An ACL2 infrastructure to formalize Kenzo
Higher-Order constructors*

Jónathan Heras and Vico Pascual

Departamento de Matemáticas y Computación, Universidad de La Rioja, Edificio Vives, Luis de Ullea s/n, E-26004 Logroño (La Rioja, Spain).
{jonathan.heras, vico.pascual}@unirioja.es

Abstract. Kenzo is a Computer Algebra system devoted to Algebraic Topology. In this paper we present an infrastructure to formalize the construction of Kenzo spaces from other ones in the ACL2 Theorem Prover. In order to evaluate how practical our framework is, we present three case studies: the construction of the direct sum of chain complexes, the Easy Perturbation Lemma and the $S.E.S.$ Theorem. As a necessary tool we have encoded a hierarchy of algebraic structures in ACL2. The algebraic hierarchy is formalized in ACL2 by applying a combination of records and macros.

1 Introduction

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 three possible development tracks of a Computer Algebra system (CAS) and a Theorem Prover (TP) interface: (1) use a CAS as a hint engine for a TP, (2) use a CAS as a proof engine for a TP, and (3) prove in the TP the correctness of CAS 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 [13] or the communication between Coq and GAP [20]. The last track allows us to build more reliable and accurate components for the CAS, for instance Buchberger’s algorithm for computing Gröbner basis (one of the most important algorithms in Computer Algebra) has been formalized in [23] using the ACL2 theorem prover. In this paper, we have focussed on the third track using the Kenzo system as CAS and ACL2 [18] as TP.

Kenzo [10] is a Computer Algebra system devoted to Algebraic Topology and Homological Algebra. This Common Lisp system implements Sergeraert’s ideas on effective homology [27] and has been successful in the sense that it has obtained some results that have not been confirmed nor refuted by any other

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Community FP7, STREP project ForMath.
One feature of Kenzo is the 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, both Isabelle/HOL and Coq proof assistant were used to verify important algorithms in Homological Algebra: the Basic Perturbation Lemma in Isabelle/HOL [2], the Effective Homology of Bicomplexes in Coq [9], or the Easy Perturbation Lemma proved both in Isabelle/HOL and Coq [3]. However, these formalizations were related to algorithms and not to the real programs implemented in Kenzo. A completely different approach has focused on the verification of actual first order Kenzo fragments using the ACL2 theorem prover, see [1, 22, 15]. The key point here is that both Kenzo and ACL2 are Common Lisp systems, then Kenzo programs can be certified in the ACL2 theorem prover.

As we commented in the previous paragraph, the Kenzo system involves both first and high order programming, to provide a better understanding of this point let us explain the general Kenzo way of working. First of all, the user constructs some initial spaces (spheres, Moore spaces and so on) from scratch by means of some built-in Kenzo functions; afterwards, she constructs new spaces from other ones by applying topological constructors (loop spaces, cartesian products, and so on); finally, the user asks Kenzo for computing the homology groups of the spaces. The first step can be modeled in a first order logic as was explained in [15], where an infrastructure allowing to prove the correctness of Kenzo programs for constructing that constant spaces was developed. However, steps 2 and 3 involve higher order functional programming.

This paper is devoted to explain an ACL2 framework 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 [5]. The feasibility of using our framework is illustrated with three examples. As a by-product, the foundations for the development in ACL2 of an algebraic hierarchy has been laid.

The rest of the paper is organized as follows. A hierarchy of algebraic structures developed in ACL2 is introduced in Section 2, this hierarchy is the foundation for the rest of the developments of this paper. Section 3 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 (Subsections 3.1 and 3.2) and subsequently explaining the general case (Subsection 3.3). The methodology is evaluated with two case studies: the Easy Perturbation Lemma (Subsection 4.1) and the $SES_1$ Theorem (Subsection 4.2). The gap between the verified ACL2 code and the actual Kenzo code is described in Section 5. The paper ends with a section of Conclusions and Further Work, and the bibliography.
2 A hierarchy of algebraic structures 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 [12] based on dependent records and used in the proof of the fundamental theorem of algebra, the SSReflect hierarchy [11] 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 [29]; other examples can be found in systems such as Nuprl [16] and Lego [4]. Nevertheless, the development of an ACL2 algebraic hierarchy had not been undertaken until now. 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 High Order Theorem Provers based on type theory; on the contrary our hierarchy is written in the untyped first order logic theorem prover ACL2.

We have defined a number of structures in ACL2 representing basic algebraic structures. Namely, we have defined the structures: magma, semigroup, monoid, group, abelian group and ring. Each algebraic structure is defined in terms of the previous one. Moreover, some of the main mathematical structures used in the Kenzo system, such as morphisms, chain complexes, reductions and so on; are also introduced (the mathematical definition of these structures will be introduced in the following sections). In this case, we do not have a proper hierarchy, but these structures are defined in terms of the others; for instance, the notion of reduction is defined from the notions of chain complex, chain complex morphism and morphism. In Figure 2 we show our hierarchy. A continue arrow describes an heritage relation and a dashed arrow means a use relationship.

Mathematical structures can be determined by means of their operations and this is the idea followed to codify algebraic structures in our hierarchy.

![Diagram of algebraic structure hierarchy](image_url)

**Fig. 1.** Algebraic structure hierarchy
particular, a record with as many functional slots as operations has the represented structure allows us to store enough information about the structure, and therefore is a suitable pattern to its codification.

To implement the structures in ACL2 we have used the `defstructure` macro [6]. This macro is used in ACL2 to define record structures. We will not present the definitions of all the structures in detail, but we urge the interested reader to consult the complete development in [14]. Although a group is a very elementary structure it will allow us to explain the essence of our codification, the rest of structures are defined in an analogous way. A group is encoded by the ACL2 record:

```
(defstructure group inv binary-op id-elem inverse)
```

This command must be read as follows: `defstructure` is the name of the macro, `group` is the name of the structure, `inv`, `binary-op`, `id-elem`, and `inverse` are four functional slots which are used to determine the group. The function `inv` is the characteristic function of the underline set of the group, `binary-op` is the binary operation, `id-elem` is the identity element and `inverse` is the inverse function. Then, to construct the unit group we define the concrete functions `inv-unit`, `op-unit`, `id-elem-unit`, and `inverse-unit` and use them to create an instance of the group record by means of the instruction `make-group`:

```
(defun inv-unit (a) (equal a 0))
(defun op-unit (a b) (declare (ignore a b)) 0)
(defun id-elem-unit () 0)
(defun inverse-unit (x) (declare (ignore x)) 0)
(make-group :inv 'inv-unit :binary-op 'op-unit :id-elem 'id-elem-unit :inverse 'inverse-unit)
```

The slots of an instance of the group structure are accessed by the reader functions `group-inv`, `group-binary-op`, `group-id-elem` and `group-inverse`.

Once, we have defined the structure, we need to state the properties about such structure. Let us remark that ACL2 is an untyped logic, it uses type information internally to deduce types. In ACL2, we provide the prover with type information by specifying type hypotheses on variables in a conjecture. Although ACL2 is syntactically untyped, that does not prevent users from having and using a notion of a type. One cannot create new types in ACL2, in the sense that one cannot create a new non-empty set of values that provably extends the ACL2 value universe; divided into 5 kinds of data objects: Numbers, Characters, Strings, Symbols and Conses (Ordered Pairs). Rather, one typically partitions the existing universe in potentially new ways to form “new” sets. These sets (“types”) are presently characterized by just a type predicate.

Taking into account the previous paragraph, we have defined type predicates by means of macros that are used to formalize the “type” of the structures. The defining property for groups is the macro `(check-group-p group name)`, where
group is an instance of the group structure and name is a symbol. The macro is an abbreviation for:

(defthm name-is-a-group
  (group-conditions group))

where group-conditions is the function:

(defun group-conditions (group)
  (and (monoid-conditions (make-monoid :inv (group-inv group)
                                         :binary-op (group-binary-op group)
                                         :id-elem (group-id-elem group)))
       (implies ((group-inv group) x)
                 ((group-inv group) ((group-inverse group) x)))
       (implies ((group-inv group) x)
                 (equal ((group-binary-op group) x ((group-inverse group) x))
                        ((group-id-elem group))))
       (implies ((group-inv group) x)
                 (equal ((group-binary-op group) ((group-inverse group) x) x)
                        ((group-id-elem group))))))

In this manner, groups are defined in terms of monoids, see that the above code invokes the function monoid-conditions which is in charge of verifying the properties of monoids. The heritage relation between groups and monoids is provided in this way. When the macro (check-group-p group name) is invoked, a proof try is launched. If the proof try succeeds, a new theorem called |name-is-a-group| (name is replaced by the name of the group) is admitted in ACL2 stating the properties of groups for that record instance. If the proof try fails that means either that the instance is not a group or that it is necessary to help ACL2 with some auxiliary lemma to guide the proof. Now, we can check that our definition of the unit group is correct. When we evaluate in the read-eval-print loop of ACL2 the command:

(check-group-p (make-group :inv 'inv-unit :binary-op 'op-unit
                            :id-elem 'id-elem-unit :inverse 'inverse-unit)
               'unit)

a proof try is automatically generated. In this case ACL2 succeeds in the first attempt and the theorem called |UNIT-is-a-group| is admitted. |UNIT-is-a-group| can be expanded and the properties of groups are shown.
As we said previously, a hierarchy of structures has been defined. The general scheme of defining an algebraic structure \( B \) in terms of an algebraic structure \( A \) consists of two steps: firstly, define a record \( B \) with all the slots of \( A \) and the slots only related to \( B \):

\[
\text{(defstructure A} \\
\text{A-slot-1} \\
\text{...} \\
\text{A-slot-n})
\]

\[
\text{(defstructure B} \\
\text{A-slot-1} \\
\text{...} \\
\text{A-slot-n} \\
\text{B-slot-1} \\
\text{...} \\
\text{B-slot-m})
\]

and secondly, define a macro \text{check-B-p} that is an abbreviation of a theorem, which checks the properties of the structure \( B \):

\[
\text{(defthm name-is-a-B} \\
\text{(B-conditions B))}
\]

where \( B\text{-conditions} \) is defined as:

\[
\text{(defun B-conditions (B} \\
\text{(and (A-conditions (make-A :A-slot-1 (A-slot-1 B) \\
\text{...} \\
\text{:A-slot-n (A-slot-n B)})) \\
\text{(ax_prop1)} \\
\text{...} \\
\text{(ax_propn))})}
\]

This definition uses the function \( A\text{-conditions} \) to state the properties of the \( A \) structure, and \text{ax_prop1} \ldots \text{ax_propn} to state the additional properties of \( B \).

In this way our algebraic hierarchy is introduced in the ACL2 Theorem prover, and is ready to be used in our proofs of the construction of topological spaces from other ones as we explain in the next section.
3 Formalization of the Kenzo constructors in ACL2

This section is devoted to present our methodology to formalize in ACL2 the Kenzo constructors of spaces from other ones. 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.

3.1 Mathematical Preliminaries

In this subsection we define the basic notions needed in our formalization of the direct sum of chain complexes. We assume as known the notions of ring, module over a ring and module morphism (see [17] for instance).

Definition 1 Given a ring \(R\), a graded module \(M\) is a family of left \(R\)-modules \((M_n)_{n \in \mathbb{Z}}\).

Definition 2 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 \(f_n : M_n \rightarrow M'_{n+k}\) for all \(n \in \mathbb{Z}\).

Definition 3 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 is defined as follows.

Definition 4 A chain complex is a family of pairs \((M_n, d_n)_{n \in \mathbb{Z}}\) where \((M_n)_{n \in \mathbb{Z}}\) is a graded module and \((d_n)_{n \in \mathbb{Z}}\) is a differential.

Definition 5 Given a pair of chain complexes \((M_n, d_n)_{n \in \mathbb{Z}}\) and \((M'_n, d'_n)_{n \in \mathbb{Z}}\), a chain complex morphism between them is a family of module morphisms \((f_n)_{n \in \mathbb{Z}}\) of degree 0 between \((M_n)_{n \in \mathbb{Z}}\) and \((M'_n)_{n \in \mathbb{Z}}\) such that \(d'_n \circ f_n = f_{n-1} \circ d_n\) for all \(n \in \mathbb{Z}\).

Based on the previous definitions, the notion of direct sum of two chain complexes can be introduced.

Definition 6 Given two chain complexes \(A = (A_n, da_n)_{n \in \mathbb{Z}}, B = (B_n, db_n)_{n \in \mathbb{Z}}\) the direct sum of \(A\) and \(B\) is the chain complex \(A \oplus B = (M_n, d_n)_{n \in \mathbb{Z}}\) such that, \(M_n = (A_n, B_n)\) and \(d_n = (da_n, db_n)\) for all \(n \in \mathbb{Z}\).

The notion of direct sum of chain complexes is implemented in the Kenzo system, so if we want to prove the correctness of that implementation, we need a proof of the following theorem.
**Theorem 1** Let the Kenzo objects $A$ and $B$ that are chain complex instances, then the implementation of the Kenzo direct sum constructor really codifies the direct sum of chain complexes, that is, it fulfills the properties about the direct sum of chain complexes stated in Definition 6.

The next subsection is devoted to present the proof of Theorem 1 using the ACL2 theorem prover.

### 3.2 The correctness of the Kenzo direct sum constructor

The first step to translate Theorem 1 to the ACL2 context is the codification of the chain complex structure. 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 codified in the way explained in Section 2, 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 `make-chain-complex`. Besides, the macro `check-chain-complex-p` verifies the correctness of the chain complex record instances. When we invoke this macro with a chain complex instance as input, a proof try is generated to check the properties of chain complexes for that instance.

Then, we can define concrete chain complex instances and verify its correctness, but Theorem 1 is valid for two generic chain complexes. To codify generic chain complexes in ACL2, we use the `encapsulate` tool [19], a mechanism that allows us to introduce function symbols by axioms constraining them to have certain properties. Thus, a representation of two generic chain complex instances can be given in ACL2 in the following way. First of all, we start the definition of the `encapsulate`.

```
(encapsulate

Afterwards, the signatures of the functions which define the chain complexes are provided. In this case, only the signatures of two of the operators which determine the chain complex structure, namely `inv` (the characteristic function of the underlying graded set 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 complex $A$ and $B$ respectively.

```

```

Subsequently, a witness for each one of the functions is given in order to avoid the introduction of inconsistencies in ACL2.
Finally, we gather the functions in two instances of the chain-complex record, constructed with the instruction make-chain-complex. Besides, in order to verify that both chain complex instances really determine chain complexes (i.e. the functions satisfy the properties of chain complexes) we use the macro check-chain-complex-p.

(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 effect of the above encapsulate is an extension of the ACL2 universe in which the functions that determine the two chain complexes have the syntactical signatures given and are axiomatized to satisfy the properties of chain complexes.

So, we have declared two generic chain complex instances. 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.

To this aim, we have inspired on the notion of functor (also called parametric modules) implemented in Coq [7]. A similar notion can be found in Paulson’s book on ML [24]. 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, from now on called functor functions, for the direct sum which are determined 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 direct sum constructor and now we want to prove that this constructor really implements the direct sum of two chain complexes. To this purpose, we need to check that the direct sum constructor is a chain complex with the macro check-chain-complex-p.
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|.

Finally, we need to prove that our definition of the direct sum satisfies \( M_n = (A_n, B_n) \) and \( d_n = (da_n, db_n) \) for all \( n \in \mathbb{Z} \), but this can be considered 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 constructor.

```lisp
(check-chain-complex-p (make-chain-complex :inv-cc 'inv-ds ... :diff-cc 'diff-ds)
'direct-sum-a-b)
```

Then, we have a proof of Theorem 1 in ACL2. This result can be reused for every pair of concrete chain complexes by means of the derived rule of inference called *functional instantiation* [5]. To make easier the reuse of this result we use a generic instantiation tool [21]. This tool is used as follows, we define a constant *ds* whose value is the list of the functor functions \{inv-ds, ..., diff-ds\} and the theorems |DIRECT-SUM-A-B-is-a-chain-complex| and correctness-ds that state the correctness of the construction. Finally, we include, at the end of the book[1], a call to the macro make-generic-theory of the generic instantiation tool with the constant *ds* as argument:

```lisp
(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 chain complexes involved to the names of the generic functions of the generic chain complexes defined in the encapsulate of the direct sum book; the second argument is provided to give new names to the new events obtained by instantiation[2], 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 \( D \):

---

1 In ACL2, a collection of definitions and theorems is called a “book”.

2 The second argument is a string that will be used as a suffix to append to the generic names.
(defun inv-C (n x) ... )
...
(defun diff-D (n a) ... )
...
(defun inv-D (n x) ... )
...
(defun diff-D (n a) ... )
...

and if we have proven 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 with the names correctness-ds-C-D and |DIRECT-SUM-A-B-is-a-chain-complex-C-D|. In this way, the direct sum of the chain complexes C and D and the proof of its correctness is automatically generated without any additional user interaction.

3.3 The general case

In the previous section, the correctness of the direct sum constructor of the Kenzo system has been stated in ACL2. The methodology followed in that case can be exported to prove the correctness of all the construction of Kenzo objects from other ones. The theorems that state the correctness of these constructors always have the same pattern.

Theorem 2 Let the Kenzo objects $X_1, \ldots, X_n$ which are instances of the mathematical structures $T_1, \ldots, T_n$ respectively and that (optionally) satisfy some properties $A_1, \ldots, A_n$, then the Kenzo constructor $\phi_{\text{Kenzo}}(X_1, \ldots, X_n)$ really codifies the mathematical constructor $\phi$ applied over objects that belong to the mathematical structures $T_1, \ldots, T_n$ and subject to the properties $A_1, \ldots, A_m$.

Let us explain now, the steps of our methodology to prove this kind of theorems.

First of all, we define each one of the mathematical structures $T_1, \ldots, T_n$ in the way explained in Section 2. Based on the ideas presented in Section 2, we
have codified the structures $T_1, \ldots, T_n$ with the records $T_1, \ldots, T_n$ (that can be instantiated with the functions \texttt{make-T1}, \ldots, \texttt{make-Tn}) and have defined the macro functions \texttt{check-T1-p}, \ldots, \texttt{check-Tn-p}, that check the properties of the respective mathematical structure for the record instances.

Afterwards, the next step is the definition of the generic instances of the mathematical structures $T_1, \ldots, T_n$, that is, the objects $X_1, \ldots, X_n$ of Theorem 2. This kind of generic development is feasible in the ACL2 theorem prover thanks to the \texttt{encapsulate} mechanism. Using this feature of ACL2, it is possible to introduce undefined objects that are instances of the mathematical structures. In the encapsulate, the signatures declare the function symbols which correspond with the operators of the mathematical objects; witnesses for these functions are provided in order to avoid the introduction of inconsistencies in the system; the calls to the macros \texttt{check-T1-p}, \ldots, \texttt{check-Tn-p} state the properties of the structures for the objects; and the theorems \texttt{property-A1}, \ldots, \texttt{property-Am} state the properties $A_1, \ldots, A_m$.

\begin{verbatim}
(encapsulate ; Signatures ...
 ; Witnesses ...
 ; Properties (check-T1-p (make-T1 ...) 'X1)
 ... (check-Tn-p (make-Tn ...) 'Xn)
 (defthm property-A1 ...) 
 ... (defthm property-Am ...) )
\end{verbatim}

The effect of the above encapsulate is an extension of the ACL2 universe in which the functions that determine the objects $X_1, \ldots, X_n$ have the syntactical signatures given and are axiomatized to satisfy the properties of the mathematical structures $T_1, \ldots, T_n$ respectively and that also satisfy the properties $A_1, \ldots, A_m$.

From the objects defined in the previous encapsulate, we develop the constructor $\phi_{K\text{enzo}}(X_1, \ldots, X_n)$ based on the Kenzo definition of the mathematical constructor $\phi$. This task is performed by the definition of a set of functions that depends on the objects $X_1, \ldots, X_n$, exporting the terminology of [7] these functions are called \textit{functor functions}. The functor functions are gathered in a record of the mathematical structure of $\phi$, let us say $T$. Then, we need to prove that the new object belongs to the mathematical structure $T$ with the macro function \texttt{check-T-p}.

\begin{verbatim}
 (check-T-p (make-T ...) 'phi)
\end{verbatim}

Let us remark that the macros like \texttt{check-T-p} generate a proof try 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 definition of the mathematical constructor $\phi$; but this can be consider a quite simple task because the Kenzo constructors have been defined following the mathematical definitions.

Hence, we have provided a methodology to proof the theorems that follow the pattern of Theorem 2. This kind of results represents approximately the 60\% of the Kenzo code.

Finally, in order to reuse the results of the generic books developed in the way presented throughout this subsection, 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 a generic instantiation tool [21]. 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\(^3\). Finally, at the end of the book, let us call this book “generic.lisp”, we include a call to the macro of the generic instantiation tool:

\begin{verbatim}
(make-generic-theory *theory*)
\end{verbatim}

This macro automatically defines a new macro named 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”; the second argument is provided to give new names to the new events obtained by instantiation, that are the functor functions and the theorems that state the correctness of the operator $\phi$. In this way, if we give objects $Y_1,\ldots,Y_n$ that are concrete instances of the mathematical structures $T_1,\ldots,T_n$ and that are subjected to the properties $A_1,\ldots,A_m$, the object $\phi_{\text{Kenzo}}(Y_1,\ldots,Y_n)$ and a proof of its correctness are automatically generated without any additional user interaction.

4 Case studies

In order to evaluate how practical our framework was, we have started formalizing some important results included in the Kenzo system. In this section, we present two of them: the Easy Perturbation Lemma and the $SE\bar{E}1$ Theorem.

\(^3\) the generic instantiation tool [21] provides a way of generate this list of events in a convenient way
The complete development of these results and also more examples can be consulted in [14]. Our main source of inspiration for the proofs of these theorems was some lectures notes on Constructive Homological Algebra by J. Rubio and F. Sergeraert [26].

4.1 The Easy Perturbation Lemma

The algebraic structures appearing in the Easy Perturbation Lemma are chain complexes, perturbations and reductions. Roughly speaking, a perturbation over a chain complex is a module endomorphism of that chain complex, which produces a chain complex when added to the original differential. A reduction is a triple of morphisms between a pair of chain complexes (usually called top and bottom) satisfying some special requirements. The notion of reduction is one of the most relevant notions in the Kenzo system since they preserve the homology of chain complexes; that means, the homology groups of the top chain complex are the same that the homology groups of the bottom chain complex. Now, we introduce in a precise way the definitions required to state the Easy Perturbation Lemma.

Definition 7 A reduction $\rho$ between two chain complexes $C_*$ and $D_*$, denoted in this article by $\rho : C_* \Rightarrow D_*$, is a triple $\rho = (f, g, h)$

\[
\begin{tikzpicture}
  \node (C) at (0,0) {$C_*$};
  \node (D) at (2,0) {$D_*$};
  \draw[->] (C) -- (D) node[midway,above] {$f$};
  \draw[->] (C) -- (D) node[midway,below] {$g$};
  \draw[->] (C) -- (D) node[midway,left] {$h$};
\end{tikzpicture}
\]

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

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

Definition 8 Let $C_* = (C_n, d_C)_n \in \mathbb{Z}$ be a chain complex. A perturbation $\delta$ of the differential $d$ is a collection of group morphisms $\delta = \{\delta_n : C_n \to C_{n-1}\}_{n \in \mathbb{Z}}$ such that the sum $d + \delta$ is also a differential, that is to say, $(d + \delta) \circ (d + \delta) = 0$.

Then, the Easy Perturbation Lemma is stated as follows.

Theorem 3 (Easy Perturbation Lemma, EPL) 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, $\rho = (f, g, h) : C_* \Rightarrow D_*$ a reduction, and $\delta_D$ a perturbation of $d_D$. Then a new reduction $\rho' = (f', g', h') : C_*' \Rightarrow D_*'$ 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 follows the pattern of Theorem 2. In this case, the instances $X_1, \ldots, X_n$ of the mathematical structures $T_1, \ldots, T_n$ of Theorem 2 are $\rho$ (an instance of a reduction) and $\delta_D$ (an instance of a graded morphism) and in this case the theorems $A_1, \ldots, A_m$ state the properties of being a perturbation for $\delta_D$. In this case, we need to prove that the constructor $EPL_{Kenzo}(\rho, \delta_D)$ codifies the reduction defined in the Easy Perturbation Lemma.

Then, following the guidelines of subsection 3.3 we proceed as follows. The necessary records and macros to define reductions and morphisms are provided in the way explained in Section 2. Afterwards, a generic reduction and a generic perturbation of the bottom chain complex of the reduction are declared using the encapsulate mechanism.

\begin{verbatim}
(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--1-p (make-morphism--1 ... :function 'delta) 'delta) 
  (defthm perturbation-property ...) 
)
\end{verbatim}

From these objects and based on the mathematical definitions given in Theorem 3, the functor functions to declare the new reduction of Theorem 3 are introduced, for instance the differential of the new top chain complex of the new reduction is $(d_{top} + g \circ \delta_D \circ f)$ which is declared in ACL2 as follows.

\begin{verbatim}
(defun new-diff-top (n a) 
  (group-operation-top (1- n) (diff-top n a) (g (1- n) (delta n (f n a)))))
\end{verbatim}

With the new definitions, the statement of the EPL is now:

\begin{verbatim}
(check-reduction-p 
  (make-reduction 
    :top-cc (make-chain-complex ... :diff-cc 'new-diff-top) 
    :bottom-cc (make-chain-complex ...) 
    :f 'f :g 'g :h 'h) 'EPL)
\end{verbatim}

The proof try generated by the call to the above macro is unsuccessful in the first try; 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 satisfies the properties of 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 proven in ACL2 by applying equational reasoning over morphisms, following closely the style of a paper and pencil proof.

4.2 The $SES_1$ Theorem

The algebraic structures appearing in the $SES_1$ Theorem are chain complexes, equivalences and short exact sequences. Let two chain complexes (usually called left and right), an *equivalence* is a triple of a chain complex (usually called top), a reduction between the top and the left chain complex, and a reduction between the top and the right chain complex. The notion of equivalence is one of the most relevant notions in the Kenzo system since, as in the case of reductions, they preserve the homology of chain complexes; that means, the homology groups of the left chain complex are the same that the homology groups of the right chain complex. Now, we introduce in a precise way the definitions required to state the $SES_1$ Theorem.

**Definition 9** An *equivalence* $\varepsilon$ between two chain complexes $C_\ast$ and $D_\ast$, denoted by $\varepsilon : C_\ast \leftrightarrow D_\ast$, is a triple $(B_\ast, \rho_1, \rho_2)$ where $B_\ast$ is a chain complex, and $\rho_1$ and $\rho_2$ are reductions from $B_\ast$ over $C_\ast$ and $D_\ast$ respectively: $C_\ast \xrightarrow{\rho_1} B_\ast \xrightarrow{\rho_2} D_\ast$.

**Definition 10** A short exact sequence is a sequence of modules:

$$0 \leftarrow C'' \xleftarrow{i} C \xrightarrow{j} C' \leftarrow 0$$

which is exact, that is in this case, the map $i$ is injective, the map $j$ is surjective and $\text{im } i = \text{ker } j$.

**Definition 11** An *effective short exact sequence* of chain complexes is a diagram:

$$0 \xleftarrow{0} A \xrightarrow{i} B \xrightarrow{\sigma} C \xrightarrow{\rho} 0$$

where $i$ and $j$ are chain complexes morphisms, $\rho$ (retraction) and $\sigma$ (section) are graded module morphisms satisfying:

- $\rho i = \text{id}_C$;
- $i \rho + \sigma j = \text{id}_B$;
- $j \sigma = \text{id}_A$.

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.

Before stating the $SES_1$ Theorem, we introduce a definition and two theorems that are required for its proof.
Theorem 4 (Composition of reductions) Let $\rho = (f, g, h) : C_\ast \Rightarrow \Rightarrow D_\ast$ and $\rho' = (f', g', h') : D_\ast \Rightarrow \Rightarrow E_\ast$ be two reductions. Another reduction $\rho'' = (f'', g'', h'') : C_\ast \Rightarrow \Rightarrow E_\ast$ is defined by:

\[
\begin{align*}
  f'' & = f' \circ f \\
  g'' & = g \circ g' \\
  h'' & = h + g \circ h' \circ f
\end{align*}
\]

Definition 12 Given two chain complexes $A = (A_n, d_{A_n})_{n \in \mathbb{Z}}, B = (B_n, d_{B_n})_{n \in \mathbb{Z}}$ and $\phi : A \leftarrow B$ a chain complex morphism. Then the cone of $\phi$ denoted by $\text{Cone}(\phi) = (C_n, d_n)_{n \in \mathbb{Z}}$ such that, $C_n = (A_n, B_{n-1})$ and $d_n = (d_{A_n} + \phi_n, -d_{B_n-1})$ for all $n \in \mathbb{Z}$.

Theorem 5 (Cone Equivalence Theorem) Let two equivalences $A \Leftarrow \Rightarrow HA, B \Leftarrow \Rightarrow HB$ and $\phi : A \leftarrow B$ a chain complex morphism between $A$ and $B$ then we can construct an equivalence for $\text{Cone}(\phi)$.

A proof of the Cone Equivalence Theorem can be seen in [26]. Now, all the ingredients to state and prove the $SES_1$ Theorem are available.

Theorem 6 ($SSES_1$ Theorem) Let

\[
0 \xleftarrow{\sigma} A \xrightarrow{\rho} B \xrightarrow{i} C \xrightarrow{\iota} 0
\]

be an effective short exact sequence of chain complexes and given the equivalences $B \Leftarrow \Rightarrow HB$ and $C \Leftarrow \Rightarrow HC$. Then an equivalence for the chain complex $A$ can be constructed.

The procedure to construct the equivalence of the chain complex $A$ given the effective short exact sequence is as follows (we present here a sketch of the procedure, the complete one can be consulted in [26]). First of all, the chain complex $\text{Cone}(i)$ is constructed; afterwards, we apply the Cone Equivalence Theorem (Theorem 5 can be applied because equivalences for both $B$ and $C$ are given) obtaining the equivalence $\text{Cone}(i) \Leftarrow \Rightarrow TC \Rightarrow RC$. Subsequently, the reduction $A \Leftarrow \Rightarrow \text{Cone}(i)$ is constructed. Finally, using the Theorem 4 the reductions $A \Leftarrow \Rightarrow \text{Cone}(i)$ and $\text{Cone}(i) \Leftarrow \Rightarrow TC$ can be composed and the equivalence $A \Leftarrow \Rightarrow TC \Rightarrow RC$ obtained.

Let us note that the $SSES_1$ Theorem follows the pattern of Theorem 2. In this case, the instances $X_1, \ldots, X_n$ of the mathematical structures $T_1, \ldots, T_n$ of Theorem 2 are $A$ (an instance of a chain complex), $B \Leftarrow \Rightarrow HB$ (an instance of an equivalence), $C \Leftarrow \Rightarrow HC$ (an instance of an equivalence), $i$ (an instance of a chain complex morphism), $j$ (an instance of a chain complex morphism), $\sigma$ (an instance of a graded morphism) and $\rho$ (an instance of a graded morphism) and in this case the theorems $A_1, \ldots, A_m$ state the properties of being an effective short exact sequence. In this case, we need to prove that the constructor $SSES_{1_{Kan}}(A, B \Leftarrow \Rightarrow HB, C \Leftarrow \Rightarrow HC, i, j, \rho, \sigma)$ codifies a equivalence for the chain complex $A$ defined in the way explained in the previous paragraph.
In order to translate the $SES_1$ theorem into the ACL2 context we need to apply the methodology presented in subsection 3.3 several times. First of all, the generic cone construction is implemented in the same way explained for the direct sum in a book called “cone.lisp” (from a generic chain complex morphism instance declared in the encapsulate, the cone chain complex is constructed). Secondly, we use again our methodology to prove the Cone Equivalence Theorem (Theorem 5) in a book called “cone-equivalence-reduction.lisp” (in that book the cone construction of the “cone.lisp” book is instantiated three times), and Theorem 4 in a book named “cmps-reductions.lisp”; these theorems are stated in the terms of Theorem 2. Finally, the proof of the $SES_1$ theorem is tackled with the presented methodology and following the schema of the above paragraph, that is, an encapsulate with the generic objects is declared, then the Cone Equivalence Theorem is instantiated and the equivalence $\text{Cone}(i) \iff TC \Rightarrow RC$ is obtained. Subsequently, the reduction $A \iff \text{Cone}(i)$ is constructed. Finally, by means of the instantiation of Theorem 4 the reductions $A \iff \text{Cone}(i)$ and $\text{Cone}(i) \iff TC$ can be composed and the equivalence $A \iff TC \Rightarrow RC$ constructed. It is worth noting the importance of the use of the generic instantiation tool because in this case the number of events to instantiate is high and the process could be considered tedious without the generic instantiation tool.

5 Distance from ACL2 to Kenzo code

With respect to the distance between the verified code and the actual Kenzo programs, it is necessary to stress that there is still some room between them. Since the ACL2 programming language is a subset of Common Lisp, the translation of several Kenzo fragments to ACL2 is quite direct. However, due to the nature of ACL2 (applicative, free of side effect and so on), there are some things that have to be defined in a different (but equivalent) way.

The representation of the mathematical structures is one of that features that needs a pre-processing step. Mathematical structures are implemented by means of classes in the Kenzo system, see [28] for a detailed description of this implementation, and records in ACL2. In both cases the idea is the same, mathematical structures are codified 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, devoted 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\textsuperscript{4} 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 of proving, but some of them are unnecessary.

\textsuperscript{4} memoization is a technique that stores previously computed results in order to avoid recomputations.
to compute so they are not included in Kenzo; for instance, the ACL2 chain complex record includes a slot that returns the inverse of an element, this function is useless in Kenzo, so it is not included in the definition of the Kenzo chain complex class.

Moreover, it is worth noting that Kenzo objects such as chain complexes or reductions are defined over the ring \( \mathbb{Z} \) (the most important case in Algebraic Topology; see [26]); 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.

In addition, the slots in the definition of the class include their type (in the sense of the implementation of that type in Kenzo). As we commented previously ACL2 is an untyped system and to provide the type (in the sense of the properties of the mathematical structure) of the slots of our records we use the macros explained in Section 2.

Once we have presented the differences between the structures used to represent the objects in both systems, let us focus now on the values of the slots. First of all, there is a difference between the representation of the underlying set in Kenzo and in ACL2. In the Kenzo system the elements of the underlying set are given by means of an explicit list; on the contrary in ACL2 we use a characteristic function. Then, we need a (direct) transformation between the explicit Kenzo list and the ACL2 invariant function.

One of the most important transformations is the following one. 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 (Theorem 4) is given in Kenzo as follows

```lisp
(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 bottom and top reductions, \( \text{cmps} \) is the function in charge of the composition of two functions, \( \text{add} \) is the sum of functions and \( \text{i-cmps} \) is a macro composing \( n \)-functions with \( n \geq 2 \). The functions \( \text{cmps}, \text{add} \) and \( \text{i-cmps} \) 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.

```lisp
(defun f (n a) (tf n (bf n a)))
(defun g (n a) (tg n (tg n a)))
(defun h (n a) (group-operation-top (1+ n) (th n a) (tg (1+ n) (bh n (tf n a)))))
```

```lisp
(make-reduction ...
 :f 'f
 :g 'g
 :h 'h)
```
The uncurrying technique is a well-known form of defunctionalization [25] and can be considered as a safe transformation.

Then, in spite of existing some differences between the Kenzo and ACL2 code the transformations between them are not risky and the verified code is very close to the actual Kenzo code.

6 Conclusions and Further Work

A framework to prove the correctness of construction of spaces from other ones implemented in the Kenzo system has been presented. As examples of application, we have given a complete correctness proof of the implementation in Kenzo of 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, more examples can be seen in [14].

As a by-product, a partial implementation for ACL2 of a algebraic hierarchy has been given. This development can be useful for works non related to our developments.

Some parts of the future work are quite natural. With the acquire experience, the presented methodology can be extrapolated to the rest of construction of spaces from other ones by means of topological constructors implemented in the Kenzo system. To this aim, the current hierarchy should be enriched by means of the rest of mathematical data structures implemented in Kenzo. For instance we do not have the coalgebra structure because it did not occur in the presented work. Let us remark that the presented methodology can be used for the rest of constructions but this does not mean that the proof of the correctness of the construction becomes trivial; for instance the construction of loop spaces from simplicial sets follows the same pattern but the proof of the correctness of the underlying algorithm can be considered as a difficult task.

The combination of the current work and the work presented in [15] provides a methodology for proving the correctness of the Kenzo constructions. Hence, the remaining work consists of proving the correctness of the algorithms in charge of the computations in the Kenzo system.

Finally, we would like to automatize the defunctionalization process explained in Section 5 avoiding in this way the manual transformation. Moreover, uncurrying is not the only defunctionalization technique, see [8] for instance, then different approaches could be studied.

References


