Final Report for Deliverable Nr. 1.4

Test of Tactics Using the Work on Reflection

Responsible: Georges Gonthier
Written by: Georges Gonthier, Georges.Gonthier@inria.fr
and Yves Bertot, Yves.Bertot@inria.fr
INRIA, France

Deliverable Date: February 2011

1 Overview

In the conext of formal proof systems, *reflection* means using some formal representation of the *text* used to formalize mathematical objects. For example to prove facts about the group element yxx^{-1}, which we write `y * x * x ^-1` in Coq, it might be astute to reason about its syntactic representation, which we would write

```
Mul (Mul (Cst 1)(Cst 0))(Inv (Cst 0))
```

using the Coq datatype

```
Inductive term :=
  | Cst of nat
  | Idx
  | Inv of term
  | Exp of term & nat
  | Mul of term & term
  | Conj of term & term
  | Comm of term & term.
```

For example, we could then automatically determine that yxx^{-1} does not depend on x by a simple computation on this data structure. We can recover the group element with a simple interpreter

```
Fixpoint eval {gT} e t : gT :=
  match t with
  | Cst i => nth 1 e i
  | Idx => 1
  | Inv t1 => (eval e t1)^-1
  | Exp t1 n => eval e t1 ^+ n
  | Mul t1 t2 => eval e t1 * eval e t2
```

```
  | Conj t1 t2 => eval e t1 ^ eval e t2
  | Comm t1 t2 => [~ eval e t1, eval e t2]
  end.
```

However, reflection is not very appealing if the user has to enter manually large syntax trees as above. Indeed Coq provides some direct support for reflection in the form of specialized tactics (`quote`, `ring`) and the $\mathcal{L}$-tac tactical metalanguage. This has the limitations that its is very syntactic and operational. This makes the use of reflection sensitive to small changes in the formalization; for example the `quote` tactic would disallow the `eval` function above, and the `ring` tactic cannot be used with the SSReflect generic rings. With $\mathcal{L}$-tac based reflection bespoke tactics encode large parts of the formalization; these cannot be composed (except sequentially) and their semantics is informal.

Our object in this task was to evaluate ways of exploiting reflection using *standard* tactics. Since these can only be controlled by supplying object and proof terms, or deduction or type inference hints, we had to devise ways of expressing reflection by these means only, somehow weaving reflection into the interpretation of Coq terms and formulae. We will refer to this approach as *logic*-based reflection.

Small-scale reflection, the proof paradigm that gave it name to the SSReflect extension, is perhaps the simplest example of this approach. It is based on the observation that quite often the natural description of a logical object is syntactic: examples includes booleans, Peano integers, lists and various symbolic expressions. In those cases reflection is trivial, as the formal and syntactic representations coincide, and can be used immediately for all recursively defined predicates and functions. This has been used extensively in the SSReflect library to deal with trivial propositional and arithmetic identities, and has fared quite well in practice — it was instrumental in completing the proof of the Four Colour theorem.

In this task we assess how well techniques that go beyond small scale reflection fare *in practice*. The first two techniques we consider, for doing reflection during parsing or type inference, respectively, were more or less known. What we have done here is develop example theories where these techniques are actually useful, namely finite group presentations and vector space direct sums. Both are included in the SSReflect 1.3 distribution, and the latter is described in a paper under submission. The first case study was presented at the MAP 2010 meeting; the second is covered by a submitted article. This report summarizes our findings for these two case studies, and outlines further work in this area.

The third technique we consider relies on developping algorithms that are independent from the SSReflect library and are later proved correct with respect to the library. These algorithms work on simple data-types and are fine-tuned for executability. This technique can be instantiated for many of the tasks in the project and the part devoted to this technique can also be used as a tutorial example. In particular, we illustrate how to compute determinants of medium size matrices directly in the Coq system.

2 Syntactic reflection

Beyong small-scale reflection, the easiest way of implementing reflection is to trick the user into doing all of the work by subverting the system parser. While this approach has limitations

(only user input can be reflected), it is very easy to implement in Coq by overloading operator syntax under the control of inherited attributes (called "scopes" in the Coq documentation).

We have experimented with this approach in the `presentation` library that develops the theory of generators-and-relations presentation of finite groups, i.e., descriptions such as

$$D_{2q} \simeq \mathrm{Grp}(x, y \ : \ x^q = y^2 = 1, x^y = x^{-1})$$

This state that the dihedral group D_{2q} can be characterised as the largest group with two generators x and y satisfying the above relations. This result is proved in the `extremal` library as

```
Lemma Grp_dihedral :
  'D_q.*2 \isog Grp (x : y : (x ^+ q, y ^+ 2, x ^ y = x^-1)).
```

In order to do any kind of reasoning with a presentation we must be able to ascertain that the relation part is monotonic with respect to group morphisms. It would seem the only practical way of doing this is inspecting the syntactic term to make sure it only uses the "right" operators, whence the need for reflection.

Syntactic reflection provides an easier way: describe the relation part of a presentation using the data structure described in the introduction, so that it is valid *by construction*, simply by overloading operators:

```
Infix "^+" := Exp : group_presentation.
Arguments Scope Cast [group_presentation].
Notation "H \isog 'Grp' ( x : p )" :=
 (iso H (fun x => Cast p)).
```

These declarations ensure that inside `p`, `x ^+ q` is read as `Exp x q`. We use a simple form of higher-order abstract syntax to deal with generator bindings:

```
Inductive formula := Eq2 of term & term | And of formula & formula.
Inductive type := Generator of term -> type | Formula of formula.
```

The interpretation function for `Presentation.type` saturates the abstracted formula with explicit constants before calling the `eval` function above. It regroups all generators in a single tuple, so that presentation witnesses are easy to introduce and provide.

This case study shows that such a simple-minded approach to reflection can be extremely effective: the `presentation` file is only 257 lines of Coq, yet it supports the usual mathematical notation and provides lemmas that let the client `extremal` and `extraspecial` libraries prove and exploit presentations with a single SSReflect tactic line.

3 Partial reflection

When reflection is needed for expressions that are not directly input by the user then the simple syntactic approach no longer suffices. This happens for expressions that appear in tactic-generated subgoals, or are hidden by a definition or notation. In this case it is necessary to harness either type inference or type and term unification. Coq allows both: the former

through `Coercion`, `Typeclass` and `Instance` declarations, and the latter (which is somewhat more powerful) through `Canonical Structure` declarations.

While it has been known that either of these can be used to implement general reflection, our initial experiment were somewhat disappointing: for odd technical reasons (unfortunate implementation choices in the Coq system), these implementations are not competitive with native or even tactic-based reflection for large computer algebra examples, e.g., handled by the `ring` tactic. Moreover bidirectionality and composability are irrelevant to the main purpose of the `ring` tactic, namely computer algebra-based simplification.

Although these negative results indicate that at present logic-based reflection is a poor fit for computer algebra, they do not rule out the existence of good use cases. Indeed we found one such in the process of formalizing the linear algebra and representation theory required for the proof of the Feit-Thompson Theorem; these are libraries `mxalgebra` and `mxrepresentation` of the SSReflect 1.3 distribution, respectively.

In the course of this work we needed to formalize the concept of a direct sum of subspaces. Recall that a (pointwise) sum of subspaces

$$V = \sum_i V_i$$

is *direct* when each vector $v \in V$ decomposes uniquely as a sum $V = \sum_i v_i$ of vectors $v_i \in V_i$. This strong combinatorial property turns out to be equivalent to the simple arithmetic equality

$$\dim V = \sum_i \dim V_i$$

Indeed, as the inequality $\dim V \leq \sum_i \dim V_i$ always holds, this is a limit case for $\dim V$.

This correspondance makes lets us move from poinwise combinatorics to setwise arithmetic, making direct sums a very effective proof tool. However, in practice it needs to be applied to sums that are not as regular as above, e.g.,

$$V = A + B + \sum_i (U_i + V_i)$$

To even give meaning to the statement "V is direct" it is necessary to analyze the right-hand expression to understand how it is a sum of subspaces. In Coq this would appear to require the use of general reflection, as elementary summands could be arbitrary expressions; the previous syntactic approach is unapplicable here since such sums are often large expressions that are produced by tactics, and need to be abbreviated by definitions.

However, on closer examination we may notice that we do not actually care about the syntactic description of V as a sum of subspaces; all we care about, to define and use the "V is direct" assertion, is the *integer*

$$d = \dim A + \dim B + \sum_i (\dim U_i + \dim V_i)$$

together with the knowledge that it corresponds to *some* sum decomposition. Hence we are content with a *specialized* reflection that only provides the correspondence between V and d.

This specialization solves the efficiency issues we previously had, *and* lets us use reflection bidirectionally.

For example, the following lemma in the `vector` library captures the "limit case" property of direct sums we mentioned above:

```
Lemma dimv_sumw_leqif : forall (S : addv_expr),
  \dim (unwrap S) <= unwrap (addv_dim S) ?= iff directv (unwrap S).
```

In this statement `x <= y ?= iff C` is the three-way comparison predicate defined in the `ssrnat` library that reads as "$x <= y$ and equality holds iff C", `addv_expr` is the Coq type for the reflection of a subspace sum, which both coerces to a subspace but can also be explicitly projected to a sum of dimensions with `addv_dim`. The `unwrap` projection is a technical device needed to implement general reflection with the `Canonical Structure` feature (see papers for details).

Thanks to appropriate `Canonical Structure` declarations, the three arguments to the comparison in this lemma can be unified with the dimension of a sum of spaces, a sum of dimensions, and a statement that a subspace sum is direct, respectively. *Any* of these will determine `S`, so this lemma can be composed in any direction.

In the `BGsection2` file that contains the formalization of the Bender and Glauberman revision of the Local Analysis for the Odd Order Theorem, we exploit this composability to chain such comparisons, collecting the equality conditions and then establishing them simultaneously by showing the chain is circular.

This case study shows that once specialized to a specific purpose, unification-based reflection can be a very effective formalization technique.

4 Reflection for computation

In its current implementation, the math-components library uses reflection and is fine-tuned for proof purposes and not for execution purposes. As a result, most of the algorithms contained in the library are blocked for execution. The experience described in this section shows an example where an algorithm is developed to be freely executable and the proof of correctness still relies on the rich mathematical contents of the math-components library. This experiment also describes one approach to add large-scale reflection in some proof steps. For this report, we concentrate on the computation of determinants. The development described in this section is available on the internet at the following address:

`http://wiki.portal.chalmers.se/cse/uploads/ForMath/FormathDeterminants.zip`

4.1 The data-type

When defining a function to compute determinants, we want to avoid the presentation of matrices as used in the "mathematical part" of math-components, because the functions from that part are fine tuned for proof purposes and obstacles have been added willfully to prevent computation.

It is also meaningful to use datatypes with less guarantees provided by types, so that less information needs to be verified at input time. Here, we can use sequences of sequences to represent bi-dimensional arrays, supposing that missing elements (when sequences are too short) will be replaced by zero elements or rows of zero elements. However, because of this, the same matrix can be represented by an infinity of data elements.

For instance, let us consider three matrices and their representations as sequences of sequences. The first matrix is a 3 by 3 matrix, with no zero element.

$$\begin{bmatrix} 3 & 4 & 1 \\ 7 & 1 & 8 \\ 12 & -2 & 4 \end{bmatrix}$$

This matrix is normally represented by a sequence containing three elements, where each element is a sequence containing three elements too.

```
Definition example1 :=
 [:: [:: 3; 4; 1]; [:: 7; 1; 8]; [:: 12; -2; 4]].
```

The second matrix is a 4 by 4 matrix, but one of the rows has a zero as the last element.

$$\begin{bmatrix} 3 & 4 & 1 & 2 \\ 7 & 1 & 8 & 0 \\ 12 & -2 & 4 & 5 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

This matrix can be represented by any of the following two examples, in particular, in the second example, one of the rows only has 3 elements. The last one is missing, but will be considered as zero in all of the algorithms we consider.

```
Definition example2_1 :=
[:: [:: 3; 4; 1; 2];
    [:: 7; 1; 8; 0];
    [::12;-2; 4; 5];
    [:: 3; 3; 3; 3]].

Definition example2_2 :=
[:: [:: 3; 4; 1; 2];
    [:: 7; 1; 8];
    [::12;-2; 4; 5];
    [:: 3; 3; 3; 3]].
```

The third matrix is also a 4 by 4 matrix, but the last rows contains only zeros.

$$\begin{bmatrix} 3 & 4 & 1 & 2 \\ 7 & 1 & 8 & 0 \\ 12 & -2 & 4 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

When representing this matrix as a sequence of sequences, we can simply omit the last row. All algorithms we consider will "complete" the matrix by adding a row of zeros at the end of the description.

```
Definition example3 :=
[:: [:: 3; 4; 1; 2];
    [:: 7; 1; 8];
    [::12;-2; 4; 5]].
```

4.2 The determinant algorithm

We write a first function that performs recursively the expansion along the first column of a matrix.

```
Fixpoint expand sign (mat : seq (seq R)) det acc : R :=
  if mat is a :: tl then
    sign * head 0 a * det (rev acc ++ (map behead tl)) +
    expand (- sign) tl det (behead a :: acc)
  else
    0.
```

The `mat` argument is supposed to contain the last k rows of a matrix, while the `acc` argument is supposed to contain the first k rows with the first element removed. The det argument is a function that computes determinants for sub-matrices. The algorithm progresses through `mat`, multiplying the first element of the row being observed by the determinant of the co-matrix, reconstructed using `acc` and the other rows of `mat` and the sign argument is changed at each row (in a first version of the algorithm I made a mistake and replaced the last occurrence of `acc` with `tl`, but this mistake was detected when proving the algorithm correct). In initial versions of this algorithm, there was a dimension argument, but it appeared that this argument was not used. On the other hand, this function proceeds until the end of the sequence of rows, even if this sequence has too many elements. Thus, since the dimension does not appear explicitely, we should keep in mind that the size of `mat` plus the size of `acc` should be less than the the intended matrix dimension.

The determinant computation is then described as a a simple function wich calls `expand` and passes itself as argument in the `det` position.

```
Fixpoint sdet n (mat : seq (seq R)) : R :=
  if n is p.+1 then expand 1 mat (sdet p) [::] else 1.
```

The technique of defining `expand` as a higher-order function (since it takes a function argument) and then using it in `sdet` is a general approach to handle mutually recursive functions.

4.3 The correctness statement

To express the correctness of the algorithm, we need to clarify the relation between sequences of sequences and matrices. This is done with the following function definitions.

```
Definition sseq2mat n m (mat: seq (seq R)) :=
  \matrix_(i < n, j < m) nth 0 (nth [::] mat i) j.

Definition mat2sseq n m (mat : 'M[R]_(n,m)) : seq (seq R) :=
map (fun i => map (fun j => mat i j) (enum 'I_m)) (enum 'I_n).
```

The function `sseq2mat` is the left inverse of the function `mat2sseq`, and we can easily prove it formally. For information, the proof fits in the following lines:

```
Lemma can_mat2sseq :
    forall n m mat, sseq2mat n m (mat2sseq mat) = mat.
Proof.
move => n m mat; apply/matrixP => x y.
rewrite /sseq2mat /mat2sseq mxE.
rewrite (nth_map x); last by rewrite size_enum_ord ltn_ord.
rewrite (nth_map y); last by rewrite size_enum_ord ltn_ord.
by rewrite !nth_ord_enum.
Qed.
```

On the first proof line, the tactic `apply/matrixP` is used to express that two matrices are equal when their components at every pair of coordinates are equal. On the second proof line the first two rewrite directives, `/sseq2mat` and `/mat2sseq`, express that one should expand the definitions of `sseq2mat` and `mat2sseq`. The last rewrite, `mxE`, expresses that the component at coordinates x and y of a matrix described as `\matrix_(i,j)` f `i j` is f x y.

On the third proof line, the theorem `nth_map` is used to express that the element at rank x in a sequence of the form `map` f s is f applied to the element at rank x in s. However, this is true only when the rank n is smaller than the size of the sequence. So it is necessary to show that the numbers in the enumeration of an ordinal (a type `'I_`n containing 0, 1, ... n-1) are always smaller than the size of this enumeration. The lemma `size_enum_ord` says that the size of the enumeration is exactly n, and the lemma `ltn_ord` says that all elements in `'I_`n are smaller than n. The lemma `nth_enum_ord` that the element at rank k in the enumeration is exactly k.

The correctness of the determinant computation algorithm is expressed using directly a theorem provided in the math-components libraries, which expresses that any determinant computation can be performed by expanding along a column.

```
expand_det_col
    : forall (R : comRingType) (n : nat) (A : 'M_n) (j0 : 'I_n),
      \det A = \sum_i A i j0 * cofactor A i j0
```

The `expand` function actually computes partially the sum that appears on the right-hand side of the equality in this theorem, taking the last $n - i$ terms, when i is the size of `acc`. The proof of correctness is thus to express that we are computing such a sum for a reference matrix r. This matrix can be re-constructed from `acc` and `mat` and some missing data v by saying that:

1. the first i rows of the matrix are obtained by taking the rows in `acc` in reverse order, adding the elements of v in front,

2. the last $n - i$ rows of the matrix are the rows of `mat`.

This is best understood by looking at a concrete example. Suppose we want to compute the determinant of the following 4 by 4 matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

After two recursive calls, the `expand` function will work with the argument `acc` instantiated to

$$\begin{bmatrix} 6 & 7 & 8 \\ 2 & 3 & 4 \end{bmatrix}$$

and the argument `mat` instantiated to

$$\begin{bmatrix} 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Thus, the combination of `acc` and `mat` is not enough to know what is the complete matrix whose determinant is being computed. The missing data is the column vector v:

$$v = \begin{bmatrix} 1 \\ 5 \end{bmatrix}.$$

Building matrices from smaller blocks is provided by the math-components libraries. For instance there is a `col_mx` function that builds a new matrix from two matrices with the same number of colunms. More precisely, given a matrix M_1 of size n_1 by m and a matrix M_2 of size n_2 by m, this function builds a matrix of size $n_1 + n_2$ by m whose first n_1 rows are given by M_1 and the last n_2 rows are given by M_2. There is also a similar function called `row_mx` that builds a new matrix from two input matrices with the same number of rows. Thus, the reference matrix for which the determinant is being computed can be described as:

```
col_mx
  (row_mx (sseq2mat k 1 v) (sseq2mat k m (rev acc)))
  (sseq2mat l m mat)
```

In this version, we take `v` to be a sequence, in which the first k elements will be observed.

An alternative description of the reference matrix can be obtained by using `sseq2mat` only once, using standard sequence operations to first construct a sequence of sequences that correspond to the matrix. To follow this approach we first defined an auxiliary function that manipulates two sequences s_1 and s_2 and constructs a single sequence where each element is a sequence whose first element comes from s_1 and the other elements come from s_2.

```
Definition czip (v:seq R) (l: seq (seq R)) :=
  map (fun p => let: (x,y) := p in x::y) (zip v l).
```

Thanks to this auxiliary function we can then describe the reference matrix as:

```
sseq2mat (k + l) m (rev (czip v' acc) ++ mat)
```

In this approach `v'` and `acc` need to have length `k`, and this is not enforced by the type system.

Thus, we have two different ways to express the correctness of the `expand` function, one that relies more on dependent types to ensure the consistency of data and one that relies more on plain manipulation of sequences.

```
Lemma expand_correct' :
  forall mat p (k l:nat) (f : seq (seq R) -> R)
  s acc v (h : (p.+1 = k + l)%N),
  size acc = k -> size mat <= l -> size v = k ->
  (forall mat', size mat' <= p -> f mat' = \det (sseq2mat p p mat')) ->
  let r := col_mx (row_mx (sseq2mat k 1 v) (sseq2mat k p (rev acc)))
             (sseq2mat l p.+1 mat) in
  expand s mat f acc =
  s * (-1)^+ k * \sum_(i < l) r (rshift k i) 0 *
    cofactor (castmx (refl_equal (k + l)%N, h) r)
      (rshift k i) (cast_ord h ord0).

Lemma expand_correct :
  forall mat p (k l:nat) (f : seq (seq R) -> R)
  s acc v (h :(p.+1 = k + l)%N),
  size acc = k -> size mat <= l -> size v = k ->
  (forall mat', size mat' <= p -> f mat' = \det (sseq2mat p p mat')) ->
  let r := sseq2mat (k+l) p.+1 (rev (czip v acc) ++ mat) in
  expand s mat f acc =
  s * (-1)^+ k * \sum_(i < l) r (rshift k i) 0 *
    cofactor (castmx (refl_equal (k + l)%N, h) r)
      (rshift k i) (cast_ord h ord0).
```

We see that these two statements differ only in the way the reference matrix is described, at the line where `r` is described (the line that starts with `let r := ...`). We tried to prove these two statement in the order they appear here, thus starting with the statement that was using more dependent types. However, it turned out to be too unpractical and we eventually gave up and proved only the second statement. The lesson that we learned from this experience was to avoid statements that relied too heavily on dependent types.

4.4 Helpful lemmas

When programming with dependent types, we are regularly confronted with the problem of manipulating matrices whose types are provably equal but not convertible. For instance, the proof of correctness of `expand` naturally gives rise to matrices of size `(1 + k) + l` and matrices of size `k + (1 + l)`, which appear to the proof system to live in different types but actually are equal. To cope with this issue, the math-components library provides a "cast" function, that helps changing points of view with respect to the dimension of matrices. More

precisely, given proofs $h_1 : n_1 = n_3$ and $h_2 : n_2 = n_4$, `castmx` $(h_1,\ h_2)$ transforms any matrix of size n_1 by n_2 in a matrix of size n_3 by n_4, without changing the coefficients at any location.

In fact, the computation of determinants depends only marginally on the size of matrices, so that two matrices that differ only in the way their dimension is expressed should have the same determinant. This is expressed by a lemma which we needed to prove.

```
Lemma castmx_det :
  forall n p (m : 'M[R]_n) (h : n = p), \det m = \det (castmx (h, h) m).
Proof.
 by move => n p m h; move: (h) m; rewrite h => h' m; rewrite castmx_id.
Qed.
```

In the same respect, two matrices obtained from the same sequence representation for dimensions that are equal should be equal, this can be expressed by a statement using `castmx`, too.

```
Lemma cast_sseq2mat :
  forall n1 n2 n3 n4 (q : n1 = n2) (h : n3 = n4) m,
    castmx (q, h) (sseq2mat n1 n3 m) = sseq2mat n2 n4 m.
Proof.
by move => n1 n2 n3 n4 q h; move: (q) (h); rewrite q h => q' h' m;
   rewrite castmx_id.
Qed.
```

In these two proofs, we need to rewrite with an equation while this equation is also used in the term being rewritten (as an argument to the `castmx` function). This is a tricky step because badly typed terms may appear when this is badly done. We manage to avoid problems by duplicating instances of the equation (for instance in the tactic `move: (h)m` in the proof of `catmx_det` and in the tactic `move: (q)(h)` in the proof of `cast_sseq2mat` and then rewriting with the equation. As a result, the equations used in the `castmx` function end up being equations between the same value and we can conclude with a lemma `castmx\_id`, which states that in this case, the `castmx` call can simply be removed.

Other lemmas establish correspondences between the function `col'` provided in mathcomponents to construct a new matrix by removing a column from a given input matrix and the operation of removing the first element from a sequence. A similar lemma is provided for the function `row'`.

4.5 Structure of the central proof

The proof of the lemma `expand_correct` is 49 lines long, so it is the lemma where most difficulties concentrate. Let us look again at the proved statement.

```
Lemma expand_correct :
  forall mat p (k l:nat) (f : seq (seq R) -> R)
  s acc v (h :(p.+1 = k + l)%N),
  size acc = k -> size mat <= l -> size v = k ->
```

```
(forall mat', size mat' <= p -> f mat' = \det (sseq2mat p p mat')) ->
let r := sseq2mat (k+l) p.+1 (rev (czip v acc) ++ mat) in
expand s mat f acc =
s * (-1)^+ k * \sum_(i < l) r (rshift k i) 0 *
  cofactor (castmx (refl_equal (k + l)%N, h) r)
    (rshift k i) (cast_ord h ord0).
```

The objective is to show that the computation performed in the function `expand` is the same as the sum performed when expanding along the first column (this column being indexed by 0). We have to be careful, because the conditions on data only ensure that the number of rows being considered is lower than the number of remaining rows in the matrix.

We perform the proof by induction on the structure of the argument `mat`, which is a sequence.

In the base case, the sequence of rows represented by `mat` is empty, so that the matrix of size `l` represented by `mat` is a zero matrix, and the big sum is a sum of zero elements, because one the factors in each of summands is 0. To use the fact that all summands are zero, we rely on the lemma `big1` from the math-component libraries. To show that all summands are zero, we first name the reference matrix (we use the name `ref` in the formal proof) and reason on the fact that the matrix is being read at a row indexed by `k + i`, where `k` is the size of `acc` and `i` is a number smaller than `l`. However, the first `k` rows of the matrix are described by `rev (czip v acc)` and the next rows are described by `[::]`. We show that computing the element of the sum at index `i` implies computing the coefficient of the matrix at position (`rshift k i`, 0). However in plain mathematical notation, `(rshift k i)` means `k + i`, viewed as a number in `'I_k+l`. By using the lemma `sseq2mat_cat`, we establish that the reference matrix is actually of the form `col_mx M (sseq2mat l p.+1 [::])`, where `M` is a matrix with `k` rows, thus the coefficient at location `(k+i, 0)` in `ref` is actually the coefficient at location `(i, 0)` in the matrix `sseq2mat l p.+1 [::]`. This element is easily shown to be zero, using the following tactic.

```
by rewrite col_mxEd mxE !nth_default.
```

The lemmas `sseq2mat_cat` requires that we verify the sequence of rows used to describe the matrix `M` mentioned in the above paragraph has size `k`, this is done by observing that `rev` and `map` preserve sizes, and that `zip` yields a result whose size is the minimum of the inputs' size. In all, the base case is verified using 4 lines of tactics.

In the step case, we name `r` the first row of `mat` and `mat'` the rest and we have an induction hypothesis `Ih` for `mat'`. Because of the size constraint, we establish that `l` is necessarily of the form `l'.+1`, and we again name `ref` the reference matrix. We then establish a few equalities between various points of view about the size of the reference matrix and its sub components and we rewrite with the induction hypothesis instantiated on `p l'`, the sign `(-s)`, and the recursive call arguments `mat'` and `behead r::acc`.

The next step is to isolate the first summand in the left hand side of the formula; we need to show that the element of the first column at position `(k, 0)` in the reference matrix is the head of the row `r`. Then a long detailed sub-proof is required to show that the cofactor for position `(k, 0)` is also correctly computed by the recursive call.

The last part of the proof consists in showing that the sums on both sides of the equality, after removing the first element, are the same. The only problem is to show that index are shifted in the same manner on both sides and that we have two ways to look at the same reference matrix. Unfortunately, this part is quite long-winded (it takes 15 lines).

4.6 The main correctness lemma

The correctness statement for the `sdet` function is proved much more easily. It relies on a proof by induction on the size of the square matrix being considered. So, in the step case, we can already assume that computations of determinants of size n are performed correctly when attempting to prove that determinants of size $n + 1$ are correct.

As we already said, we rely on a mathematical lemma from the math-components libraries, `expand_det_col` which expresses that the computation of a determinant can be performed by expanding over any column of the matrix. It is then necessary to recognize that lemma `expand_correct` does produce a statement that is similar to the statement of `expand_det_col`, this mostly means getting rid of instances of type-cast functions, like `castmx` or similar functions for other type families.

```
Lemma sdet_correct :
  forall n mat, size mat <= n -> sdet n mat = \det (sseq2mat n n mat).
Proof.
elim => [ | n Ih] mat sm /=.
 by rewrite /sdet /sseq2mat (flatmx0 (\matrix_(i,j) _))
      -(flatmx0 1%:M) det1.
rewrite (expand_correct _ (erefl _ : n.+1 = (0 + n.+1)%N)
          (erefl (size nil)) sm (erefl (size nil)) Ih).
rewrite !castmx_id (expand_det_col _ 0) expr0 !mul1r; apply: eq_bigr => i _.
by rewrite cast_ord_id (_ : rshift 0 i = i) //; apply: val_inj.
Qed.
```

4.7 Executing the code

The algorithm explicitely relies on the operations of a commutative ring structure. If we want to compute with an arbitrary type of numbers, we need to show that this type is endowed with such a ring structure. We did this work for the standard integers from the Coq library `ZArith`. In particular we used the existing functions to show that there exists a boolean equality test and we showed that the integers could be enumerated. Last we exhibited all the theorems that express the associativity, commutativity, and distributivity properties of addition and multiplication. We were eventually able to compute the determinant of an example matrix giving rise to a dialog of the following form:

```
Eval vm_compute in sdet 4
 ([::[:: 1; 2; 3; 4];
     [:: 5; 6; 7; 8];
     [:: 9; 10; 28; 0];
```

```
  [:: 3; 2; 2; -1]])%Z.
= 20%Z
: SZ_comRing
```

To conclude this section, this implementation of determinant computation can be used to perform large-scale computations to show that some concrete matrix is invertible, for instance. However, connecting this computation with the given matrix still requires some code.

5 Full reflection and future work

In a new collaboration we have recently discovered new more efficient implementation strategies for general reflection using canonical structures, and new ways to specialize such implementations to specific purposes, in the context of generalized Hoare type theory. An article describing this work is in preparation.

The topic of general reflection for computer algebra is presently actively investigated by several members of the Formath project (at Inria and Nijmegen sites) but also by other members of the Coq community of users and developers. For instance Loïc Pottier (Inria) has developed a type classes based ring tactic, which has been integrated in March 2011 to the development branch of the Coq system. A meeting on this topic will take place on March 31st to synchronize the efforts made by the different contributors. Several members of the Formath project (from Inria and Nijemegen sites) will present and discuss their experiments on the design and performance of these techniques.