Generating Certified Code from Formal Proofs: a Case Study in Homological Algebra[1]

C. Ballarin, J. Rubio, *J. Aransay*

Universidad de La Rioja
Departamento de Matemáticas y Computación

X Jornadas de Programación y Lenguajes, PROLE 2010
Valencia, 8 de Septiembre de 2010

[1]Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the FET program of the European Commission (FP7), STREP project FORMATH, n. 243847.

Introduction

Goals of our research

1. To formalize the proof of the Basic Perturbation Lemma (**BPL**) in Isabelle/HOL.
2. To generate code for the associated algorithm to the BPL with the Isabelle/HOL facilities.

Introduction

Goals of our research

1. To formalize the proof of the Basic Perturbation Lemma (**BPL**) in Isabelle/HOL.
2. To generate code for the associated algorithm to the BPL with the Isabelle/HOL facilities.

In this talk we will focus on the difficulties that had to be overcome to *move* from **goal 1** to **goal 2** above.

Introduction

Goals of our research

1. To formalize the proof of the Basic Perturbation Lemma (**BPL**) in Isabelle/HOL.
2. To generate code for the associated algorithm to the BPL with the Isabelle/HOL facilities.

In this talk we will focus on the difficulties that had to be overcome to *move* from **goal 1** to **goal 2** above.

Main topics

- Representation of algebraic structures.
- Representation of mathematical properties.
- Representation of data structures.

Kenzo

Kenzo is a Computer Algebra system specialised in the field of Homological Algebra. It applies the BPL in the computation of homology groups of chain complexes and differential groups.

Kenzo

Kenzo is a Computer Algebra system specialised in the field of Homological Algebra. It applies the BPL in the computation of homology groups of chain complexes and differential groups.

Isabelle/HOL

Isabelle/HOL is a theorem proving assistant implementing higher-order logic. It includes a tool enabling code generation (from a subset of the specification language including the executable ingredients) to functional programming languages such as SML or Haskell.

Related work

The ideas of getting programs from formal specifications and of formalizing programs are becoming rather popular:

- *ACL2* is a subset (plus something else) of Common Lisp, where proofs about Common Lisp programs can be carried out. More concretely, the *defexec* mechanism allows various "equivalent" algorithms to be used for different purposes.

Related work

The ideas of getting programs from formal specifications and of formalizing programs are becoming rather popular:

- *ACL2* is a subset (plus something else) of Common Lisp, where proofs about Common Lisp programs can be carried out. More concretely, the *defexec* mechanism allows various "equivalent" algorithms to be used for different purposes.
- *ssreflect* is an attempt of communicating non-executable and executable fragments of the Coq world.

Related work

The ideas of getting programs from formal specifications and of formalizing programs are becoming rather popular:

- *ACL2* is a subset (plus something else) of Common Lisp, where proofs about Common Lisp programs can be carried out. More concretely, the *defexec* mechanism allows various "equivalent" algorithms to be used for different purposes.
- *ssreflect* is an attempt of communicating non-executable and executable fragments of the Coq world.
- *Haskabelle* is an embedding of Haskell into Isabelle, where one writes Haskell programs and proves properties of them in Isabelle.

Related work

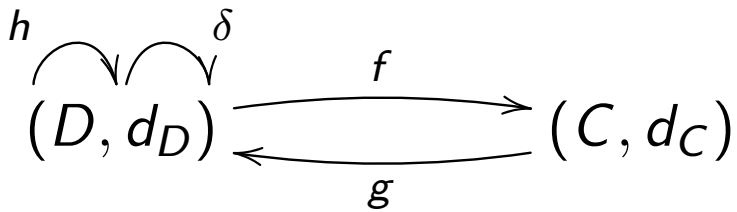
The ideas of getting programs from formal specifications and of formalizing programs are becoming rather popular:

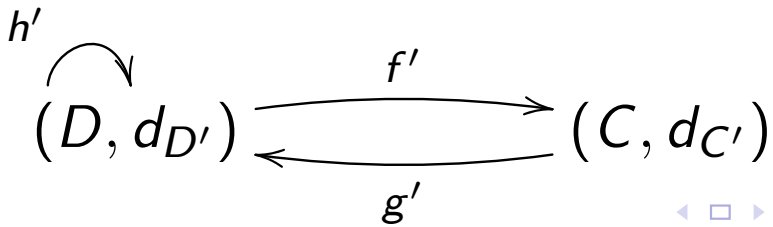
- *ACL2* is a subset (plus something else) of Common Lisp, where proofs about Common Lisp programs can be carried out. More concretely, the *defexec* mechanism allows various "equivalent" algorithms to be used for different purposes.
- *ssreflect* is an attempt of communicating non-executable and executable fragments of the Coq world.
- *Haskabelle* is an embedding of Haskell into Isabelle, where one writes Haskell programs and proves properties of them in Isabelle.
- *Data refinement* is being studied in Isabelle as an extension of the code generation facilities.

Statement of the BPL: structural and analytic part

Theorem

Let $(f, g, h)\colon (D, d_D) \Rightarrow (C, d_C)$ *be a reduction between differential groups and* $\delta\colon D \to D$ *a perturbation of the differential* d_D *satisfying the nilpotency condition w.r.t. the reduction* (f, g, h)*. Then a new reduction* $(f', g', h')\colon (D', d_{D'}) \Rightarrow (C', d_{C'})$ *can be obtained where the underlying abelian groups* D *and* D' *(resp.* C *and* C'*) are the same, but the differentials are perturbed:* $d_{D'} = d_D + \delta$, $d_{C'} = d_C + f\delta_D\psi g$, $f' = f\phi$, $g' = \psi g$, $h' = h\phi$, $\phi = \sum_{i=0}^{\infty}(-1)^i(\delta h)^i$, *and* $\psi = \sum_{i=0}^{\infty}(-1)^i(h\delta)^i$.





Main features of the proof in Isabelle

Algebraic structures, following the ideas in [Ballarin et al, 2010][2], are implemented by means of:

- Types as *extensible records* with *explicit* domains or carriers (given by an explicit predicate or characteristic function);
- Specifications with *locales* to introduce their properties.

[2] Ballarin et al., *The Isabelle/HOL Algebra Library*.

Main features of the proof in Isabelle

Algebraic structures, following the ideas in [Ballarin et al, 2010][2], are implemented by means of:

- Types as *extensible records* with *explicit* domains or carriers (given by an explicit predicate or characteristic function);
- Specifications with *locales* to introduce their properties.

Other possibilities in HOL

- The use of *types*, instead of sets, for representing the domains.
- The use of *type classes*.
- ...

[2] Ballarin et al., *The Isabelle/HOL Algebra Library*.

Record types with explicit sets as domains are the best option, since they enable:

- A unified treatment of basic algebraic structures and algebraic structures defined over endomorphisms or homomorphisms.
- An easy way to interplay with algebraic structures and their subsets.

Data type definition

```
record 'a monoid =
  carrier :: "'a set"
  mult    :: "['a, 'a] ⇒ 'a" (infixl "⊗ı" 70)
  one     :: 'a ("𝟭ı")
```

Data type definition

```
record 'a monoid =
  carrier :: "'a set"
  mult    :: "['a, 'a] ⇒ 'a" (infixl "⊗ı" 70)
  one     :: 'a ("1ı")
```

Formal specification

```
locale monoid =
  fixes G (structure)
  assumes m_closed [intro, simp]:
        "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊗ y ∈ carrier G"
      and m_assoc:
        "⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧
         ⟹ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
      and one_closed [intro, simp]: "1 ∈ carrier G"
      and l_one [simp]: "x ∈ carrier G ⟹ 1 ⊗ x = x"
      and r_one [simp]: "x ∈ carrier G ⟹ x ⊗ 1 = x"
```

Unfortunately, the previous representation does not provide a direct way for code generation (this representation admits non-computable sets). Some possible solutions:

Unfortunately, the previous representation does not provide a direct way for code generation (this representation admits non-computable sets). Some possible solutions:

- By means of a characteristic function.
- By means of a list (finite sets).
- By means of type classes.

Type classes:

Type classes are a feature of Haskell specially introduced for allowing ad-hoc overloading, abstract specifications and modular reasoning [Haftmann & Wenzel, 2007][3].

Advantages of using Isabelle implementation of type classes:

- Algebraic structures become types (the family of types satisfying the given signature and specification).
- Instances of a type class can be stated and proved inside of Isabelle/HOL.

[3] Haftmann and Wenzel, "Constructive Type Classes in Isabelle".

Type classes:

Type classes are a feature of Haskell specially introduced for allowing ad-hoc overloading, abstract specifications and modular reasoning [Haftmann & Wenzel, 2007][3].

Advantages of using Isabelle implementation of type classes:

- Algebraic structures become types (the family of types satisfying the given signature and specification).
- Instances of a type class can be stated and proved inside of Isabelle/HOL.

By means of the instances, we thus certify the input data of the certified programs.

In addition to certified programs, we also obtain certified inputs.

[3] Haftmann and Wenzel, "Constructive Type Classes in Isabelle".

Example (Type class definition, instantiation and execution)

```
class monoid_class = type +
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70)
  and one :: 'a ("1")
  assumes assoc: "x ⊗ y ⊗ z = x ⊗ (y ⊗ z)"
  and neutl: "1 ⊗ x = x"
  and neutr: "x ⊗ 1 = x"
```

Example (Type class definition, instantiation and execution)

```
class monoid_class = type +
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70)
  and one :: 'a ("1")
  assumes assoc: "x ⊗ y ⊗ z = x ⊗ (y ⊗ z)"
  and neutl: "1 ⊗ x = x"
  and neutr: "x ⊗ 1 = x"

instance nat :: monoid_class and int :: monoid_class
  mult_nat_def: "m ⊗ n ≡ m * n"
  one_nat_def: "1 ≡ (1::nat)"
  mult_int_def: "m ⊗ n ≡ m + n"
  one_int_def:  "1 ≡ (0::int)"
proof
  fix m n l :: nat
  from mult_nat_def show "m ⊗ n ⊗ l = m ⊗ (n ⊗ l)" by simp
  from mult_nat_def one_nat_def show "1 ⊗ n = n" by simp
  from mult_nat_def one_nat_def show "n ⊗ 1 = n" by simp
next
  fix i j k :: int
  from mult_int_def show "i ⊗ j ⊗ k = i ⊗ (j ⊗ k)" by simp
  from mult_int_def one_int_def show "1 ⊗ i = i" by simp
  from mult_int_def one_int_def show "i ⊗ 1 = i" by simp
qed
```

Example (Type class definition, instantiation and execution)

```
class monoid_class = type +
  fixes mult :: "'a ⇒ 'a ⇒ 'a" (infixl "⊗" 70)
  and one :: 'a ("1")
  assumes assoc: "x ⊗ y ⊗ z = x ⊗ (y ⊗ z)"
  and neutl: "1 ⊗ x = x"
  and neutr: "x ⊗ 1 = x"

instance nat :: monoid_class and int :: monoid_class
  mult_nat_def: "m ⊗ n ≡ m * n"
  one_nat_def: "1 ≡ (1::nat)"
  mult_int_def: "m ⊗ n ≡ m + n"
  one_int_def:  "1 ≡ (0::int)"
proof
  fix m n l :: nat
  from mult_nat_def show "m ⊗ n ⊗ l = m ⊗ (n ⊗ l)" by simp
  from mult_nat_def one_nat_def show "1 ⊗ n = n" by simp
  from mult_nat_def one_nat_def show "n ⊗ 1 = n" by simp
next
  fix i j k :: int
  from mult_int_def show "i ⊗ j ⊗ k = i ⊗ (j ⊗ k)" by simp
  from mult_int_def one_int_def show "1 ⊗ i = i" by simp
  from mult_int_def one_int_def show "i ⊗ 1 = i" by simp
qed

definition one_times_one_int_def: "one_times_one_int == (1::int) ⊗ 1"
definition one_times_one_nat_def: "one_times_one_nat == (1::nat) ⊗ 1"

code_gen one_times_one_int one_times_one_nat in SML
ML "ROOT.Group_instance.one_times_one_int"
ML "ROOT.Group_instance.one_times_one_nat"
```

A couple of remarks in type classes

- The proof of the BPL has to be translated into the type classes representation of algebraic structures (type classes are converted into a particular case of *records*). Here a certain degree of automation should be possible.
- The original proof was not possible in this setting (no subsets notion).

A couple of remarks in type classes

- The proof of the BPL has to be translated into the type classes representation of algebraic structures (type classes are converted into a particular case of *records*). Here a certain degree of automation should be possible.
- The original proof was not possible in this setting (no subsets notion).

A limitation of type classes *à la* Isabelle/HOL:

Type classes have to be single parameterised.

Definition of mathematical properties

Representation of the local nilpotency condition:

locale *local-nilpotent-term-existential* = *ring-endomorphisms D R* + *var a* +
  assumes *a-in-R*: $a \in carrier\ R$
  and *a-local-nilpot*: $\forall\, x{\in}carrier\ D.\ \exists\, n{::}nat.\ (a\ (\hat{})_R\ n)\ x = \mathbf{1}_D$
  fixes *deg-of-nilpot*
  defines *deg-of-nilpot-def*: $deg\text{-}of\text{-}nilpot == (\lambda x.\ (LEAST\ n.\ (a\ (\hat{})_R\ (n{::}nat))\ x = \mathbf{1}_D))$

definition (**in** *local-nilpotent-term-existential*)
  $power\text{-}series\ x == finprod\ D\ (\lambda i{::}nat.\ (a(\hat{})_R\ i)\ x)\ \{..deg\text{-}of\text{-}nilpot\ x\}$

Definition of mathematical properties

Representation of the local nilpotency condition:

locale *local-nilpotent-term-existential* = *ring-endomorphisms D R* + *var a* +
  assumes *a-in-R*: $a \in carrier\ R$
  and *a-local-nilpot*: $\forall\, x{\in}carrier\ D.\ \exists\, n{::}nat.\ (a\ (\hat{})_R\ n)\ x = \mathbf{1}_D$
  fixes *deg-of-nilpot*
  defines *deg-of-nilpot-def*: $deg\text{-}of\text{-}nilpot == (\lambda x.\ (LEAST\ n.\ (a\ (\hat{})_R\ (n{::}nat))\ x = \mathbf{1}_D))$

definition (**in** *local-nilpotent-term-existential*)
  $power\text{-}series\ x == finprod\ D\ (\lambda i{::}nat.\ (a(\hat{})_R\ i)\ x)\ \{..deg\text{-}of\text{-}nilpot\ x\}$

The previous specifications of *deg-of-nilpot* and thus of *power-series* cannot be directly *code-generated* in Isabelle/HOL (they are based on the Hilbert's ϵ-operator).

locale *local-nilpotent-term-existential* = *ring-endomorphisms D R* + *var a* +
  assumes *a-in-R*: $a \in carrier\ R$
  and *a-local-nilpot*: $\forall\, x{\in}carrier\ D.\ \exists\, n{::}nat.\ (a\ (\hat{})_R\ n)\ x = \mathbf{1}_D$
  fixes *deg-of-nilpot*
  defines *deg-of-nilpot-def*: $deg\text{-}of\text{-}nilpot == (\lambda x.\ (LEAST\ n.\ (a\ (\hat{})_R\ (n{::}nat))\ x = \mathbf{1}_D))$

definition (**in** *local-nilpotent-term-existential*)
  $power\text{-}series\ x == finprod\ D\ (\lambda i{::}nat.\ (a(\hat{})_R\ i)\ x)\ \{..deg\text{-}of\text{-}nilpot\ x\}$

locale *local-nilpotent-term-existential* = *ring-endomorphisms D R* + *var a* +
  assumes *a-in-R*: $a \in carrier\ R$
  and *a-local-nilpot*: $\forall\, x{\in}carrier\ D.\ \exists\, n{::}nat.\ (a\ (\hat{})_R\ n)\ x = \mathbf{1}_D$
  fixes *deg-of-nilpot*
  defines *deg-of-nilpot-def*: $deg\text{-}of\text{-}nilpot == (\lambda x.\ (LEAST\ n.\ (a\ (\hat{})_R\ (n{::}nat))\ x = \mathbf{1}_D))$

definition (**in** *local-nilpotent-term-existential*)
  $power\text{-}series\ x == finprod\ D\ (\lambda i{::}nat.\ (a(\hat{})_R\ i)\ x)\ \{..deg\text{-}of\text{-}nilpot\ x\}$

How can we turn this into an executable specification?

constdefs
  $local\text{-}bound\text{-}gen\ f\ (x{::}'a{::}ab\text{-}group\text{-}class)\ (n{::}nat) \equiv For\ (\lambda\ y.\ y \neq \mathbf{1}_D)\ f\ (\lambda\ y\ n.\ n{+}1)\ x\ n$
  $local\text{-}bound\ f\ (x{::}'a{::}ab\text{-}group\text{-}class) \equiv local\text{-}bound\text{-}gen\ f\ x\ 0$

locale *local-nilpotent-term-existential* = *ring-endomorphisms D R* + *var a* +
assumes *a-in-R*: $a \in carrier\ R$
and *a-local-nilpot*: $\forall x{\in}carrier\ D.\ \exists n{::}nat.\ (a\ (\hat{})_R\ n)\ x = \mathbf{1}_D$
fixes *deg-of-nilpot*
defines *deg-of-nilpot-def*: $deg\text{-}of\text{-}nilpot == (\lambda x.\ (LEAST\ n.\ (a\ (\hat{})_R\ (n{::}nat))\ x = \mathbf{1}_D))$

definition (**in** *local-nilpotent-term-existential*)
$power\text{-}series\ x == finprod\ D\ (\lambda i{::}nat.\ (a(\hat{})_R\ i)\ x)\ \{..deg\text{-}of\text{-}nilpot\ x\}$

How can we turn this into an executable specification?

constdefs
$local\text{-}bound\text{-}gen\ f\ (x{::}'a{::}ab\text{-}group\text{-}class)\ (n{::}nat) \equiv For\ (\lambda\ y.\ y \neq \mathbf{1}_D)\ f\ (\lambda\ y\ n.\ n{+}1)\ x\ n$
$local\text{-}bound\ f\ (x{::}'a{::}ab\text{-}group\text{-}class) \equiv local\text{-}bound\text{-}gen\ f\ x\ 0$

Do we know a way to turn a while loop into a (tail) recursive function?

function $(tailrec)\ While :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where $While\ continue\ f\ s = (if\ continue\ s\ then\ While\ continue\ f\ (f\ s)\ else\ s)$

Code generation from definitions

The following ML code is the one obtained from the previous definitions:

```
structure While =
struct

fun whilea continue f s = (if continue s then whilea continue f (f s) else s);

fun for' continue f aca x ac =
  whilea (fn a as (acb, aa) => continue aa)
    (fn a as (acb, xa) => (aca xa acb, f xa)) (ac, x);

fun for continue f aca x ac = Product_Type.fst (for' continue f aca x ac);

end; (*struct While*)

fun local_bound_gen (A1_, A2_) f x n =
  While.for
    (fn y =>
      not (Code_Generator.op_eq A1_ y
            (zero (ab_group_class_ab_monoid_class A2_))))
    f (fn y => fn na => (+ na 1) x n;

fun local_bound (A1_, A2_) f x = local_bound_gen (A1_, A2_) f x 0;
```

The **function** Isabelle package is due to A. Krauss [Krauss 2006][4], and S. Obua [Obua 2007][5] introduced definitions for *While* and *For* loops.

[4] Krauss, "Partial Recursive Functions in Higher-Order Logic".

[5] Obua, "Looping around the Orbit".

Execution of the programs: An example with bicomplexes

The reduction is given by the following elements: D can be understood as integer coefficients over $\mathbb{N} \times \mathbb{N}$, $C = 0$, $f = 0$, $g = 0$ and d, h, δ as depicted below.

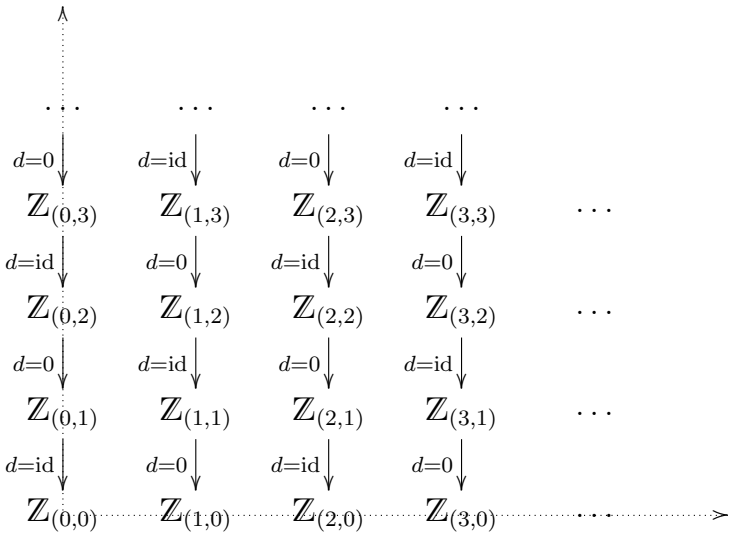


Figure: Definition of the differential d of a bicomplex.

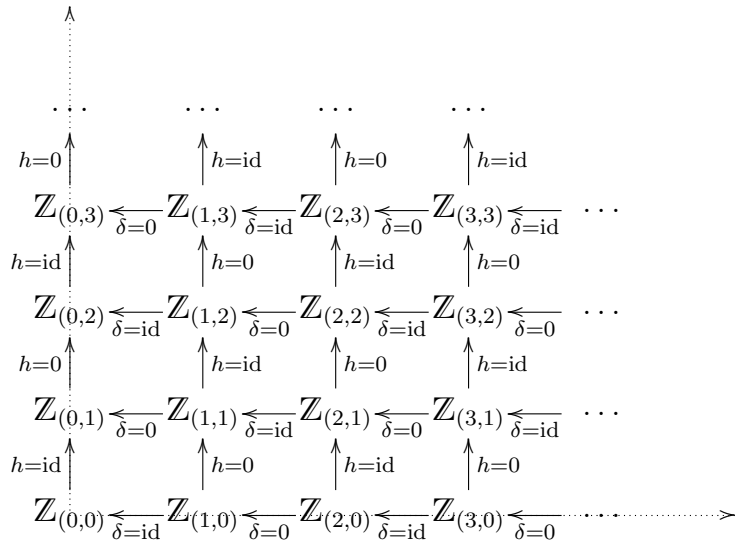


Figure: Definition of the homotopy operator h and the perturbation δ of a bicomplex.

A word on the implementation

D is implemented as integer matrices (see [Obua & Nipkow, 2010][6]).

Abstract Matrices

Matrices had to be *proved* an instance of an algebraic structure appearing in the BPL statement. The type definition that we (successfully) used was

$$\{f : \mathbb{N} \times \mathbb{N} \to \alpha | \ \mathit{finite}\,(\mathit{nonzero\_positions}\ f)\}$$

[6] Obua and Nipkow, "Flyspeck II: the basic linear programs".

A word on the implementation

D is implemented as integer matrices (see [Obua & Nipkow, 2010][6]).

Abstract Matrices

Matrices had to be *proved* an instance of an algebraic structure appearing in the BPL statement. The type definition that we (successfully) used was

$$\{f : \mathbb{N} \times \mathbb{N} \to \alpha | \; \mathit{finite}\,(\mathit{nonzero\_positions}\; f)\}$$

Sparse Matrices

Computations with abstract matrices were unfeasible. A different representation of matrices had to be figured out, fitting in the scope of the code generation facility. For instance:

$$\alpha\; \mathit{spvec} = (\mathit{nat} * \alpha)\; \mathit{list}$$
$$\alpha\; \mathit{spmat} = (\alpha\; \mathit{spvec})\; \mathit{spvec}$$

[6] Obua and Nipkow, "Flyspeck II: the basic linear programs".

How are both representations communicated?

A collection of lemmas proving that *operations* over the abstract representation $+, \ldots$ are equal to *some operations* over the sparse one has to be provided:

lemma `(sparse_row_matrix A) + (sparse_row_matrix B) = sparse_row_matrix (add_spmat (A, B))`

Nevertheless, the properties proved over *abstract matrices* have not been proved over *sparse matrices*.

The previous *datatype conversion* has two remarkable features:

The previous *datatype conversion* has two remarkable features:

- It is done inside of the Isabelle/HOL framework, formally verified.

The previous *datatype conversion* has two remarkable features:

- It is done inside of the Isabelle/HOL framework, formally verified.
- The representation based on lists does not satisfy the *good* properties of the first one (for instance, the zero matrix has multiple representations as list of lists of zeros, whereas in the first one representations where unique *w.r.t.* the extensional equality for functions).

Conclusions

- Algebraic structures admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.

Conclusions

- Algebraic structures admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.
- Mathematical specifications (and sometimes proofs) do not usually pay attention to constructive matters, even within the scope of algorithmic.

Conclusions

- Algebraic structures admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.
- Mathematical specifications (and sometimes proofs) do not usually pay attention to constructive matters, even within the scope of algorithmic.
- Mathematical structures (polynomials, matrices, finite sets) admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.

Conclusions

- Algebraic structures admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.
- Mathematical specifications (and sometimes proofs) do not usually pay attention to constructive matters, even within the scope of algorithmic.
- Mathematical structures (polynomials, matrices, finite sets) admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.

Some ideas

- To produce *minimal* representations (in the number of operations or properties stated) of mathematical structures and to define *embeddings, or functors* from every other representation to such one.

Conclusions

- Algebraic structures admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.
- Mathematical specifications (and sometimes proofs) do not usually pay attention to constructive matters, even within the scope of algorithmic.
- Mathematical structures (polynomials, matrices, finite sets) admit a wide range of representations in theorem provers; the choice usually depends on the aim of the implementation. Proof reusing among representations should be improved.

Some ideas

- To produce *minimal* representations (in the number of operations or properties stated) of mathematical structures and to define *embeddings, or functors* from every other representation to such one.
- Suggestions welcome!!