Derivatives By-Address for Fortran 77

Mike Fagan

Rice University
`http://www.cs.rice.edu/~mfagan`

Abstract.

FIXME
Automatic differentiation tools use 1 of 2 strategies to access derivative values. These strategies are:
1. By-address
2. By-name

The by-address method is typically implemented by introducing structured types for each active scalar type. For example, scalar type 'real' will have an associated structure type 'active-real'. Using this strategy, all active variables types are changed to the associated structured type. On the other hand, the by-name method introduces an associated new variable for each active variable. For example, the derivatives associated with 'pressure' would be 'd_pressure'. Since the by-address strategy employs structured types, AD tools for Fortran 77 have not employed that strategy. In this paper, we show how to use array access to implement the by-address strategy for Fortran 77. We discuss the canonicalization issues, outline our Adifor3.0 implementation of this technique, and give a few sample performance comparisons of the by-address vs the standard Adifor3.0 by-name.

1 Introduction

Automatic differentiation (AD) tools must have some mechanism for associating the *derivatives* of a program variable with the program variable. There are 2 techniques for implementing this association:

By-address This mechanism works by introducing structured types for all derivative-bearing scalar types in a program. For example, the scalar type 'real' will have an associated 'activeReal'. The 'activeReal' type looks like:

```
struct activeReal {
    real val;
    real deriv; }
```

An AD tool that uses the by-address strategy alters all derivative-bearing variables of type 'real' in the original code to have type 'activeReal' in the differentiated code. So, an original program variable `x` of type 'real' has an associated program variable `x` *of type* 'activeReal' in the differentiated code. In the differentiated code, both the original value and the derivatives of `x` are referenced by slot selection:

value of `x` **is given by** `x.val`
derivative of `x` **is given by** `x.deriv`

By-name This mechanism works by introducing a (canonically named) variable to hold the derivatives. For example, the derivatives for program variable `pressure` would be stored in an AD-introduced `g_pressure`

For most programming languages, AD implementors may choose either by-address or by-name. AD tools for C/C++ typically use by-address. For Fortran 90/95, the world is divided, some use by-address, some use by-name. For Fortran 77, however, the choice has uniformly been by-name. The simple reason is that *Fortran 77 does not have structure types.* The lack of structure types, however, does not preclude by-address AD. The main contribution of this paper shows that array indexing can be used as a basis for a by-address derivative access method. As a side benefit, this technique works well with Fortran 90 array slicing as well. Specifically, this paper:

- Elaborates of the array indexing method for implementing by-address derivative access (section 2). While no formal proof is given, section 2 shows informally why this method works. Section 2 also illustrates how array indexing by-address carries through for Fortran 90, giving a graceful way to handle array slicing.
- Demonstrates that simple Fortran 77 programs using by-address described in this paper show no performance penalty when compared to by-name(section 3). The emphasis here is on *simple*. The programs here are short programs, and would not tax the machine in any case. The main result is that the indexing technique does not appear to be horribly expensive.

2 Using Indexing for By-address Derivatives

The key idea to implementing by-address for Fortran 77 is to "add-an-index". Since changing the type is not allowed, change the dimension!

```
real x              ⟹real x(2)

real y(MSIZE,NSIZE)⟹real y(2,MSIZE,NSIZE)
```

(where the first index '2' can be expanded to '1+# of derivatives')

In this scheme, all references to the value of `x` use index 1, all derivative references use index 2. So, for a sample program statement

```
y = x * z
```

would be transformed to

```
y(1) = x(1) * z(1)               // program stmt
y(2) = x(1) * z(2) + x(2) * z(1) // derivative stmt
y = x * z
```

The example is almost a proof of correctness. If we assume that the by-name AD calculation is correct, then the transformation

$$x(e_1, ..., e_k) \Longrightarrow x(1, e_1, ..., e_k)$$
$$\mathrm{g\_}x(e_1, ..., e_k) \Longrightarrow \mathrm{g\_}x(2, e_1, ..., e_k)$$

is $1-1$ and onto, so the transform is an isomorphism among programs.

2.1 Consequences of "add-an-index"

For Fortran 77, the add-an-index technique will affect these areas:

User I/O . Reading or writing an entire array will require transforming the I/O statement to implied do form:

```
real x(10)
...
read(5)x
```

becomes

```
real x(2,10)
...
read(5)(i=1,10,x(1,i))
```

Block Data Initialization . Block data will have to be expanded with values for derivatives of active variables in the common block.

Equivalence statements . When an active value and an integer (or other inactive) are equivalenced, then the integer component must also be augmented via add-an-index.

Real-valued loop indices . Since Fortran permits scalar-only real loop indices, these loops must be altered.

```
real v
...
do v = 1.0,2.0,.05
   call add_coord(v)
enddo
```

gets changed to

```
real v(2),vtmp
...
vtmp = v(1)
do vtmp = 1.0,2.0,.05
   v(1) = vtmp
   call add_coord(v(1))
enddo
v(1) = vtmp
```

2.2 Add-an-index for Fortran 90

Even though Fortran 90 has structures (called 'derived types'), the add-an-index still provides some convenience with respect to array slicing. Suppose a derived type

```
type active
   real :: val
   real :: deriv
end type active
```

is used for by-address AD in Fortran 90. Then an active array declaration might look like:

```
type(active) :: x(10) ! formerly real :: x(10)
```

Now, consider an array slice `x(3 .. 9)`. The canonical value-reference syntax for this slice would be `x(3 .. 9)%val`. This is a Fortran 90 syntax error. There are ways around this problem (see ADOL-C manual[1]),but note that add-an-index handles the problem smoothly.

```
real :: x(2,10) ! formerly real :: x(10)
...
v(1,:) = x(1,3 .. 9) ! formerly v = x(3 .. 9)
```

3 A Comparison for Simple Fortran 77 Programs

In order to experiment with the add-an-index idea, the author created a prototype implementation inside the Adifor 3.0 system without too much pain and suffering. The prototype does *not* address any of the issues outlined in section 2.1, so it is not (yet) ready for inclusion in the Adifor 3.0 distribution.

The derivative code generated using add-an-index (obviously) does a lot more indexing operations than the standard by-name code that Adifor 3.0 normally generates. Given the larger number of index calculations, the natural question to ask is 'Does add-an-index noticeably slow down the derivative calculations?'.

To gain some intuition about the performance of add-an-index, the author used 3 test programs that are part of the Adifor internal acceptance test. The various program characteristics are:

Program	# Lines	Notes
P1	4	A simple program that does only arithmetic, using scalars
P2	12	A simple test of sin,log,exp intrinsics that also uses a some 2 dimensional arrays
P3	22	Uses 2 dimensional arrays, vaguely simulates an iterative solver

Each program was differentiated in forward mode, with 1 independent variable. The programs were compiled and run on a modest Linux box, using 'g77 -O3'. The times below were obtained with the 'time' command.

Program	By Name (sec)	Add-an-index
P1	.001	.001
P2	.0132	.0133
P3	.0273	.0272

Programs P1 and P2 ran so fast that they had to be timed using a shell script that ran them 10 times.

As can be seen from the numbers, for small programs, there is virtually no difference in performance. As indicated at the beginning of this section, this is not by any means a 'study'. It is just an attempt to see if a performance hit is readily apparent.

At some point, a serious performance study might be warranted. With a lot of derivatives, the locality-of-reference that by-address references have might be sufficient to overcome any additional over-index calculation.

4 Related Work

As far as the author knows, the add-an-index method for by-address AD is novel.

By-address using structure types is exemplified in 2 AD tools for C/C++:

1. ADOL-C[1]
2. ADIC[2]

Similarly, by-name AD for Fortran systems is exemplified in the following tools:

1. Adifor 2.0[3] and Adifor 3.0[4]
2. Tapenade[5]
3. TAF/TAMC[6]

A current by-address Fortran 90 system that is not (currently) using add-an-index is the OpenAD system[7]. The author is involved in this development effort, and the add-an-index technique is being considered as an option for the user.

Finally, the ADiMat[8] project to differentiate Matlab code employs hybrid technique using some features of both by-address and by-name.

References

1. Griewank, A., Juedes, D., Utke, J.: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Software **22**(2) (1996) 131–167
2. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software: Practice and Experience **27**(12) (1997) 1427–1456
3. Bischof, C.H., Carle, A., Corliss, G.F., Griewank, A., Hovland, P.D.: ADIFOR: Generating derivative codes from Fortran programs. Scientific Programming **1** (1992) 11–29
4. Carle, A., Fagan, M.: Adifor 3.0 overview. Technical Report CAAM-TR00-03, Dept. of Computational and Applied Mathematics, Rice University (2004)
5. Hascoët, L., Pascual, V.: Tapenade 2.1 user's guide. Technical report 300, INRIA (2004)
6. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Trans. Math. Software **24**(4) (1998) 437–474
7. Utke, J.: OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. (2004) online at ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf.
8. Bischof, C.H., Bücker, H.M., Lang, B., Rasch, A., Vehreschild, A.: Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), Los Alamitos, CA, USA, IEEE Computer Society (2002) 65–72