Data structures and algorithms for Algebraic Topology in Proof Assistants

Jesús Aransay, *César Domínguez*

Universidad de La Rioja
Departamento de Matemáticas y Computación

MAP 2009

Monastir, December 14-18, 2009

Introduction

- Symbolic Computation system in Algebraic Topology: Kenzo

Introduction

- Symbolic Computation system in Algebraic Topology: Kenzo
- Results not obtained by another method (neither theoretic, nor automatic)

Introduction

- Symbolic Computation system in Algebraic Topology: Kenzo
- Results not obtained by another method (neither theoretic, nor automatic)
- With these results in mind, an interest emerged in a formal analysis of the systems which helps us to reason about the internal processes

Introduction

- Symbolic Computation system in Algebraic Topology: Kenzo
- Results not obtained by another method (neither theoretic, nor automatic)
- With these results in mind, an interest emerged in a formal analysis of the systems which helps us to reason about the internal processes
- Formal methods
  - Algebraic specification

Introduction

- Symbolic Computation system in Algebraic Topology: Kenzo
- Results not obtained by another method (neither theoretic, nor automatic)
- With these results in mind, an interest emerged in a formal analysis of the systems which helps us to reason about the internal processes
- Formal methods
  - Algebraic specification
  - Mechanized reasoning: $\begin{cases} \textit{Isabelle/HOL} \\ \textit{Coq} \\ \textit{ACL2} \\ \dots \end{cases}$

Kenzo characteristics

- Two *layers* of data structures exist:
  - Usual data structures as (sorted) lists or trees of symbols
  - Algebraic structures as (graded) groups, chain complexes or simplicial sets

Kenzo characteristics

- Two *layers* of data structures exist:
  - Usual data structures as (sorted) lists or trees of symbols
  - Algebraic structures as (graded) groups, chain complexes or simplicial sets
- Algorithms in both layers are involved:
  - Combination addition lemma: it is possible two append two sorted lists...
  - Basic Perturbation Lemma: given two chain complexes and several morphisms between them, then...

Kenzo characteristics

- Two *layers* of data structures exist:
  - Usual data structures as (sorted) lists or trees of symbols
  - Algebraic structures as (graded) groups, chain complexes or simplicial sets
- Algorithms in both layers are involved:
  - Combination addition lemma: it is possible two append two sorted lists...
  - Basic Perturbation Lemma: given two chain complexes and several morphisms between them, then...
- From a programming point of view:
  - Implemented in CLOS
  - Symbolic manipulation of data structures (first data layer)
  - Higher-order functional programming (second data layer)
  - Algorithms are exponential: efficiency matters were crucial

Different theorem provers or proof assistants

- ACL2

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic
  - very well suited to work with data structures and algorithms in the first layer

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic
  - very well suited to work with data structures and algorithms in the first layer
- Isabelle/HOL or Coq

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic
  - very well suited to work with data structures and algorithms in the first layer
- Isabelle/HOL or Coq
  - higher-order logic

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic
  - very well suited to work with data structures and algorithms in the first layer
- Isabelle/HOL or Coq
  - higher-order logic
  - without direct relation with Common Lisp

Different theorem provers or proof assistants

- ACL2
  - extension of a sub-language of Common Lisp and an environment to produce proofs
  - first-order logic
  - very well suited to work with data structures and algorithms in the first layer
- Isabelle/HOL or Coq
  - higher-order logic
  - without direct relation with Common Lisp
  - useful to model and verify the Kenzo data structures and algorithms in both layers

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq
- Formal proof of a Kenzo's crucial lemma (the *Basic Perturbation Lemma*, or BPL) in Isabelle/HOL (for non-graded structures)

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq
- Formal proof of a Kenzo's crucial lemma (the *Basic Perturbation Lemma*, or BPL) in Isabelle/HOL (for non-graded structures)

Goals

1. To represent first data layer structures and prove some algorithms with them in Isabelle/HOL and Coq

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq
- Formal proof of a Kenzo's crucial lemma (the *Basic Perturbation Lemma*, or BPL) in Isabelle/HOL (for non-graded structures)

Goals

1. To represent first data layer structures and prove some algorithms with them in Isabelle/HOL and Coq
2. To provide a suitable implementation of *graded structures*

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq
- Formal proof of a Kenzo's crucial lemma (the *Basic Perturbation Lemma*, or BPL) in Isabelle/HOL (for non-graded structures)

Goals

1. To represent first data layer structures and prove some algorithms with them in Isabelle/HOL and Coq
2. To provide a suitable implementation of *graded structures*
3. To implement instances of these structures and to prove some significant results with them

Some preliminary results

- Formal proof in ACL2 of data structures and algorithms in the first data layer as the *Combination addition lemma*
- Representation of non-graded algebraic structures in Isabelle/HOL and Coq
- Formal proof of a Kenzo's crucial lemma (the *Basic Perturbation Lemma*, or BPL) in Isabelle/HOL (for non-graded structures)

Goals

1. To represent first data layer structures and prove some algorithms with them in Isabelle/HOL and Coq
2. To provide a suitable implementation of *graded structures*
3. To implement instances of these structures and to prove some significant results with them
4. To compare the capabilities and styles of the systems

Basic algebraic structure

Definition

A *free abelian group* $(M, +)$ is an abelian group in which each element in M can be written as a finite linear combination of elements of a set G called the generators

Basic algebraic structure

Definition

A *free abelian group* $(M, +)$ is an abelian group in which each element in M can be written as a finite linear combination of elements of a set G called the generators

Second layer implementation:

Definition

A free abelian group as a CLOS class with functional elements as slots.

Basic algebraic structure

Definition

A *free abelian group* $(M, +)$ is an abelian group in which each element in M can be written as a finite linear combination of elements of a set G called the generators

Second layer implementation:

Definition

A free abelian group as a CLOS class with functional elements as slots.

First layer implementation:

Definition

A combination as a list of pairs (integer, generator) called terms. Besides, the list of pairs is sorted in order to speed up the execution.

Algorithm in the first layer

- Two different methods can be proposed to add (sorted) combinations
  - To append and then sort
  - To add each term in the first combination in the corresponding position of the second combination

Algorithm in the first layer

- Two different methods can be proposed to add (sorted) combinations
  - To append and then sort
  - To add each term in the first combination in the corresponding position of the second combination

Combination addition lemma

Both methods are equivalent

Implementation in Isabelle/HOL

Isabelle/HOL is an implementation of Higher-Order logic.

Implementation in Isabelle/HOL

Isabelle/HOL is an implementation of Higher-Order logic.

The type system is rather simple and contains:

1. Type variables $(\alpha, \beta, \ldots)$
2. Arrow types or functions $(\alpha \Rightarrow \beta)$
3. Pairs $(\alpha \times \beta)$ (and thus labelled products, or records)

These constructors will be the ones used to represent both first and second layer structures as list or chain complexes and their morphisms.

First layer data structures implementation in Isabelle/HOL

A type class containing types with a strict total order is defined.

Type class declaration

class *order* =
  fixes *order_rel*:: "'a ⇒ 'a ⇒ bool" (**infixl** "≪" 60)
  assumes *total*: "a = b ∨ a ≪ b ∨ b ≪ a"
  and *transitive*: "a ≪ b ∧ b ≪ c ⟹ a ≪ c"
  and *irreflexive*: "¬ a ≪ a"

First layer data structures implementation in Isabelle/HOL

A type class containing types with a strict total order is defined.

Type class declaration

```
class order =
  fixes order_rel:: "'a ⇒ 'a ⇒ bool" (infixl "≪" 60)
  assumes total: "a = b ∨ a ≪ b ∨ b ≪ a"
  and transitive: "a ≪ b ∧ b ≪ c ⟹ a ≪ c"
  and irreflexive: "¬ a ≪ a"
```

Type declaration for terms, list of terms, and combinations

```
types 'a pair = "(int × 'a)"
types 'a lot = "('a pair) list"
fun cmbn :: "'a::order lot ⇒ bool"  where
   "cmbn [] = True" |
   "cmbn [x] = (fst x ≠ (0::int))" |
   "cmbn (x#y#z) = (fst x ≠ 0 ∧ snd x ≪ snd y ∧ cmbn (y # z))"
```

First layer algorithms in Isabelle/HOL

Algorithms by recursion on the structures.

Sorting lists of terms

```
fun c_f ::
"('a::order) lot ⇒ 'a lot"
where
"c_f [] = []" |
"c_f (x # y) =
(if (fst x = 0) then c_f y
  else x [+] (c_f y))"
```

Addition of lists of terms

```
fun a2c ::
 "'a lot ⇒ 'a lot ⇒ 'a lot"
where
 "a2c [] l2 = l2"  |
 "a2c (x#l) l2 = x[+](a2c l l2)"
```

with *[+]* recursive function adding a term to a sorted list.

First layer algorithms in Isabelle/HOL

Algorithms by recursion on the structures.

Sorting lists of terms

fun *c_f ::*
"('a::order) lot ⇒ 'a lot"
where
"c_f [] = []" |
"c_f (x # y) =
(if (fst x = 0) then c_f y
else x [+] (c_f y))"

Addition of lists of terms

fun *a2c ::*
"'a lot ⇒ 'a lot ⇒ 'a lot"
where
"a2c [] l2 = l2" |
"a2c (x#l) l2 = x[+](a2c l l2)"

with *[+]* recursive function adding a term to a sorted list.

Combination addition lemma

theorem assumes *cmbn l1* **and** *cmbn l2*
shows *a2c l1 l2 = c_f (l1@l2)*

First layer algorithms in Isabelle/HOL

Algorithms by recursion on the structures.

Sorting lists of terms

```
fun c_f ::
"('a::order) lot ⇒ 'a lot"
where
"c_f [] = []" |
"c_f (x # y) =
(if (fst x = 0) then c_f y
  else x [+] (c_f y))"
```

Addition of lists of terms

```
fun a2c ::
 "'a lot ⇒ 'a lot ⇒ 'a lot"
where
 "a2c [] l2 = l2"  |
 "a2c (x#l) l2 = x[+](a2c l l2)"
```

with *[+]* recursive function adding a term to a sorted list.

Combination addition lemma

theorem assumes *cmbn l1* **and** *cmbn l2*
  shows *a2c l1 l2 = c_f (l1@l2)*

Proof.

By induction on the l1 structure. □

Implementation in Coq

Coq is based on a variation of typed λ-calculus called Calculus of Inductive Constructions

Implementation in Coq

Coq is based on a variation of typed λ-calculus called Calculus of Inductive Constructions

The type system is richer than the one in Isabelle/HOL

For instance, dependent types can be defined

First layer data structures implementation in Coq

First layer structures are defined using inductive types.

A type with a strict total order can be declared

```
Record strict_total_order: Type:=
 {A:> Set;
  Alt: A -> A -> Prop;
  Alt_irreflexive: forall x:A, not(Alt x x);
  Alt_transitive: forall x y z:A, Alt x y -> Alt y z -> Alt x z;
  Alt_total: forall x y:A, {Alt x y}+{Alt y x}+{x = y}}.
```

First layer data structures implementation in Coq

First layer structures are defined using inductive types.

A type with a strict total order can be declared

```
Record strict_total_order: Type:=
 {A:> Set;
  Alt: A -> A -> Prop;
  Alt_irreflexive: forall x:A, not(Alt x x);
  Alt_transitive: forall x y z:A, Alt x y -> Alt y z -> Alt x z;
  Alt_total: forall x y:A, {Alt x y}+{Alt y x}+{x = y}}.
```

Inductive types for terms, list of terms, and combinations

```
Inductive term: Set:= term_cons: forall x:Z, x<>0->A->term.
Definition lot:= list(term).
Inductive cmbn: lot->Prop:=
| null_cmbn: cmbn(nil)
| cons_cmbn1: forall t:term, cmbn(t::nil)
| cons_cmbn2: forall (t1 t2 :term) (l:list(term)),
  (let (a,p1,b):= t1 in let (c,p2,d):= t2 in
  (Alt d b))->cmbn((t1::l))->cmbn((t2::(t1::l))).
```

First layer algorithms in Coq

Algorithms by recursion on the structures.

Sorting list of terms

```
Fixpoint c_f(l:lot):lot:=
match l with
|null => null
|t::l' => (add t (c_f l'))
end.
```

Addition of lists of terms

```
Fixpoint a2c(l1 l2:lot){struct l1}:
lot:=
match l1 with
|null => l2
|t::l => (add t (a2c l l2))
end.
```

with add recursive function adding a term to a ordered list.

First layer algorithms in Coq

Algorithms by recursion on the structures.

Sorting list of terms

```
Fixpoint c_f(l:lot):lot:=
match l with
|null => null
|t::l' => (add t (c_f l'))
end.
```

Addition of lists of terms

```
Fixpoint a2c(l1 l2:lot){struct l1}:
lot:=
match l1 with
|null => l2
|t::l => (add t (a2c l l2))
end.
```

with add recursive function adding a term to a ordered list.

Combination addition lemma

```
Lemma a2c_equivalence:  forall (l1 l2:lot), cmbn(l1)->cmbn(l2)->
(a2c l1 l2) = (c_f (app l1 l2)).
```

First layer algorithms in Coq

Algorithms by recursion on the structures.

Sorting list of terms

```
Fixpoint c_f(l:lot):lot:=
match l with
|null => null
|t::l' => (add t (c_f l'))
end.
```

Addition of lists of terms

```
Fixpoint a2c(l1 l2:lot){struct l1}:
lot:=
match l1 with
|null => l2
|t::l => (add t (a2c l l2))
end.
```

with add recursive function adding a term to a ordered list.

Combination addition lemma

```
Lemma a2c_equivalence: forall (l1 l2:lot), cmbn(l1)->cmbn(l2)->
(a2c l1 l2) = (c_f (app l1 l2)).
```

Proof.

By induction on the `cmbn(l1)` structure. □

Comparison of both approaches

Representation of first data layer

- Both systems provide suitable frameworks to implement first data layer structures and algorithms.

Comparison of both approaches

Representation of first data layer

- Both systems provide suitable frameworks to implement first data layer structures and algorithms.
- They include explicit libraries, and induction and recursion mechanisms to obtain direct implementations.

Comparison of both approaches

Representation of first data layer

- Both systems provide suitable frameworks to implement first data layer structures and algorithms.
- They include explicit libraries, and induction and recursion mechanisms to obtain direct implementations.
- Differences in their underlying logic appear. For instance, to represent generators and strict total order:
  - Isabelle: type classes with type variables (*'a*), predicates (*bool*), and classes (*class*)
  - Coq: sorts `Set` and `Prop` and dependent types

Comparison of both approaches

Representation of first data layer

- Both systems provide suitable frameworks to implement first data layer structures and algorithms.
- They include explicit libraries, and induction and recursion mechanisms to obtain direct implementations.
- Differences in their underlying logic appear. For instance, to represent generators and strict total order:
  - Isabelle: type classes with type variables (*'a*), predicates (*bool*), and classes (*class*)
  - Coq: sorts `Set` and `Prop` and dependent types

Algorithms in the first data layer

Proof by induction in both systems in an interactive way using the already built-in tactics.

Algebraic structures

Non graded structures:

Definition

A *left R-module over the ring R* consists of an abelian group $(M, +)$ and an operation $\cdot : R \times M \to M$ such that for all $r, s \in R$, $x, y \in M$, we have

1. $r \cdot (x + y) = r \cdot x + r \cdot y$
2. $(r +_R s) \cdot x = r \cdot x + s \cdot x$
3. $(r \cdot_R s) \cdot x = r \cdot (s \cdot x)$
4. $1_R \cdot x = x$

Algebraic structures

Non graded structures:

Definition

A *left R-module over the ring R* consists of an abelian group $(M, +)$ and an operation $\cdot : R \times M \to M$ such that for all $r, s \in R$, $x, y \in M$, we have

1. $r \cdot (x + y) = r \cdot x + r \cdot y$
2. $(r +_R s) \cdot x = r \cdot x + s \cdot x$
3. $(r \cdot_R s) \cdot x = r \cdot (s \cdot x)$
4. $1_R \cdot x = x$

Graded structures:

Definition

A *graded left R-module over the ring R* consists of a family of abelian groups $(M_n, +_n)_{n \in \mathbb{Z}}$ and operations $\cdot_n : R \times M_n \to M_n$ such that for all $n \in \mathbb{Z}$, M_n is a left R-module

Differential algebraic structures

Non graded structures:

Definition

A *differential* d over a left R-module M is an endomorphism of M such that it verifies the nilpotency condition, *i.e.*, $d \circ d = 0$

Differential algebraic structures

Non graded structures:

Definition

A *differential* d over a left R-module M is an endomorphism of M such that it verifies the nilpotency condition, *i.e.*, $d \circ d = 0$

Graded structures:

Definition

A *differential* $\{d_n\}_{n \in \mathbb{Z}}$ of degree -1 over a graded left R-module is a family of R-module morphisms $d_n \colon M_n \rightarrow M_{n-1}$ such that, for all $n \in \mathbb{Z}$, $d_{(n-1)} \circ d_n = 0_{\text{Hom}\, M_n\, M_{n-2}}$

Differential algebraic structures

Non graded structures:

Definition

A *differential left R-module* (M, d) is a left R-module M together with a differential d of M

Differential algebraic structures

Non graded structures:

Definition

A *differential left R-module* (M, d) is a left R-module M together with a differential d of M

Graded structures:

Definition

A *chain complex* $\{M_n, d_n\}_{n\in\mathbb{Z}}$ is a pair of a graded left R-module $\{M_n\}_{n\in\mathbb{Z}}$ together with a graded differential $\{d_n\}_{n\in\mathbb{Z}}$ of degree -1

Morphisms of differential algebraic structures

Non graded structures:

Definition

A *morphism between two differential left R-modules* (M, d) and (M', d') is a morphism of the modules such that $f \circ d = d' \circ f$

Morphisms of differential algebraic structures

Non graded structures:

Definition

A *morphism between two differential left R-modules* (M, d) and (M', d') is a morphism of the modules such that $f \circ d = d' \circ f$

Graded structures:

Definition

A *chain complex morphism of degree* $+1$ between two chain complexes $\{(M_n, d_n)\}_{n \in \mathbb{Z}}$ and $\{(M'_n, d'_n)\}_{n \in \mathbb{Z}}$ is a family of morphisms $\{f_n\}_{n \in \mathbb{Z}}$, such that, for all $n \in \mathbb{Z}$, $f_n \colon M_n \to M'_{(n+1)}$ is a morphism and $f_{n-1} \circ d_n = d'_{n+1} \circ f_n$

Algorithm in the second layer

Trivial Perturbation Lemma

Let $\rho = (D, C, f, g, h)$ be a reduction (*i.e.*, D, C chain complexes and f, g, h chain complexes morphisms verifying some known properties), and δ a perturbation of d_C (*i.e.*, a chain complex morphism defined over C of degree -1 such that $(d_C + \delta) \circ (d_C + \delta) = 0$). Then a new reduction $\rho' = (D', C', f', g', h')$ is defined where:

- D' is the chain complex obtained from D where $d_{D'} = d_D + g\delta f$
- C' is the chain complex obtained from C where $d_{C'} = d_C + \delta$
- $f' = f$, $g' = g$ and $h' = h$

Implementation in Isabelle/HOL

First we provide a type definition and specification for non graded structures (for instance, a module):

Type definition

record (α, β) *module* $= \alpha$ *ring* $+$
smult :: $\alpha \Rightarrow \beta \Rightarrow \beta$ (*infixl* $\cdot_{\_}$ *70*)

Implementation in Isabelle/HOL

First we provide a type definition and specification for non graded structures (for instance, a module):

Type definition

$$\textit{record } (\alpha, \beta) \textit{ module} = \alpha \textit{ ring} + \\ \textit{smult} :: \alpha \Rightarrow \beta \Rightarrow \beta \quad (\textit{infixl} \cdot_{\_} \ 70)$$

Specification

$$\begin{aligned} &\textit{module } R\ M = \textit{cring } R + \textit{abelian\_group } M \\ &\quad (\forall a. \forall m.\ a \cdot_M m \in \textit{carrier } M) + \\ &\quad (\forall a\, b. \forall x.\ (a + b) \cdot_M x = a \cdot_M x + b \cdot_M x) + \\ &\quad (\forall a. \forall x\, y.\ a \cdot_M (x +_M y) = a \cdot_M x +_M a \cdot_M y) + \\ &\quad (\forall a\, b. \forall x.\ (a \times b) \cdot_M x = a \cdot_M (b \cdot_M x)) + \\ &\quad (\forall x.\ 1 \cdot_M x = x) \end{aligned}$$

Implementation in Isabelle/HOL

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

definition graded_module :: α *ring* $\Rightarrow$ *(int* $\Rightarrow$ (α, β) *module)* $\Rightarrow$ *bool*

Implementation in Isabelle/HOL

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

definition graded_module :: α *ring* $\Rightarrow$ *(int* $\Rightarrow$ (α, β) *module)* $\Rightarrow$ *bool*
where graded_module R f $\equiv$ $\forall$*n. module R (f n)*

We use a function that, given a ring R, maps every integer to a R-module

Implementation in Isabelle/HOL

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

definition graded_module :: α *ring* $\Rightarrow$ *(int* $\Rightarrow$ (α, β) *module)* $\Rightarrow$ *bool*
where graded_module R f $\equiv \forall n.$ *module R (f n)*

We use a function that, given a ring R, maps every integer to a R-module

We can also provide a definition for graded module morphisms

Graded module morphism (degree -1)

definition graded_module_hom ::
α *ring* $\Rightarrow$ *(int* $\Rightarrow$ (α, β) *module)* $\Rightarrow$ *(int* $\Rightarrow$ (α, δ) *module)* $\Rightarrow$
(int $\Rightarrow (\beta \Rightarrow \delta)) \Rightarrow$ *bool*

Implementation in Isabelle/HOL

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

definition graded_module :: α ring $\Rightarrow$ (int $\Rightarrow$ (α, β) module) $\Rightarrow$ bool
where graded_module R f $\equiv \forall n.$ module R (f n)

We use a function that, given a ring R, maps every integer to a R-module

We can also provide a definition for graded module morphisms

Graded module morphism (degree -1)

definition graded_module_hom ::
α ring $\Rightarrow$ (int $\Rightarrow$ (α, β) module) $\Rightarrow$ (int $\Rightarrow$ (α, δ) module) $\Rightarrow$
(int $\Rightarrow$ $(\beta \Rightarrow \delta)$) $\Rightarrow$ bool
where graded_module_hom R M M' h
$\equiv \forall n. (h\, n) \in$ hom_module R (M n) (M' (n - 1))

Implementation in Isabelle/HOL

Chain complexes can be implemented using similar structures:

Chain complex

definition chain_complex ::
α *ring* $\Rightarrow$ *(int* $\Rightarrow$ (α, β) *module)* $\Rightarrow$ *(int* $\Rightarrow$ $(\beta \Rightarrow \beta)$*)* $\Rightarrow$ *bool*
where chain_complex R M diff $\equiv$ *graded_module R M*
$\wedge$ *graded_module_hom R M M diff*
$\wedge$ $\forall$*n. (diff (n - 1)* $\circ$ *(diff n))* $= \lambda x.zeroM(n-2)$

Implementation in Coq

First we provide a type definition for non graded structures (for instance, a module):

Type definition

Variable R : ring.

Record module : Type :=
{ crr :> abgroup;

Implementation in Coq

First we provide a type definition for non graded structures (for instance, a module):

Type definition

```
Variable R : ring.

Record module : Type :=
 { crr :> abgroup;
 mult : setoid_bin_op R crr crr;
```

Implementation in Coq

First we provide a type definition for non graded structures (for instance, a module):

Type definition

Variable R : ring.

Record module : Type :=
{ crr :> abgroup;
mult : setoid_bin_op R crr crr;
dist_mult:∀(a:R)(x y: crr),(mult a (x[+]y))[=] ((mult a x)[+](mult a y));
dist_plus:∀ (a b:R)(x:crr), (mult (a[+]b) x)[=]((mult a x)[+](mult b x));
assoc_mult:∀ (a b:R)(x:crr), (mult (a[*]b) x)[=](mult a (mult b x));
unit_mult:∀ x:crr, (mult One x)[=]x }.

Implementation in Coq

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

graded_module := Z → module R

We use a function that maps every integer to a R-module

Implementation in Coq

Now, in order to implement a graded module over a ring R, we can use the following type definition:

Graded module

graded_module := Z $\rightarrow$ module R

We use a function that maps every integer to a R-module

We can also provide a definition for graded module morphisms

Graded module morphism (degree -1)

Variables gm gm': graded_module

graded_module_hom := $\forall i \in$ Z, module_hom (gm i)(gm' $(i-1)$)

Implementation in Coq

Chain complexes can be implemented using similar structures:

Chain complex

```
Record chain_complex : Type :=
 { gm:> graded_module R ;
  diff: graded_module_hom gm gm;
  nilp: ∀ i:Z, ∀ a:(gm i), ((diff(i-1)[oh]diff i) a)[=]
                        (mod_hom_zero (gm i) (gm ((i-1)-1)) a) }.
```

Comparison of both approaches

Representation of graded structures

- Isabelle: explicit domains as sets (or predicates) over a same given type β
- Coq: structures as records with dependent types; different domains as different types

Comparison of both approaches

Representation of graded structures

- Isabelle: explicit domains as sets (or predicates) over a same given type β
- Coq: structures as records with dependent types; different domains as different types

Example

$x_n \in M(n), y_{n+1} \in M(n+1), \{x_n +_{Mn} y_{n+1}\}$ produces:

- A well-typed expression in our Isabelle representation
- A type error in Coq

Comparison of both approaches

This can be sometimes a bit annoying in Coq:

$$diff_{(n+1)}(f_n\, x_n) : M_{((n+1)-1)} \texttt{ but not } M_{(n)}$$

Explicit type conversions are required in order to obtain the expected type

Comparison of both approaches

This can be sometimes a bit annoying in Coq:

$$diff_{(n+1)}(f_n\, x_n) : M_{((n+1)-1)} \texttt{ but not } M_{(n)}$$

Explicit type conversions are required in order to obtain the expected type

Conclusion

1. The richer Coq type theory allows to build precise specifications of graded structures, but some type transformations have to be included.

Comparison of both approaches

This can be sometimes a bit annoying in Coq:

$$diff_{(n+1)}(f_n\, x_n) : M_{((n+1)-1)}\ \texttt{but not}\ M_{(n)}$$

Explicit type conversions are required in order to obtain the expected type

Conclusion

1. The richer Coq type theory allows to build precise specifications of graded structures, but some type transformations have to be included.
2. Isabelle version is more flexible, but demands from the user to ensure the correctness of the expressions provided to the system.

Soundness of the representation

Both in Isabelle and Coq we have been capable of providing (and proving) the existence of structures according to our representation

Soundness of the representation

Both in Isabelle and Coq we have been capable of providing (and proving) the existence of structures according to our representation

Example

The graded module where $\forall n \in \mathbb{Z}$, $M_n = \mathbb{Z}$ and the differentials $d_{n \in \mathbb{Z}} = 0$ form a chain complex

Usefulness of the representation

Both in Isabelle and Coq we have formally proved the *Trivial Perturbation Lemma*, a simplified modification of the *Basic Perturbation Lemma*

Usefulness of the representation

Both in Isabelle and Coq we have formally proved the *Trivial Perturbation Lemma*, a simplified modification of the *Basic Perturbation Lemma*

Proof.

Based on rewriting on graded structures and reduction properties □

Simplicial sets

A simplicial set K consists of a graded set $\{K^q\}_{q\in\mathbb{N}}$, together with *face* and *degeneracy* maps, $\partial_i^q\colon K^q \to K^{q-1}$, $q > 0$, $i \leq q$ and $\eta_i^q\colon K^q \to K^{q+1}$, $q \geq 0$, $i \leq q$ such that:

1. $\partial_i^{q-1}\partial_j^q = \partial_{j-1}^{q-1}\partial_i^q$ if $i < j$
2. $\eta_i^{q+1}\eta_j^q = \eta_{j+1}^{q+1}\eta_i^q$ if $i \leq j$
3. $\partial_i^{q+1}\eta_j^q = \eta_{j-1}^{q-1}\partial_i^q$ if $i < j$
4. $\partial_i^{q+1}\eta_j^q = id$ if $i = j$ or $i = j+1$
5. $\partial_i^{q+1}\eta_j^q = \eta_j^{q-1}\partial_{i-1}^q$ if $i > j+1$

Simplicial sets

A simplicial set K consists of a graded set $\{K^q\}_{q\in\mathbb{N}}$, together with *face* and *degeneracy* maps, $\partial_i^q : K^q \rightarrow K^{q-1}$, $q > 0$, $i \leq q$ and $\eta_i^q : K^q \rightarrow K^{q+1}$, $q \geq 0$, $i \leq q$ such that:

1. $\partial_i^{q-1}\partial_j^q = \partial_{j-1}^{q-1}\partial_i^q$ if $i < j$
2. $\eta_i^{q+1}\eta_j^q = \eta_{j+1}^{q+1}\eta_i^q$ if $i \leq j$
3. $\partial_i^{q+1}\eta_j^q = \eta_{j-1}^{q-1}\partial_i^q$ if $i < j$
4. $\partial_i^{q+1}\eta_j^q = id$ if $i = j$ or $i = j + 1$
5. $\partial_i^{q+1}\eta_j^q = \eta_j^{q-1}\partial_{i-1}^q$ if $i > j + 1$

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

Important example: *universal* simplicial set Δ

- Contains the minimal number of identifications from the equalities

Important example: *universal* simplicial set Δ

- Contains the minimal number of identifications from the equalities
- Any theorem proved on Δ, by using only these identities, will be also true for any other simplicial set.

Important example: *universal* simplicial set Δ

- Contains the minimal number of identifications from the equalities
- Any theorem proved on Δ, by using only these identities, will be also true for any other simplicial set.
- Can be represented by:
  - A q-simplex is a list of elements of length $q + 1$.
  - The face operator ∂_i deletes the i-th element of the list
  - The degeneracy operator η_i repeats the i-th element of the list.

Important example: *universal* simplicial set Δ

- Contains the minimal number of identifications from the equalities
- Any theorem proved on Δ, by using only these identities, will be also true for any other simplicial set.
- Can be represented by:
  - A q-simplex is a list of elements of length $q + 1$.
  - The face operator ∂_i deletes the i-th element of the list
  - The degeneracy operator η_i repeats the i-th element of the list.

Lemma. Second layer

The *universal* simplicial set Δ is a simplicial set.

Important example: *universal* simplicial set Δ

- Contains the minimal number of identifications from the equalities
- Any theorem proved on Δ, by using only these identities, will be also true for any other simplicial set.
- Can be represented by:
  - A q-simplex is a list of elements of length $q+1$.
  - The face operator ∂_i deletes the i-th element of the list
  - The degeneracy operator η_i repeats the i-th element of the list.

Lemma. Second layer

The *universal* simplicial set Δ is a simplicial set.

Canonical representation lemma. First layer

Any simplex l in Δ admits a *unique* representation as a pair of lists (dl, l') where dl a strictly increasing degeneracy list and l' is a list without two equal consecutive elements.

Example: $((3,5,6),(k,t,r,t,l,m))$ represents (k,t,r,r,t,t,t,l,m).

Simplicial set implementation in Isabelle and Coq

Isabelle

```
definition simplicial_set :: "(nat => 'a set) =>
                              (nat => ('a => 'a)) =>
                              (nat => ('a => 'a)) => bool"
  where "simplicial_set K δ μ ==
           (
            (∀q::nat. ∀i≤q. δ i ∈ ((K q) → K (q - 1)))  ∧
            (∀q::nat. ∀i≤q. μ i ∈ ((K q) → K (q + 1)))  ∧
            (∀q::nat. ∀j≤q. ∀i<j. ∀x∈(K q). (δ i (δ j x)) = (δ (j - 1) ((δ i) x))) ∧
            (∀q::nat. ∀j≤q. ∀i≤j. ∀x∈(K q). (μ i (μ j x)) = (μ (j + 1) (μ j x))) ∧
            (∀q::nat. ∀j≤q. ∀i<j. ∀x∈(K q). (δ i (μ j x)) = (μ (j - 1) (δ j x))) ∧
            (∀q::nat. ∀j≤q. ∀i∈{j,j+1}. ∀x∈(K q). (δ i (μ j x)) = x) ∧
            (∀q::nat. ∀j≤q. ∀i>j+1. ∀x∈(K q). (δ i (μ j (δ (i - 1) x)))
            )"
```

Coq

```
Record SimplicialSet: Type:=
{K:> nat -> Type;
Face: forall (q:nat)(i:nat), q>0 -> i<=q -> K q -> K (q-1);
Deg: forall (q:nat)(i:nat), i<=q -> K q -> K (S q);
eq1: forall(q i j:nat)(a:GS q)(p:i<j)(q:j<=q)(k:(q-1)>0),
Face(q:=q-1)(i:=i) k (le_tra' p q)(Face(q:=q)(i:=j)(cS q k) q a)=
Face(q:=q-1)(i:=j-1) k (le_traS q)(Face(q:=q)(i:=i)(cS q k)(le_tra p q)a)
...}.
```

Universal simplicial set implementation in Isabelle and Coq

Isabelle

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

fun μ :: "nat => 'a list => 'a list"
  where
  μ_0: "μ 0 (a # l) = a # a # l"
  | μ_Suc: "μ (Suc n) (a # l) = a # μ n l"

lemma
  μ_permut_a_b:
  assumes a_l_b: "a ≤ b"
  and b_l_l: "b < (length l)"
  shows "μ a (μ b l) = μ (b + 1) (μ a l)"
  using a_l_b and b_l_l
proof (induct a b arbitrary: l rule: diff_induct)
  case (1 a l)
  show "μ a (μ 0 l) = μ (0 + 1) (μ a l)"
    using l
    by (cases l, auto)
next
  case (2 b)
  note Suc_b_g_0 = 2 (1) and Suc_b_l_l = 2 (2)
  show "μ 0 (μ (Suc b) l) = μ (Suc b + 1) (μ 0 l)"
  proof (cases l)
    case Nil show ?thesis using Suc_b_l_l unfolding Nil by auto
  next
    case (Cons a1 l1)
    show "μ (0::nat) (μ (Suc b) l) = μ (Suc b + 1) (μ (0::nat) l)"
      unfolding Cons by auto
  qed
next
  case (3 a b l)
  note hypo = "3.hyps" and Suc_l = 3 (2) and Suc_b_l_l = 3 (3)
  show "μ (Suc a) (μ (Suc b) l) = μ (Suc b + 1) (μ (Suc a) l)"
  proof (cases l)
    case Nil
    show ?thesis using Suc_b_l_l unfolding Nil by simp
  next
    case (Cons a1 l1)
    show "μ (Suc a) (μ (Suc b) l) = μ (Suc b + 1) (μ (Suc a) l)"
      unfolding Cons
      unfolding μ_Suc
      using hypo [of l1]
      using Suc_l and Suc_b_l_l and Cons by auto
  qed
qed
```

Coq

```
Variable A : Type.
Let ListA :=list A.
Let ListN:= list nat.

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

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

Canonical representation lemma in Isabelle and Coq

Isabelle

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

lemma uniqueness:
  assumes can_1: "canonical (d1, l1)"
  and can_2: "canonical (d2, l2)"
  and deg_1_eq_l: "degenerate (d1, l1) = l"
  and deg_2_eq_l: "degenerate (d2, l2) = l"
  shows "(d1, l1) = (d2, l2)"
```

Coq

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

Lemma uniqueness:  forall (l1 l2:ListNxListA)
(l:ListA), canonical l1 -> canonical l2 ->
(degenerate l1)=l -> (degenerate l2)=l ->
l1 = l2
```

Canonical representation lemma in Isabelle and Coq

Isabelle

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

lemma uniqueness:
  assumes can_1: "canonical (d1, l1)"
  and can_2: "canonical (d2, l2)"
  and deg_1_eq_l: "degenerate (d1, l1) = l"
  and deg_2_eq_l: "degenerate (d2, l2) = l"
  shows "(d1, l1) = (d2, l2)"
```

Coq

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

Lemma uniqueness:  forall (l1 l2:ListNxListA)
(l:ListA), canonical l1 -> canonical l2 ->
(degenerate l1)=l -> (degenerate l2)=l ->
l1 = l2
```

Proof.

Using induction on the lists structure and rewriting on the equalities □

Conclusions

- A representation of both Kenzo's data structures layers has been provided in Isabelle/HOL and Coq

Conclusions

- A representation of both Kenzo's data structures layers has been provided in Isabelle/HOL and Coq
- The implementations obtained are sound and useful: we provide instances of the representations and formally prove some results with them

Conclusions

- A representation of both Kenzo's data structures layers has been provided in Isabelle/HOL and Coq
- The implementations obtained are sound and useful: we provide instances of the representations and formally prove some results with them
- The representations illustrate some of the special features of each system

Conclusions

- A representation of both Kenzo's data structures layers has been provided in Isabelle/HOL and Coq
- The implementations obtained are sound and useful: we provide instances of the representations and formally prove some results with them
- The representations illustrate some of the special features of each system

Further work

- Development of more formal proofs (as, for instance, the BPL in the graded case)

Conclusions

- A representation of both Kenzo's data structures layers has been provided in Isabelle/HOL and Coq
- The implementations obtained are sound and useful: we provide instances of the representations and formally prove some results with them
- The representations illustrate some of the special features of each system

Further work

- Development of more formal proofs (as, for instance, the BPL in the graded case)
- Enhancement of the graded structure hierarchy (as, for example, product of graded structures, cone, cone reductions. . .)