A Framework for Using Aldor Libraries with Maple

Cosmin Oancea Stephen M. Watt

Abstract

This paper examines what is required to use Aldor libraries to extend Maple in an effective and natural way. 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. Our method involves the generation of Maple, C and Aldor stub code, uses a number of supporting modules, and relies on an enhanced foreign function interface for Maple. Since the language models of Maple and Aldor differ, the interface code implements various semantic correspondences. The Aldor functions run tightly coupled to the Maple environment, able to directly and efficiently manipulate Maple data objects. We call the overall system *Mapal*.

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. This has led to systems either that 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 is to build much of the the mathematical support in a lower-level system implementation language, such as Lisp (e.g. in Macsyma) or C (e.g. in Mathematica). This paper examines the structure required for a different approach: to use libraries written in a programming language well-adapted for computer algebra, Aldor [6], together with an environment aimed at the general end-user, Maple [2]. We assume that the functionality of the extension library may either be a collection of very fast core routines or calls to larger software components. We therefore look beyond the solutions offered by loosely coupled computer algebra systems, e.g. OpenMath[3] or the software bus[4].

Our solution consists of two parts: the first part allows the low-level run time systems of Maple and Aldor to work together. That is, it allows Aldor functions to call Maple functions and *vice versa*, and provides a protocol whereby the garbage collectors of the two systems can interact with each other to cooperate in collecting garbage when structures span two system heaps. This low-level work has been reported elsewhere [5]. The second part, reported here, implements a high-level correspondence between Maple and Aldor concepts. The aim has been to allow Aldor domains to appear to the user as Maple modules, and to bridge the semantic differences between the two environments. For this we use a tool

to generate Aldor, C and Maple code (to wrap each Aldor library) as well as supporting run-time code (to do type dispatch and caching). The resulting package, which we call *Mapal*, allows standard Aldor libraries to be used in a standard Maple environment [2].

We see the following as contributions of the approach we outline. Firstly, Aldor has been found to offer efficiencies comparable to hand-coded C++ [1]. This approach allows user extensions to operate with efficiencies comparable to Maple kernel routines. Secondly, these extensions are in a high-level language, well-adapted for mathematical software, allowing the programmer to ignore lower-level details, and well-integrated in the Maple environment. This is very different from earlier work on foreign function interfaces. Thirdly, Aldor is designed for mathematical "programming in the large" and provides linguistic support for such concepts as generic algorithms, algebraic interface specification and enforcement, etc. Finally, authors of Aldor code often wish to make their functionality available through Maple, e.g. Bronstein's library for algorithms on differential operators and Moreno Maza's library for triangular sets.

The remainder of this paper is organized as follows: Section 1 presents an architectural overview of the *Mapal* system, while Section 2 briefly describes the key ideas used in the process of inter-operating Maple with Aldor.

1 An Overview of Mapal

In order to provide a rich connectivity between the *Maple* and *Aldor* programming environments our system should: (a) allow *Maple* to interact with *Aldor* components in an efficient manner, within a small overhead cost, (b) extend the *Maple* language (by providing mappings for concepts such as overloading, domains, ..), (c) be easy to use, rendering a *Maple* "look and feel" to the user, and (d) provide an interactive type-checking mechanism that gracefully handles user needs, and errors.

Figure 1 introduces the main components of the *Mapal* architecture. The module that does the stub code generation is located inside the *Aldor* compiler. It receives as input an Aldor program, and generates a binary representation of it, together with *Aldor, C, and Maple* stubs for the program's exports (among which there may be exports that have their definition in some *Aldor* library).

The Maple stub is in fact the interface between user and the *Mapal* system, thus 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, will most likely produce an error (somewhere inside the Aldor library) that will end the current *Maple* session: a highly undesirable event. The type-checking module is designed to provide feedback to the user in the case of an erroneous invocation, that will help him to correct his program (for example it will list the allowed export types for a given export name). If a fast implementation is desired the *Mapal* code generation module is able to produce code in which no type checking is performed (once the program has reached a mature phase, one may want to eliminate the type-checking overhead). The *type checking module* is mostly implemented in *Maple*. However, as Figure 1 shows, it consists of code written in all three

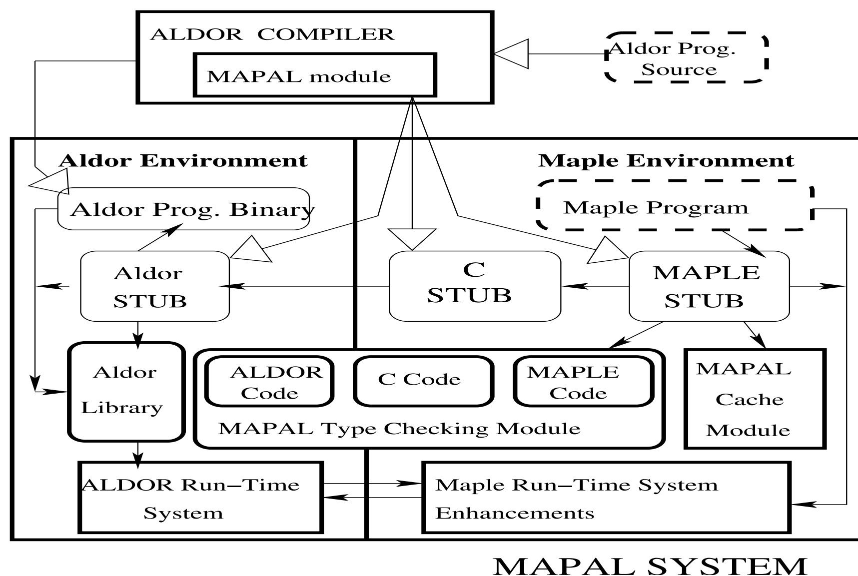

Figure 1: High-Level Architecture Overview: white arrows mean "generates," normal arrows mean "uses," dashed boxes are user source code, light boxes are generated code

languages (*Maple, C, and Aldor*) as it uses some of Aldor's runtime system functionality (the "has" operation that tests if a given domain satisfies a given category).

Our architecture allows *Maple* to share a single process space with highly optimized *Aldor* components from a library. However, the cost of calling such a code can be quite expensive, as the *Aldor* stub may require a number of expensive runtime operations (like domain instantiation) that cannot be statically optimized by the Aldor compiler (as this operations may involve parameters that are known only at runtime). Thus, for performance reasons, the *Maple* stub is using a *cache module* to remember previously computed domain/category types. Also, it is returning to the user Aldor-closure objects that can then be invoked directly, thus by-passing the *Aldor* stub.

The *C* stub is the glue between the *Maple* and *Aldor* environments. In order to successfully complete a foreign *Aldor* invocation, the *Maple* stub will call the *C* stub that will forward the request to the *Aldor* stub, and finally this one will invoke the correct *Aldor* export with valid *Aldor* parameters and return a value to the *C* stub. The *C* stub will create foreign *Maple* object(s) and return it/them to the *Maple* stub. The functionality of the "maple run-time system module", that appears in the figure is to synchronize *Maple's* and *Aldor's* garbage collectors (this work is presented elsewhere [5]).

In order to use our architecture, the *Maple*-user is required to understand the new programming language concepts introduced into the *Maple* world by our mapping mechanism (overloading, domain types, etc...). The information of how to use the *Aldor* components is provided in a *Maple*-like fashion, while the internal invocation mechanism is completely transparent for the user.

2 Toward a Rich Maple-Aldor Interoperability

Aldor is a strongly typed functional programming language with a higher order type system. The type system has two levels: 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 can specify properties of domains such as which operations they export, and are used to specify interfaces and inheritance hierarchies. In Figure 2 (B), `Module(R:Ring)` is a parameterized category representing the mathematical category of *R-Modules*. The `Poly` domain is an element of the `Module(R)` category, having the dependent mapping type: `(R:Ring) -> Module(R)` (polynomials with coefficients in `R`). In Aldor, within a domain-valued expression, the name `%` refers to the domain name being computed, is fixed-pointed, and can be used as a type name. 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, e.g. `f(R:Ring,a:R,b:R):R == a*b - b*a`.

The code in Figure 2 helps in presenting the high-level conceptual ideas used for interfacing Maple with Aldor. For a rich connectivity between the two considered languages to exist, Aldor's features like run-time domain types, overloading, dependent types, mapping types need to be mapped at the Maple level (we are thus performing implicitly a language extension). 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.

A *module* is a first-class Maple expression 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 [2]. An Aldor domain (`Poly`) is translated into a Maple function producing at run-time a module, that in addition encapsulates whatever information is necessary for type-checking its parameters and exports (see lines 7, 9 in Figure 2; `type/TC` is the *Mapal's* type checker that ensures the consistency of the mapped Maple code with the Aldor type system). Name overloading in our mapping is achieved by concatenating the different implementations sharing the same name inside a single function, in which dynamic type tests identify the right code to be executed (see lines 8, 10). Dependent types are naturally handled by Mapal's type-checking mechanism.

As can be seen in Figure 2 (C) the Aldor stub is quite simple, as Aldor's type inference mechanism is employed to identify the correct export to be returned.

The last thing to be mentioned is a summary of the *Mapal's* type-checking mechanism. To type-check that a foreign Maple object `o` is of type `d`, where `d` is a maple module corresponding to an Aldor domain type, the Aldor type of `o` (can be found through the foreign object layout), and the Aldor object representation of `d` are compared. To type-check that a Maple domain object belongs to a certain category type, the Aldor's run-time system is invoked ("has" operation) via the Mapal's Aldor stub. 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

```
1. Poly := proc()    ##(A) Maple stub      -- (B) Aldor code
2.  local ret, args1; args1 = [args];      define Module(R:Ring) : Category
3.  if(type(args[1], TC(Ring()))) then               == Ring with {
4.   ret := module() export `*`, self;        coerce: (String) -> %;
5.    `coerce` := proc()                      coerce: (R) -> %;        -- ...
6.     if(nargs=1) then                    }
7.      if(type(args[1], TC(args1[1,3])))  Poly(R: Ring) : Module(R) == add {
8.       then ## ... implem.                  coerce(s: String) : % == ...;
9.      elif(type(args[1], TC(String())))     coerce(r: R) : % == ...; -- ...
10.      then ## ... implem.               }
11.     fi; fi; error("No Such funct!");
12.   end proc; end module:                -- (C) Aldor stub for coerce:(R)->%
13.  ret:-self := ret; ## ...              acoerce...(R:Ring):(R)->Poly(R) ==
14. fi; error("No Such funct"); end proc:     coerce$Poly(R);
```

Figure 2: (A)/(C) Part of Generated Maple/Aldor Stub, (B) Aldor Code

(otherwise for mutually recursive types the algorithm will never end). The only non-trivial sub-typing lattices for our type system are the lattices of categories and functions.

Figure 3 shows a user-*Mapal* interaction example. Domain 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. They are also first class values (can be passed as parameter or be returned by a function). Line `1` creates a foreign Maple object corresponding to the `SingleInteger` *Aldor* domain. Line `2` creates a foreign Maple object for the integer `5`; its *Mapal's* type is the `SingleInteger()` module. Line `3` creates the *Mapal's* type corresponding to the *Aldor's* `Poly(SingleInteger)` domain. Finally, on line `4`, the `coerce` function is called, and as result, a list composed from a foreign Maple object of `Poly(SingleInteger)` *Aldor* type, and a closure object corresponding to the `coerce` *Aldor* function is returned (we are interested only in the foreign object therefor `[1]` in line marked with `(*)`). Line `5` shows how our framework is reacting to an erroneous input. The type checking module will detect that the parameter to the `coerce` export is neither of type `SingleInteger`, nor of type `String`, and in consequence the *Aldor* library invocation (that otherwise will end the Maple session) will be skipped. Feedback is provided to the user with respect to the valid type signatures of the `coerce` function.

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. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 9 Advanced Programming Guide*. Maplesoft, 2003.

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

```
 1. > SI_dom := SingleInteger():-asForeign;
        [‘‘d’’, 145930296, module() export ...]
 2. > si_obj := MAPAL:-AldorInt(5);
        [‘‘o’’, 5, module() export ... ]
 3. > poly_si_dom := Poly(SI_dom);
        module() export ... end module
 4. > poly_obj :=  poly_si_dom:-coerce(si_obj)[1]; ##(*)
        [‘‘o’’, 145939408, module() export ...]
 5. > wrong_obj := poly_si_dom:-coerce(SI_dom);
        no function with this signature! candidates:
        coerce:(SingleInteger)->Poly(SingleInteger)
        coerce:(String) -> Poly(SingleInteger)
```

Figure 3: User-*Mapal* interaction. The not-numbered lines shows the *Maple's* output for the executed instructions. The layout for the foreign Maple object is a set containing a classification identifier ("o"-object, "d"-domain, "c"-category), a pointer to the Aldor object (or if a basic type a value), and a *Mapal* type.

[4] 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.

[5] 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.

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

Authors Information

[1] Cosmin Oancea, University of Western Ontario, London N6A 5B7 Canada, coancea@csd.uwo.ca

[2] Stephen M. Watt, University of Western Ontario, London N6A 5B7 Canada, watt@csd.uwo.ca