J. Symbolic Computation (1996) **21**, 427–439

A Distributed and Cooperative Environment for Computer Algebra

STEPHANE DALMAS[†¶], MARC GAETANO[‡‖] AND ALAIN SAUSSE[⋆⋆⋆]

[†] *INRIA, Projet SAFIR, BP 93, 06902 Sophia-Antipolis CEDEX, France*
[‡] *I3S, CNRS URA 1376, Université de Nice-Sophia Antipolis*
[⋆] *Département de Mathématiques, Université de Nice-Sophia Antipolis*

(Received 30 May 1995)

In this paper, we describe the *Central Control*, a software component that enables several symbolic systems to cooperate and exchange data. The Central Control has been designed to be the kernel of an environment for scientific computations which can offer a common and concurrent access to several tools needed by the scientist and the engineer: general purpose and specialized computer algebra systems, visualization tools, links with numerical libraries and tools to manipulate numerical programs *etc.* The user can interact with the Central Control through one or more (graphical) user interfaces. The Central Control achieves its goals by requiring as little as possible from the tools and by using a particular programming language to provide a unified view for the objects and the operations performed by the connected tools. The Central Control will be used as the basis of the *Comprehensive Solver* that will provide common access to all the programs developed within the POSSO ESPRIT/BRA project. We give a simple example of an actual use of the Central Control for computing primary decompositions of ideals.

1. Introduction

There is currently a great need to make various tools cooperate within a single environment for scientific computations. Such tools can be sophisticated user interfaces, symbolic computation systems (general purpose as well as specialized), visualization tools, tools that generate and transform numerical programs (including interfaces to numerical libraries) *etc.* The Central Control is a software component that has been designed to be the kernel of such an environment. It can offer a common and concurrent access to various tools needed by the scientist and the engineer and the possibility to distribute computing over a network of symbolic computation systems.

Many problems in computer algebra require the features of more than one particular system. For example, a general purpose system like Maple provides all the common functionalities of symbolic computation but cannot compete with specialized packages dedicated to specific topics, like Macaulay for Gröbner Basis computation or Singular

[¶] E-mail: stephane.dalmas@sophia.inria.fr
[‖] E-mail: marc.gaetano@sophia.inria.fr
[⋆⋆] E-mail: alain.sausse@sophia.inria.fr

0747–7171/96/040427 + 13 $18.00/0

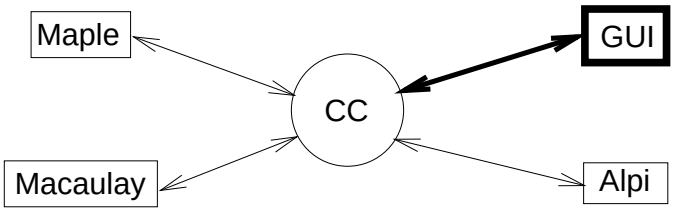

Figure 1. A typical network.

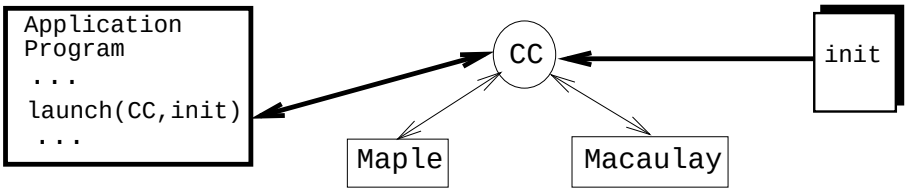

Figure 2. Using the CC from an application program.

for the study of singularities. Moreover, symbolic computation of real size problems is usually time consuming, and it could be interesting to distribute the computation over several instances of the same system.

Typically, the Central Control can be used as a communication manager to provide access to several systems through a common (graphical) user interface. This is the situation depicted in Figure 1. In this architecture, the interface (GUI) keeps the Central Control hidden from the user. It would also be possible to use an existing general purpose computer algebra system as the interface (frontend) for the CC. This would allow the general system to access more dedicated servers through the CC.

Another use can be to provide external services to an application that needs some computer algebra capabilities. Figure 2 shows a program that uses the Central Control to access the Maple and Macaulay systems. As the Central Control is programmable, the configuration of the network is read from the `init` file so that all the details of the communication can be hidden from the application program.

Another advantage of using a tool like the Central Control is the possibility to reuse existing packages with no or little extra programming. Some work is needed to provide an access through the underlying protocol for exchanging mathematical data which is currently ASAP (Dalmas *et al.*, 1994) but could be replaced by the emerging *OpenMath* standard (OpenMath, 1996) as soon it will be available.

Lastly, an architecture like the one of the Central Control allows a better organization of large applications involving several tools (not everything in one program nor on one machine).

Some systems and software architectures have been proposed in the last few years to support such an environment, such as POLYLITH (Purtillo, 1994), *CaminoReal* (Arnon *et al.*, 1988), CAS/π (Kajler, 1992) and SUI (Doleh and Wang, 1990), but our approach is quite different. None of these are really programmable in the way the Central Control is (through its command language). SUI seems to focus on the interface and is not programmable at the user level. CAS/π and POLYLITH have a "software bus" approach that renders the addition of new facilities (or access to new operations inside servers)

more difficult and not as dynamic as in the Central Control (in the Central Control architecture, you can add a new type of server "on the fly" and change any interface dynamically).

2. Architecture and Design of the Central Control

The main role of the Central Control (abbreviated as CC from here on) with respect to the tools it can manage is to provide a common and unified view of the various data they manipulate and the operations they implement. This is achieved thanks to its programming language that permits the definition of translation rules and procedures that can hide the specificities of each tool.

The CC provides the user with dynamic access to various tools ("dynamic" means that the user can run and terminate these tools from the CC as he needs them) so that the data (results) produced by one tool can be used as inputs to other tools. The "configuration process" is also transparent for a casual user. This means that all of the complexity of the CC (running programs, knowing which tool will handle a request) can be hidden from a user through the definition and use of suitable procedures and mechanisms.

As a *distributed computing environment*, the emphasis is on a structure where the interaction between tools is limited in certain ways:

1. the time spent in communications is supposed to be negligible compared to the time spent in computing (inside the programs);
2. the tools are mainly independent: they can perform their tasks without the help of other tools.

In other words, we emphasize neither efficient communication nor tight coupling of tools. This does not mean that the communications will be inefficient or that tight coupling will be impossible as we shall see later.

Basically, the CC sends a computation request to a particular tool. The tool computes and then sends back the result. This result is an object in the CC that can be treated with the full power of its programming language and sent to another tool.

In fact, a tool can ask the CC for a particular computation. Such a request is expressed as a command in the programming language of the CC. Ultimately, this allows a program to delegate all of the computations it needs to other servers through the CC. This is precisely what a user interface will need. An interface is thus treated as any other tool in our framework.

The data abstraction that we have chosen to be common to all the tools is an attributed tree (or term). An attributed tree is a tree where each operator (non-leaf node) can carry a set of pairs made of an attribute name and the associated value that can be another attributed tree. This is the most natural abstraction in our setting as most general purpose algebraic systems manipulate terms. Additional semantical information can be expressed with attributes without compromising the simplicity of the representation.

Attributed terms are the main objects in the CC as well as the objects that are transmitted between the CC and the tools. The underlying protocol used to encode these trees for interprocess communication is currently the ASAP protocol (Dalmas *et al.*, 1994). For the moment, the CC only understands the ASAP protocol to transmit the terms. We plan to use the future OpenMath (OpenMath, 1996) standard as soon as this new protocol will be ready and accepted by the community. Note that there is no

restriction in the design of the CC that will prevent the use of several protocols (one tool can use its own specific protocol).

2.1. LAZY COMPUTATION AND LAZY COMMUNICATION

As the results of the computations can be huge, it is desirable to avoid systematically transmitting results to the CC. The CC can therefore associate a "handle" for a result that can be manipulated as any other object. Only the request of specific operations on this handle should cause the effective transmission of the result. In some cases, we can even avoid transmitting the data associated with the handle, for example when a subsequent computation is addressed to the server that owns the value of the handle. This notion of lazy communication reduces the time of communication, computation (for transformation of the mathematical objects to the special form used by the interprocess communication protocol) and the space used (in the CC). This imposes only a few constraints on the server (being able to store its results) that are not mandatory (the CC can transparently work with servers that do not have the support for handles).

Since the computations performed by a tool can be long and the CC is meant to be "interactive", allowing the concurrent use of many services, it is important not to have to wait for the answer of every computation. We thus propose a mechanism of lazy computation. A computation request can return a *promise* immediately after its transmission. Promises could be manipulated as any other object in the CC. Every time an operation is requested on an object containing promises, the computation is done lazily; i.e. if the result associated to a promise is available (the server has finished its computation) it will be used, otherwise a promise is returned. The CC has a primitive operation to force the waiting of the result. This notion of promise is quite similar to the *future* construct of Multilisp (Halstead, 1985).

One interesting problem to consider (among many others) is how to handle promises that are related to aborted computations. A computation can be aborted due to a timeout or an explicit interruption from the user or a shutdown of the server. Such promises are marked "invalid" when the incident is detected and an error is returned every time their values are needed to complete a computation (there is a similar problem with handles). Note that the semantics of the CC language is such that we can easily cope with the problem of redefining identifiers referring to promises. They just go out of scope (a garbage collector is free to eventually reclaim it and interrupt the related computation).

This notion of lazy computation seems to be able to gracefully handle several problems related to concurrency inside such an environment where many services cooperate under the interactive supervision of a user.

3. Expressing Computation Requests and the Semantics of Operators

As all communications are done with attributed trees, the CC and the tools should agree on the meaning of these trees (and hence the meaning of the operators that occur). More precisely, the CC should know how to request a particular computation from a given server and what its answer means.

It is better for the user to express his or her requests in a way that is as independent as possible from the tool that will handle them. In fact, it cannot be truly system independent (as some possible partners cannot handle the desired requests) but we should

require as little knowledge as possible from the user. In practice, this means that the following conditions should be met:

1. we must be able to express exactly what is going to be computed. If the user wants to differentiate a given expression containing an expression like `factor(...)` a server should not try to evaluate (factorize) the argument of `factor` unless explicitly asked to;
2. the interpretation of the operators included in a request should be totally under the control of the user. For example, if he wants to differentiate a term containing something like `W(x)` and this `W` has no special meaning for him, the server should not impose any meaning by itself for this `W` operator.

The first condition can be easily met. Exchanging attributed terms makes it easy to express computation requests by using a particular attribute. And the user is perfectly aware of where he wants some computation to take place.

The second condition needs some kinds of conventions: assigning mathematical meanings to some operators, for example deciding that `sin` denotes the mathematical sine function. This is not always an easy task (think of the differences between the sine function in the real and complex domain...) Anyway, we must also provide a way (through attributes) to say "this has no meaning for me, so it must have no meaning for you either" (if we don't want the user to browse through all the documentation to find which operators have meanings).

3.1. TRANSLATING FOR A SERVICE

The problems described above are solved by applying a set of transformations in the CC. Each computation request expressed in the language of the CC is first transformed with respect to the head operator of the term and the service of the server (as we can have several instances of the same program managed by the CC, for example several *Maple* at the same time, we use the notion of a service). The result is then transformed with respect to a particular set of rules describing the semantics of the service before being actually sent to the server.

As an example, we work out what happens with the following computation request `let p = maple integrate(sin(x),x)` whose intent is to return the indefinite integral of the sine function.

First, a rule matching `integrate(sin(x),x)` is searched for in the `Maple` set (the name of the service associated with the `maple` server). The term is transformed accordingly: we obtain `int(sin(x),x)`. Then this term is transformed with respect to the rules in the *translation* set associated with `Maple` (this rule set is also under user control at runtime). The goal of this phase is to make the translation between the semantics of the operators within the CC and the one of the service. Here, `int` will be translated as `int`, `sin` as `sin` (as both represent the sine function) and `x` will be marked as a *dummy* variable (as it does not represent anything except a placeholder). It is important to rename the operators with no meaning in the CC because we don't want to be bothered by cases such as `integrate(sin(E),E)` when `E` has a special meaning for a server (this deals with the problems of predefined operators and words for a service).

4. The Command Language

The command language provides access to the functionalities of the Central Control. Its basic objects are: *attributed terms* (the basic data structures of the language), *servers* (external computing engines), *services* (an abstraction common to several servers) and *association tables*.

The design principles for the language of the Central Control are rather simple. Terms are the basic objects: we do not want to be bothered by evaluation rules when writing terms, so we avoid all quotation mechanisms. All evaluations (computations) of terms or subterms are requested explicitly. Computations in the Central Control are limited to conditional rule-based transformations of terms using pattern matching. All other computations are done through servers. Rules are grouped in named sets and applied explicitly (except the rules related to the "syntactic" aspects of servers). The language supports both local and global bindings of identifiers to values (including terms and servers) and a general association table facility (tables indexed by one or more terms). In addition, the language provides incremental definitions of rules and predicates (to control the application of rules), launching and termination of servers, sending interruptions to servers and manipulations of rules and rule sets (dynamically adding or deleting rules). There are also some facilities to get information about the resource usage of a server (including timings). A program in this language is a sequence of commands, commands are made up of expressions and some expressions denote terms.

Our terms are attributed trees whose leaves are (arbitrary precision) integers or byte arrays (including character strings such as `"x"`) or 0-ary operators (like `x` or `y`). A string can be used as an operator (provided it does not contain forbidden characters like white spaces) to escape the lexical conventions of the language. Byte arrays can be used to communicate efficiently (via "sub protocols") when the CC has only to act as a router between two servers.

4.1. BINDINGS

There are two ways to establish a binding between an identifier and an object in the CC. The first one is a local binding construct, introduced by the keyword `where`: `$x + f($x) where x = z + k*z^2/` binds `x` to the term `z + k*z^2` so that the value of the expression is in fact: `z + k*z^2 + f(z + k*z^2)`.

This kind of binding is local to a term. An extended form enables the simultaneous bindings of several identifiers: `$x + $y where x = y^2 and y = z^2`. Here, the two bindings occur "in parallel".

The `let` construct is used to establish a global binding between an identifier and the value of an expression. The command `let x = x + 1 and y = f(g(x))` binds two identifiers `x` and `y` to two terms. To refer to the value of `x` in an expression (a term), one writes `$x`. Here is an example of a little session:

```
CC> let x = 2; let y = $x + 2; let x = 0;
CC> $y;
    val it = 2 + 2
```

The identifier `it` is bound to the value of the last evaluated expression (when an expression is given instead of a command to the toplevel of the CC). The bindings produced by `let` or `where` are static (lexical) bindings.

4.2. ARRAYS AND VARIABLES

The CC manipulates tables that are associative collections of values (like in the `awk` language). A table is created by the `table` expression: `let tab = table`. Table elements are referenced with constructs of the form `tab[`t`]` or `tab[`$t_1, t_2, \ldots, t_n$`]` where each t_i could be any term. An element is introduced into a table with the `:=` construct, like: `tab[x] := y`. When there is no value (term) with the given indices, the symbol `null` is returned. This can be used to clear an entry (setting it to `null`).

To emulate the variables found in imperative languages, the language offers a shorthand: `x := y` is interpreted as `global[x] := y` and `deref x` as `global[x]`.

4.3. MANIPULATING SERVERS

The servers are denotable values in the language. They are created with the `create` expression. The command `let maple = createMaple(proto(ASAP), machine(psyche), command(maple))` creates a server which is an instance of the `Maple` service. This server is bound to the identifier `maple`. The arguments to the `Maple` operator are interpreted by `create` to "physically" create the connection between the server and the CC. The computations are then requested with the `compute` expression. The command `let two = compute $maple 2 + 2` returns a term representing 4. Note that the server is represented by an expression which is evaluated. Servers can thus be computed.

There is a special form for requesting "asynchronous" (lazy) computation: `let two = exec $maple 2 + 2` returns immediately a promise for the result that is bound to `two`.

4.4. RULES

Rules are the basic computation mechanism in the CC. Generally, a rule is of the form

[<rule set>`::`] <pattern> `-->` <body> [`when` <cond>]

A rule is composed of a pattern, an action and an optional condition. Rules are organized into rule sets that are named. A rule with no rule set specified is assumed to be in the `top` rule set. Rule sets can be indicated by prefixing the operator with the name of the set. Here is an example of a simple rule: `integrate(?expr, ?x) --> maple int(?expr, ?x) when symbol(?x)`.

Applying this rule will invoke a `maple` server to compute the indefinite integral of the first argument but only when the second argument satisfies the predicate `symbol`.

We are not restricted to "first order" rewriting, as the pattern variables (those beginning by the `?` character) can also denote operators as in: `map(?f, [t1, t2]) --> [?f(t1), ?f(t2)]`. The pattern variables beginning with `??` match sequences of terms.

Conditions for rules are given by predicates. Predicates are boolean-valued terms that are used to provide conditional transformations. They are combined with the classical boolean operators (`and`, `or`, `not`). Predicates are defined by the `predicate` command. Here we define a predicate `symbol` that tests the arity of its argument: `predicate symbol(?t) = arity(?t) = 0`.

Evaluation of a rule gives a term which is the value of the <body> part. This part can be an expression or a sequence of commands enclosed between `begin` and `end` (the value is then given by a `return` command).

Several rules can be entered in one rule set with the `rules for ... end` construct:

```
rules for s
 x --> 1  y --> 0
end
```

defines a rule set named `s` containing two very simple rules. Rule sets can inherit other rule sets with the `inherit` command:

```
rules for Maple
 inherit standard
 ...
end
```

Here, all the rules in the `standard` set will be used in the `Maple` set (but they will not take precedence over the explicitly given ones).

4.4.1. APPLYING A RULE

Rules are applied with the `apply` expression. For example the rule `x --> 1` can be applied by the command `let one = apply top x`. `top` is the rule set that is used here. The rule `s::x --> 1` is applied with `let one = apply s x`. In this case, `s` is the name of the rule set.

Rules and bindings are somehow similar. An expression like `let x = f(y)` is equivalent to the rule `x --> f(y)`. In the `top` rule set, `$x` can be viewed as a shorthand for `apply top x`.

4.4.2. RULES AS OBJECT

Rules can be "dynamically" added in the CC with the following construct:

`assert` < rule set >(< pattern >, < body > [, < condition >])

For example `let r = top(x, 1, true); assert $r;` is another way to add a rule in the `top` rule set. Rules can also be deleted with the `hide` command:

`hide` < rule set >(< pattern >)

For example, the expression `hide Maple(?t)` invalidates all rules in the `Maple` rule set. The rules are not really lost. The CC keeps track of the deleted ones and is able to restore them (to be able to "reinitialize").

4.4.3. EXAMPLES

Here is a more interesting example demonstrating a way to ease the creation of servers by using a table that associates a machine name to a service:

```
let default_machine = table; default_machine[maple] := ganesa;
rules for servers
 maple --> create Maple(machine(default_machine[maple]), command(maple))
 maple(?machine) --> create Maple(machine(?machine), command(maple))
end;
```

The last rule could have been protected by a condition like `when symbol(?machine)`. Another example shows how to transparently create a server:

```
rules for tmaple
?t -->  begin
         let maple = create Maple(machine(ganesa), command(maple));
         hide tmaple(current);
         assert tmaple(?t, compute $maple ?t);
         return apply tmaple ?t;
        end
end;
```

The first time a rule is applied in the `tmaple` rule set, a *Maple* server is started and the `maple` rule set is used to do the subsequent computations involving `tmaple`.

Another example shows how to implement a "best server" for the job (i.e choose the right program depending on the requested operation):

```
rules for best
 factor(??l) --> apply tmaple factor(??l)
 integrate(??l)--> apply tmacsyma integrate(??l)
end;
```

where `tmaple` and `tmacysma` are "transparent" servers as above.

Predicate commands can also make use of services (here through `tmaple` defined above). Here is a `isprime` predicate:

```
predicate isprime(?t) = apply tmaple isprime(?t);
```

5. Integrating Existing Systems

A major practical problem is the integration of existing systems in our architecture. Commercial systems are sold without source code and the sole access to their computing functionalities is through their user interface (this is untrue for some of them, as Maple and Mathematica both support some means to bypass their ordinary interface).

A general solution is to encapsulate the system within an interface (that we call a "master box") that acts as a filter, translating the requests expressed as attributed trees in the command language of the system and parsing the output to obtain the terms that will be sent back to the CC. When the user creates a service, the CC launches this interface that itself executes the system as a subprocess. A typical master box is composed of four parts devoted to:

1. Launching the program. Basically it uses the `popen` UNIX function to communicate.
2. Termination of the program. It just sends the right command (`quit` in Maple, `exit` in Macaulay).
3. Translating a request. This part does the job of "dummization" to insure that no clash will occur with reserved words and predefined operator (see 3.1). It is also responsible the assignment of results to handles.

4. Sending back the answer. A parser translates the output of the system to attributed trees that are sent back using the functions of the ASAP library. Its structure is roughly similar to that of *Mathlink* (Wolfram, 1993).

As a measure of the amount of work needed to realize such an interface, we can say that the master box corresponding to *Maple* is 550 lines of Standard ML, ML-yacc and ML-lex descriptions. There are approximately 200 lines for parsing the output of *Maple* and 350 for handling input.

This approach has been taken to grant access to *Maple* and *Macaulay* from the CC. The last connected system is *Alpi* (a system performing computation in commutative algebra, developed at the University of Pisa by Carlo Traverso). In this case, we have modified the source code to provide a direct access through the ASAP library.

6. Application

For now, our comprehensive solver is composed of *Maple*, *Alpi* and *Macaulay*. We are going to show how this distributed architecture can be applied to perform a primary ideals decomposition. We use *Maple* for its linear algebra package, *Macaulay* to compute the Groebner basis and *Alpi* to manipulate the ideals. This example is intended to illustrate a typical CC session and thus the input polynomials have been chosen to produce simple results.

Let $A = k[x, y, z]$. Let I $\subset$ A an ideal generated by the following four polynomials:

$$\begin{aligned} g_1 &= yx^2 - yx - x^2 + x \\ g_2 &= y^2z - y^2 + y^2x - yz + y - yx \\ g_3 &= xz \\ g_4 &= z^2 - z. \end{aligned}$$

We first create the needed servers:

```
CC> let map = create(Maple);
 val map=Maple
CC> let mac=create(machine(math), command(macaulay));
 val mac=Macaulay
CC> let alpi = create(Alpi);
 val alpi=Alpi
```

We define the polynomial ring and the ideal (some of the outputs have been abbreviated):

```
CC> let ring = ring([x,y,z]);
 val ring=ring(...)
CC> let g1 = y*x^2-y*x-x^2+x and g2 = y^2*z-y^2+y^2*x-y*z+y-y*x
    and g3 = x*z and g4 = z^2-z;
 ...
CC> let I = ideal([$g1,$g2,$g3,$g4]);
 val I=ideal([...])
```

We compute the codimension with *Alpi*, using the following rule:

```
CC> rules for Alpi
  codimension(?t) -->
    begin
      let hilb = exec $alpi hilbert(?t); return compute $map op($hilb,3);
    end
  end;
 val it=()
CC> let codim = apply Alpi codimension($I);
 val codim=2
```

Hence I is one-dimensional. We then eliminate the hypersurfaces of the variety generated by $g_1, \ldots g_4$ computing the GCD of the generators g_i. We use *Maple* for these GCD computations.

```
CC> rules for Maple
 gcd(?t) --> return compute $map gcd(?t) when equallength(?t, 2)
 gcd(?t) -->
  begin
    let car = compute $map op(?t, 1) and
    and cdr = compute $map op(?t,2..length(?t));
    return compute $map gcd([$car,apply Maple gcd($cdr)]);
  end
  when suplength(?t, 2)
end;
 val it=()
CC> let cf = apply Maple gcd([$g1,$g2,$g3,$g4]);
 val cf=1
```

So, there is no hypersurface. We then look at the three different projections:

$$P_{xy} = I \cap k[x, y], P_{xz} = I \cap k[x, z], P_{yz} = I \cap k[y, z].$$

```
CC> rules for Macaulay
proj(?t, ?l) -->
  begin
    let r = compute $mac ring(?l, 1) and i = compute $mac ideal(?t);
    let j = exec $mac std($i);
    return exec $mac elim($j);
  end
end;
 val it=()
CC> let Pxy = apply Macaulay proj($I,[z,x,y]);
 val Pxy=promise
CC> let Pxz = apply Macaulay proj($I,[y,x,z]);
 val Pxz=promise
CC> let Pyz = apply Macaulay proj($I,[x,y,z]);
 val Pyz=promise
```

The computation of the three projections `Pxy`, `Pxz` and `Pyz` could have been performed in

parallel if we had launched several Macaulay servers as the `Macaulay` rule `proj` ends with an `exec` call which returns immediately a promise. Then *Maple* is needed to factorize each set of generators:

```
CC> Maple::lfactor(?l) --> map(factor, ?l);
 val it=()
CC> let Pxy = $map lfactor($Pxy) and Pxz = $map lfactor($Pxz)
    and Pyz = $map lfactor($Pyz);
 ...
```

The result is then $P_{xy} = \{x(y-1)(x-1)\}, P_{xz} = \{xz, z(z-1)\}, P_{yz} = \{z(z-1)\}$.

Let us consider the two principal ideals P_{xy} and P_{yz}. We compute the quotient of I by a factor appearing in P_{xy}, then the quotient of the result by another factor, *etc.* After the factors of P_{xy}, we take the factors of P_{yz}. If the result is the whole polynomial ring, we repeat this process with the previous result. This process (in this example) allows us to get a "step by step" difference of algebraic varieties until a primary ideal is found.

```
CC> let I1 = $alpi quotient($I, y-1);
 val I1=[x*(x-1),x*z,z*(z-1),y*(z-1+x)]
CC> let I2 = $alpi quotient($I1, x-1);
 val I2=[x,y*(z-1),z*(z-1)]
CC> let I3 = $alpi quotient($I2, x);
 val I3=[1]
CC> let I3 = $alpi quotient($I2, z-1);
 val I3=[x,y,z]
CC> let I4 = $alpi quotient($I2, z);
 val I4=[x, z-1]
```

To complete the decomposition, we must compute the quotient of I by each factor appearing in P_{yz}. The quotient of this result by $y-1$, then by $x-1$, gives us the last irreducible components of I. Finally, we have the following result:

$$I = \bigcap_{i=1}^{4} I_i \qquad \text{where} \quad I_1 = (x, y, z), I_2 = (x-1, z), I_3 = (y-1, z), I_4 = (x, z-1).$$

7. Current Work

The current version of our Central Control has been successfully experimented with *Maple*, *Alpi*, *GB*† and *Macaulay*. This version is written in Standard ML and is about 3500 lines of code. It currently runs on SPARC processors running SunOS 4.x and DEC MIPS-based machines. The size of the executable is about 1 megabyte.

Besides the future connections to be established with other major scientific tools (like *Axiom*) and specialized packages (produced by the POSSO project), we plan to provide the Central Control with a more conventional programming language. We think that it will be better for the future to build such a tool on a common (standardized) existing language than use our own original language. More users should be involved and interested (and

† developed by Jean-Charles Faugère (CNRS, France).

willing to write code). Of course, this new version will use the same main concepts as the prototype (promises, handles, rewriting...). A possible solution is to embed the functionalities of the Central Control in a *Scheme* interpreter. Such an experiment is in progress based on SCM (a *Scheme* interpreter written in C and developed by Aubrey Jaffer). It should provide a more compact and portable implementation of our concepts.

Acknowledegments

This work is partially supported by the ESPRIT POSSO project (6846). The central control will be used as the heart of the so-called *comprehensive solver* which will provide a common access to the various tools developed inside the project.

References

Arnon, D., Beach, R., Mc Isaac, K., Waldspurger, C. (1988). CAMINOREAL: an Interactive Mathematical Notebook. In *EP'88 International Conference on Electronic Publishing, Document Manipulation and Typography*, Nice, France. Cambridge University Press.

Dalmas, S., Gaëtano, M., Sausse, A. (1994). ASAP: A Protocol for Symbolic Computation Systems. Technical Report 162, Institut National de Recherche en Informatique et en Automatique.

Doleh, Y., Wang, P.S. (1990). SUI : A System Independant User Interface for an Integrated Scientific Computing Environment. In *Proc. ISSAC'90*, pp. 88–94.

Halstead, R.H. (1985). Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, pp. 501–538.

Kajler, N. (1992). CAS/PI : a Portable and Extensible Interface for Computer Algebra Systems. In *Proc. ISSAC'92*, pp. 376–386. ACM Press.

OpenMath (1996). OpenMath WWW home page. `http://www.can.nl/~abbott/OpenMath/`

Purtillo, J.M. (1994). The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, **16**:151–174.

Wolfram Research (1993). *Mathlink Reference Guide.*