An environment for Symbolic and Numeric Computation

G. Dos Reis° & B. Mourrain⋆ & F. Rouillier⋆ & Ph. Trébuchet⋆

°CMLA, ENS Cachan – CNRS (UMR 8536)
61, Avenue du Président Wilson
94235 Cachan - Cedex, France

⋆ GALAAD, INRIA,
BP 93, 06902 Sophia Antipolis,
France

SPACES, INRIA,
Loria,
France

April 19, 2002

Abstract

We describe the environment for symbolic and numeric computations, called SYNAPS (Symbolic and Numeric APplicationS) and developed in C++. Its aim is to provide a coherent platform integrating many of the nowadays freely available software in scientific computing. The approach taken here is inspired by the recent paradigm of software developments called active library. In this paper, we explain the design choices of the kernel and their impact on the development of generic and efficient codes for the treatment of algebraic objects, such as vectors, matrices, univariate and multivariate polynomials. Implementation details illustrate the performance of the approach.

The objective of this paper is to describe an environment for symbolic and numeric computations, which is implemented in the library SYNAPS (Symbolic and Numeric APplicationS[1]).

The need to combine symbolic and numeric computations is ubiquitous in many problems. Starting with an exact description of the equations, in most cases, we will eventually have to compute an approximation of the solutions. Even more, in many problems the coefficients of the equations may only be known with some inaccuracy (due, for instance, to measurement errors). In this case, we are not dealing with a solely system but with a neighborhood of the "exact" system and we have to take into account the continuity of the solutions with respect to the input coefficients. This leads to new, interesting and challenging questions either from a theoretical or practical point of view at the frontier between Algebra and Analysis and witnesses the emergence of new investigations [30, 21].

The aim of the SYNAPS project is to provide a coherent platform integrating many of the nowadays freely available software in scientific computing. Connecting and configuring together these tools allow a user to get, in a single download, everything he or she needs. The kernel of this platform is based on the former library ALP [23], and provides data-structures and classes for the manipulation of basic objects, such as (dense, sparse, structured) vectors, matrices, univariate and multivariate polynomials. The aim of this paper is to describe the design of its kernel. We will also mention briefly the implementations of algorithms, which have been described in different research papers such as [18], [29], [15], [6, 3, 8, 16]. [17], [7], [12], [19], [25], [12], ...

As SYNAPS is targeting different domains of application, it has to provide various internal representations for the abstract data-types required by algorithms specifications. Thus, this library makes it possible to define parameterized but efficient data structures for fundamental algebraic objects such as vectors, matrices, monomials and polynomials... which can be used easily in the construction of more elaborated algorithms. We pay a special attention to genericity, or more precisely to parameterized types (templates in C++ parlance) which allow us to tune easily the data structure to each specific problem at hand. So called *template expression* (see §1.4) are used to guide the expansion of code during program translation and thus to get very efficient implementations. Traditional developments of libraries, in particular for scientific computations, are usually understood as a collection of "passive" routines. The approach taken here is inspired by a recent paradigm of software developments

[1] `http://www.inria.fr/galaad/logiciels/synaps/`

called *active library* [10, 31, 13] and illustrated by the library STL [27]. The software components of such libraries take active parts of the configuration and generation of codes (at compile time), which combine parameterization and efficiency. Projects like BLITZ[2], SCITL[3], MTL[4], NTL[5] successfully use those ideas. The present environment is also a step in that direction, with an orientation toward algebraic data-structures (*e.g.* polynomials).

We also carefully consider the problem of reusability of external or third-party libraries. Currently, we have connected LAPACK (Fortran library for numerical linear algebra), UMFPACK, SPARSELIB (respectively Fortran and C++ libraries for sparse matrices), SUPERLU (C library for solving sparse linear system), GMP (C library for extended arithmetics), GB[6], RS[7] ... Specialized implementations — for instance LAPACK routines — can coexist with generic ones, the choice being determined by the internal representation. We also want this library to be easy to use. Once the basic objects are defined by specialization of the parameters, including maybe the optimization of some critical routines, we want to be able to write the code of an algorithm "as it is on the paper". Tools featured by the C++ programming language to support polymorphism allows us to make a great step towards achieving the goals described above.

1 Tools for an Active Library Design

The fundamental characteristic of an active library is to provide general purpose abstractions along with domain-specific specializations, tuned to take advantage of configuration knowledge necessary to deliver optimal efficiency. The configuration knowledge of an active library must be bundled in a *uniform* and *extensible* framework. In the scope of the C++ programming language, the tools at our disposal are the type-system and its two-level aspect through the `template` support machinery.

1.1 Types Are Our Friends

As announced in the introduction, the SYNAPS framework extensively uses the *typeful programming* paradigm[4], whereby the motto is: *laws should be enforceable*. By that we mean, whenever some constraints are known to hold, we use the type-system — wherever possible — to express them[8] so that much errors can be detected very early in the development stage (*i.e.* compile-time). In doing so, the program translator (*e.g.* the compiler) can keep us from breaking hard-to-find constraints violations and save us considerable time. Another immediate benefit is that, when a program construct fails to type-check then the reasons of the failure are apparent. Last but not least, as compilers are improving, information gathered from static analysis of a program text may be profitably used by the compiler to emit efficient executable images: A popular example is the type-based alias analysis[22] used to detect whether two pointers may alias the same object or not. Template expressions (see §1.4) illustrate typical uses of the type-system to implement (in library) domain-specific optimizations not available in common-place compilers.

One feature of symbolic and numeric computations is the ubiquity of parameterized data-types and algorithms. From a numerical point of view, it is not uncommon (depending on the context) to perform the same computations with different numerical precisions – that usually implies instantiating the same algorithm for different numerical data types. From a symbolic perspective, the notion of polynomial may be defined (and actually used) for any coefficient data type that happens to meet the ring axioms. Therefore, symbolic computation naturally calls for polymorphism[5] as an effective programming tool. The C++ programming language[2] supports polymorphic data type

[2] `http://monet.uwaterloo.ca/blitz/`
[3] `http://www.acl.lanl.gov/SciTL/`
[4] `http://www.lsc.nd.edu/research/mtl/`
[5] `http://www.shoup.net/ntl/`
[6] `http://calfor.lip6.fr/jcf/Software/Gb/index.html`
[7] `http://calfor.lip6.fr/rouillie/Software/RS/index.html`
[8] Obviously, not all constraints are expressible in the type-system

and algorithms through class inheritance, virtual functions, template classes and template functions. Both programming paradigms are not exclusive — as is always the case in C++ with its supported programming styles — even though the stress on parametric polymorphism is rather recent, compared to the conventional use of inclusion polymorphism.

Inclusion polymorphism is expressed in C++ through class inheritance. Virtual functions make it possible for a derived class to *override* a particular behavior of its base class; the executing environment will use type-information to select the appropriate function while the program is running. When that happens, we speak of *dynamic binding*. Dynamic binding is an extremely flexible mecanism; it supports modular continuity. That is, using inclusion polymorphism (where appropriate), new components can be plugged into an existing environment without breaking the overall architecture, nor rewriting major components. However, inclusion polymorphism may become cumbersome or even conflicts with situations where value semantics, efficiency and static-type constraints are premium.

The `template` mechanism is the way the C++ programming language implements parametric polymorphism. By definition, it is static-checking oriented; the program-translator uses static type information to generate (or instantiate) program components which, in turn, are integrated into the program being translated. That means templates are the feature of choice where polymorphism, value semantics, execution efficiency and static constraints are discriminating criterions. Because templates operate on two levels (template classes are "type-constructors" and template functions are "function-constructors"), they embody a form of *meta-language* which has proven to be handy when it comes to use the type-system to express constraints, perform compile-time computations, or implement features (such as lazy evaluation) not directly supported by the language.

1.2 How Do We Find Our Algorithms?

A distinctive feature of the C++ programming language is the ability to name *scopes*. For a name use in a C++ program to be valid, it must be first introduced by a declaration. The scope of a declaration is the totality of the parts of the program text in which the use of the declared name, as an *unqualified name* (*i.e.* a plain identifier), is valid. There are five kinds of scopes in C++:

1. Local scope. A name declared in a block has a local scope. Names with local scope can not be accessed outside of their declarative regions.

2. Function prototype scope. Any parameter name declared in a function declaration — that is not also a function definition — has a function prototype scope. Function prototype scopes are of limited use. They are merely used to convey intentional information about each function-parameter in a forward declaration.

3. Function scope. No identifier but a label can have a function scope.

4. Class scope. Names declared in class definitions have class scope. The potential scope of a name declared in a class definition comprises not only the declarative region that contains its declaration as a member, but also the regions deliminated by the members definitions, functions definitions inside the class definition, default arguments and constructors initializers.

5. Namespace scope. Names declared outside of any of the above mentioned scopes have namespace scope; or put differently, a namespace member name has a namespace scope. The potential scope of a namespace member name starts from its point of declaration in its namespace body and extends onwards including the potential scope of each using-directive that nominates its namespace (following its point of declarations), and each of the declarative regions established by reopening its namespace for extension. The reason for this — seemingly sophisticated but actually elementary — notion of scope is due to the open nature of namespaces and the powerful mechanism of name selection and namespace composition (see below).

For the purpose of name-use validation, the program-translator has to make sure that a given name is used in conformance with the properties specified by its declaration. That, in turn, means

there ought to be a way of associating a declaration with a name. That process is what we call a *name lookup*. In general, a name used in conformant conditions must, in its scope, designate unambiguously one and only one entity. That means that, in a given scope, name lookup usually finds a single declaration for a valid name-use. Failure to do so may result in the program being rejected for:

- using undeclared name if the name lookup doesn't find any declaration matching that of the looked up name; or
- ambiguity if the name lookup finds multiple different declarations for the same name. As an exception to this rule, name lookup may find multiple function declarations for the same name. In that case the name is said to be *overloaded*; then, if the outcome of the overload resolution process still consists of multiple declarations, the program is invalid.

Understanding how name lookup works, what rules govern it, is crucial for large libraries and programs organization; even as a simple user of a third party library one might be affected as well since library implementors tend to take advantages of name lookup rules in order to provide features not possible in pre-Standard C++. There are three categories of name lookup rules: unqualified name lookup, qualified name lookup, and function argument dependent name lookup also known as Koenig lookup. Before going into the details, let's mention that name lookup rules apply uniformly to all names and that the process is ended as soon as a declaration is found in a given scope.

The rules for unqualified name lookup are rather simple. In general, a name shall have a declaration "in scope" before use (an exception to that general rule is that covered by the Koenig lookup to be discussed in the next paragraph). For that matter, declarations from namespaces nominated by using-directives are visible in the namespace enclosing the using-directives. For the name lookup machinery, names declared in namespaces nominated by using-directives are considered members of the enclosing namespace. That mechanism is known as *namespace composition*. Qualified names are used when a particular scope (a class-scope or namespace-scope) is specified to designate where the program-translator should look for a name. When the qualified name has the form `N::m`, where `N` designates a namespace, then the set of declarations of `m` comprises all declarations for `m` in `N` and in the transitive closure of all namespaces nominated by using-directives in `N` and its used namespaces, except that using-directives are ignored in any namespace directly containing one or more declaration of `m`. Since a using-declaration is a declaration, names brought from foreign namespaces through using-declarations are considered during qualified name lookup.

All the above rules are fine ..., except that they miss the way operator overloading process works. If the above rules were the only available name lookup rules then the following program would fail to compile

```
#include <iostream>
#include <string>
int main()  {
   std::string hello = "Hello, World!"; std::cout << hello << std::endl;
   return 0;
}
```

The reason is that in the expression `std::cout << hello`, the compiler infers a call for a function named `operator<<`, but a declaration for that name cannot be found, starting from the declarative region of `main()` body and using ordinary unqualified name lookup rules alone — the required function actually is defined in the standard namespace `std`. On the other hand, the (naive) tentative "`std::cout std::<< hello`" to specify the scope is no option since that syntax is not valid, and even if the language were hijacked to allow such a syntax, it won't really scale. The C++ programming language solves that problem by introducing a third group of lookup rules: When looking up an unqualified name as a function-name in a function call, namespaces associated with the type of the arguments are also examined; in those namespaces the function is looked up as a qualified name where the qualifier is the associated namespace. Koenig lookup is applicable not only to operator names but also to any function-name used in an unqualified manner. Concretely, that means that given the following:

```
namespace Our
  {
     template<typename T>  class Matrix { /* ... */ } ;
     template<typename T>  T frobenius_norm(const Matrix<T>&) { /* ... */ }
  }
int main()
{
   Our::Matrix<double> m;         // ...
   double n = frobenius_norm(m);  // ...
}
```

the function `Our::frobenius_norm` is found by Koenig lookup which considers the namespace `Our` associated with the type of the argument `m`. It is worth restating that Koenig lookup is *not* applicable for qualified names, and in its current form it only applies to function-names.

One last word about the wonderful tool that Koenig lookup is: it does not work for built-in types because the set of their associated namespaces is empty. That is, Koenig lookup works only for functions whose signatures involve user-defined types (mostly classes). Therefore, some care must be exercised when relying on argument dependent function name lookup. As an example, consider that after the definition

```
namespace Our { template<typename CoeffType> struct Polynomial { /* ... */ }; }
```

of a parameterized data-type intended to model dense univariate polynomials over some arbitrary ring, we wanted to implement the height of a polynomial $P = \sum a_k X^k$ defined as `height` $(P) = \sum |a_k|$. We assume that there is a function `abs()` defined over the coefficient ring, having the desirable properties. Then, a first take would be to write:

```
template<CoeffType> CoeffType height(const Polynomial<CoeffType>& p)
      {
         const std::size_t d = degree(p);
         CoeffType h = CoeffType();
         for (std::size_t i = 0; i <= d; ++i) h += abs(p[i]);
         return h;
      }
```

The above works well when instantiated for example with for `Your::Integer` — assuming a corresponding function `abs()` is defined in the domain `Your`. However it fails when instantiated for `int` because there is no associated namespace for built-in types. The conventional way to make the above work is to bring `std::abs` into scope either through a using-directive (not recommended) or a using-declaration (recommended):

```
template<CoeffType> CoeffType height(const Polynomial<CoeffType>& p)
      {
         using std::abs;  // make sure std::abs() is visible for built-in types
         // the rest as above...
      }
```

The act of bringing into a scope a particular name from a foreign namespace through a using-declaration is sometime referred to as *name selection*.

1.3 Classes versus Namespaces

We've seen in the previous section that there are two kinds of "nommable" scope: class-scope and namespace-scope. These scopes, while sharing some properties, work in fundamentally different ways. Whereas namespaces are "open", classes are "closed": A namespace can be "re-open" for the purpose of adding new members but classes can not, once the right-brace terminating their definitions have been seen. Another distinctive difference is the fact that namespaces cannot be used as template-arguments whereas classes can. Thus, the decision of using a class or a namespace to implement a particular instance of scope is usually guided by the intended use.

The first approach based on class-scope is used for instance in [14], or in [9]. The algorithms are enclosed in so-called kernels. A kernel is a class, containing other sub-classes representing data or function-objects or methods operating on these types. The extension of a given kernel is performed by the derivation of classes, but with some constrains.

In SYNAPS, the second approach based on namespace has been chosen, for its ease extension. Moreover, using Koenig lookup, this allows us to combine genericity and specialization, exploiting natural conventions of developments.

1.4 Template expression

Template expressions are type manipulations used to guide the compiler to produce optimized code. This technique is particularly interesting for some linear algebra operations, where intensive but independent instructions are performed. We illustrate it on the addition of vectors. Consider for instance the following instruction: `v = v1 + v2 + v3;` where `v, v1, v2, v3` are vectors (say of type `Vector`). A traditional implementation would define the operator `Vector operator+(const Vector & v1, Vector & v2);` so that the previous instruction will produce the temporary vector (say `__tmp1`) for the addition of `v1` and `v2`, another one (say `__tmp2`) for the addition of `__tmp1` and `v3` and will use the operator `Vector & Vector::operator=(const Vector &);` to assign `v` to the value of `__tmp2`, using a loop

```
for(index_type i = 0; i <v.size(); i++) v[i] = __tmp2[i];
```

In such an implementation, two temporary vectors are allocated and deleted, which is time and memory consuming, especially if the vectors are of large size.

A more optimized implementation would consist in writing the code

```
for(index_type i = 0; i <v.size(); i++) v[i] = v1[i] + v2[i] + v3[i];
```

This code can be generated directly, *at compile time*, using template expression techniques. Instead of defining the operator `+`, as before, we define it with the following prototype:

```
VAL<Op<'+',Vector,Vector> > operator+(const Vector & v1, Vector & v2);
```

where `VAL<Op<'+',Vector,Vector> >` is the type of a data-structure which contains references to the two vectors `v1, v2`, but which does not compute the sum of this two vectors. The instruction `v1+v2+v3` that we are considering, produces a data of type `VAL<Op<'+', Vector, VAL<Op<'+', Vector, Vector> > > >;` Now, the assignment `v = v1 + v2 + v3;` will involve the operator

```
template<class R> Vector & operator=(const VAL<R> & v)
```

which is implemented as `for(index_type i = 0; i <v.size(); i++) (*this)[i] = _elem(v,i);`. The key point is that the compiler will expand the call to the inline function `_elem(v,i)` into `v1[i]+v2[i]+v3[i]`, because the i^{th} element of an element `v` of type `VAL<Op<'+',...,...> >` is the sum of the i^{th} of its two components. A complete set of arithmetic unevaluated operations is available in the SYNAPS kernel, namely types of the form `OP<c,T1,T2>` with $c \in \{$`'+','-','*','.','/','%','^'`$\}$ can be used in the library. Such arithmetic operations will return an unevaluated arithmetic tree and the computation will be effectively performed only when the assignment operator will be used.

2 The Design

We have distinguished three levels of abstractions: **containers** attached to **domains**, **modules** and **views**. The next few pages are devoted to their descriptions.

2.1 Containers and domains

A container specifies the internal representation of a SYNAPS object. It provides abstractions to access, scan, create, or transform that concrete representation and thus depends closely on the data-type being modeled. Containers are implemented as classes (often parameterized by coefficient types or index types), with few functions so that they can be easily rewritten or extended. These structures are adapted to the problem they aim to solve. The algorithm will be provided by other classes or namespaces.

Our approach is in the spirit of the STL library [27], which implements basic structures such as `std::list<>`, `std::vector<>`, `std::set<>`, `std::deque<>`, ... New containers have been added, *e.g.* one-dimensional arrays `rep1d<C>` with generic coefficients, two-dimensional arrays `rep2d<C>` for dense matrices ... Domain-specific implementations, counterpart of the above generic versions, are usually found in *domains* (concretely implemented by namespaces). For instance, the container `rep1d<C>` is in namespace `linalg` and the usual way to call it, is:

```
linalg::rep1d<C>  or   using linalg::rep1d; rep1d<C>
```

Similarly, we have defined 2-dimensional containers `linalg::rep2d<C>` and `lapack::rep2d<C>`, for dense matrices. The last container is adapted to the internal representation of dense matrices in the LAPACK library, so that calling a subroutine of this library does not require to create new data. Adapted containers for the manipulations of structured matrices such as Toeplitz matrices (`linalg::toeplitz<C>`), Hankel matrices (`linalg::hankel<C>`) or sparse matrices (`linalg::sparse2d<C>`) are also available in the library.

The job of iterating over data-structures is delegated to iterators. For unidimensional containers such as `linalg::rep1d<C>`, a usual loop on such a structure can be written as:

```
for(R::iterator it=V.begin(); it !=V.end(); ++it) *it = ...
```

On a bidimensional container like `linalg::rep2d<C>`, an iterator type for the rows and the columns are also available, namely

```
R::row_type::iterator;    R::col_type::iterator;
```

In the first case, the iterator scans continuously the memory, but in the second case, it jumps by the number of rows. Notice that in this case, the types `row_type` and `col_type` are container types, which just store the addresses of the beginning and end of a memory block and which access to it through iterators.

Because namespaces are open, using this domain mechanism allows us to add easily specialized functions, and to use them, in a transparent way, in elaborated algorithms. As we will see, using the Koenig lookup mechanism described in 1.2, these specialized functions are searched in the domain of the container, if they exist. Another interesting feature of this approach, is that the functions and classes provided in a domain, can be compiled separately.

The extension of domains can be performed either by extending the namespace itself or by derivation of container classes. If we need to add a new function operating on a container, we add it directly in the domain:

```
namespace Domain { void new_function(const R & r) { /* ... */ } }
```

If we need to modify the definition of existing functions, we change the semantic of the container class `A`, by deriving a new class

```
namespace Domain { struct B : public A  {...}; }
```

and by redefining the needed functions of the domain. The specialized functions defined for `A` and which apply for `B`, will also be used through this mechanism.

2.2 Modules

A module is a collection of generic implementations which apply to a family of objects sharing common properties (such as vectors, matrices, univariate polynomials, ...). They are implemented as *namespaces* and thus can be extended easily. For instance in the module (or *namespace*) `VECTOR`, we have gathered generic implementations for vectors, which are independent of the internal representation, such as `Print`, `Norm`, ...

```
namespace VECTOR {
template<class V> ostream & Print(ostream & os, const V& v)
   {
     typename V::const_iterator it =v.begin();
     os<<"["<<*it; ++it; for( ; it !=v.end(); ++it) os<<","<< *it; os<<"]";
     return os;
   }
}
```

No constraints are imposed on the parameter `V` of the template function `Print`. This function can be applied as soon as a `const_iterator` is available in the class `V`. Thus, this function can be applied to a univariate or a even a multivariate polynomial. The output will look like a vector of coefficients or monomials, but not as is usually printed by a polynomial.

Another interesting feature of the approach, is that namespaces can be combined or extend naturally: `namespace UPOLY { using namespace VECTOR; ... }` . This yields a great flexibility in the construction of set of functions which applies for a families of objects.

The main modules of the library are `VECTOR`, `MATRIX`, `UPOLY` (univariate polynomials), `MPOLY` (multivariate polynomials).

2.3 Views

The views specify how to manipulate or to see the containers as mathematical objects and provide operations on the objects which are independent of the container. They are implemented as classes parameterized by the container type and sometimes by *trait* classes which precise the implementation. The only data available in such a class, via the member function `rep()`, is of container type.

For instance a one-dimensional array of `double` numbers can be seen as a vector, using the type

```
VectDse<double,linalg::rep1d<double> >
```

For convenience, using the facility of default template arguments, this is equivalent to the type `VectDse<double>`. Among the operations defined in this class, we have for example, the vector operators `+=, -=, +, -` and the scalar multiplications `*, *=`. But such an array can also be viewed as a univariate polynomial. In this case, we will define the following type : `UPolDse<double,linalg::rep1d<double> >` , which extends the operations given in `VectDse<double,linalg::rep1d<double> >` by adding the polynomial multiplication operators `*=, *` and changing the output operator. The implementations of these views are searched in a module In the first case, it will be the module `VECTOR` and in the second case `UPOLY`, as it is illustrated below:

```
template <class C, class R>
UPolDse<C,R> operator*(const UPolDse<C,R> & v1, const UPolDse<C,R> & v2)
{
   UPolDse<C,R> w(Degree(v1)+Degree(v2)+1,AsSize());
   using namespace UPOLY; mul(w.rep(),v1.rep(),v2.rep());
   return W;
}
```

The generic implementation of the function `mul` will be searched in the namespace `UPOLY`. This one is the naive implementation, which applies for all direct-access containers and which uses index loops. Such an approach allows genericity but also specialization as we illustrate now. In the case,

for instance, where the container of type `Domain::R` is adapted to the use of an external library and where the multiplication has already been implemented there, the connection of this multiplication function in the frame work can be done very easily. Indeed, we just have to define the function `mul` in the domain `Domain`:

```
namespace Domain { void mul(R & s, const R & r1, const R & r2);  }
```

This is the scheme that has been used to connect the external libraries LAPACK, GMP, SUPERLU, ...

Here are the main examples of view classes that are implemented in the current version of the library, `R` standing for the container type: `VectDse<C,R>` (standard vectors), `MatrDse<C,R>` (dense matrices), `MatrStr<C,R>` (structured matrices), `MatSparse<R>` (sparse matrices), `UPolDse<C,R>` (dense univariate polynomials), `UPolQuot<R>` (quotiented univariate polynomials), `Monom<C,E>` (monomials with coefficients in `C` and exponents in `E`), `MPoly<C,O,R>` (multivariate polynomials where `O` defines the order on the monomials).

Separating the internal representation from the algorithms, is the key step to construct easily views of sub-objects. Imagine for instance, that we want to assign the subvector of indices $1, 2, 3$ of `W` to the sum of two vectors `W1`, `W2`:

```
VectDse<double> V(5,"1 2 3 4"), W1(3,"0 1 0"), W2(3,"1 0 0");
V[Range(1,3)]=W1+W2;
```

This mechanism is working, thanks to the generic view of the `VectDse` and the specification of the construction `V[Range(1,3)]`. That construct returns a view `VectDse`, with a container sharing data with `V`. Now, the assignment instruction corresponds to the member function

```
template<class S> VectDse<C,R> VectDse<C,R>::operator=(const VAL<S> & w)
```

which knows how to assign a vector to another, once the direct-access operators are available. The actual expansion of this code produced by the compiler corresponds to the following loop:

```
for(unsigned i=0; i<2;++i) V[1+i]=W1[i]+W2[i];
```

2.4 Large Objects Management: Copy On Write

Because objects management in the C++ programming language defaults to *value semantics*, we have to take explicit actions to avoid copying large objects — the time spent in copying large objects may be shown to be non negligible under usual circumstances. A popular way to solve that memory management problem is to use reference counting[20] coupled with a copy-on-write strategy. Concretely, each object (of type `VectDse`, `MatrDse`, `MatrStr`, `MatrSps`, `UPolDse` or `MPol`, used as template-arguments for our containers) is attached with a counter whose purpose is to keep track of the number of references to its associated object. An assignment to a new object will increment the counter and copy the actual data's address. A modification of the shared object will induce a copy of the data leading to a new shared object with counter 0. The object's destructor either decrements the counter (if it is positive) or else effectively destroys the data, thereby freeing its storage. The overall scheme is implemented by the class `shared_object<R>`. Therefore, returning such objects by value in a function as in `VectDse<C,R> F(...) { VectDse<C,R> W; ... ; return W; }` does not incure a performance penalty. Indeed a copy and destruction of `W`, will in this case just increment or decrement a counter.

3 The algorithms

We describe briefly the algorithms, which are available in the current distribution. This one, still in progress, is distributed via a `cvs`[9] server and by `ftp`[10].

[9] cvs -d :pserver:cvs@cvs-sop.inria.fr/CVS/galaad co synaps

[10] wget http://www-sop.inria.fr/galaad/logiciels/synaps.tgz

Input/output: Streams for outputting objects in maple and latex format are available. An output stream for `vrml` format is under construction as well as a connection to `povray`. The tool for Inter Process Communication called `UDX` and developed by F. Rouillier is also linked to the environment.

Arithmetic: The extended arithmetic classes are based on the GMP and MPFR libraries, and correspond to the predefined types `ZZ, QQ, RR, RRR, CC`. A simple implementation of modular numbers is also available in the class `Z<p>`.

Linear algebra: Basic linear algebra operations such as LU, QR, Bareiss decomposition, determinant, svd, rank, eigenvalues and eigenvectors computations, FFT are available, either using the LAPACK routines for `double` number type or generic ones. In particular, the implementation of structured LU decomposition of sparse matrices [11] developed by J. Demmel, J.R. Gilbert and X.S. Li, in C for `float` and `double` number types has been extended to generic coefficients.

Resultants: A package devoted to resultant constructions including the Sylvester and Bezout matrices for univariate polynomials, the Macaulay, Bezoutian, Toric (integrating the C-library developed by I. Emiris) formulations for multivariate polynomials [17], [15], [18].

Solvers: Regarding solvers for univariate polynomials, we implement subdivision methods based on Bezier representation and Descartes rule, or on exclusion functions, iterative methods such as Sebastio a Sylva method [7]. The solver `mpsolve` developed by D. Bini and G. Firorentino is also connected to the environment. For multivariate polynomials, resultant based solvers, hiding a variable are also available. The algorithm [24, 25, 26] for solving zero-dimensional ideals, which computes first a normal form and then the eigenvalues or eigenvectors of the induced matrices of multiplications is implemented and illustrated in the next section. The extension of Weierstrass method [28] to the multivariate case is under implementation.

Geometry: A package on algebraic curves and surfaces is also under constructions, including tools for intersecting them, computing their topology or drawing them.

4 An example

We have already shown in experimentations repported for instance in [25], that genericity is compatible with efficiency. We can push further this reasonning: genericity is in fact a matter of performance. Indeed as the code is not attached to any particuliar implementation of the containers, it is possible to change them to use the most efficient implementation aviable without any code rewriting. For example, in [25] we describe an algorithm that computes a normal form system for a projective complete intersection with no zeroes at infinity. And we emphasize on the effective solving of systems comming for real world applications, thus drawing the attention on two main needs in terms of linear algebra:

- Fast Sparse LU-decomposition.
- Fast and Robust eigenvectors computations.

The SYNAPS framework allows us to use specialized implementations for these two points, namely SuperLU [11] and LAPACK [1]. Here are the results we obtained when dealing with the Katsura equations case (n is the number of variable) on an PIII 933MHz and using the double precision floating point arithmetic:

n	time normal forms	time eigenvectors	$Max(\|f_i\|)$
4	0.01s	0.01s	10^{-13}
5	0.02s	0.02s	10^{-10}
6	0.13s	0.11s	10^{-8}
7	0.54s	2.33s	10^{-8}

During the normal form computations, it is the SuperLU library which is used and then LAPACK is called for the eigenvectors computation. The so tremendous gap between $n = 6$ and $n = 7$ is due to cache miss. Beyond $n = 7$ the matrices used for eigenvectors computations do not fit entirely in the cache. Furthermore we did a port of SuperLU in C++ using the template paradigms with respect to the type of coefficients. This allowed us to use the very efficient SuperLU algorithm regardless to the type of our coefficients. Using modular arithmetic we treated the same family of equations.

n	4	5	6	7	10	11
time	0.01s	0.05s	0.17s	0.95s	256.81s	1412s

Of course we are not limited to modular arithmetic. For example the GMP link allows us to use GMP rational or GMP float number types.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling an A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992. `http://www.netlib.org/lapack/`.

[2] Stroustrup B. *The C++ Programming Language, Special Edition*. Addison-Wesley, 2000.

[3] E. Becker, J.P. Cardinal, M.F. Roy, and Z. Szafraniec. Multivariate Bezoutians, Kronecker symbol and Eisenbud-Levin formula. In L. González-Vega and T. Recio, editors, *Algorithms in Algebraic Geometry and Applications*, volume 143 of *Prog. in Math.*, pages 79–104. Birkhäuser, Basel, 1996.

[4] L. Cardelli. Typeful programming. pages 471–522. Springer-Verlag, 1991.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[6] J.-P. Cardinal. *Dualité et algorithmes itératifs pour la résolution de systèmes polynomiaux*. PhD thesis, Univ. de Rennes, 1993.

[7] J. P. Cardinal. On two iterative methods for approximating the roots of a polynomial. In J. Renegar, M. Shub, and S. Smale, editors, *Proc. AMS-SIAM Summer Seminar on Math. of Numerical Analysis, (Park City, Utah, 1995)*, volume 32 of *Lectures in Applied Math.*, pages 165–188. American Mathematical Society Press, Providence, 1996.

[8] J.P. Cardinal and B. Mourrain. Algebraic approach of residues and applications. In J. Renegar, M. Shub, and S. Smale, editors, *Proc. AMS-SIAM Summer Seminar on Math. of Numerical Analysis, (Park City, Utah, 1995)*, volume 32 of *Lectures in Applied Math.*, pages 189–210. American Mathematical Society Press, Providence, 1996.

[9] CGAL consortium. Cgal, a c++ library of geometric algorithms. `http://www-sop.inria.fr/prisme/CGAL/index.html`.

[10] K. Czarnecki, U. Eisenecker, R. Glck, D. Vandervoorde, and T. Veldhuizen. Generative Programming And Active Libraries. In *Dagstuhl Seminar on Generic Programming*, volume TBA of Lecture Notes in Computer Science, 1998.

[11] J.W. Demmel, J.R. Gilbert, and Xiaoye S. Li. *SuperLU Users' Guide*, 1997. `http://www.netlib.org/scalapack/prototype/`.

[12] O. Devillers, B. Mourrain, F. Preparata, and Ph. Trébuchet. On circular cylinders by four or five points in space. Rapport de Recherche 4195, INRIA, 2001.

[13] G. Dos Reis. Vers une nouvelle approche du calcul scientifique en c++. Rapport de Recherche 3362, INRIA, 1998.

[14] J.-G. Dumas, W. Eberly, M. Giesbrecht, E. Kaltofen D. Saunders, and G. Villard. Linbox, a c++ library for symbolic linear algebra, particularly using black box matrix methods. `http://www-sop.inria.fr/prisme/CGAL/index.html`.

[15] M. Elkadi and B. Mourrain. Some applications of bezoutians in effective algebraic geometry. Rapport de Recherche 3572, INRIA, Sophia Antipolis, 1998.

[16] M. Elkadi and B. Mourrain. Algorithms for residues and Lojasiewicz exponents. *J. of Pure and Applied Algebra*, 153:27–44, 2000.

[17] I.Z. Emiris and J.F. Canny. Efficient incremental algorithms for the sparse resultant and the mixed volume. *J. Symbolic Computation*, 20(2):117–149, 1995.

[18] I.Z. Emiris and B. Mourrain. Matrices in Elimination Theory. *J. of Symbolic Computation*, 28(1&2):3–44, 1999.

[19] O. Grellier, P. Comon, B. Mourrain, and Ph. Trébuchet. Analytical blind channel identification. *Submitted*, 2001.

[20] R. Jones and R. Lins. *Garbage Collections*. John Wiley & Sons Ltd, 1996.

[21] E. Kaltofen. Challenges of symbolic computation: May favorite open problems. *J.S.C.*, 29:891–919, 200.

[22] Mark Mitchell. Type-based alias analysis. *Dr. Dobb's Journal*, October 2000.

[23] B. Mourrain. ALP, Algèbre Linéaire pour les Polynmes, 1996-1999. Bibliothèque de structures et algorithmes C++, intervenant dans la résolution d'équations polynomiales, `http://www-sop.inria.fr/galaad/logiciels/ALP/`.

[24] B. Mourrain. A new criterion for normal form algorithms. In M. Fossorier, H. Imai, Shu Lin, and A. Poli, editors, *Proc. AAECC*, volume 1719 of *LNCS*, pages 430–443. Springer, Berlin, 1999.

[25] B. Mourrain and Ph. Trébuchet. Solving projective complete intersection faster. In C. Traverso, editor, *Proc. Intern. Symp. on Symbolic and Algebraic Computation*, pages 231–238. New-York, ACM Press., 2000.

[26] B. Mourrain and Ph. Trébuchet. Algebraic methods for numerical solving. Submitted, 2001.

[27] D.R. Musser and A. Saini. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley, 1996.

[28] O. Ruatta. A mutivariate weierstrass iterative rootfinder. In *Proc. Annual ACM Intern. Symp. on Symbolic and Algebraic Computation*, London, Ontario, 2001. ACM Press.

[29] H. J. Stetter. Eigenproblems are at the heart of polynomial system solving. *SIGSAM Bulletin*, 30(4):22–25, 1996.

[30] H.J. Stetter. Principles of numerical polynomial algebra. In *Proc. Workshop Symbolic on Symbolic-Numeric Algebra for Polynomials (SNAP-96)*, Sophia-Antipolis, France, July 1996.

[31] T. Veldhuizen and D. Cannon. Active Libaries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, 1998.