with effective homology.

Now, we can provide the second lemma associated with Theorem 5.32.

Lemma 5.36. Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then the effective exact sequence generates a connection chain complex morphism $\chi : A_* \to C_*^{[1]}$. Besides, B_* is canonically isomorphic to $Cone(\chi)$.

Once these concepts have been introduced, we can undertake our goal of defining the effective homology of a pushout.

5.3.2 Effective Homology of the Pushout

5.3.2.1 Main algorithms

The definitions related to the pushout given in Subsubsection 5.3.1.1 come from standard topology, from now on we switch to the simplicial framework where we can formulate the following algorithm.

Algorithm 5.37.
Input: two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets.
Output: the simplicial set $P_{(f,g)}$.

The above algorithm is based on the standard pushout construction presented in Subsubsection 5.3.1.1, in particular given $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, then we define $P_{(f,g)}$ as the simplicial set $(Y \amalg (X \times \Delta^1) \amalg Z)/\sim$, where Δ^1 is the unit interval $[0, 1]$ in the simplicial framework: the simplicial set Δ^1 has two 0-simplexes $(0), (1)$ and the non-degenerate 1-simplex $(0, 1)$; and $\sim$ is the equivalence relation such that for every $x \in X$, $(x \times (0))$ is identified to $f(x) \in Y$ and $(x \times (1))$ is identified to $g(x) \in Z$.

Now, if X, Y and Z are simplicial sets with effective homology, then $P_{(f,g)}$ is also an object with effective homology; in particular we can formulate the following algorithm.

Algorithm 5.38.
Input: two morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology.
Output: the effective homology of $P_{(f,g)}$.

The construction of the effective homology of $P_{(f,g)}$ involves several steps. In a nutshell, the construction of the effective homology of $P_{(f,g)}$ is based on applying the case SES_2 of Theorem 5.32 to a short exact sequence which is produced by the description of C_*P (the chain complex coming from $P_{(f,g)}$):

$$\begin{tikzcd}
0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l]
\end{tikzcd}$$

where M_* is the chain complex associated with the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled (that is to say, the simplexes like $(x, (0)) \in X \times (0)$, $(x, (1)) \in X \times (1)$ and their degeneracies are removed from $X \times \Delta^1$), the morphisms σ and i are inclusions and the morphisms j and ρ are projections.

To apply the case SES_2 of Theorem 5.32 to the above short exact sequence, the effective homology of M_* and $C_*X \oplus C_*Y$ must be available. Then, the main steps to construct the effective homology of the pushout $P_{(f,g)}$ are the following ones.

Step 1. From $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, $P_{(f,g)}$ and its associated chain complex C_*P are constructed.

Step 2. The effective homology of M_* is constructed.

Step 3. The effective homology of $C_*X \oplus C_*Y$ is constructed.

Step 4. Eventually, from C_*P, the effective homology of M_*, the effective homology of $C_*X \oplus C_*Y$ and applying case SES_2 of Theorem 5.32 to the above short exact sequence, the effective homology of the pushout $P_{(f,g)}$ is constructed.

A complete description of the algorithm will be provided in Subsubsection 5.3.2.3.

5.3.2.2 Auxiliar algorithms

Several sub-algorithms have been required in the process to define Algorithm 5.38, in particular, we have needed the following ones.

Algorithm 5.39 (Definition 1.26)**.**
Input: two simplicial sets X and Y.
Output: the simplicial set $X \times Y$.

Algorithm 5.40 (Eilenberg-Zilber Theorem, see [May67])**.**
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $X \times Y$.

Algorithm 5.41 (Definition 1.8)**.**
Input: two simplicial sets X and Y.
Output: the direct sum chain complex $C_*X \oplus C_*Y$ where C_*X and C_*Y are the chain complexes associated with X and Y respectively.

Algorithm 5.42 (Theorem 5.27)**.**
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $C_*X \oplus C_*Y$.

Algorithm 5.43 (Definition 5.28)**.**
Input: a morphism i between two chain complexes A_* and B_*.
Output: the chain complex $Cone(i)$.

Algorithm 5.44 (Theorem 5.30)**.**
Input: a morphism i between two chain complexes A_* and B_* with effective homology.
Output: the effective homology of $Cone(i)$.

Algorithm 5.45.
Input: a simplicial set X.

Output: the chain complex coming from $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

Algorithm 5.46.
Input: a simplicial set X with effective homology.
Output: the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

Algorithm 5.47 (Definition 5.34)**.**
Input: a chain complex A_*.
Output: the chain complex $A_*^{[1]}$.

Algorithm 5.48 (Theorem 5.35)**.**
Input: a chain complex A_* with effective homology.
Output: the effective homology of $A_*^{[1]}$.

Algorithm 5.49 (Lemma 5.33)**.**
Input: A, B and C simplicial sets with effective homology, i, j, σ and ρ morphism which determine the short exact sequence $\begin{tikzcd} 0 & C_*A \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & C_*B \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*C \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$ where C_*A, C_*B and C_*C are the chain complex canonically associated with A, B and C respectively.
Output: the reduction $Cone(i) \Rrightarrow C_*A$.

Algorithm 5.50 (Proposition 1.40)**.**
Input: a reduction $B_* \Rrightarrow A_*$ and the effective homology of B_*.
Output: the effective homology of A_*.

Some of these algorithms are already implemented in Kenzo (namely, algorithms 5.39, 5.40 and 5.50). On the contrary, it has been necessary to implement the rest of them.

5.3.2.3 A complete description of the algorithm

In this subsubsection, we are going to present a detailed description of the 4 main steps to construct the effective homology of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology.

As we said previously, the construction of the effective homology of $P_{(f,g)}$ is based on applying the case SES_2 of Theorem 5.32 to the short exact sequence which is produced by the description of C_*P (the chain complex associated with $P_{(f,g)}$):

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

where M_* is the chain complex associated with the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled, the morphisms σ and i are inclusions and the morphisms j and ρ are projections. To apply the case SES_2 of Theorem 5.32, the effective homology of M_* and $C_*X \oplus C_*Y$ must be available. Then, the main steps to construct the effective homology of the pushout are as follows.

Step 1. From $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, $P_{(f,g)}$ and its associated chain complex C_*P are constructed.

To this aim, we simply apply Algorithm 5.37.

Step 2. We construct the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

The construction of the effective homology of this simplicial set is a bit tricky and needs several sub-steps that are now explained.

1. We construct the chain complex associated with $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled, that will be denoted by M_*, applying Algorithm 5.45.

   Briefly, the construction of the effective homology of M_* is obtained from the application of the case SES_1 of Theorem 5.32 to the short exact sequence produced by the description of the chain complex M_*:

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma 2"] & C_*(X \times \Delta^1) \arrow[l, shift left, "j2"] \arrow[r, shift left, "\rho 2"] & C_*(X \times (0)) \oplus C_*(X \times (1)) \arrow[l, shift left, "i2"] & 0 \arrow[l] \end{tikzcd}$$

   where the morphisms $\sigma 2$ and $i2$ are inclusions and the morphisms $j2$ and $\rho 2$ are projections. To apply the case SES_1 of Theorem 5.32, the effective homology of $C_*(X \times \Delta^1)$ and $C_*(X \times (0)) \oplus C_*(X \times (1))$ must be provided.

2. We build the effective homology of $C_*(X \times \Delta^1)$.

   Since both X and Δ^1 are simplicial sets with effective homology (Δ^1 is an effective object, then it has trivially effective homology), we can construct, applying Algorithm 5.40, the effective homology of $C_*(X \times \Delta^1)$.

3. We construct the effective homology of $C_*(X \times (0)) \oplus C_*(X \times (1))$.

   Since both $X \times (0)$ and $X \times (1)$ are simplicial sets with effective homology (applying Algorithm 5.40 since X, (0) and (1) are simplicial sets with effective homology), we can construct, applying Algorithm 5.42, the effective homology of $C_*(X \times (0)) \oplus C_*(X \times (1))$.

4. We construct the effective homology of $Cone(i2)$.

   Since $i2$ is a chain complex morphism between two objects with effective homology, $C_*(X \times \Delta^1)$ and $C_*(X \times (0)) \oplus C_*(X \times (1))$, we can construct, applying Algorithm 5.44, the effective homology of $Cone(i2)$.

5. The reduction $M_* \lll Cone(i2)$ is constructed applying Algorithm 5.49.

6. The effective homology of M_* is constructed.

   Since $Cone(i2)$ is a chain complex with effective homology and we have the reduction $M_* \lll Cone(i2)$, we can construct, applying Algorithms 5.50, the effective homology of M_*.

Therefore, we have constructed the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled. The above process produces Algorithm 5.46.

Step 3. We construct the effective homology of $C_*X \oplus C_*Y$.

Since both Y and Z are simplicial sets with effective homology we can construct, applying Algorithm 5.42, the effective homology of $C_*Y \oplus C_*Z$.

Step 4. The effective homology of the pushout $P_{(f,g)}$ is constructed.

Let us present how we proceed to complete this construction.

1. We define the following short exact sequence:

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

   where the morphisms σ and i are inclusions and the morphisms j and ρ are projections.

2. We construct the effective homology of $(C_*Y \oplus C_*Z)^{[1]}$.

   Since $C_*Y \oplus C_*Z$ is a chain complex with effective homology, applying Algorithm 5.48, we can construct the effective homology of $(C_*Y \oplus C_*Z)^{[1]}$.

3. We define the morphism $shift : C_*Y \oplus C_*Z \to (C_*Y \oplus C_*Z)^{[1]}$ which assigns every element of dimension n of $C_*Y \oplus C_*Z$ to the same element in dimension $n+1$ of $(C_*Y \oplus C_*Z)^{[1]}$.

4. We define the chain complex morphism $\chi : M_* \to (C_*Y \oplus C_*Z)^{[1]}$ as the composition $\chi = shift \circ \rho \circ d_{C_*P} \circ \sigma$.

5. We construct the effective homology of $Cone(\chi)$.

   Since χ is a chain complex morphism between two chain complexes with effective homology, M_* and $(C_*Y \oplus C_*Z)^{[1]}$, we can construct, applying Algorithm 5.44, the effective homology of $Cone(\chi)$.

6. Finally, applying Lemma 5.36, C_*P is isomorphic to $Cone(\chi)$, therefore, the effective homology of C_*P is obtained.

In this way, the effective homology of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology is obtained. Therefore, we can state the following theorem.

Theorem 5.51. Let $f : X \to Y$ and $g : X \to Z$ be two simplicial morphisms where X, Y and Z are simplicial sets with effective homology. Then $P_{(f,g)}$ is a simplicial set with effective homology.

The proof of this theorem is the above construction which produces Algorithm 5.38.

5.3.2.3.1 By-product algorithms Once we have Algorithm 5.38, it can be used in several interesting cases. For instance, as we commented at the beginning of this section, many of the usual constructions in Topology can be built from the pushout. Let us consider two particular cases: the wedge and the join of simplicial sets with effective homology.

Definition 5.52. Given X and Y 1-reduced spaces with base points $x_0 \in X$ and $y_0 \in Y$, then the *wedge* $X \vee Y$ is the quotient of the disjoint union $X \amalg Y$ obtained by identifying x_0 and y_0 to a single point.

Consider the pushout diagram

$$\begin{tikzcd}
X \arrow[r, "i_1"] \arrow[d, "i_2"] & Y \arrow[d] \\
Z \arrow[r] & P
\end{tikzcd}$$

where X is the one-point simplicial set and Y, Z are 1-reduced simplicial sets and i_1, i_2 are morphism from X to the base point of Y and Z respectively. Then, $P_{(i_1,i_2)}$ is the wedge $Y \vee Z$.

Algorithm 5.53.
Input: two 1-reduced simplicial sets X and Y with effective homology.
Output: the effective homology of $X \vee Y$.

Definition 5.54. Given X and Y spaces, one can define the space of all lines segments joining points in X to points in Y. This is the *join* $X \bowtie Y$, the quotient space of $X \times Y \times I/ \sim$, under the identifications $(x, y_1, 0) \sim (x, y_2, 0)$ and $(x_1, y, 1) \sim (x_2, y, 1)$. Thus we are collapsing the subspace $X \times Y \times (0)$ to X and $X \times Y \times (1)$ to Y.

Consider the pushout diagram

$$\begin{tikzcd}
X \times Y \arrow[r, "p_X"] \arrow[d, "p_Y"] & X \arrow[d] \\
Y \arrow[r] & P
\end{tikzcd}$$

where p_X and p_Y are the projections. Then, $P_{(p_X,p_Y)}$ is the join $X \bowtie Y$.

Algorithm 5.55.
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $X \bowtie Y$.

5.3.2.3.2 Implementation The algorithms explained throughout this subsection have been implemented as a new module for the Kenzo system. The set of programs we have developed (with about 1600 lines) allows the computation of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$, and the construction of its version with effective homology when the effective homologies of X, Y and Z are available.

In the development of the new module for Kenzo that allows one to construct the pushout associated with two simplicial morphisms, the first step has consisted in implementing the algorithms presented in Subsubsection 5.3.2.2 which were not available in Kenzo (to be more concrete, algorithms 5.37, 5.41, 5.42, 5.43, 5.44, 5.45, 5.46, 5.47, 5.48 and 5.49). Subsequently, applying the construction previously explained, we have implemented Algorithm 5.38, that is to say, a program that allows us to construct the effective homology of the pushout. Eventually, we have used that program in order to implement algorithms 5.53 and 5.55.

Among the bunch of implemented functions, we highlight the following ones.

pushout *f* *g* *[Function]*

Build a simplicial set with effective homology, namely the pushout of the simplicial morphisms f: $X \to Y$ and g: $X \to Z$ where X, Y and Z are simplicial sets with effective homology.

wedge *smst1* *smst2* *[Function]*

Build a simplicial set with effective homology, namely the wedge of *smst1* and *smst2*, that are two simplicial sets with effective homology.

join *smst1* *smst2* *[Function]*

Build a simplicial set with effective homology, namely the join of *smst1* and *smst2*, that are two simplicial sets with effective homology.

Several examples are provided in the following subsection to provide a better understanding of the new tools.

5.3.3 Examples

In this subsection we present four examples of application of the programs we have developed to build the pushout of two simplicial morphisms. In the first case, we consider the particular case of the wedge of Eilenberg MacLane spaces ($K(\mathbb{Z}, 2) \vee K(\mathbb{Z}, 2)$). As a second example, we will show the join of two spheres. A sophisticated example giving a geometrical construction of $P^2(\mathbb{C})$ is presented, too. Finally, a way of computing some homotopy groups of the suspension of the classifying space of the group $SL_2(\mathbb{Z})$ is given.

We consider a fresh Kenzo session, where our pushout Kenzo module has been loaded.

Wedge of $K(\mathbb{Z}, 2)$ and $K(\mathbb{Z}, 2)$

Let us present here an example of the use of the wedge of spaces. In this case we want to compute the first six homology groups of the wedge of $K(\mathbb{Z}, 2)$ and $K(\mathbb{Z}, 2)$. First of all, we build the Eilenberg-MacLane space $K(\mathbb{Z}, 2)$.

```
> (setf bkz (k-z 2)) ✠
[K13 Abelian-Simplicial-Group]
```

We construct the wedge of $K(\mathbb{Z}, 2)$ and $K(\mathbb{Z}, 2)$.

```
> (setf bkzwbkz (wedge bkz bkz)) ✠
[K41 Simplicial-Set]
```

Finally, homology groups can be computed as usual.

```
> (homology bkzwbkz 0 6) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :
Component Z
Component Z
Homology in dimension 3 :

Homology in dimension 4 :
Component Z
Component Z
Homology in dimension 5 :

```

Join of S^2 and S^3

Let us present here an example of the use of the join of spaces. In this case we want to compute the first seven homology groups of the join of the spheres S^2 and S^3. Let us note that the join of the spheres S^n and S^m is the sphere S^{n+m+1}; so, in our case the only non null homology groups should be 0 and 6. We construct the spheres S^2 and S^3.

```
> (setf s2 (sphere 2)) ✠
[K331 Simplicial-Set]
> (setf s3 (sphere 3)) ✠
[K336 Simplicial-Set]
```

We construct the join $S^2 \bowtie S^3$.

```
> (setf s2js3 (join s2 s3)) ✠
[K353 Simplicial-Set]
```

Lastly, homology groups can be computed getting the expected result.

```
> (homology s2js3 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :
Component Z
```

$P^2(\mathbb{C})$

This example which gives a geometrical construction of $P^2(\mathbb{C})$ [Ser10] was suggested by Francis Sergeraert. Take S^2 and construct the first stage of the Whitehead tower. We access to the current definition of the sphere of dimension 2 stored in the variable `s2`.

```
> s2 ✠
[K331 Simplicial-Set]
```

We build the fundamental cohomology class.

```
> (setf ch2 (chml-clss s2 2)) ✠
[K547 Cohomology-Class on K331 of degree 2]
```

We construct the fibration over the sphere canonically associated with the above cohomology class.

```
> (setf f2 (z-whitehead s2 ch2)) ✠
[K548 Fibration K331 -> K1]
```

Finally, the total space from the fibration is generated.

```
> (setf x3 (fibration-total f2)) ✠
[K554 Simplicial-Set]
```

Then `x3` has the homotopy type of the 3-sphere S^3. More precisely `x3`$= s2\times_{\texttt{f2}}K(Z,1)$ with `f2` an appropriate twisting function producing S^3 as a total space. It is easy to deduce a projection $f : X3 \rightarrow S2$.

```
> (setf f (build-smmr ; the function to construct a simplicial morphism
           :sorc x3 ; the source simplicial set
           :trgt s2 ; the target simplicial set
           :degr 0  ; the degree of the morphism
           :sintr #'(lambda (dmns gmsm) ; the map
                             (declare (ignore dmns))
                          (absm (dgop1 gmsm) (gmsm1 gmsm)))
           :orgn '(proj ,x3 ,s2))) ✠
[K559 Simplicial-Morphism K554 -> K331]
```

Taking the pushout of this f and the map $g : X3 \rightarrow *$ (where $*$ is the simplicial set with just one vertex),

```
> (setf unipunctual (build-finite-ss '(x))) ✠
[K25 Simplicial-Set]
>(setf g (build-smmr ; the function to construct a simplicial morphism
           :sorc x3 ; the source simplicial set
           :trgt unipunctual ; the target simplicial set
           :degr 0  ; the degree of the morphism
           :sintr #'(lambda (dmns gmsm) ; the map
                          (if (and (equal dmns 0)
                                   (equal gmsm (bsgn x3)))
                              'x
                            nil))
           :orgn '(proj ,x3 ,unipunctual))) ✠
[K560 Simplicial-Morphism K554 -> K25]
```

then the pushout is $P^2(\mathbb{C})$.

```
> (setf p (pushout f g)) ✠
[K566 Simplicial-Set]
```

It can be checked that our programs obtain the right homology groups that are $(\mathbb{Z}, 0, \mathbb{Z}, 0, \mathbb{Z}, 0, 0, \ldots)$:

```
> (homology p 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :
Component Z
Homology in dimension 3 :

Homology in dimension 4 :
Component Z
Homology in dimension 5 :

Homology in dimension 6 :
```

$SL_2(\mathbb{Z})$

This example allows us to compute some homotopy groups of the suspension of $SL_2(\mathbb{Z})$ (the group of 2×2 matrices with determinant 1 over $\mathbb{Z}$, with the group operations of ordinary matrix multiplication and matrix inversion), the interest in computing homotopy groups of this kind of spaces can be seen in [MW10]. In [Ser80], it was explained that $SL_2(\mathbb{Z})$ is isomorphic to the amalgamated sum $\mathbb{Z}_4 *_{\mathbb{Z}_2} \mathbb{Z}_6$. Then, this is a pushout in the category of groups, and, in this case, applying the functor $K(-, 1)$ we obtain also a pushout (see [Bro82]):

$$\begin{tikzcd}
K(\mathbb{Z}_2, 1) \arrow[r, "i_1"] \arrow[d, "i_2"] & K(\mathbb{Z}_4, 1) \arrow[d] \\
K(\mathbb{Z}_6, 1) \arrow[r] & P
\end{tikzcd}$$

Then, we can compute the homology groups of $SL_2(\mathbb{Z})$ thanks to the programs developed to construct the pushout of simplicial sets and the programs presented in [RER09] which allow us to construct $K(G, 1)$ for G a cyclic group (in this case $K(\mathbb{Z}_2, 1)$, $K(\mathbb{Z}_4, 1)$ and $K(\mathbb{Z}_6, 1)$). Namely, we proceed as follows. First of all, we construct the spaces $K(\mathbb{Z}_2, 1)$, $K(\mathbb{Z}_4, 1)$ and $K(\mathbb{Z}_6, 1)$.

```
> (setf kz2 (k-zp-1 2)) ✠
[K2 Abelian-Simplicial-Group]
> (setf kz4 (k-zp-1 4)) ✠
[K15 Abelian-Simplicial-Group]
> (setf kz6 (k-zp-1 6)) ✠
[K28 Abelian-Simplicial-Group]
```

Subsequently, we define the two simplicial morphisms f and g by means of a function called `kzps-incl` which represents the inclusion between $K(\mathbb{Z}_n, 1)$ and $K(\mathbb{Z}_m, 1)$.

```
(defun kzps-incl (n m)
  (declare (number n m))
  (let ((i (/ m n)))
    (build-smmr ; the function to construct a simplicial morphism
     :sorc (k-zp-1 n) ; the source simplicial set
     :trgt (k-zp-1 m) ; the target simplicial set
     :degr 0 ; the degree of the simplicial morphism
     :sintr #'(lambda (dmns gmsm) ; the map
                  (absm 0 (make-list dmns :initial-element i)))
     :orgn  `(inclusion between (k-g-1 ,n) and (k-g-1 ,m)))))
```

```
> (setf kz2-kz4 (kzps-incl 2 4)) ✠
[K40 Simplicial-Morphism K2 -> K15]
> (setf kz2-kz6 (kzps-incl 2 6)) ✠
[K41 Simplicial-Morphism K2 -> K28]
```

Eventually, we construct the pushout of `kz2-kz4` and `kz2-kz6`.

```
> (setf p (pushout kz2-kz4 kz2-kz6)) ✠
[K52 Simplicial-Set]
```

Due to the fact that the abelianization map $SL_2(\mathbb{Z}) \to \mathbb{Z}/12\mathbb{Z}$ induces an isomorphism on integral homology, see [Knu], then, the homology groups of $SL_2(\mathbb{Z})$ are the same that the ones of $K(\mathbb{Z}/12\mathbb{Z}, 1)$. It can be checked that our programs obtain the right homology groups that are $(\mathbb{Z}, \mathbb{Z}/12\mathbb{Z}, 0, \mathbb{Z}/12\mathbb{Z}, 0, \mathbb{Z}/12\mathbb{Z}, \ldots)$:

```
> (homology p 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :
Component Z/12Z
Homology in dimension 2 :

Homology in dimension 3 :
Component Z/12Z
Homology in dimension 4 :

Homology in dimension 5 :
Component Z/12Z
Homology in dimension 6 :
```

Therefore, we have been able to compute the homology groups of $SL_2(\mathbb{Z})$. Now, we apply the suspension constructor to the pushout, see Definition 1.28. Subsequently, we can compute, using the Kenzo system, some homotopy groups of the suspension of $SL_2(\mathbb{Z})$ thanks to the algorithm explained in [Rea94] and using the programs developed in [RER09]. Namely, the first homotopy groups of the suspension of $SL_2(\mathbb{Z})$, denoted by $\Sigma(SL_2(\mathbb{Z}))$ are $\pi_2(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z}$, $\pi_3(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z}$ and $\pi_4(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$. The file with the procedure to compute these homotopy groups can be seen in [Her11].

5.3.4 Integration of the pushout in our framework

In the last two examples of the previous subsection, the definition of simplicial morphisms for the construction of the pushout has been shown. Namely, to define a simplicial morphism an instance of the class `SIMPLICIAL-MRPH` (which is a subclass of the `MORPHISM` class) is necessary.

The relevant slots of a `SIMPLICIAL-MRPH` instance are `sorc`, the source object of type `SIMPLICIAL-SET`; `trgt`, the target object of type `SIMPLICIAL-SET`; `degr`, the degree of the morphism; `sintr`, the internal lisp function defining the effective mapping between simplicial sets; and `orgn`, used to keep a record of information about the object.

The definition of the Lisp function installed in the `sintr` slot is the main hindrance in order to integrate the whole functionality of the pushout in *fKenzo*. The definition of this lisp function is ad-hoc in most of the cases (although in some particular cases, such as the wedge or the join, can be defined in general) and involves a study of the internal structure of the source and target simplicial sets and a knowledge of the definition of Lisp functions, and such a meticulous study is difficult to integrate in our framework, at least in an easy and *usable* way (i.e. without giving access to the internal Common Lisp code).

Nevertheless, some of the functionality developed in the new module can be integrated in *fKenzo*, namely the functionality related to construct the wedge and the join of simplicial sets, since it does not involve, at least in an explicit way, the construction of simplicial morphisms and we can use them like other Kenzo constructors such as cartesian or tensor products. This subsection is devoted to explain the necessary steps to incorporate both the wedge and join functionality to the *Kenzo* framework.

To enhance the framework with the functionality related to wedge and join constructions, we have developed a plug-in following the guidelines given in Subsubsection 3.1.2.

This new plug-in will allow us to construct the wedge and the join of simplicial sets and use them as any other object of the framework. The plug-in used to include the functionality about these constructors references the following resources:

```
<code id="pushout">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> pushout-list.lisp </data>
   <data format="Kf/microkernel"> pushout-m.lisp </data>
   <data format="Kf/adapter"> pushout-a.lisp </data>
</code>
```

As we claimed in Subsubsection 3.1.2, if we want to include new functionality in the Kenzo internal server and make it available outside the framework, all the components must be broadening. Let us explain each one of the referenced resources.

We want to introduce two new kinds of objects in our system (the wedge and the join of simplicial sets), then, it is necessary to provide a representation for that objects in our system. To that aim, we have defined two new elements: `wedge` and `join`. Both elements are defined as elements of the `SS` type, so, they can be used as any other element of this group, and have two children of some of the types `SS`, `SG` or `ASG` (see Figure 5.16).

As we explained in Subsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the `XML-Kenzo.xsd` file is modified to include the join and the wedge elements, the external server is able to check the correctness of requests such as:

```
<constructor>
    <join>
      <wedge>
        <sphere>3</sphere>
        <sphere>4</sphere>
      </wedge>
      <k-z>2</k-z>
    </join>
</constructor>
```

The `pushout-list.lisp` file includes the functionality defined in Subsection 5.3.2 to

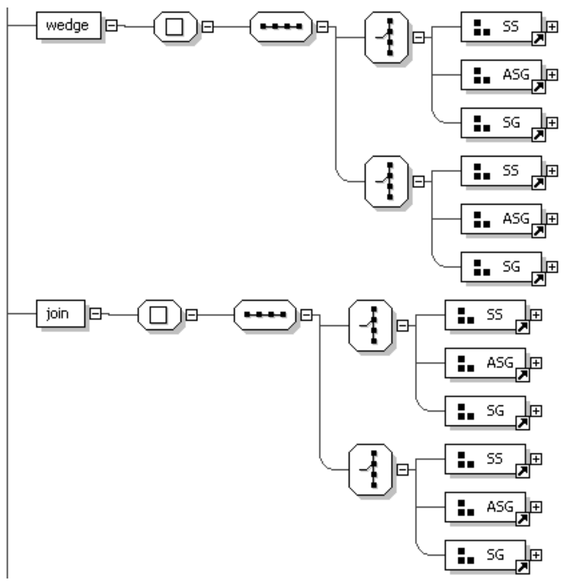


Figure 5.16: join and wedge elements in XML-Kenzo

extend the Kenzo system. Moreover, this file includes the functionality that enhances the `xml-kenzo-to-kenzo` function of the internal server to process the construction of objects from the `join` and `wedge` elements. For instance, if the internal server receives the above request. The instruction

```
(join (wedge (sphere 3) (sphere 4)) (k-z 2))
```

is executed in the Kenzo kernel. As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned.

The `pushout-m.lisp` file defines two new construction modules for the microkernel called `join` and `wedge`. The procedures implemented in this module follows the guidelines explained in Subsubsection 2.2.3.3. In this case neither of the modules checks any property to construct objects when they are activated since the requests coming from the external server related to wedge and join are always safe; that is, all the requests related to wedge and join constructions that can produce errors are stopped in the external server and never arrive to the microkernel (since the constraints related to these constructors are type restrictions which are handled in the XML-Kenzo specification).

Finally, we have extended the `SS` Content Dictionary by means of the definition of two new objects: `join` and `wedge`. Therefore, the `pushout-a.lisp` file contains the necessary functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects, devoted to the wedge and the join, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the previous XML-Kenzo request is generated by the adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="join"/>
      <OMA>
        <OMS cd="SS" name="wedge"/>
          <OMA> <OMS cd="SS" name="sphere"/> <OMI>3</OMI> </OMA>
          <OMA> <OMS cd="SS" name="sphere"/> <OMI>4</OMI> </OMA>
      </OMA>
      <OMA> <OMS cd="ASG" name="k-z"/> <OMI>2</OMI> </OMA>
   </OMA>
</OMOBJ>
```

5.3.5 Integration of the pushout in the *fKenzo* GUI

In this subsection we are going to present the necessary resources to extend the *fKenzo* GUI with the functionality about `wedge` and `join` constructors. As we presented in Section 3.2 one of the modules which customizes *fKenzo* is the Simplicial Set module. This module contains the elements that represent the simplicial set constructors of Kenzo: options to construct spaces from scratch (spheres, Moore spaces, finite simplicial sets, simplicial sets coming from simplicial complexes and so on) and from other spaces (for instance, cartesian products). We have enhanced this module by means of the necessary ingredients to construct joins and wedges of simplicial sets.

In Section 3.2 a *Simplicial Set* module with two files, `simplicial-sets-structure` (which defined the structure of the graphical constituents) and `simplicial-sets-functionality` (which provided the functionality related to the graphical constituents), was presented.

In Subsection 5.1.4, the original Simplicial Set module was improved to include the functionality about simplicial complexes. In particular, we modified the files `simplicial-sets-structure` and `simplicial-sets-functionality` referenced by the original module and we also included a reference to the plug-in about simplicial complexes.

Now, we have extended this module to include the functionality about the wedge and the join of simplicial sets. Namely, it also references the plug-in presented in the previous subsection and, in addition, both `simplicial-sets-structure` and `simplicial-sets-functionality` documents have been upgraded in order to allow the use of wedge and join constructors in the *fKenzo* GUI.

We have defined four new graphical elements, using the XUL specification language, in the `simplicial-sets-structure` file:

- A new menu option called `Join` included in the Simplicial Sets menu.
- A new menu option called `Wedge` included in the Simplicial Sets menu.

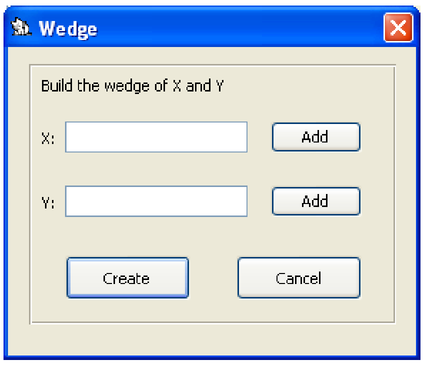


Figure 5.17: Wedge window

- A window called `Join` (very similar to the window defined to construct the cartesian product of two simplicial sets).
- A window called `Wedge` as above.

The functionality stored in the `simplicial-sets-functionality` document related to these components works as follows. The functionality is analogous for wedge and join options; so let us explain just the wedge one. A function acting as event handler is associated with the `Wedge` menu option; this function shows the `Wedge` window (see Figure 5.17) if at least a simplicial set, a simplicial group or an abelian simplicial group was built previously in *fKenzo*; otherwise, it informs the user that at least one object of one of those types must be constructed before using the `Wedge` option.

From the `Wedge` window, the user must select two objects from the two `Add` buttons; the lists of objects shown by the `Add` buttons only contain the objects of types simplicial set, simplicial group and abelian simplicial group.

Once the user has selected two objects $X1$ and $X2$, when it presses the `create` button of the `Wedge` window, an OpenMath request is generated. Subsequently, our framework is invoked with the OpenMath request. The wedge of $X1$ and $X2$ is constructed and its identification number is returned. Eventually, *fKenzo* adds to the list of constructed spaces (situated in the left side of the main tab of the *fKenzo* GUI) the new simplicial set. Figure 5.18 shows the control and navigation submodel (with the Noesis notation) describing the construction of the wedge of two simplicial sets from the `Wedge` menu option.

Moreover, we have also modified the stylesheet used to present in the *fKenzo* GUI the spaces constructed, using their mathematical notation (see Subsubsection 3.2.4.3). This stylesheet is modified in the folder of the *fKenzo* distribution to present with standard notation both wedge and join spaces.

Let us present an example, for instance if we have constructed the spaces S^3 and $M(\mathbb{Z}/2\mathbb{Z},4)$ in a *fKenzo* session, we can build the space $S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4)$ and subsequently the space $(S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4)) \bowtie (S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4))$ from `Wedge` and `Join` menus

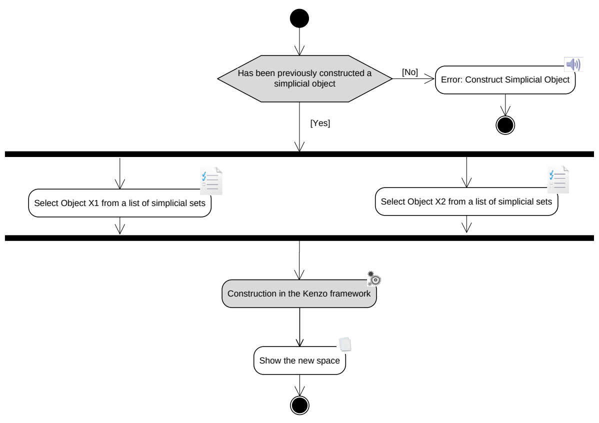

Has been previously constructed a simplicial object
[No]
Error: Construct Simplicial Object
[Yes]
Select Object X1 from a list of simplicial sets
Select Object X2 from a list of simplicial sets
Construction in the Kenzo framework
Show the new space

Figure 5.18: Control graph for the construction of the wedge of two simplicial sets

respectively. Finally, the user can ask *fKenzo* to compute the homology groups of these spaces using the `homology` option of the `Computing` menu. The results are shown, as usual, in the `Computing` tab, see Figure 5.19.

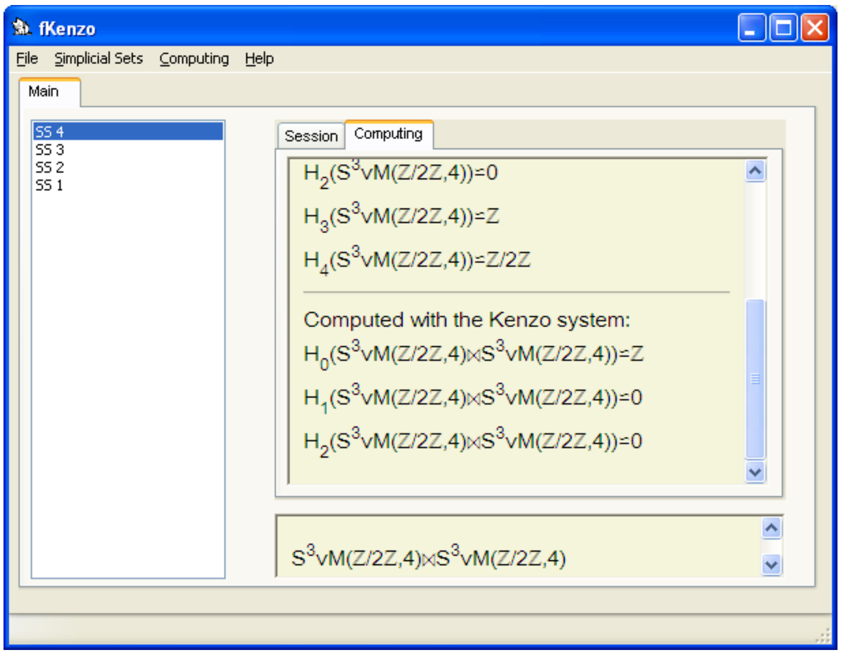
5.3.6 Formalization of the pushout in ACL2

In the same way that we certified our programs for the case of simplicial complexes and digital images, we are interested in verifying the correctness of the implementation of the algorithms which construct the pushout of simplicial sets in ACL2. This task is an ongoing work, and at this moment just some of the algorithms are verified. To be more concrete, the implementation of Algorithms 5.41 (direct sum), 5.43 (cone), 5.47 (suspension functor) and the algorithms associated with SES1 and SES3 cases of Theorem 5.32 have been formalized in ACL2.

The verification of the correctness of those algorithms and also the rest of algorithms presented is part of a wider project which tries to verify the implementation of algorithms which construct new spaces by applying topological constructions. This formalization will be presented in Section 6.2 of the following chapter.

In that section, a methodology to prove the correctness of the construction of new spaces by applying topological constructions is presented. This methodology copes with the higher-order nature of those constructors by means of the simulation of high order logic by means of ACL2 encapsulates.

Therefore, we postpone the presentation of the formalization of the implementation of algorithms of this section to Section 6.2.



$H_2(S^3vM(Z/2Z,4))=0$

$H_3(S^3vM(Z/2Z,4))=Z$

$H_4(S^3vM(Z/2Z,4))=Z/2Z$

Computed with the Kenzo system:

$H_0(S^3vM(Z/2Z,4)\rtimes S^3vM(Z/2Z,4))=Z$

$H_1(S^3vM(Z/2Z,4)\rtimes S^3vM(Z/2Z,4))=0$

$H_2(S^3vM(Z/2Z,4)\rtimes S^3vM(Z/2Z,4))=0$

$S^3vM(Z/2Z,4)\rtimes S^3vM(Z/2Z,4)$

Figure 5.19: Homology groups of $S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4)$ and $(S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4)) \bowtie (S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4))$

Chapter 6

Integration of Computation and Deduction

As we have commented previously, Kenzo has been capable of computing some unknown homology groups. This implies that increasing user's trust in the system is relevant. To this aim, a wide project to apply formal methods in the study of Kenzo was launched several years ago.

One feature of Kenzo is its use of higher order functional programming to handle spaces of infinite dimension. Thus, the first attempts to apply theorem proving assistants in the analysis of Kenzo were oriented towards higher order logic tools. Concretely, the Isabelle/HOL proof assistant was used to verify in [ABR08] a very important algorithm in Homological Algebra: the Basic Perturbation Lemma (Theorem 6.4.2.1). In the same line, we can find the work of [DR11] where the Effective Homology of the Bicomplexes (another important result in Homological Algebra) was formalized in Coq. Let us note, however, that these formalizations were related to *algorithms* and not to the real *programs* implemented in Kenzo. The problem of extracting programs from the Isabelle/HOL proofs has been dealt with in [ABR10], but even there the programs are generated in ML, far from Kenzo.

When talking about theorem proving and Kenzo, it is easy to think about ACL2. ACL2 is, at the same time, a programming language, a logic for specifying and proving properties of the programs defined in the language and a theorem prover supporting mechanized reasoning in the logic. The ACL2 programming language is an extension of an applicative subset of Common Lisp, and the logic is first-order, in which formulas do not have quantifiers and all the variables in them are implicitly universally quantified.

Since both Kenzo and ACL2 are Common Lisp programs, we can undertake the task of verifying real Kenzo code in ACL2. For instance, ACL2 has been successfully used to study some critical fragments of Kenzo in [MMRRR09] (where the programs to deal with degeneracies in Kenzo were proved correct by means of ACL2). This chapter is devoted to present the formalization of other Kenzo fragments in ACL2.

Let us recall a simplified way of understanding the user interaction with the Kenzo system, explained in Subsection 1.2.2. As a first step, the user constructs some *initial spaces* (mainly, under the form of simplicial sets) by means of some built-in Kenzo functions; then, in a second step, he constructs new spaces by applying topological constructions (as Cartesian products, loop spaces, and so on); as a third, and final, step, the user asks Kenzo for computing the homology groups of the spaces.

It is worth noting that the first step, the construction of initial spaces (that is, spaces which does not involve other spaces), can be modeled in a first order logic. On the contrary, steps 2 and 3 need higher order functional programming. Thus, at first glace, we could think that we can only use ACL2 to verify the construction of initial spaces; staying the formalization of steps 2 and 3 beyond the scope of ACL2. However, we will see that this is not true, since we have some means to simulate higher order logic in ACL2.

The rest of this chapter is organized as follows. Some necessary concepts about ACL2 will be introduced in Section 6.1. Section 6.2 is devoted to present the formalization (and verification) of the initial Kenzo simplicial set constructors; this corresponds with the first step of Kenzo user's interaction. Section 6.3 is devoted to present a way of modeling mathematical structures in ACL2. That modeling is the foundation to develop an ACL2 infrastructure to formalize and verify the construction of Kenzo spaces from other ones applying topological constructors (that is to say, the second step of Kenzo user's interaction) which is explained in Section 6.4. The formalization of the computation of homology groups of spaces remains as further work.

6.1 ACL2 concepts

In this section some interesting features of ACL2 are presented. On the one hand, a way of simulating higher order logic in ACL2 is presented in Subsections 6.1.1 and 6.1.2. On the other hand, a way of increasing the efficiency of ACL2 programs is explained in Subsection 6.1.3.

6.1.1 Functional Instantiation

In this subsection, an introduction to *functional instantiation* in ACL2 is given. The *functional instantiation* rule is the way of using higher order results in the ACL2 logic. A more detailed description of this rule can be found in [KMM00b, KMM00a, BGKM91, KM01]. This rule will be used in this chapter.

As we have said, the *functional instantiation* rule is the way of using higher order results in the ACL2 logic. A typical higher order result has the following form: "For all function f which verifies the property P, then f also satisfies the property Q". To formalize this result in ACL2 we use an encapsulate, in which we assume the existence of a function `f` which satisfies the property P described by means of the formula `P`. Subsequently, once that the encapsulate has been admitted, the formula `Q` which describes the property Q is proved. This higher order result ensures that any other function `g`, which satisfies the property `P`, fulfills the property `Q`, too.

The ACL2 system does not use the functional instantiation rule in an automatic way, but it is applied only if the user indicates it with an appropriate *hint* (hints, see [KMM00b, KM], are advices to guide ACL2 proofs). Let us retake the example presented in Subsection 1.3.2 where the following encapsulate, which assumes the existence of an associative and commutative binary function, has been admitted.

```
(encapsulate
    ; the signature
    (((op * *) => *))

    ; the witness
    (local (defun op (a b) (+ a b)))

    ; the axioms
    (defthm op-associative (equal (op (op x y) z) (op x (op y z))))

    (defthm op-commutative (equal (op x y) (op y x))))
```

After admitting the above encapsulate, we can prove in ACL2 the following property.

```
(defthm associativity-commutativity
  (equal (op y (op x z)) (op x (op y z))))
```

Then, we can verify the same property for every associative and commutative binary function. For example, if we consider the binary function `op2`:

```
(defun op2 (x y) (* x y))
```

which is both associative and commutative, as is proved with the following two theorems:

```
(defthm op2-associative
  (equal (op2 (op2 x y) z) (op2 x (op2 y z))))

(defthm op2-commutative
  (equal (op2 x y) (op2 y x))))
```

then, we can obtain the desired property.

```
(defthm associativity-commutativity-op2
  (equal (op2 y (op2 x z)) (op2 x (op2 y z)))
  :hints (("Goal" :by (:functional-instance associativity-commutativity ((op op2))))))
```

The expression which appears from the second line of the above theorem is the *hint*. In this case, it is a functional instantiation hint (`:functional-instance`), in which we reuse the property described by means of `associativity-commutativity` replacing the `op` symbol function with `op2`.

This means that once we have proved a result for the encapsulate functions, we do not need to prove over and over again the same result for concrete functions which satisfy the properties stated in the encapsulate but we just instantiate the general result. In this way, higher order results can be expressed in ACL2.

6.1.2 ACL2 Generic Theories

As we have seen in the previous subsection, we can prove results about functions just assuming some properties which define the functions partially in ACL2. These results are general in the sense that they are true for every set of functions which fulfill the assumed properties, and can be reused by means of the functional instantiation process. However, the process of functional instantiation can be tedious when we are dealing with a great amount of theorems, or we need to define new functions from the ones which we have assumed some properties. To overcome this situation a generic instantiation tool was developed by Martín-Mateos et al. [MMAHRR02]. This tool will be intensively used in this chapter.

A *theory* in ACL2 is a list of events (definitions and theorems) admitted by the system. A *generic theory* is a theory which depends on the existence of a set of functions which are partially specified by means of some assumed properties using the *encapsulate principle*; those functions will be called *generic functions.*

For instance, in the example presented in Subsection 1.3.2 we have accepted the existence of a binary function `op` which verifies both the associativity and the commutativity property using the *encapsulate principle.*

```
(encapsulate
    ; the signature
    (((op * *) => *))

    ; the witness
    (local (defun op (a b) (+ a b)))

    ; the axioms
    (defthm op-associative (equal (op (op x y) z) (op x (op y z))))

    (defthm op-commutative (equal (op x y) (op y x))))
```

In this case the unique *generic function* is the `op` function. Once this encapsulate has been admitted, we can establish a set of events (definitions and theorems) depending on the generic function and their assumed properties. For instance:

```
(defun op-op (a b c)
    (op a (op b c)))

(defthm op-op-property
    (equal (op-op a b c) (op-op c b a)))
```

In this case, the events `op-op` and `op-op-property` constitute a *generic theory* for the `op` function.

The main feature of *generic theories* is that they are used to formalize higher order results in ACL2. To use these higher order results, we must provide concrete versions of the *generic functions*. Moreover, we must prove that the concrete versions have the same properties that the ones which specify the generic functions. Subsequently, we can construct concrete versions of the events provided by the generic theory, replacing the occurrences of the generic functions with the concrete ones. To prove the concrete versions of the theorems which are provided by the generic theory, we use the functional instantiation rule where the generic functions are replaced with the concrete ones.

For instance, the function `op2` presented in the previous subsection:

```
(defun op2 (x y) (* x y))
```

satisfies the properties demanded to the `op` function.

```
(defthm op2-associative
  (equal (op2 (op2 x y) z) (op2 x (op2 y z))))

(defthm op2-commutative
  (equal (op2 x y) (op2 y x))))
```

Therefore, we can construct a concrete version of the function, `op-op`, provided by the generic theory just replacing the occurrences of the `op` function with the `op2` one:

```
(defun op-op-2 (a b c)
    (op2 a (op2 b c)))
```

Now we can use the *functional instantiation* rule to obtain the concrete version of the theorem `op-op-property`.

```
(defthm op-op-property-2
  (equal (op-op-2 a b c) (op-op-2 c b a)))
  :hints (("Goal" :by (:functional-instance op-op-property
                         ((op-op op-op-2))))))
```

The set of concrete versions of the events provided by the generic theory is called *instance* of the generic theory, and the process which obtains them, *generic instantiation.*

Let us explain now, how the generic instantiation tool of Martín-Mateos et al. is used to make the generic instantiation process easier.

To automatize the generic instantiation process using the generic instantiation tool, we proceed as follows. First of all, we assume the existence of some generic functions which verify some properties using the encapsulate principle.

```
(encapsulate
  ; Signatures
  ((f1 ...)
   ...
   (fn ...))
  ; Witnesses
  (local (defun f1 (...) ...))
  ...
  (local (defun fn (...) ...))
  ; Axioms
  (defthm axiom-1 ...)
  ...
  (defthm axiom-m ...))
```

Subsequently, the events which depend on the generic functions and their properties are stated, obtaining a *generic theory* about the *generic functions.*

```
; The following functions are defined from the generic functions f1,..., fn.
(defun g1 (...) ...)
...
(defun gk (...) ...)
; The statement of the  following theorems can involve both the f1,..., fn
; and the g1,..., gk functions. Moreover, they are proved thanks to the assumed
; properties about the f1,..., fn functions, that is to say, axiom-1,..., axiom-m.
(defthm theorem-1 ...)
...
(defthm theorem-s ...)
```

Eventually, we define a constant called `*<theory>*` where we store the events of the generic theory in the same order admitted in ACL2.

```
(defconst *<theory>*
  '((defun g1 (...) ...)
    ...
    (defun gk (...) ...)
    (defthm theorem-1 ...)
    ...
    (defthm theorem-s ...)))
```

For each generic theory a different constant must be used. In this case, `<theory>` represents the sole name which denominates the theory.

It is worth noting that not all the events which depend on the properties of the generic functions appear in the generic theory. This is due to the fact that in ACL2, we usually develop auxiliary definitions and lemmas which help in the proof of the main results.

Once we have defined the constant with the sequence of events to instantiate, to perform the functional instantiation process, we need a set of concrete functions which replace the generic functions that define the theory. These functions, and their relation with the generic ones, are provided by means of an association list.

To instantiate a definition or a theorem in the generic theory, we replace all the occurrences of the generic functions with the concrete ones, according to the relation given by the association list. Moreover, it is necessary to choose new names for the obtained instances. In addition, in the case of theorems, a hint to prove the concrete version of the theorem, as a functional instance of the generic theorem, is provided.

This process is performed by means of the generic instantiation tool macro `definstance-*<theory>*` (where `<theory>` is the name of the desired generic theory), whose arguments are the initial association list, which relates the generic functions to the concrete ones, and a string which is used as suffix to build the new names of the instantiated events, namely the suffix is appended to the generic names.

Let us present a simple example in order to clarify the use of this tool. Let us retake the example presented previously in this subsection. We have admitted the existence of an associative and commutative binary function called `op` using the encapsulate principle. Moreover, we have defined a *generic theory*, let us call it `*small-theory*`, which consists of the events `op-op` and `op-op-property` (the auxiliary lemmas that were necessary to prove `op-op-property` are not part of the generic theory). Then, we assign to the constant `*small-theory*` the list of events which constitute the generic theory and the `make-generic-theory` macro is evaluated with this constant as argument.

```
(defconst *small-theory*
  '((defun op-op (a b c) ...)
    (defthm op-op-property ...)))

(make-generic-theory *small-theory*)
```

Now the `definstance-*small-theory*` macro is available and can be used to instantiate the generic theory. For instance with the function `op2`, which satisfies the properties assumed for the function `op`, and the string `"-2"`, which will be used as suffix to build the new names of the events `op-op` and `op-op-property`, we instantiate the generic theory as follows:

```
(definstance-*small-theory*
  ((op op2)) "-2")
```

When the above macro is evaluated the following events are automatically defined in ACL2.

```
> :pe op-op-2 ✠
         14:x(DEFINSTANCE-*SMALL-THEORY* (#) "-2")
             \
>L            (DEFUN OP-OP-2 (A B C)
                     (OP2 A (OP2 B C)))

> :pe op-op-property-2 ✠
 14:x(DEFINSTANCE-*SMALL-THEORY* (#) "-2")
             \
>             (DEFTHM
               OP-OP-PROPERTY-2
               (EQUAL (OP-OP-2 A B C) (OP-OP-2 C B A))
```

In this way the effort associated with the generic instantiation process is dramatically reduced thanks to the generic instantiation tool.

6.1.3 Semantic attachment in ACL2

In this subsection, a tool to prove that efficient programs are correct in ACL2 is described. The idea consists of providing two definitions of a Common Lisp program: an elegant definition that is suitable for reasoning in ACL2, and an efficient definition for evaluation. The bridge between the logical specifications and efficient execution methods, is called "semantic attachment" of the executable code to the logical specification. The term "semantic attachment" was coined for the mechanism in the FOL theorem prover by which the user could attach programs to logical theories, see [Wey80]. Let us present here, this mechanism for ACL2, the interested reader can consult [G$^+$08] where several examples and a detailed description of this tool can be found. This tool will be used in Subsubsection 6.2.4.2.

First of all, we need to introduce the concept of *guards* in ACL2. A successful ACL2 definition adds a new axiom to the ACL2 logic and defines (and generally compiles) the new function symbol in the host Common Lisp. The function is called the *Common Lisp counterpart* of the logical definition. Due to the fact that ACL2 is built on Common Lisp, it can use the Common Lisp counterpart to compute the value of that function instead of deriving that value by repeated reductions using instantiation of the definitional axioms. For instance, if a user evaluates the form `(factorial 5)` (where `factorial` is a function which computes the factorial of a natural number), ACL2 can use the Common Lisp counterpart of `factorial` to compute 120 instead of deriving that value by repeated reductions using instantiation of the definitional axioms.

However, all that glitters is not gold since Common Lisp functions are usually partial (that is to say, they are not defined for all possible inputs) but ACL2 functions must be total; then, we cannot always use the Common Lisp counterpart to compute the value of a function. For instance, the function `endp` is defined in Common Lisp to return `t` if its argument is the empty list, `nil` if its argument is a cons pair (a cons pair is just a pair of two objects), and is not defined otherwise. Then, `(endp 2)` is undefined in Common Lisp; and usually causes an error.

The Common Lisp standard introduces the notion of *intended domain*; that is the set of inputs for which a function is defined. For instance, the intended domain for `endp` consists of the cons pairs and the empty list. This notion is formalized in ACL2 with the idea of *guards*. The guard of a function symbol is an expression that checks if the arguments are in the intended domain. Then, ACL2 can invoke the Common Lisp counterpart of a function only if the arguments have been guaranteed to satisfy the guard of that function. For example, we could define the `factorial` function as follows.

```
(defun factorial1 (n)
  (declare (xargs :guard (natp n)))
  (if (zp n)
      1
    (* n (factorial1 (- n 1)))))
```

where `(natp n)` is defined to recognize natural numbers, which are integers higher or equal than 0.

```
(defun natp (x)
  (and (integerp x) (<= 0 x)))
```

ACL2 provides a mechanism, called *guard verification*, of proving that the guards on the input of a function ensure that all the guards in the body of the function are satisfied.

Then, for instance, when the ACL2 theorem prover encounters `(factorial1 5)` it checks that the guard for `factorial1` is satisfied, i.e, `(natp 5)`. Since this is true and the guards for `factorial1` have been verified, we know that all evaluation will stay in the intended domain. So, ACL2 is free to invoke the Common Lisp definition of `factorial1` to compute the answer 120.

On the contrary, if ACL2 encounters a call to `factorial` with a cons pair as input, `(factorial1 '(1 . 2))`, the guard check fails and ACL2 cannot invoke the Common Lisp counterpart. The value of the term is computed by other means, for example, application of the axioms during a proof, or an alternative "safe" Common Lisp function that performs appropriate run-time guard-checking and type-checking at the cost of some efficiency. ACL2 defines such a function in Common Lisp, this function is called *executable counterpart.*

In general, ACL2 evaluation always calls the executable counterpart to evaluate a function call. But if the guard of the function has been verified and the call's argument satisfy the function guard, then the executable counterpart will invoke the more efficient Common Lisp counterpart to do the evaluation.

Let us suppose now that we have verified the guards of `factorial1` and we want to apply the `factorial1` function to a "big" natural number (numbers higher than 1500 are good candidates). The guard check would succeed and the Common Lisp counterpart would be invoked to compute the value. But since `factorial1` is defined recursively, we are likely to get a *stack overflow* error. In spite of being `factorial1` an elegant (and suitable for reasoning) definition of the factorial function, a tail recursive function, as the following one, will be more suitable for the purpose of efficient execution.

```
(defun factorial-aux (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (if (zp n)
      a
    (factorial-aux (- n 1) (* a n))))

(defun factorial2 (n)
  (declare (xargs :guard (natp n)))
  (factorial-aux n 1))
```

One of the advantages of ACL2 is that models permit both execution and formal analysis. However, if we define `factorial` for analysis, we may make it impossible to execute on examples of interesting scale; on the contrary, if we define it for executing, we complicate formal proofs. An approach to deal with both ways consists of using the `mbe` and `defexec` ACL2 constructs, which make it possible to write:

```
(defexec factorial (n)
  (declare (xargs :guard (natp n)))
  (mbe :logic (factorial1 n)
       :exec  (factorial2 n)))
```

In addition to normal termination and guard verifications, the `mbe` (must be equal) macro asks a proof obligation that the Common Lisp `(:exec)` counterpart will return the same answer as the logical `(:logic)` definition. In this way, when the theorem prover is reasoning about `factorial` it uses the `factorial1` definition. But, when applications satisfying the guards arise, the tail-recursive `factorial2` function is executed.

This is one of the uses of `defexec` and `mbe`. A more detailed description of the applications of `defexec` and `mbe` can be found in [G$^+$08]. On the one hand, they can be used to keep a natural and logically elegant definition of a function for reasoning in the logic, while attaching a more efficient definition for execution. On the other hand, they also can be used to preserve the use of the natural definition for execution purposes when the natural definition of a function is efficient, but needs to be modified in order to facilitate logical reasoning. We are interested in the latter case; and to be more concrete, in the way of dealing with the implementation of processes which may not terminate.

Let us suppose that we have a process implemented by a Common Lisp function `(f x1 ... xn)`; and in some situations (depending on the value of the inputs `x1`, ..., `xn`) the process implemented by this function may not terminate. Therefore, as we commented at Subsection 1.3.1, this process cannot be directly defined as a function in ACL2.

Consequently, in order to get the function admitted in the ACL2 logic we must explicitly introduce in its logical definition a condition that ensures its termination. In this case, the function which implements such condition will be called `good-input-p`. Having defined the `good-input-p` function, we may now define the `f` function by means of `defexec`, as follows.

```
(defexec f (x1 ... xn)
  (declare (xargs :guard (good-input-p x1 ... xn)))
  (mbe :logic (if (good-input-p x1 ... xn)
                  ; body of the function f
                  ...
                nil)
       :exec ; body of the function f
             ...))
```

It is worth noting that the logical definition of the function includes the condition `(good-input-p x)`, needed to be accepted by the ACL2 principle of definition. This test function can be computationally expensive and it would have to be evaluated in every recursive call of the `f` function, making the logical definition impractical for execution. The use of `mbe` guarantees that once the guards are verified, the `:exec` body, without the expensive test, can be safely used for execution when the function is called on arguments satisfying its guards. Improving in this way the efficiency of the programs.

6.2 Proving with ACL2 the correctness of simplicial sets in the Kenzo system

As we have remarked at the beginning of this chapter, a simplified view of Kenzo user's way of working can be understood in three steps: (1) construction of initial spaces (mainly, under the form of simplicial sets) by means of some built-in Kenzo functions; (2) building of new spaces by applying topological constructions, and, (3) computation of the homology groups of the spaces. In this section, we are going to focus on verifying, using the ACL2 Theorem Prover, the construction of initial spaces under the form of simplicial sets, that is to say, the first step of the usual Kenzo user's interaction.

In particular, we want to verify the correctness of Kenzo statements like the following one.

```
> (sphere 3) ✠
[K1 Simplicial-Set]
```

This means that we want to prove in ACL2 that the returned objects by Kenzo functions which construct initial simplicial sets (such as spheres, Moore spaces, standard simplicial sets and so on) really satisfy the axioms of simplicial sets.

The methodological approach to tackle these proofs has been imported from [MMRRR09]. EAT [RSS90] was the predecessor of Kenzo. The EAT system is also based on Sergeraert's ideas, but its Common Lisp implementation was closer to the mathematical theory. This means a poorer performance (since in Kenzo the algorithms have been optimized), but also that the ACL2 verification of EAT programs is easier. Thus, the main idea is to prove first in ACL2 the correctness of EAT programs, and then, by a domain transformation, to translate the proofs to Kenzo programs. To this aim, an intermediary model, based on *binary numbers*, is employed (this was introduced in [MMRRR09], too).

The organization of this section is as follows. Subsection 6.2.1 is devoted to recall some mathematical preliminaries related to simplicial sets and present their concrete materialization using the EAT, Kenzo and binary representations. Subsection 6.2.2 presents the certification of the Kenzo spheres; firstly studying the case of the sphere S^3 and subsequently tackling the general case. The proofs presented in Subsection 6.2.2 were done from scratch for the family of Kenzo spheres; however, in Subsection 6.2.3 a *generic simplicial set theory* is provided to make the proof of the fact that families of Kenzo initial simplicial sets are really families of simplicial sets easier. Finally, the practicability of the *generic simplicial set theory* is tested by means of three different families of Kenzo simplicial sets (spheres, standard simplicial sets and simplicial sets associated with simplicial complexes) in Subsection 6.2.4,

The interested reader can consult the complete development in [Her11].

6.2.1 Simplicial Sets representations

In this subsection, we present some remarks about simplicial sets which are necessary to understand the rest of this part of the memoir. Moreover, we explain how simplicial sets are encoded using EAT, Kenzo and binary representations and their distance to the ACL2 code that we use in our proofs.

6.2.1.1 The mathematical representation

The most important notion, in this part of the memoir, is the notion of *simplicial sets* presented in Definition 1.17.

Let us remember that, a *simplicial set* K, is a union $K = \bigcup_{q \geq 0} K^q$, where the K^q are disjoints sets, together with functions:

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

subject to the 5 relations:

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

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

The elements of K^q are called *q-simplexes.* A q-simplex x is *degenerate* if $x = \eta_i^{q-1}y$ for some simplex y, and for some degeneracy map η_i^{q-1}; otherwise x is *non degenerate.* Non degenerate simplexes are called *geometric* simplexes, to stress that only these simplexes really have a geometrical meaning; the degenerate simplexes can be understood as *formal* artifacts introduced for technical (combinatorial) reasons.

The essential result stated in Proposition 1.20, which was modeled and proved by means of the ACL2 theorem prover in [ALRRR07], allows us to encode all the elements (simplexes) of *any* simplicial set in a generic way, by means of a structure called *abstract simplex.* More concretely, an *abstract simplex* is a pair (*dgop gmsm*) consisting of a sequence of degeneracy maps *dgop* (which will be called a *degeneracy operator*) and a geometric simplex *gmsm.* The indexes in a degeneracy operator *dgop* must be in strictly decreasing order. For instance, if σ is a non degenerate simplex, and σ' is the degenerate simplex $\eta_1\eta_2\sigma$, the corresponding abstract simplexes are respectively $(\emptyset \;\; \sigma)$ and $(\eta_3\eta_1 \;\; \sigma)$, as $\eta_1\eta_2 = \eta_3\eta_1$, due to equality (2) in Definition 1.17.

Of course, the nature of geometric simplexes depends on the concrete simplicial set we are dealing with, but the notion of abstract simplex allows us a generic handling of all the elements in our proofs.

Equation (2) in Definition 1.17 allows one to apply a degeneracy map η_i over a degeneracy operator *dgop* to obtain a new degeneracy operator. Let us consider, for example, η_4 and the degeneracy operator $\eta_5\eta_4\eta_1$; then $\eta_4\eta_5\eta_4\eta_1 = \eta_6\eta_4\eta_4\eta_1 = \eta_6\eta_5\eta_4\eta_1$. We will use the notation $\eta_i \circ dgop$ for the resulting degeneracy operator; in our example: $\eta_4 \circ (\eta_5\eta_4\eta_1) = \eta_6\eta_5\eta_4\eta_1$.

We can also try to apply a face map ∂_i over a degeneracy operator *dgop*. But now, there are two cases, according to whether the indexes i or $i-1$ appear in *dgop*. If they do not appear, then there is a face that *survives* in the process (for instance: $\partial_4\eta_5\eta_2\eta_0 = \eta_4\partial_4\eta_2\eta_0 = \eta_4\eta_2\partial_3\eta_0 = \eta_4\eta_2\eta_0\partial_2$). Otherwise, the cancelation equation (4) from Definition 1.17 applies and the result of the process is simply another degeneracy operator (example: $\partial_4\eta_5\eta_3\eta_0 = \eta_4\partial_4\eta_3\eta_0 = \eta_4\eta_0$). We will denote by $\partial_i \circ dgop$ the output degeneracy operator (in our examples: $\partial_4 \circ (\eta_5\eta_2\eta_0) = \eta_4\eta_2\eta_0$ and $\partial_4 \circ (\eta_5\eta_3\eta_0) = \eta_4\eta_0$).

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

Formula 6.1.

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

where

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

Note that the degeneracy map expressed in terms of abstract simplexes only affects the degeneracy operator of the abstract simplex; therefore degeneracy maps can be implemented independently from the simplicial set. On the contrary, face maps depend on the simplicial set because, when $\eta_i \notin dgop$ and $\eta_{i-1} \notin dgop$, the application of a face map ∂_i *arrives* until the geometric simplex, and, this, of course, requires some knowledge from the concrete simplicial set where the computation is carried out.

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

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

Proposition 6.2. Let K be a simplicial set, $absm = (dgop \;\; gmsm)$ be a pair where $dgop$ is a degeneracy operator and $gmsm$ is a geometric simplex of K and l be the length of the sequence of degeneracies $dgop$. Then, $absm$ is an abstract simplex of K in dimension q if and only if the following properties are satisfied:

1. $l \leq q$;
2. $gmsm \in K^r$, where $r = q - l$;
3. the index of the first degeneracy in $dgop$ is lower than q.

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

6.2.1.2 The EAT representation

The data structures of the EAT system are organized in two layers: the first layer is composed of algebraic data structures (chain complexes, simplicial sets, and so on) and the second one of standard data structures (lists, trees, and so on) which are representing elements of data from the first layer. This subsubsection is devoted to present the EAT representation of both simplicial sets (which belong to the first layer) and abstract simplexes (which belong to the second layer and are the elements of simplicial sets).

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

Mathematical structures are represented in EAT as *records* with several components of functional nature. Then, a simplicial set, *smst*, is represented internally in the EAT system by means of an instance of the `ss` record. In the EAT organization, a simplicial set is made of 6 slots, but there is just one of them which is relevant for reasoning: the `gdl` slot. This slot contains a lisp function computing any face of any *geometric* simplex of the simplicial set *smst*. The function has three arguments: a face index, a simplex dimension and a geometric simplex.

It is worth noting that EAT does not include the definition of the face map over abstract simplexes. Because, as we have seen in Formula 6.1, this map can be defined in a general way and be specialized for every simplicial set only knowing the face of any *geometric* simplex of the simplicial set. So, this is the approach followed in EAT where the function `ADL` implements the procedure given in Formula 6.1 for encoding the face map.

The `ADL` function has four arguments: a simplicial set `ss`, a face index i, a simplex dimension q and an abstract simplex $absm = (dgop \;\; gmsm)$. The implementation of this function, which follows Formula 6.1, is the following one. First of all, it invokes the function `cmp-d-ls` which takes the face index i and the degeneracy list $dgop$ of the abstract simplex $absm$ as input, and returns two outputs:

- a new degeneracy list $dgop2$, obtained by systematically applying equations (3), (4) and (5) of Definition 1.17 to $dgop$, starting with ∂_i, and,
- an index $i2$ which survives in the previous process, or the symbol `nil` in the case where the equation (4) in Definition 1.17 is applied and the face map is cancelled.

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

Subsequently, if the second output returned by `cmp-d-ls` is `nil`, then the result returned by `ADL` is $(dgop2 \;\; gmsm)$. Otherwise, the `gdl` function associated with the simplicial set `ss` is invoked with arguments $i2$, $q-\{\text{length } dgop\}$ and $gmsm$ to obtain a result like $(dgop3 \;\; gmsm2)$, and, then, the result returned by `ADL` is $((dgop2 \circ dgop3) \;\; gmsm2)$.

In this way the face map over abstract simplexes is encoded in EAT. Now, let us focus on the definition of the degeneracy map over abstract simplexes. EAT does not provide a function like `ADL` for encoding the degeneracy map over abstract simplexes because it is not relevant for computing. However, it includes all the necessary functions to define a function, let us call it `ANL`, which implements the procedure given in Formula 6.1 for encoding the degeneracy map.

In the implementation of the `ANL` function, the function `cmp-ls-ls`, which performs the composition of two degeneracy lists, will be instrumental. The `ANL` function can be defined as follows. It has four arguments: a simplicial set `ss`, a degeneracy index i, a simplex dimension q and an abstract simplex $absm = (dgop \;\; gmsm)$. The implementation of this function, which follows Formula 6.1, is the following one. First of all, it invokes the function `cmp-ls-ls` which takes a list which consists of the degeneracy index i, and the degeneracy list $dgop$ of the abstract simplex $absm$ as arguments, and returns a new degeneracy list $dgop2$, obtained by systematically applying equation (2) of Definition 1.17 to $dgop$, starting with η_i.

Then, the `ANL` function returns as output $(dgop2 \;\; gmsm)$. In this way the degeneracy map over abstract simplexes can be encoded in EAT. It is worth noting that the `ANL` function represents the degeneracy map over abstract simplexes for every simplicial set of EAT because this operation is independent of the chosen simplicial set, as we have seen in Formula 6.1.

Finally, we must focus on the invariant function. The invariant function was included as a test function in the first versions of EAT; this was a natural approach to reflect the *functional coding* introduced by Sergeraert in [Ser90]. However, Sergeraert realized that, from the operational point of view, the invariant function is useless in the computational environment of EAT. It can be useful to test that the elements belong to the structure, but if we suppose that both the programs and their inputs are correct; then, applying successively the invariant test is, obviously, an unnecessary waste. Due to the fact that EAT is a computational tool, the invariant functions were removed from the programs. However, in our context (focussed on verifying) the invariant functions are instrumental to prove the correctness of our programs; so, they must also be defined.

Then, we can implement an invariant function, which is a predicate indicating when a Lisp pair is an abstract simplex of a simplicial set `ss` in EAT. To this aim, we proceed as in the definition of the face map. First of all, we define an auxiliar function, called `inv-gmsm`, which is ad-hoc for each simplicial set `ss`. The `inv-gmsm` test function takes as input a geometric simplex *gmsm* and a dimension q, and returns `t` if *gmsm* is a geometric simplex in dimension q of the simplicial set, and `nil` otherwise.

Finally, as in the case of the face map over abstract simplexes, we have a general invariant function called `INV`. The `INV` function has three arguments: a `inv-gmsm` function, a simplex dimension q and an abstract simplex $absm = (dgop \ \ gmsm)$. The invariant function `INV` translates the conditions from Proposition 6.2 for the concrete simplicial set where l is equal to the length of the *dgop* list:

1. $l \leq q$;
2. *gmsm* is an element of `ss` in dimension r, where $r = q - l$;
3. the first element of *dgop* is lower than q.

In this way, the face, degeneracy and invariant functions are encoded using the EAT representation.

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

Before explaining the Kenzo representation for simplicial sets and abstract simplexes, let us present in the following paragraph the differences between EAT code and the actual ACL2 code that we use in our proofs.

6.2.1.2.1 The EAT model in ACL2 Since the ACL2 programming language is a subset of Common Lisp, the definition of the EAT functions in ACL2, is quite direct. However, due to the nature of ACL2 (applicative, free of side effects and so on), there are some things that have to be defined in a different (but equivalent) way.

First of all, it is worth noting that the functions `ADL`, `ANL` and `INV` can be seen as functions which take as input other functions. This cannot be modeled in ACL2. Then to overcome this pitfall, we define concrete versions of the functions `ADL`, `ANL` and `INV` for

```
(defun cmp-d-ls-dgop (d ls)
  (do ((p ls (cdr p))
       (rsl
         empty-list (let ((j (car p)))
                (cons (cond ((< d j) (1- j))
                            (t (decf d) j))
                              rsl))))
      ((endp p) (nreverse rsl))
    (when (<= 0 (- d (car p)) 1)
      (return (nreconc rsl (rest p))))))
```

```
(defun cmp-d-ls-dgop-do (d p rsl)
  (cond ((endp p) (reverse rsl))
        ((< d (car p))
         (cmp-d-ls-dgop-do d (cdr p)
                   (cons (1- (car p)) rsl)))
        ((and (<= 0 (- d (car p)))
              (<= (- d (car p)) 1))
         (append (reverse rsl) (rest p)))
        (t (cmp-d-ls-dgop-do (1- d)
                (cdr p) (cons (car p) rsl)))))

(defun cmp-d-ls-dgop (d ls)
  (cmp-d-ls-dgop-do d ls nil))
```

Figure 6.1: cmp-d-ls-dgop definition in EAT and in ACL2

each simplicial set. This is a well-known form of *defunctionalization* [Rey98] and can be considered as a safe transformation.

In addition, we must deal with multivalued functions. As we have commented previously, the `cmp-d-ls` EAT function returns two values (a degeneracy list and an index); that is to say, it is a multivalued function. This situation happens in several EAT functions and is handled by means of the `values` Common Lisp function.

ACL2 does not support the `values` function. However, two different options are feasible to deal with multivalued functions in ACL2: (1) split the multivalued function in so many functions as values are returned in the multivalued function; or (2) use the `mv` ACL2 macro. In our case we opted for the former one.

Therefore, we have translated the EAT function `cmp-d-ls` into two ACL2 functions called `cmp-d-ls-dgop` and `cmp-d-ls-indx`.

The last difference between EAT and ACL2 code is related to iterative functions. The only way to iterate in ACL2 is by means of recursion, then we must define recursive versions of the functions `cmp-d-ls-dgop`, `cmp-d-ls-indx` and `cmp-ls-ls` which are defined using an internal loop. The recursive version of `cmp-ls-ls` was presented in [MMRRR09], so, let us concentrate on the recursive version of `cmp-d-ls-dgop` (the case of `cmp-d-ls-indx` is analogous).

The left side in Figure 6.1 contains the *real* EAT code for the `cmp-d-ls-dgop` function. That definition receives as inputs a natural number `d` and a decreasing integer list `ls` (to be interpreted as a degeneracy operator) and returns a decreasing integer list (encoding a degeneracy operator).

The right box of Figure 6.1 contains our ACL2 recursive version of `cmp-d-ls-dgop`. First of all, we have used an auxiliary recursive definition implementing the internal loop, trying to be as faithful as possible with the original version. Also, since destructive updates are not allowed in ACL2, we consider the local variable `rsl` as extra input parameter. Finally, since ACL2 functions have to be total, we have to define a result just in case the inputs were not of the intended type.

The transformations between EAT code and ACL2 code are instances of well-known generic transformations and, therefore, can be considered safe. Therefore, we are working with code as close as possible to the actual EAT one.

Let us present now the Kenzo representation for both abstract simplexes and simplicial sets.

6.2.1.3 The Kenzo representation

Both EAT and Kenzo systems are based on the same Sergeraert's ideas, but the performance of the EAT system is much poorer than that of Kenzo. One of the reasons why Kenzo performs better than EAT is because of a smart encoding of degeneracy operators. Since generating and composing degeneracy lists are operations which appear in an exponential way in most of Kenzo calculations (through the Eilenberg-Zilber theorem [May67]), it is clear that having a better way for storing and processing degeneracy operators is very important. But, no reward comes without a corresponding price, and the Kenzo algorithms are somehow obscured, in comparison to the clean and comprehensible EAT approach.

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

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

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

The face and degeneracy maps over abstract simplexes are implemented using the Kenzo representation by means of the functions `face` and `degeneracy` respectively. Both `face` and `degeneracy` use the functions `1dlop-dgop` and `dgop*dgop` and the `face` slot which are the counterparts of the `cmp-d-ls` and `cmp-ls-ls` EAT functions and `gdl` slot respectively.

Moreover, we can reuse the function `inv-gmsm`, which was previously defined, because it is related to the geometric simplexes of simplicial sets and this part is encoded in the same way for both Kenzo and EAT representations. From this function, the invariant function for abstract simplexes in Kenzo can be defined according to (in this case l is equal to the number of active bits in *dgop*):

1. $l \leq q$;
2. *gmsm* is an element of K^r, where $r = q - l$;
3. $dgop < 2^q$.

As in the EAT case, there is a gap between the Kenzo code and the ACL2 code that we use in our proofs. In particular we found the same problems related to multivalued functions (we have defined the functions `1dlop-dgop-dgop` and `1dlop-dgop-indx` which are the counterparts of `cmp-d-ls-dgop` and `cmp-d-ls-indx` respectively) and functions taking other functions as input; therefore, we used the safe transformation techniques presented in Paragraph 6.2.1.2.1. However, these are the only important differences between the Kenzo and ACL2 code, since all the binary operations of Common Lisp (`logxor` and the like) are present in ACL2. Thus, the ACL2 programs make up an accurate version of the Kenzo ones.

6.2.1.4 The binary representation

The distance between EAT and Kenzo models is too large in order to prove the equivalence between the two representations modulo the change of representation. Then, we have used an intermediary representation introduced by Martín-Mateos et al. in [MMRRR09].

An abstract simplex, *absm*, is represented internally in the binary model by a Lisp object: (*dgop gmsm*) where *dgop* is a list of 0's and 1's coding a strictly decreasing integer list and *gmsm* is a geometric simplex. The strictly decreasing integer list represents a sequence of η operators and is coded as a *unique* list of 0's and 1's. Let us explain this with an example: the degeneracy list (3 1) can equivalently be seen as the binary list (0 1 0 1) in which 1 is in position i if the number i is in the degeneracy list, and 0 otherwise. Then, the abstract simplexes $(\emptyset \;\; \sigma)$ and $(\eta_3\eta_1 \;\; \sigma)$ are implemented in the binary model as $(nil \;\; \sigma)$ and $((0\;\;1\;\;0\;\;1) \;\; \sigma)$, respectively.

The algorithms which implement the face and degeneracy operators are directly inspired from those of Kenzo; then, we have the `1dlop-dgop-binary`, `dgop-dgop-binary` and `face-binary` which are the counterparts of the `1dlop-dgop`, `dgop*dgop` functions and `face` slot respectively. From these functions and slot, we can define the face and degeneracy maps over abstract simplexes using the binary representation.

As in the case of the Kenzo representation, we reuse the function `inv-gmsm` because it is related to the geometric simplexes of the simplicial sets and this part is encoded in the same way for all the representations. From this function, the invariant function for abstract simplexes using the binary representation is provided.

As the binary representation is an auxiliar artifact for our ACL2 proofs, we have directly defined the necessary functions to be admissible in ACL2.

Once we have presented the different representations, we can start our proofs.

6.2.2 Proving the correctness of Kenzo spheres

We started the work of proving that Kenzo initial simplicial sets are really simplicial sets by examining the construction of the sphere S^3. That is to say, we were interested in proving the correctness of the following Kenzo statement.

```
> (sphere 3) ✠
[K1 Simplicial-Set]
```

This means that we want to prove in ACL2 that the returned object by the `sphere` Kenzo function taking as input the natural number 3 satisfies the axioms of simplicial sets. Then, we proceed as follows.

First of all, we implement the face map over abstract simplexes of S^3. To this aim, we define in ACL2 the face function which is stored in the `face` slot of the `Simplicial-Set` instance associated with S^3. This function, that will be called `face-gmsm-S^3`, has three arguments: a face index, a simplex dimension and a geometric simplex; and computes any face of any geometric simplex of S^3.

From `face-gmsm-S^3` and the functions `1dlop-dgop-dgop` and `1dlop-dgop-indx` we define a function called `face-S^3` which encodes the face operator over abstract simplexes associated with S^3 using the Kenzo representation in ACL2, that is, the Kenzo `face` macro for the particular case of S^3. The function `face-S^3` takes as input three arguments: a face index i, a dimension q and an abstract simplex $absm$ of S^3 and computes $\partial_i^q absm$.

As we have said previously, the degeneracy operator functions are independent of the simplicial set. So, from the function `dgop*dgop` we have defined a function called `degeneracy` which encodes the degeneracy operator over abstract simplexes using the Kenzo representation in ACL2. The function `degeneracy` takes as input three arguments: a face index i, a simplex dimension q and an abstract simplex $absm$ and computes $\eta_i^q absm$. This function can be used for every simplicial set.

Subsequently, as we have seen in Definition 1.22, S^3 only has two geometric simplexes: a 0-simplex and a 3-simplex, encoded respectively by $\star$ and `s3`. Then, we can define the function `inv-gmsm-S^3`, which is used to determine the invariant function of S^3 according to the S^3 definition. This test function, that will be called `invariant-S^3`, has two arguments: a simplex dimension q and an abstract simplex $absm$.

Now, our goal consists in proving that the functions `face-S^3`, `degeneracy` and `invariant-S^3` satisfy the seven properties which define a simplicial set.

```
(defthm theorem-1-Kenzo-S^3 ;; $(x \in K^q \Rightarrow \partial_i^q x \in K^{q-1})$
  (implies (and (natp i) (natp q) (< 0 q) (<= i q) (invariant-S^3 q absm))
           (invariant-S^3 (1- q) (face-S^3 i q absm))))
```

```
(defthm theorem-2-Kenzo-S^3 ;; $(x \in K^q \Rightarrow \eta_i^q x \in K^{q+1})$
  (implies (and (natp i) (natp q) (<= i q) (invariant-S^3 q absm))
           (invariant-S^3 (1+ q) (degeneracy i q absm))))
```

```
(defthm theorem-3-Kenzo-S^3 ;; $(\partial_i^{q-1}\partial_j^q = \partial_{j-1}^{q-1}\partial_i^q$ if $i < j)$
  (implies (and (natp i) (natp j) (natp q) (< i j) (invariant q absm))
           (equal (face-S^3 i (1- q) (face-S^3 j q absm))
                  (face-S^3 (1- j) (1- q) (face-S^3 i q absm)))))
```

```
(defthm theorem-4-Kenzo-S^3 ;; $(\eta_i^{q+1}\eta_j^q = \eta_j^{q+1}\eta_{i-1}^q$ if $i > j)$
  (implies (and (natp i) (natp j) (natp q) (> i j) (invariant-S^3 q absm))
           (equal (degeneracy i (1+ q) (degeneracy j q absm))
                  (degeneracy j (1+ q) (degeneracy (1- i) q absm)))))
```

```
(defthm theorem-5-Kenzo-S^3 ;; $(\partial_i^{q+1}\eta_j^q = \eta_{j-1}^{q-1}\partial_i^q$ if $i < j)$
  (implies (and (natp i) (natp j) (natp q) (< i j) (invariant-S^3 q absm))
           (equal (face-S^3 i (1+ q) (degeneracy j q absm))
                  (degeneracy (1- j) (1- q) (face-S^3 i q absm)))))
```

```
(defthm theorem-6-Kenzo-S^3 ;; $(\partial_i^{q+1}\eta_i^q = identity = \partial_{i+1}^{q+1}\eta_i^q$ if $i < j)$
  (implies (and (natp i) (natp q) (invariant-S^3 q absm) )
           (and (equal (face-S^3 i (1+ q) (degeneracy i q absm))
                       absm)
                (equal (face-S^3 (1+ i) (1+ q) (degeneracy i q absm))
                       absm))))
```

```
(defthm theorem-7-Kenzo-S^3 ;; $(\partial_i^{q+1}\eta_j^q = \eta_j^{q-1}\partial_{i-1}^q$ if $i > j + 1)$
  (implies (and (natp i) (natp j) (natp q) (> i (1+ j)) (invariant-S^3 q absm))
           (equal (face-S^3 i (1+ q) (degeneracy j q absm))
                  (degeneracy j (1- q) (face-S^3 (1- i) q absm)))))
```

The verification of these seven theorems is tackled in ACL2 with the methodology presented in the following subsubsection which is imported from the work presented in [MMRRR09].

6.2.2.1 Sketch of the proof

First of all, we have defined the EAT counterparts of `face-S^3`, `degeneracy` and `invariant-S^3` that will be called `ADL-S^3`, `ANL` and `INV-S^3` respectively.

Subsequently, we prove the following seven theorems, that are the properties which the functions `ADL-S^3`, `ANL` and `INV-S^3` must fulfill to determine a simplicial set.

```
(defthm theorem-1-EAT-S^3 ;; $(x \in K^q \Rightarrow \partial_i^q x \in K^{q-1})$
  (implies (and (natp i) (natp q) (< 0 q) (<= i q) (INV-S^3 q absm))
           (INV-S^3 (1- q) (ADL-S^3 i q absm))))
```

```
(defthm theorem-2-EAT-S^3 ;; $(x \in K^q \Rightarrow \eta_i^q x \in K^{q+1})$
  (implies (and (natp i) (natp q) (<= i q) (INV-S^3 q absm))
           (INV-S^3 (1+ q) (ANL i q absm))))
```

```
(defthm theorem-3-EAT-S^3 ;; $(\partial_i^{q-1}\partial_j^q = \partial_{j-1}^{q-1}\partial_i^q$ if $i < j)$
  (implies (and (natp i) (natp j) (natp q) (< i j) (INV q absm))
           (equal (ADL-S^3 i (1- q) (ADL-S^3 j q absm))
                  (ADL-S^3 (1- j) (1- q) (ADL-S^3 i q absm)))))
```

```
(defthm theorem-4-EAT-S^3 ;; $(\eta_i^{q+1}\eta_j^q = \eta_j^{q+1}\eta_{i-1}^q$ if $i > j)$
  (implies (and (natp i) (natp j) (natp q) (> i j) (INV-S^3 q absm))
           (equal (ANL i (1+ q) (ANL j q absm))
                  (ANL j (1+ q) (ANL (1- i) q absm)))))
```

```
(defthm theorem-5-EAT-S^3 ;; $(\partial_i^{q+1}\eta_j^q = \eta_{j-1}^{q-1}\partial_i^q$ if $i < j)$
  (implies (and (natp i) (natp j) (natp q) (< i j) (INV-S^3 q absm))
           (equal (ADL-S^3 i (1+ q) (ANL j q absm))
                  (ANL (1- j) (1- q) (ADL-S^3 i q absm)))))
```

```
(defthm theorem-6-EAT-S^3 ;; $(\partial_i^{q+1}\eta_i^q = identity = \partial_{i+1}^{q+1}\eta_i^q$ if $i < j)$
  (implies (and (natp i) (natp q) (INV-S^3 q absm) )
           (and (equal (ADL-S^3 i (1+ q) (ANL i q absm))
                       absm)
                (equal (ADL-S^3 (1+ i) (1+ q) (ANL i q absm))
                       absm))))
```

```
(defthm theorem-7-EAT-S^3 ;; $(\partial_i^{q+1}\eta_j^q = \eta_j^{q-1}\partial_{i-1}^q$ if $i > j + 1)$
  (implies (and (natp i) (natp j) (natp q) (> i (1+ j)) (INV-S^3 q absm))
           (equal (ADL-S^3 i (1+ q) (ANL j q absm))
                  (ANL j (1- q) (ADL-S^3 (1- i) q absm)))))
```

For the verification development of these theorems, we followed a standard interaction with the ACL2 theorem prover. That is, we first had an original hand proof of the results that suggested the main definitions and lemmas. Some of these lemmas were not proved in a first attempt and new lemmas were then suggested from the inspection of the failed attempts.

Afterwards, we prove the equivalence between the EAT and Kenzo functions, which determine the face and degeneracy maps and the invariant function, modulo a domain transformation. Inspecting the definition of these functions, we realize that this task is, mainly, reduced to prove four equivalences module the change of representation: (1) the equivalence between the `cmp-d-ls-dgop` EAT function and the `1dlop-dgop-dgop` Kenzo function; (2) the equivalence between the `cmp-d-ls-indx` EAT function and the `1dlop-dgop-indx` Kenzo function; (3) the equivalence between the `cmp-ls-ls` EAT function and the `dgop*dgop` Kenzo function; and (4) the equivalence between the functions of the invariants.

In [MMRRR09] the equivalence between the `cmp-ls-ls` EAT function and the `dgop*dgop` Kenzo function was proved. Thus, we must focus on the other three equivalences. Since the process to prove the rest of the equivalences is very similar, let us focus on the first one.

The arguments and outputs of `cmp-d-ls-dgop` and `1dlop-dgop-dgop` are, of course, equivalent in both functions, but recall that in Kenzo a degeneracy operator is encoded as a natural number.

The proof that the function `1dlop-dgop-dgop` is equivalent to `cmp-d-ls-dgop` is not simple, for two main reasons. On the one hand, the Kenzo function and the EAT one deal with different representations of degeneracy operators. On the other hand, the Kenzo function implements an algorithm which is not intuitive and is quite different from the algorithm of the EAT version, which is closely related to the mathematical definitions. The suitable strategy, used in [MMRRR09] for the degeneracy operator, to attack the proof consists of considering the intermediary binary representation of degeneracy operators, explained in Subsubsection 6.2.1.4.

The plan has consisted in defining the function `1dlop-dgop-binary-dgop` implementing the application of the face operator over a degeneracy list represented as a binary list, by means of an algorithm directly inspired from that of Kenzo. Thus, the equivalence between EAT and Kenzo face maps has been proved in two steps. We have proved the equivalence between the Kenzo function `1dlop-dgop-dgop` and the binary function `1dlop-dgop-binary-dgop`. Subsequently, it has also been proved that the binary version function and the EAT function are equivalent.

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

$$\begin{tikzcd}
\mathbb{N} \times \mathcal{D}_g^L \arrow[r, bend left] \arrow[d, "cmp{-}d{-}ls{-}dgop"] & \mathbb{N} \times \mathcal{D}_g^B \arrow[l, bend left] \arrow[r, bend left] \arrow[d, "1dlop{-}dgop{-}binary{-}dgop"] & \mathbb{N} \times \mathbb{N} \arrow[l, bend left] \arrow[d, "1dlop{-}dgop{-}dgop"] \\
\mathcal{D}_g^L \arrow[r, bend left] & \mathcal{D}_g^B \arrow[l, bend left] \arrow[r, bend left] & \mathbb{N} \arrow[l, bend left]
\end{tikzcd}$$

These functions, that implement the face operators in the different representations, receive as input two arguments (a natural number representing the index of the face map and a degeneracy operator encoded in the respective representation) and have as output a degeneracy operator (in the same representation of the input one) which is obtained as explained in detail in the case of `cmp-d-ls` in Subsubsection 6.2.1.2.

Thus the commutativity of the diagram ensures the equivalence modulo the change of representation between the EAT and Kenzo models. We can proceed in the same way in the other equivalences.

These equivalence can be propagated to the functions which determine the face (`face-S^3`) and degeneracy (`degeneracy`) map and the invariant function (`invariant-S^3`) using the Kenzo representation.

Therefore, thanks to both the domain transformation theorems and the EAT theorems (`theorem-i-EAT-S^3` `i`$= 1, \ldots, 7$); we produce the proof of the theorems presented at the beginning of this subsection.

Then, we have proved that the functions which encode the face and degeneracy maps and the invariant function of the sphere S^3 in Kenzo really determine a simplicial set. Therefore, the Kenzo instruction `(sphere 3)` really constructs a simplicial set.

6.2.2.2 Kenzo spheres family

Inspecting the previous proof it became clear that most of the proof is independent of number 3. Then, the next step consisted in proving, using the same ideas, that the family of Kenzo spheres was really a family of simplicial sets; that is, $\forall n \in \mathbb{N}$ `(sphere n)` is a simplicial set. To undertake this task we have been inspired by the work [LPR03], where the algebraic specification for families of EAT objects was provided.

The process to prove the correctness of the Kenzo spheres family is very similar to the one presented for the sphere S^3. First of all, it is worth noting that when we go from working with a concrete object to deal with a families of objects indexed by an argument $\mathcal{I}$, we need to include a new parameter which represents the set of indexes to the functions which were used to determine the concrete object.

Taking into account the previous comment, we define the face operator over abstract simplexes of S^n. To this aim, we define in ACL2 the face function which is associated with the `sphere` Kenzo function. This function, that will be called `face-gmsm-S^n`, has four arguments: the dimension of the sphere n, a face index, a simplex dimension and a geometric simplex, and computes any face of any geometric simplex of S^n.

From `face-gmsm-S^n` and the functions `1dlop-dgop-dgop` and `1dlop-dgop-indx` we define a function called `face-S^n` which encodes the face operator over abstract simplexes associated with S^n using the Kenzo representation. The function `face-S^n` takes as input four arguments: the dimension of the sphere n, a face index i, a simplex dimension q and an abstract simplex *absm* of S^n and computes $\partial_i^q absm$.

We could use the `degeneracy` function defined previously, but for symmetry with the face operator we define the function `degeneracy-n` which takes as input four arguments: the dimension of the sphere n, a face index i, a simplex dimension q and an abstract simplex *absm* and computes $\eta_i^q absm$. The behavior is the same that the one of the function `degeneracy`.

Subsequently, as we have seen in Definition 1.22, S^n only has two geometric simplexes: a 0-simplex and an n-simplex, encoded respectively by $\star$ and `sn` (where `n` is the dimension of the sphere). Then, we can define the function `inv-gmsm-S^n`, which is used to determine the invariant function of S^n, according to the S^n definition. This test function, that will be called `invariant-S^n`, has three arguments: the dimension of the sphere n, a simplex dimension q and an abstract simplex *absm*.

Now, our goal consists in proving that the functions `face-S^n`, `degeneracy-n` and `invariant-S^n` satisfy the seven properties which define a simplicial set.

```
...
(defthm theorem-7-Kenzo-S^n ;; (∂_i^{q+1} η_j^q = η_j^{q-1} ∂_{i-1}^q if i > j + 1)
  (implies (and (natp n) (natp i) (natp j) (natp q) (> i (1+ j))
                (invariant-S^n n q absm))
           (equal (face-S^n n i (1+ q) (degeneracy-n n j q absm))
                  (degeneracy-n n j (1- q) (face-S^n n (1- i) q absm)))))
```

The methodology followed to prove those theorems was the same previously explained. The main idea is to provide the EAT counterparts of the Kenzo functions, afterwards prove the correctness of EAT programs, and then, by a domain transformation, translate the proofs to Kenzo programs using the intermediary binary model.

Then, we have proved that $\forall n \in \mathbb{N}$, the Kenzo function `(sphere n)` really constructs a simplicial set.

6.2.3 A Generic Simplicial Set Theory

One more time, the previous proof showed us clear guidelines to repeat the process for other families of Kenzo simplicial sets. We started that work by examining several families (standard simplicial sets, Moore spaces, Eilenberg MacLane spaces, and so on). We carefully analyzed what is the common part for all the families of initial simplicial sets, and what is particular for each one of them.

This led us to develop a generic theory in ACL2, in the terms presented in Subsection 6.1.2, to make the task of certifying families of Kenzo initial simplicial sets easier.

6.2.3.1 A generic model for families of Kenzo initial simplicial sets

As we have seen in Subsection 6.1.2, the first step to create our generic theory for families of Kenzo initial simplicial sets consists in defining a generic model of a family of Kenzo simplicial sets. To this aim, we use an encapsulate with the generic functions which define the generic Kenzo family and the axioms assumed about them.

Let $\{SS\}_{\mathcal{K}}$ be a family of Kenzo initial simplicial sets indexed by $\mathcal{K}$, then it is determined through three functions:

- `(imp-face-gmsm * * * *)`: this function takes as input four arguments, an element k of the index family $\mathcal{K}$, a face index i, a simplex dimension q and an geometric simplex *gmsm*; and computes $\partial_i^q gmsm$ in the element corresponding to k of the family $\{SS\}_{\mathcal{K}}$;
- `(imp-inv-gmsm * * *)`: this function takes as input three arguments, an element k of the index family $\mathcal{K}$, a simplex dimension q and a geometric simplex *gmsm*; and returns `t` if *gmsm* is an element in dimension q of the element corresponding to k of the family $\{SS\}_{\mathcal{K}}$ and `nil` otherwise; and,
- `(indexp *)`: this function takes as input an argument, an element k; and returns `t` if k is an element of the index family $\mathcal{K}$ and `nil` otherwise.

To finish our generic model of a family of Kenzo initial simplicial sets we have to assume two properties establishing the relations between the above functions.

```
(defthm faceoface (∂_i^{q-1} ∂_j^q gmsm = ∂_{j-1}^{q-1} ∂_i^q gmsm if i < j)
   (implies (and (natp i) (natp j) (natp q) (< i j) (<= 2 q)
                 (indexp k) (imp-inv-gmsm k q gmsm))
            (equal (imp-face-gmsm k (1- q) i (imp-face-gmsm k q j gmsm))
                   (imp-face-gmsm k (1- q) (1- j) (imp-face-gmsm k q i gmsm)))))
```

```
(defthm invariant-prop (gmsm ∈ K^q ⇒ ∂_i^q gmsm ∈ K^{q-1})
  (implies (and (natp i) (natp q) (< 0 q) (<= i q)
                (indexp k) (imp-inv-gmsm k q gmsm))
           (imp-inv-gmsm k (1- q) (imp-face-gmsm k q i gmsm))))
```

All the properties above are stated inside an ACL2 `encapsulate`. At this point we have a specification for families of Kenzo initial simplicial sets. A concrete implementation of a family of Kenzo simplicial sets will be given by concrete definitions of the generic functions, verifying the assumed properties.

From these generic functions and the assumed properties, we have developed a generic theory which allows us to prove the correctness of families of Kenzo simplicial sets. It is important to remark that the properties below are proved using only the assumed axioms.

The first thing we have to do is to define the face and degeneracy maps and the invariant function. Subsequently, we must prove the seven theorems which assert that those functions really determine a family of simplicial sets. All these definitions and theorems constitutes our generic theory about families of Kenzo initial simplicial sets.

6.2.3.2 Definitions of face, degeneracy and invariant

From the generic functions of the `encapsulate` presented in the previous subsubsection, we can define the face and degeneracy maps and the invariant function of a generic family of simplicial sets by means of the following three functions which invoke the ACL2 counterparts of the real Kenzo programs.

The face operator:

```
(defun imp-face (k i q absm)
  (let ((dgop (car absm))
        (gmsm (cadr absm)))
    (if (1dlop-dgop-indx i dgop)
        (list (1dlop-dgop-dgop i dgop) gmsm)
      (let ((absm2 (imp-face-gmsm k q (1dlop-dgop-indx i dgop) gmsm))
            (dgop2 (car absm2)) (gmsm2 (cadr absm2)))
        (list (dgop*dgop (1dlop-dgop-dgop i dgop) dgop2) gmsm2)))))
```

The degeneracy operator:

```
(defun imp-degeneracy (k i q absm)
  (declare (ignore k q))
   (let ((dgop (car absm))
         (gmsm (cadr absm)))
     (list (dgop*dgop (dgop-ext-int (list i)) dgop) gmsm)))
```

the function `dgop-ext-int` transforms a degeneracy list represented as a strictly decreasing list of natural numbers to its corresponding representation as a natural number.

And, the invariant function:

```
(defun imp-invariant (k q absm)
  (let ((dgop (car absm))
        (gmsm (cadr absm)))
    (and (<= (logcount dgop) q)
         (imp-inv-gmsm k (- q (logcount dgop)) gmsm)
         (< dgop (expt 2 q)))))
```

6.2.3.3 Proving the simplicial sets properties

In the previous subsubsection we have defined the functions which determine the face and degeneracy maps and the invariant function of a generic family of Kenzo initial simplicial sets. Now, we need to prove that these functions fulfill the properties which determine a family of simplicial sets.

Then, the well definition of both face and degeneracy maps must be verified:

```
(defthm theorem-1 ;; (x ∈ K^q ⇒ ∂_i^q x ∈ K^{q-1})
  (implies (and (natp i) (natp q) (< 0 q) (<= i q) (indexp k) (imp-invariant k q absm))
           (imp-invariant k (1- q) (imp-face k i q absm))))
```

```
(defthm theorem-2 ;; (x ∈ K^q ⇒ η_i^q x ∈ K^{q+1})
  (implies (and (natp i) (natp q) (<= i q) (indexp k) (imp-invariant k q absm))
           (imp-invariant k (1+ q) (imp-degeneracy k i q absm))))
```

Moreover, the 5 relations between the face and degeneracy maps stated in Definition 1.17 must be fulfilled. This task is performed in ACL2 with the verification of the following 5 theorems:

```
(defthm theorem-3 ;; (∂_i^{q-1} ∂_j^q = ∂_{j-1}^{q-1} ∂_i^q if i < j)
  (implies (and (natp i) (natp j) (natp q) (< i j) (indexp k)
                (imp-invariant k q absm))
           (equal (imp-face k i (1- q) (imp-face k j q absm))
                  (imp-face k (1- j) (1- q) (imp-face k i q absm)))))
```

```
(defthm theorem-4 ;; (η_i^{q+1} η_j^q = η_j^{q+1} η_{i-1}^q if i > j)
  (implies (and (natp i) (natp j) (natp q) (> i j) (indexp k)
                (imp-invariant k q absm))
           (equal (imp-degeneracy k i (1+ q) (imp-degeneracy k j q absm))
                  (imp-degeneracy k j (1+ q) (imp-degeneracy k (1- i) q absm)))))
```

```
(defthm theorem-5 ;; (∂_i^{q+1} η_j^q = η_{j-1}^{q-1} ∂_i^q if i < j)
  (implies (and (natp i) (natp j) (natp q) (< i j) (indexp k)
                (imp-invariant k q absm))
           (equal (imp-face k i (1+ q) (imp-degeneracy k j q absm))
                  (imp-degeneracy k (1- j) (1- q) (imp-face k i q absm)))))
```

```
(defthm theorem-6 ;; (∂_i^{q+1} η_i^q = identity = ∂_{i+1}^{q+1} η_i^q if i < j)
  (implies (and (natp i) (natp q) (indexp k) (imp-invariant k q absm))
           (and (equal (imp-face k i (1+ q) (imp-degeneracy k i q absm))
                       absm)
                (equal (imp-face k (1+ i) (1+ q) (imp-degeneracy k i q absm))
                       absm))))
```

```
(defthm theorem-7 ;; (∂_i^{q+1} η_j^q = η_j^{q-1} ∂_{i-1}^q if i > j + 1)
  (implies (and (natp i) (natp j) (natp q) (> i (1+ j)) (indexp k)
                (imp-invariant k q absm))
           (equal (imp-face k i (1+ q) (imp-degeneracy k j q absm))
                  (imp-degeneracy k j (1- q) (imp-face k (1- i) q absm)))))
```

The verification of these seven theorems is tackled in ACL2 with the methodology presented in Subsubsection 6.2.2.2. That is, we provide the EAT counterparts of the Kenzo functions, afterwards prove the correctness of EAT programs, and then, by a domain transformation, translate the proofs to Kenzo programs using the intermediary binary model.

All these properties are proved in the book `generic-simplicial-set.lisp`, which develops what we have named our "generic simplicial set theory", that is the sequence of definitions (`imp-face`, `imp-degeneracy` and `imp-invariant`) and theorems (`theorem-i` for `i=` $1,\dots,7$) which prove that families of Kenzo initial simplicial sets are really families of simplicial sets from generic assumptions about them. Moreover, this book also contains the generic instantiation tool presented in Subsection 6.1.2 to make the instantiation of their properties for a concrete family of Kenzo simplicial sets easier, as we will explain in the next section.

6.2.4 Applications of the Generic Simplicial Set Theory

The strength of the "generic simplicial set theory" relies on the few preconditions needed in order to prove that a family of Kenzo initial simplicial sets is really a family of simplicial sets. In addition, thanks to the generic instantiation tool presented in Subsection 6.1.2, when a user provides concrete definitions of the generic functions for a family of Kenzo initial simplicial sets, verifying the assumed properties, the generic instantiation tool automatically generates the complete proof.

Let us note the importance of the automatic generation of the proof by means of some data: from 3 definitions and 2 theorems the instantiation tool generates (and instantiates) 3 definitions and 7 theorems. In addition, the proof of the 7 theorems involves 92 definitions and 969 auxiliary lemmas. In this way, the hard task of proving that the objects of a Kenzo family are simplicial sets from scratch can be relaxed to introduce 3 definitions and prove 2 theorems (let us note that the proof of these 2 theorems can be not trivial at all).

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

6.2.4.1 Kenzo spheres family

We retake the proof related to the Kenzo spheres family presented in Subsubsection 6.2.2.2. Here, we are going to write an ACL2 book `ss-sphere.lisp` which generates a proof of the fact that the family of Kenzo spheres is really a family of simplicial sets, through the functions and properties of the generic theory. To this aim, we include the following: `(include-book "generic-simplicial-sets")`. This event generates `definstance-*simplicial-set-kenzo*`, a macro which will be used to instantiate the events from the generic book.

It is now needed to define the counterparts of the generic functions `imp-face-gmsm`, `imp-inv-gmsm` and `indexp` and prove the analogous theorems to `faceoface` and `invariant-prop` given in Subsubsection 6.2.3.1. This was done previously in Subsubsection 6.2.2.2 (we only need to define the counterpart of `indexp` which is `natp`, since the family of spheres is indexed by a natural number).

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

```
(definstance-*simplicial-set-kenzo*
 ((imp-face imp-face-S^n) (imp-inv-gmsm imp-inv-S^n) (indexp natp))
 "-sphere-family")
```

At this moment, new instantiated definitions and theorems are available in the ACL2 logical world, proving that Kenzo spheres satisfy all the conditions of Definition 1.17.

Thanks to the use of this tool, the proof effort of verifying that the family of Kenzo spheres is really a family of simplicial sets is considerably reduced.

6.2.4.2 Kenzo standard simplicial sets family

In this subsubsection, some comments about the verification that the standard simplicial set Kenzo family is really a family of simplicial sets are provided. The notion of standard simplicial set was introduced in Definition 1.18. Let us remark that the non degenerate simplexes of $\Delta[m]$ are any $(n+1)$-tuple $(a_0, \ldots, a_n)$ of integers such that $0 \leq a_0 < \cdots < a_n \leq m$.

As in the case of the spheres, our goal will consist of defining the concrete versions of the functions of the encapsulate of Subsubsection 6.2.3.1 and proving the properties stated in that encapsulate for the concrete case of the standard simplicial set family; and subsequently use the generic simplicial set theory.

Before giving the ACL2 definition of the face operator for standard simplicial sets, let us present the Kenzo code for that operation. The standard simplicial sets are so important in the Kenzo system that Kenzo programmers decided to use a very efficient representation for saving memory space to deal with geometric simplexes of this kind of simplicial sets. Namely, geometric simplexes of standard simplicial sets are encoded in a similar way to degeneracy operators (see Subsubsection 6.2.1.3).

An *increasing* sequence of non-negative integers describing a geometric simplex of a standard simplicial set, is coded on a binary integer with the following convention: a number i representing the vertex i is the $(i+1)$-th binary bit of a machine word. Let us explain with an example: consider the simplex `(0 1 3 5)` which can be seen as the binary list `(1 1 0 1 0 1)` in which `1` in position i if the number i is in the degeneracy list, `0` otherwise. This list, seen as a binary number in the reverse order, is the number 43. Thus, Kenzo encodes the geometric simplex `(0 1 3 5)` as 43.

The Kenzo implementation of the face operator ∂_i for simplexes of a standard simplicial sets consists in deactivating the i-th active bit. This is better understood if we think first in the binary representation. Let us consider the application of ∂_2 to `(0 1 3 5)`. Applying the definition of the standard simplicial set, we obtain `(0 1 5)`. Using binary notation, this means that ∂_2 of `(1 1 0 1 0 1)` is `(1 1 0 0 0 1)`.

As we have said before, Kenzo does not directly use the binary notation: it uses the natural number that this binary notation represents. Then, as in the case of degeneracy and face operators explained throughout this section, the face operator over geometric simplexes of standard simplicial sets is implemented by means of Common Lisp logical operators. The following is *the real* Common Lisp code of Kenzo for the face operator over simplexes of standard simplicial sets.

```
(DEFUN DELTA-FACE (indx dmns gmsm)
   (declare (fixnum indx gmsm) (ignore dmns))
      (do ((pmark 1 (ash pmark 1))
           (bmark indx)
           (gmsm2 gmsm (ash gmsm2 -1)))
          (nil)
        (declare (fixnum pmark bmark gmsm2))
         (when (oddp gmsm2)
            (when (minusp (decf bmark))
               (return-from delta-face (absm 0 (logxor gmsm pmark)))))))
```

This definition receives as input two natural numbers `indx` and `gmsm` and another argument that is ignored, `dmns`, and executes an infinite `do` that uses three local variables `pmark,` `bmark` and `gmsm2` storing respectively the list that will be used to defuse the `indx` active bit, the number of remaining active bits to reach the one to defuse and a number which will be used to indicate if `gmsm` has an active bit in a concrete position. When the `indx` active bit is reached, the function stops and returns the result. Otherwise, it updates the local variables and executes again the `do`.

We can provide an ACL2 definition of `delta-face` in a straightforward manner, based on the above Common Lisp code, but with some things defined in a different (but equivalent) way due to the applicative nature of ACL2. The only way of iterating in ACL2 is by means of recursive functions, then we use an auxiliary recursive definition implementing the internal `do`. Also, since destructive updates are not allowed in ACL2, we consider the local variables `pmark, bmark` and `gmsm2` as extra input parameters. Moreover, it is worth noting that the internal `do` of the `delta-face` Kenzo function can produce a non-terminating process; then, in order to get our ACL2 function admitted in the ACL2 logic we have explicitly introduced in its logical definition a condition that ensures its termination. This condition is implemented by means of the `good-input-p` function checking that `pmark`, `bmark`, `gmsm2` and `gmsm` are natural numbers; and that the `bmark` value must be lower than the number of active bits of `gmsm`, otherwise the `do` does not finish.

```
(defun good-input-p (pmark bmark gmsm2 gmsm)
   (and (natp bmark) (natp gmsm2) (natp pmark) (natp gmsm)
        (< bmark (logcount gmsm2))))
```

Having defined the function `good-input-p`, we may now define the `delta-face-do` function by means of the `defexec` macro, explained in Subsection 6.1.3, as follows:

```
(defexec delta-face-do (pmark bmark gmsm2 gmsm)
  (declare (xargs :guard (good-input-p pmark bmark gmsm2 gmsm)
  (mbe :logic (if (good-input-p pmark bmark gmsm2 gmsm)
                  (cond ((oddp gmsm2)
                         (cond ((minusp (1- bmark)) (logxor gmsm pmark))
                            (t (delta-face-do (ash pmark 1) (1- bmark)
                                     (ash gmsm2 -1) gmsm))))
                        (t (delta-face-do (ash pmark 1) bmark (ash gmsm2 -1) gmsm)))
                gmsm))
        :exec (cond ((oddp gmsm2)
                     (cond ((minusp (1- bmark)) (logxor gmsm pmark))
                        (t (delta-face-do (ash pmark 1) (1- bmark)
                                 (ash gmsm2 -1) gmsm))))
                    (t (delta-face-do (ash pmark 1) bmark (ash gmsm2 -1) gmsm))))
```

Note that the logical definition of the function includes the condition `(good-input-p pmark bmark gmsm2 gmsm)`, needed to be accepted by the ACL2 principle of definition. This test has to be evaluated in *every recursive call* of the function `delta-face-do`, making the logical definition slow for execution. The use of `mbe` guarantees that once the guards are verified, the `:exec` body, without the test, can be safely used for execution when the function is called on arguments satisfying its guard.

The guard of the function ensures that `pmark`, `bmark`, `gmsm2` and `gmsm` are natural numbers; and the `bmark` value must be lower than the number of active bits of `gmsm`.

The guard verification of the `delta-face-do` function involves some auxiliary lemmas. The call to `defexec` also generates proof obligations than ensure the termination of the `:exec` body on the domain specified by its guard.

Finally, the ACL2 definition of `delta-face` is a call to the above auxiliary function, with suitable initial values:

```
(defun delta-face (indx dmns gmsm)
    (declare (ignore dmns))
    (delta-face-do 1 indx gmsm gmsm))
```

We claim that the ACL2 version is faithful with the original Kenzo definition, since we have tried to keep it as similar as possible. As we have said, the fact that ACL2 is a subset of Common Lisp makes this translation almost direct. Besides the use of `defexec` to define `delta-face-do` increases the efficiency and reduces the gap between Kenzo and ACL2 definitions. Anyway, we strengthened our claim by an intensive testing. Since both definitions can be executed on any compliant Common Lisp, it was very easy to test that they return the same result for all pairs of inputs `indx` and `gmsm`, with `gmsm` $\leq$ 10000000 and `indx` $<$ `(logcount gmsm)`.

Finally, to state the correctness property of `delta-face` (`faceoface-delta` theorem) the same methodology explained for the case of both degeneracy and face operators has been used. That is to say, we prove the desired properties using the EAT model of the `delta-face` function (based on increasing natural lists and which is much simpler) and then translating it to the Kenzo model modulo the change of representation by means of an intermediate binary representation. Let us note that in this case 5 auxiliar definitions and 66 lemmas are necessary to verify the theorems included in the encapsulate of Subsubsection 6.2.3.1 for the standard simplicial sets functions.

Then, we can write an ACL2 book `ss-delta.lisp` which generates a proof of the fact that the standard simplicial set family objects with the Kenzo implementation are simplicial sets, through the functions and properties of the generic simplicial set theory, as was explained in the case of the spheres.

6.2.4.3 Kenzo simplicial sets associated with simplicial complexes family

This subsubsection ends the verification of programs presented in Section 5.1 about simplicial complexes. To verify that the Kenzo simplicial sets associated with simplicial complexes family is really a family of simplicial sets, we proceed in the same way presented in Subsubsection 6.2.4.1.

First of all, we include in our development the `generic-simplicial-sets` book.

Subsequently we define the counterparts of the generic functions `imp-face-gmsm`, `imp-inv-gmsm` and `indexp` from the Kenzo functions. For instance the version of `imp-inv-gmsm` is defined as follows:

```
(defun imp-inv-ss-sc (sc q gmsm)
   (and (equal (len gmsm) q) (member-equal gmsm sc)))
```

Afterwards, we prove the analogous theorems to `faceoface` and `invariant-prop` given in Subsubsection 6.2.3.1.

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

```
(definstance-*simplicial-set-kenzo*
 ((imp-face imp-face-ss-sc) (imp-inv-gmsm imp-inv-ss-sc) (indexp simplicialcomplexp))
 "-ss-sc-family")
```

At this moment, new instantiated definitions and theorems are available in the ACL2 logical world, proving that our Kenzo implementation of the family of simplicial sets associated with simplicial complexes is really a family of simplicial sets.

To summarize this section, a methodology to verify that Kenzo initial simplicial sets are really simplicial sets has been presented. In addition, we have developed a generic simplicial set theory which allows us to prove that families of Kenzo simplicial sets are really families of simplicial sets. We believe that similar developments can be done for other mathematical Kenzo structures (such as chain complexes, abelian simplicial groups and so on) and the functions which construct initial spaces of that structures.

6.3 Modeling algebraic structures in ACL2

Algebraic structures are important when we are trying to define a mathematical setting. The implementation of algebraic structures in theorem proving environments is a well-known problem; and most of the theorem prover systems offer an implementation of modules that can be used to this aim. Inspired by the work presented in the previous section and with the aim of making easier the task of modeling algebraic structures, we have developed a methodology to deal with algebraic structures in ACL2.

6.3.1 Modeling simple algebraic structures in ACL2

Most often, an object of some *type* in Mathematics is a structure with several components, frequently of functional nature. In ACL2 the implementation of algebraic structures can be done by means of record structures; in addition, one must specify the rules satisfied by the algebraic structure, too. Records can be implemented in ACL2 thanks to the `defstructure` macro [Bro97].

We start by introducing the implementation of a very basic algebraic structure in

ACL2. A magma is a set with a binary operation over it. The ACL2 implementation of this structure is as follows.

```
(defstructure magma
    inv
    binary-op)
```

As it can be seen, a record, called `magma`, with two fields representing the invariant function of the magma (that is to say, the characteristic function of the underlying set), and the binary operation over it is defined.

When we define a record with the `defstructure` macro, several functions are automatically defined in ACL2: (1) a function called `make-<record>` (where `<record>` is the name of the record) which allows one to define concrete instances of the record; and (2) a function per each slot of the record called `<record>-<slot>` (where `<record>` is the name of the record and `<slot>` is the name of the slot) which is the *reader* function allowing the access to the `<slot>` value of `<record>` instances.

Now, if we want to construct a concrete magma instance (for instance, the magma with just the 0 element) we must proceed as follows. First of all, we define the invariant function and the binary operation of the magma.

```
(defun inv-unit (a) (equal a 0))
(defun op-unit (a b) (declare (ignore a b)) 0)
```

Subsequently, we use those functions to create an instance of the magma record by means of the `make-magma` instruction, where the value of the `inv` and `binary-op` slots are the function symbols of the previously defined functions:

```
(make-magma :inv 'inv-unit :binary-op 'op-unit)
```

In this way we can define the magma structure and provide concrete instances of it.

However, our specification of magmas is not complete, since the rules satisfied by this algebraic structure must be specified, too. To this aim, we have defined a function called `magma-conditions`. This function takes as argument a `magma` instance and generates a list of the rules that the functions of the `magma` instance must satisfy to define a magma structure. Using this function with the previous example, we obtain the following result.

```
> (magma-conditions (make-magma :inv 'inv-unit :binary-op 'op-unit)) ✠
(IMPLIES (AND (INV-UNIT X) (INV-UNIT Y))
         (INV-UNIT (OP-UNIT X Y)))
```

The generated list specifies the closure axiom of magmas for the concrete functions of our `magma` instance. It is worth noting that the `magma-conditions` function does not prove that the functions of a `magma` instance define a magma, but it just generates the list of rules that those functions must satisfy to determine an object which belongs to that structure. Thus, if we want to prove that the functions of a `magma` instance define a magma; we must state a theorem whose body is the output of the `magma-conditions` function.

```
(defthm unit-is-a-magma
    (implies (and (inv-unit x) (inv-unit y))
             (inv-unit (op-unit x y))))
```

When the above theorem is evaluated in ACL2, a proof attempt is started trying to prove the rules generated by `magma-conditions` for the concrete magma instance. In this case, the attempt is successful and the `unit-is-a-magma` theorem is introduced in the ACL2 logic stating the magma properties for our implementation of the magma with just the 0 element.

To make easier the definition of theorems like `unit-is-a-magma`, a macro called `check-magma-p` has been defined. This macro takes a `magma` instance and a symbol `s` as arguments and expands into a call of `defthm`. The name of the `defthm` is `s-is-a-magma`. The term generated for the `defthm` event states that the functions of the `magma` instance determine a magma; to this aim, the `magma-conditions` function is used. For instance, in our example, we proceed as follows:

```
> (check-magma-p (make-magma :inv 'inv-unit :binary-op 'op-unit) 'unit2) ✠
...
Q.E.D
unit2-is-a-magma
```

which produces the same theorem as before but with other name.

```
> :pe unit2-is-a-magma ✠
(DEFTHM UNIT2-IS-A-MAGMA
  (IMPLIES (AND (INV-UNIT X) (INV-UNIT Y))
           (INV-UNIT (OP-UNIT X Y))))
```

In this way, we can fully define the magma structure in ACL2.

Let us summarize our methodology to define mathematical structures in ACL2. Let us suppose that we want to define a mathematical structure called A with components $a1, \ldots, an$. First of all, we define a record with so many slots as components has the structure:

```
(defstructure A
    a1 ... an)
```

Afterwards, we define a function called `A-conditions` which has as argument an `A` instance. This function generates the list of rules that the functions `a1`,..., `an` of the `A` instance must satisfy to define an A structure.

```
(defun A-conditions (A)
    (A-rules A))
```

Let us remark that the `A-conditions` function does not prove that the functions of an `A` instance define an A structure, but it generates the list of rules that those functions must satisfy to determine an object which belongs to this mathematical structure. Thus, if we want to prove that the functions of an `A` instance define an A structure; we must state a theorem whose body is the output of the `A-conditions` function. To that aim, a macro called `check-A-p` is defined. This macro takes an `A` instance and a symbol `s` as arguments and expands into a call of `defthm`. The name of the `defthm` is `s-is-an-A`. The term generated for the `defthm` event states that the functions of the `A` instance determine an A structure; to this aim, the `A-conditions` function is used. It is worth noting that the verification of some of the rules may require some auxiliary lemma.

The definition of macros like `check-A-p` was inspired by the definition of the ACL2 macro `defequiv`, see [KM].

In this way, algebraic structures can be defined in ACL2.

6.3.2 Modeling algebraic structures in terms of others in ACL2

Let us note that most often algebraic structures are defined in terms of others; for instance, a semigroup *is* a magma where the binary operation satisfies the associativity property. Thus, the semigroup structure should be defined in terms of the magma one.

Let us present the definition of the semigroup structure. First of all, we define a `semigroup` record which has two functional slots representing the invariant function and the binary operator.

```
(defstructure semigroup
    inv
    binary-op)
```

Afterwards, we define a function called `semigroup-conditions` in charge of generating the list of rules which must satisfy the functions of a `semigroup` instance to define a semigroup. The fact that a semigroup is a magma is taken into account in the definition of `semigroup-conditions` which invokes `magma-conditions`. In addition, the associativity property about the binary operation is included, too.

```
(defun semigroup-conditions (semigroup)
   '(and
     (magma-conditions (make-magma :inv (semigroup-inv semigroup)
                                   :binary-op (semigroup-binary-op semigroup)))
     (implies (and ((semigroup-inv semigroup) x)
                   ((semigroup-inv semigroup) y)
                   ((semigroup-inv semigroup) z))
              (equal ((semigroup-binary-op semigroup)
                        ((semigroup-binary-op semigroup) x y) z)
                     ((semigroup-binary-op semigroup) x
                        ((semigroup-binary-op semigroup) y z)))
```

As in the case of magmas, we have defined a macro, called `check-semigroup-p`, which invokes the function `semigroup-conditions` in order to check that the functions of a `semigroup` instance define a semigroup. Let us present an example of its usage, for instance, the functions considered in the magma case also define a semigroup.

```
> (check-semigroup-p (make-semigroup :inv 'inv-unit :binary-op 'op-unit) 'unit) ✠
...
Q.E.D
unit-is-a-semigroup
```

```
> :pe unit-is-a-semigroup ✠
(DEFTHM UNIT-IS-A-SEMIGROUP
  (AND (IMPLIES (AND (INV-UNIT X) (INV-UNIT Y))
                (INV-UNIT (OP-UNIT X Y)))
       (IMPLIES (AND (INV-UNIT X) (INV-UNIT Y) (INV-UNIT Z))
                (EQUAL (OP-UNIT (OP-UNIT X Y) Z) (OP-UNIT X (OP-UNIT Y Z))))))
```

Then, we have defined the semigroup structure in terms of the magma one.

Let us summarize our methodology to define mathematical structures in terms of others in ACL2. Let us suppose that we want to define a mathematical structure called B with components $b1, \dots, bm$ in terms of other mathematical structures $A1, \dots, An$. First of all, we define a record with so many slots as components has the B structure:

```
(defstructure B
    b1 ... bm)
```

Afterwards, we define a function called `B-conditions` which has as argument a `B` instance. This function generates the list of rules that the functions of the `B` instance must satisfy to define a B structure; as B is defined in terms of $A1, \dots, An$; then, the `B-conditions` function invokes the `A1-conditions`, ..., `An-conditions` ones.

```
(defun B-conditions (B)
    '(and (A1-conditions (make-A1 ...))
          ...
          (An-conditions (make-An ...))
          (B-concrete-rules B)))
```

Subsequently, a macro called `check-B-p`, which invokes the `B-conditions` function and checks that the functions of a `B` instance define a *B* structure, is defined. This macro has an analogous behavior to the one explained previously for the other macros.

So, we have presented a way of modeling algebraic structures which are defined in terms of others in ACL2.

6.3.3 Modeling operations between algebraic structures in ACL2

Up to now, we have presented how to model algebraic structures in ACL2; now we are going to tackle the task of representing operations over those mathematical structures.

Let us consider the definition of a homomorphism between magmas. First of all, we define a `magma-homomorphism` record which has three slots which represent the source magma, the target magma and the map between them.

```
(defstructure magma-homomorphism
    magma-source
    magma-target
    map)
```

It is worth noting that in this case, the values of both `magma-source` and `magma-target` fields of a `magma-homomorphism` instance will be `magma` instances instead of function symbols as in the rest of the cases. For instance, we define the trivial map between the magma with just the element 0 and the magma with just the element 1:

```
(defun trivial (x) (declare (ignore x)) 1)
```

then, we define the trivial magma homomorphism between the magma with just the element 0 and the magma with just the element 1 as follows:

```
(make-magma-homomorphism
        :magma-source (make-magma :inv 'inv-unit-0 :binary-op 'op-unit-0)
        :magma-target (make-magma :inv 'inv-unit-1 :binary-op 'op-unit-1)
        :map 'trivial)
```

where the `inv-unit-0` and `op-unit-0` functions are respectively the `inv-unit` and `op-unit` functions that have been defined in Subsection 6.3.1, and `inv-unit-1` and `op-unit-1` are analogous to the `inv-unit` and `op-unit` functions but with 1 as element.

Afterwards, we define a function called `magma-homomorphism-conditions` in charge of generating the list of rules which must satisfy the functions of a `magma-homomorphism` instance to define a magma homomorphism. A magma homomorphism involves two magmas; then the `magma-homomorphism-conditions` function invokes the `magma-conditions` one, with both `magma-source` and `magma-target` values of the `magma-homomorphism` instance; and includes the properties about homomorphisms between magmas.

```
(defun magma-homomorphism-conditions (magma-homomorphism)
  '(and
     (magma-conditions (magma-source magma-homomorphism))
     (magma-conditions (magma-target magma-homomorphism))
     (implies ((magma-inv (magma-source magma-homomorphism)) x)
              ((magma-inv (magma-target magma-homomorphism))
                  ((map magma-homomorphism) x)))
     (implies (and ((magma-inv (magma-source magma-homomorphism)) x)
                   ((magma-inv (magma-source magma-homomorphism)) y))
              (equal ((map magma-homomorphism)
                        ((magma-binary-op (magma-source magma-homomorphism)) x y))
                     ((magma-binary-op (magma-target magma-homomorphism))
                        ((map magma-homomorphism) x) ((map magma-homomorphism) y)
                        ))))))
```

Subsequently, as in the previous cases, we have defined a macro called `check-magma-homomorphism-p` with the expected behavior. For instance, in the case of the trivial magma homomorphism between the magma with just the 0 element and the magma with just the element 1, we use the following command.

```
> (check-magma-homomorphism-p
   (make-magma-homomorphism
        :magma-source (make-magma :inv 'inv-unit-0 :binary-op 'op-unit-0)
        :magma-target (make-magma :inv 'inv-unit-1 :binary-op 'op-unit-1)
        :map 'trivial) 'trivial) ✠
...
Q.E.D
trivial-is-a-magma-homomorphism
```

```
> :pe trivial-is-a-magma-homomorphism ✠
(DEFTHM TRIVIAL-IS-A-MAGMA-HOMOMORPHISM
  (AND (IMPLIES (AND (INV-UNIT-0 X) (INV-UNIT-0 Y))
                (INV-UNIT-0 (OP-UNIT-0 X Y)))
       (IMPLIES (AND (INV-UNIT-1 X) (INV-UNIT-1 Y))
                (INV-UNIT-1 (OP-UNIT-1 X Y)))
       (IMPLIES (INV-UNIT-0 X)
                (INV-UNIT-1 (TRIVIAL X)))
       (IMPLIES (AND (INV-UNIT-0 X) (INV-UNIT-0 Y))
                (EQUAL (TRIVIAL (OP-UNIT-0 X Y))
                       (OP-UNIT-1 (TRIVIAL X) (TRIVIAL Y))))))
```

Then, we have presented a way of defining and verifying homomorphisms between magmas in ACL2.

Let us summarize our methodology to define operations between mathematical structures in ACL2. Let us suppose that we want to define an operation, called *op*, between the mathematical structures A and B. First of all, we define a record with three slots: (1) the source structure, (2) the target structure and (3) the map between the structures.

```
(defstructure op
     source target map)
```

For an `op` instance, the value of its `source` slot is an instance of the `A` structure, the value of its `target` slot is an instance of the `B` structure and the value of its `map` slot is a function symbol.

Afterwards, we define a function called `op-conditions` which has as argument an `op` instance. This function generates the list of rules that the fields of the `op` instance must satisfy to determine an *op* operation; namely, this function generates the list of properties that the source must satisfy to be an A structure, the list of properties that the target must satisfy to be a B structure, and the concrete properties associated with the operation *op*; then, the `op-conditions` function invokes both `A-conditions` and `B-conditions` functions with the values of the `source` and `target` slots of the `op` instance respectively.

```
(defun op-conditions (op)
    '(and (A-conditions (source op))
          (B-conditions (target op))
          (op-concrete-rules op)))
```

Subsequently, a macro called `check-op-p` using the `op-conditions` function is defined to state that the functions of the `op` instance determine an *op* operation.

Therefore, we can define in ACL2 operations between mathematical structures.

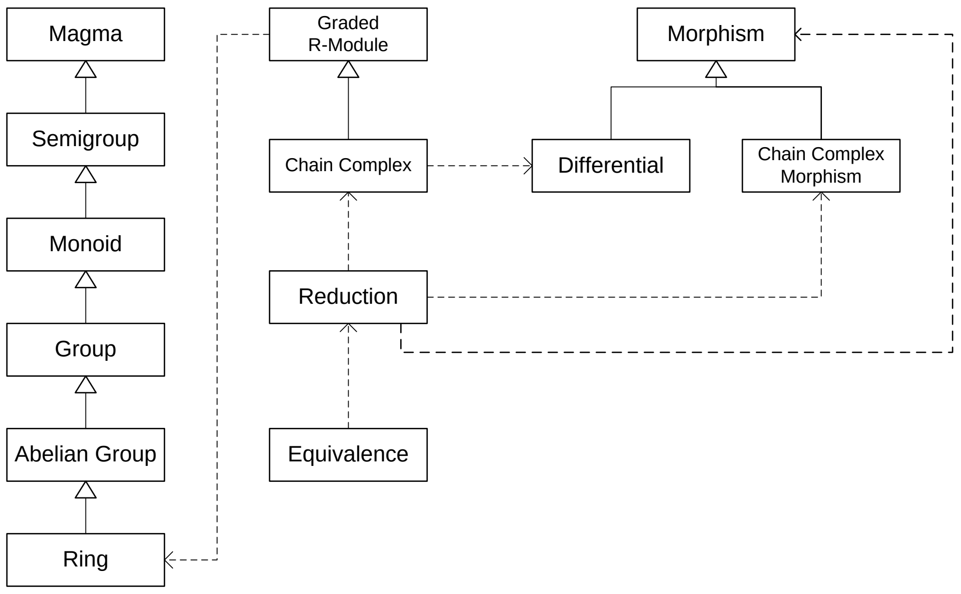


Figure 6.2: Algebraic structure hierarchy

6.3.4 An algebraic hierarchy in ACL2

Algebraic hierarchies are, in some cases, the foundation for large proof developments. Several algebraic hierarchies have been proposed for the Coq system: the CCorn hierarchy [GPWZ02] based on dependent records and used in the proof of the fundamental theorem of algebra, the SSReflect hierarchy [GGMR09] based on packed classes and currently used in the development of the proof of the Feit-Thompson Theorem, and also an approach based on the Coq's type class mechanism [SvdW10]; other examples can be found in systems such as Nuprl [Jac95] and Lego [Bai99]. Nevertheless, the development of an ACL2 algebraic hierarchy had not been undertaken until now up to our knowledge. The main difference between the cited approaches and the presented in this paper comes from the nature of the systems since all the quoted hierarchies are written in higher order Theorem Provers based on type theory; on the contrary our hierarchy is written in the untyped first order logic ACL2 Theorem Prover.

Following the guidelines provided in the previous subsections, we have defined the structures: magma, semigroup, monoid, group, abelian group and ring. Moreover, some of the main mathematical structures used in Algebraic Topology, such as morphisms, chain complexes, reductions and so on; are also introduced. In Figure 6.2 we have depicted the current status of our hierarchy and the relations among the structures. A continue arrow entails that the source structure is defined in terms of the target structure (see Subsection 6.3.2), and a dashed arrow means a use relationship (see Subsection 6.3.3).

6.3.5 Generic theories about algebraic structures in ACL2

Up to now, we have presented how to model algebraic structures in ACL2. However, we are also interested in reasoning with them in a generic way; that is to say, allowing the user to state general properties about the structures and, subsequently, instantiate those properties with concrete instances of the structures. Let us note, that this goal involves the introduction of higher order results in ACL2.

A typical result involving mathematical structures has the following form: "Let x be a mathematical structure A, then x satisfies the property P". To formalize this kind of results, we use a well known ACL2 technique to simulate higher order logic explained in Subsection 6.1.2: the combination of *encapsulates* and *functional instantiation*. To be more concrete, we proceed as follows.

First of all, we define the mathematical structure A with components $a1$,..., an as was explained in the previous subsections. Afterwards, we use an encapsulate, where we assume the existence of functions `a1`, ..., `an` which determine an A structure, to check this fact we use the `check-A-p` macro. Eventually, once that the encapsulate has been admitted, we prove the formula `P` which describes the property P. This higher order result ensures that any instance `x` of the `A` structure fulfills the property `P`.

Let us consider, the definition of a *generic theory* about groups. As we have said in Subsection 6.3.4 we have defined the `group` structure following the guidelines presented in the previous subsections. Then, we can use the following encapsulate to define a *generic* group.

```
(encapsulate
   ; the signatures
   (((invariant *) => *)
    ((prod * *)    => *)
    ((inverse *)   => *)
    ((nullel )     => *))
   ; witnesses
   ...
   ; the axioms
   (check-group-p (make-group :inv 'invariant :binary-op 'prod
                              :inverse 'inverse :nullel 'nullel) 'group-encapsulate))
```

The first expression of the encapsulate is a list with the arity description of the functions which are introduced with the encapsulate principle. In this case, we indicate the arities of the four functions which describe a group: the `invariant` unary operation (the characteristic function of the underlying set), the `prod` binary operation (the "product"), the `inverse` unary operation (the "inverse") and the `nullel` constant (the "identity element").

Since we are defining `invariant`, `prod`, `inverse` and `nullel` functions in this way, the functions defined from them are not executable. However, there exists a great advantage from the verification point of view; due to the fact that we are formalizing and verifying a generic result which can be instantiated for concrete cases later on.

Finally, the properties assumed over the encapsulate functions (that is to say, the group properties) are stated by means of the `check-group-p`. It is worth noting the profit of using this approach to define the properties assumed over the encapsulate functions; since otherwise we should introduce manually each one of the group axioms. In this way, our definition of a generic group is more compact and less likely to forget some group axiom.

After admitting the encapsulate in the ACL2 logic, we want to prove some properties about groups. For instance, let $(G, \cdot, ^{-1}, 1)$ be a group (where $\cdot$ denotes the binary operation, the exponent -1 denotes the inverse operation and 1 the null element) then $g1, g2 \in G \Rightarrow (g1 \cdot (g1^{-1} \cdot g2)) \cdot g2^{-1} = 1$. This result can be proved in ACL2 with the generic functions `invariant`, `prod`, `inverse` and `nullel`, thanks to the group axioms stated in the encapsulate about them:

```
(defthm property
   (implies (and (invariant g1)
                 (invariant g2))
            (equal (prod (prod g1 (prod (inverse g1) g2)) (inverse g2))
                   (nullel))))
```

Now, we are interesting in proving the same property but for a concrete group instance. For instance, let us consider the implementation of the cyclic group of dimension two, which can be defined in ACL2 as follows.

```
(defun invariant-binary (g) (or (equal g 0) (equal g 1)))
(defun prod-binary (g1 g2) (mod (+ g1 g2) 2))
(defun inverse-binary (g) g)
(defun nullel-binary () 0)
(make-group :inv 'invariant-binary :binary-op 'prod-binary
            :inverse 'inverse-binary :nullel 'nullel-binary)
```

Now instead of directly proving the `property` theorem for this concrete group, since the above functions determine a group,

```
> (check-group-p (make-group :inv 'invariant-binary :binary-op 'prod-binary
                             :inverse 'inverse-binary :nullel 'nullel-binary)
                'cyclic2) ✠
...
Q.E.D
cyclic2-is-a-group
```

we can instantiate the looked for result, with the `functional-instance` hint.

```
(defthm property-binary
    (implies (and (invariant-binary a)
                  (invariant-binary b))
             (equal (prod-binary (prod-binary a (prod-binary  (inverse-binary a) b))
                                 (inverse-binary b))
                    (nullel-binary)))
    :hints (("Goal" :by (:functional-instance property
                          (invariant invariant-binary)
                          (prod prod-binary)
                          (inverse inverse-binary)
                          (nullel nullel-binary)))))
```

This means that once we have proved a result for the generic group, we do not need to prove over and over again the same result for concrete group instances, a task that in some cases can be not trivial, but we just instantiate the general result.

In this way, higher order results related to algebraic structures are obtained in ACL2. It is worth noting that we can use this procedure to packing the "Generic Simplicial Set Theory" presented in Subsection 6.2.3 making that development more readable. Moreover, we can use our mathematical structures modeling to provide an ACL2 infrastructure to formalize Kenzo higher order constructors as we will see in the following section; but, first, let us present some differences between Kenzo and ACL2 mathematical structures that deserve to be explained.

6.3.6 Distance from ACL2 mathematical structures to Kenzo mathematical structures

We are interesting in using our ACL2 mathematical structures to verify properties about Kenzo mathematical structures. However, there are some differences between them, so, let us briefly present the differences between ACL2 and Kenzo encodings for mathematical structures. Mathematical structures are implemented by means of classes in the Kenzo system, see Subsection 1.2.1, and records in ACL2. In both cases the idea is the same, mathematical structures are encoded by means of functions that represent their operators. However, in most of the cases the number of slots of a Kenzo class and the slots of the correspondent ACL2 record are different. This gap comes from the goal pursued by Kenzo, devoted to compute homology groups, and the goal of ACL2, related to verify the correctness of the programs.

On the one hand, some key slots to perform computations are irrelevant to develop a proof, hence they are removed from the definition of the ACL2 record; for instance, the definition of morphisms in Kenzo includes a slot used to implement a *memoization* strategy which is not included in the ACL2 definition.

On the other hand, ACL2 records must include all the necessary slots of the mathematical structure to be able to prove, but some of them are unnecessary to compute; so, they are not included in Kenzo. For instance, the ACL2 chain complex record includes a slot that represents the invariant function of the structure, which is instrumental to prove the correctness of our programs; however, as we explained in Paragraph 6.2.1.2.1, due to the fact that Kenzo is a computational tool, the invariant functions were removed from the Kenzo programs.

Then, we have a separation of concerns where Kenzo mathematical structures are used for computing and its ACL2 version (which removes the computing slots of the Kenzo version and includes new slots which are necessary for deduction purposes) can be used for reasoning.

6.4 An ACL2 infrastructure to formalize Kenzo Higher Order constructors

In Section 6.2 an ACL2 infrastructure allowing us to prove the correctness of Kenzo programs for constructing initial simplicial sets has been explained. This can be interpreted as first step of a (simplified) Kenzo user interaction and can be modeled in a first order logic. However, the construction of Kenzo spaces from other ones involves higher order functional programming; thus, at least at first glance, higher order logic tools (such as Isabelle or Coq) seem more suitable than ACL2 (a first order tool) to this task. Nevertheless, as we commented at the beginning of this chapter, we can simulate higher order logic results in ACL2, obtaining code much more closer to actual Kenzo code than the one extracted from the proofs of the higher order logic tools.

This section is devoted to explain an ACL2 infrastructure, whose foundation is the method for modeling mathematical structures explained in the previous section, allowing us to prove the correctness of topological spaces constructed by the application of topological constructors. We simulate higher order by applying a combination of the *encapsulate* tool and the derived rule of inference named *functional instantiation*, this is a well-known procedure to cope with this kind of situations in ACL2, see [BGKM91]. The feasibility of using our framework is illustrated with several examples.

The rest of this section is organized as follows. Subsection 6.4.1 is devoted to present our methodology to formalize in ACL2 the constructors of spaces from other ones; first by using the example of the direct sum of chain complexes (Subsubsection 6.4.1.1) and subsequently explaining the general case (Subsubsection 6.4.1.2). The methodology is evaluated with several case studies in Subsection 6.4.2. The gap between the verified ACL2 code and the actual Kenzo code is described in Subsection 6.4.3.

6.4.1 Formalization of the Kenzo constructors in ACL2

This section is devoted to present our methodology to formalize the Kenzo constructors of spaces from other ones in ACL2. In order to provide a better understanding of our methodology the basic example of the direct sum of chain complexes is presented. Subsequently, the general case is explained.

6.4.1.1 The correctness of the Kenzo direct sum constructor

The notion of direct sum of two chain complexes was introduced in Definition 1.8. The Kenzo system implements the direct sum of chain complexes, then, we want to verify that its implementation acts as it is supposed to. This is translated to the ACL2 context as follows.

A chain complex is an algebraic structure determined by means of their operations and whose properties are given in an axiomatic way. In our framework chain complexes are encoded in the way explained in Section 6.3; that is, we use a record with 12 functional slots that correspond with the operations of the chain complex structure. This record is called `chain-complex` and can be instantiated by means of the `make-chain-complex` functions. In addition, we have defined the macro `check-chain-complex-p` which verifies that chain complex record instances satisfies the axiomatical properties of chain complexes. This macro takes a `chain-complex` instance and a symbol `s` as arguments and expands into a call of `defthm`. The name of the `defthm` is `s-is-a-chain-complex`. The term generated for the `defthm` event states that the functions of the `chain-complex` instance determine a chain complex.

Then, we can define concrete chain complex instances and verify that they really are chain complexes, but we want to prove that the direct sum Kenzo implementation is correct for every pair of chain complexes. Then, we follow the approach presented in Subsection 6.3.5 where a way of developing generic theories about mathematical structures was provided by means of the combination of encapsulates and the functional instantiation rule. Thus, a representation of two generic chain complex instances is given in ACL2 by means of the following encapsulate.

```
(encapsulate
 ; Signatures
 ((inv-A  * *) => *)
 ((diff-A * *) => *)
 ...
 ((inv-B  * *) => *)
 ((diff-B * *) => *)
 ...
 ; Witnesses
 ...
 ; Properties
 (check-chain-complex-p (make-chain-complex :inv-cc 'inv-A ... :diff-cc 'diff-A) 'A)

 (check-chain-complex-p (make-chain-complex :inv-cc 'inv-B ... :diff-cc 'diff-B) 'B))
```

The above encapsulate should be read as follows. First of all, the signatures of the functions which determine two generic chain complexes are provided. In this case, only the signatures of two of the functions which determine the chain complex structure, namely `inv` (the characteristic function of the underlying graded set of generators of the chain complex) and `diff` (the differential of the chain complex) are shown. The suffixes `-A` and `-B` indicate that the functions correspond with the chain complexes A and B respectively.

Subsequently, a witness for each one of the functions is given in order to avoid the introduction of inconsistencies in ACL2. The ellipsis in this part of the encapsulate means that we omit the local definitions (since they are irrelevant).

Finally, we gather the functions in two instances of the `chain-complex` record, constructed with `make-chain-complex`. Besides, we assume the properties of chain complexes for the functions which are declared in the encapsulate by means of the `check-chain-complex-p` macro.

The effect of the above encapsulate is an extension of the ACL2 universe in which the functions that determine two generic chain complexes have been defined. Now, our intention is to develop the direct sum constructor, based on the Kenzo definition, and prove that such constructor corresponds with the mathematical definition of the direct sum of two chain complexes given in Definition 1.8.

To this aim, we have been inspired by the notion of *functor* (also called parametric module) implemented in Coq [Chr03]. A similar notion can be found in Paulson's book on ML [Pau96]. In these works, the notion of functor builds a new module of a determined type from some given modules, in other words the new module is a parametric module. We translate this idea to our case, where the initial modules are the two generic chain complex instances declared in the encapsulate, the functor is the set of function definitions coming from the Kenzo system for the direct sum (that will be called *functor functions*) which will be defined from the encapsulate functions:

```
(defun inv-ds (n x)
   (and (inv-A n (car x))
        (inv-B n (cadr x))
        (equal (list (car x) (cadr x)) x)))
...
(defun diff-ds (n a)
   (list (diff-A n (car a)) (diff-B n (cadr a))))
...
```

and the new module is a new chain complex instance gathering these functions.

```
(make-chain-complex :inv-cc 'inv-ds ... :diff-cc 'diff-ds)
```

Then, we have defined the functions which determine the direct sum constructor in ACL2. Now we want to prove that this constructor really acts as the direct sum of two chain complexes. To this purpose, we need to verify that the direct sum constructor is a chain complex which satisfies the rules imposed on Definition 1.8.

The former task, the verification that the ACL2 direct sum constructor is a chain complex, is carried out with the `check-chain-complex-p` macro.

```
(check-chain-complex-p (make-chain-complex :inv-cc 'inv-ds ... :diff-cc 'diff-ds)
                       'direct-sum-a-b)
```

In this case, the proof fails in the first attempt, but we only need the introduction of six auxiliary lemmas to succeed, and then the theorem stating the properties of chain complexes for the direct sum is admitted in ACL2; this theorem is called `DIRECT-SUM-A-B-is-a-chain-complex`.

The latter task, that is to say, prove that our definition of the direct sum constructor satisfies the rules imposed on Definition 1.8 (namely, $M_n = (C_n, D_n)$ and $d_n((x, y)) = (d_{C_n}(x), d_{D_n}(y))$), can be consider a quite simple task (ACL2 is able to find the proof of the theorem without any additional help) because the functor functions have been defined following the mathematical definition of the direct sum of chain complexes.

```
(defthm properties-ds
    (and (implies (and (integerp n)
                       (inv-ds n x))
                  (and (equal (list (car x) (cadr x)) x)
                       (inv-A n (car x))
                       (inv-B n (cadr x))))
         (implies (integerp n)
                  (equal (diff-ds n a)
                         (list (diff-A n (car a))
                               (diff-B n (cadr a)))))))
```

Then, we have two ACL2 theorems which guarantee that our implementation of the direct sum of chain complexes really models the direct sum of chain complexes.

This result can be reused for every pair of concrete chain complexes by means of the derived rule of inference called *functional instantiation*. To make the instantiation of this result easier, we have used the generic instantiation tool developed by Martín-Mateos et al. presented in Subsection 6.1.2. This tool provides a way to develop a generic theory and to instantiate the definitions and theorems of the theory for different implementations. To be more concrete, the generic instantiation tool is used as follows. First of all, we define a constant `*ds*` whose value is the list of the functor functions (`inv-ds`, ..., `diff-ds`) and both `DIRECT-SUM-A-B-is-a-chain-complex` and `properties-ds` theorems. Subsequently, we include, at the end of the book where we have developed our generic theory about the direct sum constructor, a call to the macro `make-generic-theory` of the generic instantiation tool with the constant `*ds*` as argument:

```
(make-generic-theory *ds*)
```

The above macro generates the macro named `definstance-*ds*` when the generic book is included in other developments. When we want to construct the direct sum of two particular chain complexes in a new book, we simply call the macro `definstance-*ds*` with two arguments: the first one is an association list that relates the names of the concrete functions of the two particular chain complexes to the names of the functions of the generic chain complexes defined in the encapsulate of the direct sum book; and the second argument is provided to give new names to the events obtained by instantiation, that are the functor functions and the two theorems that state the correctness of the direct sum constructor; that is, the events of the constant `*ds*`. For instance, if we have the definitions of the chain complexes C and D:

```
(defun inv-C (n x) ... )
 ...
(defun diff-D (n a) ... )
 ...
(defun inv-D (n x) ...)
 ...
(defun diff-D (n a) ...)
 ...
(make-chain-complex :inv-cc 'inv-C ... :diff-cc 'diff-C
(make-chain-complex :inv-cc 'inv-D ... :diff-cc 'diff-D)
```

and if we have proved that these chain complex instances are really chain complexes with the macro `check-chain-complex-p`:

```
(check-chain-complex-p (make-chain-complex :inv-cc 'inv-C ... :diff-cc 'diff-C) 'C)

(check-chain-complex-p (make-chain-complex :inv-cc 'inv-D ... :diff-cc 'diff-D) 'C)
```

then, the simple macro call:

```
(definstance-*ds*
    ((inv-A inv-C) ... (diff-A diff-C) ...
     (inv-B inv-D) ... (diff-B diff-D) ...) "-C-D")
```

automatically defines the functions `inv-ds-C-D`, ..., `diff-ds-C-D` from the original ones `inv-ds`, ..., `diff-ds` and instantiates the theorems related to the correctness of the direct sum construction producing two theorems: `properties-ds-C-D` and `DIRECT-SUM-A-B-is-a-chain-complex-C-D`. In this way, the direct sum of the concrete chain complexes C and D is defined and the proof of the fact that it is a chain complex and follows the definition given in Kenzo is automatically generated without any additional user interaction.

6.4.1.2 The general case

In the previous subsubsection, the correctness of the direct sum constructor of the Kenzo system has been stated in ACL2. The methodology followed in that case can be extrapolated to prove the correctness of all the construction of Kenzo objects from other ones. This is due to the fact that they implement mathematical constructions which always follow the same pattern.

Pattern 6.3. Let $X_1, \ldots, X_n$ be objects belonging to the mathematical structures $T_1, \ldots, T_n$ respectively such that they satisfy the properties $A_1, \ldots, A_m$. Then, the ϕ constructor of $X_1, \ldots, X_n$, $\phi(X_1, \ldots, X_n)$, is an object belonging to the mathematical structure T such that the rules $R_1, \ldots, R_k$ are satisfied.

We have undertaken the task of developing a methodology in ACL2 which allows us to verify that the Kenzo constructors which implement the different constructors which follow the above pattern act as they are supposed to.

First of all, we define each one of the mathematical structures $T_1, \ldots, T_n$ in the way explained in Section 6.3. That is to say, we encode the algebraic structures $T_1, \ldots, T_n$ with the records `T1`, ..., `Tn` (that can be instantiated with the functions `make-T1`, ..., `make-Tn`) and define the macros `check-T1-p`, ..., `check-Tn-p`, that check the properties of the respective mathematical structure for different record instances.

Then, we can define concrete instances of the mathematical structures T_1,..., T_n and verify that they really are objects belonging to those mathematical structures, but we want to prove that the Kenzo implementation of $\phi(X_1, \ldots, X_n)$ is correct for every sequence of objects $X_1, \ldots, X_n$ belonging to the mathematical structures $T_1, \ldots, T_n$ such that they satisfy the properties $A_1, \ldots, A_m$. Then, we follow the approach presented in Subsection 6.3.5 where a way of developing generic theories about mathematical structures is provided by means of the combination of encapsulates and the functional instantiation rule. Thus, a representation of a sequence of generic objects $X_1, \ldots, X_n$ belonging to the mathematical structures $T_1, \ldots, T_n$ such that they satisfy the properties $A_1, \ldots, A_m$ is given in ACL2 by means of the following encapsulate.

```
(encapsulate
 ; Signatures
 ...
 ; Witnesses
 ...
 ; Properties
 (check-T1-p (make-T1 ...) 'X1)
 ...
 (check-Tn-p (make-Tn ...) 'Xn)

 (defthm property-A1 ...)
 ...
 (defthm property-Am ...))
```

The effect of the above encapsulate is an extension of the ACL2 universe in which the functions that determine a sequence of generic objects $X_1, \ldots, X_n$ belonging to the mathematical structures $T_1, \ldots, T_n$ such that satisfy the properties $A_1, \ldots, A_m$ (specified respectively by means of `property-A1`, ..., `property-Am` theorems) have been defined. Now, our intention consists of implementing $\phi(X_1, \ldots, X_n)$, based on the Kenzo definition, and prove that such constructor acts like $\phi(X_1, \ldots, X_n)$; that is to say, it belongs to the mathematical structure T and satisfies the rules $R_1, \ldots, R_k$.

From the generic functions defined in the previous encapsulate, we develop the constructor $\phi(X_1, \ldots, X_n)$ based on its Kenzo definition. This task is performed by the definition of a set of functions that depends on the functions which define the generic objects $X_1, \ldots, X_n$; which are called *functor functions*. The functor functions are gathered in a `T` record which encodes the type T of $\phi(X_1, \ldots, X_n)$. Then, we need to prove that the new object belongs to the mathematical structure T with the macro function `check-T-p`.

```
(check-T-p (make-T ...) 'phi)
```

Let us remark that macros like `check-T-p` generate a proof attempt that can be unsuccessful; then some auxiliary lemmas are needed to guide the ACL2 system to prove the result.

The last step consists of proving that our constructor fulfills the rules R_1,..., R_k of the constructor $\phi(X_1, \ldots, X_n)$. These rules are proved by means of a bunch of ACL2 theorems `rule-R1`, ..., `rule-Rk` which describe respectively the rules $R_1, \ldots, R_k$. The task of verifying these theorems is usually straightforward because the Kenzo constructors are normally defined following the rules $R_1, \ldots, R_k$; so, in some cases ACL2 is able to find the proof in the first attempt without any additional help; but the trickiest parts of the proofs usually need external help.

Hence, we have provided a methodology to prove theorems which guarantees that the Kenzo implementation of $\phi(X_1, \ldots, X_n)$ acts like it is supposed to. This kind of results is involved in approximately the 60% of the Kenzo code. However, in spite of being a good starting point to verify the correctness of Kenzo constructors from other spaces, our methodology does not become this task trivial because of the fact that the Kenzo implementation of some operators, such as the loop space of simplicial sets or the cobar of a coalgebra, are quite complicated and, therefore, verifying its correctness can be a challenging task.

Eventually, in order to reuse the results of the generic books developed in the way presented throughout this subsubsection, we use the derived rule of inference named *functional instantiation*. This tool allows us to instantiate theorems about partially defined functions for new functions if they are known to have the same properties. Nevertheless, the number of events to instantiate can be high, so, to overcome this problem we have used the generic instantiation tool presented in Subsection 6.1.2. This tool provides a way to develop a generic theory and to instantiate the definitions and theorems for different implementations in an easy way. This tool is used as follows, we define a constant `*theory*` whose value is the list of functor functions, the theorem produced in the call `(check-T-p (make-T ...) 'phi)` and the properties related to the mathematical structure: `rule-R1`, ..., `rule-Rk`. Finally, at the end of the book, let us call it "generic.lisp", we include a call to the macro of the generic instantiation tool `make-generic-theory` with the constant `*theory*` as argument.

This macro automatically defines a new macro called `definstance-*theory*` when the book "generic.lisp" is included by a user in other developments. Then, the events (the functor functions and the theorems that state the correctness of the constructor) of the generic book can be easily instantiated with only a simple macro call to `definstance-*theory*` with two arguments: the first one is an association list that relates the names of the concrete functions to the names of the generic functions defined in the encapsulate of the book "generic.lisp"; and the second argument is provided to give new names to the events obtained by instantiation, that are the functor functions and the theorems that state the correctness of the operator $\phi(X_1, \ldots, X_n)$. In this way, if we give objects $Y_1, \ldots, Y_n$ that are concrete instances of the mathematical structures $T_1, \ldots, T_n$ satisfying the properties $A_1, \ldots, A_m$, the object $\phi(Y_1, \ldots, Y_n)$, based on the Kenzo implementation, is defined and a proof of its correctness is automatically generated without any additional user interaction.

6.4.2 Case studies

In order to evaluate how practical our methodology was, we have started formalizing some important results included in the Kenzo system. In this section, we present some of them. The complete development of these results and also more examples can be consulted in [Her11]. Our main source of inspiration for the proofs of the results presented in this subsection was some lectures notes on Constructive Homological Algebra by J. Rubio and F. Sergeraert [RS06].

6.4.2.1 The Easy Perturbation Lemma

The Easy Perturbation Lemma was stated in Theorem 1.44. This lemma is one basic result in the effective homology method and has been previously formalized both in Isabelle and Coq [AD09]. To facilitate the reading of this part of the memoir, the Easy Perturbation Lemma is stated as follows.

Theorem (Easy Perturbation Lemma, EPL, Theorem 1.44). Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{N}}$ be two chain complexes, $\rho = (f, g, h) : C_* \Rightarrow\!\!\!\!\!\Rightarrow D_*$ a reduction, and δ_D a perturbation of d_D. Then a new reduction $\rho' = (f', g', h') : C'_* \Rightarrow\!\!\!\!\!\Rightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from C_* by replacing the old differential d_C by

the perturbed differential $d_C + g \circ \delta_D \circ f$;

2) the new chain complex D'_* is obtained from the chain complex D_* only by replacing the old differential d_D by $d_D + \delta_D$;

3) $f' = f$;

4) $g' = g$;

5) $h' = h$.

Let us note that the Easy Perturbation Lemma defines an operator which follows Pattern 6.3:

- the objects $X_1, \ldots, X_n$ are: $\rho = (f, g, h) : C_* \Rrightarrow D_*$ (where $C_* = (C_n, d_{C_n})_{n \in \mathbb{N}}$ and $D_* = (D_n, d_{D_n})_{n \in \mathbb{N}}$) and δ_D;
- the mathematical structures $T_1, \ldots, T_n$ are: reduction (the type of ρ) and morphism (the type of δ_D);
- there is just one property about ρ and δ_D, namely the fact that δ_D is a perturbation of d_D;
- $\phi(\rho, \delta_D)$ is the object which constructs $\rho' = (f', g', h') : C'_* \Rrightarrow D'_*$,
- the mathematical structure T is reduction, and
- the rules $R_1, \ldots, R_k$ are: $C'_* = (C_n, d_{C_n} + g \circ \delta_D \circ f)_{n \in \mathbb{Z}}$, $D'_* = (D_n, d_{D_n} + \delta_D)_{n \in \mathbb{Z}}$, $f' = f$, $g' = g$ and $h' = h$.

Then, following the guidelines given in Subsubsection 6.4.1.2 we proceed as follows. First of all, we have defined the necessary records and macros related to reductions and morphisms. Due to the fact that both reductions and morphisms are operators, we followed the guidelines given in Subsection 6.3.3 to model these operations in ACL2. Namely, a reduction is encoded by means of the following record.

```
(defstructure reduction
    tcc bcc f g h)
```

In the instances of the `reduction` record, the value of the slots `tcc` and `bcc` are chain complex instances and the value of the slots `f`, `g` and `h` are function symbols.

Afterwards, a generic reduction and a generic perturbation of the chain complex D_* of the generic reduction are declared using the encapsulate mechanism.

```
(encapsulate
...
(check-reduction-p
 (make-reduction
  :tcc (make-chain-complex :group-operation-cc 'group-operation-top ...
                           :diff-cc 'diff-top)
  :bcc (make-chain-complex ...) :f 'f  :g 'g  :h 'h) 'original-reduction)

(check-morphism-p (make-morphism ... :map 'delta) 'delta)

(defthm perturbation-property ...) )
```

From these objects, the functor functions can be defined to declare the new reduction. For instance the differential of the new chain complex C'_* of the new reduction is $(d_{top} + g \circ \delta_D \circ f)$ which is declared in ACL2 as follows.

```
(defun new-diff-top (n a)
  (group-operation-top (1- n) (diff-top n a) (g (1- n) (delta n (f n a)))))
```

Once we have defined all the functions which determine the new reduction, we can gather them in a new `reduction` instance and prove that they really determine a reduction.

```
(check-reduction-p
 (make-reduction
  :tcc (make-chain-complex ... :diff-cc 'new-diff-top)
  :bcc (make-chain-complex ...)
  :f 'f :g 'g :h 'h) 'EPL)
```

The proof attempt generated by the call to the above macro is unsuccessful in the first attempt; so we need the proof of some auxiliary lemmas. It is worth noting that the macro `check-reduction-p` checks 83 properties that are necessary to certify that an object of the `reduction` structure determines a reduction; however, most of them are automatically obtained by ACL2 and only the most tricky ones (namely 13 properties) need auxiliary lemmas. These auxiliary lemmas are formally proved in ACL2 by applying equational reasoning over morphisms, following closely the style of a *paper and pencil proof*.

Eventually, we need to prove that the new reduction is the one that we are looking for; that is to say, we need to prove the rules $R_1, \ldots, R_n$ which in this case are: $C'_* = (C_n, d_{C_n} + g \circ \delta_D \circ f)_{n \in \mathbb{Z}}$, $D'_* = (D_n, d_{D_n} + \delta_D)_{n \in \mathbb{Z}}$, $f' = f$, $g' = g$ and $h' = h$. However, as usual, this ACL2 proof is straightforward and does not need any help from the user to succeed.

6.4.2.2 Composition of reductions

The example presented in this subsubsection comes from the composition of reductions theorem, see Theorem 1.40. As in the case of the EPL, in order to facilitate the reading of this part of the memoir, the composition of reductions theorem is stated as follows.

Theorem (Composition of reductions, Theorem 1.40)**.** Let $\rho = (f, g, h) : C_* \Rrightarrow D_*$ and $\rho' = (f', g', h') : E_* \Rrightarrow F_*$ be two reductions such that $D_* = E_*$. Then a new reduction $\rho'' = (f'', g'', h'') : C'_* \Rrightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex C_*;

2) D'_* is the chain complex F_*;

3) $f'' = f' \circ f$;

4) $g'' = g \circ g'$;

5) $h'' = h + g \circ h' \circ f$.

Let us note that the composition of reductions Theorem defines an operator which follows Pattern 6.3:

- the objects $X_1, \ldots, X_n$ are: $\rho = (f, g, h) : C_* \Rrightarrow D_*$ and $\rho' = (f', g', h') : E_* \Rrightarrow F_*$;
- the only mathematical structure is the reduction (the type of ρ and ρ');
- there is just one property about ρ and ρ', namely the fact that $D_* = E_*$;
- $\phi(\rho, \rho')$ is the object which constructs $\rho'' = (f'', g'', h'') : C'_* \Rrightarrow D'_*$;
- the mathematical structure T is the reduction, and
- the rules $R_1, \ldots, R_k$ are: $C'_* = C_*$, $D'_* = F_*$, $f'' = f' \circ f$, $g'' = g \circ g'$ and $h'' = h + g \circ h' \circ f$.

The verification of the correctness of the Kenzo constructor which implements the composition of reductions follows the same schema presented in Subsubsection 6.4.2.1 to verify the correctness of the construction related to the Easy Perturbation Lemma.

First of all, two generic reductions are declared using the encapsulate mechanism.

```
(encapsulate
...
(check-reduction-p
 (make-reduction
  :tcc (make-chain-complex ...)
  :bcc (make-chain-complex ...)
  :f 'f1  :g 'g1  :h 'h1) 'reduction-1))

(check-reduction-p
 (make-reduction
  :tcc (make-chain-complex ...)
  :bcc (make-chain-complex ...)
  :f 'f2  :g 'g2  :h 'h2) 'reduction-2))
```

Subsequently, the functions which determine the new reduction are defined. For instance the f function of the new reduction is defined as follows.

```
(defun f (n a)
   (f1 n (f2 n a)))
```

Afterwards, we can gather those functions in a `reduction` instance and prove that they really determine a reduction.

```
(check-reduction-p
 (make-reduction
  :tcc (make-chain-complex ...)
  :bcc (make-chain-complex ...)
  :f 'f :g 'g :h 'h) 'cmps-reduction)
```

As in the case of the Easy Perturbation Lemma, the proof effort is considerably reduced: from the 83 properties that are necessary to certify that the functions of a `reduction` instance determine a reduction, only 11 of them need some auxiliary lemma to be proved.

Eventually, we need to prove that the new reduction is the one that we are looking for.

6.4.2.3 The Cone constructor

The next example is related to the cone construction of a chain complex morphism, see Definition 5.28. This example is similar to the construction of the direct sum of two chain complexes.

Let us note that the cone of a chain complex morphism defines an operator which follows Pattern 6.3:

- there is only one initial object: $\varphi : B \to A$;
- there is only one initial mathematical structure: chain complex morphism (the type of φ);
- in this case there are not properties $A_1, \ldots, A_m$;
- $\phi(\varphi)$ constructs the object $Cone(\varphi) = (C_n, d_n)_{n\in\mathbb{Z}}$;
- the mathematical structure T is chain complex, and
- the rules $R_1, \ldots, R_k$ are $C_n = (A_n, B_{n-1})$ and $d_n = (da_n + \varphi_n, -db_{n-1})$.

The verification of the correctness of the Kenzo constructor which implements the cone of a chain complex morphism follows the same schema presented in Subsubsection 6.4.1.1 to verify the correctness of the construction related to the direct sum of chain complexes.

First of all, a generic chain complex morphism is defined using the encapsulate mechanism.

```
(encapsulate
...
(check-chain-complex-morphism-p
 (make-chain-complex-morphism-p
  :source (make-chain-complex ...)
  :target (make-chain-complex ...)
  :map 'varphi) 'varphi))
```

Subsequently, the functions which determine the cone chain complex are provided. Afterwards, we can gather those functions in a `chain complex` instance and prove that they really determine a chain complex.

```
(check-chain-complex-p (make-chain-complex ...) 'cone-varphi)
```

Finally, we ACL2 successfully prove that $C_n = (A_n, B_{n-1})$ and $d_n = (da_n + \varphi_n, -db_{n-1})$, that are the rules $R_1, \ldots, R_n$ in this case.

6.4.2.4 Cone Equivalence Theorem

In the previous subsubsections we have presented several generic constructors which can be instantiated for concrete objects thanks to the generic instantiation tool presented in Subsection 6.1.2. However, we can also use the generic theories previously developed to generate new generic theories as we are going to see in the following two examples.

The following result is related to the construction of the equivalence of the cone of a chain complex morphism. This result was presented in Theorem 5.29, but we restate it here to make the reading of this subsubsection easier.

Theorem (Cone Equivalence Theorem, Theorem 5.29). Let $A_* \Lleftarrow\!\!\!\Rrightarrow HA_*$ and $B_* \Lleftarrow\!\!\!\Rrightarrow HB_*$ be two equivalences and φ a chain complex morphism between B_* and A_*:

$$\begin{tikzcd}
 & DA_* \arrow[loop above, "lha"] \arrow[loop above, "rha"'] \arrow[dl, shift left, "lfa"] \arrow[dr, shift left, "rfa"] & & & DB_* \arrow[loop above, "lhb"] \arrow[loop above, "rhb"'] \arrow[dl, shift left, "lfb"] \arrow[dr, shift left, "rfb"] & \\
A_* \arrow[ur, shift left, "lga"] & & HA_* \arrow[ul, shift left, "rga"] & B_* \arrow[lll, bend left, "\varphi"] \arrow[ur, shift left, "lgb"] & & HB_* \arrow[ul, shift left, "rgb"]
\end{tikzcd}$$

then we can construct an equivalence for $Cone(\varphi)$.

The proof of this theorem provides all the formulas to define an operator which follows Pattern 6.3:

- the objects $X_1, \ldots, X_n$ are: $A \Lleftarrow\!\!\!\Rrightarrow HA$, $B \Lleftarrow\!\!\!\Rrightarrow HB$ and $\varphi : B \to A$;
- the mathematical structures $T_1, \ldots, T_n$ are: equivalence (the type of $A \Lleftarrow\!\!\!\Rrightarrow HA$ and $B \Lleftarrow\!\!\!\Rrightarrow HB$) and chain complex morphism (the type of φ);
- in this case there are not properties $A_1, \ldots, A_m$;
- $\phi(A \Lleftarrow\!\!\!\Rrightarrow HA, B \Lleftarrow\!\!\!\Rrightarrow HB, \varphi)$ constructs the object:

$$\begin{tikzcd}
 & DC_* \arrow[loop above, "lh"] \arrow[loop above, "rh"'] \arrow[dl, shift left, "lf"] \arrow[dr, shift left, "rf"] & \\
Cone(\varphi) \arrow[ur, shift left, "lg"] & & HC_* \arrow[ul, shift left, "rg"]
\end{tikzcd}$$

- the mathematical structure T is the equivalence, and
- the rules $R_1, \ldots, R_k$ are:
  - $DC_* = Cone(lga \circ \varphi \circ lfb)$;
  - $HC_* = Cone(rfa \circ lga \circ \varphi \circ lfb \circ rgb)$;
  - $lf = (lfa, lfb)$;
  - $lg = (lga, lgb)$;

- $lh = (lha, -lhb)$;
- $rf = (rha + rfa \circ lga \circ \varphi \circ lfb \circ rhb, rhb)$;
- $rg = (rga - rha \circ lga \circ \varphi \circ lfb \circ rgb, rgb)$, and
- $rh = (rha + rha \circ lga \circ \varphi \circ lfb \circ rhb, -rhb)$.

In this case, the verification of the correctness of the Kenzo constructor which implements the equivalence constructor associated with the Cone Equivalence Theorem has some steps that we consider interesting enough to be detailed.

First of all, as we have done in the rest of the cases, an encapsulate which defines two generic equivalences $A \Lleftarrow\!\!\Rrightarrow HA$, $B \Lleftarrow\!\!\Rrightarrow HB$ and a generic chain complex morphism $\varphi : B \to A$ is constructed.

```
(encapsulate
...
(check-heq-p
 (make-heq
  :lrdc (make-reduction ... :f 'lfa :g 'lga :h 'lha)
  :rrdc (make-reduction ... :f 'rfa :g 'rga :h 'rha)) 'heq-a)

(check-heq-p
 (make-heq
  :lrdc (make-reduction ... :f 'lfb :g 'lgb :h 'lhb)
  :rrdc (make-reduction ... :f 'rfb :g 'rgb :h 'rhb)) 'heq-b)

(check-chain-complex-morphism-p
 (make-chain-complex-morphism-p
  :source (make-chain-complex ...)
  :target (make-chain-complex ...)
  :map 'varphi) 'varphi))
```

Subsequently, the book where we have defined the cone constructor, let us call it "cone.lisp", is included.

```
(include-book "cone")
```

When the "cone.lisp" book is included, the macro `definstance-*cone*` is defined. This macro allows us to build the cone of a chain complex morphism and instantiate the theorems which assert the correctness of that construction. In this way, our amount of work is dramatically reduced because with this tool the chain complexes: $C_* = Cone(\varphi)$, $DC_* = Cone(lga \circ \varphi \circ lfb)$ and $HC_* = Cone(rfa \circ lga \circ \varphi \circ lfb \circ rgb)$ are defined and the theorems which assert their correctness are instantiated.

```
(definstance-*cone*
  (...
   (varphi varphi))
  "-cone-A-B"))
```

```
(defun lga-varphi-lfb (n a)
  (lga n (varphi n (lfb n a))))

(definstance-*cone*
  (...
   (varphi lga-varphi-lfb))
  "-cone-DA-DB"))
```

```
(defun rfa-lga-varphi-lfb-rgb (n a)
  (rfa n (lga-varphi-lfb n (rgb n a))))

(definstance-*cone*
  (...
   (varphi rfa-lga-varphi-lfb-rgb))
  "-cone-HA-HB"))
```

Afterwards, the morphisms lf, lg, lh, rf, rg and rh are defined following the rules given in the proof of the Cone Equivalence Theorem, which are also followed by the Kenzo implementation. Afterwards, we can gather the functions generated in the instantiation process and the ones used to define the morphisms lf, lg, lh, rf, rg and rh, in an `equivalence` instance and prove that they really determine an equivalence.

```
(check-heq-p
 (make-heq
  :lrdc (make-reduction ... :f 'lf :g 'lg :h 'lh)
  :rrdc (make-reduction ... :f 'rf :g 'rg :h 'rh)) 'cone-equivalence)
```

Eventually, we need to prove that the above equivalence is the one that we are looking for. However, it is worth noting that the rules related to $DC_* = Cone(lga \circ \varphi \circ lfb)$ and $HC_* = Cone(rfa \circ lga \circ \varphi \circ lfb \circ rgb)$ were instantiated when we defined these objects from the generic cone theory; so, here the proof effort is reduced thanks to the generic instantiation tool and we only need to prove that lf, lg, lh, rf, rg and rh are well defined, a straightforward task.

6.4.2.5 The SES_1 Theorem

This last example uses the generic theories previously developed to generate a generic theory related to SES_1 Theorem, see Theorem 5.32.

Theorem (SES_1 Theorem, Theorem 5.32)**.** Let

$$\begin{tikzcd}
0 & A \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C \arrow[l, shift left, "i"] & 0 \arrow[l]
\end{tikzcd}$$

be an effective short exact sequence of chain complexes and $B \twoheadleftarrow\!\!\twoheadrightarrow HB$, $C \twoheadleftarrow\!\!\twoheadrightarrow HC$ be two equivalences. Then an equivalence for the chain complex A can be constructed.

The proof of this theorem provides all the formulas to define an operator which follows Pattern 6.3:

- the objects $X_1, \ldots, X_n$ are: A, $B \twoheadleftarrow\!\!\twoheadrightarrow HB$, $C \twoheadleftarrow\!\!\twoheadrightarrow HC$, i, j, σ and ρ;
- the mathematical structures $T_1, \ldots, T_n$ are: chain complex (the type of A), equivalence (the type of $B \twoheadleftarrow\!\!\twoheadrightarrow HB$ and $C \twoheadleftarrow\!\!\twoheadrightarrow HC$), chain complex morphism (the type of i and j) and morphism (the type of σ and ρ);
- the properties $A_1, \ldots, A_m$ state that A, B, C, i, j, σ and ρ determine an effective short exact sequence;
- $\phi(A, B \twoheadleftarrow\!\!\twoheadrightarrow HB, C \twoheadleftarrow\!\!\twoheadrightarrow HC, i, j, \sigma, \rho)$ constructs the object $A \twoheadleftarrow\!\!\twoheadrightarrow HA$,
- the mathematical structure T is the equivalence, and
- the rules $R_1, \ldots, R_k$ are the properties which determine the construction of $A \twoheadleftarrow\!\!\twoheadrightarrow HA$, the interested reader can see them in [RS06].

The procedure to construct the equivalence of the chain complex A given in the proof of SES_1 Theorem is as follows (we present here a sketch of the procedure, the complete one can be consulted in [RS06]). First of all, the chain complex $Cone(i)$ is constructed; afterwards, we apply the Cone Equivalence Theorem obtaining the equivalence $Cone(i) \twoheadleftarrow TC \twoheadrightarrow RC$. Subsequently, a reduction $A \twoheadleftarrow Cone(i)$ is constructed. Eventually, using Theorem 1.40 the reductions $A \twoheadleftarrow Cone(i)$ and $Cone(i) \twoheadleftarrow TC$ can be composed and the equivalence $A \twoheadleftarrow TC \twoheadrightarrow RC$ is obtained.

This process is translated into ACL2 as follows. First of all, as usual, an encapsulate which defines a chain complex A, two equivalences $B \twoheadleftarrow\!\!\twoheadrightarrow HB$ and $C \twoheadleftarrow\!\!\twoheadrightarrow HC$, two chain complex morphisms i and j and two morphisms σ and ρ such that they determine a short exact sequence is constructed.

```
(encapsulate
...
(check-chain-complex-p (make-chain-complex ...) 'A)

(check-heq-p (make-heq ...) 'heq-b)

(check-heq-p (make-heq ...) 'heq-c)

(check-chain-complex-morphism-p (make-chain-complex-morphism-p ... :map 'i) 'i)

(check-chain-complex-morphism-p (make-chain-complex-morphism-p ... :map 'j) 'j)

(check-morphism-p (make-morphism-p ... :map 'sigma) 'sigma)

(check-morphism-p (make-morphism-p ... :map 'rho) 'rho)

(defthm rho-i ...)
(defthm i-rho-sigma-j ...)
(defthm j-sigma ...)
(defthm rho-sigma ...)
(defthm j-i ...)
```

Subsequently, the book where we have defined the Cone Equivalence Theorem, let us call it “cone-equivalence.lisp”, is included.

```
(include-book "cone-equivalence")
```

The macro `definstance-*cone-equivalence*` is defined when the “cone-equivalence.lisp” book is included. This macro builds from two equivalences $E \Lleftarrow\!\!\!\Rrightarrow HE$, $F \Lleftarrow\!\!\!\Rrightarrow HF$ and a chain complex morphism $\varphi : F \to E$ the equivalence $Cone(\varphi) \Lleftarrow\!\!\!\Rrightarrow H$ with the formulas presented in the previous subsection. In this way, our amount of work is dramatically reduced because with this tool we can obtain $Cone(i) \Lleftarrow TC \Rrightarrow RC$ from $B \Lleftarrow\!\!\!\Rrightarrow HB$, $C \Lleftarrow\!\!\!\Rrightarrow HC$ and $i : B \leftarrow C$.

```
(definstance-*CONE-equivalence*
  (...
   (varphi i))
  "-cone-SES1")
```

Afterwards, the reduction $A \Lleftarrow Cone(i)$ is defined following the formulas given in the proof of the SES_1 Theorem.

```
(check-reduction (make-reduction ...) 'cone=>>A)
```

Subsequently, the book where we have defined the composition of reductions, let us call it “reduction-cmps.lisp”, is included.

```
(include-book "reduction-cmps")
```

When the "reduction-cmps.lisp" book is included, the macro `definstance-*cmps*` is defined. This macro builds from two reductions $A1 \Lleftarrow B1$ and $B1 \Lleftarrow C1$ a new reduction $A1 \Lleftarrow C1$ with the formulas provided in Theorem 1.40. In this way, our amount of work is dramatically reduced because with this tool we can obtain $A \Lleftarrow TC$ from $A \Lleftarrow Cone(i)$ and $Cone(i) \Lleftarrow TC$.

```
(definstance-*cmps*
  (...)
  "-tc-A")
```

Eventually, gathering the reductions $A \Lleftarrow TC$ and $TC \Rrightarrow RC$, the equivalence $A \Lleftarrow TC \Rrightarrow RC$ is obtained.

Here again, we see the importance of using the generic instantiation tool to instantiate the generic theories because of the fact that the number of events which must be instantiated when we want to use the generic theories related to the Cone Equivalence Theorem and the composition of reductions is quite big; therefore, this tool makes this process easier.

6.4.3 Distance from ACL2 to Kenzo code

Let us present now the gap between the code that is used in our ACL2 proofs, related to this part of the memoir, and the actual Kenzo code.

In addition to the differences between the representation of mathematical structures in Kenzo and ACL2, which was explained in Subsection 6.3.6, there are other two main differences.

Kenzo objects such as chain complexes or reductions are defined over the ring $\mathbb{Z}$ (the most important case in Algebraic Topology; see [RS06]); on the contrary in our ACL2 formalization we work over a generic ring R; then the slots to define the ring R are included in our records. As we work over a generic ring R instead of working with $\mathbb{Z}$ the Kenzo results are a particular case of our formalization.

The second difference is related to the values of the slots of the mathematical structures. Several Kenzo functions have as input other functions, this feature is not allowed in the ACL2 theorem prover. Then, an uncurrying process is needed. For instance the definition of the values of `f`, `g` and `h` in the composition of reductions (Subsubsection 6.4.2.2) is given in Kenzo as follows.

```
(make-instance 'reduction
  ...
  :f (cmps bf tf)
  :g (cmps tg bg)
  :h (add th (i-cmps tg bh tf)))
```

where `bf`, `bg`, `bh`, `tf`, `tg` and `th` are respectively the functions f, g and h of the $C_* \Rrightarrow D_*$ and $D_* \Rrightarrow F_*$ reductions, `cmps` is the function in charge of the composition of two functions, `add` is the sum of functions and `i-cmps` is a macro composing n-functions with $n \geq 2$. The functions `cmps`, `add` and `i-cmps` take functions as arguments. Then, we need to uncurry these functions and then the functions `f`, `g` and `h` are defined in ACL2 as follows.

```
(defun f (n a) (tf n (bf n a)))
(defun g (n a) (bg n (tg n a)))
(defun h (n a) (group-operation-top (1+ n) (th n a) (tg (1+ n) (bh n (tf n a)))))
...
(make-reduction
  ...
  :f 'f
  :g 'g
  :h 'h)
```

The uncurrying technique is a well-known form of defunctionalization [Rey98] and can be considered as a safe transformation.

Therefore, we can consider that the transformations between real Kenzo code and the one that we are using in our ACL2 verification developments are safe.

Conclusions an further work

Conclusions

This work should be understood as a first step towards an integral assistant for research and teaching in Algebraic Topology. From our point of view, the main contributions accomplished in this work, gathered by chapters, are the following ones.

- Chapter 2:
    - We have showed how some *guidance* can be achieved in the field of Computational Algebraic Topology, without using standard Artificial Intelligence techniques. The idea is to build an infrastructure, giving a *mediated access* to an already-written symbolic computation system. Putting together both Kenzo itself and this infrastructure, we have produced the Kenzo framework which is able to be connected to different clients (desktop GUIs, web applications and so on). In general, this can imply a restriction of the full capabilities of the kernel system, but the interaction with it is easier and enriched, contributing to the objective of increasing the number of users of the system. This work has been presented in [HPRS08, HPR08b].
    - We have included the following intelligent enhancements in the Kenzo framework: (1) controlling the input specifications on constructors, (2) avoiding some operations on objects which would raise errors, (3) chaining methods in order to provide the user with new tools, and (4) using expert knowledge to obtain some results not available in the Kenzo system. This work has been published in [HP10b].
    - As a by-product, we have developed OpenMath Content Dictionaries devoted to Simplicial Algebraic Topology. Namely, for each one of the mathematical structures of the Kenzo system, an OpenMath Content Dictionary has been defined. The definitions given in these Content Dictionaries include the axiomatic parts and have been used to interoperate with the ACL2 deduction system. This work can be found in [HPR09c].
    - We have developed a first version of a system with the same aim that the Kenzo framework, but which can process queries concurrently and where persistent results are available. This work has been presented in [HPR09b].

- Chapter 3:
  - We have implemented a plug-in framework which allows us to add new functionality to the Kenzo framework without modifying the original source code. This plug-in framework has allowed us to increase the capabilities of the Kenzo framework by means of new Kenzo functionalities or the connection with other systems such as Computer Algebra systems or Theorem Prover tools.
  - Since the final users of the Kenzo framework are Algebraic Topology students, teachers or researchers that usually do not have a background in Common Lisp, we have developed a user-friendly front-end allowing a topologist (student, teacher or researcher) to use the Kenzo program guiding his interaction by means of the Kenzo framework, without being disconcerted by the Lisp technicalities which are unavoidable when using the Kenzo system. This work has been presented in [HPR08a, HPR09a].
  - Several profiles of interaction with our system can be considered; therefore, we have used the plug-in framework to be able to customize our graphical user interface. In this way, the user can configure the application depending on his needs. This work can be found in [HPR09d].
  - We have gathered the two frameworks (Kenzo framework and plug-in framework) and the graphical user interface in a new system called *fKenzo*. This work has been published in [HPRS11].
- Chapter 4:
  - We have achieved the integration, in the same system, of computation (thanks to both Kenzo and GAP) and deduction (by means of ACL2). Even if our proposal has a limited extend, both thematically and from the point of view of the core systems, we think it shows a solid line of research that could be exported to other areas of mathematical knowledge management. This work has been presented in [HPRR10].
  - We have not only achieved the integration of several tools in the same system, but also the interoperability among them (a problem much more difficult). On the one hand, we have automated the composability between Kenzo and GAP which was presented in [RER09] hiding the technical details of the integration of these systems to a final user. On the other hand, Kenzo, GAP and ACL2 work together in our framework to provide a powerful and reliable tool related to the construction of Eilenberg MacLane spaces of type $K(G, n)$ where G is a cyclic group; those spaces are instrumental in the computation of homotopy groups. This work has been published in [HPRR10].
- Chapter 5:
  - We have presented some programs that improve the functionality of Kenzo, generating simplicial complexes from its facets and building the simplicial set

canonically associated with a simplicial complex. Moreover, we have certified the correctness of the programs in the ACL2 Theorem Prover, increasing in this way the reliability of this new Kenzo module. This work has been presented in [HP10a].

  - We have developed a new Kenzo module devoted to study 2D and 3D monochromatic images based on the simplicial complex module. In addition, we have formalized our algorithms in ACL2; in this way, if we use our programs in real life problems (for instance, in the study of medical images), we are completely sure that the results produced by our programs are always correct.
  - We have presented an algorithm to build the effective homology of the pushout of simplicial sets with effective homology. As a by-product the wedge and join of simplicial sets can be built as particular cases of the pushout. The algorithms are implemented as a new Kenzo module which in turn is tested with some interesting case studies such as the projective space $P^2(\mathbb{C})$ or $SL_2(\mathbb{Z})$. This work has been presented in [Her10].
  - We have integrated all the modules presented in this chapter in *fKenzo*.

- Chapter 6:
  - We have designed a framework to prove the correctness of simplicial sets as implemented in the Kenzo system. As examples of application we have given a complete correctness proof of the implementation in Kenzo of spheres, standard simplicial sets and simplicial sets coming from simplicial complexes (modulo a safe translation of Kenzo programs to ACL2 syntax). By means of the same generic theory the correctness of other Kenzo simplicial sets can be proved. This work has been published in [HPR11].
  - We have provided a methodology to model mathematical structures in the ACL2 Theorem Prover. This modeling process has been used to partly implement an ACL2 algebraic hierarchy which is the foundation for verifying the correctness of the construction of spaces from other ones, and can be useful for works non related to our development.
  - We have presented a framework to prove the correctness of construction of spaces from other ones implemented in the Kenzo system. As examples of application, we have given, among others, a complete correctness proof of the implementation in Kenzo of the direct sum of chain complexes, the Easy Perturbation Lemma and the SES_1 theorem. By means of the same framework the correctness of other constructions can be proved.

To sum up, we can claim that we have developed an assistant for Algebraic Topology called *fKenzo*. This system not only provides a friendly front-end for using the Kenzo kernel, but also guides the user in his interaction with the system. In addition, this system can evolve at the same time as Kenzo. We have tested this feature with the development of several new Kenzo modules which have been integrated in *fKenzo* without any special hindrance. Moreover, the functionality of the system can also be extended by means of the integration with other systems such as Computer Algebra systems or Theorem Prover tools which not only work individually, but also cooperate to produce new tools. Eventually, we can claim that we have developed a reliable tool since we have partly verified the kernel of our system with the ACL2 Theorem Prover.

Open problems and further work

Several research lines, related to the work presented in this memoir, are still open, we gather them by chapters:

- Chapter 2:
    - The Homotopy Expert System (Paragraph 2.2.3.2.3) could be endowed with a higher level language to describe rules that could be made available for an algebraic topologist without any special computer science knowledge.
    - The algorithm implemented in the Homotopy Algorithm Module (Paragraph 2.2.3.2.2) can be completed thanks to the development presented in [RER09] which allows us to compute the effective homology of Eilenberg MacLane spaces of type $K(G, n)$ where G is a cyclic group. In this way, all the homotopy groups of 1-reduced simplicial sets with effective homology could be computed.
    - One of the most important issues to be tackled in the next versions of the Kenzo framework is how organizing the decision on when (and how) a calculation should be derived to a remote server. Some ideas about this problem were presented in Section 2.4.
    - The architecture presented in Section 2.5 will allow us to move to a distributed computing context. We are thinking of a federated architecture, where several rich clients communicate with a central powerful computing server. The central server acts as a general repository and also provides computing power to deal with difficult calculations. The most complicated problem is to devise good heuristics to decide what is the meaning of the fuzzy predicate "to be a difficult computation".
- Chapter 3:
    - One of the feasible enhancements for *fKenzo* could be to find a suitable way (free from the Common Lisp syntax) of editing and handling elements of each constructed space. Thus we could approach the difficult question of introducing in *fKenzo* computations as the homology groups of $P^2(\mathbb{C})$, see Subsubsection 5.3.4.

  - In the area of user interfaces, a feasible objective could be the development of a Web User Interface (WUI). At this moment the structure of a possible WUI is provided by means of the XUL specifications, the job now would consist of connecting the WUI with the framework to obtain a new usable front-end.
  - Also in the area of user interfaces, we could integrate a "drag and drop" equation editor, like DragMath [Bil], in the *fKenzo* user interface. The editor would let users build up mathematical expressions in a traditional two dimensional way, and then output the expression in OpenMath format. The OpenMath expression will be used to invoke the Kenzo framework.

- Chapter 4:

  - More computational and deduction capabilities could be integrated in our framework. From the Computer Algebra side, we can consider, for instance, the Cocoa [CoC] system, where some works related to Kenzo has been already done, see [dC08]. From the Theorem Proving side, we can consider Coq and Isabelle, which have been already used to formalize Kenzo algorithms, see [ABR08, DR11]. Moreover, as in the case of GAP and ACL2, the real interest does not lie in using them individually, but making them work in a coordinate and collaborative way to obtain new tools and results not reachable if we use individually each system.
  - It would be necessary to improve the interaction with the ACL2 system. At this moment the queries must be pre-processed (even if parametrized specification allows the system to cover complete families of structures: cyclic groups, spheres, simplicial complexes and so on); a comfortable way of introducing questions about the truth of properties of intermediary objects, dynamically generated during a computing session, should be provided.
  - We want to integrate more ACL2 certification capabilities in our system. The idea is to interact in a same friendly front-end with Kenzo and ACL2. For instance, the user could give information to construct a new simplicial set with Kenzo; then, he could provide minimal clues to ACL2 explaining why his construction is sensible; then ACL2 would produce a complete proof of the correctness of the construction. This kind of interaction between Computer Algebra systems and Theorem Provers would be very valuable, but severe difficulties related to finding common representation models are yet to be overcome.
  - A meta-language should be designed to specify how and when a new kernel system can be plugged in the framework. This capability and the necessity of orchestrating the different services suppose a real challenge, which could be explored by means of OpenMath technologies.

- Chapter 5:

  - A quite natural research line consists of developing new algorithms and programs. For instance, a challenging task is the implementation of the dual notion of pushout, called pullback [Mat76].

- ◦ In Subsubsection 5.1.5.2, we have presented two different programs which construct the simplicial complex associated with a list of simplexes. One of those programs was only implemented in Common Lisp since it uses a memoization strategy which cannot be directly implemented in ACL2. However, the work presented in [BH06] provides the mechanism to memoize functions in ACL2, then an optimized version of our algorithm could be implemented in ACL2. Moreover, we could use the semantic attachment procedure presented in Subsection 6.1.3 to use the already implemented not optimized version for reasoning and the new optimized one for executing.
- ◦ In the same way that we have applied the algorithms related to simplicial complexes in the study of digital images, we could develop new programs to be applied in different contexts such as coding theory or robotics. Likewise that in the case of digital images, if we want to apply our programs to real life problems, we must be completely sure that the results produced by our programs are correct. Therefore, the formal verification of our programs with a Theorem Prover would be significant.

- Chapter 6:
  - ◦ With the acquired experience, the methodology presented in Section 6.2 could be extrapolated to other algebraic Kenzo data structures. So, the work presented in that section can be considered a solid step towards our objective of verifying in ACL2 first order fragments of the Kenzo Computer Algebra system.
  - ◦ The hierarchy presented in Subsection 6.3.4 should be enriched by means of the rest of mathematical data structures implemented in Kenzo, see Subsection 1.2.1.
  - ◦ The methodology presented in Section 6.4 could be applied to several cases. In particular, it would be interesting the formalization of the construction of the effective homology of the pushout which involves several of the constructions already presented in this memoir.
  - ◦ The formalization of the computation of homology groups of spaces remains as further work. In this case we could focus on the computation of homology groups of spaces of finite type thanks to the Effective Homology method. Then, computing each homology group can be translated to a problem of diagonalizing certain integer matrices. Therefore, we should formalize this diagonalization process.

All these problems are interesting enough to be undertaken, however, we cannot tackle all of them at the same time. Therefore, our main priorities are:

- *Formalization of Homological Algebra and Algebraic Topology libraries.* Throughout this memoir we have presented the formalization of several results related to

Homological Algebra and Algebraic Topology in the ACL2 theorem prover. We are planning the formalization of more results in the future. To undertake this goal, we foresee the use of not only the ACL2 theorem prover but also the proof assistant Coq [BGBP08] as well as its SSReflect extension [GM09] and the libraries it provides. A first step in this line is the work [HPDR11], where the formalization in Coq of incidence matrices associated with simplicial complexes as well as the main theorem that gives meaning to the definition of homology groups are presented.

- *Integration of different Theorem Prover tools.* As explained in the previous item, we are planning to work simultaneously with two different proof assistants: ACL2 and Coq. This goal can be split in two different subgoals. On the one hand, we want to compare the different formalizations of the same problem in ACL2 and in Coq/SSReflect; a related work to this topic can be found in [AD10]. On the other hand, we are interesting in using ACL2 as oracle of inductive schemas for Coq/SSReflect; obtaining an interoperability which is, in some sense, similar to the one presented for the Kenzo and GAP Computer Algebra systems.

- *Applications to medical imagery.* The methods presented in this memoir related to digital images can be applied in the study of medical images. However, to this aim it is necessary to implement new tools in software systems. In addition, if we want to apply our methods in the study of medical images we must be completely sure that they are correct, therefore, a formalization process is required, too.

The main reason for choosing these goals, and no others, lies in the fact that they are three of the tasks assigned to La Rioja node of the Formath project [For]. This project has partially supported the work presented in this memoir.

Glossary

ACID A set of properties (Atomicity, Consistency, Isolation and Durability) that guarantee that transactions over a shared space are processed reliably. The atomicity rule said that a modification over the shared space is fully completed; otherwise the shared space state is unchanged. The consistency rule said that a modification of the shared state will take the shared space from one consistent state to another. The isolation rule refers to the requirement that other process cannot access data that has been modified during a modification that has not yet completed. Finally, durability rule ensures that a datum inserted in the shared space remains until a process explicitly deletes it. 85

active socket An active socket is connected to a remote socket via an open data connection. 93

architectural pattern An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. 42

binary semaphore A semaphore which has two operations (`P` and `V`) for controlling access by multiple processes to a common resource. The `P` operation sleeps the process until the resource controlled by the semaphore becomes available, at which time the resource is immediately claimed. The `V` operation is the inverse, it makes a resource available again after the process has finished using it. 93

bmp Image format used to store bitmap digital images. 209

byu Classic Brigham Young University file format for surfaces in 3D. 209

CLOS Common Lisp Object System (CLOS) [Gra96] is the facility for object-oriented programming which is part of ANSI Common Lisp. 21, 22, 48

component A part of a software system. A component has an interface that provides access to its services. On a programming language level components may be represented as modules, classes, objects or a set of related functions. 42, 53, 72

concurrency Property of systems in which several computations are executing simultaneously. 97

CORBA The Common Object Request Broker Architecture, a distributed object computing middleware standard defined by the Object Management Group (OMG), see [Gro]. 40

Curry A logic *functional programming* language, see [Han06], based on the *Haskell* language. It merges elements of functional and logic programming. 111

event An action initiated either by the user or the computer. An example of a user event is any mouse movement or a keystroke. 322

event handler A software routine that provides the processing for various *events* such as mouse movement, a mouse click, a keystroke or a spoken word. 112

expert system An expert system is an interactive computer-based decision tool that uses both facts and heuristics to solve decision problems based on knowledge acquired from an expert. 60

framework A software system intended to be instantiated. A framework defines the architecture for a system. 42

functional programming A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. 20, 322

hash table Data structures which use a hash function to map certain identifiers (keys) to associated values. 57

Haskell A *functional programming* language, see [Hut07]. 322

IDE Integrated Development Environment (IDE). An editor that lets the programmer edit, compile and debug all from within the same program. 106, 112

initial space In this text, a space whose definition does not involve any other space. 248

interface A publicly accessible portion of a component, subsystem, or application. 53, 55

jpeg Image format used by digital cameras and other photographic image capture devices. 209

jvx Geometry file format for curves, surfaces and volumes in n-dimensional space. 209

Maple A general-purpose commercial Computer Algebra system. 39

Mathematica A computational software program used in scientific, engineering, and mathematical fields and other areas of technical computing. 39

MathML Mathematical Markup Language (MathML) [A$^+$08] is a standard XML language adopted by the World Wide Web Consortium (W3C) as the approved way of expressing math on the web. 46

Maxima A free Lisp system for the manipulation of symbolic and numerical expressions, see [Sch09]. 40

memoization An optimization technique used to speed up computer programs by storing the results of function calls for later reuse, rather than recomputing them at each invocation of the function. 30, 44

module A portion of a program that carries out a specific function and may be used alone or combined with other modules of the same program. 116

native XML databases A data persistence software system that allows data to be stored in XML format; the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files. 85

obj Popular geometry format 3D geometry file format for surfaces in 3D originally used in Wavefront's Advanced Visualizer and now by Sun's Java3D. 209, 210, 215, 216

OMDoc An open markup language for mathematical documents, and the knowledge encapsulate in them, see [Koh06]. This format extends OpenMath and hence provides some features not available in OpenMath, for example a theory level and a way of incorporating executable code. 101

OpenMath An XML standard for representing mathematical objects with their semantics, allowing them to be exchanged between computer programs, stored in databases, or published on the worldwide web, see [Con04]. 46, 73

passive socket A passive socket is not connected, but rather awaits an incoming connection, which will spawn a new active socket. 93

pattern A pattern describes a problem that happens over and over, and also the solution to that problem. Patterns cover various ranges of scale and abstraction. Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components. 42

pbm Portable Bit Map (PBM). The PBM format is a monochrome file format for images. 209, 210, 214, 216

Phrasebook A piece of software which transforms from the internal representation of a data structure of an application to its OpenMath representation and viceversa. 45, 75

platform The software that a system uses for its implementations. Software platforms include operating systems, libraries, and frameworks. A platform implements a virtual machine with applications running on top of it. 42

plug-in A set of software components that adds specific capabilities to a larger software application. Plug-in based applications can be executed with no plug-ins. 99–102, 116

png Image format, that does not requiring a patent license, used to store bitmap digital images. 209

prefix notation Mathematical notation where the function is noted before the arguments it operates on. 36

process The actual running of a program module. 97

prompt A message on the computer screen indicating that the computer is ready to accept user input. In a command-line interface, the prompt may be a simple ">" symbol or "READY >" message, after which the user may type a command for the computer to process. 105

raster A rectangular grid of picture elements representing graphical data for displaying. 209

request An event sent by a client to a service provider asking it to perform some processing on the client's behalf. 45

semaphore A mechanism for controlling access by multiple processes to a common resource in a concurrent programming environment. 93

service A set of functionality offered by a service provider or server to its clients. 53, 72, 73

SOAP Simple Object Access Protocol is a standard protocol for exchanging structured information in the implementation of Web Services. 90

socket A mechanism for interprocess communication. A socket is an end-point of communication that identifies particular network address and port number. 93, 132–134

stack overflow An error which occurs when too much memory is used on the data structure that stores information about the active computer program. 256

stylesheet A program used to render an XML document into another format. 113, 121

thread One subprocess in a system which allows the concurrently execution of multiple streams of instructions within the same program. 98

UI User Interface. 111

web service A web service is traditionally defined by the World Wide Web Consortium as a software system designed to support interoperable machine-to-machine interaction over a network. 82, 90

well-formed XML XML that follows the XML tag rules listed in the W3C Recommendation for XML 1.0. A well-formed XML document contains one or more elements; it has a single document element, with any other elements properly nested under it; and each of the parsed entities referenced directly or indirectly within the document is well formed. 73, 75

XML enabled databases A data persistence software system that allows data to be stored in XML format; the internal model of such databases is a tradicional database (relational or object oriented). This databases do the conversion between XML and its internal representation. 85

XML schema A formal specification of element names that indicates which elements are allowed in an XML document, and in what combinations. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It defines whether an element is empty or can include text. The schema can also define default values for attributes. 46

XPath XML Path Language, is a query language for selecting nodes from an XML document. 85

XQuery XQuery is a query and functional programming language that is designed to query collections of XML data. 85

XSLT eXtensible Stylesheet Language Transformations (XSLT) [K$^+$07] are procedures, defined in XML, for converting one kind of XML into another. For viewing on the Web, an XSLT can be written for conversion to XHTML. 121

XUL XML User Interface [H$^+$00], it is Mozilla's XML-based user interface language which lets us build feature rich cross-platform applications defining all the elements of a User Interface. 111

Bibliography

[A$^+$08] R. Ausbrooks et al. Mathematical Markup Language (MathML). version 3.0 (third edition), 2008. `http://www.w3.org/TR/2008/WD-MathML3-20080409/`.

[Aas05] J. Aasman. Allegrocache: A high-performance object database for large complex problems. *In 5th International Lisp Conference. Standford University*, 2005.

[ABR08] J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4), pp. 271–292, 2008.

[ABR10] J. Aransay, C. Ballarin, and J. Rubio. Generating certified code from formal proofs: a case study in homological algebra. *Formal Aspects of Computing*, 2(22), pp. 193–213, 2010.

[AD09] J. Aransay and C. Domínguez. Modelling Differential Structures in Proof Assistants: The Graded Case. In *Proceedings 12th International Conference on Computer Aided Systems Theory (EUROCAST'2009)*, volume 5717 of *Lecture Notes in Computer Science*, pp. 203–210, 2009.

[AD10] J. Aransay and C. Domínguez. Formalizing simplicial topology in Isabelle/HOL and Coq. In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones científicas en honor de Mirian Andrés Gómez*, pp. 21–42. Universidad de La Rioja, 2010.

[ADFQ03] R. Ayala, E. Domínguez, A.R. Francés, and A. Quintero. Homotopy in digital spaces. *Discrete Applied Mathematics*, 125, pp. 3–24, 2003.

[ALR07] M. Andrés, L. Lambán, and J. Rubio. Executing in Common Lisp, Proving in ACL2. In *Proceedings 14th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2007)*, volume 4573 of *Lectures Notes in Artificial Intelligence*, pp. 1–12. Springer-Verlag, 2007.

[ALRRR07] M. Andrés, L. Lambán, J. Rubio, and J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. *Proceedings of ACL2 Workshop 2007*, pp. 34–39, 2007.

[APRR05] M. Andrés, V. Pascual, A. Romero, and J. Rubio. Remote Access to a Symbolic Computation System for Algebraic Topology: A Client-Server Approach. In *Proceedings 5th International Conference on Computational Science (ICCS'2005)*, volume 3516 of *Lecture Notes in Computer Science*, pp. 635–642. Springer-Verlag, 2005.

[B$^+$96] F. Buschmann et al. *Pattern-Oriented Software Architecture. A system of Patterns*, volume 1 of *Software Design Patterns*. Wiley, 1996.

[B$^+$07] F. Buschmann et al. *Pattern-Oriented Software Architecture. A Pattern Language for Distributed Computing*, volume 4 of *Software Design Patterns*. Wiley, 2007.

[B$^+$08] T. Bray et al. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. `http://www.w3.org/TR/REC-xml/`.

[Bai99] A. Bailey. *The machine-checked literate formalisation of algebra in type theory*. PhD thesis, Manchester University, 1999.

[BGBP08] Y. Bertot, G. Gonthier, S.O. Biha, and I. Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics (TPHOLS'08)*, volume 5170 of *Lecture Notes in Computer Science*, pp. 86–101, 2008.

[BGKM91] R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In V. Lifschiz, editor, *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honor of John McCarthy*, pp. 7–26. Academic Press, 1991.

[BH06] R. S. Boyer and W. A. Hunt. Function Memoization and Unique Object Representation for ACL2 Functions. In *Proceedings 6th International Workshop on the ACL2 Theorem Prover and Its Applications*, pp. 81–89, 2006.

[Bil] A. Billingsley. DragMath. `http://www.dragmath.bham.ac.uk/`.

[BKM96] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'1996)*, volume 1166 of *Lecture Notes in Computer Science*, pp. 275–293, 1996.

[BM84] R. Boyer and J S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3), pp. 441–458, 1984.

[BM97] R. Boyer and J S. Moore. Nqthm, the Boyer-Moore theorem prover, 1997. `ftp.cs.utexas.edu/pub/boyer/nqthm/index.html`.

[Bra96] R. Bradford. An Implementation of Telos in Common Lisp. *Object Oriented Systems*, 3, pp. 31–49, 1996.

[Bro57] E. H. Brown, Jr. Finite computability of Postnikov complexes. *Annals of Mathematics*, 65(1), pp. 1–20, 1957.

[Bro67] R. Brown. The twisted Eilenberg-Zilber theorem. *Celebrazioni Archimedi de Secolo XX, Simposio di Topologia*, pp. 34–37, 1967.

[Bro82] K. S. Brown. *Cohomology of Groups*. Springer-Verlag, 1982.

[Bro97] B. Brock. defstructure for ACL2 Version 2.0. Technical report, Computational Logic, Inc., 1997.

[BS84] B. G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems The MYCIN Experiments of the Standford Heuristic Programming Project*. Addison-Wesley, 1984.

[C$^+$07] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0, 2007. `http://www.w3.org/standards/techs/wsdl#w3c_all`.

[Cal] The Calculemus Project. `http://www.calculemus.net/`.

[CC99] O. Caprotti and A. Cohen. Connecting proof checkers and computer algebra using OpenMath. In *Proceedings 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLS'1999)*, volume 1690 of *Lecture Notes in Computer Science*, pp. 109–112, 1999.

[CDDP04] O. Caprotti, J. Davenport, M. Dewar, and J. Padget. Mathematics on the (semantic) net. In *Proceedings 1st European Semantic Web Symposium (ESWS'2004)*, volume 3053 of *Lecture Notes in Computer Science*, pp. 213–224, 2004.

[CH97] J. Calmet and K. Homann. Towards the Mathematics Software Bus. *Theoretical Computer Science*, 187, pp. 221–230, 1997.

[Chr03] J. Chrzaszcz. Implementing Modules in the Coq System. In *Proceedings 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *Lecture Notes in Computer Science*, pp. 270–286, 2003.

[CM95] G. Carlsson and R. J. Milgram. Stable homotopy and iterated loop spaces. In I.M. James, editor, *Handbook of Algebraic Topology*, pp. 505–583. North-Holland, 1995.

[CMDZ06] C. Cordero, A. De Miguel, E. Domínguez, and M. A. Zapata. Modelling interactive systems: an architecture guided by communication objects. *HCI related papers of Interaction 2004*, pp. 345–357, 2006.

[CoC] CoCoATeam. CoCoA: a system for doing Computations in Commutative Algebra. `http://cocoa.dima.unige.it`.

[Con04] The OpenMath Consortium. Openmath standard 2.0, 2004. http://www.openmath.org/standard/om20-2004-06-30/omstd20.pdf.

[CRZ03] A. B. Chaundry, A. Rashid, and R. Zicari. *XML data management: native XML and XML-enabled database systems.* Addison-Wesley, 2003.

[Dav99] J. Davenport. A Small OpenMath Type System. Technical report, University of Bath, 1999. http://www.openmath.org/standard/sts.pdf.

[Dav00] J. Davenport. On Writing OpenMath Content Dictionaries. Technical report, University of Bath, 2000. www.openmath.org/documents/writingCDs.pdf.

[dC08] E. Saenz de Cabezón. *Combinatorial Koszul homology: Computations and Applications.* PhD thesis, University of La Rioja, 2008.

[DLR07] C. Domínguez, L. Lambán, and J. Rubio. Object oriented institutions to specify symbolic computation systems. *Rairo - Theoretical Informatics and Applications*, 41, pp. 191–214, 2007.

[DMMV07] P. Dillinger, P. Manolios, J S. Moore, and D. Vroon. ACL2s: "The ACL2 Sedan". In *proceedings 29th International Conference on Software Engineering, Research Demonstration Track (ICSE'2007)*, pp. 59–60, 2007.

[Doe98] J. P. Doeraene. Homotopy pull backs, homotopy push outs and joins. *Bulletin of the Belgian Mathematical Society*, 5, pp. 15–37, 1998.

[DR11] C. Domínguez and J. Rubio. Effective Homology of Bicomplexes, formalized in Coq. *Theoretical Computer Science*, 412, pp. 962–970, 2011.

[DRSS98] X. Dousson, J. Rubio, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/.

[DST93] J. Davenport, Y. Siret, and E. Tournier. *Computer Algebra.* Academic Press, 1993.

[DSW05] M. Dewar, E. Smirnova, and S. M. Watt. XML in Mathematical Web Services. In *Proceedings of XML 2005 Conference Syntax to Semantics (XML'2005)*, 2005.

[Dus06] A. Duscher. Interaction patterns of Mathematical Services. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler Univeristy, Linz, 2006.

[DZ07] E. Domínguez and M. A. Zapata. Noesis: Towards a situational method engineering technique. *Information Systems*, 32(2), pp. 181–222, 2007.

[E$^+$07] B. Evjen et al. *Professional XML.* Wiley Publishing Inc., 2007.

[Ecl03] Eclipse platform technical overview, February 2003. `http://www.eclipse.org`.

[Ell09] G. Ellis. HAP package for GAP, 2009. `http://www.gap-system.org/Packages/hap.html`.

[EZ50] S. Eilenberg and J. A. Zilber. Semi-simplicial complexes and singular homology. *Annals of Mathematics*, 51(3), pp. 499–513, 1950.

[F$^+$08] S. Freundt et al. Symbolic Computation Sofware Composability. In *Proceedings 7th International Conference on Mathematical Knowledge Management (MKM'2008)*, volume 5144 of *Lectures Notes in Computer Science*, pp. 285–295. Springer-Verlag, 2008.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *Javaspaces. Principles, Patterns, and Practice.* Addison Wesley, 1999.

[FHK$^+$09] S. Freundt, P. Horn, A. Konovalov, S. Lindon, and D. Roozemond. Symbolic Computation Software Composability Protocol (SCSCP) specification, version 1.3, 2009. `http://www.symbolic-computation.org/scscp`.

[For] Formalisation of Mathematics. `wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.

[G$^+$] M. Gudgin et al. Soap version 1.2 specification. `http://www.w3.org/TR/soap12-part1/`.

[G$^+$08] D. A. Greve et al. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(01), pp. 15–46, 2008.

[GAP] GAP - Groups, Algorithms, Programming - System for Computational Discrete algebra. `http://www.gap-system.org`.

[GDMRSP05] R. González-Díaz, B. Medrano, P. Real, and J. Sánchez-Peláez. Algebraic Topological Analysis of Time-Sequence of Digital Images. In *Proceedings 8th International Conference on Computer Algebra in Scientific Computing (CASC'2005)*, volume 3718 of *Lecture Notes in Computer Science*, pp. 208–219, 2005.

[GDR05] R. González-Díaz and P. Real. On the Cohomology of 3D Digital Images. *Discrete Applied Math*, 147(2-3), pp. 245–263, 2005.

[Gel85] D. Gelenter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7, pp. 80–112, 1985.

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'2009)*, volume 5674 of *Lecture Notes in Computer Science*, pp. 327–342, 2009.

[GJ94] P. G. Goerss and J. F. Jardine. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GL$^+$09] M. J. González-López et al. TutorMates, 2009. `http://www.tutormates.es`.

[GM09] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[GPWZ02] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4), pp. 271–286, 2002.

[GR05] J. C. Giarratano and G. D. Riley. *Expert Systems: Principles and Programming.* PWS Publishing Company, 2005.

[Gra96] P. Graham. *ANSI Common Lisp.* Prentice Hall, 1996.

[Gro] Object Management Group. Common object request broker architecture (corba). `http://www.omg.org`.

[Gro09] The Object Management Group. UML specification version 2.2, 2009. `http://www.omg.org`.

[H$^+$00] D. Hyatt et al. XML User Interface Language (XUL) 1.0, 2000. `http://www.mozilla.org/projects/xul/`.

[Hac01] M. Hachimori. Simplicial complex library, 2001. `http://infoshako.sk.tsukuba.ac.jp/~hachi/math/library/index_eng.html`.

[Han06] M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. `http://www.curry-language.org`.

[Hat02] A. Hatcher. *Algebraic Topology.* Cambridge University Press, 2002. `http://www.math.cornell.edu/~hatcher/AT/ATpage.html`.

[Her03] L. J. Hernández. Orografía homológica cúbica para la minería de datos. Technical report, University of La Rioja, 2003. `http://www.unirioja.es/cu/luhernan/datamining`.

[Her09] J. Heras. The *fKenzo* program. University of La Rioja, 2009. `http://www.unirioja.es/cu/joheras`.

[Her10] J. Heras. Effective Homology of the Pushout of Simplicial Sets. In *Proceedings of the 12th Encuentro de Álgebra Computacional y Aplicaciones (EACA'10)*, pp. 152–156, 2010.

[Her11] J. Heras. `http://www.unirioja.es/cu/joheras/Thesis`, 2011.

[HK09] M. Hanus and C. Kluß. Declarative Programming of User Interfaces. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL'2009)*, volume 5418 of *Lectures Notes in Computer Science*, pp. 16–30. Springer-Verlag, 2009.

[HP10a] J. Heras and V. Pascual. ACL2 verification of Simplicial Complexes programs for the Kenzo system. In *Proceedings of the Algebraic computing, soft computing, and program verification workshop*, 2010.

[HP10b] J. Heras and V. Pascual. Mediated access to symbolic computation systems: An openmath approach. In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones científicas en honor de Mirian Andrés Gómez*, pp. 85–105. Universidad de La Rioja, 2010.

[HPDR11] J. Heras, M. Poza, M. Dénès, and L. Rideau. Incidence simplicial matrices formalized in Coq/SSReflect. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/PapersAndSlides`, 2011.

[HPR08a] J. Heras, V. Pascual, and J. Rubio. A graphical user interface for the Kenzo system, a program to compute in Algebraic Topology. In *Proceedings of the 11th Encuentro de Álgebra Computacional y Aplicaciones (EACA'08)*, pp. 93–96, 2008.

[HPR08b] J. Heras, V. Pascual, and J. Rubio. Mediated access to Symbolic Computation Systems. In *Proceedings of the 7th International Conference on Mathematical Knowledge Management (MKM'08)*, volume 5144 of *Lectures Notes in Artificial Intelligence*, pp. 446–461. Springer-Verlag, 2008.

[HPR09a] J. Heras, V. Pascual, and J. Rubio. A customizable GUI through an OMDoc documents repository. In *Proceedings of the 4th Mathematical User-Interfaces Workshop (MATHUI'09)*, 2009. `http://www.activemath.org/workshops/MathUI/09/`.

[HPR09b] J. Heras, V. Pascual, and J. Rubio. Applying Generative Communication to Symbolic Computation in Common Lisp. In *Proceedings of the 2nd European Lisp Symposium (ELS'09)*, pp. 27–42, 2009.

[HPR09c] J. Heras, V. Pascual, and J. Rubio. Content Dictionaries for Algebraic Topology. In *Proceedings of the 22nd OpenMath Workshop*, pp. 112–118, 2009.

[HPR09d] J. Heras, V. Pascual, and J. Rubio. Using Open Mathematical Documents to Interface Computer Algebra and Proof Assistant Systems. In *Proceedings of the 8th International Conference on Mathematical Knowledge Management (MKM'09)*, volume 5625 of *Lectures Notes in Artificial Intelligence*, pp. 467–473. Springer-Verlag, 2009.

[HPR11] J. Heras, V. Pascual, and J. Rubio. Proving with ACL2 the correctness of simplicial sets in the Kenzo system. In *Proceedings of the 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'2010)*, volume 6564 of *Lectures Notes in Computer Science*, pp. 37–51. Springer-Verlag, 2011.

[HPRR10] J. Heras, V. Pascual, A. Romero, and J. Rubio. Integrating multiple sources to answer questions in Algebraic Topology. In *Proceedings of the 9th International Conference on Mathematical Knowledge Management (MKM'10)*, volume 6167 of *Lectures Notes in Artificial Intelligence*, pp. 331–335. Springer-Verlag, 2010.

[HPRS08] J. Heras, V. Pascual, J. Rubio, and F. Sergeraert. Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology. In *Proceedings of the 1st European Lisp Symposium (ELS'08)*, pp. 155–176, 2008.

[HPRS11] J. Heras, V. Pascual, J. Rubio, and F. Sergeraert. *fKenzo*: A user interface for computations in Algebraic Topology. *To be published in Journal of Symbolic Computation*, 2011. DOI: 10.1016/j.jsc.2011.01.005.

[HT98] J. Harrison and L. Théry. A skeptic approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21, pp. 279–294, 1998.

[Hur35] W. Hurewicz. Beiträge zur Topologie der Deformationen I-II. *Proceedings of the Akademie van Wetenschappen*, 38, pp. 112–119, 521–528, 1935.

[Hur36] W. Hurewicz. Beiträge zur Topologie der Deformationen III-IV. *Proceedings of the Akademie van Wetenschappen*, 39, pp. 117–126, 215–224, 1936.

[Hut07] G. Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[HW67] P. J. Hilton and S. Wylie. *Homology Theory.* Cambridge University Press, 1967.

[IAM] Internet Accessible Mathematical Computation (IAMC). `http://icm.mcs.kent.edu/research/iamc.html`.

[Inca] Franz Inc. A SOAP 1.1 API for Allegro CL. `http://www.franz.com/support/documentation/current/doc/soap.htm`.

[Incb] Franz Inc. Allegro CL Socket Library. `http://www.franz.com/support/documentation/current/doc/socket.htm`.

[Incc] Franz Inc. Allegro Common Lisp. `http://www.franz.com/`.

[Incd] Franz Inc. jLinker - A Dynamic Link between Lisp and Java. `http://www.franz.com/support/documentation/current/doc/jlinker.htm`.

[Inc05] Franz Inc. Common Lisp for Service Oriented Architecture Programs. Technical report, Franz Inc., 2005. `http://www.franz.com/resources/educational_resources/white_papers/CL_for_SOA.pdf`.

[Jac95] P. Jackson. *Enhancing the Nuprl proof-development system and applying it to computational abstract algebra.* PhD thesis, Cornell University, 1995.

[K$^+$07] M. Kay et al. XSL transformations (XSLT) version 2.0, 2007. `http://www.w3.org/TR/xslt20/`.

[Kau00] M. Kaufmann. Modular proof: the fundamental theorem of calculus. In P. Manolios M. Kaufmann and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pp. 75–92. Kluwer Academic Publishers, 2000.

[KBN04] S. J. Koyani, R. W. Bailey, and J. R. Nall. *Research-Based Web Design and Usability Guidelines.* U.S. Dept. of Health and Human Services, 2004.

[KKL] V. Komendantsky, A. Konovalov, and S. Linton. Interfacing Coq + SS-Reflect with GAP. `http://www.cs.st-andrews.ac.uk/vk/Coq+GAP/`.

[KL09] A. Konovalov and S. Lindon. GAP package SCSCP, 2009. `http://www.gap-system.org/Packages/scscp.html`.

[KM] M. Kaufmann and J S. Moore. ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`.

[KM01] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2), pp. 161–203, 2001.

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An approach.* Kluwer Academic Publishers, 2000.

[KMM04] T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*, volume 157 of *Applied Mathematical Sciences.* Springer, 2004.

[Knu] Kevin P. Knudson. Amalgamated free products, unstable homotopy invariance, and the homology of $sl_2(z[t])$.

[Koh06] M. Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2].* Springer Verlag, 2006.

[KSN10] K. Kofler, P. Schodl, and A. Neumaier. Limitations in Content MathML. Technical report, University of Vienna, Austria, 2010. http://www.mat.univie.ac.at/~neum/FMathL/content-mathml-limitations.pdf.

[LBFL80] R. K. Lindsay, B. G. Buchanan, E. A. Feigenbaum, and J. Lederberg. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill Companies, Inc, 1980.

[Lip81] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Benjamin/Cummings Publishing Company, Inc., 1981.

[LMMRRR10] L. Lambán, F. J. Martín-Mateos, J. Rubio, and J. L. Ruíz-Reina. When first order is enough: the case of Simplicial Topology. Preprint. http://wiki.portal.chalmers.se/cse/uploads/ForMath/wfoe, 2010.

[LPR99] L. Lambán, V. Pascual, and J. Rubio. Specifying Implementations. In *Proceedings 12th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (ISSAC'1999)*, pp. 245–251, 1999.

[LPR03] L. Lambán, V. Pascual, and J. Rubio. An object-oriented interpretation of the EAT system. *Applicable Algebra in Engineering, Communication and Computing*, 14, pp. 187–215, 2003.

[Mac63] S. MacLane. *Homology*, volume 114 of *Grundleren der Mathematischen Wissenschaften.* Springer, 1963.

[Mac03] D. Mackenzie. Topologists and Roboticists Explore and Inchoate World. *Science*, 8, pp. 756, 2003.

[Mah67] M. E. Mahowald. *The Metastable Homotopy of S^n*, volume 72 of *Memoirs of the American Mathematical Society.* American Mathematical Society, 1967.

[MAP] MAP network. http://map.disi.unige.it/.

[Mata] MathBroker: A Framework for Brokering Distributed Mathematical Services. http://www.risc.uni-linz.ac.at/projects/basic/mathbroker/.

[Matb] MathBroker II: Brokering Distributed Mathematical Services. http://www.risc.uni-linz.ac.at/projects/mathbroker2/.

[Matc] MathServe Framework. http://www.ags.uni-sb.de/~jzimmer/mathserve.html.

[MATd] MATHWEB-SB: A Software Bus for MathWeb. http://www.ags.uni-sb.de/~jzimmer/mathweb-sb/.

[Mat76] M. Mather. Pull-Backs in Homotopy Theory. *Canadian Journal of Mathematics*, 28(2), pp. 225–263, 1976.

[Mau96] C.R.F. Maunder. *Algebraic Topology.* Dover, 1996.

[May67] J. P. May. *Simplicial objects in Algebraic Topology*, volume 11 of *Van Nostrand Mathematical Studies.* University of Chicago Press, 1967.

[McD82] J. McDermott. R1: A rule based configurer of computer systems. *Artificial Intelligence*, 19(1), 1982.

[Mic99] Sun Microsystems. JavaMail Guide for Service Providers, 1999. `http://www.oracle.com/technetwork/java/index-138643.html`.

[MKM] The Mathematical Knowledge Management Interest Group. `http://www.mkm-ig.org/`.

[MlBR07] E. Mata, P. Álvarez, J. A. Bañares, and J. Rubio. Formal Modelling of a Coordination System: from Practice to Theory and back again. In *Proceedings 7th Annual International Workshop Engineering Societies in the Agents World (ESAW'2006)*, volume 4457 of *Lectures Notes in Artificial Intelligence*, pp. 229–244. Springer-Verlag, 2007.

[MMAHRR02] F. J. Martín-Mateos, J. A. Alonso, M. J. Hidalgo, and J. L. Ruiz-Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. *Proceedings of the Third ACL2 workshop. Grenoble, Francia*, pp. 188–203, 2002.

[MMRRR09] F. J. Martín-Mateos, J. Rubio, and J. L. Ruiz-Reina. ACL2 verification of simplicial degeneracy programs in the kenzo system. In *Proceedings 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'2009)*, volume 5625 of *Lectures Notes in Computer Science*, pp. 106–121. Springer-Verlag, 2009.

[MMS03] J. Mayer, I. Melzer, and F. Schweiggert. Lightweight plug-in-based application development. In *Proceedings International Conference NetObjectDays (NODe'2002)*, volume 2591 of *Lecture Notes in Computer Science*, pp. 87–102. Springer-Verlag, 2003.

[Mon] MoNET: Mathematics on the Net. `http://monet.nag.co.uk/cocoon/monet/index.html`.

[Moz] Mozilla. An Introduction To Hacking Mozilla. `http://www.mozilla.org/hacking/coding-introduction.html`.

[MPRR10] I. Medina, F. Palomo, and J. L. Ruiz-Reina. A verified Common Lisp implementation of Buchberger's algorithm in ACL2. *Journal of Symbolic Computation*, 45(1), pp. 96–123, 2010.

[MW10] R. Mikhailov and J. Wu. On homotopy groups of the suspended classifying spaces. *Algebraic & Geometric Topology*, 10, pp. 565–625, 2010.

[Nie94] J. Nielsen. Heuristic evaluation. In J. Nielsen and R. L. Mack, editors, *Usability Inspection Methods.* John Wiley & Sons, New York, NY, 1994.

[NSS59] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pp. 256–264, 1959.

[OS03] D. Orden and F. Santos. Asymptotically efficient triangulations of the d-cube. *Discrete and Computational Geometry*, 30(4), pp. 509–528, 2003.

[OSG03] OSGi Service Platform, Release 3, March 2003. `http://www.osgi.org`.

[P^{+}02] K. Polthier et al. JavaView - Interactive 3D Geometry and Visualization. In *Multimedia Tools for Communicating Mathematics*, 2002. `http://www.javaview.de/index.html`.

[Pau96] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[Pec08] A. Peck. *Beginning GIMP: From Novice to Professional.* Apress, 2008. `http://gimpbook.com/`.

[Poi95] H. Poincaré. Analysis Situs. *Journal de l'École Polytechnique*, 1, pp. 1–121, 1895.

[Rav86] D. C. Ravenel. *Complex cobordism and stable homotopy groups of spheres.* Academic Press, 1986.

[Rea94] P. Real. Sur le calcul des groupes d'homotopie. *C. R. Acad. Sci. Paris Sér. I Math.*, 319(5), pp. 475–478, 1994.

[RER09] A. Romero, G. Ellis, and J. Rubio. Interoperating between computer algebra systems: computing homology of groups with kenzo and gap. In *Proceedings 34th International Symposium on Symbolic and Algebraic Computation (ISSAC'2009)*, pp. 303–310, 2009.

[Rey98] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4), pp. 363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[Rom10] A. Romero. Effective homology and discrete morse theory for the computation of homology of groups. In *Mathematics Algorithms and Proofs*, 2010. `http://www.unirioja.es/dptos/dmc/MAP2010/Slides/Slides/talkRomeroMAP2010.pdf`.

[RRS06] A. Romero, J. Rubio, and F. Sergeraert. Computing spectral sequences. *Journal of Symbolic Computation*, 41(10), pp. 1059–1079, 2006.

[RS97] J. Rubio and F. Sergeraert. Constructive Algebraic Topology, Lecture Notes Summer School on Fundamental Algebraic Topology. Institut Fourier, Grenoble, 1997. `http://www-fourier.ujf-grenoble.fr/~sergerar/Summer-School/`.

[RS02] J. Rubio and F. Sergeraert. Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques*, 126(5), pp. 389–412, 2002.

[RS06] J. Rubio and F. Sergeraert. Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs. University of Genova, 2006. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Genova-Lecture-Notes.pdf`.

[RSS90] J. Rubio, F. Sergeraert, and Y. Siret. EAT: Symbolic Software for Effective Homology Computation. Institut Fourier, Grenoble, 1990. `ftp://fourier.ujf-grenoble.fr/pub/EAT`.

[Rus92] D. Russinoff. A mechanical prof of quadratic reciprocity. *Journal of of Automated Reasoning*, 8(1), pp. 3–21, 1992.

[Rus98] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1, pp. 148–200, 1998.

[SC09] A. Solomon and M. Costantini. GAP package OpenMath, 2009. `http://www.gap-system.org/Packages/openmath.html`.

[Sch97] F. B. Schneider. *On Concurrent Programming*. Springer, 1997.

[Sch09] W. Schelter. Maxima v. 5.18.1, 2009. `http://www.maxima.sourceforge.net/index.shtml`.

[Ser80] J. P. Serre. *Trees*. Springer, 1980.

[Ser87] F. Sergeraert. Homologie effective. *Comptes Rendus des Séances de l'Academie des Sciences de Paris*, 304(11 and 12), pp. 279–282 and 319–321, 1987.

[Ser90] F. Sergeraert. Functional coding and effective homology. *Astérisque*, 192, pp. 57–67, 1990.

[Ser92] F. Sergeraert. Effective homology, a survey. Technical report, Institut Fourier, 1992. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Survey.pdf`.

[Ser94] F. Sergeraert. The computability problem in Algebraic Topology. *Advances in Mathematics*, 104(1), pp. 1–29, 1994.

[Ser01] F. Sergeraert. Common Lisp, Typing and Mathematics. Technical report, University of La Rioja, 2001. `http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Ezcaray.pdf`.

[Ser10] F. Sergeraert. Triangulations of complex projective spaces. In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones científicas en honor de Mirian Andrés Gómez*, pp. 507–519. Universidad de La Rioja, 2010.

[SGF03] F. Ségonne, E. Grimson, and B. Fischl. Topological Correction of Subcortical Segmentation. In *Proceedings 6th International Conference on Medical Image Computing and Computer Assisted Intervention (MICCAI'2003)*, volume 2879 of *Lecture Notes in Computer Science*, pp. 695–702, 2003.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of Association for Computing Machinery*, 35(3), pp. 475–522, 1988.

[Sha94] N. Shankar. *Metamathematics, Machines, and Godel's Proof.* Cambridge University Press, 1994.

[Shi62] W. Shih. Homologie des espaces fibrés. *Publications mathématiques de l'Institut des Hautes Études Scientifiques*, 13, pp. 1–88, 1962.

[Slu07] T. A. Sluga. *Modern C++ Implementation of the LINDA coordination language.* PhD thesis, University of Hannover, 2007.

[SSW04] E. Smirnova, C. M. So, and S. M. Watt. An architecture for distributed mathematical web services. In *Proceedings 3rd International Conference on Mathematical Knowledge Management (MKM'2004)*, volume 3119 of *Lecture Notes in Computer Science*, pp. 363–377, 2004.

[Sta81] R. Stallman. Emacs: The extensible, customizable display editor. In *ACM Conference on Text Processing*, 1981.

[Ste] W. Stein. SAGE mathematical software system. `http://www.sagemath.org`.

[SvdW10] B. Spitters and E. van der Weegen. Developing the Algebraic Hierarchy with Type Classes in Coq. In *Proceedings International Conference on Interactive Theorem Proving (ITP'2010)*, volume 6172 of *Lecture Notes in Computer Science*, pp. 490–493, 2010.

[Tec] Wavefront Technologies. Object files (.obj). `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/`.

[Tod62] H. Toda. *Compositional methods in homotopy groups of spheres.* Princeton University Press, 1962.

[Tur36] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings London Mathematical Society*, 42, pp. 230–265, 1936.

[Veb31] O. Veblen. *Analysis Situs.* AMS Coll. Publ., 1931.

[Wey80] R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence Journal*, 13(1), pp. 133–170, 1980.

[Whi78] G. W. Whitehead. *Elements of Homotopy Theory*, volume 61 of *Graduate texts in Mathematics.* Springer-Verlag, 1978.

[Woo89] J. Wood. Spinor groups and algebraic coding theory. *Journal of Combinatorial Theory*, 50, pp. 277–313, 1989.

Gestión mecanizada del conocimiento matemático en Topología Algebraica

Jónathan Heras Vicente

Memoria presentada para la
obtención del grado de Doctor

Directores: Dra. Dña. Vico Pascual Martínez-Losa
Dr. D. Julio Rubio García

Universidad de La Rioja
Departamento de Matemáticas y Computación

Logroño, marzo 2011

Este trabajo ha sido parcialmente subvencionado por una beca FPI de la Comunidad Autónoma de La Rioja, los proyectos MTM2009-13842-C02-01 y MTM2010-09087-E del Ministerio de Educación y Ciencia, el proyecto Europeo ForMath n^o 243847 del 7th Framework Programme, el proyecto API08-08, el contrato COLABORA 2007/16 y las becas ATUR10/18, ATUR09/14, ABOV10-51, ABOV09-50 de la Universidad de La Rioja.

Agradecimientos

A Vico y Julio
A mis padres

Por medio de estas líneas quiero dar las gracias a mis directores, Vico Pascual y Julio Rubio; sin ellos está claro que este trabajo no hubiera sido posible. Gracias por haberme escuchado siempre que lo he necesitado, por vuestras brillantes ideas, vuestra comprensión, paciencia y confianza. No hay palabras suficientes para expresar toda mi gratitud.

También quiero agradecer su apoyo a todos los miembros de grupo de investigación al que pertenezco, Jesús María Aransay, César Domínguez, Francisco García, Arturo Jaime, Laureano Lambán, Eloy Mata, Juan José Olarte, Beatriz Pérez, María Poza, Ángel Luis Rubio y Eduardo Sáenz de Cabezón, que desde el primer momento me trataron como a uno más. Debo destacar a Ana Romero por sus ánimos en los momentos difíciles. No puedo olvidarme de Mirian Andrés, compañera y amiga que se nos fue demasiado pronto y sólo pudo conocer los comienzos de este trabajo.

Gracias también al Departamento de Matemáticas y Computación de la Universidad de La Rioja, por proporcionarme todos los medios necesarios, y a todos sus miembros, por el trato amable que siempre he recibido de todos ellos. Quiero dar las gracias en especial a Luis Javier Hernández y Mª Teresa Rivas, que me ayudaron a dar los primeros pasos en este mundo de la investigación, y también a Clara Jiménez, por aguantarme durante todo este tiempo.

Por último, quisiera agradecer el cariño recibido de familiares y amigos, que siempre han confiando en mí, incluso en los momentos en los que yo no lo hacía. Y un agradecimiento especial a mis padres, entre otras cosas, por no dejar que me hundiera en los momentos difíciles.

A todos vosotros, gracias.

Índice General

Introducción

El interés por la mecanización de las Matemáticas apareció en los inicios de la Informática, o incluso antes: léase bajo esta perspectiva el trabajo de Alan Turing [Tur36], uno de los fundadores de las Ciencias de la computación. Dejando de lado las aplicaciones científicas del cálculo numérico (que dieron literalmente origen a los primeros ordenadores), desde 1950 se comenzó a desarrollar el cálculo simbólico, que aspira a reproducir con la asistencia de ordenadores el modo de trabajo de los matemáticos. El cálculo simbólico tiene dos ramas: el álgebra computacional y el razonamiento mecanizado basado en la lógica computacional. Para la historia inicial del álgebra computacional puede consultarse el texto de referencia [DST93]. Sobre el razonamiento mecanizado y los demostradores automatizados de teoremas se puede citar el influyente GPS (General Problem Solver) de Newell y Simon [NSS59], cuyas primeras versiones aparecieron hacia 1955, vinculado al nacimiento de la Inteligencia Artificial.

Estas dos líneas de la manipulación simbólica (el álgebra computacional, o cálculo simbólico, y el razonamiento mecanizado) conocieron en los siguientes 40 años desarrollos paralelos, sin mucha relación entre ambas. Sin embargo, el panorama cambió radicalmente hace unos 15 años, cuando la iniciativa de investigación Calculemus [Cal], financiada por proyectos europeos, planteó el desafío de integrar los sistemas de álgebra computacional (cálculo) con los de razonamiento mecanizado (deducción). En la misma línea la red de investigación MAP (Mathematics, Algorithms and Proofs) [MAP], creada en 2003, trabaja en los fundamentos de esa integración, basándose sobre todo en la *matemática constructiva.*

Además, concurrentemente en el tiempo, con la eclosión de Internet, aparecen múltiples propuestas para hacer posible las Matemáticas (simbólicas) en la red. Tal vez la contribución más sólida sea la que se articula a través de la iniciativa IAMC (Internet Accessible Mathematical Computation) [IAM]. A partir de ahí, se han propuesto distintas arquitecturas (como “buses” [CH97], [MATd], “brokers” [Mata], [Matb], servidores genéricos [Matc], o basadas en servicios web [DSW05], [SSW04]) y patrones de diseño [Dus06], que han desembocado en el paradigma de las “Matemáticas en Red” MoNET [Mon] y Matemáticas en y para la red semántica [CDDP04]. Instrumental para todas estas contribuciones ha sido la especificación de los estándares XML llamados MathML [A^+08] y OpenMath [Con04], que proporcionan los protocolos de comunicación para la transmisión del conocimiento matemático. Una iniciativa de éxito, pero que es independiente de estos estándares es SAGE [Ste]; una alternativa a los paradigmas que ahora son dominantes.

De la confluencia de estas líneas (la que pretende integrar cálculo y deducción, y la que propugna unas Matemáticas en red) surge en 2001 la comunidad dedicada al Mathematical Knowledge Management (Gestión del Conocimiento Matemático) MKM [MKM]. Su objetivo final es el desarrollo de asistentes integrales para las Matemáticas, que incorporen cálculo, deducción, acceso en red e interfaces de usuario potentes que puedan mejorar el trabajo cotidiano de los investigadores en Matemáticas, así como influir en las estrategias de enseñanza de las Matemáticas. Se trata de un objetivo sumamente ambicioso, que retoma alguno de los temas de la Inteligencia Artificial clásica.

Tras esta introducción general al campo en el que se enmarca esta tesis, vayamos a nuestra área de aplicación particular: la Topología Algebraica. Es ésta una rama de las Matemáticas que se ocupa de la distinción de espacios según su forma intrínseca (y no según su tamaño u otras características extrínsecas), por medio de unos invariantes algebraicos (grupos o anillos, usualmente).

A pesar de ser una rama abstracta de las Matemáticas, los métodos de la Topología Algebraica pueden ser implementados en sistemas software y ser aplicados a distintos contextos como la teoría de códigos [Woo89], la robótica [Mac03] o el análisis de imágenes digitales [GDMRSP05, GDR05, SGF03].

Podemos decir que el presente trabajo pretende particularizar el MKM al campo de la Topología Algebraica.

En este contexto jugará un papel fundamental el sistema Kenzo [DRSS98], un programa de álgebra computacional dedicado a la investigación en Topología Algebraica. Dicho sistema ha sido desarrollado principalmente por Francis Sergeraert y está programado en el lenguaje Common Lisp [Gra96]. Este sistema implementa el método de la Homología Efectiva que apareció en los años 80 tratando de proporcionar algoritmos reales para el cálculo de grupos de homología y homotopía (dos de los invariantes más importantes en Topología Algebraica). Esta técnica fue introducida por Francis Sergeraert en [Ser87] y [Ser94], y su estado actual queda descrito en [RS97] y [RS06].

Desde el año 2000 el trabajo del grupo de Programación y Cálculo Simbólico de la Universidad de La Rioja, dirigido por Julio Rubio, puede decirse que ha sido, en cierto modo, paralelo al que se ha esbozado anteriormente para la disciplina en general. Tras una primera fase en la que los esfuerzos se dedicaron a la mejora y ampliación de algoritmos y programas de Kenzo, se inició una nueva línea de investigación (sin abandonar la anterior) en la que se pretendía aplicar los Métodos Formales de la Ingeniería del Software al sistema Kenzo. Fruto de esta investigación se obtuvieron resultados relevantes en cuanto a la especificación algebraica de Kenzo y, más en general, de sistemas software orientados a objetos (véanse, a título de ejemplo, [LPR03] y [DLR07]). Profundizando aún más en esta línea, se utilizó el asistente para la demostración Isabelle para certificar la corrección de uno de los algoritmos centrales en Kenzo, el Lema Básico de Perturbación [ABR08]. En esta misma línea encontramos la formalización de la Homología Efectiva de los Bicomplejos en el asistente para la demostración Coq [DR11] o la demostración del Teorema de Normalización en el sistema ACL2 [LMMRRR10]. En cuanto a la Topología Algebraica en la red, una primera propuesta fue publicada en [APRR05], donde el acceso remoto (pero parcial) a Kenzo fue conseguido utilizando la tecnología CORBA [Gro].

En esta memoria las tres líneas de investigación presentadas (*desarrollo de algoritmos*, *formalización con herramientas de ayuda a la demostración* e *implementación de interfaces de usuario* para la Topología Algebraica) convergen. En concreto, el objetivo de este trabajo ha consistido en el desarrollo de un asistente integral para la investigación en Topología Algebraica. El adjetivo "integral" se refiere a que el asistente proporciona no solo una interfaz para utilizar el motor de cálculo Kenzo, sino también guía al usuario y, en la medida de lo posible, da certificados de corrección de los cálculos realizados. A su vez, distintos subobjetivos han sido logrados. En primer lugar, hemos intentado incrementar la usabilidad y accesibilidad del sistema Kenzo, aumentando de este modo el número de usuarios que se puede beneficiar de Kenzo. Además, se han incrementado las capacidades computacionales de Kenzo por medio de nuevos módulos para Kenzo y de la conexión con el sistema de cálculo simbólico GAP [GAP]. Asimismo, el objetivo de integrar cálculo (Kenzo) y deducción (ACL2 [KM]) ha sido logrado en este trabajo gracias a la formalización de los programas de Kenzo en el demostrador de teoremas ACL2.

La organización de la memoria es la siguiente. El primer capítulo incluye algunas nociones y resultados preliminares que serán usados en el resto del trabajo. En la primera sección se presentan las nociones fundamentales sobre Algebra Homológica, Topología Simplicial y Homología Efectiva que serán utilizadas en el resto del trabajo. La segunda sección introduce el sistema de Álgebra Computacional Kenzo así como algunas de sus características más importantes. Por último el sistema ACL2 es presentado brevemente.

Después de este primer capítulo, la memoria se divide en tres partes diferentes. Los capítulos 2 y 3 están dedicados a incrementar la accesibilidad y usabilidad del sistema Kenzo. La capacidad computacional del sistema Kenzo es aumentada en los capítulos 4 y 5 mediante la integración con otros sistemas de cálculo y el desarrollo de nuevos programas verificados en ACL2 para Kenzo. Finalmente, el capítulo 6 está totalmente dedicado a probar la corrección de algunos programas del sistema Kenzo mediante ACL2.

El capítulo 2 presenta un entorno, llamado Kenzo framework, que proporciona un acceso mediado al sistema Kenzo restringiendo su funcionalidad pero guiando al usuario en su interacción con el sistema. Se describen en este capítulo la arquitectura, las componentes y el flujo de ejecución del Kenzo framework. Además, al final de este capítulo, se presentan dos líneas de trabajo futuro para incrementar las capacidades computacionales del sistema Kenzo mediante ideas sobre el balanceo de cálculos y computación distribuida.

Una vez que hemos presentado el Kenzo framework, queremos ser capaces de incrementar sus capacidades ya sea a través de la inclusión de nuevos módulos de Kenzo o mediante la integración de otras herramientas como sistemas de cálculo simbólico o demostradores de teoremas, pero sin modificar el código fuente del núcleo. Además, nos dimos cuenta de que la interfaz del Kenzo framework, basada en XML, no es usable para un usuario humano, por lo que un modo más agradable de interactuar con el sistema tenía que ser proporcionado. El capítulo 3 presenta como se han resuelto estos dos objetivos. La primera parte de este capítulo se centra en presentar un framework de plug-ins que permite extender el Kenzo framework. En la segunda parte se presenta una interfaz de usuario adaptable que permite usar el Kenzo framework de manera fácil, incrementando de este modo la usabilidad y accesibilidad del sistema. El sistema completo, es decir, la combinación de los dos frameworks y la interfaz de usuario, se llama *fKenzo* (del inglés friendly Kenzo). Los dos capítulos siguientes se centran en extender *fKenzo* por medio de la integración de nuevos sistemas y nuevos programas para Kenzo.

El capítulo 4 describe la integración de varios sistemas en *fKenzo*. En primer lugar, el sistema de cálculo simbólico GAP [GAP] es incorporado al sistema permitiendo el cálculo de homología de grupos. Además, el sistema GAP no sólo se utiliza de manera individual, sino que algunos de sus programas son combinados con el sistema Kenzo para obtener nuevas herramientas, que vuelven a aumentar las capacidades de cálculo de *fKenzo*. La segunda parte de este capítulo presenta la integración del demostrador de teoremas ACL2 en *fKenzo*, consiguiendo de este modo la integración de cálculo y deducción dentro de un mismo sistema. Del mismo modo que para el caso de GAP, el sistema ACL2 es combinado con Kenzo para incrementar la fiabilidad de los programas usados en nuestro sistema. A partir de este capítulo el sistema ACL2 será utilizado de manera intensiva.

El capítulo 5 realiza una aportación a los algoritmos y programas del sistema Kenzo extendiendo a su vez el sistema *fKenzo*. En primer lugar se presenta un nuevo módulo para Kenzo que permite trabajar con complejos simpliciales. Este módulo sobre complejos simpliciales es utilizado como base para desarrollar un entorno para el análisis de imágenes digitales monocromáticas. En particular, un nuevo módulo para analizar imágenes digitales es presentado en la segunda parte de este capítulo. Por último, se desarrollan los algoritmos y programas necesarios para la construcción de la homología efectiva del pushout de conjuntos simpliciales. Además, este capítulo no está sólo dedicado a presentar los nuevos módulos para el sistema Kenzo sino también a la verificación de la corrección de dichos programas en ACL2.

El capítulo 6 está dedicado a probar la corrección de alguno de los programas implementados en el sistema Kenzo por medio de ACL2. En la primera parte de este capítulo se describen una serie de herramientas del sistema ACL2 que serán utilizadas a lo largo de dicho capítulo. La segunda parte está centrada en verificar los constructores de conjuntos simpliciales constantes del sistema Kenzo en ACL2. A continuación se presenta una manera de modelar estructuras algebraicas en ACL2. Por último se introduce una metodología para demostrar la corrección de los programas Kenzo utilizados para construir espacios a partir de otros por medio de constructores topológicos.

La memoria termina con un capítulo que contiene las conclusiones y posibles líneas de trabajo futuro, un glosario de términos y finalmente incluimos la bibliografía.

Resumen de los capítulos

Presentamos a continuación un breve resumen de cada uno de los capítulos de esta memoria.

1 Preliminares

En el primer capítulo incluimos las definiciones y resultados básicos que serán utilizados en el resto de la memoria. Este capítulo está dividido en tres secciones. La primera está dedicada a presentar las nociones fundamentales sobre Algebra Homológica, Topología Simplicial y Homología Efectiva que serán utilizadas en el resto del trabajo. Las referencias fundamentales para cada una de esas subsecciones son respectivamente [Mac63], [May67] y [RS06]. En la segunda sección se introduce el sistema de Álgebra Computacional Kenzo [DRSS98] así como algunas de sus características más importantes. Finalmente, presentamos el demostrador de teoremas ACL2. La referencia principal para esta última parte del capítulo es el libro [KMM00b].

2 Un framework para realizar cálculos en Topología Algebraica

El capítulo 2 presenta un entorno, llamado Kenzo framework, que proporciona un acceso mediado al sistema Kenzo, restringiendo la funcionalidad de Kenzo, pero guiando al usuario en su interacción con el sistema y también introduciendo ciertas mejoras en Kenzo. Para diseñar este framework hemos utilizado metodologías y patrones que habían sido utilizados de manera exitosa anteriormente.

La organización del capítulo es la siguiente. En primer lugar se presentan la arquitectura, las componentes y el flujo de ejecución del Kenzo framework. Al final de este capítulo se presentan dos posibles líneas de trabajo futuro para incrementar la capacidad de cálculo del Kenzo framework mediante ideas sobre balanceo de cálculos y computación distribuida.

3 Extensibilidad y Usabilidad del Kenzo framework

El tercer capítulo de la memoria se centra en dos de los aspectos más importantes en el desarrollo de un sistema software: la extensibilidad y la usabilidad. Para abordar el reto de la extensibilidad, desarrollamos un framework de plug-ins que nos permite añadir nueva funcionalidad al Kenzo framework de dos modos distintos: (1) mediante la integración de nueva funcionalidad para el sistema Kenzo o (2) mediante la conexión con otros sistemas tales como sistemas de cálculo simbólico o asistentes para la demostración. Para incrementar la usabilidad del Kenzo framework una interfaz de usuario amigable es presentada en este capítulo. Dicha interfaz es personalizable gracias al framework de plug-ins. El sistema completo (es decir, la combinación de los dos frameworks y la interfaz de usuario) ha sido llamado *fKenzo*.

El capítulo se divide en dos partes. La primera está dedicada al framework de plug-ins. En esta primera sección se presenta la arquitectura del framework de plug-ins y el modo general de incrementar la funcionalidad del Kenzo framework a través de este sistema. La segunda parte se centra en presentar la interfaz de usuario cubriendo las perspectivas tanto de un usuario como de un desarrollador.

4 Interoperabilidad para la Topología Algebraica

Este capítulo está dedicado a explicar la integración de diferentes herramientas, tales como sistemas de cálculo simbólico y demostradores de teoremas, en el Kenzo framework. Como sistema de cálculo simbólico hemos elegido el sistema GAP, un sistema conocido por sus contribuciones en el área de la Teoría de Grupos Computacional, y como demostrador de teoremas el sistema que integramos fue ACL2. La integración de estos sistemas en nuestro framework no es el objetivo final, sino que es un medio para lograr un entorno que combine los distintos sistemas para crear herramientas potentes y fiables.

El capítulo está dividido en dos partes, la primera dedicada a la integración del sistema de cálculo simbólico GAP y la segunda a la integración de ACL2. La estructura de ambas es similar, con una breve introducción, la integración de dichos sistemas en el Kenzo framework, el modo en el que se extiende la interfaz gráfica de *fKenzo* para incluir la nueva funcionalidad y por último la combinación de los sistemas con la funcionalidad ya disponible para obtener nuevas herramientas.

5 Nuevos módulos certificados de Kenzo

El quinto capítulo de la memoria está dedicado a incrementar la funcionalidad del sistema Kenzo mediante programas certificados en ACL2, así como la inclusión de dichos programas en *fKenzo*. En concreto, tres nuevos módulos para Kenzo son presentados.

El primero de ellos implementa una noción básica en Topología Algebraica que son los complejos simpliciales. A partir del módulo de complejos simpliciales se desarrollan una serie de algoritmos que permiten el estudio de imágenes digitales monocromáticas. La implementación de dichos algoritmos ha dado lugar a nueva funcionalidad para el sistema Kenzo. Por último los algoritmos que definen la homología efectiva del pushout de conjuntos simpliciales, una de las construcciones más utilizadas en Topología Algebraica, son desarrollados e implementados.

El capítulo está dividido en tres partes, la primera dedicada a los complejos simpliciales, la segunda al análisis de imágenes digitales y la tercera a la construcción de la Homología Efectiva del pushout de conjuntos simpliciales. La estructura de todas ellas es similar, con una breve introducción, la implementación de los programas como nuevos módulos para Kenzo, la integración de dichos programas en el Kenzo framework, el modo en el que se extiende la interfaz gráfica de *fKenzo* para incluir la nueva funcionalidad y finalmente la verificación de los programas en ACL2.

6 Integración de Cálculo y Deducción

En el último capítulo de la memoria utilizamos el demostrador de teoremas ACL2 para verificar que los programas utilizados en Kenzo para construir diferentes tipos de espacios son correctos.

Este capítulo está dividido en cuatro partes. La primera está centrada en presentar tres herramientas de ACL2 que serán utilizadas en el resto del capítulo. La segunda parte contiene una descripción de una infraestructura implementada en ACL2 que nos permite probar que los programas de Kenzo dedicados a la construcción de conjuntos simpliciales constantes realmente construyen conjuntos simpliciales. A continuación, se introduce una metodología para modelar las estructuras matemáticas en ACL2. Finalmente, se presenta en la última parte, una metodología en ACL2 para probar que los espacios de Kenzo construidos a partir de otros por medio de la aplicación de constructores topológicos son correctos.

Conclusiones y trabajo futuro

Conclusiones

Este trabajo debería ser entendido como un primer paso hacia un asistente integral para la investigación y la enseñanza en Topología Algebraica. Desde nuestro punto de vista, las principales contribuciones logradas en este trabajo, organizadas por capítulos, son las siguientes.

- Capítulo 2:
  - Hemos mostrado cómo se puede guiar a un usuario en el uso de herramientas para la Topología Algebraica Computacional, sin recurrir a las técnicas estándares de la Inteligencia Artificial. La idea ha consistido en desarrollar una infraestructura que proporciona un *acceso mediado* a un sistema de cálculo simbólico desarrollado previamente. Juntando Kenzo y esta infraestructura, hemos producido el Kenzo framework que se puede conectar a diferentes clientes (interfaces de escritorio, aplicaciones web, etcetera). En general, esto supone una restricción a las capacidades del sistema base, pero la interacción con dicho sistema se facilita y enriquece, contribuyendo de este modo a incrementar el número de usuarios del sistema. Este trabajo ha sido presentado en [HPRS08, HPR08b].
  - Hemos incluido las siguientes capacidades en el Kenzo framework: (1) control de la especificación de entrada de los constructores, (2) se evitan operaciones sobre objetos que darían lugar a errores, (3) se encadenan métodos para proporcionar nuevas herramientas al usuario, y (4) se usa conocimiento experto para obtener resultados que no estaban disponibles en el sistema Kenzo. Este trabajo ha sido publicado en [HP10b].
  - Como sub-producto, hemos escrito Diccionarios de Contenido OpenMath dedicados a la Topología Algebraica Simplicial. Concretamente, para cada una de las estructuras matemáticas disponibles en Kenzo hemos definido un Diccionario de Contenido. Las definiciones dadas en estos Diccionarios de Contenidos incluyen partes axiomáticas que han sido utilizadas para interactuar con el sistema de deducción ACL2. Este trabajo puede encontrarse en [HPR09c].

- ◦ Hemos desarrollado una primera versión de un sistema con el mismo objetivo que el Kenzo framework pero que permite procesar peticiones de forma concurrente y donde los resultados son almacenados de forma persistente. Este trabajo ha sido presentado en [HPR09b]

- Capítulo 3:
  - ◦ Hemos implementado un framework de plug-ins que permite añadir nueva funcionalidad al Kenzo framework sin modificar el código fuente. Este framework de plug-ins nos ha permitido incrementar las capacidades del Kenzo framework mediante la integración de nueva funcionalidad para el sistema Kenzo o la conexión con otros sistemas tales como sistemas de cálculo simbólico o asistentes para la demostración.
  - ◦ Debido a que los usuarios finales del Kenzo framework son estudiantes, profesores o investigadores de Topología Algebraica que normalmente no conocen el sistema Common Lisp, hemos desarrollado una interfaz de usuario amigable que permite la interacción con el Kenzo framework, sin necesidad de preocuparse por las cuestiones técnicas relacionadas con Lisp que son inevitables cuando se usa el sistema Kenzo. Este trabajo ha sido presentado en [HPR08a, HPR09a].
  - ◦ Varios tipos de interacción con nuestro sistema son posibles dependiendo del tipo de usuario, por lo tanto, hemos usado el framework de plug-ins para ser capaces de personalizar nuestra interfaz gráfica dependiendo de sus necesidades. Este trabajo puede encontrarse en [HPR09d].
  - ◦ Hemos reunido los dos frameworks (Kenzo framework y framework de plug-ins) y la interfaz gráfica de usuario en un único sistema llamado *fKenzo*. Este trabajo ha sido publicado en [HPRS11].

- Capítulo 4:
  - ◦ Hemos logrado la integración, en un mismo sistema, de cálculo (gracias a Kenzo y GAP) y deducción (por medio de ACL2). Incluso si nuestra propuesta tiene una extensión limitada, tanto desde el punto de vista temático como desde los sistemas base, creemos que hemos mostrado una línea de investigación lo suficientemente sólida para ser exportada a otras áreas de la gestión mecanizada del conocimiento matemático. Este trabajo ha sido presentado en [HPRR10].
  - ◦ No sólo hemos logrado la integración de varias herramientas en el mismo sistema, sino que las hemos hecho cooperar (un problema mucho más difícil). Por un lado, hemos automatizado la integración entre Kenzo y GAP que fue propuesta en [RER09] ocultando los detalles técnicos de la integración de estos sistemas al usuario final. Por otro lado, Kenzo, GAP y ACL2 han trabajado de manera conjunta en nuestro sistema para proporcionar una herramienta potente y fiable relacionada con la construcción de espacios de Eilenberg MacLane de tipo $K(G, n)$ donde G es un grupo cíclico; dichos espacios son

fundamentales a la hora de calcular grupos de homotopía. Este trabajo ha sido presentado en [HPRR10].

- Capítulo 5:

  - Hemos presentado un programa que incrementa la funcionalidad del sistema Kenzo, por medio de la generación de complejos simpliciales a partir de sus elementos maximales y de la construcción de los conjuntos simpliciales asociados con complejos simpliciales. Además, hemos certificado la corrección de dicho programa en ACL2, mejorando de este modo la fiabilidad del nuevo módulo de Kenzo. Este trabajo puede encontrarse en [HP10a].
  - Hemos desarrollado un nuevo módulo Kenzo dedicado a estudiar imágenes monocromáticas 2D y 3D que está basado en el módulo de los complejos simpliciales. Asimismo, hemos formalizado nuestros algoritmos en ACL2; de este modo, si usamos nuestros sistemas en problemas de la vida real (por ejemplo, el estudio de imágenes médicas) podemos estar completamente seguros de que los resultados producidos por nuestros programas son siempre correctos.
  - Hemos presentado un algoritmo para construir la homología efectiva del pushout de conjuntos simpliciales con homología efectiva. Como subproducto el wedge y el join de conjuntos simpliciales pueden ser construidos como casos particulares del pushout. Los algoritmos han sido implementados como un nuevo módulo para Kenzo que ha sido probado con algunos casos de estudio interesantes como el espacio proyectivo $P^2(\mathbb{C})$ ó $SL_2(\mathbb{Z})$. Este trabajo ha sido presentado en [Her10].
  - Hemos integrado los nuevos módulos presentados en este capítulo en *fKenzo*.

- Capítulo 6:

  - Hemos diseñado un marco para probar la corrección de los conjuntos simpliciales implementados en el sistema Kenzo. Como ejemplos de aplicación hemos proporcionado la prueba completa de la corrección de la implementación en Kenzo de las esferas, los conjuntos simpliciales estándar y los conjuntos simpliciales asociados con complejos simpliciales (módulo una transformación segura entre los programas Kenzo y la sintaxis ACL2). Por medio de la misma teoría genérica la corrección de otros conjuntos simpliciales de Kenzo puede ser demostrada. Este trabajo ha sido publicado en [HPR11].
  - Hemos proporcionado una metodología para modelar estructuras matemáticas en ACL2. Este proceso de modelado ha sido utilizado para implementar parcialmente una jerarquía algebraica en ACL2 que es la base para verificar la corrección de la construcción de espacios a partir de otros, y que puede ser útil para trabajos no relacionados con nuestros desarrollos.
  - Hemos presentado un marco para probar la corrección de la construcción de espacios a partir de otros a partir de su implementación en Kenzo. Como ejemplos de aplicación, hemos dado, entre otros, una prueba completa de

la corrección de la implementación de la suma directa de dos complejos de cadenas, del Lema de Perturbación Fácil y del Teorema SES_1. Usando el mismo marco, la corrección de otros constructores puede ser probada.

En resumen, podemos afirmar que hemos desarrollado un asistente para la Topología Algebraica llamado *fKenzo*. Este sistema no sólo proporciona una interfaz de usuario amigable para utilizar Kenzo, sino que además guía al usuario en su interacción con el sistema. Además, este sistema puede evolucionar al mismo tiempo que Kenzo. Esta característica ha sido probada mediante el desarrollo de varios módulos para el sistema Kenzo que han sido integrados en *fKenzo* sin mayor dificultad. Asimismo, la funcionalidad del sistema puede también ser extendida mediante la integración con sistemas de cálculo simbólico o demostradores de teoremas, que no sólo trabajan de manera individual, sino que cooperan para producir nuevas herramientas. Finalmente, podemos decir que hemos desarrollado una herramienta fiable ya que su núcleo ha sido parcialmente verificado mediante el sistema ACL2.

Problemas abiertos y trabajo futuro

La investigación presentada en esta memoria se puede continuar en varias direcciones, las agrupamos por capítulos:

- Capítulo 2:
  - El módulo HES (Párrafo 2.2.3.2.3) debería ser dotado con un lenguaje de alto nivel para describir reglas de modo que un investigador de Topología Algebraica sin ningún conocimiento específico sobre ordenadores pudiera introducir nuevas reglas.
  - El algoritmo implementado en el módulo HAM (Párrafo 2.2.3.2.2) puede ser completado gracias al desarrollo presentado en [RER09] que permite el cálculo de la homología efectiva de espacios de Eilenberg MacLane de tipo $K(G, n)$ donde G es un grupo cíclico. De este modo, todos los grupos de homotopía de conjuntos simpliciales 1-reducidos con homología efectiva podrían ser calculados.
  - Uno de los temas más importantes que debe ser abordado en las nuevas versiones del Kenzo framework es la organización de cuándo (y cómo) un cálculo debe ser enviado a un servidor remoto. Algunas ideas sobre este tema fueron presentadas en la Sección 2.4.
  - La arquitectura presentada en la Sección 2.5 nos permitirá trabajar en un contexto distribuido. Estamos pensando en una arquitectura, donde varios clientes se comunica con un servidor central. Dicho servidor actúa como repositorio general y proporciona potencia de cálculo para tratar con los cálculos complejos. El mayor problema consiste en diseñar buenas heurísticas para decidir cuál es el significado de “ser un cálculo difícil”.

- Capítulo 3:
    - Una de las posibles mejoras para *fKenzo* consistiría en encontrar un modo cómodo (sin los detalles técnicos de la sintaxis Common Lisp) de editar y manipular los elementos de los espacios construidos. De este modo podríamos tratar la difícil cuestión de introducir en *fKenzo* cálculos como los grupos de homología de $P^2(\mathbb{C})$, ver subsubsección 5.3.4.
    - En el área de interfaces de usuarios, un objetivo posible sería el desarrollo de una interfaz de usuario web. En este momento la estructura de dicha interfaz ya es proporcionada mediante la especificación XUL, el trabajo ahora consistiría en conectar dicha interfaz con el Kenzo framework.
    - También en el área de las interfaces de usuario, se podría integrar un editor de ecuaciones, del estilo a DragMath [Bil], en la interfaz de usuario de *fKenzo*. El editor podría ser usado para escribir expresiones matemáticas en el estilo tradicional, y producir expresiones OpenMath. Por último, las expresiones OpenMath serán usadas para invocar al Kenzo framework.
- Capítulo 4:
    - Más capacidades de cálculo y deducción deberían ser integradas en nuestro sistema. Desde el punto de vista de los sistemas de cálculo simbólico podemos considerar, por ejemplo, el sistema Cocoa [CoC], que ya ha sido utilizado en trabajos relacionados con Kenzo, véase [dC08]. Desde el punto de vista de los demostradores de teoremas podemos considerar Coq e Isabelle, ya utilizados para la formalización de distintos algoritmos de Kenzo, véase [ABR08, DR11]. Además, como en el caso de GAP y ACL2, el verdadero interés no reside en usar los sistemas de forma individual, sino en hacerlos cooperar para obtener nuevas herramientas y resultados que no podrían alcanzarse trabajando por separado.
    - Sería necesario mejorar la interacción con el sistema ACL2. En este momento las peticiones deben ser preprocesadas (incluso cuando la especificación paramétrica permite al sistema cubrir familias completas de estructuras: grupos cíclicos, esferas, complejos simpliciales, etcetera). Se debería proporcionar un modo cómodo de introducir preguntas sobre la veracidad de propiedades de objetos intermedios, generadas de manera dinámica durante una sesión de cálculo.
    - Queremos integrar más certificados ACL2 en nuestro sistema. La idea sería interactuar como hasta ahora en la misma interfaz con Kenzo y ACL2. Por ejemplo, el usuario podría dar información sobre cómo construir un conjunto simplicial en Kenzo. Además, él podría proporcionar ciertas pistas a ACL2 explicando el motivo de que esta construcción sea correcta; entonces ACL2 produciría una prueba completa de la corrección de la construcción. Este tipo de interacción entre sistemas de cálculo simbólico y demostradores de teoremas sería muy valiosa, sin embargo las dificultades relacionadas con encontrar una representación común todavía no han sido resueltas.

    - Un meta-lenguaje debería ser diseñado para especificar cómo y cuándo un nuevo sistema puede ser integrado en el framework. Esta capacidad y la necesidad de organizar los diferentes servicios supone un reto, que puede ser explorado mediante la tecnología OpenMath.

- Capítulo 5:

    - Una línea natural de investigación consiste en desarrollar nuevos algoritmos y programas. Por ejemplo, una tarea que supone un desafío es la implementación de la noción dual de pushout llamada pullback [Mat76].
    - En la subsubsección 5.1.5.2, hemos presentado dos programas diferentes que permiten construir el complejo simplicial asociado con una lista de símplices. Uno de estos programas fue implementado únicamente en Common Lisp debido a que usaba una estrategia de memoización que no podía ser directamente implementada en ACL2. Sin embargo, el trabajo presentado en [BH06] proporciona las herramientas necesarias para memoizar funciones en ACL2, por lo que una versión optimizada de nuestro algoritmo puede ser implementada en ACL2. Además, se podría utilizar la adscripción semántica presentada en la subsección 6.1.3 para utilizar la versión no optimizada para razonar y la nueva versión optimizada para ejecutar.
    - Del mismo modo que se han aplicado los algoritmos relacionados con complejos simpliciales en el estudio de imágenes digitales, se pueden desarrollar nuevos programas para ser aplicados en distintos contextos como la teoría de códigos o la robótica. Del mismo modo que en el caso de las imágenes digitales, si queremos utilizar nuestros programas en aplicaciones de la vida real, debemos estar seguros de que los resultados que producen son correctos. Por lo tanto, la verificación formal de nuestros programas mediante un demostrador de teoremas es importante.

- Capítulo 6:

    - Con la experiencia adquirida, la metodología presentada en la sección 6.2 puede ser utilizada en otras estructuras algebraicas de Kenzo. Por lo tanto, el trabajo presentado en dicha sección puede ser considerado una buena base para nuestro objetivo de verificar en ACL2 los fragmentos de primer orden del sistema Kenzo.
    - La jerarquía presentada en la subsección 6.3.4 debería ser enriquecida por medio del resto de estructuras matemáticas implementadas en Kenzo, ver la subsección 1.2.1.
    - La metodología presentada en la sección 6.4 puede ser aplicada en distintos casos. En particular, sería interesante formalizar la construcción de la homología efectiva del pushout que involucra varias de las construcciones ya presentadas en esta memoria.
    - La formalización del cálculo de grupos de homología queda como trabajo futuro. En este caso sólo nos deberíamos centrar en el cálculo de grupos

de homología de espacios de tipo finito gracias al método de la Homología Efectiva. En estos casos, el cálculo de cada grupo de homología se traduce a un problema de diagonalizar ciertas matrices de enteros. Por lo tanto, deberíamos formalizar este proceso de diagonalización.

Todos estos problemas son interesantes para ser abordados, sin embargo no podemos atacarlos todos a la vez. Por lo tanto, nuestras principales prioridades son las siguientes:

- *Formalizar librerías para el Álgebra Homológica y la Topología Algebraica.* A lo largo de esta memoria hemos presentado la formalización de varios resultados relacionados con el Álgebra Homológica y la Topología Algebraica por medio del sistema ACL2. En el futuro planeamos la formalización de más resultados. Para llevar a cabo esta tarea, no sólo vamos a utilizar el sistema ACL2, sino que también tenemos pensado usar el sistema Coq [BGBP08] así como su extensión SSReflect [GM09]. Un primer paso en esta dirección es el trabajo [HPDR11], donde se presenta una formalización en Coq de las matrices de incidencia asociadas a conjuntos simpliciales así como el principal teorema que da sentido a la definición de los grupos de homología.

- *Integración de diferentes demostradores de teoremas.* Como hemos explicado en el punto anterior, estamos planeando el uso simultáneo de dos demostradores de teoremas: ACL2 y Coq. Este objetivo se puede partir en dos tareas diferentes. Por un lado, queremos comparar distintas formalizaciones del mismo problema en Coq/SSReflect; un trabajo relacionado con esta línea de investigación puede verse en [AD10]. Por otra parte, estamos interesados en utilizar ACL2 como oráculo de esquemas inductivos para Coq/SSReflect, obteniendo de este modo una interoperabilidad, similar a la presentada para los sistemas de cálculo simbólico Kenzo y GAP.

- *Aplicaciones al estudio de imágenes médicas.* Los métodos presentados en esta memoria relacionados con imágenes digitales pueden ser aplicados en el estudio de imágenes médicas. Sin embargo, para lograr este objetivo es necesario implementar nuevas herramientas en sistemas software. Además, si queremos aplicar nuestros métodos en el estudio de imágenes médicas debemos estar completamente seguros de que estos son correctos, por lo tanto es también necesario un proceso de formalización.

La razón principal para escoger estos objetivos y no otros, es debido a que ellos corresponden con tres de las tareas asignadas al nodo de La Rioja del proyecto ForMath [For] que ha financiado parcialmente el trabajo presentado en esta memoria.

Publicaciones

Incluimos a continuación los resúmenes de las publicaciones a las que ha dado lugar el trabajo presentado en esta memoria.

- **J. Heras, V. Pascual, J. Rubio y F. Sergeraert. Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology. En Actas del 1st European Lisp Symposium (ELS'08), pp. 155–176. Universidad de Bourdeaux, 2008.**
  Kenzo es un sistema de cálculo simbólico dedicado a la Topología Algebraica. Escrito en Common Lisp, este programa ha sido exitoso en el cálculo de grupos de homología y homotopía hasta entonces desconocidos. El objetivo ahora consiste en incrementar el número de usuarios y mejorar la usabilidad del sistema. En lugar de diseñar simplemente una interfaz de usuario, decidimos abordar el desarrollo de un *acceso mediado* al sistema, restringiendo su funcionalidad, pero proporcionando orientación al usuario en la interacción con el sistema. Este objetivo ha sido alcanzado mediante la construcción en Common Lisp de una *capa intermedia*, permitiendo *un acceso inteligente* a algunas funciones del sistema. Esta capa intermedia esta sostenida por la tecnología XML y sirve de enlace entre la interfaz gráfica de usuario y el sistema Kenzo *puro*.

- **J. Heras, V. Pascual y J. Rubio. A graphical user interface for the Kenzo system, a program to compute in Algebraic Topology. En Actas del XI Encuentro de Álgebra Computacional y Aplicaciones (EACA'08), pp. 93–96. Universidad de Granada, 2008.**
  Kenzo es un sistema de cálculo simbólico dedicado a la Topología Algebraica. Fue desarrollado por F. Sergeraert principalmente como una herramienta de investigación. El objetivo ahora consiste en incrementar su número de usuarios y mejorar su usabilidad. La primera tarea llevada a cabo ha sido el diseño de una interfaz de usuario sencilla, después de esta tarea nos hemos centrado en el desarrollo de un marco completo que envuelve el sistema Common Lisp Kenzo *puro* y hace el uso de Kenzo más sencillo para diferentes clientes.

- **J. Heras, V. Pascual y J. Rubio. Mediated access to Symbolic Computation Systems. En Actas del 7th International Conference on Mathematical Knowledge Management (MKM'08). Lectures Notes in Artificial Intelligence 5144, pp. 446–461, 2008.**

Kenzo es un sistema de cálculo simbólico dedicado a la Topología Algebraica. Fue desarrollado por F. Sergeraert principalmente como una herramienta de investigación. El objetivo ahora consiste en incrementar su número de usuarios y mejorar su usabilidad. En lugar de diseñar simplemente una interfaz de usuario, decidimos abordar el desarrollo de un *acceso mediado* al sistema, restringiendo su funcionalidad, pero proporcionando orientación al usuario en la interacción con el sistema. Este objetivo ha sido alcanzado mediante la construcción en Common Lisp de una *capa intermedia*, permitiendo *un acceso inteligente* a algunas funciones del sistema. Esta capa intermedia esta sostenida por la tecnología XML y sirve de interacción entre la interfaz gráfica de usuario y el sistema Kenzo *puro*.

- **J. Heras, V. Pascual y J. Rubio. Applying Generative Communication to Symbolic Computation in Common Lisp. En Actas del 2nd European Lisp Symposium (ELS'09), pp. 27–42. Universidad de Milán, 2009.**
  En este artículo, se describe una arquitectura para interactuar con un sistema llamado Kenzo. Kenzo es un sistema Common Lisp dedicado a la computación simbólica en Matemáticas, concretamente en el área de la Topología Algebraica. La arquitectura presentada en este artículo es una evolución de una propuesta previa, donde varios aspectos han sido mejorados. En particular, se han desacoplado los diferentes componentes y se han implementado mecanismos para reusar los cálculos intermedios. La nueva tecnología que nos permite estas mejoras está basada en el Modelo Linda (un modelo donde los procesos se comunican mediante un espacio de tuplas compartido), implementado a través de AllegroCache (una base de datos orientada a objetos de Allegro Common Lisp). El marco desarrollado actúa como mediador entre algunos clientes de Kenzo y el propio Kenzo.

- **J. Heras, V. Pascual y J. Rubio. A customizable GUI through an OMDoc documents repository. En Actas del 4th Mathematical User-Interfaces Workshop, 2009.**
  `http://www.activemath.org/workshops/MathUI/09/`
  Kenzo es un sistema de cálculo simbólico dedicado a la Topología Algebraica. En algunos trabajos previos se presentó un framework que envolvía a Kenzo proporcionando un acceso mediado a dicho sistema haciendo su uso más fácil. En este trabajo, se presenta un cliente de este framework, concretamente una interfaz gráfica de usuario. Por medio de un repositorio de documentos OMDoc, esta interfaz es totalmente personalizable. Además, permite a Kenzo interoperar con otros sistemas, concretamente con el demostrador de teoremas ACL2. De este modo, representación, cálculo y deducción son integrados en un mismo sistema.

- **J. Heras, V. Pascual y J. Rubio. Content Dictionaries for Algebraic Topology. En Actas del 22nd OpenMath Workshop, pp. 112–118, 2009.**
  Kenzo es un sistema de cálculo simbólico dedicado a la Topología Algebraica que trabaja con las principales estructuras matemáticas usadas en esta disciplina. En este artículo, presentamos Diccionarios de Contenido OpenMath para cada una de las estructuras matemáticas con las que trabaja el sistema Kenzo. Además, se

explica el modo en que se usan para interactuar con un demostrador de teoremas y para obtener cálculos certificados de Kenzo.

- **J. Heras, V. Pascual y J. Rubio. Using Open Mathematical Documents to Interface Computer Algebra and Proof Assistant Systems. En Actas del 8th International Conference on Mathematical Knowledge Management (MKM'09). Lectures Notes in Artificial Intelligence 5625, pp. 467–473, 2009.**
  El conocimiento matemático puede ser codificado por medio de documentos matemáticos abiertos (cuyas siglas en inglés son OMDoc) para interactuar tanto con sistemas de cálculo simbólico como con sistemas de ayuda a la demostración. En este artículo, mostramos como una única estructura de OMDoc puede ser usada para generar dinámicamente tanto una interfaz gráfica de usuario para un sistema de cálculo simbólico como un fichero para un asistente a la demostración. Así que, el formato OMDoc puede ser usado para representar diferentes aspectos. Esta aproximación genérica ha sido puesta en práctica por medio de un primer prototipo que conecta el sistema de cálculo simbólico Kenzo y el demostrador de teoremas ACL2, ambos basados en el lenguaje de programación Common Lisp. Un repositorio de documentos OMDoc ha sido desarrollado permitiendo al usuario la personalización de la aplicación de una manera sencilla.

- **J. Heras, V. Pascual. Mediated Access To Symbolic Computation Systems: An OpenMath Approach. Contribuciones Científicas En Honor De Mirian Andrés Gómez, pp. 85–105, 2010.**
  En este artículo, se describe una arquitectura para interactuar con el sistema de Cálculo Simbólico Kenzo (dedicado a la Topología Algebraica). El acceso que se proporciona esta mediado por medio de varios artefactos basados en XML, produciendo una capa intermedia inteligente. La parte más externa de la arquitectura ha sido creada usando Diccionarios de Contenido OpenMath y el correspondiente Phrasebook. Como consecuencia, varios Diccionarios de Contenido OpenMath para la Topología Algebraica Simplicial han sido desarrollados.

- **J. Heras, V. Pascual. ACL2 verification of Simplicial Complexes programs for the Kenzo system. En Actas del Algebraic computing, soft computing, and program verification workshop. Intl Center for Mathematical Meetings (CIEM), Castro Urdiales, Spain, 2010.**
  Kenzo es un sistema de álgebra computacional dedicado a la Topología Algebraica, escrito en el lenguaje de programación Common Lisp. A pesar de ser más sencilla la noción de complejo simplicial que la de conjunto simplicial, el sistema Kenzo sólo incluye la segunda. En este artículo, se presentan los programas que permiten trabajar en Kenzo con los complejos simpliciales, además de proporcionar una demostración automatizada de su corrección. La demostración es llevada a cabo por medio de ACL2, un sistema para probar propiedades de programas escritos en (un subconjunto de) Common Lisp.

- **J. Heras. Effective Homology of the Pushout of Simplicial Sets. En Actas del XII Encuentro de Álgebra Computacional y Aplicaciones (EACA'2010), pp. 152–156. Universidad de Santiago de Compostela, 2010.**
  En este artículo, el algoritmo que construye la versión con homología efectiva del pushout de los morfismos simpliciales $f : X \to Y$ y $g : X \to Z$, donde X, Y y Z son conjuntos simpliciales con homología efectiva es presentado.

- **J. Heras, V. Pascual, A. Romero y J. Rubio. Integrating multiple sources to answer questions in Algebraic Topology. En Actas del 9th International Conference on Mathematical Knowledge Management (MKM'2010). Lectures Notes in Artificial Intelligence 6167, pp. 331–335, 2010.**
  Presentamos en este artículo la evolución de una herramienta desde una interfaz de usuario para un sistema de cálculo simbólico dedicado a la Topología Algebraica (el sistema Kenzo), hasta un front-end que permite la interoperabilidad entre diferentes fuentes de información para el cálculo y la deducción. La arquitectura permite al sistema no sólo interactuar con varios sistemas, sino también hacer que cooperen entre ellos para obtener cálculos de manera distribuida.

- **J. Heras, V. Pascual y J. Rubio. Proving with ACL2 the correctness of simplicial sets in the Kenzo system. En Actas del 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'2010). Lectures Notes in Computer Science 6564, pp. 37–51, 2011.**
  Kenzo es un sistema Common Lisp dedicado a la Topología Algebraica. En este artículo mostramos cómo el demostrador de teoremas ACL2 puede ser usado para probar la corrección de fragmentos de primer orden de Kenzo. De forma más concreta, nosotros presentamos la verificación en ACL2 de la implementación de los conjuntos simpliciales. Por medio de un mecanismo de instanciación genérica, nosotros conseguimos reducir el esfuerzo de la demostración para cada una de las familias de conjuntos simpliciales, dejando a ACL2 automatizar las partes rutinarias de la demostración.

- **J. Heras, V. Pascual, J. Rubio y F. Sergeraert. *fKenzo*: a user interface for computations in Algebraic Topology. Será publicado en Journal of Symbolic Computation.**
  fKenzo (del inglés friendly Kenzo) es una interfaz gráfica de usuario que proporciona una interfaz amigable al sistema Kenzo, un programa Common Lisp dedicado a la Topología Algebraica. El sistema *fKenzo* proporciona la interfaz de usuario, una capa intermedia basada en XML y, finalmente el núcleo Kenzo. Describimos en este artículo las principales propiedades del programa *fKenzo*, y también explicamos las ventajas y limitaciones de *fKenzo* con respecto a Kenzo.