1

Extension of TAPENADE towards Fortran 95

Valérie Pascual and Laurent Hascoët

INRIA, TROPICS team, 2004 route des Lucioles 06902 Sophia-Antipolis France,
Valerie.Pascual@sophia.inria.fr, Laurent.Hascoet@sophia.inria.fr

Summary. We present extensions to the automatic differentiation tool TAPENADE to increase coverage of the Fortran 95 language. We show how the existing architecture of the tool, with a language independent kernel and separate front-ends and back-ends, made it easier to deal with new syntactic forms and new control structures. However, several new features of Fortran 95 required us to make important choices and improvements in TAPENADE. We present these features, sorted into four categories: about the top-level structure of nested modules, subprograms, and interfaces; about structured data types; about overloading capabilities; and about array features. For each category, we discuss the choices made, and we illustrate their impact on small Fortran 95 examples. Dealing with pointers and dynamic memory allocation is delayed until extension to C begins. We consider this extension to Fortran 95 as a first step towards object-oriented languages.

Key words: TAPENADE, Fortran 95, tools, program transformation

1.1 Introduction

We present extensions to the automatic differentiation [8, 4] tool TAPENADE [11, 10] to increase coverage of the Fortran 95 language [13]. Other AD tools already took this direction, such as TAF [6, 7, 12] and the NAGWare AD-enabled Fortran 95 compiler [14]. ADIFOR [3] has already considered the differentiation of code with structured data types.

Given the source of an original program, plus a description of which output variables must be differentiated, and with respect to which input variables, TAPENADE produces a new source program that computes the requested derivatives.

We recall the internal architecture of TAPENADE [11] in Fig. 1.1, with a central module for program analysis and transformation, surrounded by language-specific front-ends and back-ends. This allows the central module to forget about mostly syntactic details of the analyzed language, and to

concentrate on the language's semantic constructs. To this end, TAPENADE defines an internal abstract language, called IL, able to represent all constructs of classical imperative languages. In particular, extension to Fortran 95 drove us to add several new constructs into IL. Some of these constructs will thus be directly available when we start extension to C. Furthermore, programs are internally represented as Call Graphs, Control Flow Graphs [1], and, only at the deepest level of individual statements, Syntax Trees. This yields a general representation for all control structures.

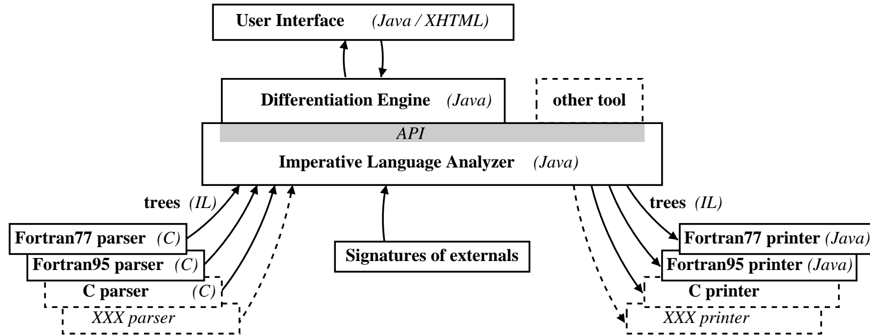

Fig. 1.1. Architecture of TAPENADE

Concerning Fortran 95 syntax, everything is taken care of by a specific new parser (front-end) and pretty-printer (back-end). A major difference compared to Fortran 77 is the free format source form, where statements may start in any column. Our new Fortran 95 parser accepts programs that combine the old fixed format and the free format, and the back-end can regenerate programs using both formats. Thanks to the internal representation as Control Flow Graphs, new constructs such as the `SELECT CASE` were easily added and treated by the differentiation engine as any other flow of control. The same remark applies to the new `CYCLE` and `EXIT` constructs.

In this paper, we focus on the features of Fortran 95 that required us to make important choices and improvements in TAPENADE. We put these features into four categories: Sect. 1.2 deals with the nesting of modules, subprograms, and interfaces, Sect. 1.3 deals with the treatment of structured types, called “derived” types in Fortran 95, Sect. 1.4 deals with the overloading capabilities, and Sect. 1.5 deals with array features. Sect. 1.6 concludes with the soon to come pointer analysis, and the more distant extension to object-oriented programming.

1.2 Nesting of Modules and Subprograms

The internal representation has to be extended to capture the new top-level nesting of procedures, with modules, internal/external subprograms, and in-

terfaces. In comparison, the structure of Fortran 77 was flat, apart from statement functions and internal subprograms in some dialects.

We choose to introduce an internal tree representation of module nesting, in addition to the existing Call Graph. Each node stands for one "unit," i.e. subprogram or module, and holds the list of its enclosed units. In particular, the regenerated differentiated program must comply with this unit tree structure, so that this program is stand-alone and can be compiled directly. Each unit defines two symbol tables, for the public and private symbols (i.e. arguments, variables, subprograms, types, etc). The private symbol table naturally inherits from the public one. Symbol table nesting already existed in TAPENADE for scoping. The `USE` statement just states that a unit has access to the public symbol table of a module.

Classically, program analyses and transformations need to run in an appropriate order on the subprograms, depending on the Call Graph. The novelty is that this order now must take into account the `USE` of modules. Moreover, recursion may introduce cycles in this dependence, so the best order can only be an approximation.

Where differentiation is concerned, the question is what must belong to a differentiated unit? When Fortran 77 was considered, differentiated symbols could be defined independently from their original symbols. Now that modules can define their own private symbols, some differentiated unit must often be declared in the same context as its original unit, i.e. must live inside the same enclosing module. In general the question is whether the differentiated object can exist independently of the original object, or must they be attached inside the same enclosing level. For example, a differentiated statement must be in the same subprogram as the original statement because they share a common control. Similarly a differentiated subprogram must be in the same module as the original if both access a private entity of this module. On the contrary, a differentiated component `x` of a derived type `T` need not be added into `T`, but can rather go into a stand-alone "differentiated" derived type `T'`, therefore saving memory space.

To illustrate, consider the source program of Fig. 1.2, containing a type definition, variables and a function. The corresponding differentiated program contains the same declarations as the source program, plus the differentiated function.

1.3 Derived Types

Fortran 95 allows the user to define "derived" types (this name has no relation with differentiation) in order to manipulate composite objects containing several components. As we said in Sect. 1.2, our choice during differentiation is to define a differentiated derived type, whose components hold the derivatives of the original components. However, it happens that some variables of some derived type have only some components that are active. Then the

```
module example1
 implicit none
 type vector
   real :: x,y,z
 end type vector

 type(vector) :: u,v,w

contains
 function dot_prod(a,b)


  type(vector) :: a,b

  real :: dot_prod

  dot_prod = a%x*b%x + &
&       a%y*b%y + a%z*b%z




 end function dot_prod









end module
```

```
!  Generated by TAPENADE
!      Version 2.0.12
MODULE EXAMPLE1_D

  TYPE VECTOR
      REAL :: x,y,z
  END TYPE VECTOR

  TYPE(VECTOR) :: u, v, w

CONTAINS
  FUNCTION DOT_PROD_D(a, ad, &
&        b, bd, dot_prod)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    TYPE(VECTOR) :: ad, bd
    REAL :: dot_prod
    REAL :: dot_prod_d
    dot_prod_d = ad%x*b%x + &
&      a%x*bd%x + ad%y*b%y + &
&      a%y*bd%y + ad%z*b%z + &
&      a%z*bd%z
    dot_prod = a%x*b%x + &
&       a%y*b%y + a%z*b%z
  END FUNCTION DOT_PROD_D

  FUNCTION DOT_PROD(a, b)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    REAL :: dot_prod
    dot_prod = a%x*b%x + &
&       a%y*b%y + a%z*b%z
  END FUNCTION DOT_PROD
END MODULE EXAMPLE1_D
```

Fig. 1.2. Differentiation of nested modules and subprograms

differentiated type need not allocate space for the other components. Therefore, differentiated derived types depend on the activity pattern. On the other hand, we don't want to *specialize* too far, creating several differentiated types for a given derived type. Therefore, our choice is very similar to differentiation of subprograms with several activity patterns: there is only one differentiated type `T'` for each derived type `T`. During activity analysis, if a component `x` of some variable of type `T` can be active, the differentiated type `T'` must hold a component `x` too. The price for this non-specialization is that sometimes a differentiated variable of type `T'` will not use all of its components.

In the previous example, the differentiated type of the "vector" type is equal to the initial type as all components are active, so no differentiated type appears in the differentiated module.

In the example of Fig. 1.3, the "name" and "y" components of type "vector" are not active. Therefore they do not appear in the differentiated type "vector_d."

```fortran
module example2
 implicit none
 type vector
   character(256) :: name
   real :: x,y,z
 end type vector

 type(vector) :: u,v,w

contains
 function test(a,b)

  type(vector) :: a,b

  real :: test
  print *, a%name , b%name
  test = a%x + b%x + u%z

 end function test

end module
```

```fortran
!  Generated by TAPENADE
!      Version 2.0.12
MODULE EXAMPLE2_D

  TYPE VECTOR_D
      REAL :: x,z

  END TYPE VECTOR_D

  TYPE VECTOR
      CHARACTER*(256) :: name
      REAL :: x,y,z
  END TYPE VECTOR

  TYPE(VECTOR) :: u, v, w
  TYPE(VECTOR_D) :: ud

CONTAINS
  FUNCTION TEST_D(a, ad, b, &
&   bd, test)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    TYPE(VECTOR_D) :: ad, bd
    REAL :: test, test_d
    PRINT*, a%name , b%name
    test_d = ad%x + bd%x + ud%z
    test = a%x + b%x + u%z
  END FUNCTION TEST_D

  FUNCTION TEST(a, b)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    REAL :: test
    PRINT*, a%name , b%name
    test = a%x + b%x + u%z
  END FUNCTION TEST
END MODULE EXAMPLE2_D
```

Fig. 1.3. Differentiation of derived types

1.4 Overloading

The term overloading refers to calling different subprograms or operators by the same generic name. Whereas overloading in object-oriented languages is resolved only at run time, the limited form of overloading in Fortran 95 can be resolved statically at compile time, and therefore at differentiation time. This is done during the type-checking phase. Furthermore, Fortran 95 also allows the user to overload predefined operators such as `+`, `-`, `*`, `/`, or assignment `=`.

We must thus modify the type-checking algorithm carefully. Each use of a predefined operator or call to a subprogram is compared to available overloaded subprograms according to the arguments' types. If necessary, it is replaced by the appropriate subprogram call, and treated as such in the following differentiation phase. Therefore, at the end of the type-checking phase, overloading is completely resolved.

When differentiation is concerned, the predefined operators are treated in a very particular, built-in manner, so we must be careful not to replace these operators by ordinary subprograms calls when not necessary.

In our third example of Fig. 1.4, the addition of vectors is overloaded.

1.5 Array Features

The array programming features of Fortran 95 are represented through syntactic notations and intrinsic functions. In Fortran 95 programs, it is possible to use whole array operations. In the differentiated program, we keep this property whenever possible, and we also use array operations on differentiated arrays.

For example, for arrays A, B and scalar x, the loop

```
do i=1,N
    A(i) = 3*B(i-1) + x
end do
```

can be written equivalently as:

```
A(1:n) = 3*B(0:n-1) + x
```

Array features can be advantageous for static data flow analysis. A reset of a whole array to a constant, for example, can be easily detected, whereas the equivalent loop requires array region analysis [5] to reach the same conclusion.

When differentiation is concerned, two intrinsic array functions play a very special role: the `SUM` intrinsic and the spread operation (which is often implicit or otherwise is done by the `SPREAD` intrinsic). In particular the adjoint of a `SUM` is a spread, and vice-versa. For example the adjoint statement of

```
x = x + SUM(B(:))
```

```
module example3
 implicit none
 type vect
   real :: x,y
 end type vect
 type(vect) :: u,v,w
 interface operator (+)
  module procedure addvect
 end interface

contains
 function test(a,b)


  type(vect) :: a,b,test


  test = a + b + u



 end function test

 function addvect(a,b)


  type(vect),intent(in)::&
&            a,b
  type(vect) :: addvect

  addvect%x = a%x + b%x

  addvect%y = a%y + b%y
 end function addvect







end module
```

```
!  Generated by TAPENADE
!     Version 2.0.12
MODULE EXAMPLE3_D

  TYPE VECT
      REAL :: x,y
  END TYPE VECT
  TYPE(VECT) :: u, ud, v, w
  INTERFACE OPERATOR(+)
      MODULE PROCEDURE addvect
  END INTERFACE

CONTAINS
  FUNCTION TEST_D(a, ad, b, &
&                 bd, test)
    IMPLICIT NONE
    TYPE(VECT) :: a, ad, b, bd
    TYPE(VECT) :: test, test_d
    TYPE(VECT) :: arg1, arg1d
    arg1d = ADDVECT_D(a, ad,&
&                  b, bd, arg1)
    test_d = ADDVECT_D(arg1,&
&            arg1d, u, ud, test)
  END FUNCTION TEST_D

  FUNCTION ADDVECT_D(a, ad,&
&     b, bd, addvect)
    IMPLICIT NONE
    TYPE(VECT),INTENT(IN):: a,b
    TYPE(VECT) :: ad, bd, &
&     addvect, addvect_d
    addvect_d%x = ad%x + bd%x
    addvect%x = a%x + b%x
    addvect_d%y = ad%y + bd%y
    addvect%y = a%y + b%y
  END FUNCTION ADDVECT_D

  FUNCTION TEST(a, b)
    ...
  END FUNCTION TEST
  FUNCTION ADDVECT(a, b)
    ...
  END FUNCTION ADDVECT
END MODULE EXAMPLE3_D
```

Fig. 1.4. Differentiation of overloaded procedures and operators

is the following, with an implicit spread on $\overline{\mathtt{x}}$

$\overline{\mathtt{B}}$`(:) = `$\overline{\mathtt{B}}$`(:) + `$\overline{\mathtt{x}}$

Actually these two intrinsics blend into the internal representation for partial derivatives and reappear when generating the differentiated code.

To illustrate how this is effectively performed by TAPENADE, we need a more complete example. Consider the following array assignment of a scalar value to array `A`, at indices 0 to 100 by stride of 3, where `x` is a scalar:

```
A(0:100:3) = x * SUM(B(:))
```

The elements of `A` that are not in the section `A(0:100:3)` play no role in this statement, and can therefore be neglected. In the rest of this example, we shall thus simplify the notation by writing `A` instead of `A(0:100:3)` when there is no ambiguity. This array assignment is differentiated in the reverse mode using the transposed local Jacobian:

$$\left\{\frac{\partial[\mathtt{A},\mathtt{x},\mathtt{B}]_{out}}{\partial[\mathtt{A},\mathtt{x},\mathtt{B}]_{in}}\right\}^* = \begin{pmatrix} 0 & \mathtt{SUM(B(:))} & \mathtt{x} \\ 0 & Id & 0 \\ 0 & 0 & Id \end{pmatrix}^* = \begin{pmatrix} 0 & 0 & 0 \\ \mathtt{SUM(B(:))} & Id & 0 \\ \mathtt{x} & 0 & Id \end{pmatrix}$$

which is a block matrix, whose structure is emphasized in Fig. 1.5, using rectangles to represent blocks. Notice that the four larger blocks are actually rectangular blocks, whereas the other blocks are two row matrices, two column matrices, and a 1x1 block. The *Id* blocks correspond to the identity matrix. The `SUM(B(:))` block is a row matrix where each entry is equal to `SUM(B(:))`.

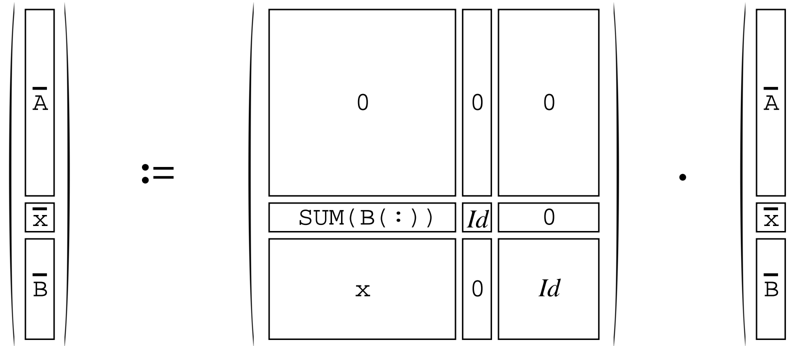

Fig. 1.5. Transposed Jacobian block matrix for an array assignment

$\overline{\mathtt{A}}$ is a column matrix whose values are $\overline{\mathtt{A}}$`(0:100:3)`. Figure 1.5 shows the vector assignment that the differentiated assignments must implement.

The shape of the blocks determines where `SUM`'s and spreads must appear in the differentiated array assignments. For example, the assignment that updates $\overline{\mathtt{x}}$ implements the product of the `SUM(B(:))` row vector by the $\overline{\mathtt{A}}$ column vector, yielding the following reduction:

$\overline{\mathtt{x}}$` = `$\overline{\mathtt{x}}$` + SUM(B(:))*SUM(`$\overline{\mathtt{A}}$`(0:100:3))`

The complete set of differentiated assignments is:

$\overline{x} = \overline{x}$ + SUM(B(:))*SUM($\overline{A}$(0:100:3))
$\overline{B}$(:)= $\overline{B}$(:)+ x*SUM($\overline{A}$(0:100:3))
$\overline{A}$(0:100:3) = 0.0

Only SUM and spread blend with the local Jacobian notation as described above. All the other array intrinsics are treated like black-box routines, whose differentiation is given to TAPENADE in special library files.

1.6 Conclusion

We have described extensions of the AD tool TAPENADE towards coverage of Fortran 95. This extension is made easier by the fact that Fortran 95 derives from Fortran 77, and also by TAPENADE's internal representation of programs independent from the language. For example, the notion of structured data types was already in TAPENADE even before extension to Fortran 95 was considered. Obviously, all constructs that exist both in Fortran 77 and Fortran 95 required no new development at all. The features of Fortran 95 that called for new developments are those which had been overlooked or not fully tested because Fortran 77 did not use them. Similarly, TAPENADE already handles nested scoping blocks inside a procedure, and this feature will be available immediately for the extension to C, just like structured types.

One major feature of Fortran 95 is still overlooked in the present work: pointers and dynamic allocation. The internal representation already captures pointers, but the program analyses do not take them into account. This is the next development on our list. We plan to share this with the soon-to-come extension to C. In addition to the development of a classical pointer analysis, this will require a conceptual study of a differentiation model for pointers, especially in the reverse mode of AD. Notice that differentiation of dynamic allocation and pointers is only problematic for reverse mode AD. Tangent AD, as well as reverse AD implemented through a tape like in ADOL-C [9] handle these features in a straightforward fashion.

In the long run, we shall consider extending TAPENADE to the object programming concepts from C++ or JAVA. This will introduce dynamic overloading, which is still a challenge for automatic differentiation. For example, the activity pattern of the actual parameters of a given call may contribute to the activity pattern of several subprograms. However, the module nesting management developed here is a major step towards handling the global structure of object-oriented programs.

There are two ways you can use TAPENADE. It can be used as a server at the url

http://tapenade.inria.fr:8080/tapenade/index.jsp

Alternatively, it can be downloaded from

ftp://ftp-sop.inria.fr/tropics/tapenade

and locally installed. In that case it is run by a simple command line, which can be included into a Makefile. TAPENADE also provides a user-interface to visualize the results in a HTML browser. An on-line documentation is available at the url

`http://www.inria.fr/tropics`

We encourage you to use TAPENADE and to report any problems, therefore helping us making it an industrial quality AD tool for Fortran 95.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.
3. Alan Carle and Mike Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
4. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann(editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization.* Springer, 2001. Selected proceedings of AD2000, Nice, France.
5. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.
6. R. Giering, T. Kaminski, and T. Slawig. Applying TAF to a NavierStokes solver that simulates an Euler flow around an airfoil. In *Future Generation Computer Systems (to appear).* Elsevier Science, 2004.
7. R. Giering, T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow. Generating tangent linear and adjoint versions of NASA/GMAO's Fortran-90 global weather forecast model. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.
8. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Frontiers in Applied Mathematics, 2000.
9. A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.
10. L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint ANL-MCS/P936-0202, Argonne National Laboratory, 2002. also *Research Report RR-4856*, INRIA.
11. L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical Report RT-0300, INRIA, 2004.
12. T. Kaminski, R. Giering, M. Scholze, P. Rayner, and W. Knorr. An example of an automatic differentiation-based modelling system. In *Computational Science - ICCSA 2003 - Lecture Notes in Computer Science.* Springer, 2003.
13. M. Metcalf, J. Reid, and M. Cohen. *Fortran 95/2003 Explained.* Oxford University Press, 2004.

14. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.