Domains and Expressions: An Interface Between Two Approaches to Computer Algebra

Cosmin E. Oancea
Computer Science Department
The University of Western Ontario
London Ontario, Canada N6A 5B7
coancea@csd.uwo.ca

Stephen M. Watt
Computer Science Department
The University of Western Ontario
London Ontario, Canada N6A 5B7
watt@csd.uwo.ca

ABSTRACT

This paper describes a method to use compiled, strongly typed Aldor domains in the interpreted, expression-oriented Maple environment. This represents a non-traditional approach to structuring computer algebra software: using an efficient, compiled language, designed for writing large complex mathematical libraries, together with a top-level system based on user-interface priorities and ease of scripting.

We examine what is required to use Aldor libraries to extend Maple in an effective and natural way. Since the computational models of Maple and Aldor differ significantly, new run-time code must implement a non-trivial semantic correspondence. Our solution allows Aldor functions to run tightly coupled to the Maple environment, able to directly and efficiently manipulate Maple data objects. We call the overall system *Alma*.

Categories and Subject Descriptors

I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems; D.2.12 [**Software Engineering**]: Interoperability; D.2.2 [**Software Engineering**]: Modules and interfaces, Software libraries; D.2.13 [**Software Engineering**]: Reusable software, Reusable libraries; D.2.11 [**Software Engineering**]: Languages—*interconnection*

General Terms

Languages, Design, Performance

1. INTRODUCTION

One of the positions held over the past two decades of mainstream computer algebra system design has been that there should be one over-arching language that serves both the end user and library developer. The idea has been if the language is good enough for end-users, it should be good enough for system developers, and otherwise it needs fixing. This has led to systems that either use modified scripting languages for their libraries (e.g. Maple), or that use modified library-building languages for their user interface (e.g. Axiom). A variant of this approach has been to build much of the mathematical support in a lower-level system implementation language, such as Lisp (e.g. with Macsyma) or C (e.g. with Mathematica). The result is that large parts of the current computer algebra systems are written in languages poorly adapted to the purpose, resulting in systems that are less flexible, less efficient and less reliable than we might wish.

This paper examines the structure required for a different approach: to write libraries in a language well-adapted to large-scale computer algebra programming, together with an environment aimed at ease of use by the general end-user.

It is not difficult to see that the style of programming for top-level problem solving and for libraries is quite different. For interactive problem solving, or for one-off scripts, it is important to be able to write commands quickly and succinctly. In this context, manipulation of some sort of general expression provides flexibility. On the other hand, to program large-scale computer algebra libraries, there are advantages to a language that allows efficient compilation, secure interfaces, and flexible code re-use. However, to achieve efficiency, safety and composibility requires more declarative structure. In this context, it is more natural to work with objects in precisely defined algebraic domains. Since libraries are used many times more than top-level scripts, programmers are more willing to provide this structure.

Extensions to computer algebra systems are not alwasys calls to larger software components; they may equally well be collections of very fast light-weight routines. We therefore look beyond the solutions offered by loosely coupled computer algebra systems, e.g. OpenMath[7] or the software bus[8]. We choose Aldor [10] as a suitable library-building language, Maple [4] as a suitable interactive environment, and we require that Aldor libraries to be tightly coupled to Maple. That is, Aldor libraries will receive and directly operate on Maple objects in the same address space.

Our solution consists of two parts: The first part allows the low-level run-time systems of Maple and Aldor to work together. It allows Aldor functions to call Maple functions and *vice versa*, and provides a protocol whereby the garbage collectors of the two systems can cooperate when structures span the two system heaps. As any low-level foreign function interface, it holds the user responsible for correct usage. This work has been reported elsewhere [9].

The second part of our solution, reported here, implements a high-level correspondence between Maple and Aldor concepts. The aim has been to bridge the semantic differences between the two environments, to allow Aldor domains to appear to the user as Maple modules, and Maple modules to appear as Aldor domains. While our semantic correspondence works both ways, in practice we are primarily interested in using Aldor libraries in the Maple environment. We use a tool to generate Aldor, C and Maple code that wraps the Aldor library exports, as well as supporting run-time support code to do dispatch and caching.

The resulting package, which we call *Alma*, allows standard Aldor libraries to be used in a standard Maple environment [4]. More precisely, Alma can be seen as a software component architecture to achieve connectivity among two computer algebra systems. It gracefully handles user-errors (type-checking), supports reflective features to describe components' types and functionalities, provides a user-oriented interface (Maple "look and feel"), and employs high-level optimizations. Thus, our approach is more challenging and quite different than previous work on low-level foreign function interfaces, and consequently the internal architecture of the proposed framework is more complex.

We present two validations of this architecture: First, we describe the mappings of the Aldor language features to Maple, and the Alma type-checking process (Section 5). Second, we present a comprehensive example, in which approximatively 1160 Aldor exports have been made available to the Maple user (Sections 2 and 7). Earlier results leading to this approach have been reported in [3] [6].

We see the following as contributions of this work:

- Aldor has been found to offer efficiencies comparable to hand-coded C++ [1]. Our approach therefore allows extension libraries to operate with efficiencies comparable to Maple kernel routines.
- These extensions are written in a high-level language, well-adapted for mathematical software. It allows the programmer to ignore lower-level details and have natural integration of dynamic components into the Maple environment.
- Aldor is designed for mathematical "programming in the large" and provides linguistic support for such concepts as generic algorithms, algebraic interface specification and enforcement, dynamic instantiation, etc. Our approach allows the Maple system to benefit from these features. Alternatives, such as `C++`, do not provide this.
- Authors of large Aldor libraries often wish to make their functionality available through a main-stream computer algebra system. Two examples are Bronstein's library for differential operators, Sumit [2], and Moreno Maza's library for triangular sets, Triade [5]. The current work makes this relatively easy.

The remainder of this paper is organized as follows: We start with an example in Section 2: we show a Maple session computing the polynomial GCD over a tower of algebraic extensions using Aldor's BasicMath library. Section 3 briefly introduces the aspects of the Maple and Aldor programming environments needed to understand Alma. Section 4 presents a high level architectural view of the Alma framework, together with an example of user-Alma interaction. Section 5 describes the Maple mapping, together with our type-checking mechanism. Section 6 describes the key ideas used in the Aldor and C mappings. Section 7 shows the implementation side of the example started in Section 2. Finally, Section 8 presents some conclusions.

2. EXAMPLE

This section presents an example where a Maple user employs the functionality of the Aldor BasicMath library to solve a mathematical problem in a way not supported natively in Maple. The BasicMath library was developed at NAG by Moreno Maza and others as part of the FRISCO project, and provides Aldor with a set of types and algorithms for computer algebra. It is an extensive library, comprising about 103700 lines of Aldor code.

```
read "mtestgcd-wrap.mpl":

# Construct polys
af1   := MapleToAldorPoly(x*y^2 - 4*y + 5*x):
af2   := MapleToAldorPoly(6*x*y - y^2 + 5):
am    := MapleToAldorPoly(x^2 + 1):

# Form triangular set and gcd by Aldor package.
trset := TriPack:-empty():
rchain:= TriPack:-regularChain(am, trset):
ggcd  := TriPack:-regularGcd(af1, af2, rchain):
ggcd  := genstep(ggcd):
ggcd  := TriPack:-reducedForm(ggcd, rchain):

# Get the GCD as a Maple expression.
AldorToMaplePoly(ggcd);

         y - x
```

Figure 1: A Maple session computing a GCD in $(R[x]/\mathbf{Sat}(m_x))[z,y]$ using the Alma framework

The Maple session presented in Figure 1 shows the Alma interface. The example computes the greatest common divisor of two polynomials in $(R[x]/\mathrm{Sat}(x^2+1))[z,y]$ by invoking the Aldor BasicMath library and using its support for regular chains. The session uses the file `mtestgcd-wrap.mpl` to act as a wrapper between the Alma system and the user. The implementation of this file is explained in Section 7, after we have described the necessary concepts.

The example first creates the Alma objects corresponding to the given Maple polynomials. The regular chain containing the polynomial m is constructed, and the greatest common divisor of $f1$ and $f2$ with respect to the regular chain is computed. Finally, the reduced form of *ggcd* is computed, and it is converted to a Maple polynomial.

The functions `empty`, `regularChain` and `regularGcd` have the interfaces exactly as exported by the Aldor library. `TriPack` is the instantiation of an automatically generated Maple module wrapper corresponding to the Aldor package `RegularTriangularSet`.

3. ASPECTS OF MAPLE AND ALDOR

This section briefly presents some of the aspects of Aldor and Maple systems that we used in our architectural design.

Maple uses a dynamically typed language that supports first class functions. Typically, functions use dynamic type tests to implement polymorphism, and name overloading is not supported. Modern versions of Maple have adopted the concept of *modules* to organize packages and libraries. A module is a first-class Maple object and provides a collection of name bindings. Some of these bindings are accessible to Maple code outside the module, after the module has been constructed; these are the *exports* of the module [4]. Figure 2 shows a Maple *module* and its use.

```
makeZp := proc( p )
   module()
      export plus;
      plus := (a,b) -> a + b mod p;
   end module:
end proc:

z5 := makeZp(5); # create the module
z5:-plus(2,4);   # add 2 and 4 mod 5.
```

Figure 2: A Maple Module and Its Use

As they are first class objects, modules can be returned by functions. A module's exported functions can reference environment variables visible at the moment of their creation (i.e. it is a closure). In Figure 2 the module returned by the `makeZp` function references `makeZp`'s parameter `p`. It exports the `plus` operation whose functionality is to add numbers modulo `p`.

Aldor is a strongly typed functional programming language with a higher order type system. The type system has two levels: domains and categories. Each value belongs to some unique type, known as its *domain*. Domains are in principle run-time values, but they belong to *type categories* which can be determined statically. Categories specify properties of domains, such as which operations they export, and are used to specify interfaces and inheritance hierarchies. The main difference between the two-level domain/category model and the single-level subclass/class model of object-oriented programming is that a domain *is an element of* a category, whereas a subclass *is a subset of* a class. This difference eliminates a number of deep problems in the definition of functions with multiple related arguments. Dependent products and mapping types are fully supported in Aldor. Generic programming is achieved through explicit parametric polymorphism, using functions which take types as parameters and which operate on values of those types, eg: `f(R:Ring,a:R,b:R):R == a*b - b*a`.
In Aldor, within a domain-valued expression, the name `%` refers to the domain being computed, is fixed-pointed, and can be used as a type name.

An example of an Aldor program is presented in Figure 3. It defines a parametrized category `Module(R)`, representing a simplified version of the mathematical category of *R-Modules*. `Module(R)` declares as exports a scalar multiplication and two conversion operations. `Polynomial` has the dependent mapping type: `(R: Ring)-> Module(R)`, taking one parameter `R`, which is a domain satisfying the `Ring` category, and produces a type belonging to the category of *R*-modules. Static analysis can use the fact that `R` provides all the operations required by `Ring`, thus allowing static resolution of names and separate compilation of parameterized modules. Names can be overloaded, and are resolved based on their static type. The first line in the Aldor code in Figure 3 makes the exports of the `SingleInteger` domain available throughout the file.

```
-- File Example.as:
import from SingleInteger;
define Module(R:Ring) : Category ==
AbelianGroup with {
   *:      (R, %) -> %;      ++ Scalar multiplication
   coerce: R      -> %;
   coerce: String -> %;
}

++ Polynomial domain over ring R
Polynomial(R: Ring) : Module(R) == add {
   (r: R) * (x: %) : %    == ...;
   coerce(r: R) : %       == ...;
   coerce(s: String) : %  == ...;
   ...
}
```

Figure 3: An Aldor Category/Domain Example

4. ALMA DESIGN

This presents an overview of Alma's design. The main ideas that guided the design are summarized below:

- Alma should automatically generate any needed (Maple, C, and Aldor) stubs, and keep the system's internals hidden from the user.
- Alma should provide a dynamic (interactive) type-checking mechanism that gracefully handles user needs, and errors.
- Alma should allow Maple to interact with Aldor components in an efficient manner, introducing only a minimal overhead cost.
- Alma should extend the Maple language only as needed, by providing mappings for foreign programming language concepts such as overloading, domains, etc.
- Alma should be simple to use, rendering a Maple "look and feel" to Aldor code.

4.1 Rationale of the Design

Figure 4 introduces the main components of the Alma architecture. The module that does the stub code generation is located inside the Aldor compiler. It receives as input an Aldor program, and generates the usual compiled binary representation of it, together with *Aldor, C, and Maple* stubs for the program's exports. Among these there may be exports that have their definition in some Aldor library.

The Maple stub becomes the interface between the user and the Alma system. It uses the functionality of the *type checking module*, in order to ensure a correct call to the Aldor library. Otherwise, if no type-checking is performed, an incorrect call on the user's part would most likely produce a low-level fatal error. The type-checking module is designed to provide useful feedback to the user in the case of an erroneous invocation. For example it will list the allowed export types for a given export name.

Once the program has reached a mature phase, one may want to eliminate the type-checking overhead. If a fast implementation is desired, the Alma code generation module is able to produce code in which no type checking is performed. The *type checking module* is implemented mostly in Maple, but it also uses Aldor run-time system enhancements (the "has" operation that tests if a given domain satisfies a given category).

Our architecture allows Maple to share a single address

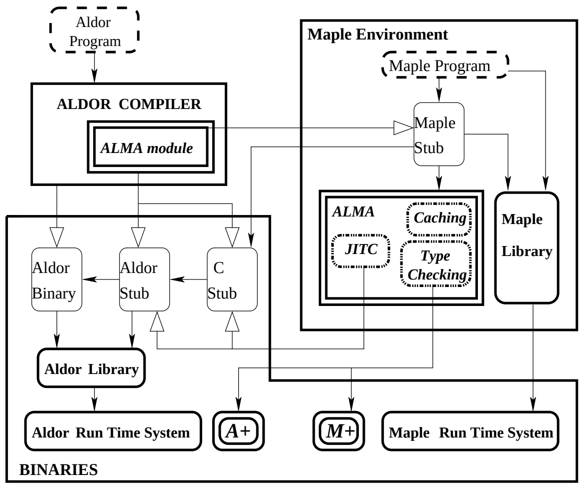

Figure 4: High-Level Architecture Overview: white arrows mean "generates," normal arrows mean "uses," dashed boxes are user source code, light boxes are generated code.

space with optimized Aldor components from a library. However, the cost of calling an Aldor function can be somewhat expensive: The initial use of the Aldor stub may require a number of expensive runtime operations, such as domain instantiation, that cannot be statically optimized by the Aldor compiler. (These operations may involve parameters that are known only at runtime.) Thus, for performance reasons, the Maple stub uses a *cache* module (implemented based on Maple's *remember* option) to store previously computed domain/category types. Aldor-closure objects corresponding to functional Aldor exports are also cached, so that they can be invoked directly, thus by-passing the Aldor stub. Alma supports function-level just in time re-compilation of the C and Aldor stubs (JITC module). More precisely, if an export is found to be "hot" and depends on type-parameters known only at run-time, Alma will build, and recompile a specialized C and Aldor stub corresponding to that export. Since most type-parameters are now instantiated, the Aldor compiler will find better opportunities for aggressive optimizations (like inlining), thus improving the application's performance.

In order to successfully complete a foreign Aldor invocation, the Maple stub calls the C stub that forwards the request to the Aldor stub, and this invokes the correct Aldor export on valid Aldor parameters, returning a value to the C stub. The C stub creates foreign Maple objects and returns them to the Maple stub. The functionality of the Aldor and Maple run-time system enhancements modules (`A+` and `M+` in the figure) is to synchronize *Maple's* and *Aldor's* garbage collectors (see [9]).

Alma's objects expose rich reflective features that can be queried by the user. This allows one to find the functionality of the corresponding Aldor component, its type, etc. Alma's foreign objects can also be manipulated in the same way as any ordinary Maple objects: They may be used in Maple operations (such as `map, apply`), while Alma's internal invocation mechanism is completely transparent to the user.

4.2 Example of Correspondence

We present a simple interaction between the Maple-user and the Alma framework. Assume the user wants to use the functionality of the Aldor code in Figure 3, and the stubs have already been generated by calling the Aldor compiler with the appropriate options on the `Example.as` file.

```
>read("MapleExampleStub.mpl"):                  # line 1
>with(Example);                                 # line 2
  module() export Polynomial, ... end module
>Polynomial("help");                            # line 3
  Domain Type: Polynomial(R: Ring) : Module(R)
  Exports:*:(R,%)->%;coerce:(R)->%;coerce:(String)->%;
  Comment: Polynomial domain over ring R
>Polynomial("help", "*");                       # line 4
  Functional Type: *: (R, %) -> %;
  Comment: Scalar multiplication
>SI_dom := SingleInteger:-Info:-asForeign;      # line 5
  ["d", 1856856, module() export ...]
>int_obj := Alma:-AldorInt(5);                  # line 6
  ["o", 5, module() export ... ]
>poly_si_dom := Polynomial(SI_dom);             # line 7
  ["d", 1848300, module export ...]
>poly_obj := poly_si_dom:-coerce(int_obj);      # line 8
  module() export ... end module
>wrong_obj:=Polynomial(SI_dom):-coerce(SI_dom);#line 9
  no function with this signature! candidates:
    coerce:(SingleInteger)->Polynomial(SingleInteger)
    coerce:(String) -> Polynomial(SingleInteger)
```

Figure 5: User-Alma interaction. Lines starting ">" are user input; the others are Maple output

The first line in Figure 5 imports the Maple stub into *Maple's* environment. On line 3, the user asks for information about the `Polynomial` domain. Alma answers by providing the type information, exports, and the comments associated with the `Polynomial` domain (see Figure 3). Similarly, on line 4, the user asks about the `*` export of the `Polynomial` domain. All Aldor domains/categories are translated into Maple modules, or functions producing modules, if parameterized. They export an `Info` module that encapsulates the type's reflective features. The `asForeign` export of the `Info` module stores a Maple foreign object corresponding to the Aldor domain it represents. At present, our implementation represents a foreign Maple object as a list that contains a classification identifier ("d" means domain, "c" means category, "f" means function, "o" means object, etc.), a pointer to the Aldor object (for primitive types this will be the value), a Maple structure representing the Aldor type, and some additional information used to synchronize the garbage collectors This is illustrated by Alma's response to the user command at lines 5, 6, 7.

Line 5 creates a *foreign domain type object* corresponding to the Aldor `SingleInteger` domain. Line 6 creates an object of type `SingleInteger` which in fact is just a primitive integer value, as one can see in the Maple representation of the `int_obj`. Next, on line 7, another domain-type object is created, corresponding to the `Polynomial(SingleInteger)` Aldor type. If the user would like to verify first that the `SingleInteger` domain satisfies the `Ring` category, he can look in the `SingleInteger:-Info:-supertypes` export. Note that the interaction with our framework is quite intuitive, as our mapping closely follows *Aldor's* specification structure and semantics. Types are run-time values both in Aldor and in our mappings: the user has to construct them first in order to use their exports. Types are also first class values, therefore they are constructed and used in the same way a regular object is used. Finally, on line 8, the `coerce` function is called, and as result, a foreign Maple object of `Polynomial(SingleInteger)` Aldor type is returned.

The last line in our example (line 9) shows how our framework reacts to an erroneous input: The type checking module detects that the parameter to the coerce export is neither of type `SingleInteger` nor of type `String`, so the incorrect Aldor library invocation is aborted. In addition, feedback is provided to the user with respect to the valid type signatures of the `coerce` function. Also note that while Maple does not support overloading, our mapping behaves as though it does. To the user it seems as though one can call two functions with the same name and with different parameter types, as they appear in the Aldor specification.

5. THE *MAPLE* STUB

We now turn our attention to the internals of the system, starting with the generated Maple stubs.

The Maple mapping addresses the issues that arise from matching the Aldor's strongly typed system with Maple's dynamically typed system. In particular, one of the challenges is in matching the compile-time parametric polymorphism of Aldor's dependent types with the dynamic polymorphism of Maple's module-producing functions. For a rich connectivity between Maple and Aldor to exist, Aldor's features, such as run-time domain types, overloading, dependent types and mapping types, need to be mapped to Maple. The key for the translation of these features is to create, via the Maple stub, dynamic types corresponding to the hierarchy of available Aldor types, and to design a dynamic type-checking mechanism for the foreign Maple objects. Alma's type-checking phase is greatly simplified, compared to the static Aldor type-checking, as it happens at the application's run-time where most parameters have completely instantiated types.

5.1 Mapping Rules

The code in Figure 6, is an extract of the Maple stub corresponding to the Aldor `Polynomial` domain defined in Figure 3. We use this to help present the high-level concepts ideas used to interface Maple with Aldor. For space reasons, we have excluded the code for the `coerce` exports, the "help" option, or some of the exports of the `Info` module.

An Aldor domain-producing function (e.g., `Polynomial`) is translated into a Maple function which at run-time yields a module. In addition it encapsulates the necessary information for type-checking its parameters and exports. This is done lines 7, 36, and 37 in Figure 6; `type/TC` is the Alma type-checker that ensures the consistency of the mapped Maple code with the Aldor type system. Aldor's nested domains are mapped into nested Maple modules. The rest of the Aldor domain exports are mapped to Maple module exports. Name overloading in our mapping is achieved by concatenating the different implementations for the same name and using a single function in which dynamic type tests identify the right code to be executed.

Modules corresponding to Aldor's domains and categories export an `Info` module containing metadata (reflective features and profiling information) associated with that type. These standardized exports of the `Info` module are computed at the domain/category-type module creation time.

Our mapping exploits Maple's support for closures. Each

```
1 `Polynomial` := proc() option remember; ## type cache
2  local ret,tmp_fct,b,ret_param,ALMA_getObject,args4;
3  if(args[1]="help") then ... return; fi;
4  args4 := args;
5  if nargs=1 then
6   b := true;
7   if b then b:=type(args[1],TC(Ring()));fi;
8   if b then ## TYPE: Module(R:Ring)
9    ret := module()
10    export `*`, Info, fcts;
11    Info := module() ##metadata:reflective+profiling
12     export GenExports,GenInfo,hash,self,asForeign,
13            type,asForeign,supertypes,printExports,
14            domArgs,domArgsOpt,optimizeOn,profile;
15     GenInfo:=["Polynomial",[["Ring"]],
16               ["Apply","Module",[args4[1]]]];
17     GenExports:=[["*",[args4[1],"%"],["%"]], ...];
18     domArgs:=args4; domArgsOpt:=[];
19     optimizeOn:=[0]; profile:=[[0]];
20    end module;
21    fcts := module()
22     export `*`, `*clos`, `coerce`, `coerceclos`;
23     local `*cstubname`;`*cstubname`:="starFrMyPolyT";
24     `*clos`:=proc(arg)option remember;#closure cache
25      local tmp_fct, ret;
26      tmp_fct:=define_external(convert(`*cstubname`,
27           symbol),'MAPLE', 'LIB="./libctestJIT.so"):
28      ret:=tmp_fct(Alma_map(lst->lst[2], [op(arg)]),
29         ["f",[args4[1,3],Info:-self],[Info:-self]]);
30      return ret;
31     end proc;
32     `*` := proc()
33      local ret, cached_clos, b, ret_param;
34      if nargs=2 then
35       b := true;
36       if b then b:=type(args[1],TC(args4[1,3]));fi;
37       if b then b:=type(args[2],TC(Info:-self));fi;
38       if b then
39        if (Info:-optimizeOn[1]=1) then
40             cached_clos:=`*clos`(domArgsOpt);
41        else cached_clos:=`*clos`(domArgs); fi;
42        Info:-profile[1,1]:=Info:-profile[1,1]+1;
43        if(Info:-profile[1,1]=Alma:-JITtreshold) then
44         `*cstubname`:="starFrMyPolySpec";
45         Info:-optimizeOn[1]:=1;
45         OptimizeAldor(Info:-self, 1); fi;
46        ret:=callAldorClosure(cached_clos,
47           map(lst->lst[2],[args]),[Info:-self]);
48        return ret;
49      fi; fi;
50      print("Context: Polynomial(R:Ring);");
51      print("Candidates: *(R,%)->(%)");
52      error "No function with this signature";
53    end proc; ... end module;
54    `*`:=fcts:-`*`;
55   end module;
56   ret:-Info:-self:=ret;
57   Alma_getObject:=proc() local tmp_fct, ret1;
58    tmp_fct:=define_external('cPolynomialOfT',
59              'MAPLE', 'LIB'="./libctestJIT.so");
60    ret1:=tmp_fct(map(lst->lst[2],[op(args4)]),
61              [ret:-Info:-self]); return ret1;
62   end proc;
63   ret:-Info:-asForeign:=`Alma_getObject`();
64   ret:-Info:-type:=Module(args[1]);
65   ret:-Info:-supertypes:=[Type]; return ret;
66 fi; fi;
67 print("Context:");
68 print("Candidate:Polynomial:(R:Ring())->Module(R)");
69 error "No function with this signature";
70end proc;
```

Figure 6: Part of MapleExampleStub.mpl

of the generated functions that produces a type will set a variable with a unique name to point to its parameters list, thus guaranteeing access to its parameters from a function declared in a nested scope. Line 36 and 37 in Figure 6 type-check the `*: (r:R, x:%)->%` export of the `Polynomial: (R:Ring)->Module(R)` parameterized domain. Notice that here `R` is a type variable, as it is given as a parameter to the `Polynomial` domain, and is used as a type in its implementation. `R` can be accessed by means of the `args4` variable in the `Polynomial`'s function outer scope. The `type(args[1],TC(args4[1,3]))` call invokes the Alma type-checker (`type/TC`) to verify if `r` is of type `R`. This uses the representation knowledge that the third element in *Aldor's domain foreign object* representation is the Maple module object that maps the corresponding Aldor domain). Both `r` and `R` are known only at run-time, and are accessible through the closure's environment.

Lines 32-53 in Figure 6 show the implementation of the `*` export of the `Polynomial` domain. Lines 34-37 verify that the number and type of the parameters are consistent with the Aldor definition. If the `optimizeOn` entry associated with the `*` export is set, then this export has already been type specialized and re-compiled (see Section 6). In this case, the `*clos` function is invoked on the non-inlined parameters (`domArgsOpt`, line 40). Otherwise it is invoked on all domain parameters (`domArgs`, line 41). The `*clos` function returns an Alma closure-object corresponding to the Aldor `*` export. This is invoked with valid parameters on line 46 by means of the `callAldorClosure` Alma's system-function. Its first parameter is the Alma closure object, the second is a list of Aldor valid arguments, while the third is the Alma type of the result. If the profiled information corresponding to the `*` export shows that it is advantageous to JIT recompile the C/Aldor stub (line 43), the Alma system-function `OptimizeAldor` is called (line 45).

In order to return the Alma closure-object corresponding to the Aldor `*` export, the `*clos` function requires access to the C stub via the *Maple's* `define_external` function. The call to `define_external` links in an externally defined function, and produces a Maple procedure that acts as an interface to this external function [4]; the `tmp_fct`, computed on line 26 of Figure 6, is such an interface. The first parameter of the `tmp_fct` is a list of Aldor parameters on which the Aldor stub is to be invoked, while the second argument (`["f",[args4[1,3],Info:-self],[Info:-self]]`) is the Alma type of the closure object (`*` receives two parameters: one of type `R` – where `R` is a type-parameter of the `Polynomial` domain, and another one of type `%`, yielding an object of type `%`).

We note that, the type/closure cache of the Alma system is easily implemented with Maple's *option remember* (lines 1 and 24).

5.2 Foreign Object Layout

We now briefly describe the Alma foreign object layout. Where necessary we shall provide details on *Aldor's* type system semantics.

In Aldor, types and functions are first class values. Therefore, besides "regular" objects, we have to define proper formats for foreign Alma type/closure objects, and to design proper Maple types for them.

Figure 8 shows the foreign object layout for the Aldor expressions: `MyCat(SI)` (category-type object), `MyDom(SI)` (domain-type object), `a` (object of type `SingleInteger`), and `fun` (function), which have all been defined in Figure 7. For objects that do not correspond to domain/category Aldor types, the third element of Alma's foreign object layout (rows 3 and 4, column 2) is their Alma type. The layout of the domain/category-type objects does not include their types, but rather themselves (see rows 1 and 2 in Figure 8). For example `MyCat(SI)` is the Alma module-type associated with the Aldor `MyCat(SingleInteger)` category. This is because their Alma type is readily accessible by means of their reflective features (the `Info:-type` export).

```
-- any Aldor domain satisfies Type
-- any Aldor category type satisfies Category
define MyCat(T:Type):Category == with; --Type: Category
MyDom(T:Type):MyCat(T) == add;         --Type: Domain
fun(A:Type, o:A, obj:MyDom(A)): A == o;--Type: Function
SI == SingleInteger; a: SI := 3::SI;   --Type: Object
```

Figure 7: Aldor Specification

Row 4 in Figure 8 shows the Alma type corresponding to the `fun` function. It is composed from a classification identifier "f", a list containing the Alma types of its parameters and another list containing the Alma types of its returns. A list whose first argument is the "l" tag (link) indicates that the parameter's type is itself passed as a parameter to this function, the remaining list's arguments giving the index in the current type where the type-parameter was introduced. The `"a"` tag stands for "apply the second argument of the set to the rest of the set's arguments." It is used only if the type expression involves a type-parameter that has not yet been computed (thus the type of the "apply" cannot be computed yet in this case).

5.3 Type Checking

Let us now consider Alma's type-checking mechanism. In Aldor, every value is a member of a *unique domain* that determines the interpretation of its data. For the current version of the language, only the domain of all domains, and the domain of all functions produce non-trivial subtype lattices [10][11]. This means that user-developed domains cannot create subtypes, the only non-trivial sub-typing lattices for our type system are the lattices of categories and functions; a non-function object is of a unique type, and cannot satisfy any other type.

To type-check that a foreign Maple object `o` is of Alma type `d` (*i.e.* a Maple module corresponding to an Aldor domain type), the Aldor type of `o` (found through the foreign object layout), and the Aldor object representation of `d` are compared, either directly or by hash codees. To verify that an Alma domain-object belongs to a certain category type, the Aldor run-time system is invoked ("has" operation) via Alma's Aldor stub.

Aldor Expr.	Associated Alma Foreign Object Layout
MyCat(SI)	["c", ptr_to_obj, MyCat(SI)]
MyDom(SI)	["d", ptr_to_obj, MyDom(SI)]
a	["o", 3, SI]
fun	["f",ptr_to_clos,f_tp] where l1:=["l",1], f_tp:=["f",[Type,l1,["a",MyDom,[l1]]],[l1]];

Figure 8: Foreign Object Layout. The Aldor expressions in the first column are defined in Figure 7

To test that a foreign Maple functional object `S1->T1` is of a functional type `S2->T2`, one has to verify that `S2` is a subtype of `S1`, and `T1` is a subtype of `T2`. When testing this, a run-time unification algorithm is used, which computes and works with the fix-point representation of a type. Otherwise for mutually recursive types the algorithm will never end.

6. THE *C* AND *ALDOR* STUBS

The role of the Aldor and C stub is to re-direct the user's call to the Aldor library. These are not necessarily accessible to the user, and do not resemble the structure of the mapped Aldor specification. The C and Aldor mappings, if not used inside our framework form un-safe code, as they assume that the type-checking has already been performed at the Maple stub level.

The C stub is the glue between the Maple and Aldor stubs, as both languages expose a basic interoperability layer with C. The C stub for the Aldor export `*:(R, %)->%` is presented in Figure 9. When invoked, the C stub (`starFrPolynomialT`) identifies the Aldor objects to be passed as parameters to the Aldor stub (`list_args`), and calls the Aldor stub (represented by `astarFrPolynomialT`) on these arguments (casted to void pointers). The resulting Aldor object (`ret`) is combined with its Alma type (`ret_type`, also received as a parameter by the C stub) to form an Alma foreign closure-object that is returned to the Maple stub.

This is accomplished through the use of the Alma system function `makeForeignObject`. The created closure-object, when called (via `callAldorClosure` Alma system-function), uses the Aldor-C interoperability layer for executing a closure call (`CCall`). The C stub generated by the JIT re-compilation module looks very much like the standard one. The only difference is that it invokes a different Aldor function (`astarFrPolynomialSpec`) which takes fewer parameters. In our case it takes no parameters as the `SingleInteger` parameter of the `Polynomial` has been already inlined in the `*` Aldor stub export generated by the JIT re-compilation module.

```
/************ C stub for *$Polynomial(T) ************/
extern FiClos astarFrPolynomialT(void* D);
ALGEB starFrPolynomialT(MKernelVector kv, ALGEB args){
 ALGEB list_args, ret_type, result; FiClos ret;
 list_args = (ALGEB)args[1];
 ret_type  = (ALGEB)args[2];
 ret = astarFrPolynomialT((void*)MapleToInteger32(kv,
                   MapleListSelect(kv,list_args,1)));
 result = makeForeignObject(kv,"f",ret,ret_type);
 return result;
}

/****** C stub for *$Polynomial(SingleInteger)  ******/
/** Code generated by the JIT re-compilation module **/
extern FiClos astarFrPolynomialSpec();
ALGEB starFrPolynomialSpec(MKernelVector kv,ALGEB args)
{
 ALGEB ret_type, result; FiClos ret;
 ret_type  = (ALGEB)args[1];
 ret = astarFrPolynomialSpec();
 result = makeForeignObject(kv,"f",ret,ret_type);
 return result;
}
```

Figure 9: C Stub Mapping

Figure 10 illustrates the main ideas employed in the Aldor stub generation. The Aldor stub exposes the parametric polymorphism of the Aldor specification/library to the Maple user who can now instantiate Aldor types at application's run-time and call their exports. A domain functional export is represented as a function that takes as parameters all the parameters of the domains in which it is nested (starting with the uppermost one), and that returns the desired closure to the C stub. All the other exports shall return an object (for example domain/category types).

```
astarFrPolynomialT(T:Ring) :
  (Ring,Polynomial(T)) -> Polynomial(T) == {
    _*$Polynomial(T);  -- (***)
}

-- Code generated by the JIT re-compilation module --
SI == SingleInteger;
astarFrPolynomialSpec() :
  (SI,Polynomial(SI)) -> Polynomial(SI) == {
    _*$Polynomial(SI);
    -- this can be aggresively optimized --
}
```

Figure 10: Aldor Stub Mapping

As can be seen, the Aldor stub is quite simple. We employ the type inference mechanism to do the difficult work of identifying which of the possible overloaded `*` functions we return. The Aldor compiler will identify more opportunities to aggressively optimize the `astarFrPolynomialSpec` Aldor stub export (generated by JIT re-compilation) – for example it can inline all the `SI` operations (`+, -, *`) that appear in the `*` export of the `Polynomial(SI)` domain.

7. EXAMPLE IMPLEMENTATION

We now show the details a library author must be aware of to use Alma. This completes the example of Section 2, which showed only the end-user's point of view.

```
-- File testgcd.as:
#include "basicmath"

N == NonNegativeInteger; R == Integer;
lv: List Symbol == [+"z",+"y",+"x"];
V == OrderedVariableList(lv);   Q == Fraction(Integer);
P == SparseMultivariatePolynomial(Q, V);
gcdPack==GcdOverTowersOfAlgebraicExtensionsPackage(lv);
T  == RegularTriangularSet(Q, lv);
VT == ValueWithRegularChain(P, T);
BB == Boolean; SI == SingleInteger;
```

Figure 11: Aldor specification used as input to the Alma framework

Figure 11 shows the Aldor specification that must be provided as input to the Alma compiler to make available *part* of BasicMath's exports to the Maple environment. The compilation generates Maple, C, and Aldor stubs that each have about 1160 exports. It is, in our opinion, easy to see why a naive, non-automatic integration of this library in the Maple environment is not a practical solution: It requires a good deal of effort, not to mention the maintenance cost. If the exports of the library are changed, the Maple mapping must be altered as well.

Figure 12 presents the hand-written wrapping code that will create the necessary Alma types and functions and will ease the use of the Alma system. This is not, strictly speaking, necessary and could be done by the end user. However it is likely that it needs to be created only once, and may be used in different programs. We underscore that the Maple and Aldor generated stubs are generic and can be instantiated over various types. The Alma user may also work with `SparseMultivariatePolynomial(R,V)`, not only with the `P` defined in Figure 11.

```
# Import from generated Aldor file.
read "mtestgcd.mpl":   with(testgcd):

# Problem-independent abbreviations.
STR  := String:  CHAR := Character:  S := Symbol:

# Wrapper for this package.
TriPack := RegularTriangularSet(Q:-Info:-asForeign,lv):
GenP    := Generator(P:-Info:-asForeign):
GenCHAR := Generator(CHAR:-Info:-asForeign):
GenVT   := Generator(VT:-Info:-asForeign):

MapleToAldorPoly :=
   almaPolyToAldor(P,SI,N,R,Q,S,CHAR,STR):
AldorToMaplePoly :=
   almaPolyToMaple(P,GenP,Q,N,R,S,STR,GenCHAR):

# Utility function.
genstep := proc(ggcd0)
   local ggcd, vgcd, str;
   ggcd := GenVT:-`step!`(ggcd0):
   vgcd := GenVT:-value(ggcd):
   str  := AlmaNewString(SI,CHAR,STR)("val"):
   VT:-apply(vgcd, str):
end:
```

Figure 12: Maple wrapper used in Figure 1

The code in Figure 12 constructs various Alma types and constants corresponding to BasicMath types/constants which will be needed in the computations already presented in Figure 1. Note that Alma has also generated Maple exports corresponding to the Aldor constants `lv, N, R, Q, T, ...` (Figure 11), and these can now be directly manipulated in the Maple file. The utility function (`genstep`) receives as parameter a generator object containing $(gcd_i, tower_i)$ pairs, obtained by calling the `TriPack:-regularGcd` function, and returns gcd_1.

The functions `almaPolyToAldor` and `almaPolyToMaple` are Alma system components that return to the user Maple to Aldor and Aldor to Maple polynomial conversion closures.

8. CONCLUSIONS

We have described an approach to using efficient, externally defined, high-level mathematical libraries within Maple. These can extend Maple in an effective and natural way. Our implementation allows Aldor domains to appear as Maple modules, and allows Aldor programs unfettered direct access to Maple objects. This allows very efficient interaction between the two environments.

At this point Alma is most useful in two settings: The first setting is to allow kernel-like efficiency in core mathematical extensions of Maple. The difference between Alma and using C code via Maple's foreign function interface is that it is possible to work at a high mathematical conceptual level and not worry about details such as garbage collection. The second setting is to allow complex Aldor packages to be used naturally from Maple. These packages typically have their own internal representation for the mathematical objects they manipulate. We forsee a third setting where Alma will be used: as an alternative for writing new libraries for Maple. New programs can work naturally with both Maple and Aldor native objects, while the Aldor compiler enforces mathematical interface requirements and generates efficient code.

Most importantly, we have re-examined one of the most basic assumptions of modern computer algebra system design: that algebra code should be written either in the top-level user language or in the low-level systems implementation language. We believe that we have demonstrated that top-level problem solving and library development can successfully use different mathematical programming languages.

9. REFERENCES

[1] L. Bernardin, B. Char, and E. Kaltofen. Symbolic computation in java: An appraisement. In *Proc. ISSAC 1999*, pages 237–244. ACM, 1999.

[2] M. Bronstein. Sum-it: A strongly-typed embeddable computer algebra library. In *Proceedings of DISCO'96, Karlsruhe.* Springer LNCS 1128, 1996.

[3] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric polymorphism for computer algebra software components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, pages 119–130. Mirton Publishing House, 2004.

[4] *Maple User Manual.* Maplesoft, a division of Waterloo Maple Inc., 2005.

[5] M. Moreno Maza. Technical report TR 4/99, on triangular decompositions of algebraic varieties. Technical report, NAG Ltd, Oxford, UK, 1999.

[6] C. Oancea and S. M. Watt. A Framework for Using Aldor Libraries with Maple. In *Actas de los Encuentros de Algebra Computacional y Aplicaciones*, pages 219–224, 2004.

[7] Special issue on OpenMath. *ACM SIGSAM Bulletin*, 34(2), June 2000.

[8] J. Purtilo. Applications of a software interconnection system in mathematical problem solving environments. In *Symposium on Symbolic and Algebraic Manipulation (SYMSAC 86)*, pages 16–23. ACM, 1986.

[9] S. M. Watt. A study in the integration of computer algebra systems: Memory management in a Maple-Aldor environment. In *Proc. International Congress of Mathematical Software*, pages 405–411, 2002.

[10] S. M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Handbook of Computer Algebra*, pages 154–160, 2003.

[11] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. *AXIOM Library Compiler User Guide.* Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.