# Some issues concerning Computer Algebra in AToM$^3$

Indrani A.V.

April 11, 2003

This document describes some issues which must be considered while adding Computer Algebra capabilities to AToM$^3$.

## 1 Introduction

We would like to do the following in AToM$^3$: to take sets of equations - these could be just algebraic, or differential algebraic equations (DAEs), and manipulate them symbolically before invoking a solver. These manipulations include: first doing dependency analysis on the equations and variables, followed by causality assignment so that it is clear what equation must be used to solve for which variable. Next, cycle detection and sorting is performed, so that the order in which the equations must be solved is determined, and sets of coupled equations are identified. These manipulations should together be implemented as a 'preliminary transformation' in AToM$^3$.

To do this we must first of all be able to create meta-models of sets of algebraic expressions and equations, and then the objects that appear in these expressions. Typically algebraic expressions are made up of *numerical constants* (numbers) such as $9, 2.3$; variables representing unknowns, which are denoted by symbols such as $x, y$; and arithmetical operators such $+, *$. In addition, in the simulation context there are known quantities or *parameters*, which are denoted by symbols (such as an elastic constant $\eta$, a refractive index $\mu$), which are given certain values for a particular simulation. We could also extend the 'basic entitites' we deal with to include vectors, matrices, intervals, and so on; the set of operators to include non-arithmetical operators such as logical, Boolean, differential, and possibly other operators. There are universal constants such as $\pi, e$, the speed of light $c$, which are denoted by symbols but have a fixed value, which should also be taken into account.

Possible entities in the meta-model for algebraic expressions and their constituent relationships could be (the symbol $\Downarrow$ indicates the relation 'contains'):

Sets of Equations
$\Downarrow$
Equations
$\Downarrow$
Expressions
$\Downarrow$
Terms
$\Downarrow\Downarrow\Downarrow\Downarrow$
Constants, parameters, variables, operators.

It is helpful at this juncture to look at how Computer Algebra Software (CAS) systems like Mathematica, Maple and MuPAD work. MuPAD ([Fucssteiner et al., 1996]) is freely available for researchers, and the source code is also included in the distribution. The recent versions of MuPAD work in an object-oriented way. The most important objects in MuPAD are *domains*, which are used to define abstract algebraic structures such as rings and fields, and also the *'basic type'* of an object. There are fundamental domains such as the domain **Z** of integers, the field **R** of real numbers, etc. Other domains are constructed which inherit properties of the basic domains - thus there is a hierarchy of domains. A domain is constructed using a set of pre-defined *axioms* and *constructors*. Example domains are the domain of expressions, the

domain of polynomials, the domain of matrices, etc. The operations within a domain are interpreted in its own context. Thus for instance, the action of the operator '+' (or '_plus') depends on the objects being added. Domains with common features are further grouped into *categories*, which are in turn constructed using category constructors and axioms.

The various CAS systems differ in the way the basic objects are represented and stored. A major concern in all of these is the important problem of *simplification* and *evaluation*. Simplification involves manipulation and *comparison* of expressions, at one level. The only way to compare expressions is to 'reduce them to a standard unique form', or *canonical form*. We discuss these ideas further in subsequent sections.

I have succeeded in taking a small set of algebraic equations, doing dependency analysis and causality assignment, and sorting them (using the built-in network flow algorithms) in MuPAD. (See Appendix 1 for the details.) We could in principle use MuPAD to do all these operations instead of AToM$^3$. However, we then do not have a handle on the simplification. Although the source code for all the methods used in MuPAD is available, basic things such as the canonical representation used, are not transparent, because they are part of the system kernel. (Note that future versions of MuPAD are planned to have 'modular' simplification). Also, the network flow algorithms used are not the most efficient ones. Of course, one could write better algorithms using the MuPAD language itself.

Another important question for us is: what level of description do we use to model algebraic quantities ? Do we start from the axiomatic number theory level, or just model basic objects and operators without worrying about their algebraic structure ?

Here are some more ideas leading to the question of simplification and canonical representation. These are taken from [Geddes et al., 1992].

## 2 Levels of Abstraction

While we mentioned above that integers, polynomials, etc. are basic objects in their respective domains with operations such as $\{+,*\}$ suitably defined, these mathematical, abstract definitions cannot be implemented in a simple manner. The data structures required to represent polynomials or series are far more complicated than those for integers, for example. Also, at the machine level the algorithms used to implement ring operations depend on the ring elements being considered, and so vary in complexity. We see that there are different levels of abstraction in dealing with abstract mathematical objects. In [Geddes et al., 1992] three levels of abstraction are identified:

1. The *object* level: this is the abstract mathematical level where the elements of a domain are considered to be primitive objects.

2. The *form* level: this is intermediate between the abstract and the data structure levels. Here we are concerned with how to represent (or express) an object symbolically, keeping in mind that this representation may not be unique. For example, there are different, but equivalent, representations for the same bivariate polynomial in the domain $\mathbf{Z}[x,y]$ (bivariate polynomials with integer coefficients):

$$\begin{aligned} a(x,y) &= 10x^2y - 2xy + 5x - 1; \\ a(x,y) &= (5x-1)(2xy+1); \\ a(x,y) &= (10y)x^2 + (-2y+5)x - 1. \end{aligned} \tag{1}$$

   In the first of these, $a(x,y)$ is in the *distributed form*, that is, $x$ and $y$ are treated at the same level; it is also in *expanded form*, without factors. In the second, $a(x,y)$ is left in *factored form*. In the last form, $a(x,y)$ is expressed in a *recursive way* - that is, it is considered to be a polynomial in $x$ with coefficients which are polynomials in $y$. More on this below.

3. The *data structure* level: here we are concerned with what data structures to use to represent particular objects. For example the polynomial $a(x,y)$ in the first form in (1) above can be represented by a linked list consisting of four elements (one for each term), or we can store the coefficients and exponents in separate arrays.

The important questions of simplification and evaluation are related to the form level of abstraction.

## 3 The Simplification Problem

Which is the *simplest* form of $a(x,y)$ in (1) ? The answer depends on what we mean by 'simplest form' for a given expression. This is not obvious, however, and it is a very difficult problem to specify simplification algorithmically. Consider polynomials again. For instance, suppose we encounter:

$$(x^2+3x+2)-(x+1)(x+2). \tag{2}$$

We would like the above expression to be replaced by 0. Suppose we specify that to simplify a polynomial expression, we first multiply out all factors term by term (that is, expand all factors) and collect like terms. Then we ensure that the expression in (2) will be automatically replaced by 0. Also, an expression such as:

$$x^n - y^n, \tag{3}$$

for $n$ large, is definitely 'simpler' in this expanded form than in its factored form where $(x-y)$ has been factored out. However, if we have the expression:

$$(x+y)^{1000} - y^{1000}, \tag{4}$$

the expanded form would contain a thousand terms. From both human and resource points of view, the expression in (4) would be considered 'simpler' as it is, unexpanded.

Other examples of simplification include reducing rational expressions to their irreducible form; taking account of trigonometric identities (so that $sin^2(x)+cos^2(x)$ would be replaced by 1, for example); factorization, etc. Thus simplification is essential for the following reasons:

1. large amounts of computer resources (both memory and execution time) may be consumed simply storing and manipulating unsimplified expressions;
2. for better readability, expressions must be in their simplest possible form.

The example in (2) leads us to the *zero equivalence problem*.

## 4 Zero Equivalence

The zero equivalence problem is a special case of the general simplification problem. Here we are interested in recognizing when an expression is equivalent to zero. This case is treated specially because it is a well-defined problem, and also because an algorithm for determining zero equivalence is considered to be a sufficient 'simplification' algorithm in many practical cases. It is not easy however. For example, if we have to simplify:

$$\ln(\tan(\frac{x}{2})+\sec(\frac{x}{2}))-\sinh^{-1}(\frac{\sin(x)}{(1+cos(x)}), \qquad -1\le x\le 1; \tag{5}$$

this can be recognized to be zero only after non-trivial transformations. At this level of generality, the zero equivalence problem is known to be unsolvable by any algorithm in a 'sufficiently rich' class of expressions. However it can be done for the simpler cases of polynomials, rational expressions and power series.

## 5 Normal and Canonical Transformations

In the general case of simplification, and in order to compare expressions, it is necessary to impose an ordering on the terms. We are then led to consider *normal* and *canonical* representations for expressions. A *canonical transformation* renders every expression in a form that is unique. When all expressions are expressed in a canonical form, comparison and simplification become easier.

Formally, the simplification problem can be expressed as follows. Let $E$ be a set of expressions. Let $\sim$ be an equivalence relation on $E$. Then $\sim$ partitions $E$ into equivalence classes and the quotient set $E/\sim$ denotes the set of all equivalence classes. The simplification problem is then stated by specifying a transformation $f : E \to E$ such that for any expression $a \in E$, the transformed expression $f(a)$ belongs to the same equivalence class as $a$ in the quotient set $E/\sim$. We desire that $f(a)$ be simpler than $a$. We obviously choose equality $(=)$ as the equivalence relation. We note that all the expressions for $a(x)$ in (1) are *equal* to each other (and hence belong to the same equivalence class), but are not *identical* $(\equiv)$ *in form*.

A *normal function* for $[E;\sim]$ is a computable function $f : E \to E$ and satisfies the following properties:

1. $f(a) \sim a$ for all $a \in E$;
2. $a \sim 0 \Rightarrow f(a) \equiv f(0)$ for all $a \in E$.

A *canonical function* for $[E;\sim]$ is a normal function $f : E \to E$ which satisfies the additional property of uniqueness:

3. $a \sim b \Rightarrow f(a) \equiv f(b)$ for all $a, b \in E$.

If $f$ is a normal function for $[E;\sim]$ then an expression $\tilde{a} \in E$ is said to be a *normal form* if $f(\tilde{a}) \equiv \tilde{a}$. If $f$ is a canonical function for $[E;\sim]$ then an expression $\tilde{a} \in E$ is a *canonical form* if $f(\tilde{a}) \equiv \tilde{a}$.

Thus if we are able to define a normal function in a set $E$ of expressions, the zero equivalence problem is solved. There is no unique normal form for all expressions in a particular equivalence class unless this class contains 0. A canonical form however provides a unique representative for each equivalence class. This is proved by the following theorem:

**Theorem:** If $f$ is a canonical function for $[E;\sim]$ then the following properties hold:

1. $f$ is idempotent: $f \circ f = f$, where $\circ$ denotes composition of functions;
2. $f(a) \equiv f(b)$ iff $a \sim b$;
3. in each equivalence class in $E/\sim$ there exists a unique canonical form.

We will not prove this here however.

Normal and canonical forms can be constructed for polynomials, rational expressions and power series fairly easily. Once the canonical form or canonical representation has been determined, appropriate data structures have to be constructed.

## 6 Evaluation

*Evaluation* is closely intertwined with simplification. In all CAS systems, evaluation can be controlled by the user. For example, if we have:

$$\begin{aligned} y &= 3 \\ x &= y+5 \\ z &= x^3-5x+1; \end{aligned} \tag{6}$$

should **eval(z)** lead to the 'simplified' expression in *y*, or should the value of *y* be further substituted ? Obviously in a very complex expression, the result of evaluation depends on the substitution depth of its variables.

## 7 Representation, normal and canonical forms for polynomials

For multivariate polynomials, there are different choices to be made for their representation, at the form level of abstraction: whether to use a *recursive* or *distributive* representation, and whether to use a *sparse* or *dense* respresentation.

### 7.1 Recursive and distributive representations

In the recursive representation, a polynomial in several variables $a(\mathbf{x}) = a(x_1, x_2, \ldots, x_m)$ (in the domain $D[x_1, x_2, \ldots, x_m]$ of polynomials of $m$ variables ) is represented as:

$$a(x_1, x_2, \ldots, x_m) = \sum_{i=0}^{\deg_1(a(\mathbf{x}))} a_i(x_2, \ldots, x_m)\, x_1^i. \tag{7}$$

Here $\deg_1(a(\mathbf{x}))$ is the degree of $a(\mathbf{x})$ as a polynomial in $x_1$. That is, $a(\mathbf{x})$ is written as an element of $D[x_2, \ldots x_m][x_1]$, where, recursively, the polynomial coefficients $a_i(x_2, \ldots, x_m)$ are represented as elements of the domain $D[x_3, \ldots, x_m][x_2]$, and so on. Finally, the polynomial is viewed as an element of the domain $D[x_m][x_{m-1}] \ldots [x_1]$. An example of a polynomial in three variables with integer coefficients written recursively is:

$$a(x,y,z) = (3y^2 + (-2z^3)y + 5z^2)\, x^2 + 4x + ((-6z+1)y^3 + 3y^2 + (z^4+1)). \tag{8}$$

In the distributive representation, however, the polynomial $a(\mathbf{x})$ (in the domain of multivariate polynomials $D[\mathbf{x}]$) is represented as:

$$a(\mathbf{x}) = 3x^2y^2 - 2x^2yz^3 + 5x^2z^2 + 4x - 6y^3z + y^3 + 3y^2 + z^4 + 1. \tag{9}$$

### 7.2 Sparse and dense representaions

In the sparse representation of polynomials, only the terms with non-zero coefficients are represented, while in the *dense* representation, each and every term that can possibly appear in a polynomial of a given degree is represented (even if some coefficients are zero). This depends to some extent on the data structures used. The dense representation is unwieldy for large degrees, however. Most CAS systems use the sparse representation.

Another choice is whether to write constant terms as terms with exponents all equal to zero.

### 7.3 Normal and canonical forms

Normal and canonical forms can be defined for polynomials in their *expanded* or *factored* forms.

An *expanded normal form* for multivariate polynomials in a domain $D[x_1, x_2, \ldots x_m]$ can be specified by the following normal function $f_1$, whose rules are:

1. multiply out all products of polynomials;
2. collect terms of the same degree.

We can arrive at an *expanded canonical form* using a canonical function $f_2$: apply $f_1$ above; then:
3. rearrange the terms into some particular order (ascending or descending) of their degrees.

A *factored normal form* and a *factored canonical form* for polynomial expressions can also be defined by specifying corresponding functions. (See [Geddes et al., 1992] for details).

However, the functions $\{f_1, f_2\}$ and those for the factored forms, are not all easy to implement practically. Especially the factored canonical form, is rarely used because polynomial factorization is relatively expensive. Usually variants of $f_2$, the expanded canonical form, and the factored normal form, are used, with some control over the 'simplification' of the resulting expressions. Thus in practical implementations, the transformation functions which are most convenient may be neither normal nor canonical !

## 8 Normal and canonical forms for rational expressions

A field of rational expressions $D(x_1, x_2, \ldots x_m)$ can be considered to be the quotient field of a polynomial domain $D[x_1, x_2, \ldots x_m]$. Therefore normal and canonical forms for rational expressions depend on the polynomial forms used.

If we assume that GCDs exist in the domain $D$ of rational expressions, and assuming the expanded canonical form for polynomials, an *expanded canonical form* for rational expressions is defined by $f_3$:

1. (form common denominator): put the expression into the form $a/b$ where $(a,b)$ are polynomials in $D[x_1,x_2,\ldots x_m]$;
2. (remove GCD): compute g = GCD(a,b), which is also a polynomial in $D[x_1,x_2,\ldots x_m]$. Replace the expression $a/b$ by $a'/b'$ where $a = a'g$ and $b = b'g$;
3. (normalize): replace the expression $a'/b'$ by $a''/b''$ where the denominator $b''$ is the 'normalized' form of $b'$ - that is, with the leading coefficient positive.
4. (make polynomials canonical): replace the expression $a''/b''$ by $f_2(a'')/f_2(b'')$ where $f_2$ is the expanded canonical form for polynomials defined above.

In $f_3$ above, it is not explicitly stated if the numerator and denominator polynomials $(a,b)$ must be first put into some normal or canonical form before the GCD is found. As a practical point, if $(a,b)$ are in factored normal form, the GCDs of the various factors can be computed separately and multiplied.

There are other normal forms possible for rational expressions, which are more useful in a practical implementation. Some of these are:

- *factored/factored*: the numerator and the denominator are both in factored normal forms;
- *factored/expanded*: the numerator is in factored normal form and the denominator is in expanded canonical form;
- *expanded/factored*: the numerator is in expanded canonical form and the denominator is in factored normal form.

It can be seen that the canonical form $f_3$ would be the *expanded/expanded* form. The *expanded/factored* form has been found to be particularly useful, along with an efficient computation of GCDs. Again, however, sometimes other simplification procedures may have to be employed in a practical situation, so that the representation corresponds to neither the canonical nor normal forms.

# 9 Computer Algebra in AToM$^3$

In the sections above we have reviewed some important issues in CAS systems. The idea of transforming an algebraic expression into a unique, canonical form, is an important one, and it is noteworthy that different algebraic objects need different normal and canonical representations.

We would like to be able to symbolically manipulate expressions and equations in AToM$^3$. However we certainly should not aim to incorporate all the capabilities in available CAS systems. We identify the following transformations that we should be able to do in AToM$^3$:

- A 'canonicalize' transformation should be able to render any expressions or equations (algebraic, DAEs) into their unique, canonical form. This implies collecting terms, rearranging them in a particular order, and performing constant folding.
- Given a set of equations to solve, a 'preliminary solver' must do the following:
  1. dependency analysis;
  2. causality assignment;
  3. cycle detection;
  4. sorting.

  We must be able to handle both purely algebraic objects and DAEs as well. (Note: The meaning of all the above operations w.r.t DAEs is to be further investigated).

In order to implement all of the above in AToM$^3$, we need to resolve some things:

- The canonical representation to be used to represent algebraic objects. From the discussion in the preceding sections, we must in principle define different canonical forms for different objects, such as polynomials and rational expressions. However, rather than following the standard procedure, it may be best to choose a *convenient* and *practical* representation to begin with, wherein simplification will be performed to a limited extent. One such possible set of rules for purely algebraic expressions and equations is presented in the following section.
- The appropriate data structures and algorithms to be used to manipulate different algebraic objects.

## References


[Davenport et al., 1993] Davenport, J., Siret, Y., and Tournier, E. (1993). *Computer Algebra, Systems and Algorithms for Algebraic Computation*. Academic Press.

[Fucssteiner et al., 1996] Fucssteiner, B. et al. (1996). *MuPAD User's Manual; Multi-Processing Algebra Data Tool*. Wiley. (`http://www.mupad.de`).

[Geddes et al., 1992] Geddes, K. O., Czapor, S. R., and Labahn, G. (1992). *Algorithms for Computer Algebra*. Kluwer Academic Publishers.

[Shi and Steeb, 1998] Shi, T. K. and Steeb, W.-H. (1998). *Symbolic C++, An Introduction to Computer Algebra Using Object-Oriented Programming*. Springer-Verlag.

[Wester, 1999] Wester, M. J., editor (1999). *Computer Algebra Systems: A Practical Guide*. John Wiley and Sons Ltd., New York, first edition.

# Appendix 1

```
* Here is an example from MuPAD. This text file was generated from a
* MuPAD session. In this session, I have taken a set of four
* equations, performed dependency analysis and  causality assignment
* by finding a maximal flow on a bipartite graph defined over the equation numbers and the
* unknown variables. After we know which equation is to be used to
* solve for what variable, the equations are sorted in order to find
* the correct order of solution.
-----------------------------------------------------------------

* Here are the four equations:
-----------------------------

>>  eq1 := x+y+z =0;

                                 x + y + z = 0
>> eq2 := x+3*z+u*u =0;

                                          2
                               x + 3 z + u  = 0
>> eq3:= z -u -16 =0;

                                z - u - 16 = 0
>>  eq4 := u -5 = 0;

                                   u - 5 = 0
>> eqlist:= [eq1,eq2,eq3,eq4]

                                    2
       [x + y + z = 0, x + 3 z + u  = 0, z - u - 16 = 0, u - 5 = 0]


* Here we define a procedure which collects all the variables in a given
* equation (or expression) by looking at the type of the operands of
* the expression. If the operand is of type IDENTIFIER, it is added to
* the set of variables, var_set.

>> collect_vars := proc(expression) begin operands := op(expression):
if expression = operands then if type(expression) = DOM_IDENT
then var_set := var_set union {expression} end_if
else map(operands,collect_vars) end_if end_proc;

                   proc collect_vars(expression) ... end

* Below we initialize two sets: total_var_set is the union of all the
*  variables in all the equations, each_var_list is a list which
* contains the set of variables for each equation, where the position
* of each set in the list corresponds to the equation number.

>>  total_var_set := {}

                                      {}
>>  each_var_list := []

                                      []
* Building the two objects....

>> for elem in eqlist do var_set :={}: collect_vars(elem):
total_var_set:= total_var_set union var_set: each_var_list:= append(each_var_list,var_set):end_for:

>>  each_var_list
```

```
                    [{x, y, z}, {u, x, z}, {u, z}, {u}]
>>  total_var_set

                                {u, x, y, z}


* Now we need to define the directed (bipartite) graph. This is done by forming a graph whose
* vertices are the equations, together with all the variables. To use the Networks package of
* MuPAD, we need to supply a list of vertices and a list of edges.
* I have chosen [1,2,3,4] to refer to the members of
* eqlist = [eq1,eq2,eq3,eq4] above, and the other vertices are given by the set of all the
* variables, total_var_set. The edges are defined as follows.
* There is an edge from the equation number to a variable whenever the variable occurs in
* the  equation. Note that we could have also directly used the equations themselves,
* or their labels such as 'eq1'.

* Generating the vertices corresponding to the equations:

>>  conlist:= [i $i = 1..nops(eqlist)]

                                [1, 2, 3, 4]

* Making a list of all the variables:


>> varlist:= [op(total_var_set)]

                                [y, u, x, z]

* Building the vertex list:

>> vertex_list := conlist.varlist

                          [1, 2, 3, 4, y, u, x, z]

* Initializing the list of edges:

>> edge_list := []

                                    []

* Building the edge list:


>> for mem in each_var_list do for elem in op(mem) do edge_list :=
append(edge_list, [contains(each_var_list,mem), elem]): end_for: end_for:

* This is what edge_list looks like:

>> edge_list

 [[1, z], [1, x], [1, y], [2, u], [2, x], [2, z], [3, u], [3, z], [4, u]]



* We need to define a list of vertex weights. This particular list of
* values is chosen so that the graph can be converted to a
* single-source single-sink network. More on this below.

* Building the list of vertex weights:

>> vw := [1,1,1,1,-1,-1,-1,-1]
```

```
                        [1, 1, 1, 1, -1, -1, -1, -1]

* Creating our directed graph G by invoking the Networks
* package. Since we have not assigned edge weights and edge
* capacities, these are all set to the default value 1 :

>> G:= Network(vertex_list, edge_list, Vweight = vw);

                                  Network()

* Displaying the details of the graph G:

>>  Network::printGraph(G):

                    Vertices: [1, 2, 3, 4, y, u, x, z]

Edges: [[1, z], [1, x], [1, y], [2, u], [2, x], [2, z], [3, u], [3, z], [4\
, u]]

        Vertex weights: table(z=-1,x=-1,u=-1,y=-1,4=1,3=1,2=1,1=1)

Edge capacities: table([4, u]=1,[3, z]=1,[3, u]=1,[2, z]=1,[2, x]=1,[2, u]\
=1,[1, y]=1,[1, x]=1,[1, z]=1)

Edge weights: table([4, u]=1,[3, z]=1,[3, u]=1,[2, z]=1,[2, x]=1,[2, u]=1,\
[1, y]=1,[1, x]=1,[1, z]=1)

Adjacency list (out): table(z=[],x=[],u=[],y=[],4=[u],3=[u, z],2=[u, x, z]\
,1=[z, x, y])

Adjacency list (in): table(z=[1, 2, 3],x=[1, 2],u=[2, 3, 4],y=[1],4=[],3=[\
],2=[],1=[])

* After defining the directed graph, we convert it into a
* single-source single-sink network by adding the source node 'q' and
* sink node 's' to the graph G. The call:
*       Network::convertSSQ(G,q,s)
* converts a graph G into a single-source
* single-sink network by adding the nodes 'q' and 's'. These are
* connected to the other nodes (or vertices) in the network as follows:
*     A new edge [q,i] is added for every vertex i with a positive weight.
*     A new edge [i,s] is added for every vertex i with a negative weight.

* We obtain G1, the single-source single-sink network formed from G:

>>  G1:= Network::convertSSQ(G,q,s);

                                  Network()

* Showing details of G1:

>> Network::printGraph(G1)

                 Vertices: [1, 2, 3, 4, y, u, x, z, q, s]

Edges: [[1, z], [1, x], [1, y], [2, u], [2, x], [2, z], [3, u], [3, z], [4\
, u], [q, 1], [q, 2], [q, 3], [q, 4], [y, s], [u, s], [x, s], [z, s]]

      Vertex weights: table(s=-4,q=4,z=0,x=0,u=0,y=0,4=0,3=0,2=0,1=0)

Edge capacities: table([3, z]=1,[2, z]=1,[1, z]=1,[2, x]=1,[1, y]=1,[4, u]\
=1,[1, x]=1,[3, u]=1,[2, u]=1,[q, 4]=1,[q, 3]=1,[q, 2]=1,[q, 1]=1,[z, s]=1\
,[y, s]=1,[x, s]=1,[u, s]=1)
```

```
Edge weights: table([3, z]=1,[2, z]=1,[1, z]=1,[2, x]=1,[1, y]=1,[4, u]=1,\
[1, x]=1,[3, u]=1,[2, u]=1,[q, 4]=0,[q, 3]=0,[q, 2]=0,[q, 1]=0,[z, s]=0,[y\
, s]=0,[x, s]=0,[u, s]=0)

Adjacency list (out): table(s=[],q=[1, 2, 3, 4],z=[s],x=[s],u=[s],y=[s],4=\
[u],3=[u, z],2=[u, x, z],1=[z, x, y])

Adjacency list (in): table(s=[y, u, x, z],q=[],z=[1, 2, 3],x=[1, 2],u=[2, \
3, 4],y=[1],4=[q],3=[q],2=[q],1=[q])

*  We now compute the maximal flow f from the source to the sink in the
*  graph G1, with the default edge capacity values set to 1 for all
*  edges. The result is a sequence containing the flow value (the
*  inflow at s which equals the outflow at q), and the flow value
*  which is in the form of a table, t, where the flow from node i to
*  node j is given by t[[i,j]].
*  The algorithm used by MuPAD is the preflow-push algorithm of
*  Goldberg and Tarjan (' A new approach to the maximum-flow problem',
*  Journal of the ACM, 35 (4), 1988), with the FIFO selection strategy
*  and an exact distance labelling. The running time is O(n^3), where
*  n is the number of vertices in the network.
*  Note : Here is a limitation of using MuPAD, since the above
*  algorithm is not very efficient. It has been shown that
*  Dinic's algorithm for maximal flow is the most efficient.


>>  f:= Network::maxFlow(G1,q,s)

                                table(
                                  [3, z] = 1,
                                  [2, z] = 0,
                                  [1, z] = 0,
                                  [2, x] = 1,
                                  [1, y] = 1,
                                  [4, u] = 1,
                                  [1, x] = 0,
                            4,    [3, u] = 0,
                                  [2, u] = 0,
                                  [q, 4] = 1,
                                  [q, 3] = 1,
                                  [q, 2] = 1,
                                  [q, 1] = 1,
                                  [z, s] = 1,
                                  [y, s] = 1,
                                  [x, s] = 1,
                                  [u, s] = 1
                                )



* Extracting the table of flows:
```

```
>>  table1:=op(f,2)
```

```
table(
  [3, z] = 1,
  [2, z] = 0,
  [1, z] = 0,
  [2, x] = 1,
  [1, y] = 1,
  [4, u] = 1,
  [1, x] = 0,
  [3, u] = 0,
  [2, u] = 0,
  [q, 4] = 1,
  [q, 3] = 1,
  [q, 2] = 1,
  [q, 1] = 1,
  [z, s] = 1,
  [y, s] = 1,
  [x, s] = 1,
  [u, s] = 1
)
```

```
* We are interested in the table entries which have non-zero flow
* values, and which are not flows to or from the source or sink. The
* edges which correspond to this are those which 'optimally match'
* equations and variables. That is, we have now assigned causality, by
* determining which equation must be used to solve for a particular variable:

>> table2 :=  select(table1, _not@has, {0,q,s})
```

```
table(
  [3, z] = 1,
  [2, x] = 1,
  [1, y] = 1,
  [4, u] = 1
)
```

```
* From the table above, we see that eq3 must be used to solve for the
* variable z, eq2 for x, eq1 for y and eq4 for u. We next have to sort
* these equations in the order that they must be solved. This is done
* by building a 'dependency graph' as follows:  we take the edges
* identified above which assign causality, and reverse them. We then
* replace the four original edges in the graph G by their reversed
* versions, yielding the dependency graph G_dep:

* forming a list of the edges which assign causality:

>> causal_edge_list := []
```

```
[]
```

```
>> for elem in op(table2) do causal_edge_list := append(causal_edge_list,lhs(elem)): end_for:

>> causal_edge_list
```

```
[[3, z], [2, x], [1, y], [4, u]]
```

```
* We create a list of their reverses:

>> reverse_edge_list := []
```

```
                                    []

>> for elem in causal_edge_list do reverse_edge_list := append(reverse_edge_list, revert(elem)):end_for:

* We have the edges reversed:

>> reverse_edge_list

                     [[z, 3], [x, 2], [y, 1], [u, 4]]

* The original edge list:

>> edge_list

 [[1, z], [1, x], [1, y], [2, u], [2, x], [2, z], [3, u], [3, z], [4, u]]


* We build an intermediate list which consists of all the edges in the
* original graph,  but excluding the edges in the causal list.

>> temp_edge_list := []

>> temp_edge_list := select(edge_list,  _not@has, causal_edge_list):

>> temp_edge_list

                 [[1, z], [1, x], [2, u], [2, z], [3, u]]

* We build the new edge list by combining the intermediate list and
* the list of reversed edges:

>> new_edge_list := temp_edge_list. reverse_edge_list

 [[1, z], [1, x], [2, u], [2, z], [3, u], [z, 3], [x, 2], [y, 1], [u, 4]]


* Building the dependency graph:

>> G_dep := Network(vertex_list, new_edge_list)

                                  Network()

* Showing details of the dependency graph:

>> Network::printGraph(G_dep)

                    Vertices: [1, 2, 3, 4, y, u, x, z]

Edges: [[1, z], [1, x], [2, u], [2, z], [3, u], [z, 3], [x, 2], [y, 1], [u\
, 4]]
          Vertex weights: table(z=0,x=0,u=0,y=0,4=0,3=0,2=0,1=0)

Edge capacities: table([u, 4]=1,[y, 1]=1,[x, 2]=1,[z, 3]=1,[3, u]=1,[2, z]\
=1,[2, u]=1,[1, x]=1,[1, z]=1)

Edge weights: table([u, 4]=1,[y, 1]=1,[x, 2]=1,[z, 3]=1,[3, u]=1,[2, z]=1,\
[2, u]=1,[1, x]=1,[1, z]=1)

Adjacency list (out): table(z=[3],x=[2],u=[4],y=[1],4=[],3=[u],2=[u, z],1=\
[z, x])

Adjacency list (in): table(z=[1, 2],x=[1],u=[2, 3],y=[],4=[u],3=[z],2=[x],\
1=[y])
```

```
* Finally we perform a topological sort on the dependency graph. The
* call Network::topSort(H) computes a topological sorting of the graph
* G. That is, we obtain a numbering N of the nodes such that
* N(i) < N(j) whenever there is an edge [i,j] in the graph. The result
* is a table of nodes and their numbering. This ordering is not unique
* however. The routine returns an error when there is a cycle.

* The table whose entries give a sorted order of the nodes:

>> table_sorted := Network::topSort(G_dep)

                                  table(
                                    4 = 8,
                                    u = 7,
                                    3 = 6,
                                    z = 5,
                                    2 = 4,
                                    x = 3,
                                    1 = 2,
                                    y = 1
                                  )

* If we write out the sorted nodes in their ascending order (that is,
* the LHSs of the table entries), replacing the equation numbers by
* their labels, we get the sequence:
*        y eq1 x eq 2 z eq3 u eq4.
* We read this as saying : solve y from eq1, then x from eq2, z from
* eq3, u from eq4. In other words, we must solve the equations in the
* sequence (eq1, eq2, eq3, eq4).
*
* However, from examining the equations themselves it is obvious that
* we need the equations to be sorted in exactly the opposite way,
* in the order (eq4, eq3, eq2, eq1). We have run up against the
* limitations of MuPAD, and the fact that the topological sorting
* algorithm is not unique.
* Another limitation is that cycles, if they exist, are not detected,
* but MuPAD just raises an error and aborts.
*
* We would like to be able to identify sets of equations which form cycles
* in the dependency graph (and which then have to be solved
* separately), and be able to sort the remaining equations. One
* argument is that we must detect cycles first of all, before
* embarking on causality assignment and the rest.


* Just for the sake of completion, I obtained the correct sorted order
* by simply reversing all the edges in the dependency graph:


* Forming the edge list with all the edges reversed in G_dep:

>> rev_list := map(new_edge_list, revert)

 [[z, 1], [x, 1], [u, 2], [z, 2], [u, 3], [3, z], [2, x], [1, y], [4, u]]


* A new graph, with the edges reversed in G_dep:

>> G2 := Network(vertex_list, rev_list);

                                  Network()
```

```
* Topologically sort this graph:

>> table_2 := Network::topSort(G2)

                                  table(
                                    y = 8,
                                    1 = 7,
                                    x = 6,
                                    2 = 5,
                                    z = 4,
                                    3 = 3,
                                    u = 2,
                                    4 = 1
                                  )

* This yields the correct order :
*  (eq4 u eq3 z eq2 x eq1 y).
```

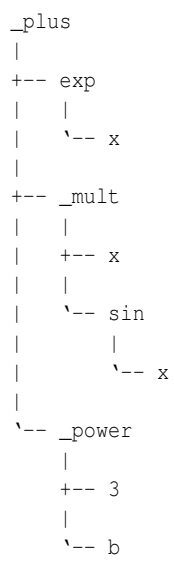
# Appendix 2

```
** Here are some examples of the internal tree representation of expressions
** in MuPAD. The representation is also canonical. The command
**    prog::exprtree(expression)
** displays the tree structure of 'expression'.

>> ex1 := (a-b)/c

                                    a - b
                                    -----
                                      c

>> prog::exprtree(ex1)

                                _mult
                                |
                                +-- _power
                                |   |
                                |   +-- c
                                |   |
                                |   '-- -1
                                |
                                '-- _plus
                                    |
                                    +-- a
                                    |
                                    '-- _mult
                                        |
                                        +-- b
                                        |
                                        '-- -1

                                     Tree1


>> ex2 := (a-b)/(c+d) : prog::exprtree(ex2)


                                _mult
                                |
                                +-- _power
                                |   |
                                |   +-- _plus
                                |   |   |
                                |   |   +-- c
                                |   |   |
                                |   |   '-- d
                                |   |
                                |   '-- -1
                                |
                                '-- _plus
                                    |
                                    +-- a
                                    |
                                    '-- _mult
                                        |
                                        +-- b
                                        |
                                        '-- -1

                                     Tree2
```

```
>> ex3 := x*sin(x) + exp(x) + 3^b
```

$$\exp(x) + x \sin(x) + 3^b$$

```
>> prog::exprtree(ex3)
```




Tree3

```
>> ex4 := 1/x - 1/y + exp(-2*x) + ln(y) = 5
```

$$\ln(y) + \frac{1}{x} - \frac{1}{y} + \exp(-2\,x) = 5$$

```
>> prog::exprtree(ex4)
```

```
_equal
|
+-- _plus
|   |
|   +-- ln
|   |   |
|   |   `-- y
|   |
|   +-- _power
|   |   |
|   |   +-- x
|   |   |
|   |   `-- -1
|   |
|   +-- _mult
|   |   |
|   |   +-- _power
|   |   |   |
|   |   |   +-- y
|   |   |   |
|   |   |   `-- -1
|   |   |
|   |   `-- -1
|   |
|   `-- exp
|       |
|       `-- _mult
|           |
|           +-- x
|           |
|           `-- -2
|
`-- 5
```

Tree4