

There is a strong analogy between the way how these algorithms employ straight-line programs and the way how functional programming languages treat functions as first-class citizens. Taking advantage of this circumstance, the MILONGA language enables us to analyze the relevance of the functional programming paradigm for the particular kind of task of polynomial equation solving.

The paper contains an exhaustive do-it-yourself description of the programming philosophy of MILONGA, of the development of its compiler, of the operational semantics of its run-time system and of the implementation of a couple of fundamental computer algebra procedures in this language.
The practical efficiency of this philosophy and implementation is outlined by comparative benchmarking on significant test examples.

© 2002 IMACS. Published by Elsevier Science B.V. All rights reserved.

Keywords: Symbolic computation; Polynomial equation solver; Abstract machine

1. Introduction

General purpose computer algebra packages—as e.g. Maple, Mathematica, AXIOM, Macaulay, CoCoA, Magma—contain tools for the infinite precision manipulation of integers, rationals, polynomials and rational functions as fundamental ingredients. Usually polynomials and rational functions are represented in sparse (or dense) form by their coefficients, whereas integers and rationals are represented bitwise.

Unfortunately, in geometric and algebraic elimination procedures—aimed at multivariate polynomial equation solving—these representations lead to a superexponential intermediate expression swell (see e.g. [9,10,36,37]). The most common elimination procedures tend to produce dense exponential degree expressions from sparse input polynomials of low degree. A similar thing happens to the integers involved in the process.

For objective geometric and arithmetic reasons, formalized in the corresponding versions of the Bézout theorem, it is impossible, at least in worst case, to avoid the exponential increment of degree and bit length (logarithmic height) of the polynomials and integers involved in any elimination process. This means that any elimination procedure based on the traditional sparse or dense representation of multivariate polynomials has necessarily a superexponential sequential time complexity, at least in worst case. Using the more problem adapted representation of polynomials by straight-line programs—or arithmetic circuits—this superexponential complexity behaviour may be reduced to a singly exponential one [15,29]. Moreover, it is possible in this context to design incremental algorithms which distinguish well-suited geometric and arithmetic situations—with low degrees and logarithmic heights—from bad ones [13,16,18,23].

MILONGA is an acronym for modular implementation language oriented to nonlinear geometry applications. This language is aimed at supporting the implementation of the new elimination procedures mentioned before. As we represent polynomials by means of straight-line programs, we interpret them as functions and not as symbolic expressions. This view suggests the use of the functional programming paradigm. Following this idea, and using the straight-line program representation of polynomials and integers as basic data structure, the MILONGA language allows the implementation of most fundamental subalgorithms commonly used for polynomial equation solving. In particular, MILONGA is expressive enough in order to allow the implementation of the elimination algorithms in [13,16,18,23].

For its execution, a given MILONGA program is mapped into C++ code. This is done by a compiler written in Haskell [38], a nonstrict functional programming language. Nonstrict functional programming languages in general, and Haskell’s monads [26] in particular, are well suited for the development of compilers. Thanks to this tool, namely Haskell’s monads, it took only one man-month to develop the whole MILONGA compiler.

The design of the MILONGA compiler is based on an abstract machine, which on its turn is inspired in the G-machine of Johnson [27], well known in the functional programming community. The use of this kind of machine allows the application of very specific, performant optimization methods. As a consequence, our compiler transforms a given MILONGA program in highly optimized C++ code.
The abstract MILONGA machine represents a theoretical computer model for the execution of MILONGA programs. The machine starts working with an initial state which codifies the given MILONGA program and evolves changing step by step its state until a final state is reached or the computation stops by lack of commands. In the first case, the final state codifies the result of the (successful) execution of the MILONGA program under consideration, whereas in the second case a run-time error is returned. The MILONGA compiler contains a C++ implementation of the working mechanism of this abstract machine. This allows the simulation of the abstract MILONGA machine on any computer which permits running of C++ programs.

Let us now explain what the compiler does with a given MILONGA program. The program is first transformed into a semantically equivalent intermediate representation (code) called pCode. To this intermediate pCode representation there corresponds a well-defined initial state of the abstract MILONGA machine and this allows the theoretical "execution" of the pCode representation (and hence of the original MILONGA program) on this machine. The compiler translates then the given pCode representation into C++ code. Using the above C++ simulation of the abstract MILONGA machine we are now able to run the given MILONGA program on any real world computer equipped with a C++ compiler.

An alternative approach to programming with straight-line programs focuses the attention on the syntactical—and not on the functional—aspect of this concept. This view is realized in the computer implementations [6,7,12,14]. In these implementations straight-line programs are explicitly stored as directed acyclic graphs (dag’s).

As pointed out in [10] (compare also [14]), the explicit representation of straight-line programs by their underlying dag’s is too expensive for an efficient treatment of real world sized problems of polynomial equation solving. However, elimination algorithms do not usually require the explicit storage of the functions they work with, but rather the evaluation of these functions on well-specified numerical input arguments. This observation leads in [10] to the realization of the straight-line program concept in the computational model of black boxes. A black box represents an implicit codification of a straight-line program by means of an algorithmically generated evaluation process which allows a space economic computation of the value of the underlying function on any given numerical input argument. The theoretical black box concept is realized in [10] by means of object-oriented programming techniques. The MILONGA language exploits the same idea using the functional programming paradigm.

For the design of MILONGA it was necessary to have in mind the following considerations and requirements we are going to list now, together with their justifications:

- **Straight-line programs are the main object our algorithms deal with.**
  - As mentioned before, the MILONGA language was designed in order to support the basic data structures and elimination algorithm developed in [13,16] (see also [18,23]). The main theoretical outcome of this algorithm is the insight that suitable projections of so-called reduced algebraic complete intersection varieties are hypersurfaces whose minimal equations have “short” straight-line program representations. This means that such a hypersurface equation has a straight-line program representation of length polynomial in the degree of the hypersurface (whereas its sparse or dense representation may be exponential in that quantity). For this reason it is mandatory that MILONGA is able to deal with objects like straight-line programs. Moreover, the language must be able to receive straight-line programs as inputs, because of the recursivity of the elimination algorithm for which MILONGA was designed. This requires to treat straight-line programs as first-class citizens which may occur as inputs or outputs of the procedures implemented in MILONGA.
- **Our algorithms require partial evaluation of straight-line programs.**
  Partial evaluation is of common use for greatest common divisor computations or factorization of multivariate polynomials. It is generally applied in order to reduce the multivariate problem under consideration to the univariate or bivariate case. In [14] this tool is also used for the efficient computation of the dimension of projective varieties. The algorithm whose implementation by MILONGA we have in mind makes an extended use of this tool.

- **Our algorithms require a kind of polymorphism.**
  In order to avoid intermediate expression swell, we have to make use of efficient forms of modular computations. Typical applications of modular computations are related to the Chinese remainder theorem or to Hensel’s lemma (see [9,47]). For the application of this technique we need the evaluation of straight-line programs representing algebraic objects (as integers, rationals, polynomials or rational functions) in certain (generally modular) commutative rings. Such rings may appear as the integers modulo a given integer, as the univariate polynomials over the integers (or the rationals) modulo a given univariate polynomial or also as certain commutative matrix algebras.

In contrast to these considerations and requirements we had also to take into account the following points:

- **The necessity of internal program transformations.**
  This requirement appears in the context of automatic differentiation of straight-line programs (see [3] or [20]). MILONGA was designed for an elimination algorithm which applies Newton-type methods for the computation of symbolic and numeric approximations to the solutions of polynomial equation systems. Therefore, we need automatic differentiation of straight-line programs in order to compute the Jacobian of a multivariate polynomial equation system.

- **The necessity of changes of arithmetic within the same program.**
  The modular methods we are going to use in our procedures require a permanent switching from integer to modular arithmetic.

- **As far as possible we should try to profit from implementation work already done by means of imperative programming.**
  The elimination algorithm at the origin of MILONGA uses substantially loops, branchings and destructive assignments. This clashes with the functional programming philosophy and requires therefore a specific treatment in MILONGA.

We tried to satisfy these two contrasting groups of requirements in the design of the MILONGA language and its compiler.

Efficient algorithms do not lead automatically to efficient computer implementations. Therefore, we shall show by experimental benchmarks and by theoretical arguments that the MILONGA language allows the implementation of straight-line programs based elimination algorithms in such a way that their practical behaviour can be predicted by their asymptotic complexity. Our arguments shall make substantial use of the particular syntactical structure and the semantics of these algorithms.

Summarizing, with the development of the MILONGA language and its compiler we aimed to provide a programming environment for the implementation of a new generation of elimination algorithms. MILONGA was designed with a functional programming philosophy, i.e. functions are first-class citizens of the language without side effects. A particular feature of MILONGA is its capability of partial evaluation and differentiation of functions and of imperative programming within them.
In what follows, functions will always be defined as sequences of assignments of arithmetic expressions to state variables. This implies that a function with instantiated state variables becomes a straight-line program in the usual sense. Therefore, we shall use from now on the terms “function” and “straight-line program” in the following way: “function” will mean a suitably indexed family of straight-line programs, whereas with “straight-line program” we shall refer to a suitably instantiated function.

This paper is organized as follows. In Section 2 we present the syntax and semantical structure of the MILONGA language and their programs. The definition of functions, modular arithmetic and automatic differentiation in MILONGA are described and illustrated by examples.

In Section 3 we describe the MILONGA compiler and the tasks it solves. For this purpose we develop in analogy to Johnsson’s $G$-machine [27] a specific, task adapted abstract machine. Special attention is paid to type-checking mechanisms, automatic differentiation, generation of intermediate code, to the static structure of the abstract MILONGA machine and to the heap management.

In Section 4, the initial, intermediate and final states of the abstract MILONGA machine are specified and, by means of a list of instructions, the state transition rules are introduced. Then we analyze—under certain restrictions—the execution time complexity of the C++ compilation of a given MILONGA program in terms of the syntactical structure of the program and its semantics. This allows to predict the practical complexity of MILONGA implemented theoretical elimination algorithms.

Finally, in Section 5 we describe the implementation of a couple of fundamental computer algebra procedures in the MILONGA language. The practical outcome of these implementations is outlined by comparative benchmarking on significant test examples.

1.1. Computational models and programming philosophy

Before starting the description of the syntax and semantics of the MILONGA programming language we shall briefly sketch the aim for which MILONGA was developed: the implementation of straight-line program based elimination algorithms.

Let us visualize the fundamental idea by the following classical elimination problem: let us be given a generic matrix $A = (A_{ij})_{1 \leq i,j \leq n}$ with $A_{11}, \ldots, A_{nn}$ being indeterminates over $\mathbb{Q}$. The matrix $A$ determines a homogeneous linear equation system which can be expressed by the following first-order formula $\Phi(A)$ containing the entries of the matrix $A$ as free variables and the indeterminates $X_1, \ldots, X_n$ as bounded variables:

$$\exists X_1 \cdots \exists X_n ((X_1 \neq 0 \lor \cdots \lor X_n \neq 0) \land \sum_{i=1}^{n} A_{ij} X_j = 0)$$

(1)

This formula $\Phi(A)$ is equivalent to the following quantifier-free formula $\Psi(A)$:

$$\det(A) = 0$$

where

$$\det(A) := \sum_{\sigma \in \text{sym}(n)} \text{sign}(\sigma) A_{\sigma(1)} \cdots A_{\sigma(n)}$$

is a polynomial with $n!$ terms which belongs to the polynomial ring $\mathbb{Q}[A_{ij}; 1 \leq i, j \leq n]$. 
The usual way to verify (in a suitable computation model) the validity of the formula \( \Phi(A) \) for a given specialization of the generic matrix \( A \) into a numerical matrix \( a = (a_{ij})_{1 \leq i,j \leq n} \in \mathbb{C}^{n \times n} \), consists in the computation of \( \text{det}(a) \) and the decision whether \( \text{det}(a) = 0 \) holds. If we write down \( \text{det}(A) \) in the traditional manner as sparse polynomial in the entries of the matrix \( A \), we replace the formula \( \Phi(A) \), which has length \( O(n^2) \), by a formula of length \( O(n!) \), namely \( \Psi(A) \). Instead of proceeding in this way we shall represent in this paper the polynomial \( \text{det}(A) \) by an algorithm which computes the value of \( \text{det}(A) \) from the inputs \( A_{11}, \ldots, A_{nn} \). In order to realize this idea one may think to use as algorithm Gaussian elimination, arriving in this way to a new representation of the formula \( \Psi(A) \) of length \( O(n^3) \). Since Gaussian elimination involves divisions (and therefore branchings), we shall proceed in this paper in another way. In order to avoid divisions we shall choose the division-free procedure of Samuelson (see e.g. [11]) which allows us to represent \( \text{det}(A) \) (and hence \( \Psi(A) \)) by a (branching-free) straight-line program of length \( O(n^4) \) (see Section 5.1 for more details and a MILONGA implementation).

Of course, this basic idea may be applied to the representation of the resultant of two generic univariate polynomials and to many other elimination problems in computational algebraic geometry.

The idea of using algorithms, and in particular straight-line programs, for the representation of mathematical objects of big size (as e.g. polynomials occurring in elimination procedures) appeared explicitly for the first time at the beginning of the 1980s (see e.g. [24,25] and the references cited in [16]), moreover it can be traced back at least until Newton. Initially this idea was only applied to problems dealing with the elimination of a single variable. The first successful application of the idea to the elimination of an arbitrary number of variables was made in [16]. This progress was achieved by the introduction of a global, symbolic version of Newton’s algorithm which allows successive data compression during the recursive elimination procedure (see also [13,18,22,23]).

Many typical problems of elimination theory depend not only on continuous but also on certain discrete parameters, such as the number of (continuous) variables involved in the problem or some suitably defined “degree”. For example, in the case of the family of elimination problems (1), the matrix size \( n \) determines the concrete elimination problem under consideration. Once the relevant discrete parameters are fixed, the concrete elimination problem becomes completely determined. In the context of this paper, the algorithmic solution of a given concrete elimination problem will always be reduced to the evaluation of suitable straight-line programs. This leads us to the consideration of families of straight-line programs, depending on discrete parameters.

In the MILONGA language, a family of straight-line programs is represented by a MILONGA function (see Section 2). A MILONGA function is a sequence of instructions which depends on two types of parameters: fixed and extra. The fixed parameters determine the “concrete” shape of the MILONGA function under consideration, and correspond to the discrete parameters of the family of straight-line programs represented by that function. The extra parameters correspond to the “continuous” variables of the family of straight-line programs under consideration.

The evaluation of a MILONGA function is modeled by means of an abstract machine, inspired in the G-machine of Johnsson [27], which we call the abstract MILONGA machine. The abstract MILONGA machine operates on vectors of fixed length which represent the initial, intermediate and final states of the machine. These state vectors encode the instruction sequence of the given MILONGA program and certain memory addresses required for its execution.

To a given MILONGA program there corresponds a well-defined initial state of the abstract MILONGA machine. From this initial state as input, the machine evolves, changing at each step its state, until a
final step is reached or the computation stops by lack of suitable commands. The working mechanism of the abstract MILONGA machine is determined by a list of (nondeterministic) transition rules whose application becomes deterministic by the use of certain working instructions.

The C++ compilation of a given MILONGA program proceeds in two stages. In the first stage, the given MILONGA program is transformed into a semantically equivalent intermediate representation called pCode. For this purpose, in a first step lexing and parsing of the original MILONGA program are performed and an abstract syntax tree, called slpAbs, is generated. Then, if necessary, automatic differentiation is performed on the abstract slpAbs syntax tree and the resulting code is optimized. Finally, the intermediate pCode representation of the MILONGA program under consideration is generated.

In the second stage, the given pCode representation is translated to a semantically equivalent C++ object code. This stage is performed by a compiler written in the functional programming language Haskell. The MILONGA compiler contains a C++ implementation of the working mechanism of the MILONGA abstract machine.

Finally, the C++ execution of the resulting C++ object code is performed. For this purpose, the object code is linked to the C++ library which simulates the abstract MILONGA machine.

2. The MILONGA language

A MILONGA program has a modular structure. We call its components modules. A MILONGA program contains also a distinguished function main that triggers the computations.

2.1. Modules

MILONGA allows modular programming by code abstraction. MILONGA modules offer namespace management and separate computation. They are used for the encapsulation of implementation details and for the protection against unexpected access to unexported functions.

A MILONGA module consists of the following constitutive elements:

- A name for the module.
- A list of names of functions to be exported.
- A list of function definitions which includes the functions for exportation.

2.2. MILONGA functions

In this section we explain how the MILONGA language represents functions and operates with the partial evaluation of functions.

MILONGA functions are equipped with two types of parameters which we call fixed and extra. The fixed parameters determine the concrete “shape” of the function under consideration. Once the shape of a function is fixed, this function depends on certain arguments which we call extra parameters. Fixed and extra parameters occur never bounded in function definitions.

For example, let A be a subprogram of a MILONGA program which represents a univariate polynomial function $G$ depending on a certain parameter $B$, and suppose that $B$ may represent any polynomial function which on its turn depends on certain program variables. For any instantiation of the program parameter $B$ by a concrete polynomial, say $F$, in given variables, say $X_1, \ldots, X_n$, the program variable
A represents the composite function $G(F(X_1, \ldots, X_n))$. In this situation we shall consider the name of the polynomial $F$ as a fixed parameter, and the variables $X_1, \ldots, X_n$ as extra parameters.

The definition of a function in MILONGA requires the declaration of its fixed and extra parameters. Whereas any MILONGA function contains a constant number of fixed parameters, the number of its extra parameters may depend on the concrete instantiation of the fixed parameters. We may use arbitrary names in order to denote fixed parameters, whereas extra parameters have to be denoted by the letter “X” labeled by an ordinal number. The range of this ordinal number is determined by the quantity of extra variables of the MILONGA function under consideration. Therefore, we have to provide any MILONGA function with an expression which allows the computation of the number of extra parameters for a given instantiation of its fixed parameters. This permits MILONGA to operate with functions which contain an indeterminate number of arguments.

MILONGA allows selective instantiation of function parameters in a “left to right” fashion, according to the order of declaration. A consequence of this “left to right” rule is that extra parameters cannot be instantiated before all fixed parameters have their meaning assigned.

We may visualize graphically a MILONGA function as a box $N$ as follows (see Fig. 1):

- $F_1, \ldots, F_k$ are $k$ fixed parameters.
- $E$ is an integer-valued expression depending on the fixed parameters $F_1, \ldots, F_k$. The value $E$ for each instantiation of the parameters $F_1, \ldots, F_k$ determines the number $m$ of extra parameters of the function under consideration.
- $X_1, \ldots, X_m$ are the $m$ extra parameters (here we suppose that we have already assigned a concrete meaning to the fixed parameters $F_1, \ldots, F_k$, determining in this way the number $m$ by means of the expression $E$).
- $L_1, \ldots, L_k$ are local program variables.
- $C$ denotes the function body.
- $R$ denotes the output value of the function under consideration.

Usually, the body of a MILONGA function is written in an imperative language style, using assignments, branchings, loops and specific commands for changes in the underlying arithmetic. It is also possible to generate automatically the first-order derivative of a MILONGA function.
The distinction between fixed and extra parameters is motivated by the interest in having partial evaluation features. In static programming languages one usually chooses a fixed number of arrays for the task of representing a given function with moving number of arguments, such as e.g. the inner product of two vectors of indeterminate length. This kind of representation is incompatible with partial evaluation. For example, if \( n \) is a program parameter and the program variables \( X_1, \ldots, X_n \) are represented by an array of dynamic length \( n \), partial evaluation of the variables \( X_1, \ldots, X_n \) is not realizable in a static programming language. One is obliged to instantiate all variables \( X_1, \ldots, X_n \) simultaneously or none of them.

Fixed parameters of MILONGA functions may be specialized into partially evaluated functions. Moreover, MILONGA procedures are able to return (possibly partially evaluated) functions as outputs.

Another important point is the following: MILONGA programs do not contain global variables and MILONGA functions are free of side effects, their parameters are passed by value and they cannot be modified during the execution of a program. Therefore, it is guaranteed that successive calls to the same function with the same parameters return always the same result. This means in particular that MILONGA functions are referentially transparent. The modular environment (see Section 2.3.2 for details) has to be considered as an additional parameter.

2.2.1. Primitives

Using primitives it is possible to include C++ code in a MILONGA program. A primitive is specified by a name, a list of parameters (both name and type must be declared for each parameter) and the type of the result. The following is an example of the syntax of a primitive:

\[
\text{primitive random (i:TInt):TInt}
\]

Primitives cannot contain extra parameters. Primitives should be introduced with care, since they may destroy the referential transparency of MILONGA functions.

2.3. The main features of the MILONGA language

MILONGA’s syntax can be found in [5]. In this section we shall only outline its most relevant properties.

2.3.1. Data types

MILONGA supports the following data types:

- **TBool**: Boolean values True and False.
- **TInt**: Bounded integers as the native integers provided in any computer. Their length depends on the particular computer on which we run MILONGA.
- **TNum**: Unbounded integers of infinite precision. They can be combined with modular arithmetic. In order to operate with the type TNum, the LiDIA library is used [31].
- **TVec**: Vectors whose elements are of the type TObj below. Their dimension may be specified dynamically.
- **TMat**: Matrices whose elements are of the type TObj below. Their dimensions may be specified dynamically.
- **TObj**: Objects. Elements of this type may contain, in run-time, values of type TNum, TVec or TMat.
- **TSlp t**: Functions that return values of type \( t \), where \( t \) is any of the types TNum, TVec, TMat or TObj.
2.3.2. Algebraic operations in MILONGA

MILONGA supports the usual arithmetic, vector and matrix (i.e. algebraic) operations over \( \mathbb{Z} \) and over the univariate polynomial ring \( \mathbb{Z}[X] \), keeping their usual meaning. Operations may be executed in infinite or finite precision mode and the user controls completely which mode is going to be applied. For this purpose, MILONGA provides the user with the modular change commands CRV expression and CRV vector defining the dense representation of the remainder of the division of \( f(X) \) is understood as the multiplication of \( f_1(X) \) and \( f_2(X) \) modulo \( G(X) \), i.e. the vector representing the element \( v \) by the deg \( G \)-dimensional vector defining the dense representation of the remainder of the division of \( f(X) \) by \( G(X) \). Therefore, the multiplication of two vectors \( v_1, v_2 \), representing the equivalence classes defined by integer polynomials \( f_1(X), f_2(X) \) in the ring \( \mathbb{Z}[X]/(G(X)) \) is understood as the multiplication of \( f_1(X) \) and \( f_2(X) \) modulo \( G(X) \), i.e. the vector representing \( v_1 \ast v_2 \) will be the dense representation of the remainder of the division of the polynomial \( f_1 \ast f_2 \) of \( \mathbb{Z}[X] \) by the polynomial \( G(X) \).

The most basic vector and matrix operations are implemented and available in our MILONGA library. These implementations are based on nowadays most efficient theoretical algorithms in the algebraic complexity model. For their execution these implementations are compiled into C++ code and executed for infinite or finite precision integers using the arithmetic of the LiDIA library. The critical operation with \( h \)-bit numbers is their multiplication, whose theoretical complexity is denoted by \( m(h) \) (e.g. \( m(h) = \Omega(\log h \log \log h) \) using the Schönhage–Strassen algorithm [41]). The corresponding execution time in the LiDIA library is denoted by \( M(h) \) (see Section 4.3 for a detailed discussion of the execution time of MILONGA programs).

Let us finally point out the following syntactical details of the MILONGA language:

- In any operation involving a number and a vector (or a number and a matrix), the number under consideration is considered as a scalar multiple of the neutral element of the type \( \text{TV} \) ("vector"), i.e. as a scalar multiple of the constant polynomial 1 (or of a scalar multiple of the identity matrix).
For example, the expression $3 + m$ with $m$ being a matrix variable of the type $\text{TMat}$ (with $n$ rows and $n$ columns), is interpreted as $3 \ast I_{n \times n} + m$ (with $I_{n \times n}$ denoting the $n \times n$ identity matrix).

- For the determination of the dimension of a vector, MILONGA uses a function denoted by $\#v$. There are also available functions $\#r$ and $\#c$ that return the number of rows and columns of a matrix, respectively, and a function $\#e$ that returns the number of extra variables of a given MILONGA function.

2.3.3. Automatic differentiation

Automatic differentiation is a process which depends on a given algorithmic specification of the function to be differentiated. This process transforms computer programs by means of a systematic application of the rules of calculus and it is usually implemented either by source code manipulation or operator overloading. A detailed description of automatic differentiation can be found in [3,20].

Automatic differentiation is based on the fact that any function realized by a computer program consists of a sequence of arithmetic operations and calls to elementary procedures. Applying repeatedly the chain rule to the composition of elementary procedures, it is possible to compute the derivative of arbitrarily complex functions. Automatic differentiation may be applied to any routine using branchings, loops and function calls, even when no closed formula is available. Automatic differentiation is a code transformation process which allows to transform a given code evaluating a function $f$ into a code evaluating some (or all) first-order derivatives of $f$.

Automatic differentiation is more powerful than simple symbolic differentiation. Although symbolic manipulators such as Maple and Mathematica are able to compute the derivative of a function given by a closed formula, they are unable to do this with procedures which contain branchings, loops and function calls.

Only a very limited use of automatic differentiation has been admitted by MILONGA: for $f$ being the name of a MILONGA function and $e$ being the index of one of its variables (extra parameters), the expression $\text{derive } f \ e$ calls a procedure which returns the first-order partial derivative of the function $f$ with respect to the $e$th variable. The type of this expression is $\text{function}$.

For instance, if $f$ is a function in two variables $X$ and $Y$, then the expression $\left(\text{derive } f \ 1\right) \ 2$ represents the univariate function which, for any input value $y$, returns the value $(\partial f / \partial X)_{X=2,Y=y}$.

Let $P$ and $P'$ be MILONGA functions such that $P'$ is generated from $P$ by automatic differentiation. For any variable $X$ occurring in $P$—which may be a fixed parameter, an extra parameter or a local variable—there will appear two variables $X$ and $X'$ in the definition of the MILONGA function $P'$. The construction of $P'$ is guided by the following rule: if the precondition of $P'$ asserts $X' = \partial X / \partial T$ then $r$ and $r'$, the evaluation results of the procedures $P$ and $P'$, satisfy the condition $r' = \partial r / \partial T$. The algorithm used to generate the code for the derivative is the simple “forward” mode (see [39] for details).

The names of all the functions to be differentiated must be bound in compiling time. Therefore, none of the parameters of the function $P$ is allowed to be itself a MILONGA function.

We shall exhibit in Section 3.2.1 the automatic differentiation rules of MILONGA.

2.3.4. Multiple function applications in MILONGA

In order to define multiple applications of functions to variables, MILONGA uses the expression $\text{dot}$. For example, the substitution of the first $n$ variables of a given function $f$ by the values $1, 2, \ldots, n$ can be expressed in the following way:

\[
 f \left(\text{dot}\ i=1 \text{ton api}\right)
\]
Moreover, the substitution of the first 3n variables of a given function $f$ by the values 1, 2, 3, ..., 1, 2, 3 can be expressed in the following way:

$$f \{\text{dot } i := 1 \text{ to } n \text{ ap } 1, 2, 3\}$$

This allows to capture the sense of the mathematical notation $e^{[X_1/v_1][X_2/v_2] \cdots [X_n/v_n]}$, where $e$ denotes an expression with free variables $X_1, \ldots, X_n$ which are replaced by the admissible terms $v_1, \ldots, v_n$.

### 2.3.5. A remark on assignments

In MILONGA, assignments may only be used to copy values, there is no possibility of aliasing. This means that, e.g. the sequence of commands:

$$\{v := w; v[1] := 53\}$$

does not affect the value of $w[1]$.

### 2.3.6. Vectors and matrices defined by means of a generating function

A vector or a matrix may be given by means of a generating function. The expression `newV n f`, where $n$ is a positive integer and $f$ is a partially evaluated function with exactly one unspecified extra parameter, denotes a vector of length $n$ such that for $1 \leq i \leq n$ the expression $f_i$ represents the $i$th entry of the vector.

In a similar way, the expression `newM n1 n2 f`, where $n1$ and $n2$ are positive integers and $f$ is a partially evaluated function with exactly two unevaluated extra parameters, denotes a matrix of size $n1 \times n2$ such that for $1 \leq i \leq n1, 1 \leq j \leq n2$ the expression $f_{ij}$ represents the entry $(i, j)$ of the matrix.

For example, it is possible to specify a Vandermonde matrix (of any size) by means of the following fragment:

```milonga
vanderF extras x(2) result n:TNum
{n := pot x[0] x[1]}
vandemonde (n:TInt) result m:TMat
{m := newM n n vanderF}
```

If, during run-time, the required arity of the generating function is not met, MILONGA announces run-time error.

The representation of vectors and matrices by generating functions allows to save memory space. However, in this way, the property of constant-time access to each entry of the given vector or matrix is lost. The access time depends on the evaluation of the generating function. Therefore, it is only meaningful to make use of this feature in presence of some kind of "uniformity", i.e. when the generating function allows a fast, direct access to any entry of the given vector or matrix. The decision about the use of a generating function or an explicit array representation must be taken separately in each particular case and depends on a time–space tradeoff.
2.3.7. Examples

In this section we present a couple of toy examples in order to illustrate the programming with MILONGA. In Section 5, we shall exhibit more realistic applications of MILONGA to multivariate polynomial equation solving and related problems.

- Let $v_1 := (v_1[1], \ldots, v_1[n])$ and $v_2 := (v_2[1], \ldots, v_2[n])$ be two vectors. The function $\text{prodEsc}$ computes the inner product $v_1 \cdot v_2 := v_1[1]v_2[1] + \cdots + v_1[n]v_2[n]$ of $v_1$ and $v_2$.

$\text{prodEsc} (v_1, v_2: \text{TVec}) \text{ result } r: \text{TObj} \text{ local } (i: \text{TInt})$

$\begin{align*}
\{ \\
r &:= 0; \\
\text{for } i := 0 \text{ to } (# vv 1 - 1) \text{ do } r := r + v_1[i] \ast v_2[i] \\
\}
\end{align*}$

- The function $\text{zeroN}$ takes a function $f$ and an integer $n$ as input and returns $f$ with its first $n$ extra variables replaced by 0 (if $f$ has less than $n$ extra variables, then all of them are replaced by 0).

$\text{zeroN} (f: \text{TSlp TObj}, n: \text{TInt}) \text{ result } r: \text{TSlp TObj} \text{ local } (i, e: \text{TInt})$

$\begin{align*}
\{ \\
e &:= #e f; \\
\text{if } (n <= e) \text{ then } r := f (\text{dot } i := 0 \text{ to } (n - 1) \text{ ap } \emptyset) \\
\text{else } r := f (\text{dot } i := 0 \text{ to } e - 1 \text{ ap } \emptyset) \\
\}
\end{align*}$

- Using Horner’s rule, the function $\text{horner}$ evaluates a polynomial $P := v[n - 1]X^{n-1} + \cdots + v[0]$, represented by a vector $v := (v[n - 1], \ldots, v[0])$, in either a number or a matrix $a$.

$\text{horner} (v: \text{TVect}, a: \text{TObj}) \text{ result } r: \text{TObj} \text{ local } (i, f, g, n, h: \text{TInt})$

$\begin{align*}
\{ \\
n &:= #v v; \\
h &:= n - 1; \\
r &:= v[h]; \\
f &:= 0; \\
g &:= 1; \\
i &:= n - 2; \\
\text{while } i >= f \text{ do } \\
\{ \\
r &:= r \ast a + v[i]; \\
i &:= i - g \\
\}
\end{align*}$

- The function $\text{main}$ computes the characteristic polynomial of the $(3 \times 3)$-identity matrix $m$, whose entries are defined by the generating function id, i.e. the entry $m[i, j]$ is defined as $m[i, j] := \text{id}(i, j)$, where $\text{id}(i, j) = 1$ if $i = j$ and $\text{id}(i, j) = 0$ otherwise. The function $\text{main}$ depends
on a function \texttt{samuelson} for the computation of the characteristic polynomial of a given matrix. The function \texttt{samuelson} will be described in Section 5.1.

```plaintext
module main
{
    id extras x(2) result r:TNum
    {
        if (x[0] == x[1]) then r := 1
        else r := 0
    }
    main result r:TObj
    local (m:TMat)
    {
        m := newM 3 3 id;
        r := samuelson m
    }
}
```

3. The compiler

In this section we present an overview of the different tasks performed by the \texttt{MILONGA} compiler. The input of the compiler is a \texttt{MILONGA} script, and its output is a program written in object language, equivalent to the original script from the semantic point of view.

Let us be given a \texttt{MILONGA} script. Then the compiler performs the following tasks:

- **Lexing and parsing:** An abstract syntax tree, called \texttt{slpAbs}, is generated and a couple of semantic verifications (for example type-checking) are performed on this tree.
- **Differentiation and optimization of the abstract syntax tree:** If necessary automatic differentiation is performed on the abstract \texttt{slpAbs} syntax tree generated by the parsing procedure and the resulting code, which is again an abstract \texttt{slpAbs} syntax tree, is optimized. The standard optimization rules for source code are applied [2].
- **Generation of intermediate code:** An intermediate representation of the given \texttt{MILONGA} script, called \texttt{pCode}, is generated, and a second optimization is performed.
- **Generation of object code:** The intermediate \texttt{pCode} representation is translated to a semantically equivalent \texttt{C++} object code.

In order to produce a computer architecture independent tool which allows to run the intermediate \texttt{pCode} representation of a given \texttt{MILONGA} script, we develop in this section an abstract state transformation machine. The design of this abstract \texttt{MILONGA} machine is inspired in the \texttt{G-machine} of Johnson [27]. The working mechanism of the abstract \texttt{MILONGA} machine is translated into a \texttt{C++} library. This allows to simulate the machine in any real world computer which is able to compile \texttt{C++} code. In order to run a given \texttt{MILONGA} program on such a computer, the object code of this program is linked to the \texttt{C++} library which simulates the abstract \texttt{MILONGA} machine. The result is a highly optimized \texttt{C++} code which allows to run the given \texttt{MILONGA} program on any real world computer able to
compile C++ code. It is possible to merge native C++ code with MILONGA scripts (see Sections 1 and 2.2.1).

The MILONGA compiler is implemented in Haskell [38], a pure nonstrict functional language. Functional programming languages are efficient tools for rapid prototyping. This is due to the fact that functional programming languages are well adapted to the description of the fundamental features of the functions to be programmed, since they focus on the “whats” while ignoring the “hows” (see [4,43]). Moreover, when based on a monadic style [45,46], functional languages are particularly well suited for compiler construction [26]. In fact, using these tools it took only one man-month to develop the whole MILONGA compiler from scratch. However, the performance of functional programming languages is very slow compared to that of imperative languages. Nevertheless, since we were not interested in the efficiency of the compiler itself, but in the efficiency of the resulting object code, we felt legitimated to choose a functional language for the compiler construction and an imperative language, namely C++, as object language.

3.1. Lexing and parsing in MILONGA

In this section we suppose a basic knowledge of the grammar of MILONGA. This grammar can be found in [5]. Moreover, we refrain from describing all details of the lexer and parser of the MILONGA compiler (except for the type-checking algorithm), since these details are more or less routine and can be found in [5].

3.1.1. The abstract syntax tree

We briefly describe the abstract syntax tree of a given MILONGA script in Section 3.1.1. The abstract syntax tree of a given MILONGA script shall be denoted by the following grammar (see [4,43]):

- \([t] \) represents the lists of elements of type \( t \).
- \((t_1, t_2, \ldots, t_n)\) represents an \( n \)-tuple of types \( t_1, \ldots, t_n \).
- \( C t_1 \cdots t_n \) denotes the type constructor \( C \) applied to types \( t_1, \ldots, t_n \). It can be thought as a \( n \)-tuple with an identifier \( C \).
- \( t_1 | t_2 \) stands for the disjoint union of types \( t_1 \) and \( t_2 \).
- The types \( \text{Int} \) and \( \text{Name} \) are basic types and they represent the integer numbers and character strings, respectively.

The following data types describe the abstract syntax tree of a MILONGA program:

```plaintext
Program = [Module]  
Module = [Name, [Name], [Func]]  
Func = UserDef Name [VarType] Name, Expr [VarType] Code  
| Prim Name [VarType] Type  
Code = [Instr]  
Instr = IAssign IVar Expr  
| IIF Expr Code Code  
| IWhile Expr Code  
| IFor Name Bool Expr Expr Expr Code  
| ICRN Expr  
| ICRV Expr
```
3.1.2. Type-checking

We will use (partial) functions in order to specify the type system. These functions define the type of a given construction in a bottom-up way (i.e. using the types of the components of the construction). Moreover, these functions can be seen as a syntax directed definition that uses synthesized attributes. MILONGA is not a strongly typed language and type errors may occur during run-time.

Let Types be the set of all possible types in MILONGA. The following two auxiliary functions on Types specify the type system in MILONGA:

1) The relation mayBe.

The function mayBe represents a binary relation on types: an expression “a mayBe b” has to be considered as “(a, b) \in mayBe”. Its meaning is the following: “may type a be accepted when type b is expected”. We list now typical situations in which the relation mayBe occurs:

- TNum mayBe TVec, TMat. This means that a number may be used when a vector or a matrix is expected. For example, the expression $3 + m$, where m is a matrix with $n$ rows and $n$ columns, is considered as $3 * I_{n \times n} + m$ with $I_{n \times n}$ denoting the $n \times n$ identity matrix.
- TNum, TVec, TMat mayBe TObj. The meaning is obvious by the definition of the type TObj.
- TObj mayBe TNum, TVec, TMat. Here the meaning is again obvious. However, a dynamic type error can occur during run-time if the actual type of the parameter does not match the expected type.
- TSlp a mayBe TSlp b, if a mayBe b. The meaning is again obvious.
- TSlp a mayBe TSlp b, if a mayBe b. The meaning is obvious. However, a dynamic type error can occur during run-time if the element of type TSlp a still contains extra variables.

2) To each binary operation binop(x, y) we associate a (partial) function, denoted by $\text{cast}_{\text{binop}}$, which maps the types a and b of the two arguments x and y of binop into a third type, namely $\text{cast}_{\text{binop}}(a, b)$. For each binary operation, the corresponding function returns the type of the result of $\text{binop}(x, y)$ when the types a and b of the arguments x and y are given.
We are going to consider the following binary operations \( \text{binop} \): addition and subtraction \((+,-)\), multiplication \((\ast)\), division \((/)\) and suitable relational \((\text{relop})\) and boolean \((\text{boolop})\) operations. The corresponding functions \(\text{cast}_{\text{binop}}\) are defined as follows:

<table>
<thead>
<tr>
<th>Type ( a )</th>
<th>Type ( b )</th>
<th>( \text{cast}_{\ast}(a,b) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{TSlp} u )</td>
<td>( v )</td>
<td>( \text{cast}_{\ast}(u,v) )</td>
</tr>
<tr>
<td>( u )</td>
<td>( \text{TSlp} v )</td>
<td>( \text{cast}_{\ast}(u,v) )</td>
</tr>
<tr>
<td>( \text{TInt} )</td>
<td>( \text{TInt} )</td>
<td>( \text{TInt} )</td>
</tr>
<tr>
<td>( \text{TObj} )</td>
<td>( \text{TNum}, \text{TVec}, \text{TMat} )</td>
<td>( b )</td>
</tr>
<tr>
<td>( \text{TNum}, \text{TVec}, \text{TMat} )</td>
<td>( \text{TObj} )</td>
<td>( a )</td>
</tr>
<tr>
<td>( \text{TNum} )</td>
<td>( \text{TNum}, \text{TVec}, \text{TMat} )</td>
<td>( b )</td>
</tr>
<tr>
<td>( \text{TNum}, \text{TVec}, \text{TMat} )</td>
<td>( \text{TNum} )</td>
<td>( a )</td>
</tr>
<tr>
<td>( \text{TVec} )</td>
<td>( \text{TVec} )</td>
<td>( \text{TVec} )</td>
</tr>
<tr>
<td>( \text{TMat} )</td>
<td>( \text{TMat} )</td>
<td>( \text{TMat} )</td>
</tr>
<tr>
<td>( \text{TObj} )</td>
<td>( \text{TObj} )</td>
<td>( \text{TObj} )</td>
</tr>
</tbody>
</table>

If \( a = \text{TVec} \) and \( b = \text{TMat} \) then \( \text{cast}_{\ast}(a,b) = \text{TVec} \)
If \( a = \text{TMat} \) and \( b = \text{TVec} \) then \( \text{cast}_{\ast}(a,b) = \text{TVec} \)
If \( a \notin \{\text{TVec}, \text{TMat}\} \) or \( b \notin \{\text{TVec}, \text{TMat}\} \) then \( \text{cast}_{\ast}(a,b) = \text{cast}_{\ast,-}(a,b) \)
If \( a \in \{\text{TInt}, \text{TNum}, \text{TSlp} \text{TInt}, \text{TSlp} \text{TNum}\} \) then \( \text{cast}_{\ast}(a,b) = \text{cast}_{\ast,-}(a,b) \)
If \( a = b \land a \in \{\text{TInt}, \text{TNum}, \text{TSlp} \text{TInt}, \text{TSlp} \text{TNum}\} \) then \( \text{cast}_{\text{relop}}(a,b) = \text{TBool} \)
If \( a = \text{TBool} \) and \( b = \text{TBool} \) then \( \text{cast}_{\text{boolop}}(a,b) = \text{TBool} \)

Table 1 specifies the function \(\text{type}\) for any expression in the MILONGA language. In the first column the expressions are enumerated, the second column indicates the corresponding type and the third column contains the preconditions that have to be satisfied for the correct definition of the function \(\text{type}\). If these preconditions are not satisfied, the value of \(\text{type}\) remains undefined.

3.1.3. Type-checking instructions

Table 2 defines the function \(\text{type}\) on the variables. The columns in Table 2 have the same meaning as in Table 1.

Table 3 lists the basic instructions for the type-checker and its preconditions. A sequence of instructions is applied correctly if and only if each of the instructions is applied correctly.

3.2. Differentiation and code optimization

The compilation of a given a MILONGA script starts with a procedure of lexing and parsing, whose output is an abstract \text{sPlAbs} syntax tree (see Section 3.1). To such an abstract \text{sPlAbs} syntax tree, automatic differentiation may be applied. The result is again an abstract \text{sPlAbs} syntax tree. The abstract \text{sPlAbs} syntax tree produced by the parser or by automatic differentiation is finally optimized. In Sections 3.2.1 and 3.2.2 below we describe how differentiation and optimization is applied to abstract \text{sPlAbs} syntax trees.
Table 1
The type of an expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENum n</td>
<td>TInt</td>
<td>–</td>
</tr>
<tr>
<td>True</td>
<td>TBool</td>
<td>–</td>
</tr>
<tr>
<td>False</td>
<td>TBool</td>
<td>–</td>
</tr>
<tr>
<td>EVar v</td>
<td>TInt</td>
<td>v is declared in the function</td>
</tr>
<tr>
<td>EVProj e_0 e_1</td>
<td>TObj</td>
<td>type(e_0) mayBe TVec, and type(e_1) mayBe TInt</td>
</tr>
<tr>
<td>EMProj e_0 e_1 e_2 e_3</td>
<td>TObj</td>
<td>type(e_1) mayBe TMat, type(e_2) mayBe TInt, and type(e_3) mayBe TInt</td>
</tr>
<tr>
<td>EDot e_0 e_1 e_2 e_3</td>
<td>TObj</td>
<td>type(e_0) = TSlp x, type(e_1) mayBe TInt, type(e_2) mayBe TInt, type(e_3) mayBe TInt, and the types of e_0, e_1, and e_2, are defined</td>
</tr>
<tr>
<td>EUnOp $ e</td>
<td>TNum</td>
<td>type(e) mayBe TInt</td>
</tr>
<tr>
<td>EUnOp @ e</td>
<td>TInt</td>
<td>type(e) mayBe TNum</td>
</tr>
<tr>
<td>EUnOp ! e</td>
<td>TBool</td>
<td>type(e) mayBe TBool</td>
</tr>
<tr>
<td>EUnOp ∼ e</td>
<td>TInt</td>
<td>type(e) mayBe TInt, or type(e) mayBe TNum</td>
</tr>
<tr>
<td>EUnOp # v e</td>
<td>TInt</td>
<td>type(e) mayBe TMat</td>
</tr>
<tr>
<td>EUnOp # f e</td>
<td>TMat</td>
<td>type(e) mayBe TMat, and type(f) mayBe TObj, and type(e) mayBe TInt</td>
</tr>
<tr>
<td>EUnOp # e e</td>
<td>TInt</td>
<td>type(e) mayBe TMat, and type(e) mayBe TMat</td>
</tr>
<tr>
<td>EUnOp # e e</td>
<td>TInt</td>
<td>type(e) mayBe TMat</td>
</tr>
<tr>
<td>EUnOp # e e</td>
<td>TInt</td>
<td>type(e) mayBe TMat</td>
</tr>
<tr>
<td>EBinop b e_1 e_2</td>
<td>TInt</td>
<td>cast(b) (e_1, e_2) cast(b) is defined</td>
</tr>
<tr>
<td>EPrime v</td>
<td>TInt</td>
<td>type(v) mayBe TNum</td>
</tr>
<tr>
<td>EAStore v e_1 e_2</td>
<td>TInt</td>
<td>type(v) mayBe TMat, and type(e_1) mayBe TInt, and type(e_2) mayBe TInt</td>
</tr>
<tr>
<td>EWhile v e_1 e_2 e_3</td>
<td>TInt</td>
<td>type(e_1) mayBe TInt, type(e_2) mayBe TInt, and type(e_3) mayBe TInt, and c is correct</td>
</tr>
<tr>
<td>IF v e_1 e_2</td>
<td>TInt</td>
<td>type(v) = TNum</td>
</tr>
<tr>
<td>ICR v e</td>
<td>TVec</td>
<td>type(e) mayBe TVec</td>
</tr>
</tbody>
</table>

Table 2
Types of assigned variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single v</td>
<td>TInt</td>
<td>–</td>
</tr>
<tr>
<td>VProj v e</td>
<td>TObj</td>
<td>type(v) mayBe TVec, and type(e) mayBe TInt</td>
</tr>
<tr>
<td>MProj v e_1 e_2</td>
<td>TObj</td>
<td>type(v) mayBe TMat, type(e_1) mayBe TInt, and type(e_2) mayBe TInt</td>
</tr>
</tbody>
</table>

Table 3
Instruction typing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAssign v e</td>
<td>type(v) mayBe type(e)</td>
</tr>
<tr>
<td>If v e_1 e_2</td>
<td>type(v) mayBe TBool, e_1 is correct, and e_2 is correct</td>
</tr>
<tr>
<td>IWhile v e_1 e_2</td>
<td>type(v) mayBe TBool, and c is correct</td>
</tr>
<tr>
<td>IFor v e_1 e_2 e_3</td>
<td>type(v) mayBe TInt, type(e_1) mayBe TInt, type(e_2) mayBe TInt, and type(e_3) mayBe TInt, and c is correct</td>
</tr>
<tr>
<td>ICR v e</td>
<td>type(v) mayBe TNum</td>
</tr>
<tr>
<td>ICR v e</td>
<td>type(v) mayBe TVec</td>
</tr>
</tbody>
</table>
3.2.1. The differentiation algorithm

In this section we describe the differentiation rules for an arbitrary MILonga language construct:

(1) Expressions: Table 4 contains the differentiation rules for expressions in the MILonga language. They correspond basically to the standard differentiation rules for algebraic expressions.

(2) Instructions: Table 5 contains the corresponding differentiation rules. Let us underline the following points:

(a) Deriving an assignment variable \( v[p_1][p_2] \cdots [p_n] \) one obtains \( v'[p_1][p_2] \cdots [p_n] \).

(b) When deriving an instruction, a new sequence of instructions is obtained.

(c) When deriving a sequence of instructions, the sequence of derivatives of each subinstruction is produced.

Only assignments have to be derived. Instructions different from assignments “mimic” the behaviour of the original function. This implies that any function and its derivative have the same control graph. In particular, neither branchings nor loops are derived.

(3) Functions: The derivative of a function is a new function. Its name is the primed (’ ) version of the original name (i.e. if the name of the original function is \( N \) then the name of its derivative is \( N' \)).
Table 6
Components of a function and its derivative

<table>
<thead>
<tr>
<th>Component</th>
<th>Original function</th>
<th>Derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>N</td>
<td>N'</td>
</tr>
<tr>
<td>Fixed parameters</td>
<td>$f_1, \ldots, f_n$</td>
<td>$f_1, f_1', \ldots, f_n, f_n'$</td>
</tr>
<tr>
<td>Extra variables</td>
<td>$v(e)$</td>
<td>$v(2e)$</td>
</tr>
<tr>
<td>Result</td>
<td>res</td>
<td>res'</td>
</tr>
<tr>
<td>Local variables</td>
<td>$l_1, \ldots, l_n$</td>
<td>res, $l_1', \ldots, l_n, v'$</td>
</tr>
<tr>
<td>Code</td>
<td>C</td>
<td>C'</td>
</tr>
</tbody>
</table>

A given code is derived according to the corresponding rules. Table 6 describes the relationship between the components of the original function and its derivative.

3.2.2. Straight-line program code optimization

MILONGA applies the following common steps to source code [2]:

- **Translation of instructions for**: An instruction $for$ is translated into the corresponding instruction $while$.
- **Algebraic simplifications**: Distinct algebraic simplifications are performed: expressions in which every operand is a constant are replaced by their results, addition and subtraction of 0 and multiplication and division by 1 are simplified.
- **Elimination of unnecessary code**: Code that does not contribute to the current computation of the result of the function is eliminated.
- **Elimination of unused variables**: Unused local variables are eliminated from the local declaration.

3.2.3. An example

Here is an example which illustrates how the source code is transformed after performing automatic differentiation and code optimization.

Let $poly$ be the function $poly(x, y, n) := n(2n + 7)$, with fixed parameters $n, x$ and $y$ (observe that the corresponding mathematical function does not depend on $x$ and $y$). Suppose the following definition of the function $poly$ appears in the source code:

```plaintext
define $poly (x, y):TNum, n:TInt$ result $r:TNum$ local $(i, j, k:TInt)\newline
\{\newline
  $r := 0; i := 1; k := 1;\newline
  \text{while } (i < 2 * n + 1) \text{ do}\newline
  \{\newline
    r := r + i + 3; \newline
    i := i + 1\newline
  \}\newline
\}\newline
```

Suppose furthermore that somewhere else in the MILONGA program the expression $(\text{derive } poly 1)$ appears. Observe that the expression $(\text{derive } poly 1)$ represents the function $\frac{\partial poly}{\partial x} = 0.$
Then, after the differentiation and optimization step, the following code will be added to the (optimized version of the) original code:

\[
\text{poly}'(x,x',y,y': \text{TNum}, n,n': \text{TInt}) \text{ result } r': \text{TNum} \text{ local }
\]

\[
(i,i',w_1: \text{TInt}, w_2: \text{TNum})
\]

\[
\begin{align*}
  r' & := 0; \\
  i & := 1; \\
  i' & := 0; \\
  w_1 & := 2 \times n + 1; \\
  w_2 & := i'; \\
  \text{while } i < w_1 \text{ do }
    \begin{align*}
      r' & := r' + w_2; \\
      i & := i + 1
    \end{align*}
\end{align*}
\]

3.3. Intermediate code generation

As announced at the beginning of Section 3, the MILONGA compiler transforms a given MILONGA script into an intermediate representation called pCode. This intermediate pCode representation is then optimized and the result is translated into C++ code. Linking this translation to the before mentioned C++ library which simulates the abstract MILONGA machine (see the beginning of Sections 3 and 4), we finally obtain a semantically equivalent representation of the original MILONGA script in the language C++.

The compilation rules which generate this intermediate pCode representation depend on two ingredients: the already generated pCode and a so-called environment component (see Sections 3.3.1 and 3.3.2).

3.3.1. The MILONGA environment component

The component environment is used in order to tabulate properties of variables occurring in MILONGA functions. The environment \( \rho \) of a function \( f \) is a list of four-tuples of the form \( (v, t, p, l) \). The meaning of each entry of such a four-tuple is the following:

- \( v \) is an identifier. The identifier determines the four-tuple where the identifier occurs.
- \( t \) is the type of the identifier \( v \).
- \( p \) is the dynamic depth of \( v \) in the corresponding stack (see Sections 3.4.1 and 3.5 for the definition of stacks).
- \( l \) determines whether the variable \( v \) is local or fixed.

In order to denote the properties of \( \rho \), we use the following notation:

- \( \rho[v].type \) denotes the type of variable \( v \) in the environment \( \rho \).
- \( \rho[v].depth \) denotes the depth of variable \( v \) in the environment \( \rho \).
- \( \rho.ls \) counts the variables of \( f \) that are local and whose types are neither TInt nor TBool.
\( \rho.\text{lis} \) counts the variables of \( f \) that are local and whose types are either \( \text{TInt} \) or \( \text{TBool} \).

\( \rho.\text{fs} \) counts the variables of \( f \) that are not local and whose types are neither \( \text{TInt} \) nor \( \text{TBool} \).

\( \rho.\text{fis} \) counts the variables of \( f \) that are not local and whose types are either \( \text{TInt} \) or \( \text{TBool} \).

\( \rho.\text{res} \) denotes the identifier that has been declared as the result of the current function \( f \).

The following functions are used to modify the environment:

- \( \rho.\text{+Sn} \) increases by \( n \) the depth of variables whose types are neither \( \text{TBool} \) nor \( \text{TInt} \).
- \( \rho.\text{+In} \) increases by \( n \) the depth of variables whose types are either \( \text{TBool} \) or \( \text{TInt} \).

### 3.3.2. Compilation rules

The transformation of a MILONGA script into an intermediate pCode representation is performed by means of compilation rules. These are rewriting rules which generate an intermediate pCode representation from a given MILONGA program construct and a given environment.

The compiled version of any function is determined by a four-tuple \( \langle n, tF, cE, cB \rangle \), where \( n \) denotes the name of the function, \( tF \) the vector which describes the types of the fixed variables of the function, \( cE \) the compiled code that calculates the number of extra parameters of the function and \( cB \) is the compiled code for the function body. If the function is primitive, a special compiled version is generated, containing the name of the function, a vector which describes the types of the fixed variables and the type of the result of the function.

The compilation rules for copying an expression avoid aliasing. Whenever two identifiers refer to the same expression, a “copy” of the original expression is placed on top of stack \( S \) (see Sections 3.4.1 and 3.5 for the definition of the stack \( S \)). Each MILONGA script instruction is translated into a constant number of pCode instructions and the basic syntactical structure is maintained. This will allow us to conclude that the execution time of the \texttt{C++} compilation of a given MILONGA script encoding a theoretical elimination algorithm is of the same order as the theoretical complexity of this algorithm (see Section 4.3).

The complete list of compilation rules can be found in [5].

### 3.3.3. pCode optimization

MILONGA applies a series of syntactical optimization procedures to the intermediate pCode representation of a MILONGA script. They are of the type listed below (see Section 4 for the semantics of the mentioned instructions):

- **Elimination of redundant instructions**: Instructions such as Pop 0, IPop 0, Alloc 0 and IAlloc 0 are eliminated from the intermediate code.
- **Elimination of pairs of instructions with null net effect**: Whenever a pair of instructions with null net effect—such as \([\text{toI}, \text{toS}]\) or \([\text{toI}, \text{toS}]\)—is found, the pair is eliminated leaving unchanged the original semantics.
- **Simplification of control structures**: The following simplifications involving jumps and labels are possible:
  - The pair \([\text{Label} \ i, \ \text{Label} \ j]\) is replaced by \( \text{Label} \ j \) and all jumps to \( i \) are replaced by jumps to \( j \).
  - The pair \([\text{Label} \ i, \ \text{Jump} \ j]\) is replaced by \( \text{Jump} \ j \) and all jumps to \( i \) are replaced by jumps to \( j \).
  - Unreachable instructions are eliminated.
  - Unreferenced labels are eliminated.
3.4. The abstract machine

It is harder to implement a language of the characteristics of MILONGA than a conventional one. The implementation has to produce expressions whose evaluation may be postponed until their value is really needed. Conventional imperative languages are abstractions of the machines they run on, and so they are fairly easy to be implemented in an efficient way.

An abstract machine is a performant tool when there is no direct way to map the constructs of the given programming language into the target machine. It is also a performant tool in situations of unspecified target architecture. In this case the abstract machine represents an intermediate stage between source and target.

In this section we will describe the basic ingredients of the abstract MILONGA machine. A description of its operational semantics can be found in Section 4.2.

3.4.1. The components of the abstract MILONGA machine

The abstract MILONGA machine operates on eight-tuples of the form \( \langle C, IP, S, D, H, IS, G, R \rangle \) which represent the (initial, intermediate and final) states of the machine. To the pCode representation of a given MILONGA script there corresponds a well-defined initial state of the abstract MILONGA machine. From this initial state as input, the machine evolves, changing at each step its momentaneous state to another one, until a final state is reached or the computation stops by lack of suitable commands. The working mechanism of the abstract MILONGA machine is determined by a list of (nondeterministic) transition rules whose application becomes deterministic by the use of so-called instructions (see Section 4 for details). The entries of an eight-tuple \( \langle C, IP, S, D, H, IS, G, R \rangle \) representing a given state of the abstract MILONGA machine, have the following meaning:

- \( C \) (code) is a numbered instruction sequence of type pCode.
- \( IP \) (instruction pointer) is a number that points to the next instruction.
- \( S \) (stack) is an address stack (whose entries point to the MILONGA heap component introduced in the next section).
- \( D \) (dump) is a stack of pairs of type \( (C, IP) \) used in procedure-activation and procedure-return.
- \( H \) (heap) is an address-node mapping.
- \( IS \) (integer stack) is a basic value stack (used to store integer numbers).
- \( G \) (globals) is a function that maps function names to heap addresses.
- \( R \) (refs) is a pair of addresses which contains the information needed to perform modular arithmetic operations with numbers and vectors.

3.4.2. The MILONGA heap component

In this section we describe the component heap of the abstract MILONGA machine. This component assembles nodes and Table 7 lists the different classes of nodes that may occur in the heap. Table 7 makes use of the following new types:

- \( T\text{Integer} \) denoting a bounded integer.
- \( T\text{Addr} \) denoting an address.
- \( T\text{Number} \) denoting an infinite precision integer.
Table 7
Heap nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NInt</td>
<td>Integers</td>
<td>i</td>
<td>TInteger</td>
</tr>
<tr>
<td>NNum</td>
<td>Numbers</td>
<td>n</td>
<td>TNumber</td>
</tr>
<tr>
<td>NVec</td>
<td>Address vector</td>
<td>dV</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v</td>
<td>TAddr[1,...,dV]</td>
</tr>
<tr>
<td>NFVec</td>
<td>Generated vector</td>
<td>dV</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>TAddr</td>
</tr>
<tr>
<td>NMat</td>
<td>Address matrix</td>
<td>dF</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dC</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m</td>
<td>TAddr[1,...,dF,1,...,dC]</td>
</tr>
<tr>
<td>NFMat</td>
<td>Generated matrix</td>
<td>dF</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dC</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>TAddr</td>
</tr>
<tr>
<td>NFun</td>
<td>Function</td>
<td>totF</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t</td>
<td>TType[1,...,totF]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eC</td>
<td>TCode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bC</td>
<td>TCode</td>
</tr>
<tr>
<td>NBox</td>
<td>Partially evaluated function (the number of extras has not been computed yet)</td>
<td>fun</td>
<td>TAddr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>numF</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xF</td>
<td>TAddr[1,...,numF]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derive</td>
<td>TInteger</td>
</tr>
<tr>
<td>NSlp</td>
<td>Partially evaluated function (the number of extras has already been computed)</td>
<td>box</td>
<td>TAddr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>totE</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>numE</td>
<td>TInteger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xE</td>
<td>TAddr[1,...,totE]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derive</td>
<td>TInteger</td>
</tr>
<tr>
<td>NForward</td>
<td>Indirection (GC)</td>
<td>a</td>
<td>TAddr</td>
</tr>
</tbody>
</table>

TCode denoting the abstract code the machine generates.
T[1,...,n] denoting an array of elements of type T.
T[1,...,m,1,...,n] denoting a bidimensional array of elements of type T.
Type denoting the enumerated type whose components are TBool, TInt, TNum, TVec, TMat, TObj, TSlp.
The components of most of the nodes of the heap do not need explanation. However, some of them do so.
In nodes NFVec and NFMat, the component f denotes the address of the generating function.
In node NFun, the component totF denotes the number of fixed parameters, t the vector that describes the types of the fixed parameters, eC the code that computes the number of extra parameters and bC denotes the code of the function body.
In node $N_{Box}$, the component $fun$ denotes the address of the corresponding node $N_{Fun}$, $numF$ expresses how many fixed parameters have been applied, $vF$ the vector that contains the addresses of the fixed parameters applied so far, and $derive$ denotes the index of the partial derivative that has to be considered (if the component $derive$ takes the value zero, the node does not represent a derivative).

In node $N_{Slp}$, the component $box$ denotes the address of the corresponding node $N_{Box}$, the component $totE$ expresses the expected total number of extra parameters, $numE$ expresses how many extra parameters have already been applied, $vE$ the vector that contains the addresses of the extra parameters applied so far and $derive$ denotes the index of the partial derivative that has to be considered.

Finally, the node $N_{Forward}$ is used during garbage collection (see Section 3.6).

3.5. Generation of object code

We have now established the basic structure of the abstract MILONGA machine. In order to be able to translate its working mechanism into C++ object code, we need two ingredients:

- The representation of each state of the abstract machine on a concrete computer.
- A translation mechanism to the object language that preserves the semantics.

The eight components in the definition of the abstract MILONGA machine (see Section 3.4.1) are represented in the object language in the following way:

- The component $C$ is represented by means of the object code of the function that is currently executed and the component $IP$ points to an instruction in a concrete computer.
- $S$ is represented by means of a storage area and a stack pointer.
- $IS$ is represented by objects that are pushed on top of the system stack.
- $H$ is represented by means of a storage area divided into two halves, and a heap pointer indicates the next available address.
- $D$ is represented by objects that are pushed on top of the stack. In fact, only pointers to pieces of code are pushed into the stack.
- $G$ is represented by the object code of functions.
- $R$ is represented by a pair of variables.

We use “macro-expansion” as translation mechanism. Each abstract instruction is translated into a fixed sequence of object instructions independent of context. The impact of this translation mechanism on the execution time of a given MILONGA procedure is discussed in Section 4.3.

3.6. Manipulation of the MILONGA heap component

The heap $H$ is a big memory space divided into two halves. Only one of these halves is used at any moment. In this way, memory is used “from left to right”, i.e. from the lower addresses up to the higher ones. When the actual half becomes exhausted, active nodes are copied to the other half using the following garbage collection technique.

Suppose that we are running a MILONGA program. While the program is running, nodes (which may become inactive with time) are successively added to the active half of the heap component until this half becomes exhausted. At this moment the system triggers a garbage collector which frees the heap from inactive nodes. This garbage collector is based on a variable size cell version of the Fenichel–Yochelson
algorithm [28] and works in two stages as follows:

(1) Let us suppose that the first half of the heap is active when the garbage collector starts working. Then any active node (to which some component of the abstract MILONGA machine points, as the stack S, the globals G or the references R) is copied by the garbage collector, together with its content, to the second half of the heap (supposed to be empty at this moment). The original node is then overwritten by a NForward node pointing to its copy in the second half of the heap. Inactive nodes are not copied.

(2) The content of all nodes of the second half of the heap are now scanned one by one and all pointers to uncopied (inactive) nodes are eliminated from the content and the remaining pointers are actualized with respect to the new nodes in the second half of the heap. Then the first half of the heap is cleaned from nodes and the second half is declared as active.

This memory management mechanism, namely the copy family of garbage collectors, has the following advantages [27]:

• There is no need for a special treatment of nodes of different size. This makes operating with big arrays easy.
• The garbage collector performs a compactification of the heap, such that “contemporary” nodes become to “live in the same neighborhood”. This proximity (proximity in time is reflected by proximity in space) is an advantage when virtual memory is used.
• The lifetime of a copy is proportional to the size of the current heap, i.e. proportional to the size of the active nodes of the heap. Thus, for small problems, the garbage-collecting time becomes short although the size of the heap may be very big.

The main disadvantage of this garbage collector is that only half of the memory is available at any time. Nevertheless, this is not a real drawback if virtual memory is used, because of the possibility that pages corresponding to the unused areas of the heap can migrate to secondary storage.

Suppose that there is given a theoretical elimination algorithm implemented as a MILONGA procedure S. Then the size of the heap required to run the procedure S on a real world computer or on the abstract MILONGA machine is of the same order as the space complexity (expressed in terms of a pebble game) of the given theoretical algorithm (see Section 4.3). In particular, for such an algorithm with well-analyzed space complexity the necessary heap size may be determined a priori, before running the corresponding MILONGA procedure. Minimization of the predetermined heap size leads to frequent garbage collection and therefore to time loss. A rough profiling of the corresponding time–space tradeoff is possible by measuring the frequency of garbage collection.

4. Semantics of the abstract machine

In this section we explain the internal working mechanism of the abstract MILONGA machine. In Section 3.4 we have already established the basic structure of this machine. In order to specify its run-time support, we introduce first in Section 4.1 the notion of initial and final state of the machine. Then we are going to explain the mechanism which allows the machine to move from one state to another one until a final state is reached or the computation stops by lack of commands. The abstract MILONGA machine evolves applying certain (non-deterministic) state transition rules. The application of the state transition rules to the working mechanism of the abstract MILONGA machine becomes deterministic by the use of so-called (working) instructions. These instructions are explained in Section 4.2.
4.1. Initial and final states

Taking into account its components (see Section 3.4.1), a particular state of the abstract MILONGA machine is denoted as follows:

\[
\begin{array}{cccccccc}
C & IP & S & D & H & IS & G & R \\
\end{array}
\]

The component \( C \) of this machine state representation contains the ingredients which determine the evolution of the machine to the next state. These ingredients consist of a numbered sequence of instructions (see Section 4.2 below).

For the sake of simplicity, we shall now make the following notational conventions:

- Array notation will be used when referring to access and update of code, to the heap and to globals. For example, \( C[3] \) denotes the third instruction of object code \( C \) and \( C[4] \leftarrow I \) represents an update of \( C \) in which the fourth instruction is replaced by \( I \).
- The list notation will be used when referring to the stacks \( S \), \( IS \), \( D \). For example, the empty stack will be denoted by \([\] \) and \( \langle a : as \rangle \) denotes the stack with top \( a \) and rest \( as \).

Using these notations we represent the initial state of the abstract MILONGA machine as follows:

\[
\begin{array}{cccccccc}
C_{inic} & 1 & 1 & H_{inic} & 1 & G_{inic} & \langle \text{null}, \text{null} \rangle \\
\end{array}
\]

An initial state of the abstract MILONGA machine encodes the pCode of a MILONGA function. The following explanation of the meaning of the entries of an initial machine state contains the necessary information about this encoding:

- \( C_{inic} \): PushFunc “main”.
- \( H_{inic} \): heap, with a NFunct node for each compiled function.
- \( G_{inic} \): maps a function to a node address of \( H_{inic} \).
- \( \text{null} \): denotes the null pointer.

A final state of the machine is any state that satisfies the following conditions:

- \( C[IP] = \text{Ret} \).
- \( D = [\] \) in the meaning of: the dump is empty.

As result of the machine computation appears the node of the heap \( H \) with the address located at the top of the stack \( S \).

4.2. State transition rules

We have now established the basic structure of the states of the abstract MILONGA machine and in particular specified its initial and final states. It remains now to specify the behaviour of the machine in each particular state. Given an initial state encoding a pCode representation of a MILONGA script, the machine evolves from the initial state, according to certain state transition rules, until it reaches either a final state or a state for which no transition rule exists. If the machine reaches a final state, the result of the machine computation, namely the node of the heap \( H \) corresponding to the top of the stack \( S \), is returned. Otherwise the computation stops and no result is returned (this may happen when a dynamic type error or division by zero occurs).
Each transition rule of the abstract MILONGA machine is formulated by means of a pair of machine states. The first element of such a pair corresponds to the precondition of the rule and the second element of the pair corresponds to the postcondition of the rule. After the application of the rule the abstract machine evolves to the second entry (postcondition) of the rule. The application of the state transition rules of the abstract MILONGA machine is triggered by means of instructions (see below). For example, jump instructions mainly modify the instruction pointer IP, the Pop instruction is mainly related to the stack S, etc. Other rules involve vectors, matrices and straight-line programs and unary and binary operations.

As already mentioned before, the application of the state transition rules of the abstract MILONGA machine becomes deterministic by the use of certain working instructions, listed in the component C of each state. We are going to enumerate and to explain these instructions.

4.2.1. Instructions that involve the pointer IP

These instructions provide mechanisms for implementing unconditional and conditional jumps.

- **Label n** is used to determine the destination of a conditional or unconditional jump.
- **Jmp n, JmpF n and JmpT n** modify the instruction pointer in a nontrivial way. **Jmp n** is an unconditional jump. The instructions **JmpF n** and **JmpT n** are conditional jumps, and their execution depends on the content of the top of the stack IS, while the truth value True is represented by the integer 1 and the truth value False is represented by the integer 0. These instructions are generated when branchings, iterations and multiple applications are compiled.
- **Ret** returns the control from a called function to its calling function. It restores the code of the calling function and the value of the corresponding instruction pointer (here we suppose that both functions have been previously saved on the top of the stack D). If the stack D is empty, then a final state has been reached, and the computation of the abstract MILONGA machine stops.

4.2.2. Instructions that involve the stack S

- **Alloc n** adds n elements to the top of stack S. It is used to create space in order to allocate place for local variables of a function.
- **Chk t** performs dynamic type-checking. It is used to cope with situations in which no static type-checking is possible. The following code is an example of that kind of situation:
  
  ```
  test (o:TObj) result r:TNum
  {
      r := o[3]
  }
  ```

  This code does not admit static type-checking, since it is not possible to know during the compilation whether, in run-time, variable o will be of type vector, or whether o[3] will be of type number, as required by the semantics. Therefore, we need to perform dynamic type-checking during run-time.
- **Push n** gives access to the n-th element of a stack of noninteger variables (it is used whenever a noninteger variable appears within an expression).
- **Pop n** cleans up the stack after the execution of a function.
- **Update n** is responsible for maintaining the actual address of an unevaluated expression.
- **Swap** swaps the first two elements of the stack. It is used after code optimization.
MkCopy creates a copy of the node pointed by the top of the stack \( S \). In fact, only when the type of this node is of type \( \text{NVec}, \text{NMat}, \text{NBox} \) or \( \text{NSlp} \) a copy is produced. If the type of this node is \( \text{NBool}, \text{NInt}, \text{NNum} \), the node becomes shared. The macro-instruction MkCopy avoids aliasing.

Split \( n \) is used to decompose an interleaved vector in its original components. This instruction is used in the following situation: suppose that we have to derive a \textsc{MILONGA} function which contains \( n \) extra parameters. Then its derivative contains \( 2n \) parameters which become stored in an interleaved vector of length \( 2n \). \textsc{MILONGA} stores each extra parameter of the original function followed by the derivative of the parameter. The instruction Split \( n \) decomposes this vector into two vectors of length \( n \) separating the extra parameters of the original function from those of its derivative.

MkAp is used when a partially evaluated function has to be applied to new parameters.

PushFunc \( t \) and PushDeriv \( t \) are instructions which are applied when a constant of function type is used within an expression.

4.2.3. Instructions that involve the integer stack \( IS \)

IAlloc \( n \), IPop \( n \), IPush \( n \) and IUpdate \( n \) are instructions which operate on the stack \( IS \) in the same way as the instructions Alloc \( n \), Pop \( n \), Push \( n \) and Update \( n \) do on the stack \( S \).

PushInt \( i \) is used to move an integer constant on the top of stack \( IS \).

4.2.4. Instructions that involve the relationship between the stack \( S \) and the integer stack \( IS \)

Instructions ToI and ToS are used for the transfer of contents between the heap \( H \) and the integer stack \( IS \).

Instruction ToI works as follows: first it scans the upmost element of the stack \( S \). This element is an address, say \( a \), pointing to node of the heap containing an integer value, say \( i \). The value \( i \) is then pushed on stack \( IS \), whereas the address \( a \) is popped from stack \( S \). Finally the corresponding node is erased from the heap.

Instruction ToS works in the same way, but in the opposite direction: it scans the upmost element of the stack \( IS \), which is again an integer \( i \), and pushes the address of the heap node with content \( i \) on stack \( S \). Finally, instruction ToSN works like instruction ToS changing additionally the type of the integer \( i \) from \( \text{NInt} \) to \( \text{NNum} \).

4.2.5. Instructions that control modular arithmetic

The instruction CRN depends on an integer module, whereas the instruction CRV depends on an ideal of the ring of univariate polynomials over \( \mathbb{Z} \). Both instructions represent modular change commands which allow during the execution of a \textsc{MILONGA} procedure to switch from infinite to finite precision arithmetic modulo a given ideal (and vice versa). The effect of any of the instructions CRN and CRV starts with its instantiation and runs through all its dynamic descendants until the command is overwritten by a new change command. For more details, see Section 2.3.2.

4.2.6. Instructions that involve vectors, matrices and straight-line programs

MkVec and MkMat build vectors and matrices whose initial value is set to zero for every component.

MkFVec and MkFMat build vectors and matrices whose entries are determined by a generating function.

GetVec and GetMat allow access to individual entries of vectors and matrices.
Table 8

Binary instructions operating on stack IS

<table>
<thead>
<tr>
<th>IOp</th>
<th>IOp,</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAAdd</td>
<td>+</td>
</tr>
<tr>
<td>IMul</td>
<td>*</td>
</tr>
<tr>
<td>IMod</td>
<td>%</td>
</tr>
<tr>
<td>IMDiv</td>
<td>/</td>
</tr>
<tr>
<td>IEq</td>
<td>==</td>
</tr>
<tr>
<td>INe</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>ILe</td>
<td>&lt;=</td>
</tr>
<tr>
<td>ILt</td>
<td>&lt;</td>
</tr>
<tr>
<td>IGe</td>
<td>&gt;=</td>
</tr>
<tr>
<td>ILt</td>
<td>&gt;</td>
</tr>
<tr>
<td>IAnd</td>
<td>&amp;</td>
</tr>
<tr>
<td>IOr</td>
<td></td>
</tr>
</tbody>
</table>

- **PutVec** and **PutMat** allow to update individual entries of vectors and matrices. However, it is not possible to update individual entries if the vector or matrix under consideration has been created by means of an instruction **MkFVec** or **MkFMat**.
- **GetDimV**, **GetDimF** and **GetDimC** are used for the inspection of the length or the number of rows and columns of a given vector or matrix.
- When **MILONGA** compiles an expression which computes the number of extra parameters of a given function, then the instruction **MkSlp** is generated. This instruction creates a **NSlp** node taking into account the following information:
  - the address of the corresponding node **NBox**;
  - the variable that eventually has to be derived;
  - the number of extra variables.

- **GetDimE** is used for the inspection of the number of extra parameters that remain unevaulated in a given node. If this node is of type **NBox**, the truth value 0 is returned, since the number of extra parameters is still unknown.

4.2.7. Unary and binary instructions

**Neg**, **INeg** and **INot** are unary instructions with the meaning of sign change of integers, nonintegers and boolean values, respectively. **MILONGA** contains also a couple of instructions associated to binary arithmetic with integers and relational boolean operations. These instructions are listed in Tables 8 and 9.

Finally, **MILONGA** contains also instructions corresponding to arbitrary arithmetic operations. Their nature is polymorphic. Their operands may be either numbers, vectors or matrices. In each case the result has the appropriate type (the rules for typing arithmetic operations were already discussed in Section 2.3.2). These instructions are listed in Table 10.

Table 9

Relational instructions operating on stack IS

<table>
<thead>
<tr>
<th>OpRel</th>
<th>OpRel,</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEq</td>
<td>==</td>
</tr>
<tr>
<td>ILt</td>
<td>&lt;</td>
</tr>
<tr>
<td>IGe</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>
Table 10

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>+</td>
<td>Sub</td>
<td>−</td>
</tr>
<tr>
<td>Mul</td>
<td>*</td>
<td>Div</td>
<td>/</td>
</tr>
</tbody>
</table>

4.2.8. Special rules to be applied in case that some NBox or NSlp node becomes saturated

There are two special transition rules that are not triggered by an instruction (contained in the component C[IP]). The first rule is triggered when some NBox node of the heap becomes saturated and the second one is triggered when the same thing happens to some NSlp node of the heap. The priority of these two rules is higher than that of any other transition rule. The application of these two rules depends on a condition of node saturation. The verification of this condition is simplified by the following remark:

- A NBox node may become saturated only after execution of instruction PushGlobal or MkAp.
- A NSlp node may become saturated only after execution of instruction MkAp.

4.3. Execution time of MILONGA programs

In this section we discuss the impact of theoretical complexity estimates of geometric elimination procedures in the execution time of their MILONGA implementations.

The elimination algorithms for which MILONGA was designed require only recursion with bounded quantifiers expressible by for iterators. The a priori complexity bounds of these algorithms allow to bound effectively the size of their for loops. In conclusion, these algorithms may be implemented using only for loops of predetermined size, branchings and assignments.

Let us now consider a given MILONGA script \( S \) encoding in the way described before a subroutine of an elimination algorithm. Suppose that the goal of this subroutine is the evaluation of a certain function \( f \). For the sake of simplicity, let us first suppose that \( S \) is a primitive program, i.e. that \( S \) does not contain calls to other procedures than itself (we postpone the discussion of more complex programs until later).

The fragment of \( S \) which refers to the input of the algorithm contains certain fixed and extra parameters.

The execution time of the C++ compiled version of \( S \) is determined by the following measures:

- The maximum for loop size \( \delta \) of \( S \) (observe that in general \( \delta \) will not be a constant, but a function which depends on certain integer-valued (fixed) parameters of \( S \) having an algebraic or geometric meaning, e.g. number of variables and degree of the polynomials constituting the input of the elimination algorithm encoded by \( S \)).
- The maximum depth \( \ell \) of nested for’s contained in \( S \) (here \( \ell \) will always be an input independent constant of \( S \)).
- The number \( L \) of assignments contained in \( S \) (again \( L \) is an input independent constant of \( S \)).
- The maximum bit length of integers which have to be stored during the execution of the algorithm encoded by \( S \) on a given input.

We claim that the boolean time complexity of the theoretical algorithm encoded by the script \( S \) is of order \( O(\ell L^\delta m(h)) \), where \( m(h) \) denotes the boolean time complexity of the multiplication of \( h \)-bit numbers. Furthermore we claim that the execution time of the C++ compilation of the MILONGA
The instructions representing arithmetic operations; MkMat and MILONGA to another procedure. Observe that, in the context of a larger pCode representation is straightforward and that the length of the resulting pCode is linear in the length of S. This is due to the fact that each MILONGA script instruction is translated into a constant number of pCode instructions. In particular, the parameters δ, ℓ, L and h still characterize the execution time of the pCode translation of S.

Let us now consider the execution time of the C++ compilation of the pCode representation of the MILONGA script S. For this purpose, we subdivide the pCode instructions (introduced in Section 4.2), according to their execution time, into four groups:

(i) the instructions MkVec, MkFVec, MkMat and MkFMat;
(ii) the instruction MkAp;
(iii) the instructions representing arithmetic operations;
(iv) all the remaining pCode instructions.

It is easy to see that the execution (running) time of any pCode instruction belonging to group (iv) is of order O(n log δ). Let us also observe that by definition the execution time of any instruction belonging to group (iii) is of order M(h).

Let us consider now the running time of the instructions MkVec, MkFVec, MkMat and MkFMat. Let δ1, δ2 be two positive integers. Then the instructions MkVec δ1 and MkFVec δ1 build a δ1-dimensional vector whose entries are determined by a generating function. Therefore, their running time is of order O(δ1). Similarly, the instructions MkMat (δ1, δ2) and MkFMat (δ1, δ2) build a (δ1 × δ2)-dimensional matrix, and their running time is of order O(δ1δ2). From [23] we deduce that the number of register allocations of arithmetic data used during the execution of the C++ compilation of the MILONGA script S is controlled by the number of arithmetic operations involved in the underlying theoretical algorithm. In the given context this implies that the total running time of the pCode instructions MkVec, MkFVec, MkMat and MkFMat during the execution of S is of order O(Lδ′).

Now we are going to analyze the effect of the instruction MkAp on the execution time of the MILONGA script S. This analysis will finally allow us to show our claim, namely that the execution time of the C++ compilation of the MILONGA script S is of order O(Lδ′M(h)) and that the underlying theoretical algorithm has boolean time complexity O(Lδ′m(h)).

We prove our claim by induction in the depth of nesting of function (procedure) calls by the MILONGA script S (these functions calls are represented by MkAp instructions in the corresponding pCode).

As before, let us suppose that the MILONGA script S under consideration does not contain any call to another procedure. Observe that, in the context of a larger MILONGA program S′, any call to the procedure S triggers an evaluation of the function f in certain argument values and that this call is represented in the pCode of S′ by a MkAp instruction, which points to the pCode of S. Thus, during the execution of the program S′, the C++ code of this MkAp instruction triggers the evaluation of the function f computed by the MILONGA script S. Taking into account the running time of the instructions belonging to groups (i), (ii) and (iv), we conclude that the running time of C++ implementation of S is of order O(Lδ′(n log δ + M(h))) = O(Lδ′M(h)). The analysis of the boolean time complexity of
the theoretical algorithm encoded by the MILONGA script $S$ is straightforward and yields the bound $O(L\delta\ell M(h))$.

Now suppose that the function $f$ computed by the MILONGA script $S$ depends on $r$ functions $g_1, \ldots, g_r$, which on their turn are represented by MILONGA scripts $S_1, \ldots, S_r$. Let us introduce the following notations:

- Let $\tilde{L}, L_1, \ldots, L_r$ denote the number of assignments—different from function calls—of the MILONGA scripts $S, S_1, \ldots, S_r$.
- Let $\tilde{\ell}, \ell_1, \ldots, \ell_r$ denote the depth of nested for's of the MILONGA scripts $S, S_1, \ldots, S_r$.
- Let $\tilde{\delta}, \delta_1, \ldots, \delta_r$ denote maximum for loop size of the MILONGA scripts $S, S_1, \ldots, S_r$.
- Let $k_1, \ldots, k_r$ denote the number of times that the functions $g_1, \ldots, g_r$ are called by the MILONGA script $S$.

By hypothesis of induction, for any $1 \leq i \leq r$, the function $g_i$ can be evaluated for the given input $x$ using $O(L_i\delta_i\ell_i m(h))$ boolean operations. Moreover, the execution time of the C++ implementation of the MILONGA script $S_i$ is $O(L_i\delta_i\ell_i M(h))$. Thus, the theoretical algorithm encoded by the MILONGA script $S$ evaluates the function $f$ on the given input $x$ using $O(L\delta\ell M(h)) = O((\tilde{L}\tilde{\delta} + \sum_{i=1}^r k_i L_i \delta_i^2) m(h))$ boolean operations. On the other hand, the MILONGA script $S$ contains $\sum_{i=1}^r k_i$ calls to the functions $g_1, \ldots, g_r$, namely for $1 \leq i \leq r$, there are $k_i$ calls to the evaluation of the function $g_i$. In the code of $S$ these function calls are represented by MkAp instructions.

This implies that all the calls to the evaluation of the functions $g_1, \ldots, g_r$ for the given input $x$ can be executed in time $O(M(h) \sum_{i=1}^r k_i L_i \delta_i^2)$. Therefore, in the context of a larger MILONGA program $S^*$, any call to the procedure $S$ (which is represented in the pCode of $S^*$ by a MkAp instruction pointing to the pCode of $S$), can be executed in time $O\left((\tilde{L}\tilde{\delta} + \sum_{i=1}^r k_i L_i \delta_i^2) M(h)\right) = O(L\delta M(h))$.

This finishes the proof of our claim.

Let us observe that the same argumentation may be applied to the execution time of the given MILONGA $S$ script on the abstract MILONGA machine.

Moreover, a similar analysis allows us to conclude that the space complexity (expressed in terms of a pebble game) of a given elimination algorithm is transformed into a space requirement of the same order for the execution of the corresponding MILONGA procedure on a real world computer or on the abstract MILONGA machine. This space requirement has to be taken into account for the determination of the heap size during the execution of the MILONGA procedure. Observe the following time–space tradeoff for the execution of MILONGA programs: minimization of the heap size triggers frequent garbage collection and therefore loss of run-time (see Section 3.6).
In this section we present implementation and comparative benchmarking of some key algorithms in effective geometric elimination theory. The tests for the benchmarking were performed on a Pentium II processor of 400 MHz and 128MB of RAM memory under Linux 2.0.36 as operating system. Timings are measured in CPU seconds. We did run the MILONGA compiler using at most 1MB of RAM memory. As contrasting software for our MILONGA implementation we used Maple V3.0R4.

The aim of this section is two-fold: on one hand, by the exhibition of concrete algorithms and implementations we shall illustrate how programming in MILONGA works. On the other hand, the theoretical time complexity analysis of the C++ compilation of MILONGA programs achieved in Section 4.3 does not describe all practical aspects of the execution of MILONGA codes. The practical relevance of our theoretical analysis has to be validated by the run-time and memory space profile of concrete implementations.

5.1. Characteristic polynomial of polynomial matrices

One of the main aspects of symbolic computation concerns operating and computing with matrices having polynomial entries. This type of matrices typically occurs in multivariate polynomial equation solving over the complex or real numbers (geometric elimination). Polynomial equation solving is nowadays one of the main research areas of symbolic computation in view of potential practical applications of mathematical theory to science and engineering (see e.g. [19]).

Unfortunately, the currently available standard software for symbolic computation (for example the general purpose software packages Maple or Mathematica) is still unable to perform in real time or memory space the necessary computations for the solution of any multivariate polynomial equation system of practical interest. Furthermore, typic linear algebra problems with matrices having their entries in the polynomial ring \( \mathbb{Q}[X_1, \ldots, X_m] \), such as the computation of the determinant, the characteristic polynomial, the rank or the resolution of the associated system of linear equations, require a huge amount of computational resources, even for matrices of extremely small size (compare the benchmarks in [7,14]).

In this section, using the straight-line program approach to symbolic computation implemented in MILONGA, we are going to treat with the problem of computing the characteristic polynomial of an \( n \times n \) matrix \( A = (a_{ij})_{1 \leq i,j \leq n} \) whose entries \( a_{ij} \) belong to \( \mathbb{Q}[X_1, \ldots, X_m] \). We assume that the entries of the matrix \( A \) are given by a division-free straight-line program \( \beta \) in \( \mathbb{Q}[X_1, \ldots, X_m] \). Our aim is to compute the coefficient vector of the characteristic polynomial \( \chi_A(T) := \det(A - T \cdot \text{Id}) \) of the matrix \( A \). Our approach will return the entries of this coefficient vector in form of a division-free straight-line program in \( \mathbb{Q}[X_1, \ldots, X_m] \).

For this purpose, we shall consider Samuelson’s algorithm (see [11]) which computes the characteristic polynomial of any square matrix whose entries lie in a zero-divisor free commutative ring. Samuelson’s algorithm is known to be very adapted to polynomial matrices, since it succeeds to keep small the size of its intermediate results (see [1,29,32]).

Samuelson’s algorithm can be described by a straight-line program which, taking the entries of the square matrix \( A \) as input, computes the coefficient vector of its characteristic polynomial as output. Therefore, by appending the code of this algorithm to that of the straight-line program \( \beta \) computing the entries of the matrix \( A \), we obtain a straight-line program that, for a given input point \( \alpha := (\alpha_1, \ldots, \alpha_m) \in \mathbb{Q}^m \), computes the characteristic polynomial of the matrix \( A(\alpha) := (a_{ij}(\alpha))_{1 \leq i,j \leq n} \). We exhibit now the MILONGA code of Samuelson’s algorithm for the computation of the characteristic polynomial of a square matrix (see e.g. [1,11]).
calcC (m: TMat, k: TInt) result v:TVec local (i, j: TInt, r, s: TVec, subM: TMat)
{
  v := newv (k + 1);
  v[0] := 1;
  v[1] := m [k, k];
  if k >= 2 then
  {
    r := getRow m k 1 (k - 1);
    s := getCol m k 1 (k - 1);
    subM := subMatrix m 1 1 (k - 1) (k - 1);
    for i = 2 to k - 1 do
      v[i] := innerProd r s;
      s := subM * s
  }
  v[k] := innerProd r s
}
samuelson (sm: TSlp TMat) extras x(#e sm) result v:TVec
local (i:TInt, m:TMat)
{
  m := sm (dot i := 0 to #v x - 1 ap x[i]);
  v := calcC m 1;
  for i := 2 to (#f m) do v := toepMult v (calcC m i);
  v := reverse v
}

As an application of this MILONGA routine, we compute the characteristic polynomial of a test sequence of square matrices $A$ whose entries are random sparse homogeneous polynomials of degree 100 in 9 variables with 10 nonzero integer coefficients from the interval $[1, 10]$. The following table contains the comparison of the run-time behaviour of the MILONGA implementation of Samuelson’s algorithm and of the Maple standard procedure for the computation of the characteristic polynomial. The value of 462.97 in this example (MILONGA) was obtained using a heap size of 10MB.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>MILONGA</th>
<th>Maple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
<td>0.10</td>
</tr>
<tr>
<td>2</td>
<td>0.17</td>
<td>0.43</td>
</tr>
<tr>
<td>3</td>
<td>0.37</td>
<td>1.23</td>
</tr>
<tr>
<td>4</td>
<td>0.68</td>
<td>error.otl</td>
</tr>
<tr>
<td>10</td>
<td>4.36</td>
<td>error.otl</td>
</tr>
<tr>
<td>20</td>
<td>21.88</td>
<td>error.otl</td>
</tr>
<tr>
<td>30</td>
<td>71.89</td>
<td>error.otl</td>
</tr>
<tr>
<td>40</td>
<td>213.30</td>
<td>error.otl</td>
</tr>
<tr>
<td>50</td>
<td>462.97</td>
<td>error.otl</td>
</tr>
</tbody>
</table>
We observe that Maple run out of memory in case of matrices of higher size than $3 \times 3$. This is due to the fact that Maple writes down the dense representation of the intermediate results of the computation, which exceed the memory space assigned (11MB). Let us also remark that in the case of matrix $A$ having size $50 \times 50$, MILONGA uses about 500KB of memory space. This illustrates that an alternative data structure which improves dramatically the space requirements of a procedure may still be insufficient for practical computer implementations. The enormous memory space required by our MILONGA procedure for the computation of the characteristic polynomial is due to the explicit storing of the resulting straight-line program. This problem is discussed in detail in the case of the Samuelson algorithm in [14] and the conclusion is the following: deforestation of straight-line programs representing polynomials [44] furnishes the most efficient pattern for the execution of concrete computer programs in elimination theory.

The MILONGA code for the generation of our test suite is the following:

```plaintext
myPolynomial (i,j:TInt) extras x(9) result res:TNum
local (k:TInt)
{
    res := ... // the code used for the generation of
    // the random polynomials comes here
}
myMatrix (dim:TInt) extras x(9) result m:TMat
// SLP computing the matrix M
local (i,j,k:TInt)
{
    m := newM dim dim;
    for i := 1 to dim do for j := 1 to dim do
        m[i,j] := myPolynomial i j (dot k := 0 to 8 ap x[k])
}
main result res:TObj local (mySlp:TSlp TVec)
{
    mySlp := samuelson (myMatrix askTInt);
    res := mySlp $1 $2 $3 $4 $5 $6 $7 $8 $9
}
```

5.2. Vermeidung von Divisionen in MILONGA

A typical technical task in the manipulation of straight-line programs is the elimination of (essential) divisions in the evaluation of a given polynomial. The prototype of algorithms which solve this task efficiently is due to Strassen [42]. These algorithms have the generic name Vermeidung von Divisionen (avoiding divisions). In this section we present a MILONGA implementation of such an algorithm.

Let us be given a division-free straight-line program $\beta$ in $\mathbb{Q}[X_1, \ldots, X_n]$ evaluating polynomials $F_0, F_1$ of $\mathbb{Q}[X_1, \ldots, X_n]$. Assume that $F_0$ divides $F_1$ in $\mathbb{Q}[X_1, \ldots, X_n]$ and let $F \in \mathbb{Q}[X_1, \ldots, X_n]$ denote the resulting polynomial $F := F_1/F_0$. The task of avoiding divisions in the evaluation of $F$ can now be stated as follows: find a division-free straight-line program $\tilde{\beta}$ in $\mathbb{Q}[X_1, \ldots, X_n]$ evaluating the polynomial $F$. 

Avoiding divisions in the evaluation of a given polynomial $F \in \mathbb{Q}[X_1, \ldots, X_n]$ is an indispensable tool to obtain a reliable computation scheme for the polynomial $F$. Such schemes represent a basic ingredient for the kind of elimination algorithms at the origin of the design of the MILONGA language. In [42], Strassen describes an algorithm solving this task assuming that an upper bound $d$ for the degree of the polynomial $F$ is given. We are now going to solve the same task using an adaptation of Strassen’s idea. This adaptation can be found in [23] (see also [29]).

Let us be given an integer point $\gamma := (\gamma_1, \ldots, \gamma_n) \in \mathbb{Z}^n$ such that $\rho := F_0(\gamma)$ is a value different from zero. Let us observe that, using the Zippel–Schwartz test (see [40] or [48]), such a point $\gamma$ can be found randomly in the hypercube $[1, 2d]^n \cap \mathbb{Z}^n$ at an average cost of time twice the length of the straight-line program $\beta$.

For $i = 0, 1$, let us write $G_i(X_1, \ldots, X_n) := F_i(X_1 + \gamma_1, \ldots, X_n + \gamma_n)$. One easily sees that the polynomial $F(X_1 + \gamma_1, \ldots, X_n + \gamma_n) = F_1(X_1 + \gamma_1, \ldots, X_n + \gamma_n)/(F_0(X_1 + \gamma_1, \ldots, X_n + \gamma_n))$ can be computed as the sum of the first $d + 1$ homogeneous components of the polynomial $(G_1/\rho) \sum_{k=0}^{d}(\rho - G_0)/\rho$.

For the computation up to degree $d$ of the homogeneous components of a polynomial $G \in \mathbb{Q}[X_1, \ldots, X_n]$, supposed to be given by a straight-line program $\mu$, we follow an idea of [34], which reduces the task to polynomial arithmetic in $\mathbb{Q}[X_1, \ldots, X_n][T]$ modulo $T^{d+1} + 1$. For this purpose, we simply observe that the homogeneous components of $G$ arise as the coefficients of the representation of the residue class of the polynomial $G(TX_1, \ldots, TX_n)$ modulo $T^{d+1}$ in the basis $1, T, \ldots, T^d$. In order to compute these coefficients, we execute the straight-line program $\mu$ on input $TX_1, \ldots, TX_n$ in the quotient ring $B := \mathbb{Q}[X_1, \ldots, X_n][T]/(T^{d+1} + 1)$, representing each intermediate result of the computation by the vector of $\mathbb{Q}[X_1, \ldots, X_n][T]$ which corresponds to the coefficients of the result in the basis $1, T, \ldots, T^d$.

This idea is now applied to the polynomial $G := (G_1/\rho) \sum_{k=0}^{d}(\rho - G_0)/\rho$.

**Algorithm for the computation of the homogeneous components of a straight-line program represented polynomial**

```plaintext
hom (beta:TSlp TObj), delta:ToInt) extras x (#e beta) result v:TVect local (i:ToInt, e:TVect) 
{ 
e2 := eVec (delta + 1) 2;
CRV (eVec (delta + 1) (delta + 1)); // Modular arithmetic
v := beta (dot i := 1 to (#e beta) ap e2 * x[i])
}
```

**Algorithm Vermeidung von Divisionen**

```plaintext
changeVars (f:TSlp TObj, alfa:TVect) extras x(#e f) result r:TObj local (i:ToInt) 
{ 
r := f (dot i := 0 to (#e f - 1) ap (x[i] + alfa[i]))
}
```

```plaintext
getAlfa (f:TSlp TObj, delta:ToInt) result alfa:TVect local (i:ToInt, rho:TObj) 
{ 
```
\[
\text{rho} := 0;
\]
while (rho == 0) do
\{
\text{alfa} := \text{randomVector} \{\#e f\} \text{ (delta)} \text{ delta};
\text{rho} := f \{\text{dot i} := 0 \text{ to (} \#e f - 1 \text{) ap alfa}[i]\} \\
\}
\]
calcP (g, h:TSlp TObj, rho:TNum, delta:TInt) \text{ extras x(} \#e g\text{) result r:TObj} \text{ local (i:TInt, q,qPow:TObj, rhoPow:TNum)} \{ 
q := \text{rho} - g \{\text{dot i} := 0 \text{ to (} \#e g - 1 \text{) ap x}[i]\};
rhoPow := \text{pow rho delta};
qPow := \$1; r := \$0;
for i := 0 \text{ to delta} \text{ do} \\
\{ 
\text{r} := r + qPow * rhoPow;
qPow := q * qPow;
rhoPow := \text{rhoPow/rho} \\
\}
r := r * h \{\text{dot i} := 0 \text{ to (} \#e g - 1 \text{) ap x}[i]\} \\
\}
calcQ (p:TSlp TVec) \text{ extras x(} \#e p\text{) result r:TObj} \text{ local (v:TVec, i:TInt)} \{ 
\text{r} := \$0;
v := p \{\text{dot i} := 0 \text{ to (} \#e p - 1 \text{) ap x}[i]\};
\text{for i := 0 \text{ to (} \#v v - 1 \text{) do r} := r + v[i]} \\
\}
\text{vvd(f1,f0:TSlp TObj, delta:TInt) extras x(} \#e f0\text{) result res:TObj} \text{ local (alfa:TVec, rho:TNum, p,q,h:TSlp TObj, i:TInt)} \{ 
\text{alfa} := \text{getAlfa f0 delta};
\text{rho} := f0 \{\text{dot i} := 0 \text{ to (} \#e f0 - 1 \text{) ap alfa}[i]\};
p := \text{calcP} \{\text{changeVars f0 alfa} \text{ (changeVars f1 alfa) rho delta}\};
q := \text{calcQ} \text{ (hom p delta)};
h := \text{changeVars q \{\$1 * alfa\}};
\text{res} := h \{\text{dot i} := 0 \text{ to (} \#e h - 1 \text{) ap x}[i]/\text{pow rho (delta + 1)}\} \\
\}

We applied the MILONGA implementation of our Vermeidung von Divisionen algorithm to a test suite which was generated in the following way.

Let \(X_1, \ldots, X_n, Y\) be indeterminates over \(\mathbb{Q}\) and let \(P, Q\) the following polynomials of \(\mathbb{Q}[X_1, \ldots, X_n]\):
\[ P(X_1, \ldots, X_n, Y) = \det \begin{bmatrix} 1 & X_1 & \cdots & X_1^n \\ 1 & X_2 & \cdots & X_2^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_n & \cdots & X_n^n \end{bmatrix} = \prod_{i \neq j} (X_i - X_j) \prod_{i=1}^n (Y - X_i), \]

\[ Q(X_1, \ldots, X_n) = \det \begin{bmatrix} 1 & X_1 & \cdots & X_1^{n-1} \\ 1 & X_2 & \cdots & X_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_n & \cdots & X_n^{n-1} \end{bmatrix} = \prod_{i \neq j} (X_i - X_j). \]

It is clear that \( Q \) divides \( P \) in \( \mathbb{Q}[X_1, \ldots, X_n][Y] \) and that the polynomial \( P/Q \) is defined by the formula \( P/Q = \prod_{i=1}^n (Y - X_i) \) which is well defined for any specialization of the variables \( X_1, \ldots, X_n \).

Applying now the MILONGA implementation of our Vermeidung von Divisionen algorithm to this situation, we obtain a straight-line program which evaluates the polynomial \( P/Q \) without using divisions.

The following is the MILONGA code for this example:

```markdown
numerator (n:TInt, v:TNum) extras x(n) result o:TObj local (i:TInt) {
  o := (detVander (n + 1)) v (dot i := 0 to (n - 1) ap x[i])
}
denominator (n:TInt) extras x(n) result o:TObj local (i:TInt) {
  o := detVander n (dot i := 0 to (n - 1) ap x[i])
}
detVander (i:TInt) extras x(i) result n:TObj local (f,c,k:TInt, m:TMat) {
  m := newM i i;
  for f := 1 to i do
    for c := 1 to i do m[f, c] := pot (x[f - 1]) (c - 1);
  n := det m
}
main result res:TObj local (slp:TSlp TObj, i,n:TInt) {
  n := askTInt;
  slp := vvd (numerator n $@((n + 1)) (denominator n) (n + 2);
  res := slp (dot i := 1 to n ap $i)
}
```

On the other hand, for Maple there are two different options to evaluate the polynomial \( P/Q \) on a given point \( \alpha := (\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}^n \): either we compute first the integers \( P(\alpha) \) and \( Q(\alpha) \) and then the
integer value $P(\alpha)/Q(\alpha)$, or we compute the polynomial $P/Q$ by symbolic simplification and then we evaluate the result in the point $\alpha$.

Using the first option for $n = 3$, we have the following Maple code:

```maple
fract := (x1, x2, x3, y) -> det(vandermonde([x1, x2, x3, y])) / det(vandermonde([x1, x2, x3]))
```

We observe that Maple is unable to compute $\text{fract}(2,2,2,5)$, returning

```
Error, (in fract) division by zero
```

whereas MILONGA returns the correct value 27.

In order to avoid this difficulty we have chosen the second Maple option for our comparison with MILONGA. These are the corresponding Maple code and the timings of the comparison:

```maple
with(linalg):
numerator := n -> vandermonde([seq(x[i], i = 1, ..., n)]);
denominator := n -> vandermonde([seq(x[i], i = 1, ..., n - 1)]);
main_result := n -> simplify(det(numerator(n))/det(denominator(n)));
```

<table>
<thead>
<tr>
<th>n</th>
<th>MILONGA</th>
<th>Maple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>0.06</td>
<td>0.01</td>
</tr>
<tr>
<td>4</td>
<td>0.19</td>
<td>0.01</td>
</tr>
<tr>
<td>5</td>
<td>0.58</td>
<td>0.14</td>
</tr>
<tr>
<td>6</td>
<td>1.33</td>
<td>1.66</td>
</tr>
<tr>
<td>7</td>
<td>3.06</td>
<td>error.otl</td>
</tr>
<tr>
<td>8</td>
<td>6.55</td>
<td>error.otl</td>
</tr>
<tr>
<td>9</td>
<td>12.94</td>
<td>error.otl</td>
</tr>
<tr>
<td>10</td>
<td>23.63</td>
<td>error.otl</td>
</tr>
<tr>
<td>15</td>
<td>321.46</td>
<td>error.otl</td>
</tr>
<tr>
<td>20</td>
<td>3077.34</td>
<td>error.otl</td>
</tr>
</tbody>
</table>

5.3. An application of automatic differentiation

In this section we consider an application of automatic differentiation which is fundamental for automatic integration: we exhibit a probabilistic algorithm which tests whether a vectorial field, whose coordinates are rational functions computed by a straight-line program, allows a potential function [33].

Let $F_1, \ldots, F_n, G_1, \ldots, G_n$ be polynomials of $\mathbb{Q}[X_1, \ldots, X_n]$ given by a division-free straight-line program. Assume that for $i = 1, \ldots, n$ the polynomial $G_i$ is nonzero and let $P_1 := F_1/G_1, \ldots, P_n := F_n/G_n$. Our goal is to decide whether the vectorial field $(P_1, \ldots, P_n)$ is a gradient field.

We observe that the vectorial field $(P_1, \ldots, P_n)$ is generically a gradient field if and only if $\partial P_i/\partial X_j = \partial P_j/\partial X_i$ holds for any $1 \leq i, j \leq n$. 

This condition can be rephrased requiring that for any $1 \leq i, j \leq n$ the identity 
\[
(G_iG_j)^2(\partial P_i/\partial X_j) = (G_iG_j)^2(\partial P_j/\partial X_i)
\]
holds. This leads us to a system of identity relations for polynomials involving only the polynomials $F_1, \ldots, F_n, G_1, \ldots, G_n$ and their first-order derivatives.

Using automatic differentiation one obtains the first derivatives of all polynomials $F_1, \ldots, F_n, G_1, \ldots, G_n$. Thus, the original problem is reduced to checking identities of polynomials which are given by straight-line programs. Let us here remark that efficient polynomial identity checking is a fundamental tool of most algorithms which manipulate polynomials in straight-line program representation. This tool is of crucial importance for elimination algorithms.

A simple minded way of checking polynomial identities consists in the interpolation of the (two) given polynomials, computing their coefficients from their values in suitably chosen interpolation points. Since straight-line program represented polynomials may have (in terms of the length of the given straight-line program) exponentially many coefficients, this procedure may become inefficient.

Instead of this approach we are going to use probabilistic polynomial identity checking for our MILONGA
d implementation. For this purpose we apply the so-called Zippel–Schwartz test \cite{40,48} in order to test the identity of two straight-line program represented polynomials of degree $d$ in $n$ variables.

In this way we obtain an algorithm which requires just one evaluation of the polynomials under consideration in a hypercube (grid) of the form $[1, g]^n \cap \mathbb{Z}^n$ with $g > d$. This algorithm has probability of success at least $(1 - ((d - 1)/g))^n$. Let us here observe that an alternative, more reliable (but less efficient) way to check polynomial identities is given by a test contained in \cite{25}.

Applying the mentioned Zippel–Schwartz test to our original problem of testing whether a given vector field is a gradient field, we obtain a probabilistic algorithm which tests whether the vectorial field $(P_1, \ldots, P_n)$ is generically the gradient field of a potential function. The probability of success $P_{\text{succ}}$ that this algorithm returns the correct answer can be estimated by
\[
P_{\text{succ}} \geq \left(1 - \left(\frac{d - 1}{g}\right)^n\right)^2
\]
where $d$ denotes the maximal degree of the input polynomials $F_1, \ldots, F_n, G_1, \ldots, G_n$ and $g$ denotes the side length of the hypercube used in the test. The following MILONGA code solves our original problem:

```milonga
maybeGradient extras x(n) result v:TVec
// (n has to be replaced by an appropriate value in order
// for this program to run)
// This is the input vector field
{
  v := newV n;
  v[0] := ...; // This is (dp/dx_0)
  ...;
  v[n-1] := ... // This is (dp/dx_(n - 1))
}
hessian extras x(#e maybeGradient) result m:TMat local
(i,j,e:TInt, v:TVec)
{
  e := #v x;
```
m := newM e e;
for i := 1 to e do
{ v := (derive maybeGradient i) (dot j := 0 to e - 1 ap x[j]);
  // Trigger for the automatic differentiation procedure
  for j := 1 to e do m[i,j] := v[j - 1]
}

isSymmetric (m:TMat) result r:TBool local (i,j:TInt)
{ r := True;
  for i := 1 to # of or j := 1 to # of m do
    if (m[i,j] >= m[j,i]) then r := False
}

main result isGradient:TBool local (i,gridSize:TInt)
{ gridSize := askTInt;
  isGradient := isSymmetric (hessian (dot i := 1 to (# of maybeGradient)
  ap random 1 gridSize))
}

For example, in case $d = n = 10$ it suffices to choose $g \geq 36$ in order to obtain probability of success of the test $P_{\text{succ}} > 0.9999$.

Let us now apply this algorithm to the following test suite. For any $1 \leq i, j \leq n$, let $X_{ij}$ be an indeterminate over $\mathbb{Q}$ and let us denote by $D$ the polynomial defined as the determinant of the generic $n \times n$ matrix $A := (X_{ij})_{1 \leq i,j \leq n}$. Let $B = (b_{ij})_{1 \leq i,j \leq n}$ be the adjoint matrix of $A$ with $b_{ij} = (-1)^{i+j} \det M_{ij}$, where $M_{ij}$ denotes the $(j,i)$-minor of the matrix $A$ (i.e. the determinant of the $(n - 1) \times (n - 1)$ matrix obtained from $A$ deleting the $j$th row and the $i$th column). Observe that for any $1 \leq i, j \leq n$ the identity $\partial D/\partial X_{ij} = (-1)^{i+j} \det M_{ij} = b_{ij}$ holds. Therefore, the adjoint matrix $B$ represents for a suitable order of the variables $X_{ij}$, $1 \leq i, j \leq n$, the gradient field of the polynomial $D = \det(A)$.

In order to test the run-time of the automatic differentiation procedure of \textsc{MILONGA}, we used the algorithm of the beginning of Section 5.3 for the purpose of the experimental verification of the mathematical insight that the matrix $B$ represents a gradient field.

We now explain how \textsc{MILONGA} computes the entries of the adjoint matrix $B$. Let $p(T) := T^n + \cdots + p_1 T + p_0$ be the characteristic polynomial of the matrix $A$ and let us denote the $n \times n$ identity matrix by $A^0$. Then the following identity holds true:

$$B = -A^{n-1} - p_{n-1} A^{n-2} - \cdots - p_1 A^0$$

(2)

In order to evaluate the right-hand side of identity (2), we use Horner's rule. This requires the previous computation of the coefficients of the characteristic polynomial of the matrix $A$, which is performed by means of Samuelson's algorithm (see Section 5.1).
Algorithm for the computation of the adjoint matrix in MILONGA

Algorithm for the computation of the adjoint matrix in MILONGA

horner (v: TVec, x: TObj) result r: TObj local (f, i, h, k: TInt) {
  f := (#v v) - 1;
  r := v [f];
  h := 0;
  k := 1;
  i := (#v v) - 2;
  while i >= h do
    { r := r \* x + v [i ];
    i := i - k 
  }
}

adj (m:TMat) result resM:TMat local (i:TInt, b, b2:TVec) {
  b := samuelson m;
  b2 := newV (#v b - 1);
  for i := 0 to #v b2 - 1 do b2[i] := $0 - (b[i] + 1));
  resM := horner b2 m
}

Checking whether the adjoint matrix is a gradient field

maybeGradient (n:TInt) extras x(n \* n) result v:TVec
local (i, j, k: TInt, xm:TMat, ym: TMat) {
  v := newV (n \* n);
  xm := newM n n;
  ym := newM n n;
  k := 0;
  for i := 1 to n do
    for j := 1 to n do
      { xm[i,j] := x[k];  // define xm[i,j] = Xij
      k := k + 1
    }
  ym := adj xm;  // ym = adjoint (xm)
  k := 0;
  for i := 1 to n do
    for j := 1 to n do
      { v[k] := ym [i,j];
      k := k + 1
    }
main result isGradient:TBool local (n,i,gridSize:TInt)
{
    gridSize := 1000;
    n := askTInt;
    isGradient := isSymetric (hessian n (dot i := 1 to (n * n)
                           ap random 1 gridSize))
}

Executing the MILONGA routine maybeGradient (and all subsequent subroutines) on a test suite with \( n \) varying from 5 to 25, one obtains the following timings:

<table>
<thead>
<tr>
<th>( n )</th>
<th>MILONGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9.33</td>
</tr>
<tr>
<td>10</td>
<td>340.15</td>
</tr>
<tr>
<td>15</td>
<td>6231.10</td>
</tr>
<tr>
<td>20</td>
<td>28861.58</td>
</tr>
<tr>
<td>25</td>
<td>86950.24</td>
</tr>
</tbody>
</table>

The values of 6231.10, 28861.58 and 86950.24 were obtained using a heap size of 10, 20 and 50MB, respectively.

Let us remark that the comparison of these timings with the corresponding ones of the Maple package becomes senseless. In view of the experimental results of Section 5.1, Maple is unable to generate the input matrix \( B \), even for moderate \( n \).

5.4. An application of partial evaluation in MILONGA

Effective elimination theory in algebraic geometry goes back to Kronecker [30]. We present here a modern, complexity theoretic approach to this subject, based on the straight-line program representation of multivariate polynomials (see [13,16,37]). Let us consider the following fundamental task of effective elimination with polynomials:

Let \( X_1, \ldots, X_n \) be indeterminates over the rational numbers \( \mathbb{Q} \) and let \( \mathbb{Q}[X_1, \ldots, X_n] \) denote the ring of \( n \)-variate polynomials over \( \mathbb{Q} \). Let \( d \) be a natural number and let be given polynomials \( F_1, \ldots, F_s \in \mathbb{Q}[X_1, \ldots, X_n] \) of degree at most \( d \). Let \( V \) be the algebraic variety defined by the polynomials \( F_1, \ldots, F_s \) in \( \mathbb{C}^n \), i.e. \( V := \{ x \in \mathbb{C}^n ; F_1(x) = \cdots = F_s(x) = 0 \} \).

We assume now that the algebraic variety \( V \) is nonempty and equidimensional (this means that all of its irreducible components have the same dimension, say \( r \)). We associate to the equidimensional variety \( V \) a parameter \( \delta \), called the degree of \( V \), which is defined as follows (see e.g. [21]): let \( D \) be the class of all affine linear subspaces of dimension \( \text{codim}(V) := n - r \). Then we define the degree of \( V \) in the following way:

\[
\delta := \sup \# \{ L \cap V_r ; L \in D \text{ and } \# (L \cap V_r) < \infty \}
\]

In [21] it is proved that \( \delta \) never takes the value zero or infinity and that \( \delta \) can be bounded by the minimum of \( d' \) and \( d'' \).
Let us assume that the variables $X_1, \ldots, X_r$ are in generic position with respect to the variety $V$ (this roughly means that the image $\pi(V)$ of $V$ under the projection $\pi : \mathbb{C}^r \to \mathbb{C}^r$ defined by $\pi(x_1, \ldots, x_r) = (x_1, \ldots, x_r)$ has "the same geometric characteristics" as $V$). Then the equidimensional variety $V$ may be represented in a "pseudoparametric" way à la Kronecker [30]. This representation consists of the following items:

- A generic $\mathbb{Q}$-linear form $U := \lambda_1 Y_{r+1} + \cdots + \lambda_n Y_n$.
- The minimal (in the sense of its degree) polynomial $q \in \mathbb{Q}[X_1, \ldots, X_n] [T]$ such that $q(X_1, \ldots, X_n, U)$ vanishes on any point of the variety $V$. This polynomial always exists and its degree is bounded by $\delta$.
- A generic "parametrization" of the variety $V$ by the zeros of $q$. This parametrization is given by polynomials of the form $\rho_{r+1} Y_{r+1} - \lambda_1 T \rho_1 - \cdots - \lambda_n T \rho_n \in \mathbb{Q}[X_1, \ldots, X_n] [T]$ with $\rho_{r+1}, \ldots, \rho_n \in \mathbb{Q}[X_1, \ldots, X_n] [0]$ and $\rho_1, \ldots, \rho_n \in \mathbb{Q}[X_1, \ldots, X_n][T]$. We require that $\max(\deg q, \deg \rho_1, \ldots, \deg \rho_n) < \deg q$ holds and that for $1 \leq j \leq n - r$ the polynomial $\rho_{r+1} X_{r+j} - \lambda_j T \rho_j$ vanishes on any point of $V$.

Let us observe that, for any point $x = (x_1, \ldots, x_n) \in \mathbb{Q}$ with $\prod_{j=1}^n \rho_j (x) \neq 0$ and any $1 \leq j \leq n - r$, the identities $q(x_1, \ldots, x_n, U(x)) = 0$ and $\lambda_j T \rho_j (x_1, \ldots, x_n) U(x) / \rho_j (x_1, \ldots, x_n$) hold. This implies that for a suitable root $\alpha_j$ of the univariate polynomial $q(x_1, \ldots, x_r, T) \in \mathbb{Q}[T]$ the identity

\[
x = \left( x_1, \ldots, x_r, \frac{\lambda_j T \rho_j (x_1, \ldots, x_n)}{\rho_j (x_1, \ldots, x_n)} \right)
\]

holds. This is the reason why we call a representation à la Kronecker of the algebraic variety $V$ "pseudoparametric".

It is well known that many fundamental problems of effective elimination theory require the solution of the following task (see [13,17,23,35]).

Given a geometric solution of the equidimensional variety $V$ and a straight-line program computing a polynomial $F \in \mathbb{Q}[X_1, \ldots, X_n]$, compute the image of the variety $V$ under the projection $\pi_T : \mathbb{C}^{n+1} \to \mathbb{C}^{n+1}$ defined by $\pi_T(x_1, \ldots, x_n, F(x_1, \ldots, x_n))$.

Let us observe that the image $\pi_T (V)$ defines a hypersurface of $\mathbb{C}^{n+1}$ which can be represented by means of its minimal polynomial, i.e. by the polynomial $m_T \in \mathbb{Q}[X_1, \ldots, X_n][T]$ of minimal (total) degree such that $m_T (X_1, \ldots, X_n, F(X_1, \ldots, X_n))$ vanishes on any point $(x_1, \ldots, x_n)$ of the variety $V$. In [23] the computation of the polynomial $m_T$ is reduced to polynomial arithmetic modulo $q$ (where $q$ is considered as a univariate polynomial in the variable $T$ with coefficients in $\mathbb{Q}[X_1, \ldots, X_n]$). More precisely, let us write $X := (X_1, \ldots, X_n)$ and let us consider the polynomial $f(X, T) := F(X, \lambda_1 T / \rho_1, \ldots, \lambda_n T / \rho_n (X, x_n)) \in \mathbb{Q}[X_1, \ldots, X_n][T]$.

In [24, Lemma 8], a description of a procedure which returns a suitable multiple $g(X, T) \in \mathbb{Q}(X_1, \ldots, X_n, T)[T]$ of the polynomial $m_T$ in the univariate polynomial ring $\mathbb{Q}(X_1, \ldots, X_n, T)[T]$ is given. The goal of this section is to exhibit a MÜLLINGA procedure which computes this polynomial. The minimal polynomial $m_T$ is then easily obtained using a problem adapted greatest common divisor procedure. We compute the polynomial $g(X, T)$ as follows.

First we find for each $1 \leq k \leq \delta$ the leading coefficient $s_k \in \mathbb{Q}(X_1, \ldots, X_n)$ of the remainder of the division of the polynomial $(f(X, T) \partial q(X, T) / q(X, T))$ by $q(X, T)$, where $q(X, T)$ denotes the partial derivative of the polynomial $q(X, T)$ with respect to the variable $T$. It follows that for each $1 \leq k \leq \delta$ the rational function $s_k$ equals the $k$th Newton polynomial $T^k + \cdots + T^k$ evaluated in the roots of the polynomial $g(X, T)$ (these roots belong to the algebraic closure of the field $\mathbb{Q}(X_1, \ldots, X_n)$). Therefore, using the Newton formulas (which allows us to compute the coefficients of a polynomial $p(T) := \prod_{i=1}^n (T - \alpha_i)$
from the power sums $s_k := \sum_{i=1}^{\alpha^{k-1}} a_i$, $1 \leq k \leq n$, by a simple recursive algorithm, see e.g. [8]), we obtain

the coefficients of the polynomial $g(X, T)$. Here is the MILOCAL code of this algorithm:

```plaintext
transfPol (p,q:TVec) result r:TVec local (dp,s,t,rest:TVec, i,j:TInt)
{ dp := deriveVec p;
  CRV p;
  s := newV (# vp - 1);
  for i := 0 to # v s - 1 do
  { dp := dp * q;
    if # v dp > # v p - 2 then s[i] := dp[# vp - 2] else s[i] := $0
  };
  // code for Newton formulas
  t := newV (# vp - 1);
  for i := 0 to # v t - 1 do
  { t[i] := s[i];
    for j := 0 to i - 1 do t[i] := t[i] - (s[(i - j) - 1] * t[j]);
    t[i] := t[i]/(i + 1)
  };
  r := ... // code for computing the output r by reversing
  // and changing the sign of the vector t
}
partial(f: TSlp TObj, // A SLP representing the input polynomial F
  vp: TSlp TVec, // A vector of vectors of SLPs representing
  // the coefficients of the input polynomials
  vp: TSlp TVec, // A vector of SLPs representing the coefficients
  // of the input polynomial q with respect to the variable T
  q: TSlp TVec, // A vector of SLPs representing the coefficients
  // of the input polynomial q with respect to the variable T
  rho: TSlp TObj, // A SLP representing the input polynomial rho
  r: TInt)
extras x(r) result res:TObj local (i,j:TInt, vps:TVec)
{ CRV(q(dot i := 0 to r - 1 ap x[i]));
  // Polynomial arithmetic modulo q
  vps := vps(dot i := 0 to r - 1 ap x[i])/rho (dot i := 0 to r - 1 ap x[i]);
  res := f(dot i := 0 to r - 1 ap x[i])
    (dot j := 0 to # v vps - 1 ap vps[j]);
  CRV $0 // Restoring modular arithmetic
}
main result res:TObj local (parcslp:TSlp TVec, w: TVec, m, i,j:TInt)
```
\[
\begin{align*}
&\{ \text{m := askTInt; \hspace{1em} // m = r + 1 = \# free variables} \\
&\text{parcSlp := partial (myF m) (myVP m) (myQ m) (myRho m) m; } \\
&w := \text{newV m; \hspace{1em} // w[0], w[1], ..., w[m - 1] free variables} \\
&\text{for i := 0 to m - 1 do w[i] := askTNum;} \\
&\text{res := (transfPol (myQ m (dot i := 0 to om-1 ap w[i])))} \\
&\text{(parcSlp (dot j := 0 to m - 1 ap w[j])))}
\end{align*}
\]

We apply now this algorithm to a test suite which was generated in the following way. Let \( r \) be a natural number and let \( n = r + 2 \). Let \( V^{(r)} \) be the \((r + 1)\)-dimensional algebraic subvariety of \( \mathbb{C}^{n+1} \) represented à la Kronecker by the polynomials

\[
q^{(r)}(X_0, \ldots, X_r, T) := \prod_{\substack{\varepsilon_0, \ldots, \varepsilon_r \in \{0, 1\} \cap \{0, 1\}}} (T - \varepsilon_0 X_0 - \cdots - \varepsilon_r X_r),
\]

\[
\rho^{(r)}(X_0, \ldots, X_r) := \prod_{\substack{\sigma_0, \ldots, \sigma_r \in \{-1, 0, 1\} \cap \{0, 1\}}} (\sigma_0 X_0 + \cdots + \sigma_r X_r)
\]

and by suitable polynomials \( v^{(r)}_{1,1}, v^{(r)}_{1,2} \) in \( \mathbb{Q}[X_0, \ldots, X_r, T] \) satisfying the condition max\( \{\text{deg}(v^{(r)}_{1,1}), \text{deg}(v^{(r)}_{1,2})\} < \text{deg}_{T} q \). Let us choose an arbitrary “projection” polynomial \( F^{(r)} \in \mathbb{Q}[X_0, \ldots, X_r] \). We observe that the algebraic variety \( V^{(r)} \), parametrized by the polynomials \( q^{(r)}, \rho^{(r)}, v^{(r)}_{1,1} \) and \( v^{(r)}_{1,2} \), has dimension \( r + 1 \) and degree \( 2^{r+1} \). Therefore, the output polynomial \( g^{(r)} \) is an element of \( \mathbb{Q}[X_0, \ldots, X_r, T] \) of degree at least \( 2^{r+1} \). Let us also remark that the polynomial \( \rho^{(r)} \) represents a square-free representation of the discriminant of the polynomial \( q^{(r)} \) with respect to the variable \( T \). The following is the MILONGA code for this test suite:

\[
\begin{align*}
\text{myF (m:TInt) extras x(m + 2) \hspace{1em} // F = F[X0, \ldots, Xr, Xr + 1, Xr + 2]} \\
\text{result res:TObj \hspace{1em} // m = r + 1} \\
\{ \\
\text{res := x[0] + x[1] \ast x[m] \ast x[m] \ast x[m + 1] \ast x[m + 1]} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{myVP (m:TInt) extras x(m) \hspace{1em} // m = r + 1} \\
\text{result vp:TObj} \\
\{ \\
\text{vp := newV 2;} \\
\text{vp[0] := newV 2;} \\
\text{vp[0][0] := x[0] + x[1];} \\
\text{vp[0][1] := x[0];} \\
\text{vp[1] := newV 2;} \\
\text{vp[1][0] := x[0] \ast x[0];} \\
\text{vp[1][1] := x[1] \ast x[1]} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{myQ (m:TInt) extras x(m) result res:TVec}
\end{align*}
\]

\[
\begin{align*}
&\text{for i := 0 to m - 1 do w[i] := askTNum; } \\
&\text{res := (transfPol (myQ m (dot i := 0 to om-1 ap w[i])))} \\
&\text{(parcSlp (dot j := 0 to m - 1 ap w[j])))}
\end{align*}
\]
This MILONGA code was tested for the parameter $r$ varying from 1 to 4. In order to visualize the complexity behaviour of the code, each of the resulting polynomials $g'(r) \in \mathbb{Q}[X_0, \ldots, X_r, T]$ was evaluated in a "generic", i.e. in a random test point belonging to the hypercube $[1, 100]^{r+1} \cap \mathbb{Z}^{r+1}$. Here are the corresponding timings:

<table>
<thead>
<tr>
<th>$r$</th>
<th>MILONGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.79</td>
</tr>
<tr>
<td>2</td>
<td>17.21</td>
</tr>
<tr>
<td>3</td>
<td>95.75</td>
</tr>
<tr>
<td>4</td>
<td>35041.98*</td>
</tr>
</tbody>
</table>

The value 35041.98 was obtained using a heap size of 10MB.

Analyzing this example one observes the strong dependence of the timings on the length of the straight-line program generated by the given MILONGA code for the representation of the polynomial $g'(r)$. In fact, the length of this straight-line program is $O(2^r)$. This explains the jump of the timing between $1 \leq r \leq 3$ and $r = 4$.

6. Conclusions

By the example of the MILONGA language we have tried to demonstrate that the implementation of algorithms based on nonstandard object representation may profit enormously from the expressive power of the functional programming paradigm.

We have illustrated this generic statement by an ad hoc implementation of a short list of procedures which play a central rôle in a new generation of highly efficient (and involved) elimination algorithms for multivariate polynomial equation solving. These algorithms are based on a nonstandard object representation, namely the straight-line program representation of polynomials.
At the actual stage of our theoretical and practical research on effective polynomial equation solving it would have been too ambitious to implement a complete software package for such an elimination algorithm in a functional programming (e.g. the MILONGA) language, since such a task requires the implementation of a large library of highly efficient basic subroutines. Nevertheless, this paper shows that such an implementation is realizable if the necessary programming effort is made.

On the other hand, a theoretical complexity analysis of the execution time of MILONGA programs, the benchmarks of the list of procedures implemented in MILONGA and their comparison with standard symbolic software (Maple) give evidence for the advantages which can be expected from a careful programming of such an elimination algorithm.

Acknowledgements

The authors thank the anonymous referee for his many pertinent comments, which contributed to considerable improvements in the presentation of this paper. R. Wachenchauzer and G. Matera thank the Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires and the Departamento de Computación, Universidad Favaloro, Buenos Aires, Where they did part of their work. This research was partially supported by the following Argentinian, Spanish and French grants: UBACyT UBA X-198, PIP CONICET 4571, UNLP X-272, DGCYT BFM 2000-0349, Acción Integrada Hispano–Argentina Universidad de Cantabria–Universidad de Buenos Aires, ECOS A99E06 and HF-1999-055.

References


