Mixed-language Automatic Differentiation

Valérie Pascual, Laurent Hascoët

To cite this version:

Valérie Pascual, Laurent Hascoët. Mixed-language Automatic Differentiation. AD2016 - 7th International Conference on Algorithmic Differentiation, Sep 2016, Oxford, United Kingdom. hal-01393424

HAL Id: hal-01393424

https://hal.archives-ouvertes.fr/hal-01393424

Submitted on 10 Nov 2016

Mixed-language Automatic Differentiation

Valérie Pascual[*] and Laurent Hascoët[†]

March 29, 2016

As AD usage is spreading to larger and more sophisticated applications, problems arise for codes that use several programming languages. Many AD tools have been designed with one application language in mind. Only a few use an internal representation that promotes language-independence, at least conceptually. When faced with the problem of building (with AD) the derivative code of a mixed-language application, end-users may consider using several AD tools, one per language. However, this leads to several problems:

- Different AD tools may implemented very different AD models such as overloading-based versus source-transformation based or association-by-address versus association-by-name. These models are often not compatible.
- When selecting the source-transformation model (for efficiency of the differentiated code), performance of the differentiated code strongly depends on the quality of data-flow analysis, which must be global on the code. A global analysis with separate AD tools would require inter-tool communication at the level of data-flow analysis, which does not exist at present.

In any case, interoperable data-flow analysis between tools imply that the tools share their analysis strategy, which is almost never the case. Consequently, we think the only viable approach is to use a single tool, with a single internal representation and data-flow analysis strategy, therefore converting each source file to this unique representation regardless of its original language. It turns out that Tapenade [1] provides such an internal representation, accessible at present from C or Fortran sources.

Other AD tools provide a language-independent internal representation. OpenAD provides such a representation based on the XAIF formalism. However, this gives birth to two separate tools, OpenAD/F for Fortran, and ADIC2 for C. Still, it seems that their is no deep reason to prevent OpenAD application to mixed-language codes. We are lacking information about common architecture between TAF and TAC++ that would allow such mixed-language AD.

Rapsodia [2] was the first AD tool to support algorithmic differentiation in tangent mode of mixed-language components, specifically C++ and Fortran. As Rapsodia uses operator overloading, it performs no global analysis of the code. To our knowledge the extension of mixed-language differentiation with Rapsodia to adjoint mode is not yet provided.

1 Language standards and interoperability

Language standards often say very little about interoperability with other languages, leaving much freedom to compilers. Still, usage has progressively brought de facto standards, in particular between C and Fortran. The Fortran 2003 standard has specified its interaction with C in more detail. As far as AD is concerned, AD tools should in any case not commit to any specific interoperability strategy, and in particular to parameter-passing behaviors. Those might change with new languages and versions of languages. Instead, an AD tool must be able to handle a set of behaviors (hopefully small) from which one can describe all reasonable ways of parameter-passing.

The main issue raised by analysis and transformation of mixed-language codes is parameter-passing. Other issues are related to matching elements across languages, mainly types and procedures. These matching issues seem less complex than parameter-passing, but they should be discussed first.

Interoperability between types (and between variables of these types) across languages relies on identical memory representations built by compilers. Obviously interoperable types must match in the sense that they have the same structure, number of fields, and these fields must recursively be of interoperable types. Compilers often grant a natural interoperability between structured types. However Fortran 2003 provides the `bind` attribute to tell at compile time that a Fortran type has a C equivalent. It is essential to identify interoperable types in both languages, in particular because it may help distinguish candidate interoperable procedures according to the types of their arguments.

Similarly, interoperability between procedures relies on the compiler identifying the called procedure from another language, usually comparing procedure names, arguments number, and interoperable arguments types. Interoperable procedures in Fortran 2003 are declared with an explicit interface. The bind attribute allows one to associate the Fortran procedure name to the C procedure name. With Fortran 77 and Fortran 90, different conventions exist to associate both names: either by adding an underscore character at the end of the Fortran name, either with the same name, or with the name in uppercase, depending on the compilers.

[*]Corresponding Author, ECUADOR team, INRIA Sophia-Antipolis, France, `Valerie.Pascual@inria.fr`
[†]ECUADOR team, INRIA Sophia-Antipolis, France, `Laurent.Hascoet@inria.fr`

The parameter-passing strategy is already a property of each given, single language. Mixed-language calls may be at the interface between two different parameter-passing strategies, which adds extra complexity. Still all resulting parameter-passing behaviors should be eventually expressed using a small number of elementary tactics.

Inside a given language, parameter-passing may use one of a few classical strategies: call by value, call by reference, call by value-result, call by sharing, call by name .... Call by value is the most common strategy. In call by value, the argument expression is evaluated, and the resulting value is copied to the corresponding variable in the function. If the function assigns values to its parameters, only its local copy is assigned and the argument passed into the function call is unchanged when the function returns. In call by reference, a procedure receives a reference to a variable and can modify the variable used as argument. Call by value-result, also named call by copy-restore, is a special case of call by reference. It differs from call by reference when two arguments alias one another. Under call by reference, writing to one will affect the other. Call by value-result gives the function distinct copies, but result in the callers environment depends on which of the aliased arguments is copied back first. Call by sharing is a terminology used by languages such as Python, Java and other object oriented languages. It is analogous to call by value where the passed value is either the argument when it is of a primitive type or its address when is is an object. Whereas in call by value, the arguments are computed before calling the function, in call by name, each occurrence of the formal argument is replaced with the actual argument in the style of macro-expansion.

Consider now the classical mix of Fortran and C. Parameter-passing mechanisms differ in the two languages. In Fortran, call by value-result and call by reference are used. Fortran 2003 introduces the `VALUE` attribute to specify call by value. In C, call by value is the only parameter-passing mechanism. All parameters are passed by value, except for arrays, which are translated into the address of the first element. In C, one simulates a call by reference by passing a pointer to this parameter. In mixed language calls, the caller and the called procedure must agree on how parameters are passed, e.g. passing a pointer from C to a Fortran procedure if the parameter has no `VALUE` attribute.

We believe that every mixed-language parameter-passing strategy can boil down to a few simple behaviors at the time of entry into and return from the called procedure. At call time, we define what we call the *passed argument*, which may be the same as the actual argument, or the memory pointed to by the actual argument, or conversely the address of the actual argument, depending on the mixed-language strategy that must be captured. Then the internal memory corresponding to this passed argument is copied into the internal memory corresponding to the called procedure's formal argument. At return time, the internal memory corresponding to the formal argument may be either copied back to the actual parameter, or not copied in which case it will vanish when the called procedure is popped from the call stack. When there is a back copy, it follows the link from the passed argument back to the actual argument: if they are the same, the copy is written into the actual argument, and if the passed argument is the destination of the actual, then the copy is written at the address designated by the actual argument.

Fig. 1 illustrates these behaviors for a few example multi-language calls, and also for pure Fortran calls distinguishing the scalar case from the array case and for a C call using pointers. For each situation, we give the choice of the passed argument and the choice about back copy that implement the desired behavior. Anticipating on the next section, Fig. 1 also shows for each situation, the `Translator` object used by Tapenade to specify these choices to the data-flow analyses. We will now investigate how to extend our AD tool to analyze mixed-language codes, and how differentiation must be adapted.

2 Extension of Tapenade algorithms for interoperability

Tapenade was originally designed to support different imperative languages. The motivation was to share the model of the tool and its implementation between these languages (at present, Fortran and C). We believe that this architecture also lets us deal with mixed-language source for a minimal implementation effort, affecting only a few components of the tool.

Tapenade represents a code as a call graph whose nodes represent procedures and arrows represent calls. Each call graph node contains a flow graph, in which nodes are blocks of elementary instructions (in particular calls), and arrows represent control jumps. At present, Tapenade called on C source or on Fortran source builds an internal representation of the same nature, using the same components for procedures, instructions, variables, types ... We exploit this unicity of representation to deal with mixed-language codes, in particular mixing Fortran and C.

Let's first take a look at matching of types and procedures. Two interoperable entities are represented with the same internal representation in the AD tool. This representation distinguishes each component of structured types, and distinguishes pointer variables from their pointee destination variables. Representation of arrays on the other hand do not distinguish array elements, and therefore we need not worry about their possibly different memory layout in Fortran and C. Sometimes it is not possible to preserve this nice structural matching, for instance for the `complex` Fortran type which should match a 2-fields structure in C. Then the correspondence must be enforced by implementation, and possibly incurs some loss of information.

The question of procedure matching amounts to finding, for a given procedure call, the node of the call graph that will be effectively called. This depends on the mixed-language conventions on procedure names and on type matching. Procedure name conventions may vary, and Tapenade offer parameterization to define one convention, using command-line arguments or directives at the call site. It also interprets attributes of Fortran 2003. For instance, these parameters let us specify that inside a Fortran code all calls to a procedure named `FOO` will connect to a C procedure named `foo_`.

C calls Fortran

```
float *y;                    SUBROUTINE BAR(V)
...                            REAL V
bar(y);
```

- Passed argument is *y
- Upon return, value of V is copied back into *y

```
Translator: V -> *y (Back copy)
```

C calls Fortran, by value

```
float y;                     SUBROUTINE BAR(V)
...                            REAL, VALUE:: V
bar(y);
```

- Passed argument is y
- Upon return, no copy takes place into y

```
Translator: V -> y (No back copy)
```

Fortran calls C

```
REAL X                       void foo(float *a)
...
CALL FOO(X)
```

- Passed argument is address of X
- Upon return, no copy takes place into &X

```
Translator: a  -> &X    (No back copy)
           *a  -> X     (Back copy)
```

C calls C

```
float *y;                    void bar(float *a)
...
bar(y);
```

- Passed argument is y
- Upon return, no copy takes place into y

```
Translator: a  -> y    (No back copy)
           *a  -> *y   (Back copy)
```

Fortran calls Fortran, scalars

```
REAL X                       SUBROUTINE GEE(V)
...                            REAL V
CALL GEE(X)
```

- Passed argument is X
- Upon return, value of V is copied back into X

```
Translator: V -> X (Back copy)
```

Fortran calls Fortran, arrays

```
REAL Y(100)                  SUBROUTINE GEE(B)
...                            REAL B(20)
CALL GEE(Y(10))
```

- Passed argument is &(Y(10))
- Upon return, no copy takes place into Y

```
Translator: B -> &(Y(10)) (No back copy)
```

Figure 1: Mixed-language calls and the Translator that implements their behaviors

Parameter passing comes into play during data-flow analysis. All data-flow analyses (e.g. in-out, activity, liveness) inherit from a base analysis class that provides primitives to transfer data-flow information between a caller procedure and a callee. Most adaptions to mixed-language code must be done in this base class. This information transfer is driven by an object we call a *Translator*, which describes how actual arguments are matched with formal arguments. Variables, and in particular those in arguments, may have a finer structure. In addition to the classical scalar and arrays that existed for Fortran 77, languages have introduced variables of structured type and pointers. Since the data-flow properties that we analyze may be different for each component of these structured objects, we distinguish all of their elementary components. For example, a formal argument of type `mystruct *arg` where mystruct is a record:

```
struct mystruct {
   int numElems;
   float *elems;
}
```

is represented by 4 elementary formal arguments, for `*(arg->elems)`, `arg->elems`, `arg->numElems`, and the top-level `arg` respectively. To each elementary formal argument of the called procedure, the `Translator` associates the corresponding elementary actual argument at the call site, which is either an expression or an elementary variable known to the calling procedure. In addition, the `Translator` associates to each elementary formal argument a boolean that specifies whether it must be copied upon return. For each example situation in Fig. 1, the `Translator` that implements the desired behavior is shown below the textual description of the behavior, as a set of arrows from formal elementary argument to actual elementary argument and back copy boolean. The rule of thumb is that the `Translator` associates the root formal argument with the passed argument.

The "Fortran calls C" situation deserves further comment: since the C formal argument is a pointer to a float, there are in fact two elementary formal arguments, one for `a` and one for `*a`. The same happens for "C calls C". As `a` is associated with the passed argument `&X`, `*a` is naturally associated with `*(&X)` in other words with `X`. It is in fact thought this second elementary argument that the actual value of (or information regarding) `X` is propagated to the callee. Consequently, even if no back copy is done upon return into &X itself, every write into *a in foo is automatically reflected into X.

The way each data-flow analysis, which computes a given data-flow value, uses the specification from the Translator can be sketched as follows: immediately before the call, and for each elementary formal argument (left column of Translator), we retrieve the corresponding elementary actual argument (right column), and we retrieve the current data-flow value for it. The initial data-flow value of the elementary formal argument is set to this retrieved value. Analysis can then run on the called procedure. Upon return from the call, the top-level elementary arguments that bear the "No back copy" retain the data-flow value they had before the call. For all other elementary arguments the data-flow value of the elementary formal argument upon return is back copied into the data-flow value of the elementary actual argument. This description applies to forward data-flow analyses. Adaption to backward analyses require only minor technical changes.

```
void bar(float a, float *b);
void foo(float *x, float *y) {
  bar(*x, y);
}
```

```
void foo_b(float *x, float *xb,
           float *y, float *yb) {
    bar_b(*x, xb, y, yb);
```

```
SUBROUTINE BAR(u, v) BIND(C)
  IMPLICIT NONE
  REAL, VALUE :: u
  REAL :: v
  u = 2 * u
  v = u * u
END SUBROUTINE
```

```
SUBROUTINE BAR_B(u, ub0, v, vb) BIND(c)
  IMPLICIT NONE
  REAL, VALUE :: u
  REAL :: ub, ub0
  REAL :: v
  REAL :: vb
  u = 2*u
  ub = 2*u*vb
  *vb = 0.0
  ub = 2*ub
  ub0 = ub0 + ub
END SUBROUTINE BAR_B
```

Figure 2: Mixed-language differentiation with Tapenade, C calling Fortran case

The main impact of dealing with mixed-language code on actual differentiation is related to arguments being copied to the formal arguments and *not* being copied back. For adjoint differentiation, the consequence is that the copied formal argument must not be identified with the actual argument of the call. In particular, the adjoint formal argument must be a different variable from the adjoint actual argument, and the adjoint code must follow the pattern illustrated in Fig. 2: variable u is the original formal argument, which is not copied back because of its "value" attribute. The differentiated actual argument is (conceptually) ub0, whereas the differentiated formal argument is ub. Only at the end of BAR_B, the differentiated formal argument ub is added into the differentiated actual argument ub0. Consequently, ub0 must be passed a reference xb, even if the corresponding non-differentiated argument was passed a dereferenced *x. In other words the adjoint of a passed by value argument must sometimes be passed by reference.

When differentiating a procedure that calls a function F that returns a value, the last argument of the differentiated procedure contains the result of F. This argument and its corresponding passed parameter must be declared and used according to the parameter-passing mechanism, e.g. by passing the address of the argument when calling a Fortran function from C, instead of just passing the argument for a C function.

3 Conclusion and further work

Tapenade is now able to differentiate codes that mix C and Fortran, with calls in either direction. Apart from short non-regression tests, we are validating this extension on the "CalculiX" finite element library. The architecture of Tapenade allowed us to implement this functionality quite easily, with modifications in only a few component of the tool. We still need to study interoperability of global variables, e.g. using the bind statement of Fortran 2003. As the implementation is still at an early stage, we need to experiment on a few representative examples.

References

[1] L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.

[2] Jean Utke, Bradley T. Rearden, and Robert A. Lefebvre. Sensitivity analysis for mixed-language numerical models. *Procedia Computer Science*, 18:1794 – 1803, 2013. 2013 International Conference on Computational Science.