Living (*i.e.* executing and proving) with different representations of datatypes

Jesús Aransay[1]

Universidad de La Rioja
Departamento de Matemáticas y Computación

Algebric computing, soft computing and program verification
Castro Urdiales, 21-23 April, 2010

[1]Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Commission FP7, STREP project ForMath.

Contents of the talk:

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Some facts

The Isabelle/HOL Library offers:

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Some facts

The Isabelle/HOL Library offers:

- At least three different representations of *univariate polynomials*.

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Some facts

The Isabelle/HOL Library offers:

- At least three different representations of *univariate polynomials*.
- At least two different representations of *matrices* (used in the proof of the Kepler conjecture).

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Some facts

The Isabelle/HOL Library offers:

- At least three different representations of *univariate polynomials*.
- At least two different representations of *matrices* (used in the proof of the Kepler conjecture).
- Two different representations of algebraic structures, by means of *records* + *locales* and by means of type classes.

Introduction

Goals of our research

1. To ease the communication between data type representations.
2. To *reuse* proofs carried out for a datatype representation to different representations of that datatype.

Some facts

The Isabelle/HOL Library offers:

- At least three different representations of *univariate polynomials*.
- At least two different representations of *matrices* (used in the proof of the Kepler conjecture).
- Two different representations of algebraic structures, by means of *records* + *locales* and by means of type classes.
- At least two different representations of *natural numbers* (with binary representation and the usual inductive one).

Our original problem

Abstract Matrices

In our experiments for code generation of the *Basic Perturbation Lemma* (*BPL*) in Isabelle, matrices had to be *proved* an instance of an algebraic structure appearing in the BPL statement.
The type definition that we (successfully) used was

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

Our original problem

Abstract Matrices

In our experiments for code generation of the *Basic Perturbation Lemma* (*BPL*) in Isabelle, matrices had to be *proved* an instance of an algebraic structure appearing in the BPL statement.
The type definition that we (successfully) used was

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

Sparse Matrices

Direct code generation from such abstract matrices is *not possible* (with the current Isabelle/HOL technology). *Computations* with matrices were unfeasible. A different representation of matrices had to be figured out, fitting in the scope of the code generation facility. For instance:

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

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*.

Some ideas on proof reusing and datatype representation

- Proof reusing has been already dealt with in Isabelle (Johnsen and Lüth), based on *signature morphisms*. Some of their ideas will be useful, even if they heavily rely on identifying data type constructors of the different representations.

Some ideas on proof reusing and datatype representation

- Proof reusing has been already dealt with in Isabelle (Johnsen and Lüth), based on *signature morphisms*. Some of their ideas will be useful, even if they heavily rely on identifying data type constructors of the different representations.
- *Locale interpretation* (Ballarin) is also a way to *transfer* theorems between the different *interpretations* of an abstract structure.

Some ideas on proof reusing and datatype representation

- Proof reusing has been already dealt with in Isabelle (Johnsen and Lüth), based on *signature morphisms*. Some of their ideas will be useful, even if they heavily rely on identifying data type constructors of the different representations.
- *Locale interpretation* (Ballarin) is also a way to *transfer* theorems between the different *interpretations* of an abstract structure.
- Theory morphisms (mappings between the *domains* and the *operations*) will be also helpful.

Abstract polynomials (over a given ring R)

definition *up* :: (α, β) *ring_scheme* $\Rightarrow$ *(nat* $\Rightarrow$ α*) set*
where *up R* $\equiv$ *{f. f* $\in$ *UNIV* $\rightarrow$ *carrier R & (EX n. bound* $\mathbf{0}_R$ *n f)}*

Features

- Uniqueness of representation (*w.r.t.* the extensional equality) $(\lambda x.0_R)$.
- Natural definition of operations $p + q = (\lambda n.\, p\, n + q\, n)$.
- Code generation not possible.

Abstract polynomials (over a given ring R)

definition *up :: (α, β) ring_scheme $\Rightarrow$ (nat $\Rightarrow$ α) set*
where *up R $\equiv$ {f. f $\in$ UNIV $\rightarrow$ carrier R & (EX n. bound $\mathbf{0}_R$ n f)}*

Features

- Uniqueness of representation (*w.r.t.* the extensional equality) ($\lambda x.0_R$).
- Natural definition of operations $p + q = (\lambda n.\, p\, n + q\, n)$.
- Code generation not possible.

Sparse polynomials

types *α pair = (α::ring * nat)*
types *α sppol = α pair list*

Features

- Not uniqueness of representation
  $[(1 :: int, 1)] = [(0 :: int, 0), (1, 1)] = [(0 :: int, 2), (1, 1)]$.
- Direct code generation.

Definition of the invariants

- The invariant in the abstract representation will be *every function* (with finite domain).
- The invariant in the sparse representation will be:
  definition *canonical:: (α::ring) sppol* $\Rightarrow$ *bool*
  where *canonical ms* $\equiv$ *sparse ms* $\wedge$ *ssorted ms*

Definition of the invariants

- The invariant in the abstract representation will be *every function* (with finite domain).
- The invariant in the sparse representation will be:
  definition *canonical:: (α::ring) sppol ⇒ bool*
  where *canonical ms ≡ sparse ms ∧ ssorted ms*

Definition of the *representation* and *abstraction* functions

- **fun** *abstr :: α::ring sppol ⇒ α up* **where**
  abstr_Nil: abstr [] = 0
  | abstr_Cons: abstr((i, c) # ms) = monom i c + abstr ms
- **definition** *repr :: (α::ring) up ⇒ α sppol* **where**
  repr p = (THE ms. canonical ms ∧ (∀c::nat. coeff_sppol ms c = coeff p c))

Definition of the invariants

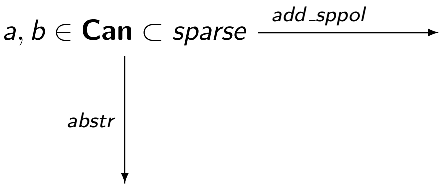
- The invariant in the abstract representation will be *every function* (with finite domain).
- The invariant in the sparse representation will be:
  definition *canonical:: (α::ring) sppol ⇒ bool*
  where *canonical ms ≡ sparse ms ∧ ssorted ms*

Definition of the *representation* and *abstraction* functions

- **fun** *abstr :: α::ring sppol ⇒ α up* **where**
  abstr_Nil: abstr [] = 0
  | abstr_Cons: abstr((i, c) # ms) = monom i c + abstr ms
- **definition** *repr :: (α::ring) up ⇒ α sppol* **where**
  repr p = (THE ms. canonical ms ∧ (∀ c::nat. coeff_sppol ms c = coeff p c))

The only requirement that *must* be satisfied by both representations is a *coefficient* operation (*coeff* or *coeff_sppol*), a constructor *monom* over the abstract representation, and an inductive definition over the sparse one.

- Define operations (*add_sppol*, *mult_sppol*, ...) over the sparse representation for addition, multiplication, ...



- Define operations (*add_sppol*, *mult_sppol*, ...) over the sparse representation for addition, multiplication, ...
- These operations have to be proved *closed* over the invariant.

$$a, b \in \mathbf{Can} \subset sparse \xrightarrow{add\_sppol} add\_sppol(a, b) \in \mathbf{Can} \subset sparse$$

$$\Big\downarrow abstr$$

$$abstr\ a, abstr\ b \in \mathbf{Abs}$$

- Define operations (*add_sppol*, *mult_sppol*, ...) over the sparse representation for addition, multiplication, ...
- These operations have to be proved *closed* over the invariant.
- They have to be proved equal to the *abstract* ones.

$$
\begin{array}{ccc}
a, b \in \mathbf{Can} \subset sparse & \xrightarrow{add\_sppol} & add\_sppol(a, b) \in \mathbf{Can} \subset sparse \\
\Big\downarrow abstr & & \Big\downarrow abstr \\
abstr\ a, abstr\ b \in \mathbf{Abs} & \xrightarrow{+} & abstr\ a + abstr\ b \stackrel{eqv}{=} abstr(add\_sppol(a, b))
\end{array}
$$

- Define operations (*add_sppol*, *mult_sppol*, ...) over the sparse representation for addition, multiplication, ...
- These operations have to be proved *closed* over the invariant.
- They have to be proved equal to the *abstract* ones.

$$\begin{array}{ccc}
a, b \in \mathbf{Can} \subset \mathit{sparse} & \xrightarrow{\mathit{add_sppol}} & \mathit{add_sppol}(a, b) \in \mathbf{Can} \subset \mathit{sparse} \\
\Big\downarrow \mathit{abstr} & & \Big\downarrow \mathit{abstr} \\
\mathit{abstr}\ a, \mathit{abstr}\ b \in \mathbf{Abs} & \xrightarrow{+} & \mathit{abstr}\ a + \mathit{abstr}\ b \overset{\mathit{eqv}}{=} \mathit{abstr}(\mathit{add_sppol}(a, b))
\end{array}$$

Prove the following lemma:

lemma `id_ra:` **assumes** `canonical a` **shows** `repr (abstr a) = a`

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
```

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
  have add_sppol a b = repr (abstr(add_sppol a b)) using id_ra
```

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
  have add_sppol a b = repr (abstr(add_sppol a b)) using id_ra
  also have ... = repr (abstr a + abstr b) using eqv
```

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
  have add_sppol a b = repr (abstr(add_sppol a b)) using id_ra
  also have ... = repr (abstr a + abstr b) using eqv
  also have  ... = repr (abstr b + abstr a) by abstract_result
```

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
  have add_sppol a b = repr (abstr(add_sppol a b)) using id_ra
  also have ... = repr (abstr a + abstr b) using eqv
  also have  ... = repr (abstr b + abstr a) by abstract_result
  also have ... = repr (abstr(add_sppol b a)) using eqv [symm]
```

Proof development

lemma assumes *canonical a* **and** *canonical b*
shows *add_sppol a b = add_sppol b a*
proof –
  have *add_sppol a b = repr (abstr(add_sppol a b))* **using** *id_ra*
  also have *... = repr (abstr a + abstr b)* **using** *eqv*
  also have *... = repr (abstr b + abstr a)* **by** *abstract_result*
  also have *... = repr (abstr(add_sppol b a))* **using** *eqv [symm]*
  also have *... = add_sppol b a* **using** *id_ra [symm]*

Proof development

```
lemma assumes canonical a and canonical b
shows add_sppol a b = add_sppol b a
proof -
  have add_sppol a b = repr (abstr(add_sppol a b)) using id_ra
  also have ... = repr (abstr a + abstr b) using eqv
  also have  ... = repr (abstr b + abstr a) by abstract_result
  also have ... = repr (abstr(add_sppol b a)) using eqv [symm]
  also have  ... = add_sppol b a using id_ra [symm]
  finally show ?thesis by simp
qed
```

Some other use cases:

- A different representation of polynomials, based on *dense lists*. For instance, $[0, 1, 1]$ represents $x + x^2$.
  definition *canonical :: α::{semiring_0,ring} list $\Rightarrow$ bool*
  where *canonical ps $\equiv$ (pnormalize p = p)*

Some other use cases:

- A different representation of polynomials, based on *dense lists*. For instance, $[0, 1, 1]$ represents $x + x^2$.
  definition *canonical :: α::{semiring_0,ring} list $\Rightarrow$ bool*
  where *canonical ps $\equiv$ (pnormalize p = p)*
- The representation of matrices that we introduced at the beginning of the talk.
  definition *canonical :: α::zero spmat $\Rightarrow$ bool* **where**
  canonical x $\equiv$ sorted_sparse_matrix x $\wedge$ mnormalized x

  definition *repr :: α matrix $\Rightarrow$ α::{ring} spmat* **where**
  repr A $\equiv$ (THE x. canonical x $\wedge$ ($\forall$ m n. coeff_spmat x m n = coeff A m n))

Results obtained

We have reused the proofs of the abstract representation for the polynomials to prove that sparse and dense polynomials are a *commutative ring and a domain*.

We have reused the proofs of the abstract representation of matrices to prove that sparse matrices are a commutative group (*w.r.t.* addition) and multiplication is associative and distributive *w.r.t.* addition. There is no unit for multiplication.

Results obtained

We have reused the proofs of the abstract representation for the polynomials to prove that sparse and dense polynomials are a *commutative ring and a domain*.

We have reused the proofs of the abstract representation of matrices to prove that sparse matrices are a commutative group (*w.r.t.* addition) and multiplication is associative and distributive *w.r.t.* addition. There is no unit for multiplication.

Drawbacks

Operations over the *executable* representations must be *closed w.r.t.* the invariant, which means that they can be really slow from an efficiency point of view.

The previous infrastructure can be enriched with some further ideas:

- Define a minimal representation of the data type, that allows carrying out proofs.

  locale *polynomials* =
  fixes *R* **assumes** *ring R*
  fixes *pol_equal* :: $\beta \Rightarrow \beta \Rightarrow$ *bool*
  and *zero* :: β **and** *coeff* :: $\beta \Rightarrow$ *nat* $\Rightarrow \alpha$ **and** *bound* :: $\beta \Rightarrow$ *nat*
  defines *pol_equal* $\equiv$ (λp q. ($\forall$n. coeff p n = coeff q n))
  assumes $\forall$p n. coeff p n $\in$ carrier R
  and $\forall$n::nat. coeff zero n = $\mathbf{0}_R$
  and bound zero = 0
  and $\forall$p. $\exists$n. bound p = n
  and $\forall$m. bound p < m $\longrightarrow$ coeff p m = $\mathbf{0}_R$
  fixes *add_monom*:: $\beta \Rightarrow$ nat $\Rightarrow \alpha \Rightarrow \beta$
  assumes *add_monom_coeff*: $\forall$n::nat. coeff (add_monom p m x) n = (if n = m then (coeff p m $\oplus_R$ x) else coeff p n)

- Embedding the *abstract* representation into the minimal one.
- Embedding the *executable* representation into the abstract one.

Conclusions and further work

- In the Isabelle/HOL type system sets are not first order citizens. Direct code generation from them is unfeasible, as well as defining signature morphisms based on type constructors.

Conclusions and further work

- In the Isabelle/HOL type system sets are not first order citizens. Direct code generation from them is unfeasible, as well as defining signature morphisms based on type constructors.
- Formal proofs and code generation posse different challenges, and demand different solutions/implementations.

Conclusions and further work

- In the Isabelle/HOL type system sets are not first order citizens. Direct code generation from them is unfeasible, as well as defining signature morphisms based on type constructors.
- Formal proofs and code generation posse different challenges, and demand different solutions/implementations.
- Proof reusing in that field can be achieved, but code efficiency also has to be preserved.