Automatic Differentiation of Polymorphic Fortran 77 Programs Using Adifor 3.0

Mike Fagan*

October 10, 2003

Abstract

Adifor 3.0 is a source-to-source transformation tool used to augment programs that compute derivatives. As part of the transformation process, Adifor analyzes certain aspects of program behaviour. Furthermore, that analysis depends on the original program being type correct. Since standard Fortran is officially monomorphic, the assumption of type correctness is not normally a difficult constraint to satisfy. There is a class of (non-standard) Fortran programs, however, that take advantage of the pass-by-reference semantics to be, in effect, polymorphic. This report details some techniques for differentiating these polymorphic programs using monomorphic Adifor 3.0. In particular, we report on our efforts to compute derivatives for the structural analysis code STAGS.

1 Introduction

In order to use optimization in the design of aerodynamic structures, we were asked to compute derivatives for outputs of the STAGS analysis code. We expected to apply our automatic differentiation (AD) tool, Adifor 3.0, to the STAGS coded to compute the desired derivatives. While we eventually succeeded in differentiating the STAGS code, the task was far from straightforward. The coding practices employed by STAGS developers were clever, but not technically in compliance with the Fortran 77 standard. Adifor depends on standard compliance to produce correct results. Consequently, a great deal of our task involved transforming the STAGS code to a standard-conforming state. This report describes the problems, solutions, and technical justifications for the solutions.

1.1 The STAGS code

The STAGS code computes stresses, strains, loads of of a user-specified structural model. The analytical technique employed by STAGS is the finite element method, using overlapping shells.

The program properties of STAGS are as follows:

*Thanks to Los Alamos Computer Science Institute(LACSI), and NASA for funding this work

- 2504 Fortran files + 10 C files
- 818207 lines (after expansion)
- 17209 call sites

2 Problem Description

Prior to our work on STAGS, many users familiar with STAGS expressed doubt that an AD tool could produce viable derivative code. These users cited several *perceived* problems with the STAGS code:

1. Malloc is used
2. Miscellaneous C utilities employed
3. Heavy use of I/O for differentiable quantities.

Minor source code modifications, however, are sufficient to enable Adifor to differentiate a code with these features.

Nevertheless, the STAGS code was difficult to differentiate. The main problem with STAGS turned out to be *type clashes*. A type clash occurs when (some of) the argument types at a call site do not match the types declared for the routine.

Example:

```
integer ix,iy
...
call sub(ix,iy) ! integer ix does NOT match real
                ! in definition below
...
subroutine sub(x,y)
real x
int y
```

Even though this programming "technique" is officially *not* Fortran 77, all compilers (that I know or have heard about) accept type incorrect code without complaint. In most cases, the computation of such a code will be wrong. Furthermore, Adifor *does* complain about type clashes. In the STAGS code, out of 17209 call sites, 6824 of them had type clashes. This staggeringly large fraction of type clashes proved to be the difficult part of differentiating the STAGS code.

Having identified the type clash problem for STAGS, the next section analyzes these type clashes with an eye to finding a method to allow Adifor to generate a correct derivative code.

3 Polymorphic Programming in Fortran 77

As noted in the previous section, if a program exhibits a type clash, then the computed value will often be wrong. For STAGS, however, there were a massive number

of type clashes, yet the output is not wrong. Rather, the STAGS programmers had been exploiting the lack of type checking in Fortran 77 compilers to write de facto polymorphic routines. The STAGS code used de facto polymorphism heavily.

Aside: What is Polymorphism?

Polymorphism is a property of a subprogram that means that the subprogram will work correctly when supplied with arguments of different type. Fortran 77 supplies a limited number of polymorphic functions. For example, all 4 arithmetic operations ($+, -, *, /$) all work correctly on integer,single precision,double precision, or complex operands. Fortran 77 does not *officially* provide for user-defined polymorphic routines.

The main advantage of polymorphism is code reuse: routines that work on many different types of arguments will not need to be copied.

In STAGS, the polymorphic routines separated into 3 classifications:

1. Memcopy
2. Temporary space reuse
3. Flag selectable

NOTE: The current maintainer of STAGS (Charles Rankin) claims that this polymorphic programming style is typical of strutural analysis codes.

A brief description of each of the polymorphic classifications follows

The memcopy Family

Consider the following code:

```
subroutine mcpy (n,a1,i1,o1,a2,i2,o2)
    integer n,a1(1),i1,l1,o1,a2(1),i2,o2

    ia1 = i1
    ia2 = i2
    do i=1,n
        a2(ia2) = a1(ia1)
        ia1 = ia1 + o1
        ia2 = ia2 + o2
    enddo
end
```

According to the Fortran standard, an integer and a real have the same size. Note, however, that the routine performs no arithmetic: it simply copies values from 1 array to another. So, even though the above routine is defined to take integer arrays as arguments, it will work on real arrays, or double precision arrays as well (although double precision will require adjustments to length and offset arguments that are passed in).

The memcopy family is especially problematic, as Adifor will fail to identify all of the active variables. For example:

```
common /foo/ ibase,rv1(10),rv2(5)
! rv1, rv2 should be active


! Call below is type correct
call mcopy(16,ibase,1,1,target,1,1)
```

The call to `mcopy` is type correct, with `mcopy` defined to take integer arguments. Furthermore, since the `mcopy` is defined to take integers, then it is treated as inactive. This inactivity, in turn, causes derivatives for the `rv1` and rv2— variables to not be copied.

Temporary Space Reuse

The following paradigm is common is many Fortran applications

```
integer ibase(1)
call fmalloc(ibase,MEMSZ,res)
call realr(ibase(res),...) ! realr takes real
                           ! workarray as 1st arg
```

The work array `ibase` is temporary workspace for many routines, including `realr`. The `realr` routine, however, expects a *real* workspace array as the 1st argument. In this case, however, the type of the workspace is irrelevant, as it is just temporary storage The workspace may be reused by many routines.

Flag Selectable

Flag selectable code simulates the overloading feature found in languages like C++ or Fortran 90.

For example:

```
subroutine ick (t1,t2,t3,flag)
    if (flag .eq. 1) then
         call dblpre(t1,t2,t3)
     else if (flag .eq. 2) then
          call sglpre(t1,t2,g3)
    else ...
```

In the above code, either `sglpre` or `dblpre` will operate on the arguments, depending on the value of `flag`. The arguments `t1`,`t2`,`t3`, however, may be either `real` or `double`.

Given the source of type clashes in STAGS derives from a polymorphic programming style, the next section describes a mechanism to enable Adifor to differentiate this type of code

4 Differentiating Polymorphic Programs with Adifor 3

There are 3 possible solutions that come to mind for differentiating polymorphic programs:

1. Simultaneously improve the Adifor activity analysis to cover integer values, and turn off type checking during Adifor processing.

   Improving Adifor activity analysis to cover integer values is a laudable goal, but such an improvement would require major development time. In addition, deactivating the type checking of Adifor would also require significant refactoring (and concomitant development time). This large development time eliminated method 1 from consideration.

2. Rewrite the STAGS code so that it is both type correct and does *not* use polymorphism.

   Rewriting STAGS so that it is both type correct and does not use polymorphism is a daunting task. For every polymorphic call site, a special purpose polymorphic routine must be written. This means that for each common block signature, a separate move routine must be written. The development time for such a task is daunting. In addition, such a massive rewrite violates the spirit of AD tools.

3. Find a (semi) mechanical way to make the code appear type correct via a similarity transform. This method is different from method 2 above in that the rewrite does not have to preserve the semantics of the original program.

   The development time for this method was (perceived as) at least an order of magnitude smaller. Consequently, this was the method of choice.

Before giving the details of the specific similarity transforms for STAGS, the next subsection outlines and overviews similarity transforms in general.

Similarity Transforms

A *similarity transform* for source code is analogous to a similarity transform in group theory : 2 elements a, b are similiar if there is an element S such that

$$a = S^{-1}bS$$

For source code transformation, with respect to AD, let d denote the differentiation source transformation. Then, for any other source transformation P, the source transformation

$$P^{-1}dP$$

is a similarity transform.

Intuitively, an AD-similarity transform first rewrites the initial source code, differentiates the rewritten code, and then undoes the initial rewrite.

The key advantage of a similarity transform is that P need not preserve the original program semantics. The only requirement to ensure that the differentiated code is correct is that (one of) P, P^{-1} commute with the differentiation transformation d.

There are 2 more properties of interest:

1. Source transformations are composable. If P_1, P_2 are source transformations, then so is $P_1 \circ P_2$

2. If a source transformation T preserves the semantics of the program, then there will be no need to apply T^{-1} after the differentiation transform.

The rest of this section describes the 4-component similarity transformation used to differentiate the STAGS code. The 4 components are:

1. Integer-to-real for Polymorphic Routines

2. Nameshifting

3. Equivalencing

4. Shadow Parameters

Each of these transforms will be discussed in following subsections.

Integer-to-real for Polymorphic Routines

All of the naturally polymorphic routines in STAGS were of the memcopy variety. Furthermore, the source and target arrays were of type `integer`. The `integer` type makes the source and target arrays inactive (no derivatives). So, the 1st transformation needed is a change of type from `integer` to `real`. So,

```
subroutine mcpy (n,a1,i1,o1,a2,i2,o2)
    integer n,a1(1),i1,l1,o1,a2(1),i2,o2

    ia1 = i1
    ia2 = i2
    do i=1,n
        a2(ia2) = a1(ia1)
        ia1 = ia1 + o1
        ia2 = ia2 + o2
    enddo
end
```

gets a simple change to

```
subroutine mcpy (n,a1,i1,o1,a2,i2,o2)
    integer n,,i1,l1,o1,i2,o2
    real a1(1),a2(1)

    ia1 = i1
    ia2 = i2
    do i=1,n
        a2(ia2) = a1(ia1)
        ia1 = ia1 + o1
```

```
        ia2 = ia2 + o2
    enddo
end
```

Note that since memcopy routines are naturally polymorphic, the type change does not change the semantics of the program. Consequently, it is not necessary to change the types back after differentiation (although it is harmless).

Nameshifting

Nameshifting is simple transformation that canonically changes the name of a routine at a call site whenever the type of arguments is different than the defined types. For example:

```
double dp(1)
call r1(dp,...) ! double dp mismatched w defn
...
subroutine r1(r,...)
real r
...
```

would nameshift to

```
double dp(1)
call r1_d(dp,...) ! nameshift to r1_d
...
subroutine r1(r,...)
real r
...
```

NOTE: Any routine that has nameshifted variants must get a stub routine for each variant.

Nameshifting suffices for any `real`/`double precision` mismatch, but it cannot correct an `integer`/`real` or `integer`/`double` mismatch.

To see that d (differentiation transform) and nameshifting commute, consider the following:

$$d(\texttt{call r\_d(a,...)}) = \texttt{call g\_r\_d(a,g\_a,...)}$$

Let P be the nameshift operator, so P^{-1} does name unshifting. Then

$$P^{-1}(\texttt{call r\_d(a,...)})) = \texttt{call r(a,...)}$$

So, we have

$$P^{-1}(d(\texttt{call r\_d(a,...)})) = \texttt{call g\_r(a,d\_a,...)}$$

Inverting the order of transformations shows:

$$d(P^{-1}(\texttt{call r\_d(a,...)})) = \texttt{call g\_r(a,d\_a,...)}$$

So, nameshifting is a similarity transform.

Equivalencing

Fortran 77 supports equivalencing of local variables or common block variables. So, for these classes of variables, an equivalence can be used to make a given site type correct. For example, given the definition

```
common /foo/ ibase,dv(10),...
```

we can introduce an equivalence

```
equivalence (ibase,r_ibase)
```

The equivalence definition may be substituted for the original in order to make a call site type correct.

```
call rrr(ibase) ! rrr needs arg of type real
```

could be replaced by

```
call rrr(r_ibase) ! correct type equivalent
```

Equivalence is semantics preserving, so it need not be undone after differentiation.

Shadow parameters

The most complicated transformation employed to differentiate STAGS is shadow parameters. Shadow parameters simulate equivalencing for subroutine parameters, since the equivalence statement only works for common block variables or local variables. Instead, a shadow parameter of the appropriate type is created, and replaced at the call site.
Example: Starting with

```
subroutine s(intp,x1,y1,...)
integer intp
...
call rs(intp,x1,...) ! NOT type correct,
                     ! Rs wants real arg in intp
```

Shadowing the `intp` variable gives:

```
subroutine s(intp,s_intp,x1,y1,...)
           ! s_intp = shadow
integer intp
real s_intp
...
call rs(s_intp,x1) ! Now rs is type correct
```

The shadowing technique fixes type clashes *inside* a routiney, but it introduces argument mismatches for calls to the shadowed routine.

```
! original sub defn          ! shadowed sub defn
subroutine s(ip,x1,...)      subroutine s(ip,s_ip,x1,...)
...                          ...
end ! subroutine s           end ! subroutine s
...                          ...
!                            !
! original correct call      ! origin call now WRONG
!                            !
call s(iv,xx,...) ! ok       call s(iv,xx,...) ! BAD
```

Note that *all* calls to a shadowed routine now have the wrong number of arguments. So, all call sites of the shadowed routine must be fixed. This means that shadow routine call sites must introduce equivalencing or further shadowing. To see this, consider the following code fragment, where `s` is a shadowed routine, that requires a `real` argument shadow for the first argument.

```
subroutine ss(ii,x1,...)
integer ii
common /foo/ iv,r(10),...
...
call s(iv,x1) ! site 1
...
call s(ii,x1) ! site 2
```

Call site 1 requires a shadow for `iv`. Since `iv` is a common block variable, equivalencing will work. This gives

```
subroutine ss(ii,x1,...)
integer ii
common /foo/ iv,r(10),...
real r_iv
equivalence (iv,r_iv)
...
call s(iv,r_iv,x1) ! site 1 shadowed
...
call s(ii,x1) ! site 2
```

For call site 2, however, variable `ii` is a subroutine parameter, so shadowing must be used. The final transformed routine `ss` is

```
subroutine ss(ii,r_ii,x1,...)
integer ii

real r_ii  ! shadow var

common /foo/ iv,r(10),...
```

```
real r_iv
equivalence (iv,r_iv)
...
call s(iv,r_iv,x1) ! site 1 shadowed
...
call s(ii,r_ii,x1) ! site 2 shadowed
```

Note that the shadowing process must terminate, as the top level routine cannot have any call sites. So, equivalencing must eventually terminate the process. To see the commutativity of the inverse shadowing operation S^{-1} with d, the differentiation operation, consider the following:

$$S^{-1}(\texttt{call s(a,s\_a,...)}) = \texttt{call s(a,...)}$$

In other words, S^{-1} simply removes the shadowing variable. It does not, however, remove the associated derivative variables, so

$$S^{-1}(d(\texttt{call s(a,s\_a,...)})) = \texttt{call g\_s(a,g\_a,...)}$$

and similarly

$$d(S^{-1}(\texttt{call s(a,s\_a,...)})) = \texttt{call g\_s(a,g\_a,...)}$$

With this grasp of the components, the next section describes the semi-mechanical implementation of the transform strategy.

5 The Semi-Mechical Transformation Scheme

The semi-mechanical transformation system relies on Adifor to do the type checking, and perl scripts to effect either nameshifting, equivalencing, or shadowing for a given variable / call site.

The algorithm is:

1. Run Adifor (repeatedly), nameshifting all type clashes to discover the natural polymorphic (e.g. memcopy) routines.
2. Change the definition of the natural polymorphic routines to `real` instead of `integer`.
3. Undo the nameshifting from step 1.
4. Run Adifor again to discover new type clashes
5. - nameshift routines if `real`$\Longleftrightarrow$`double`
   - equivalence or shadow if `integer`$\Longrightarrow$`real`

   All of these transforms are accomplished with perl scripts
6. repeat steps 4 and 5 until all type clashes have been fixed.
7. The last run of Adifor will generate derivative code
8. Finally, run the inverse transforms to step 5 on the derivative code