Report on the implementation of a revised version of Hilbert function related algorithms

Carlo Traverso

Deliverable 3.3.4.1.1

Contents

1 Abstract

The Hilbert function generalizations have been provided in C++.

The delivered implementation contains:

1. Hilbert function for modules;
2. weighted Hilbert function;
3. weighted Hilbert function for modules;
4. applications to the fast computation of intersection and transporter;
5. server interface.

The multi-weighted Hilbert function for modules has not been implemented, since a preliminary implementation has shown that the polynomial arithmetic involved is too heavy, and in the use for multi-Hilbert-driven Buchberger algorithm, the weighted Hilbert with varying weights may be better used instead.

The files `lib/Groebner/hilbert.H` and `lib/Groebner/hilbert.C` have been substantially rewritten; moreover small changes have been made to other files of the library. The PoSSo Library 2.1.1 contains these files.

The algorithms fo the fast computation of intersection and transporterr are described in [3].

2 Introduction

The Hilbert function generalizations have been provided in C++.

The delivered implementation contains:

1. Hilbert function for modules;
2. weighted Hilbert function;
3. weighted Hilbert function for modules;
4. applications to the fast computation of intersection and transporter;
5. server interface.

The multi-weighted Hilbert function for modules has not been implemented, since a preliminary implementation has shown that the polynomial arithmetic involved is too heavy, and in the use for multi-Hilbert-driven Buchberger algorithm, the weighted Hilbert with varying weights may be better used instead, with an algorithm outlined in section 3 of this report.

The first 4 items are contained in the update of the library files lib/Groebner/hilbert.*, while the fifth item consists in an addition of the functions O_FastTransporter, O_TransporterSum, O_FastIntersect, O_FastMultipleIntersect, PL_ModuleHilbertFunction, PL_WeightedHilbertFunction, PL_WeightedModuleHilbertFunction and of the test files fastTransporter.xcl, FMIntersect_1.xcl, WeightedHilbertFunction.xcl.

The algorithms fo the fast computation of intersection and transporter are described [3].

3 The user interface

The files `lib/Groebner/hilbert.H` and `lib/Groebner/hilbert.C` have been substantially rewritten; moreover small changes have been made to other files of the library. Here we comment the new `hilbert.H` file, and the additions to the server. The other changes are marginal and strictly necessary for the implementation (and are already integrated in the version 2.1 of the library).

3.1 HilbertFunctionNumerator functions

The function

```
PL_List<int> & PolyListHilbertSeriesNumerator( PL_PolyList &IN,
                                                const int interreduced = 0);
```

has been replaced by the following five:

```
void
IntListShiftedSum (PL_List<int> & l1, PL_List<int> & l2, int shift);

void
PolyListGetHead (PL_List<PL_PP> & ppl, PL_PolyList &IN);

PL_List<int> &
PPListHilbertSeriesNumerator(PL_List<PL_PP> &IN, PL_PPMonoid & PPM,
                             const int interreduced = 0,
                             Directives weight = NULL);

PL_List<int> &
PolyListHilbertSeriesNumerator(PL_PolyList &IN, const int interreduced = 0,
                               Directives weight = NULL);


PL_List<int> &
PolyListHilbertSeriesNumerator( PL_PolyList &IN , const int interreduced,
                                PL_List< PL_PP >  &LQ ,PL_List< int >  &LM,
                                Directives weight = NULL);
```

with the following semantics:

IntListShiftedSum takes two integer lists, and sums destructively to the first one the second one, inserting `shift` zeros in the second one after the first element. If the first list is the Hilbert series numerator of a graded module M and the second is the Hilbert series numerator of a graded module N, it modifies the first one to contain the Hilbert series numerator of $M \oplus N(shift)$, where $N(s)$ is the same of N with the graduation index shifted by s.

PolyListGetHead extracts the leading PP from a list of polynomials. This utility function should be inserted in another place in the library (together with other functions of lib/Groebner/algebra.*).

PPListHilbertSeriesNumerator is the function that realizes the interface with the core Hilbert library (that has not been changed).

PPListHilbertSeriesNumerator in two variants (the first one is for ideals, the second for modules) has a new optional field weight, that if present calls the weighted Hilbert function. The two additional parameters for modules are a list of PP, LQ (that must be the in the module variables, and are the module generators) and a list of integers LM (that are the shifts to be given to these generators); the result is the Hilbert series numerator of $\bigoplus_i M_i(s_i)$ where M_i is generated by the i-th element of LQ and s_i is the i-th element of LM.

3.2 the class HilbertReductor

The class HilberReductor has been modified allowing two constructors, one for the ideals and the other for modules; the second has two additional arguments:

```
    PL_List<PL_PP> & InModuleGenerators,
    PL_List<int> & InModuleShift,
```

The class moreover has new methods

```
static PL_PolyList & fastIntersection(PL_PolyList &pl1, PL_PolyList &pl2,
                                      PL_PolyRing &pr1, PL_PolyRing &pr2);

static PL_PolyList & fastMultipleIntersection(PL_List<PL_PolyList> &InList,
                                              PL_PolyRing &pr1,
                                              PL_PolyRing &pr2);


static PL_PolyList &
fastModuleIntersection(PL_PolyList &pl1, PL_PolyList &pl2,
                       PL_PolyRing &pr1, PL_PolyRing &pr2,
                       PL_List< PL_PP >  &LQ = NULL,
                       PL_List< int >  &LM = NULL);

static PL_PolyList & fastTransporter(PL_PolyList &InList, PL_Poly &InPoly,
                                     PL_PolyRing &pr1, PL_PolyRing &pr2,
                                     PL_PolyList * sum = NULL);

static PL_PolyList & fastMultipleTransporter(PL_PolyList &InList,
                                             PL_PolyList &InList2,
                                             PL_PolyRing &pr1,
                                             PL_PolyRing &pr2);
```

that implement fast versions (using the module Hilbert function) of intersection and transporter. These can be used in the computation of the radical, and provide sensible speed-up.

The implementation of these methods was not anticipated, but the discovery of the new fast algorithms (see [2] and [3]) that use the module Hilbert function has suggested the inclusion in this delivery.

The interface is similar to the interface of the methods Intersection and Transporter of the GroebnerReductor class, with the following differences:

- the ring pr2, that is an auxiliary intermediate ring, is different: in the fast version it must be pr1 (the ring of the PolyList) to which a "large" module variable of weight 0 and a "small" homohenization variable of weight 1 are added (in the GroebnerReductor functions pr2 must be obtained a large and a small module variable, both of weight 0).
- fastIntersection and fastModuleIntersection have two PolyList arguments (and implement the intersection of two ideals or modules), while Intersection and fastMultipleIntersection have a PL_List<PL_PolyList> argument, and implement the intersection of an arbitrary set of ideals.
- fastTransporter, compared to transporter, has the second argument that is a PL_Poly instead of PL_PolyList, and implements the transporter $I : (f)$ of an ideal by a principal ideal. The generic transporter is obtained by intersection. Moreover an optional parameter sum allows to retrieve the sum $I + (f)$ that is computed as a side-effect with marginal additional cost.

Although to use the Hilbert-driven algorithm homogeneous ideals are needed, the arguments here may be non-homogeneous, but (as for the GroebnerReductor methods) the result might be incorrect if the input is not a Groebner basis; since checking it may be costly, it is the respoinsability of the user to provide adequate data.

The following further modifications or additions are considered necessary, and will be added in a future version; they will be included in the implementation of radical and decomposition computations that will use them.

1. allow defaults for pr1 and pr2;
2. implement an optimized version of fastMultipleTransporter
3. add the functions fastModuleIntersection and fastMultipleModuleIntersection;
4. add the function moduleTransporter and maybe multipleModuleTransporter.

The first requires the introduction of adequate ring constructions (for the default parameter pr2: pr1 can be obtained from the other parameters, and will remain only for backward compatibility). This will be done in short time: a prototype is already present in the implementation of the class NFPoly.

The second point requires a tuning of the algorithm: the present implementation just combines fastMultipleIntersection and fastTransporter and gives a non-optimal algorithm (in generic position all the intersections will be equal, and the algorithm is just a repetion of the same step many times; adding a simple check avoids this repetition).

The third and fourth require the design of an adequate interface, and will be implemented after the revision of the present interface with the introduction of defaults for the rings.

3.3 The server additions

The following functions have been added to the server; we use the notation of the server documentation.

```
PL_ModuleHilbertFunction [id(PL)][id(PL)][IL]             -> [IL]
PL_WeightedHilbertFunction [id(PL)][IL]                   -> [IL]
PL_WeightedModuleHilbertFunction [id(PL)][id(PL)][IL][IL] -> [IL]
O_FastTransporter [id(PL)][id(P)][id(PR)][id(PR)][id(PL)]
O_FastMultipleTransporter [id(PL)][id(P)][id(PR)][id(PR)][id(PL)]
O_TransporterSum [id(PL)][id(P)][id(PR)][id(PR)][id(PL)][id(PL)]
O_FastIntersect [id(PL)][id(PL)][id(PR)][id(PR)]
O_FastMultipleIntersect {[id(PL)],...,[id(PL)]}[id(PR)][id(PR)][id(PL)]
```

the arguments being:

PL_ModuleHilbertFunction: the PolyList of which we compute the Hilbert function, a PolyList whose heads are the module generators, the module shifts.

PL_WeightedHilbertFunction: the PolyList of which we compute the Hilbert function, a list of weights (missing items default to the UserWeight of the PolyRing).

PL_WeightedModuleHilbertFunction: the same as PL_ModuleHilbertFunction, with a last argument for the weight.

O_FastTransporter: the ideal I, the polynomial f, the ring of I, the working ring, the result $I:(f)$.

O_TransporterSum: the same as above, adding an argument for the sum.

O_FastIntersect: the ideal I, the ideal J, the ring of I, the working ring, the result $I \cap J$

O_FastMultipleIntersect: a list of ideals, their ring, a working ring, the result.

4 Multigraded Hilbert function and Kronecker trick

We recall shortly how the Hilbert-driven and the multi-Hilbert-driven Buchberger algorithms work.

Assume that we have an ideal that is generated by homogeneous (resp. multi-homogeneous) polynomials, and we know a Groebner basis w.r.t an ordering; we want to compute the Groebner basis w.r.t. another ordering, and the Hilbert functions allows to know the number of expected pairs in the lowest degree in which one is expected: the number and lowest degree of expected pairs is equal to the difference and degree of the coefficients of the smallest non-matching monomial of the Hilbert series numerator between the known Groebner basis and the partially computed basis.

In the multi-homogeneous case, when we have a main weight (corresponding to a main variable of the multivariate Hilbert series numerator) and one or more partial weights (in which the main weight is always larger than the partial weights, and that correspond to secondary variables) the multivariate Hilbert series numerator is a multivariate polynomial, and as above the difference of coefficients of a same monomial corresponds to the expected pairs with given multidegrees. A single computation allows to compute the number of missing pairs of all the weights corresponding to the same main weight.

The multi-Hilbert-driven Buchberger algorithm has been implemented, see [1], but it is often only partially efficient since the bottleneck is now the computation of the multivariate Hilbert function: the savings in pairs are substantial, but the multivariate polynomial is huge, and its computation takes a substantial time.

Specializations of the weights correspond to specializations of the variables. Hence for example the (univariate) Hilbert function in the main weight corresponds to specializing all the secondary variables to 1, etc.; hence for example Kronecker trick allows to compute all the monomials up to some degree specializing the variables to suitably chosen powers of an auxiliary variable (i.e. considering a new weight being a suitably chosen integer linear combination of the original weights).

A complete implementation of the multi-graded Hilbert function with Kronecker trick is possible, but not convenient because of the high degree of the resulting polynomial (that might result in a kind of "dense multivariate polynomial arithmetic"). However we are not really interested to have the multivariate polynomial, but only to identify the smallest non-matching monomial and find the difference of the coefficients, it is sufficient to compute a small number of specializations with small linear combinations to find it. This might be a key improvement of the multi-Hilbert-driven Buchberger algorithm, and its implementation only requires weighted (univariate) Hilbert functions and elementary integer arithmetic operations on the results. The algorithm can be implemented in the present frame of the HilbertReductor class, with a marginal modification to allow different implementations of the `ComputeExpectedPairs()` method.

The possibility of computing a truncated Hilbert function —the higher degree terms are discarded anyway— might be useful, but experiments are needed to decide if this is worth the effort, since it has to be implemented at very low level: it will be done if the computation of the Hilbert function is still a bottleneck.

The details will be given in a fortcoming technical paper, and an implementation of the resulting multi-Hilbert-driven Buchberger algorithm is planned.

References

[1] M. Caboara, G. De Dominicis, L. Robbiano, *Multigraded Hilbert function and Buchberger algorithm*,ISSAC 1996.

[2] M. Caboara, P. Conti, C. Traverso, *Yet Another Ideal Decomposition Theorem*, AAECC-12, Springer LNCS (1997).

[3] M. Caboara, C. Traverso, *How to compute an intersection*, submitted to ISSAC 1998.