Alma User Guide

Cosmin Oancea and Stephen M. Watt

Ontario Research Centre for Computer Algebra
Department of Computer Science
University of Western Ontario
London Ontario, Canada N6A 5B7

Abstract. The *Alma* framework allows Aldor and Maple to be used together. This can be used to access safe and efficient mathematical libraries, written in Aldor, from Maple. A high-level mapping is provided between the concepts of Aldor and the concepts of Maple, allowing, Aldor domains to be used as Maple modules. We describe how to install and use *Alma*.

1 Introduction

This document provides instructions to help the user to install and use *Alma*, a framework that allows Aldor and Maple to be used together.

The ideas behind *Alma* have been described earlier. The most relevant article is "Domains and Expressions: An Interface between Two Approaches to Computer Algebra" [4]. That paper presents a high-level correspondence between Maple and Aldor concepts that bridges the semantic differences between the two environments. It introduces the *Alma* framework, describes its architecture, the *C, Aldor and Maple* mappings, and shows some interaction examples. The paper "A Study in the Integration of Computer Algebra Systems: Memory Management in a Maple-Aldor Environment" [5] investigates what is required to make the low-level run-time systems of Maple and Aldor to work together. It proposes a foreign function interface between the two languages and a protocol whereby the garbage collectors of the two systems can cooperate when structures span the two system heaps. The aim is that the *Alma* framework should work on top of this enhanced foreign function interface. Other related papers describe earlier results [3] or synthesize experiments of supporting parametric polymorphism across language boundaries [1]. These documents can be found in the *AlmaArchFolder/Doc/RelevantArticles* folder.

This guide is organized as follows: Section 2 describes what is required to attach the *Alma* code generator to the Aldor compiler. Section 3 guides the user in buiding and running an *Alma* application. Finally, Section 4 looks into the Maple mapping *Alma* generates. This is the interface between the user and the application, and consequently, the mapping ideas and the structure of the "reflective" features need to be well understood.

2 Installation

We assume you have already downloaded the *AlmaArchFolder.tar.gz* file and uncompressed it:

```
$ tar -zxvf AlmaArchFolder.tar.gz
```

To install *Alma* you need to take the following steps:

1. Install Maple [2] (www.maplesoft.com).
2. Install Aldor [6, 7] (www.aldor.org).
3. If you are using a version of Aldor *before* version 1.1, you must apply a bug fix to the compiler, as described in Appendix A.
4. Integrate the Alma code generator into the Aldor compiler, as described below.

To integrate the *Alma* code generator into the Aldor compiler:

- Copy the *AlmaArchFolder/src/MapleInterop* folder at the location that contains the Aldor compiler sources:

  ```
  $ cp -r AlmaArchFolder/src/MapleInterop $ALDOR_COMPILER/src/
  ```

- Copy the *AlmaArchFolder/src/maplegen.h* file at the location that contains the Aldor compiler sources:

  ```
  $ cp AlmaArchFolder/src/maplegen.h $ALDOR_COMPILER/src/
  ```

- Search the *AlmaArchFolder/src/emit.c* file to find the commented lines that contain the *ALMA* word:

  ```
  $ grep "ALMA" AlmaArchFolder/src/emit.c
  ```

  Follow the comments and do the proper modifications in the *emit.c* file of your Aldor compiler sources. In principle, you need to include the *maplegen.h* file (`#include "maplegen.h"`) and to add the call:

  ```
  genMapleStub(finfo, symes, macs);
  ```

  as the first executable instruction of the `emitTheSymbolExpr` function.

- Rebuild the Aldor compiler: `src$ make`
- Copy the folder *AlmaArchFolder/src/AlmaBase* somewhere. This folder contains the mappings for some very common Aldor domains such as `String`, `Character`, `Integer`, together with some "static" functionality of the *Alma* framework, such as system functions to construct basic type values: integer, strings, characters, polynomials. We shall assume the `$ALMA_BASE` environment variable points to that location. Compile the *amaldorbase.as* file:

  ```
  $ cd $ALMA_BASE; aldor -Fo amaldorbase.as}
  ```

  This will generate the *amaldorbase.o* file.

3 Building an Example Alma Application

The Maple worksheet *AlmaArchFolder/Tests/ComplexEg1/testingGcd.mw* computes the greatest common divisor of two polynomials in the presence of regular chains. In order to achieve this, it accesses the functionality of Moreno Maza's library for triangular sets decomposition (part of Aldor's BasicMath library). This section describes the steps involved in buiding this *Alma* application.

- `$ cd AlmaArchFolder/Tests/ComplexEg1/`
- Generate the Alma Aldor, C and Maple stubs that encapsulate the BasicMath library exports found when parsing the `testgcd.as` file by running

  ```
  $ aldor -Fasy -Y$BMROOT/lib -lbasicmath testgcd.as
  ```

  where `$BMROOT` is an environment variable that points to the Aldor BasicMath library instalation directory.
- Open the `atestgcd.as` file for editing. Search for `QuotientFieldCategory`. Replace
  `QuotientFieldCategory(D)` with `QuotientFieldCategory(MALDOR0_R)`
  `QuotientFieldCategory0(D)` with `QuotientFieldCategory0(MALDOR0_R)`
  `CanonicalInjection(S)` with `CanonicalInjection(RationalNumber)`.
  There are three places where you have to perform the substitutions above.
- Compile the Aldor stub:

  ```
  $ aldor -Fo atestgcd.as
  ```
- Compile the C stub:

  ```
  $ cc -I$ALDORROOT/include -I$MAPLE/extern/include -I$ALMA_BASE} \
     -c ctestgcd.c} \newline
  ```

  where `$ALDORROOT` is the environment variable that points to the directory where Aldor is installed (see Aldor documentation), and similar for `$MAPLE`. `$ALMA_BASE` has been defined in Section 2 (last paragraph/bullet).
- Link the Aldor and C stub together with the "static" Alma functionality:

  ```
  $ cc -shared $ALMA_BASE/amaldorbase.o ctestgcd.o atestgcd.o}   \
     -olibctestgcd.so -L$MAPLE/bin.IBM_INTEL_LINUX -L$ALDORROOT/lib \
     -L$BMROOT/lib/ -lmaplec -lbasicmath -laldor -lfoam -lm
  ```

  This will create the `libctestgcd.so` library.
- Open the *testingGcd.mw* worksheet: `$ xmaple testingGcd.mw&`. The worksheet uses the helper file *mtestgcdwrap.mpl*. See [4] for details. In *mtestgcdwrap.mpl* modify the `ALMA_library` variable to point to the right location. Also read the static *Alma* Maple wrapper from *$ALMA_BASE* (line 6). Now you can run the worksheet.

4 Alma bindings for Maple

This section discusses some of the ideas involved in the Maple mapping. We assume your current directory is the one in which you have uncompressed the *AlmaArchFolder.tar.gz* file. Type:

```
$ cd  AlmaArchFolder/Tests/SimpleEg1
$ aldor -Fasy -Y$BMROOT/lib -lbasicmath MapleTest.as
```

This will generate three files: *aMapleTest.as*, *cMapleTest.c* and *mMapleTest.mpl*. These are the Aldor, C and Maple Alma-mappings corresponding to the *MapleTest.as* input file. The C and Aldor mappings are straight-forward. An Aldor function `f` is mapped to a function that returns a pointer to the `f` function. An Aldor domain/category, or a domain/category-producing function, or a constant is mapped to a function that when called will return the category/domain or constant as a pointer. The C stub is responsible to call the Aldor wrapper function. If the export is a function then it also gets the closure of the desired function and call it through the Aldor-C low-level foreign function interface. The remainder of this section concentrates on the structure of the Maple stub (*mMapleTest.mpl*).

Start by compiling the *MapleTest.as* file:

```
$ aldor -Fasy -Y$BMROOT/lib -lbasicmath MapleTest.as
```

Take a look at the generated *mMapleTest.mpl* file, which represents the Maple stub. The Maple-mapping is hierarchical: The `MapleTest` module encapsulates the `ModuleA`, `PolynomialA`, `MyCat1`, `MyDom1` exports, which are themselfs modules or functions that produce modules. The `PolynomialA(...)` and `MyDom1` modules correspond to the Aldor domains with the same names. Applying the `PolynomialA` function on a valid coeffcient ring generates a module that exports at its turn the functionality defined in the Aldor input file *MapleTest.as*: the `Q` constant of type `MyCat1`, four functions named `coerce` and one named `ff`.

The mapping separates the exports of a domain into *constants*, which are placed in the `CTS` module, and *functionals*, which are placed in the `FCTS` module (see the `PolynomialA(...)` module). The same is true for the outermost module, in our case `MapleTest`. This separation ensures that no naming problems appear. For example in Aldor it is legal to have a function and a constant sharing the same name, but if not for this separation, the Maple names would clash. However, except as we have mentioned, all the names are exported by the given module and they do not have to be directed through the `CTS`/`FCTS` modules: `PolynomialA:-ff` and `PolynomialA:-FCTS:-ff` are identical and the same holds for `PolynomialA:-Q` and `PolynomialA:-CTS:-Q`.

Look at the implementation of the `coerce` export of the `PolynomialA:-FCTS` module. If the first argument is the string `"AlmaHelp"` then the function will just display static information related to what this export does and how it is supposed to be used (see Figure 1). Otherwise the body of the `coerce` export is composed from four `if` expressions. Each corresponds to one of the `coerce`

```
print("Context: PolynomialA(MALDORO__R : Ring);  ");
print("Candidate: coerce: (MALDORO__R) -> (PolynomialA(MALDORO__R))");
print("Comment:  coerces an element of the ring to a poly");
print("Candidate: coerce: (String) -> (PolynomialA(MALDORO__R))");
print("Comment:  coerces a String element to a poly");
print("Candidate: coerce: (Integer) -> (PolynomialA(MALDORO__R))");
print("Comment:  coerces an Integer element to a poly");
print("Candidate: coerce: (MALDORO__R) ->
                          (MALDORO__R, PolynomialA(MALDORO__R))");
print("Comment:  receives as arg an elem of the ring
                 and returns an elem of the ring and a poly");
```

Fig. 1. Helper information for the `MapleTest:-PolynomialA:-coerce` export

exports of the `PolynomialA` Aldor domain. This is how Aldor's overloading is supported in Alma: by concatenating the implementations and employing run-time tests to determine which is the correct code to be executed. The expression `type(args[1], maldorTypeCheck(String))` will return true if and only if the first argument of the function has the *Alma*-type `String`. Similarly `type(args[1], maldorTypeCheck(args4[1, 3] ))` will return true if and only if the first argument of the function is a value of the ring type of the `PolynomialA` domain (the parameter of the `PolynomialA` function).

Look now at the signatures of the first and last `coerce` exports of the `PolynomialA` Aldor domain. They are identical if excluding the return types. This is of course legal in Aldor since the type-inference will disambiguate the call. However, in order to support this in our Maple mapping, we ask the user to send an extra parameter in these cases: a list containing the return types of the desired function. For example, if the test `type(["t", args[2, 1]], maldorTypeCheck(Info:-ALMA_self))` succeeds the first `coerce` function will be called: the one that returns a value of `PolynomialA(R)` Alma-type.

Figure 2 shows the Maple code that calls the C stub. The `Alma_getClosure` call returns the Alma-object corresponding to the current function. The first parameter is the name of the C function that achieves this, the second is the domain parameters, the third and fourth parameters indicate the indexes in the `Info:-ALMA_GenExports` export where the meta-information corresponding to this function is to be found. The intent was that the `Alma_getClosure` function should use the Maple support for caching (`option remember`), such that the C and Aldor mapping involved in returning the closure of this function are called only once. (The Aldor-Maple garbage collector synchronization is not yet integrated in the Alma framework, so the closures are not cached yet.) The `Info:-ALMA_GenExports` array is filled lazily, as needed, through a call to `Info:-ALMA_printExports[ind_fc, ind]()`. This is because Aldor do-

```
cached_clos := `Alma_getClosure`(Info:-ALMA_GenExports[2, 2, 5],
                                Info:-ALMA_GenExports[2, 2, 9], 2, 2);

tmp_fct := define_external('coerce_3PolynomialAALMA13',
                           'MAPLE',LIB="./libcMapleTest.so"):

ret := tmp_fct( cached_clos[2],
        write_ret_MALDOR_map(lst->lst[2], [args]), [Info:-ALMA_self] );
```

Fig. 2. Maple mapping calling the C stub

mains may encapsulate hundreds of exports, but the Maple user will probably only use a few of them.

The usage of the export meta-information array `Info:-ALMA_GenExports` and of the `Alma_getClosure` function allows some flexibility: for example method-based recompilation with type-inlining, and incremental export-information filling. The first argument uniquely identifies the desired closure-object, so the Maple `option remember` will always return the correct closure object.

The `define_external` call in Figure 2 returns an interface to the C function that handles the closure invocation (`tmp_fct`). Finaly the `tmp_fct` is called: the first argument is the pointer to the Aldor closure, then the Aldor objects on which the closure is to be called, and finally the Alma-type for the return(s). Note that the index 2 in an Alma-object is exactly the Aldor pointer/object (see [4]).

Method level recompilation and closure caching are two features that are not supported in the current implementation. The first can be serviced at the Maple level by static code, but since we have not yet found an example where this yields a speed-up, we left it un-implemented. The second is easy to implement: just add `option remember` to all `Alma_getClosure` functions as soon as the Maple-Aldor garbage collectors are synchronized.

One may reduce the size of the C stub by shrinking down the number of C functions that call Aldor closures. In principle you need one function for distinct pair of arguments number, returns number.

References

1. 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.
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. 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.
4. C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface between Two Approaches to Computer Algebra. In *Proceedings of the ACM ISSAC 2005*, pages 261–269, 2005.
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.
7. 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.

Appendix

A Fixing an Aldor Compiler Bug

For Aldor compilers *before* version 1.1, it is necessary to fix a compiler bug to use Alma: In the type inference module some operations are checking to see whether two symes are equal by testing pointer equality. It should use the `symeEqual` function instead. The fix involves two Aldor source files: `absub.c` and `tfsat.c`, which can be found in the *AlmaArchFolder/src* folder. Try to find the commented lines that contain the word `ALMA`, for example by typing at the command prompt:

```
src$ grep "ALMA" *.c
```

Follow the comments to do the necessary modifications in your Aldor compiler sources to fix the bug. Recompile the compiler:

```
src$ make
```

and run the test suites.