Multi-Valued Computer Algebra

Christèle Faure, James Davenport, Hanane Naciri

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Multi-valued Computer Algebra

Christèle Faure — James Davenport — Hanane Naciri

N° 4001

Septembre 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4001--FR+ENG

Multi-valued Computer Algebra

Christèle Faure*, James Davenport†, Hanane Naciri‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet CAFE

Rapport de recherche n° 4001 — Septembre 2000 — 36 pages

Abstract: One of the main strengths of computer algebra is being able to solve a family of problems with one computation. In order to express not only one problem but a family of problems, one introduces some symbols which are in fact the parameters common to all the problems of the family.
The user must be able to understand in which way these parameters affect the result when he looks at the answer. Otherwise it may lead to completely wrong calculations, which when used for numerical applications bring nonsensical answers. This is the case in most current Computer Algebra Systems we know because the form of the answer is never explicitly conditioned by the values of the parameters. The user is not even informed that the given answer may be wrong in some cases then computer algebra systems can not be entirely trustworthy. We have introduced multi-valued expressions called *conditional* expressions, in which each potential value is associated with a condition on some parameters. This is used, in particular, to capture the situation in integration, where the form of the answer can depend on whether certain quantities are positive, negative or zero. We show that it is also necessary when solving modular linear equations or deducing congruence conditions from complex expressions.

Key-words: Computer Algebra, reliable formal computation, modular arithmetic, formal integration, simplification.

* Email : Christele.Faure@sophia.inria.fr, URL : http://www.inria.fr/tropics/Christele.Faure
† Email : jhd@maths.bath.ac.uk
‡ Email : Hanane.Naciri@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Calcul Formel multi-valué

Résumé : La force principale du Calcul Formel est l'obtention de familles d'expressions en un seul calcul. Pour exprimer la notion de famille d'expressions, des symboles sont introduits. Ces symboles font office de paramètres et en changeant leur valeur toutes les expressions de la famille peuvent être atteints.
A la lecture de la solution, l'utilisateur doit pouvoir comprendre l'influence des paramètres sur les calculs effectués. En particulier, les incohérences de calculs dues à des valeurs particulières des paramètres doivent être détectées. Si aucune détection de cohérence n'est effectues, les calculs n'ont plus de sens. C'est ce qui se produit dans les systèmes de Calcul Formel courants. L'utilisateur n'est pas prévenu que la solution générique calculée n'est pas valide pour certaines valeurs des paramètres.
Pour résoudre ce problème, nous proposons la notion de calcul conditionnel multivalué. Nous avons suivi cette approche pour implémenter un intégrateur formel fiable ainsi qu'un mécanisme de résolution d'équations linéaire modulaires et de déduction de conditions de congruence sur des expressions complexes.

Mots-clés : Calcul Formel, calcul formel fiable, arithmétique modulaire, intégration formelle, simplification.

Contents

Chapter 1

Description of the problem

1.1 Introduction

Very often, it is necessary to perform an algebraic operation on data containing not just the variables involved in the operation, but also some other variables which are meant to be parameters affecting the solution. Indeed, this is one of the main strengths of computer algebra — to be able to handle not just one problem, but a whole family of problems with a single computation. Clearly, if the parameter is in the input, we would expect to see the parameter in the output. This paper is not concerned with this issue, but rather the case when the *form* of the answer can depend on the parameter, or, more precisely, on the potential values that the parameter can take.

A classic example of this is in the area of integration.

Petit Bois in [16] gives the following example.

$$\begin{aligned}\int \frac{dx}{ax^2+bx+c} &= \frac{1}{\sqrt{b^2-4ac}} \log \frac{2ax+b-\sqrt{b^2-4ac}}{2ax+b+\sqrt{b^2-4ac}} && (b^2 > 4ac)\\ &= \frac{2}{\sqrt{4ac-b^2}} \operatorname{atan} \frac{2ax+b}{\sqrt{4ac-b^2}} && (b^2 < 4ac)\\ &= \frac{-2}{2ax+b} && (b^2 = 4ac)\end{aligned}$$

The two answers for $b^2 > 4ac$ and $b^2 < 4ac$ are symbolically[1] equivalent, under the definitions for logarithms and arc tangents of complex numbers, but, if the wrong one is used,

[1] We carefully do not say mathematically equivalent, since nasty questions of complex analysis can intervene. If the first answer is used for real a, b, c, x and $b^2 < 4ac$, then under today's usual conventions for the branch cut of log [2], there is a sudden jump at $x = -b/2a$, of magnitude $\frac{-2\pi}{\sqrt{4ac-b^2}}$. Put another way, the two formulae are equivalent except of a set of measure zero [8].

a real number is computed via two complex numbers. This may be a minor confusion for the direct human user, but it has potentially serious implications, in particular in numerical use of the answers such as evaluation for some values of parameters, automatic code generation etc.

The answer for $b^2 = 4ac$ is qualitatively different: to begin with it involves no transcendental functions. It would be tempting to think that it could be obtained as a limit from the other solutions, but in fact computer algebra system find this impossible in general (since, for the atan form, there is a hidden constant of integration that tends to infinity as $a \to 0$).

This problem has been highlighted in previous papers, for example in [1, 9].

1.2 Existing systems

Let us first analyse how existing computer algebra systems deal with this question of answers which differ for different ranges of parameter values, or for special values of the parameter.

They mix several ways to treat multi-valued expressions, depending on the kind of operation they deal with (integration, limit, solve ...). But one can recognise four different strategies :

Minimal strategy Under this approach, the system gives the answer that "looks better", as in Reduce :

```
1: int(1/(x**2+a),x);

                   x
 sqrt(a)*atan(---------)
                sqrt(a)
-------------------------
            a

2: int(1/(x**2-a),x);

(sqrt(a)*( - log( - sqrt(a) - x) + log(sqrt(a) - x)))/(2*a)
```

This strategy is a little crude: the answer may be wrong in some cases, but the user cannot do anything about that. In this example, it uses an atan formulation unless this would involve taking the square root of a negative constant or a negative variable power. For example, the following answer is not particularly helpful if we are thinking of b as a small quantity.

```
3: int(1/(x**2-(1-b**2)),x);

       2                 X
 SQRT(B  - 1)*ATAN(--------------)
```

```
                          2
                    SQRT(B  - 1)
---------------------------------
               2
              B  - 1
```

Proviso strategy Under this approach, the user may guide the system. Before the start of the computation he may assert [17] some informations (about the sign or the type of a symbol ...) as in `Maple, Mathematica, Macsyma`. But if he doesn't, the system makes some choices stored in a `proviso` accessible after computation: `Maple, Mathematica` and the last version of `Macsyma`. In previous versions of `Macsyma` the user was asked some questions by the system.

```
(C1) integrate (1/(x**2+a),x);
Is  A  positive or negative?

neg;
                        2 X - 2 SQRT(- A)
                    LOG(-----------------)
                        2 X + 2 SQRT(- A)
                    ----------------------
                         2 SQRT(- A)
(C2) assume (a>0);
(D2)                          [A > 0]
(C3) integrate (1/(x**2+a),x);
                                X
                           ATAN(-------)
                                SQRT(A)
(D3)                       -------------
                              SQRT(A)
```

Using this strategy makes the answer fit what is needed by the user. The main drawback of this approach is that when such a calculation is involved in a complicated computation, the user may not know how to guide the system.

Implicit strategy `Maple` and `Derive` in some cases produce conditional answers using the `signum` or the `Heaviside` function such as for a limit :

```
> limit (a*x,x=infinity);
                   signum(a) infinity
```

In this way a multiple result is encoded in a mono-shaped expression, that's why we call this strategy implicit. The drawback of this strategy is that the result may be really complicated and then intractable for the user. It is also hard to do, or at least express the results, in general, as in

```
> limit(exp(a*x),x=infinity);

                  limit         exp(a x)
                  x -> infinity
```

where the answer should be $\begin{cases} \infty & a > 0 \\ 1 & a0 \\ 0 & a < 0 \end{cases}$, which does not easily fit the `signum` paradigm.

Explicit strategy We describe as "explicit" the strategy that consists of explicitly returning multiple results.

`Axiom` may return a list of solutions :

```
(1) -> integrate(1/(x**2+a),x)

   (1)
          2      +---+                 2
    log((x  - a)\|- a  + 2a x) - log(x  + a)
   [--------------------------------------,
                     +---+
                    2\|- a
          x
    atan(----)
          +-+
         \|a
    ----------]
        +-+
       \|a
     Type: Union(List Expression Integer,...)
```

The user may not be able to handle the result directly, but it reminds him that multiple results are possible.

`Mathematica` may answer with an `If` expression to some requests :

```
In[2]:= Integrate[Cos[c*x]*x^n, {x,0,Infinity}]

Out[2]= If[Im[c] == 0,
            2 (-1 - n)/2                 1 + n
          (c )           Sqrt[Pi] Gamma[-----]
                                         2
          ------------------------------------,
                    1 n/2       -n
                  (-)    Gamma[--]
                   4           2
             ComplexInfinity]
```

But when working with `If` expressions, one can notice that `Mathematica` doesn't treat them in a uniform way: just a few operations (like `D`) deal with them, those operations do not do systematic evaluations or simplifications, the `Map` operator applies the function even to the test ...

```
In[2]:= D [If[x >= 0, x, -x],x]

Out[2]= If[x >= 0, 1, -1 1]

In[3]:= Simplify [%]

Out[3]= If[x >= 0, 1, -1 1]

In[4]:= Map [Evaluate,%]

Out[4]= If[x >= 0, 1, -1]

In[5]:= % + x

Out[5]= x + If[x >= 0, 1, -1]

In[6]:= Map [f,Out[3]]

Out[5]= If[f[x >= 0], f[1], f[-1 1]]
```

Those `If` expressions are unevaluated control structures, and are not piecewise functions.

This strategy seems to us the best one to give right answers fitting the mathematical definition of operators, but the two existing implementations of multi-valued expressions are not powerful enough.

None of those general systems give the mathematically exact solution of the integral example, nor do they implement the form of expressions quoted from Petit Bois.

The application of theorem proving to computer algebra is also beeing studied: [10] presents a general deductive database for mathematical formulas, whereas [3] presents verified table look-up applied to symbolic definite integral. We do not take this data base approach where the solutions are recorded within a table, to be extracted on demand.

1.3 Our approach

The last strategy seems the right one to us, because the user may be given the mathematically right solutions, but the existing implementations of this strategy are not powerful enough. We therefore choose to implement an explicit strategy. But using this strategy by itself

leads to computing with large (in terms of the number of cases) expressions. In order to diminish the size of multi-valued expressions, the system must allow the user to choose some solutions if he knows something about the parameter involved in the corresponding conditions. Therefore we have implemented a mixed strategy: an explicit strategy using the functionalities of a proviso-like strategy.

In order to treat multi-valued expressions as piecewise functions we define a new kind of expressions called conditional expressions. Each conditional expression consists in a list of constrained values, which may be written as `IFF ((cond1,val1),(cond2,val2) ...)`. We choose not to take into account values such as `Undefined`, `Error`, but rather to produce a solution that gives generic (but correct) answers. For example, `1/a` will not be computed as `IFF (a=/=0,1/a,Error)`.

One wants to involve such objects in computations such that (in a fictitious system) :

```
IFF ((x>0,x),(x=0,0),(x<0,-x)) + y
--> IFF ((x>0,x+y),(x=0,y),(x<0,y-x))

Abs (x+y)
--> IFF ((x+y>0,x+y),(x+y=0,0),(x+y<0,-x-y))

Map (f,Abs (x+y))
--> IFF ((x+y>0,f (x+y)),(x+y=0,f (0)),
         (x+y<0,f (-x-y)))

Assert (x+y<0)
--> x+y<0

Abs (x+y)
--> -x-y
```

Chapter 2

Generic Conditional Expressions

2.1 General expressions

General expressions are implemented as **trees**. This is the most general representation and has been chosen for our study. The root of the tree is the main operator and the sub-trees are operands. For example, the expressions $x^2 + 3 * x + 1$ and $x * y - 2$ are represented by the trees shown in Figure 2.1.

The canonical form of these expressions make both expressions $x + y$ and $y + x$ to be the same. More generally, if F is associative then $F(x_1, x_2, .., x_n)$ is flatten and if F is commutative, $F(op_1, op_2, .., op_n)$ is rewritten so that $op_1, op_2, .., op_n$ are ordered. It is the case for $+$ and $*$ in algebraic expressions or $\vee$ and $\wedge$ in boolean expressions. A generic simplification algorithm can be written using some operators associated to F: neutre$_F$?, absorbant$_F$?, combine$_F$ and $<_F$.

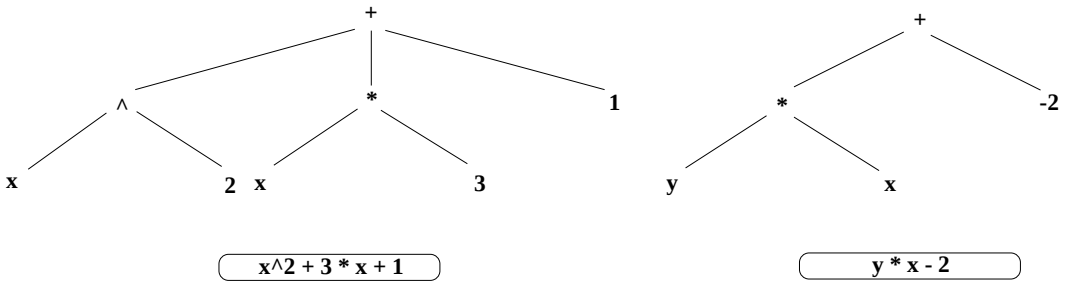

Figure 2.1: Tree representation of expressions

Algorithm 1 Simplification of F [op1,op2]

Require: op1 et op2 are simplified expressions
  if F? op1 and F? op2 **then**
    l ← append(operand op1, operand op2)
  else
    if F? op1 **then**
      l ← insert(op2, operand op1)
    else
      if F? op2 **then**
        l ← insert(op1, operand op2)
      else
        l ← insert(op1, [op2])
      end if
    end if
  end if
  {Construction of F [e1,..., en]}
  return F [e1, ..., en]

Algorithm 2 Insertion of e in the ordered list l

  neutre$_F$? $e \Rightarrow$ return l
  absorbant$_F$? $e \Rightarrow$ return $[e]$
  vide? $l \Rightarrow$ return $[e]$
  $e <_F$ first $l \Rightarrow$ return append(first l , insert(e, rest l))
  $e >_F$ first $l \Rightarrow$ return append(e, l)
  { e and first l can be combined}
  $z \leftarrow$ combine$_F$(e, first l))
  neutre$_F$? $z \Rightarrow$ return rest l
  absorbant$_F$? $z \Rightarrow$ return $[z]$
  return append(z, rest l)

2.2 Algebraic Expressions

Algebraic expressions are implemented as general expressions. Formal simplifications are performed as described above. For example, if F is the product $*$, the operator neutre$_F$? tests if its argument is 1 and absorbant$_F$? tests if its arguments is 0. The operator combine$_F$ applied on x^3 and x computes x^4. The operator $<_F$ uses the general lexicographic order.

Some simplification due to integer arithmetic are applied: $2 + 3 \rightarrow 5$.

2.3 Boolean expressions

Boolean expressions are built from atomic boolean expressions and general boolean operators $\wedge$, $\vee$ and *not*. They are implemented as boolean expressions where the leaves are basic conditions. Boolean expressions are simplified using Algorithm 1 and Algorithm 2 with the correct parameters. For example, for operator $\vee$, neutre$_\vee$ tests if the expression is `False` and absorbant$_\vee$ tests if the expression is `True`.

We add one general simplification using distributivity of $\wedge$ with respect to $\vee$:

$$(e_1 \vee e_2) \wedge e_3 \quad \rightarrow \quad (e_1 \wedge e_3) \vee (e_2 \wedge e_3). \tag{2.1}$$

The *not* operator applied to conditions follows standard rules:

$$\begin{aligned} not(False) &\rightarrow True & (2.2)\\ not(True) &\rightarrow False & (2.3)\\ not(not(C)) &\rightarrow C & (2.4)\\ not(a \vee b) &\rightarrow not(a) \wedge not(b) & (2.5)\\ not(a \wedge b) &\rightarrow not(a) \vee not(b). & (2.6) \end{aligned}$$

For each atomic condition type, some more simplifications can be added. For congruence conditions, two supplementary simplifications are added:

1. use of the Chinese remainder theorem,
2. solving of modular linear equations.

2.4 Conditional expressions

Mono-valued conditional expressions are defined as pairs (V, C) `=IFF(C,V)` where V is a value of type general expression and C is a condition of type boolean expression. For example $(k + 1, k \equiv 15 \pmod{45})$ is a conditional expression and $(x + 1, x > 0)$ is also a conditional expression. The difference between these two examples comes from the type of atomic boolean expression.

This definition is extended to multi-valued conditional expressions $\{(V_i, C_i)\}$ where V_i is a value of type general expression and C_i is a condition of type boolean expression. For

example, $\{(x, x \geq 0)(-x, x < 0)\}$ is a conditional expression that means `IFF (x,x≥ 0) (-x,x<0)`.

The simplification of conditional expression consists first in the simplification of all expressions and all conditions. Some more simplifications can be performed on multi-valued conditional expressions depending on the application. For example, if a pair $(V, False)$ appears in a multi-valued conditional expression it may be discarded. The coherency, completeness can be verified to stop the computation.

Chapter 3

Formal integration

This chapter presents formal integration using conditional computation. The VSDITLU[3] system, a verifiable symbolic definite integral table look-up, has been developed to solve equivalent problems. It is theorem proving applied to computer algebra whereas our approach is conditional computer algebra.

3.1 Conditional expressions with comparison conditions

We have augmented `Axiom` (see [7, 15]) with a new data type known as `Conditional Expression` (abbreviated as `CEX`), together with various supporting structures and categories, notably `EXC` (short for `ExpressionCondition`) which is the type of the sign conditions allowed on (real-valued) expressions.

Those conditions are boolean expressions that compare real-valued expressions with zero (for example ($a > 0$ and $a + b < 0$) or $a < 0$ ...). Those boolean expressions may be arbitrarily complex so we try to diminish their size.
We evaluate the conditions, using the standard Axiom package `PolynomialFunctionSign` to compute the sign of the involved expressions. For example $-(a^2 + b^2) > 0$ and $x < 0$ evaluates to $False$.
Moreover some simplifications, based on the Robinson algorithm to detect tautologies, are computed on conditions. The simplifications may be classified in three classes :

- redundancy ($a > 0$ and $a > 0 \Rightarrow a > 0$),
- coherency ($a > 0$ and $a < 0 \Leftrightarrow False$)
- completeness ($a < 0$ and $a = 0$ and $a > 0 \Leftrightarrow True$).

This mechanism for conditions is taken from [12] and the general principles from [14].

```
(2) -> Or [positive (a*(b-1)),negative (a*(b-1)),_
```

```
          positive (a*(b-1)),null a,null (b-1)]$EXC Integer

   (2)   Or    (a=0)
             (b - 1=0)
           (a b - a<0)
           (a b - a>0)
               Type: ExpressionCondition Integer
```

In order to improve this mechanism, we have added an `assert` function that enables the user or another Axiom program to assert tautologies. All the assertions are stored in a "context", and the conditions are then simplified modulo this context.

```
(3) -> assert (Equi (null (a*(b-1)),Or (null a,null (b-1))))$EXC Integer

   (3)  [lhs= (a b - a=0),rhs=  Or   (a=0)  ]
                                   (b - 1=0)
                                Type: Equivalence
(4) -> simplify %% 2

   (4)   True
                 Type: ExpressionCondition Integer
```

The type `ConditionalExpression` represents expressions of the form quoted from Petit Bois. One can compute the conditional expression `if C then V1 else V2` where `C` is an `ExpressionCondition` and `V1,V2` are two expressions, using `conditionalValue(C,V1,V2)`. One must notice that the `ConditionalExpressions` only contain simplified conditions. To give an explicit example:

```
(5) -> conditionalValue(%,first %% 1, second %% 1)$CEX INT

   (5)
                 2      +---+                2
          log((x  - a)\|- a  + 2a x) - log(x  + a)
((a<0) -> ----------------------------------------)
                             +---+
                           2\|- a
                      x
              atan(----)
                   +-+
                 \|a
(Or (a=0) -> ----------)
    (a>0)        +-+
                \|a
                 Type: ExpressionCondition Integer
```

Each computation valid on `Expression` is also valid on `ConditionalExpression`. For example the particular domain `ConditionalExpression` over `Integer` actually forms a field, with the operator `abs` added, as shown in figure 3.1. We have chosen not to deal with `Error`

values in our conditional objects, the only operator defined in `CEX` to split an expression into a conditional one is `abs`.

3.2 Conditional Integration

We have augmented the Axiom integrator, so that, instead of returning a list of alternatives containing atan and log expressions, it returns a `ConditionalExpression`, with the appropriate conditions.

The shape of $\int(1/(ax^2+bx+c))$ depends on the sign of $\delta = b^2 - 4ac$. `Axiom` tries to compute the sign δ, if it is constant it returns the answer, otherwise it computes the values for $\delta > 0$ and $\delta < 0$.

Those tests appear clearly in the code of the function `quadratic` from `IR2F`[1], then we just had to collect those two solutions and their associated conditions to create the new package `IR2CF`[2]. This expression (with two solutions for $a > 0$ or $a < 0$) is similar to the example given in the previous section. But such an answer is not sufficient because the case $a = 0$ cannot be computed as the limit of one of those two solutions: hence the requirement to place a conditional analysis of critical cases *in* the integrator, rather than *after* it.

We had a little more work to do, to get the solutions for the critical cases. For each critical case (a result of $solve(P * Q = 0)$ where $\delta = P/Q$), a new integral has to be computed. Then a complete integration of $1/(ax^2+bx+c)$ is performed. In fact, to make the integrator answer a conditional expression instead of a list of expressions, we just had to rewrite a piece of code in `IR2F` and then in `CFSINT`.

In order to help the system simplify conditions thereafter, the `integral` function asserts

$$null(P * Q) \Leftrightarrow \bigwedge_{v \in solve(P * Q = 0)} null(v).$$

Hence we get answers such as in Figure 3.2. The requirement to coerce into `EXPR INT`, i.e. Expressions of Integers, is caused by the fact that we have currently only modified the Expression integrator, and not the special-purpose one that works for rational functions.

The facility for assertions described above is used in the equality case, to decide what simpler cases the equality case reduces to. Figure 3.3 shows the behaviour of our new integrator on a simple example.

3.3 Limitations of this implementation

It is worth-while noting what this modification does *not* do.

1. It does not deal with spurious divisions, so that the integral of x^n is returned as $\frac{x^{n+1}}{n+1}$, or, to be precise,

[1] `IR2F` is the abbreviation of `IntegrationResultToFunction`
[2] `IR2CF` is the abbreviation of `IntegrationResultToConditionalFunction`

```
(6) -> %% 5 + x

                        2      +---+              +---+        2
                   log((x  - a)\|- a  + 2a x) + 2x\|- a  - log(x  + a)
   (6)   ((a<0) -> ------------------------------------------------)
                                      +---+
                                   2\|- a
                                     x        +-+
                                atan(----) + x\|a
                                     +-+
                                    \|a
                         ( Or (a=0) -> ------------------)
                              (a>0)            +-+
                                              \|a

                                                Type: ConditionalExpression Integer
(7) -> (1/x)::CEX INT

                1
   (7) (True -> -)
                x

                                                Type: ConditionalExpression Integer
(8) -> abs (-3 /(x**2+a**4))$CEX INT

             2    4          - 3
   (8)   ((x  + a < 0) -> -------)
                           2    4
                          x  + a

                 2    4           3
          ((Or (x  + a =0)) -> -------)
                 2    4         2    4
               (x  + a >0)     x  + a

                                                Type: ConditionalExpression Integer
```

Figure 3.1: Computations with `ConditionalExpressions`

```
(13) -> 1/(x**2+a) :: EXPR INT
           1
   (13)  ------
          2
         x  + a
                        Type: Expression Integer
(14) -> integrate(%,x)

   (14)
                 2      +---+                2
           log((x  - a)\|- a  + 2a x) - log(x  + a)
  ((a<0) -> ----------------------------------------)
                            +---+
                          2\|- a
                             1
                   ((a=0) -> ---)
                             x
                               x
                          atan(----)
                                +-+
                               \|a
                ((a>0) -> ------------)
                                +-+
                               \|a
             Type: ConditionalExpression Integer
```

Figure 3.2: First example of conditional integration

```
(15) -> a/(x**2-a**2+1) :: EXPR INT

              a
   (15)  -----------
                2    2
         Type: Expression Integer
(16) -> integrate(%,x)

                                                                  1
   (16)                                        ((a - 1=0) -> - -)
                                                                  x
                                                  ((a=0) -> 0)
                                                                  1
                                                ((a + 1=0) -> -)
                                                                  x
                                                                      x
                                                     a atan(-----------)
                                                                +--------+
                                                                |   2
                                          4    2                \|- a  + 1
                                       ((a  - a <0) -> --------------------)
                                                              +--------+
                                                              |   2
                                                             \|- a  + 1
                                             +------+
                                 2    2      | 2             2                 2    2
   4    2       a log((x  + a  - 1)\|a  - 1  + (- 2a  + 2)x) - a log(x  - a  + 1)
((a  - a >0) -> ------------------------------------------------------------------)
                                               +------+
                                               | 2
                                              2\|a  - 1

                                  Type: ConditionalExpression Integer
```

Figure 3.3: Output from Command 16.

```
(17) -> integrate (x**n,x)

                n log(x)
             x %e
   (17)  ------------
                n + 1
              Type: Union(Expression Integer,...)
```

This is a fundamentally different kind of problem: the generic answer is valid away from a point singularity. Here there are two possibilities: the methodology suggested by Duval (see for example [13]), which would involve performing the whole of the integration algorithm with `ConditionalExpressions` rather than `Expressions`); or a retrospective analysis of the result to decide if any special cases need treatment, and, if so, a special-case integration of these cases. We intend to pursue this latter approach in further studies.

2. It does not deal with cases where the expression as posed is un-integrable in closed form, but certain special cases are integrable. This is a very difficult problem and in the case of algebraic functions, probably undecidable [11], since an infinite number of special rational values of a transcendental parameter may need to be checked.

Chapter 4

Solving modular equations

4.1 Theoretical back-ground

We recall that modular arithmetic is performed in the ring $(\mathbb{Z}_n, +, \cdot)$ where $+$ and $\cdot$ are performed modulo n. The unit group of $\mathbb{Z}_n$ is $\mathbb{Z}_n^* = \{m \in \mathbb{Z}_n : \gcd(m, n) = 1\}$. The solving of a modular linear equation is described and finally the Chinese Remainder Algorithm is given.

The Bezout theorem (Extended Euclidean algorithm)

The Bezout theorem states that the GCD (Greatest Common Divisor) d of two integers a and b denoted by $\gcd(a, b)$ is a linear combination of a and b :

$$d = \gcd(a, b) = au + bv.$$

A simple modification of the Euclidean algorithm allows for the calculation of the two coefficients u and v. This modified version of the Euclidean algorithm is called the *Extended Euclidean algorithm*. These coefficients are used within the Chinese remainder theorem as well as in the modular multiplicative inverse computation. The Bezout algorithm described in Algorithm 3 takes as input two nonnegative integers and returns a pair of integers (u, v) that satisfies the equation $\gcd(a, b) = au + bv$.

For example, the Bezout coefficients of $a = 1292$ and $b = 798$ in $\mathbb{Z}$ are $u = -8$ and $v = 13$ (the gcd is 38).

4.1.1 Modular linear equations solving

We consider solving the equation $ax \equiv b \pmod{n}$ where $a > 0$ and $n > 0$. a, b and n are given and the values x that satisfy this linear equation are looked for. This equation may have zero, one or more solutions in $\mathbb{Z}_n$.

Here is a summary of some results necessary for solving the equation $ax \equiv b \pmod{n}$.

Algorithm 3 The Bezout algorithm

{Computation of Bezout coefficients (u, v) for (a, b)}
$(u, u') \leftarrow (1, 0)$
$(v, v') \leftarrow (0, 1)$
$(r, r') \leftarrow (a, b)$
while $r' \neq 0$ **do**
  $q \leftarrow r/r'$
  $(u_0, v_0, r_0) \leftarrow (u, v, r)$
  $(u, v, r) \leftarrow (u', v', r')$
  $(u', v', r') \leftarrow (u_0 - qu', v_0 - qv', r_0 - qr')$
end while
return (u, v)

Theorem 1 *The equation* $ax \equiv b \ (mod \ n)$ *either has* d *distinct solutions modulo* n *such that* $d = \gcd(a, n)$ *and* $1 \leq d \leq n$*, or has no solution if* $\gcd(a, n)$ *does not divide* b*.*

Theorem 2 *Let* u *and* v *be the Bezout coefficients of* a *and* n *such that* $d = au + nv$*. If* $d|b$ *(*d *divides* b*),* x_0 *is one of the solutions of equation* $ax \equiv b \ (mod \ n)$ *where*

$$x_0 \equiv u(b/d) \ (mod \ n).$$

Theorem 3 *Suppose that equation* $ax \equiv b \ (mod \ n)$ *is solvable (that is* $d|b$ *where* $d = \gcd(a, n)$*) and that* x_0 *is one of its solutions. Then, this equation has exactly* d *distinct solutions modulo* n *given by* $x_i = x_0 + i(n/d)$ *for* $i = 1, 2, \cdots, d - 1$*.*

Algorithm 4 for solving modular linear equations returns all the solutions of any equation of form $ax \equiv b \pmod{n}$. The inputs are two arbitrary positive integers a and n and one arbitrary integer b.

Let consider one run of this algorithm on equation $14x \equiv 30 \pmod{100}$ ($a = 14$, $b = 30$, and $n = 100$). The result is $x_0 = 95$ and $x_1 = 45$. This equation is equivalent to two congruence equations: $x \equiv 95 \pmod{100} \vee x \equiv 45 \pmod{100}$.

4.1.2 Chinese remainder theorem

The Chinese remainder theorem can be applied to integers or polynomials. In the following, we explain only the integer case. Two cases have to be dealt with: (1) the two integers are supposed relatively prime and (2) the general case. Algorithm 5 presents an extended version of the Chinese remainder theorem.

Algorithm 4 Solving of equation $ax \equiv b \pmod{n}$

{INPUTS: a, b and n }
$d \leftarrow \gcd(a, n)$
if $d|b$ **then**
  $(u, v) \leftarrow bezout(a, n)$ { u and v are Bezout coefficients}
  $x_0 \leftarrow u(b/d) \ mod \ n$
  for $i \leftarrow 0$ to $d - 1$ **do**
    $x_i \leftarrow (x_0 + i(n/d)) mod n$
    return x_i
  end for
else {d does not divide b}
  False {No solution }
end if

Relatively prime case

Theorem 4 *Let M and N be two relatively prime integers. For each pair of integers (a, b) there exists a unique $0 \leq c \leq MN$ such that:*

$$\left\{ \begin{array}{l} x \equiv a \ (mod \ M) \\ x \equiv b \ (mod \ N) \end{array} \right. \Leftrightarrow \quad x \equiv c \ (mod \ MN)$$

Proof. From the extended Euclidean algorithm, there exists two integers f and g such that $fM + gN = 1$ (M and N are considered relatively prime).

Let choose c such that $c = a + (b - a)fM = a + (b - a)(1 - gN) = b + (a - b)gN$.

$$\left. \begin{array}{l} x \equiv a \pmod{M} \\ c \equiv a \pmod{M} \end{array} \right\} \Rightarrow \quad x \equiv c \pmod{M} \Rightarrow \ \exists \ k \in \mathbb{Z} \text{ s.t. } x = c + kN$$

$$\left. \begin{array}{l} x \equiv b \pmod{N} \\ c \equiv b \pmod{N} \end{array} \right\} \Rightarrow \quad x \equiv c \pmod{N} \Rightarrow \ \exists \ l \in \mathbb{Z} \text{ s.t. } x = c + lM$$

$$\Rightarrow \quad kN = lM$$

If $kN = lM$ then $M|kN$, but $\gcd(M, N) = 1$ so $M|kN$. As $x = c + kN$, $x \equiv c \pmod{MN}$.

This proves the first implication and the rest of the proof is devoted to the reciprocal implication.

$$x \equiv c \pmod{MN} \quad \Rightarrow \quad \left. \begin{array}{l} x \equiv c \pmod{M} \\ c \equiv a \pmod{M} \end{array} \right\} \Rightarrow \quad x \equiv a \pmod{M}$$

$$x \equiv c \pmod{MN} \quad \Rightarrow \quad \left. \begin{array}{l} x \equiv c \pmod{N} \\ c \equiv b \pmod{N} \end{array} \right\} \Rightarrow \quad x \equiv b \pmod{N}$$

General case

Theorem 5 *Let M and N be two integers. For each pair of integers (a, b),*
(1) if $a \not\equiv b \ (mod\ \gcd(M,N))$ then the equations $x \equiv a \ (mod\ M)$ and $x \equiv b \ (mod\ N)$ are never satisfied,
(2) if $a \equiv b \ (mod\ \gcd(M,N))$ then there exists a unique c such that:

$$\left\{ \begin{array}{l} x \equiv a \ (mod\ M) \\ x \equiv b \ (mod\ N) \end{array} \right. \Leftrightarrow \quad x \equiv c \ (mod\ \mathrm{lcm}(M,N))$$

Proof.
(1) If $x \equiv a \pmod{M}$, then $x \equiv a \pmod{\gcd(M,N)}$. In the same way, if $x \equiv b \pmod{N}$, then $x \equiv b \pmod{\gcd(M,N)}$, so $a \equiv b \pmod{\gcd(M,N)}$.
(2) if $a \equiv b \pmod{\gcd(M,N)}$, from the Extended Euclidean algorithm, there exists two integers s and t such that $sM + tN = g = \gcd(M,N)$.

Let choose c such that $c = a + (b-a)s\frac{M}{g} = a + (b-a)(1 - (t\frac{N}{g})) = b - tN\frac{b-a}{g}$.

$$\left. \begin{array}{rcl} x \equiv c \pmod{\mathrm{lcm}(M,N)} & \Rightarrow & x \equiv c \pmod{M} \\ c = a + (b-a)s\frac{M}{g} & & \\ a \equiv b \pmod{g} & \Rightarrow & c \equiv a \pmod{M} \end{array} \right\} \Rightarrow \quad x \equiv a \pmod{M}$$

$$\left. \begin{array}{rcl} x \equiv c \pmod{\mathrm{lcm}(M,N)} & \Rightarrow & x \equiv c \pmod{N} \\ c = b - tN\frac{b-a}{g} & & \\ a \equiv b \pmod{g} & \Rightarrow & c \equiv b \pmod{N} \end{array} \right\} \Rightarrow \quad x \equiv b \pmod{N}$$

This proves the reciprocal implication and the rest of the proof is devoted to the first implication.

$$\left. \begin{array}{rcl} x \equiv a \pmod{M} & \Rightarrow & \exists k \in \mathbb{Z} \text{ s.t. } x = a + kM \\ & \Rightarrow & sx = sa + k(g - tN) \\ & \Rightarrow & sx \equiv sa + kg \pmod{N} \\ x \equiv b \pmod{N} & \Rightarrow & sx \equiv sb \pmod{N} \end{array} \right\} \Rightarrow \quad kg \equiv s(b-a) \pmod{N}$$

$kg \equiv s(b-a) \pmod{N}$ is equivalent to $kg = s(b-a) + rN$ where r is an integer. Then,

$$x = a + kM = a + \frac{M}{g}gk = a + \frac{M}{g}(s(b-a) + rN) = a + \frac{M}{g}s(b-a) + \frac{MN}{g}r,$$

Moreover, $\mathrm{lcm}(M,N) = \frac{MN}{g}$ and $c = a + \frac{M}{g}s(b-a)$, so $x = c + \mathrm{lcm}(M,N)r$. This implies that $x \equiv c \pmod{\mathrm{lcm}(M,N)}$.

Chinese Remainder Algorithm

The Chinese Remainder Algorithm 5 is applied to reduce two congruence equations $x \equiv a \pmod{M}$ and $x \equiv b \pmod{N}$ to one equation $x \equiv c \pmod{\mathrm{lcm}(M,N)}$ by the calculation of c.
For example, $x \equiv 1 \pmod{180}$ and $x \equiv 61 \pmod{600}$ are reduced to $x \equiv 1261 \pmod{1800}$ using algorithm 5 where $a = 1$, $M = 180$, $b = 61$, $N = 600$, $c = 1261$ and $m = 1800 = \mathrm{lcm}(180, 600)$.

Algorithm 5 Application of the Chinese Remainder Algorithm

$\{ x \equiv a \pmod{M} \text{ and } x \equiv b \pmod{N}\}$
$g \leftarrow \gcd(M,N)$
$(s,t) \leftarrow bezout(M,N)$ { s and t are Bezout coefficient}
$m \leftarrow \mathrm{lcm}(M,N) = \frac{MN}{g}$
if $a \not\equiv b \pmod{g}$ **then**
  return False
else
  $c \leftarrow (a + (b-a)s\frac{M}{g}) \bmod m$
  return (c,m) $\{ x \equiv c \pmod{m}\}$
end if

4.2 Conditional Modular computations

This package is developed in the ALDOR [6] language as a separate module. This section shows in an ALDOR session basic conditional modular computations: algebraic expressions, congruence conditions and conditional expressions.

4.2.1 Algebraic expressions

A library dedicated to the computation of algebraic expressions is developed. Simplification of these expressions is described in 2.2.

Loading of our library in the ALDOR interpreter:

```
%1 >> #include "expcond"
```

Simplification of our algebraic expressions:

```
%3 >> e1 := 3*x+2-x; e2 := 62+x-100
                () @ AlgExpression
%4 >> stdout << e1 << endnl << e2
2 . x + 2
x - 38  () @ OutputStream
```

Canonicalisation of the expression by an ordering of the terms:

```
%7 >> e3 := -1+2*x+x^3
                () @ AlgExpression
%8 >> stdout << e3
x ^ 3 + 2 . x - 1  () @ OutputStream
```

Automatic simplification of terms: x^2 and $-4x^2$ are combined to obtain a new term $-3x^2$:

```
%9 >> e4 := x^2+x-4*x^2
                () @ AlgExpression
%10 >> stdout << e4
-3 . x ^ 2 + x  () @ OutputStream
```

Application of the standard simplification rules:

```
%11 >> e5:=x*y+2*x^2*y+y^2*x^2-2*y*x
                () @ AlgExpression
%12 >> stdout << e5
(y ^ 2 . x ^ 2) + 2 . (x ^ 2 . y) - x . y  () @ OutputStream
```

Some specific operators have been defined to compare (`=`), or compute the type of an expression (`integer?`, `somme?`, `power? ...`) :

```
%3 >> e:= x+3
               () @ AlgExpression
%4 >> stdout << somme? e << endnl << power? e
T
F () @ OutputStream

%6 >> e1:= x+2*y+4; e2:=4*y+10+x-6-2*y
               () @ AlgExpression
%7 >> stdout << ( e1 = e2 )
T  () @ OutputStream
%9 >> e1:= x+2*y+2
               () @ AlgExpression
%10 >> stdout << ( e1 = e2 )
F  () @ OutputStream
```

4.2.2 Formal congruence conditions

Formal congruence conditions are built as general boolean expressions with congruence conditions as atomic boolean expressions. General simplification rules are described in Section 2.3.

Atomic congruence conditions allow for specifics simplification. They are constructed with the operator `congru`. For example, the ALDOR command `congru(x,4,5)` built the condition $x \equiv 4 \pmod 5$:

```
%3 >> c1:= congru(x,4,5)
                        () @ GeneralCondition
%4 >> stdout << c1
{x = 4 (mod 5)}  () @ OutputStream
%5 >> c2:= congru(x,4,5) \/ congru(x,3,5)
                        () @ GeneralCondition
%6 >> stdout << c2
{{x = 4 (mod 5)} \/ {x = 3 (mod 5)}}  () @ OutputStream
%7 >> c3:= congru(y,2,6) /\ congru(y,0,4)
                        () @ GeneralCondition
%8 >> stdout << c3
{y = 8 (mod 12)}  () @ OutputStream
%9 >> c:= c2 /\ c3
                        () @ GeneralCondition
%10 >> stdout << c
{{y = 8 (mod 12)} /\ {x = 4 (mod 5)} \/
{{y = 8 (mod 12)} /\ {x = 3 (mod 5)}}
```

A *normal form* is defined over these conditions. For example, condition $3 \equiv 1 \pmod 2$ is simplified to `Always` (`True`) whereas condition $3 * x \equiv 4 \pmod 6$ is reduced to `Never` (`False`) because this equation has no solution:

```
%3 >> c:= congru(3::AlgExpression,1,2)
                       () @ GeneralCondition
%4 >> stdout << c
Always  () @ OutputStream
%5 >> c:= congru(3*x,4,6)
                       () @ GeneralCondition
%6 >> stdout << c
Never   () @ OutputStream
```

Key example:
The problem is to find n such that $n = (k+1)(5k+1)(9k+1)$ is a Carmichael number

where $(k+1)$, $(5k+1)$ and $(9k+1)$ are relatively prime. The symbol k is associated to a congruence condition C defined as follows:

$$C = \begin{cases} k \equiv 0 \pmod{7} \\ k \equiv 2 \pmod{4} \\ k \equiv 15 \pmod{45} \\ k \equiv 0 \pmod{11} \vee k \equiv 1 \pmod{11} \end{cases}$$

This example illustrates the simplification problem that arise when finding counterexamples to pseudo primality tests [4]. The system simplifies the condition C to a new condition:

```
%13 >> c:= congru(k,0,7)/\congru(k,2,4)/\congru(k,15,45)
    /\(congru(k,0,11)\/congru(k,1,11))
                          () @ GeneralCondition
%14 >> stdout << c
{{k = 4830 (mod 13860)} \/ {k = 2310 (mod 13860)}}
                           () @ OutputStream
```

The simplified condition $C1 = (k \equiv 2310 \pmod{13860} \vee k \equiv 4830 \pmod{13860})$ allows for the enumeration of all possible values for k by varying k' in equalities $k = 13860k' + 2310$ and $k = 13860k' + 4830$.

4.2.3 Conditional expressions with congruence conditions

Conditional expressions with congruence conditions are algebraic expressions where each expression may be associated to a congruence condition. For example, the expression $\{2x + 3, 2x + 3 \equiv 3 \pmod{4}\}$ is computed in our package by:

```
%5 >> c1:= condition(2*x+3, congru(2*x+3,3,4))
                           () @ ConditionalExpression
%6 >> stdout << c1
[ 2 . x + 3,{x = 2 (mod 4)} \/ {x = 0 (mod 4)} ]
                           () @ OutputStream
```

A condition may be simplified to `Never` within a conditional expression without cancellation of the total expression:

```
%7 >> c2:= condition(2*x+3, congru(2*x+3,2,4))
                           () @ ConditionalExpression
%8 >> stdout << c2
[ 2 . x + 3 , Never ]
                           () @ OutputStream
```

Conditional expressions may be combined thanks to classical arithmetic operators:

```
%4 >> ec1:= condition(2*x, congru(x,1,5))
  () @ ConditionalExpression
%5 >> stdout << ec1
[ 2 . x,{x = 1 (mod 5)}]  () @ OutputStream
%6 >> ec2:= condition(x, congru(x,0,2))
  () @ ConditionalExpression
%7 >> ec3:= ec1 + condition(3::AlgExpression)
          - ec2 + condition(4::AlgExpression)
  () @ ConditionalExpression
%8 >> stdout << ec3
[ x + 3,{x = 6 (mod 10)}]  () @ OutputStream
```

Even if a symbol does not appear within a simplified expression any condition concerning this symbol is kept in the conditional expression:

```
 %3 >> ec:= condition(x,congru(x,0,2)) - condition(x, congru(x, 5, 7))
                            () @ ConditionalExpression
 %4 >> stdout << ec5
 [ 0,{x = 12 (mod 14)}]  () @ OutputStream
```

Key application:
Deduction easily implements in this context. We implement a minimal *Deduction* mechanism over congruence conditions based on modular arithmetic results presented in Section 4.1.1. For example, from $n \equiv 2 \pmod 4$ and $m \equiv 2 \pmod 4$, condition $n + m \equiv 0 \pmod 4$ is deduced.

$$\begin{cases} n \equiv 2 \pmod 4 \\ m \equiv 2 \pmod 4 \end{cases} \Rightarrow \quad n + m \equiv 0 \pmod 4$$

```
%14 >> c1:= condition(n,congru(n, 2, 4)) + condition(m,congru(m, 2, 4))
                    () @ ConditionalExpression
%15 >> stdout << c1
[ n + m, {n = 2 (mod 4)} /\ {m = 2 (mod 4)}]
                    () @ OutputStream
%16 >> c2:= deduction(c1)
                    () @ ConditionalExpression
%17 >> stdout << c4
[ n + m,  {n + m = 0 (mod 4)} /\ {n = 2 (mod 4)} /\ {m = 2 (mod 4)} ]
                    () @ OutputStream
```

The `deduction` operator deduces the condition associated with an expression from the conditions associated with its sub-expressions. Operator `modulo?` solves the equation $expr \equiv x \pmod n$ in x where $expr$ is a generic expression and n a fixed integer:

```
%11 >> c1:= modulo? (n*(n+1), 2)
                () @ GeneralCondition
%12 >> stdout << c1
{(n + 1) . n = 0 (mod 2)}
                () @ OutputStream
%13 >> c2:= modulo? (n*(n+1), 3)
                () @ GeneralCondition
%14 >> stdout << c2
{{(n + 1) . n = 2 (mod 3)} \/ {(n + 1) . n = 0 (mod 3)}}
                () @ OutputStream
```

Chapter 5

Conclusions

There are various ways to handle the problem that a mathematical calculation may have more than one valid response. We can summarise the existing ways as:

1. give the user just one answer :
   - which cannot be controlled by the user (Reduce),
   - which can be guided by the user (Macsyma, Maple, Mathematica);
2. give the user some of the answers :
   - in an implicit way using `Heaviside, signum ...` functions (Maple, Derive ...),
   - in an explicit way (Axiom, Mathematica) but not in a structure which can be easily computed with.

With conditional computation, all the mathematical results are computed. Such multi-result is produced during the computation by introducing conditions on parameters. But not only can conditional expressions express several results, but they can be included in some other calculation.
Generally speaking, introducing conditional objects in Computer Algebra Systems enables complete computation with analytic, as opposed to purely algebraic objects. Then, the user knows all the solutions of his problem and how the parameters influence this result. `Conditional expressions` have proven there interest by the two applications presented in this report. But there are clearly many other areas of computer algebra which could benefit from conditional computation. One that springs to mind is the computation of limits.

Giving the different values of a computation is now possible. But the approach presented above does not allow the expression of cases when no solution can be found.
As we pointed out, we intend to modify the treatment of `ConditionalExpressions`: not only different solutions, but also point singularity must be taken into account. This leads to the introduction of a new generic value which is `"Error"`. But a more fundamental

question arises from this problem: "what is a parameter and what is a variable?" because only parameters are supposed to appear in the conditions. Some properties of conditional expression are also in question. Is it necessary to insure that all the solutions have been computed ? If the answer is yes, the completness must be automatically verified even if it is costly. Is it reasonable to have several solutions for the same value of the parameters. If it is not, then the coherency of the conditions must be checked.

The second point that must be worked on is the evaluation and the simplification of conditions. First of all, the system must be able to evaluate a condition to `True, False` in order to diminish the size of conditional expressions. We have looked at the BDD [5] point of view and it doesn't fit our needs because this approach does not take into account the links between elementary conditions such as `null (a*(a+b))` and `null (a)` or `null (a+b)` except if the system has made them explicit (as in the case of the integrator above). This method seems only interesting if a lot of independent boolean variables are involved in a conditional computation.

Bibliography

[1] K. Aberer. Combinatorial models and symbolic computation. In *Proc. of DISCO'92*, pages 116–131, 1992.

[2] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. US Government Printing Office, 1964. 10th Printing December 1972.

[3] A. Adams, H. Gottliebsen, S. Linton, and U. Martin. Automated Theorem Proving in Support of Computer Algebra: Symbolic Definite Integration as a Case Study. In *Proc. of ISSAC'99*, pages 253–260. ACM Press, 1999.

[4] D. Bleichenbacher. *Efficient and Security of Cryptosystems based on Number Theory*. PhD thesis, Swiss Federal Institute Of Technology Zurich, Zurich, 1996.

[5] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, volume 721, pages 40–45. IEEE, 1990.

[6] P. Broadbery and M. Bronstein. *A First Course on Aldor with salli*, Novembre 1998.

[7] Manuel Bronstein, William H. Burge, Timothy P. Daly, Patrizia Gianni, Johannes Grabmeier, Richard D. Jenks, Scott C. Morrison, Jonathan M. Steinbach, Robert S. Sutor, Barry M. Trager, Stephen M. Watt, and Clifton J. Williamson. *Axiom User Guide*, 1991.

[8] R.M. Corless, J.H. Davenport, D.J. Jeffrey, G. Litt, and S.M. Watt. Reasoning about the elementary functions of complex analysis. In *Proc. AISC 2000*. Springer Lecture Notes in Computer Science, 2000.

[9] R.M. Corless and D.J. Jeffrey. Well ... it isn't quite that simple. *SIGSAM Bulletin*, 26 No 3, August 1992.

[10] S. Dalmas, M. Gaëtano, and C. Huchet. A Deductive Database for Mathematical Formulae. In J. Calmet and C. Limongelli, editors, *Design and Implementation of*

Symbolic Computation Systems, International Symposium, DISCO'96, volume 1128 of *LNCS*. Springer Verlag, 1996.

[11] J.H. Davenport. On the integration of algebraic functions. In *Springer Lecture Notes in Computer Science*, volume 102. Springer Lecture, 1981. [Russian ed. MIR, Moscow, 1985].

[12] J.H. Davenport and C. Faure. The "unknown" in computer algebra. *Programming and Computer Software*, 20:1–5, 1994. Traduction from Programmirovanie.

[13] D. Duval and L. Gonzalez Vega. Dynamic evaluation and real closure. In *Proc. of IMACS'93*, pages 209–213, 1993.

[14] C. Faure. *Quelques aspects de la Simplification en Calcul Formel*. PhD thesis, Université de Nice Sophia-Antipolis, 1992.

[15] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System*. Springer-Verlag, 1992.

[16] G. Petit Bois. *A Table of Indefinite Integrals*. Dover, 1961.

[17] T. Weibel and G.H. Gonnet. An assume facility for cas, with a sample implementation for maple. In J.P. Fitch, editor, *Proc. of DISCO'92, Springer Lecture Notes in Computer Science*, volume 721, pages 95–103, 1993.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399