Computing With Polynomials: A Personal Odyssey

Ruchira S. Datta

May 25, 2002

This paper chronicles my recent work in polynomial computation. Rather than focusing on a single coherent theme, my narrative includes the diverse nitty-gritty details I had to take care of and the choices that I made—and in some cases, rethought multiple times—while pursuing my dual goals of solving particular problems and finding the best framework for solving similar problems in the future. My own future efforts, at least, will benefit from this consolidation of my experience; I certainly hope that others may find some interesting nuggets among my various adventures as well. Here I will focus more on the computational aspects than on the mathematical aspects.

1 Mathematical Background

First I describe the problem on which I have spent most of my recent efforts. Consider polynomials in n variables, with integer, rational, or real coefficients, as functions $\mathbb{R}^n \to \mathbb{R}$. The problem is to find the global minimum of a polynomial, either over all of $\mathbb{R}^n$ or over a compact semialgebraic set (that is, a bounded set cut out by other polynomials). Minimization problems arise in many kinds of applications in science and engineering, and polynomials are often used because of their nice properties.

It might be thought that this problem is simple: just differentiate the polynomial, find the zeroes of the resulting (Jacobian) system, and check which is minimum. However, finding all the zeroes of a multivariate polynomial system is itself a nontrivial task. Thus other techniques which can compute the minimum, or an approximation to it, effectively, are still of interest.

Suppose that, given any function $f : \mathbb{R}^n \to \mathbb{R}$, we had a method for certifying that $f(x_1, \ldots, x_n) \geq 0$ for all $(x_1, \ldots, x_n)$ in the region of interest. Then we would have a way to minimize f over that region. We would simply apply our method to the function $f - \lambda$ for various values of λ, and it would tell us whether $f - \lambda \geq 0$, that is, whether $f \geq \lambda$, at all points of interest. Thus we would have a decision procedure for whether λ is a lower bound. Then we could find the greatest lower bound by bisection, if nothing else. (It turns out that the methods in question can find λ directly.)

One way of certifying that $f \geq 0$ is to find a *sum of squares*, or *sos*, representation of f, that is, find polynomials g_i such that $f = \sum_i g_i^2$. This will ensure that f is nonnegative over all of $\mathbb{R}^n$. On the other hand, if we are only interested in a bounded region given by polynomial constraints, that is, a set S of the form

$$S = \{(x_1, \ldots, x_n) \mid p_1(x_1, \ldots, x_n) \geq 0, \ldots, p_m(x_1, \ldots, x_n) \geq 0\},$$

then we could certify that $f \geq 0$ on that region by showing that

$$f = \sum_i c_i \prod_{j \in \mathcal{A}_i} p_j$$

where the c_i's are some coefficients which are also known to be nonnegative. In fact these methods can be combined, so we may try to find: an LP-duality representation

$$f = \sum_i c_i p_i, \qquad 0 \leq c_i \in \mathbb{R};$$

a Handelman representation

$$f = \sum_{\varepsilon \in \mathbb{N}^m} c_\varepsilon p_1^{\varepsilon_1} \cdots p_m^{\varepsilon_m} \qquad 0 \leq c_\varepsilon \in \mathbb{R};$$

a Putinar representation

$$f = s_0 + \sum_i s_i p_i, \qquad s_i = \sum_j g_{ij}^2 \text{ is sos};$$

or a Schmüdgen representation

$$f = \sum_{\varepsilon \in \{0,1\}^m} s_\varepsilon p_1^{\varepsilon_1} \cdots p_m^{\varepsilon_m}, \qquad s_\varepsilon = \sum_j g_{\varepsilon j}^2 \text{ is sos.}$$

Such a representation is a *certificate* that f is nonnegative, that is, it can be checked independently of the method used to compute it.

2 The Gram Matrix Method

So the first task is to decide whether a polynomial is sos, that is, given f, to find g_i's such that $f = \sum_i g_i^2$. As explained by N. V. Shor, this can be done using the Gram Matrix method. Write 1 and all the monomials that may occur in the g_i's in a vector, such as $(1, x, y, xy)$. (Only monomials corresponding to lattice points within half the Newton polytope of f may occur [Res78].) Then a sum of squares using these monomials can be written as a quadratic form in the monomials:

$$\begin{pmatrix} 1 & x & y & xy \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ xy \end{pmatrix}$$

where $c_{ij} = c_{ji}$. The condition that this sum of squares be equal to f is equivalent to a set of affine equations in the c_{ij}'s. For example, if the coefficient of xy in f is 3, then $2c_{14} + 2c_{23} = 3$. If the matrix $Q = (c_{ij})$ is positive semidefinite, there exists a singular value decomposition $Q = V^T \Lambda V$, where V is not necessarily square. Absorbing the square roots of the entries of Λ into V gives an expression $Q = S^T S$. Then the entries of $S(1, x, y, xy)^T$ are the g_i's. Such a matrix Q can be found by semidefinite programming, as explained in [Par01]; this is an SDP feasibility problem. If c_0 is the constant coefficient of f, the corresponding constraint is $c_{11} = c_0$. If instead we let c_{11} vary freely we can assume $c_{11} = c_0 + \lambda$, that is, we set $\lambda = c_{11} - c_0$. This is the situation described above, where we certify $f + \lambda \geq 0$; we want to minimize λ. This can also be done by semidefinite programming, as explained in [PS01]; this is an SDP optimization problem.

3 Setting Up the Semidefinite Programming Problem

I wrote Ocaml code to help set up the above semidefinite programming problem. The end result of this code was a file in SDPA format [SDPA], a particular input format used by several SDP solvers. I would then submit this file to one of the SDP solvers at NEOS [NEOS]. This would eventually return a positive semidefinite matrix. It would also return the minimal value of λ. I would enter the positive semidefinite matrix by hand into Mathematica and

get the singular value decomposition, and then $S^T S$. I would then read off the g_i's by inspection.

It would clearly be desirable to automate the above process entirely. After I wrote the above program Pablo Parrilo released a package called `SOStools` [SOS], which automates the submission of the problem to the solver, and returns the value of λ. It still does not automatically return the g_i's. Next I describe in more detail the Ocaml code I wrote. Much of this was "quick and dirty" code, to which many obvious improvements could be made.

I represented polynomials as lists of pairs of coefficients and monomials; I represented monomials as arrays of integers. I wrote a function to take an array of variable names and a character stream and parse it into a polynomial. This function used the parsing facilities provided by Camlp4, which facilitates writing recursive-descent parsers for LL(1) languages. The parser would represent the monomials as lists of pairs of variable names and integers (their powers); another subroutine would convert these into an array of integers, using the given array of variable names. To print a polynomial, the array of variable names would again have to be passed as an argument.

I wrote a function `poly_prods` to take the number of variables, the polynomial, and a list of monomials which could appear in the sum of squares ("divisors"). At the time I was using this code I was computing the list of monomials (the lattice points lying inside the Newton polytope) by hand, although I subsequently was able to do so using Fukuda's `cdd+`[cdd] to compute the Newton polytope, that is, the convex hull of the lattice points corresponding to the monomials, and Pottier's `bastat` [bastat] to compute the interior lattice points. The function I wrote would return as output a pair consisting of the list of divisor entries and a list of product entries. Each divisor entry would be a pair consisting of an integer index and a divisor (recall that the divisors were passed in as an argument). Each product entry would be a triple consisting of an integer index, a monomial, and a pair consisting of the coefficient of that monomial in the polynomial, and a list of pairs of index-divisor pairs whose product is that monomial. In the above example, the product entry for xy might be

```
(4, [|1;1|],
  (3.0,
    [ ( (1,[|0;0|]), (4,[|1;1|]) );
      ( (2,[|1;0|]), (3,[|0;1|]) ) ] ) );
```

I wrote functions which would take the list of product entries output by

`poly_prods` and return, respectively, a file in SDPA format encoding the SDP feasibility problem mentioned above (to certify nonnegativity); a file in SDPA format encoding the SDP optimization problem mentioned above (to find a lower bound for the minimum); and a file with a human readable interpretation of the product relations embodied by this list. This last file consisted of two parts, one beginning, for example:

```
There are 4 possible monomial divisors:
(1) 1
(2) x
...
```

and another beginning

```
There are 9 possible monomial products:
...
3 of (4) x*y= (3) y * (2) x= (4) x*y * (1) 1
...
```

This file was printed as an aid in interpreting the SDPA files and also the output of the SDP solver. In this last line, 3 is the coefficient of xy in the polynomial, and the index 4 of xy corresponds to the 4th constraint in the SDPA file, which will encode that $2c_{23} + 2c_{14} = 3$. (The constraint is in implicit form, as given here, for the feasibility problem; it is converted to parametric form for the optimization problem.)

The function `poly_prods` creates a hashtable, which will contain the product entries (except for the integer index). It loops through the divisors and creates a list of divisors decorated with an integer index (such as `(2,x)` in the above example). It iterates over the polynomial, adding an entry for each term with the monomial as key and a pair of the coefficient and an empty list as value. Then it iterates over the cartesian product of the divisors, adding the product of those monomials as a new key in the hashtable as necessary, and adding that pair of divisors to the list of divisor-pairs for that product. (Both iterations are actually implemented as tail-recursive functions.) : The entries in the hashtable are not in any particular order. It is desirable that they be output in a given order. Specifically, the monomials of a given degree d should be grouped together. Now consider the monomials x^2y, y^2z, and xz^2. These monomials are mapped into each other by permuting the variables. Their multi-indices `[|2;1;0|]`, `[|0;2;1|]`, and `[|1;0;2|]` all

correspond to the same partition $0 + 1 + 2 = 3$ of the total degree 3. Such monomials should be grouped together. This facilitates exploitation of any symmetry present in the original polynomial to reduce the size of the SDP computation that must be done, as explained in [GP01].

To output the product entries in this order requires enumerating the monomials in each degree in that order, seeking them in the hash table, and adding them to the output list. To enumerate monomials in this order, we first must enumerate the partitions of the the given degree with at most a certain number of parts (the number of variables). For this I use Hindenburg's algorithm [And76] for each possible number of parts. Now suppose there are 5 variables a, b, c, d, and e. A fixed partition such as $0 + 1 + 1 + 2 + 2 = 6$ can be encoded as $(0^1, 1^2, 2^2)$. Then each partition of the set of variables into subsets C, L, and Q where $|C| = 1$, $|L| = 2$, and $|Q| = 2$ corresponds to a different monomial; for example, $\{a\}, \{b, c\}, \{d, e\}$ corresponds to bcd^2e^2. So the partitions of the set of variables into subsets of the prescribed sizes must be enumerated. All enumerations are done with tail-recursive functions.

4 Schweighofer's Algorithm

The above method is used for global (unconstrained) optimization. It can be extended to the constrained case, for example to calculate a Putinar (or Schmüdgen) representation instead of an sos representation. But instead I next considered Schweighofer's algorithm, which calculates a Schmüdgen representation using linear programming rather than semidefinite programming. This algorithm actually augments the set of constraining polynomials with additional polynomials, using the Schmüdgen representation of a large constant, and then calculates a Handelman representation using all these as the constraining polynomials. My implementations only treat the case where the constraining set is a box, $l_i \leq x_i \leq u_i$ for each variable x_i, and I will only discuss this case, which is simpler.

If n is the number of variables, the $2n$ constraint polynomials are $x_1 - l_1, \ldots, x_n - l_n, u_1 - x_1, \ldots, u_n - x_n$. Let $b = \sum_{i=1}^{n}(u_i - l_i)$. We take $2n$ indeterminates $Y_1, \ldots, Y_{2n}$ and define a homomorphism $\mathbb{R}[Y_1, \ldots, Y_{2n}] \to \mathbb{R}[x_1, \ldots, x_n]$ by $Y_{2i-1} \mapsto (x_i - l_i)/b$ and $Y_{2i} \mapsto (u_i - x_i)/b$, for $i = 1, \ldots, n$. The scaling has been chosen so that $Y = \sum_{i=1}^{2n} Y_i \mapsto 1$. We can easily represent f, the polynomial we are minimizing, as a sum of coefficients times squarefree products of the Y_i's. For example, we could substitute

$x_i = bY_{2i-1} + l_i$ into our expression for f. Let g be such a representation. For a Handelman representation, we must ensure that all the coefficients are nonnegative.

Let $(Y_1 + \cdots + Y_{2n} - 1, r_1, \ldots, r_m)$ be a Gröbner basis for the kernel of our homomorphism. We can add any element of the kernel to g and still have a representation of f. So we let $g' = g + s(r_1^2 + \cdots + r_m^2)$, where s is a nonnegative parameter. As before we need to let the constant term of f vary freely. So we let $g'_0 = g + s(r_1^2 + \cdots + r_m^2) + t$. Finally, we homogenize the whole polynomial g'_0 by multiplying each term by Y (which maps to 1). In particular, the constant term t will have to be multiplied by Y^d where d is the total degree, so the resulting polynomial g_0 will be dense.

Now g_0 maps to $f + t$, so g_0 encodes a representation for $f + t$ in terms of the constraining polynomials. The coefficients are now affine forms in the parameters s and t. We want to minimize t such that all these coefficients are nonnegative. This is a linear program in the two variables s and t, which is always feasible. Solving it gives us a lower bound $-t$ on f. We can always multiply g_0 by Y and still represent $f + t$. Doing so successively gives polynomials $g_0, g_1, g_2, \ldots$. Each time the lower bound from the corresponding linear program is improved.

5 First Runs of Schweighofer's Algorithm

I carried out Schweighofer's algorithm as follows. In Singular, I used a lexicographic term order in the Y_i's and s and t, where s and t came last. I set defined polynomials $x_i = bY_{2i-1} + l_i$ and then defined the polynomial f in terms of the x_i's. I carried out the rest of the steps described above to obtain the g_i's.

Let $u = \prod_\varepsilon Y_i^{\varepsilon_i}$ be a monomial in the Y_i's. Due to the choice of term order, when Singular displays the g_i's the terms in u, su, and tu appear next to each other. I wrote a `flex` program to read off the affine constraint for each such u and write a linear programming formulation in AMPL format. I submitted the resulting file to an LP solver at NEOS.

Homogenizing the polynomial requires multiplying the terms by various powers of Y, and getting successive g_i's again requires multiplying by Y_i. The dense polynomial quickly grows to have hundreds of thousands of terms. Thus although the LP solver was returning after delays of a few seconds, Singular was taking longer and longer to multiply out the g_i's, and eventu-

ally would refuse due to running out of memory. (Mathematica was tried before Singular, and would cave in several steps sooner.) This problem led me to consider alternative ways to carry out the polynomial arithmetic. I also wished to try an exact LP solver rather than a numerical one. It seemed to me that in situations where approximate answers are acceptable, a pure numerical approach would usually be preferable to this sort of combined symbolic-numerical approach, since the symbolic-numerical approach would be slower and would give an approximate answer anyway. I considered that symbolic methods would be useful when a lower bound needs to be known exactly. (Since then I have been told that pure numerical methods for constrained optimization are not necessarily as satisfactory as I thought them to be.)

6 Modules and Functors

Much has been written about object-oriented programming. Some of its main advantages are encapsulation, which enforces modularity, and polymorphism. However, these features can be implemented without object-orientation. Modular programs can be written in Ocaml using either objects or another construction, modules, or a mixture of both. Ocaml has a strong type system with type inference. Its module system, which is effectively another language at a higher level, has an analogous type system.

Ocaml programs are usually a sequence of declarations, of types and values. (Since Ocaml is a functional programming language, functions are also values.) A module is a group of such declarations. The specification of a module, analogous to the type of an expression, is called a signature. It specifies that certain types, and certain values with given types, must be declared in the module. It is somewhat like a C++ header file. The implementation of a module, analogous to the value of an expression, is called a structure. It declares certain types and values. Just as Ocaml infers the most general possible (polymorphic) type of an expression, but a less general type can be declared with a type constraint, so Ocaml infers the most general possible signature of a module, but a less general signature can be declared with a signature constraint. For example, if some of the values declared in the module do not appear in the signature, they will be hidden from outside the module: they will be encapsulated.

Ocaml implements parametric polymorphism at the level of expressions.

When it sees a function declaration, it looks at the body and determines what the types of the arguments must be to sustain the operations in the body. Some aspects of the type of the argument may not be constrained by the operations in the body. For example, the function `hd` which takes the head of the list doesn't care what type of elements are in the list. Those aspects are free to vary, so they are left as parameters, called type variables, in the inferred type of the function. For example, the type of `hd` is `'a list`, where `'a` is a type variable.

A functor is the analogue of a function in the module language. It takes a module with a specified signature as argument and returns a module with another signature as result.

7 Algebraic Type Systems for Polynomial Arithmetic

When I began to implement my own polynomial arithmetic in order to gain more control over how the computations were arranged, I wanted to use an algebraic type system. I had encountered bugs in some frequently used computer algebra programs. I found them quite disturbing, since in these cases I could see the bug, but I would never know if the same bug had occurred in another case where the output was too big to comprehend. Since symbolic software is often used when an exact answer or an answer that can be applied to many problems is needed, and since a symbolic answer may be harder to check against physical quantities, it seemed to me that bugs in symbolic software are even less desirable than in other software. Therefore I felt that ultimately computer algebra software should be formally verified, and while I did not actually take any steps toward formal verification, I did want to structure my software to help and not hinder such a verification in the future. The use of ML was one step in this direction, and I felt that algebraic type systems were another. The program would eventually have to be compared to formalized mathematics, and so the clearer the relationship the better.

I consider only multivariate polynomials, as collections of monomials with coefficients. Here are some of the definitions from the network of signatures I first adopted for my system, which I called "Dedicas" (in the hope that this CAS would someday be *deduced*, i.e., proved, to be correct). As is often

the case, finding the right way to slice up the module system is difficult, and several choices may need to be reworked.

```
module type LEXER = sig val lexer: Token.lexer end
```

This encapsulates a lexical analyzer of the type required by Camlp4.

```
module type PRINTABLE_OBJ =
  sig
    type t
    val type_name: string
    val string_of: t -> string
    val make_grammar_entry: Grammar.g -> t Grammar.Entry.e
    val is_equal: t -> t -> bool
  end
```

Everything in this algebraic type system is a printable object. The function `make_grammar_entry` is a parser for things of this type, and `string_of` is a printer. Here every type has an equality predicate. (Actually everything in Ocaml has an equality predicate `=` anyway, and one can always implement `is_equal` using `=`.)

```
module type READ_EVAL_PRINT =
  sig
    type t
    val grammar: Grammar.g
    val main_entry: t Grammar.Entry.e
    val eval : string -> string * t
  end
module RepPrintableObject( Arg :
                       sig module Obj : PRINTABLE_OBJ module Lex : LEXER end ) :
    READ_EVAL_PRINT with type t = Arg.Obj.t =
  struct
    type t = Arg.Obj.t
    let grammar = Grammar.create Arg.Lex.lexer
    let main_entry = Arg.Obj.make_grammar_entry grammar
    let eval str =
      let x = Grammar.Entry.parse main_entry ( Stream.of_string str ) in
      ( Arg.Obj.string_of x, x )
  end
```

A read-eval-print is a meta-level construct consisting of a grammar to hold the grammar entries; the actual entry for this type; and a shorthand function which parses a string and returns a string containing a printed representation of the thing described, as well as the thing itself.

I defined it this way assuming only one top-level read-eval-print would be used, on the top-level object (which subsumes everything else). The grammar would be created and the main entry would be defined using `make_grammar_entry`. During the definition of the main entry, `make_grammar_entry` would be called to make other entries (parsers) lower down in the hierarchy as needed. These would all be stored in the main grammar. However, I am no longer sure whether this scenario is not too restrictive.

```
module type MULTIPLICATIVE_MONOID =
  sig
    include PRINTABLE_OBJ
    val mult: t -> t -> t
    val one: t
  end
```

Here `include` indicates that all multiplicative monoids are printable objects, on which additional structure is defined.

Similarly I defined additive groups, commutative rings with one, ordered rings, and euclidean rings. I defined multi-precision integers to use the Numerix library, which allows the use of GMP, the Gnu Multi-Precision library, or Bigint, another implementation written for Ocaml, or Numerix, the native implementation of this package, all by simply changing module names. (There is a way to set which one is used at run-time, but I did not use this; I just used native Numerix.) I defined rational numbers using the multi-precision integers.

To define polynomials, first I defined an ordered dictionary type as follows:

```
module type OrderedDictionaryType =
  sig
    type key
    type +'a t
    val empty: 'a t
    val size: 'a t -> int
    val split_less_than: key -> 'a t -> 'a t * 'a option
    val split_greater_than: key -> 'a t -> 'a t * 'a option
```

```
val min: 'a t -> key * 'a
val delmin: 'a t -> ( key * 'a ) * 'a t
val max: 'a t -> key * 'a
val delmax: 'a t -> ( key * 'a ) * 'a t
val singleton: key -> 'a -> 'a t
(* the first argument is applied to values in the two dictionaries
associated with the same key, and the result is stored with that key in
the new dictionary *)
val union_merge_values: ( 'a -> 'a -> 'a ) -> 'a t -> 'a t -> 'a t
(* this is the same as above, but values in the two dictionaries
associated with the same key may ``annihilate'' each other, in which
case the key is not bound at all in the new dictionary *)
val union_merge_values_opt:
  ( 'a -> 'a -> 'a option ) -> 'a t -> 'a t -> 'a t
val intersection_merge_values:
  ( 'a -> 'a -> 'a ) -> 'a t -> 'a t -> 'a t
val intersection_merge_values_opt:
  ( 'a -> 'a -> 'a option ) -> 'a t -> 'a t -> 'a t
val difference: 'a t -> 'a t -> 'a t
val mem: key -> 'a t -> bool
val find: key -> 'a t -> 'a
val remove: key -> 'a t -> 'a t
val add: key -> 'a -> 'a t -> 'a t
val add_with_merge_values:
  ( 'a -> 'a -> 'a ) -> key -> 'a -> 'a t -> 'a t
val add_with_merge_values_opt:
  ( 'a -> 'a -> 'a option ) -> key -> 'a -> 'a t -> 'a t
val iter_high_to_low: ( key -> 'a -> unit ) -> 'a t -> unit
val iter_low_to_high: ( key -> 'a -> unit ) -> 'a t -> unit
val map: ( 'a -> 'b ) -> 'a t -> 'b t
val mapi: ( key -> 'a -> 'b ) -> 'a t -> 'b t
val fold_low_to_high: ( 'b -> 'a -> key -> 'b ) -> 'b -> 'a t -> 'b
val fold_high_to_low: ( 'b -> 'a -> key -> 'b ) -> 'b -> 'a t -> 'b
val is_equal: ( 'a -> 'a -> bool ) -> 'a t -> 'a t -> bool
(* two different implementations of union *)
val old_union_merge_values: ( 'a -> 'a -> 'a ) -> 'a t -> 'a t -> 'a t
val old_union_merge_values_opt:
  ( 'a -> 'a -> 'a option ) -> 'a t -> 'a t -> 'a t
```

```
    val hedge_union_merge_values: ( 'a -> 'a -> 'a ) -> 'a t -> 'a t -> 'a t
    val hedge_union_merge_values_opt:
      ( 'a -> 'a -> 'a option ) -> 'a t -> 'a t -> 'a t
  end
```

The `merge_values` and `merge_values_opt` operations require some explanation. The first argument to these operations is a `merge` function. The next arguments are the key-value pair to be added, and the last argument is the dictionary itself. Now if the key is already in the dictionary, the value to be added is merged with the value already present using the `merge` function. In the `_opt` version, the two values may "annihilate" each other. In that case, instead of anything being added, that key and its associated value will be deleted from the dictionary.

I then defined the signature of formal sums:

```
module type FORMAL_BASIS =
  sig
    include PRINTABLE_OBJ
    val zeroth_element : t (* assumed to be 1 *)
    val next : t -> t
    val compare : t -> t -> comparison
  end
module type FORMAL_SUM =
  sig
    include ADDITIVE_GROUP
    type coeff_t
    type basis_t
    val add_term : t -> coeff_t -> basis_t -> t
    val lowest_term : t -> coeff_t * basis_t
    val highest_term : t -> coeff_t * basis_t
    val iter_high_to_low : ( coeff_t -> basis_t -> unit ) -> t -> unit
    val iter_low_to_high : ( coeff_t -> basis_t -> unit ) -> t -> unit
    val fold_high_to_low : ( 'a -> coeff_t -> basis_t -> 'a ) ->
      'a -> t -> 'a
    val fold_low_to_high : ( 'a -> coeff_t -> basis_t -> 'a ) ->
      'a -> t -> 'a
    val coeff_of : basis_t -> t -> coeff_t
  end
```

In this formulation, a formal basis comes in a prescribed order. Two basis elements can be compared, and the basis can be enumerated by starting with the zeroth element and then going to each successive one. I then defined functors `DenseFormalSum`

```
module DenseFormalSum( Arg :
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module Basis: FORMAL_BASIS end ) : FORMAL_SUM
```

and `SparseFormalSum`

```
module SparseFormalSum( Arg :
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module Basis: FORMAL_BASIS end ) : FORMAL_SUM
```

to implement formal sums. The latter uses an ordered dictionary to store the terms. As might be expected, addition of sparse formal sums uses the `merge_values` operations of the ordered dictionary.

I defined an integer sequence to be a dense formal sum of multi-precision integers, together with the sum of those integers. Of course the latter could always be computed, but it is more convenient to store it. I define a term order as follows:

```
module type TERM_ORDER =
  sig
    val name: string
    val compare: IntegerSequence.t -> IntegerSequence.t -> comparison
    val next: IntegerSequence.t -> IntegerSequence.t
    val max_num_vars: int
  end
```

Clearly choice of a term order makes integer sequences into a formal basis. The following functor does so:

```
module MakeIntegerSequenceBasis( TermOrder: TERM_ORDER ) : FORMAL_BASIS
  with type t = IntegerSequence.t
```

Finally we come to the definition of polynomials. I defined an algebra with generators as a formal sum in which the basis elements could be multiplied:

```
module type ALGEBRA_WITH_GENERATORS =
  sig
    include FORMAL_SUM
    val mult: t -> t -> t
    val one: t
  end
```

I defined a type `degree_t`

```
type degree_t = DegInt of MultiPrecInt.t | DegNegInf
```

to be the integers extended by $-\infty$. This uses Ocaml sum type construction. Functions handling degrees can be defined by using pattern-matching on the type constructor, `DegInt` or `DegNegInf`.

I defined functors `DensePolynomialRepresentation`

```
module DensePolynomialRepresentation( Arg:
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module TermOrder : TERM_ORDER end ) : ALGEBRA_WITH_GENERATORS
```

and `SparsePolynomialRepresentation`

```
module SparsePolynomialRepresentation( Arg:
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module TermOrder : TERM_ORDER end )
```

which represent polynomials as formal sums of monomials represented as integer sequences ordered by the term order, together with the total degree.

Now, I defined a functor

```
module SparsePolynomial( Arg:
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module TermOrder : TERM_ORDER end ) =
```

which yields a polynomial with a sparse representation. What I should have done is define a signature `POLYNOMIAL_REP` (more polynomial-specific than `ALGEBRA_WITH_GENERATORS`) which both `DensePolynomialRepresentation` and `SparsePolynomialRepresentation` would satisfy, and take that as an argument. So the functor would be

```
module Polynomial( Arg: sig module Representation: POLYNOMIAL_REP end ) =
```

Then this functor would be invoked as, for example:

```
module SparseLexPoly = Polynomial(
  SparsePolynomialRepresentation(
    struct
      module Coeffs = MultiPrecInt
      module TermOrder = LexicographicTermOrder end ) )
```

Unfortunately, modules are not yet first-class values in Ocaml, although this is an active area of research. So the representation could not be chosen at run-time.

Polynomials are represented as lists of variable names together with the representation. Any operation on two polynomials must merge their variable name lists first before operating on their representations. This involves inserting nulls into the integer sequences representing the monomials to include variables which do not occur in the polynomial.

Reduction of the polynomial to this form is done during parsing. Here is the definition of the parser:

```
let make_grammar_entry gram =
  let coeff_entry = Arg.Coeffs.make_grammar_entry gram in
  let intentry = MultiPrecInt.make_grammar_entry gram in
  let varpower_entry = Grammar.Entry.create gram "variable power" in
  let _ =
  EXTEND
    varpower_entry:
      [ [ var = RSDIdent ->
        ( [ var ], ( Arg.Coeffs.one,
                     ( MultiPrecInt.one, [ MultiPrecInt.one ] ) ) )
        | var = RSDIdent; "^"; exp = intentry ->
        ( [ var ], ( Arg.Coeffs.one, ( exp, [ exp ] ) ) ) ] ];
  END
  in
  let mult_terms ( env1, ( c1, b1 ) ) ( env2, ( c2, b2 ) ) =
    let ( newenv, nulls1, nulls2 ) = merge_vars env1 env2 in
    let newb1 = add_nulls b1 nulls1 in
    let newb2 = add_nulls b2 nulls2 in
    ( newenv, ( Arg.Coeffs.mult c1 c2, IntegerSequence.add newb1 newb2 ) )
  in
```

```
  let term_entry = Grammar.Entry.create gram
    ( Printf.sprintf "multiple of monomial with coefficient from %s"
      Arg.Coeffs.type_name )
  in
  let _ =
  EXTEND
    term_entry:
      [ [ t1 = term_entry; OPT "*"; t2 = term_entry -> mult_terms t1 t2 ]
      | [ c = coeff_entry -> ( [], ( c, IntegerSequence.zero ) )
        | v = varpower_entry -> v ] ];
  END
  in
  let entry = Grammar.Entry.create gram ( type_name ^ " sans negation" ) in
  let _ =
  EXTEND
    entry:
      [ [ e1 = entry; "+"; e2 = entry -> add e1 e2
        | e1 = entry; "-"; e2 = entry -> add e1 ( unaryminus e2 ) ]
      | [ e1 = entry; OPT "*"; e2 = entry -> mult e1 e2 ]
      | [ term = term_entry -> add_term_with_env zero term
        | "("; e = entry; ")" -> e
        ] ];
  END
  in
  let entrywithnegation = Grammar.Entry.create gram type_name in
  let _ =
  EXTEND
    entrywithnegation:
      [ [ "-"; term = term_entry; "+"; e = entry ->
          unaryminus ( add_term_with_env ( unaryminus e ) term )
        | "-"; term = term_entry; "-"; e = entry ->
          unaryminus ( add_term_with_env e term ) ]
      | [ e = entry -> e
        ] ];
  END
in entrywithnegation
```

The parser uses the parsers for the coefficient ring and for the integers at the

lowest-level. Then a variable power is a variable name, optionally raised to an integer power. (Technically this allows Laurent polynomials, i.e., negative powers, but I have not tested whether this works.) A term is either a coefficient, or a variable power, or several of these optionally separated by `'*'` characters. In the last case, the variables are merged and the factors are multiplied. Thus, variables are accumulated as the parser goes through the stream of characters. The `entry` parser accepts a sum, difference, or product of terms; division and exponentiation are not allowed. Finally, the `entrywithnegation` parser deals with an *initial* unary minus sign. It took me some trouble to get this right: because juxtaposition denotes multiplication (the `'*'` characters are optional), at first subtraction of a term was mistakenly parsed as multiplication by the unary negation of the term.

Last but not least, I defined substitution, of which evaluation is a special case. I defined rewrite rules to have type: a variable name together with a polynomial, the expression to be substituted for the variable. (I only implemented variable substitution.) I wrote a function `use_rule_in` to apply a rule to a polynomial. First it checks whether the variable occurs in the body and raises an exception in that case. Otherwise, it makes the rule and the argument (of the application) have the same variables and checks whether the variable to be substituted occurs in the argument. If so, it reformulates the argument as a list of pairs of polynomials (in which the variable does *not* occur) and integers, the powers of the variable. In other words, it represents the argument in the polynomial ring in that one variable over the ring of polynomials in the rest of the variables. Then it starts exponentiating the body, multiplying it by itself one factor at a time. As it does so, it goes down the list, multiplying the current power by its coefficient in the polynomial ring in the rest of the variables and summing the result into an accumulator.

Finally, I defined `SparsePolynomialWithSubstitutions`

```
module SparsePolynomialWithSubstitutions( Arg:
  sig module Coeffs: COMMUTATIVE_RING_WITH_ONE
      module TermOrder: TERM_ORDER end )
```

to have the same type as `SparsePolynomial`, but to allow the use of rewrite rules. The syntax is `let x -> body in expr`.

8 Critique of My Polynomial Arithmetic Implementation

There are several problems and issues with this implementation. The most fundamental problem is the integration of an ordering at every level of the representation. When I first implemented this I used balanced binary trees for the ordered dictionary. I used Stephen Adams's implementation of ordered sets based on binary search trees of bounded balance, for which he cites [NR73]. I modified it to implement dictionaries, with the `merge_values` functionality. (Incidentally, he implemented a "lazy" version of union called `hedge_union`, which he describes as "much more complex and usually 20% faster." Including the `merge_values` functionality made this function even more complex, and it would be interesting to study whether it is still any faster than the simple version.) As often happens when trying to design a system with generality and future evolution in mind, I failed to see how this implementation choice was influencing my design of the modules. I will say more about this below.

I also should check how much my abstraction and generality may be costing me in speed. For example, all polynomial arithmetic operations are done using `fold_low_to_high` or `fold_high_to_low`, which are implemented in the ordered dictionary. Would operating directly on the data structures allow some optimizations not available in this formulation? I should run some benchmarks.

My use of multi-precision integers for the powers of variables is extremely questionable. Powers of variables which do not fit in a machine integer are seldom seen in practice. Of course, the native Numerix implementation defaults to machine integers until overflow occurs, as most implementations probably do, but still some overhead is probably involved. I should measure the slowdown.

As mentioned before, my placement of the grammar which contains all the parsers may not be optimal. If one read-eval-print is defined for integers, and another for polynomials with integer coefficients, then the integer parser will be defined twice, once in each grammar. On the other hand, I am not sure how to reformulate this.

9 Schweighofer's Algorithm Revisited

As mentioned previously, Schweighofer's algorithm quickly leads to polynomials with hundreds of thousands of terms. These took longer and longer to deal with, and eventually caused computer algebra systems to run out of memory. It is likely that most computer algebra systems use sparse polynomial representations internally, since most polynomials seen in practice are sparse. However, these particular polynomials are dense, and so a dense representation should be used.

Considering that my computations were running out of memory and that hard disk space is cheap, I thought that larger computations could be successfully completed if portions of the polynomial were somehow written to disk during the computation. At first I thought the whole polynomial might be stored in a relational database. A good choice might be MySQL, which is tuned for performance and lackadaisical about transaction bureaucracy. I was always multiplying by the same factor $Y = Y_1 + \cdots + Y_{2n}$. The coefficient of a monomial $Y_1^{\varepsilon_1} \cdots Y_{2n}^{\varepsilon_{2n}}$ in g_i is the sum of the coefficients of $Y_1^{\varepsilon_1 - 1} \cdots Y_{2n}^{\varepsilon_{2n}}$ up to $Y_1^{\varepsilon_1} \cdots Y_{2n}^{\varepsilon_{2n}-1}$ in g_{i-1}. A database index should be defined such that this block of $2n$ terms could be retrieved rapidly.

However, I eventually decided that this was not necessary. Portions of the polynomial should only be written to disk when memory was almost full, since access to memory is much faster than access to disk. So polynomials should be represented by a data structure that could easily be fragmented, with portions residing on disk and portions residing in memory. A B-tree is a usual example of such a data structure.

On the other hand, hash tables appeared to be an attractive alternative. Access to a value in a tree takes time proportional to the log of the depth, whereas access to a value in a hash table takes essentially constant time (until the hash table becomes too full). A hash table can essentially be considered as a sparse vector. To fragment it, this sparse vector should be partitioned at various points. Another level of indirection is required: a sorted list of starting and ending points, with a reference to where the hash table for monomials between those points is stored, whether on disk or in memory. But then this list itself must be searched, apparently by binary search. This search will take time proportional to the log of the length of the list. It seems likely that the list would be wider than the B-tree would have been deep, but it would require experiment to be sure.

In this connection I also looked at the minos database implemented as

part of FORM, which also writes intermediate results of computations to disk. Specifically, it is used for calculations with Feynman diagrams. The database code implements low-level operations writing blocks to disk. It seems to me it could be easily replaced by any other ISAM database, although it has better documentation and support specifically for Feynman diagram calculations.

It occurred to me that the virtual memory manager is already swapping out portions of memory to disk when memory gets full. So the implementation used here would have to do better than the virtual memory manager to be worthwhile.

Regardless of whether computations are written to disk or not, the abovementioned performance characteristics of hash tables make them an attractive solution for implementing polynomial arithmetic. However, this is where the ordering of monomials plays a crucial role. The only way to retrieve elements of the hash table in a particular order is to look for each one specifically by key, one by one in that same order. Even if the polynomial is sparse, a number of lookups must be performed as if it were dense. Furthermore, some term orders do not admit enumerating their elements one by one without further information. For example, in the lexicographic term order with $x > y$, the successive monomials are $1 < x < x^2 < x^3 < x^4 < \cdots$. One would never reach any monomials involving y. Thus, one has to store the total degree of the polynomial. On the other hand, in the degree lexicographic term order with $x_1 < x_2 < \cdots$, the successive monomials are $1 < x_1 < x_2 < x_3 < x_4 < \cdots$. One would never reach any monomials of degree 2. Thus, one has to store the number of variables.

These problems would be alleviated by no longer considering a polynomial as a collection of monomials, but rather using a recursive representation, where each hash table involves only one variable. This could be accomodated in my algebraic type system, if modules were first-class values. As it is, however, each coefficient ring, a polynomial ring in the previous variables, has to be declared by hand. Thus polynomial rings in $1, 2, 3, \ldots$ variables would have to be declared beforehand.

Incidentally, the above considerations give insight as to why polynomials in Maple appear in a different order depending on what computations were done previously, and why in Singular one must fix the number of the variables before doing any computations. I know Singular and I believe Maple both use hash tables to represent polynomials.

10 Schweighofer's Algorithm Critiqued

Despite the above, I did reimplement Schweighofer's algorithm using the polynomial arithmetic implementation I had, making liberal use of substitution rules. At the same time, I used Fukuda's `cdd+` for exact linear programming. With a few little shell scripts and a Python script, I fully automated the process. The polynomial arithmetic would take longer and longer, but didn't crash.

However, from my experience and after some discussion with Pablo Parrilo I realized that Schweighofer's algorithm itself had some drawbacks. One must somehow choose an initial representation for f. This choice has a substantial effect on the bounds achieved. Furthermore, only homogeneous representations are allowed. Any restriction on the form of the representation may result in a worse bound being found (although Schweighofer proved that this representation will come arbitrarily close to the actual minimum, provided the degree is some very large number). Although Schweighofer used his algorithm to prove the existence of a Handelman representation constructively, it was not necessary to use it to *find* such a representation. Instead I looked for it directly by linear programming. At this point I found that `cdd+` would fail before the polynomial arithmetic was overwhelmed. Just a couple of days ago I learned that this is probably due to some hardcoded limitation compiled into `cdd+` and not for any algorithmic reason. I should recompile `cdd+` and try it again soon.

11 References

[And76] George Andrews, *The Theory of Partitions*, 1976.

[bastat] `http://www-sop.inria.fr/croap/personnel/Loic.Pottier/Bastat/bastatdemo.html`

[cdd] `http://www.ifor.math.ethz.ch/ fukuda/cdd_home/cdd.html`

[GP01] Karin Gatermann and Pablo Parrilo, "Symmetry groups, semidefinite programs, and sums of squares". Preprint, to appear.

[GPS01] G. M. Greuel, G. Pfister, and H. Schönemann, "Singular 2-0-0. A Computer Algebra System for Polynomial Compu-

tations." Centre for Computer Algebra, University of Kaiserslautern (2001). http://www.singular.uni-kl.de.

[NEOS] http://www-neos.mcs.anl.gov

[NR73] Nievergelt and Reingold, SIAM Journal of Computing **2**(1) (March 1973).

[Ocaml] http://www.ocaml.org

[Par00] Pablo Parrilo, *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*, California Institute of Technology Ph.D thesis, 2000.

[Par01] Pablo Parrilo, "Semidefinite programming relaxations for semialgebraic problems", Preprint, 2001.

[PS01] Pablo Parrilo and Bernd Sturmfels, "Minimizing Polynomial Functions. Preprint, 2001. math.OC/0103170

[Rez78] Bruce Reznick, "Extremal psd forms with few terms," Duke Math. J. **45** (1978), 363–374.

[Rez00] Bruce Reznick, "Some Concrete Aspects of Hilbert's 17th Problem," *Real Algebraic Geometry and Ordered Structures*, 2000, 251–272.

[SDPA] ftp://plato.la.asu.edu/pub/sdpa_format.txt

[SOS] http://www.cds.caltech.edu/sostools/