# AD *in* Fortran: Implementation via Prepreprocessor

**Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind**

**Abstract** We describe an implementation of the Farfel Fortran77 AD extensions (Radul et al. AD in Fortran, Part 1: Design (2012), http://arxiv.org/abs/1203.1448). These extensions integrate forward and reverse AD directly into the programming model, with attendant benefits to flexibility, modularity, and ease of use. The implementation we describe is a "prepreprocessor" that generates input to existing Fortran-based AD tools. In essence, blocks of code which are targeted for AD by Farfel constructs are put into subprograms which capture their lexical variable context, and these are closure-converted into top-level subprograms and specialized to eliminate `EXTERNAL` arguments, rendering them amenable to existing AD preprocessors, which are then invoked, possibly repeatedly if the AD is nested.

**Keywords** Nesting • Multiple transformation • Forward mode • Reverse mode • Tapenade • ADIFOR • Programming-language implementation

## 1 Introduction

The *Forward And Reverse Fortran Extension Language* (Farfel) extensions to Fortran77 enable smooth and modular use of AD [7]. A variety of implementation strategies present themselves, ranging from (a) deep integration into a Fortran

A. Radul (✉)
Hamilton Institute, National University of Ireland, Maynooth, Ireland
e-mail: alexey.radul@nuim.ie

B.A. Pearlmutter
Department of Computer Science and Hamilton Institute, National University of Ireland, Maynooth, Ireland
e-mail: barak@cs.nuim.ie

J.M. Siskind
Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA
e-mail: qobi@purdue.edu

S. Forth et al. (eds.), *Recent Advances in Algorithmic Differentiation*, Lecture Notes in Computational Science and Engineering 87, DOI 10.1007/978-3-642-30023-3_25,

compiler, to (b) a preprocessor that performs the requested AD and generates Fortran or some other high-level language, to (c) a prepreprocessor which itself does no AD but generates input to an existing Fortran AD preprocessor. An earlier attempt to implement a dialect of Fortran with syntactic AD extensions bearing some syntactically similarity to Farfel used strategy (a) [5]. Here we adopt strategy (c), which leverages existing Fortran-based AD tools and compilers, avoiding re-implementation of the AD transformations themselves, at the expense of inheriting some of the limitations of the AD tool it invokes.

Farfallen transforms Farfel input into Fortran77, and invokes an existing AD system [2, 3] to generate the needed derivatives. The process can make use of a variety of existing Fortran-based AD preprocessors, easing the task of switching between them. There is significant semantic mismatch between Farfel and the AD operations allowed by the AD systems used, necessitating rather dramatic code transformations. When the Farfel program involves nested AD, the transformations and staging become even more involved. Viewed as a whole, this tool automates the task of applying AD, including the detailed maneuvers required for nested application of existing tools, thereby extending the reach and utility of AD.

The remainder of the paper is organized as follows: Sect. 2 reviews the Farfel extensions. A complete example program on page 277 illustrates their use. Section 3 describes the implementation in detail using this example program. Section 4 summarizes this work's contributions. Comparison with other ways to offer AD to the practitioner can be found in the design paper [7].

## 2 Language Extensions

Farfel provides two principal extensions to Fortran77: syntax for AD and for general nested subprograms.

### *2.1 Extension 1: AD Syntax*

Farfel adds the **ADF** construct for forward AD:

```
ADF(TANGENT(var) = expr ...)
statements
END ADF(var = TANGENT(var) ...)
```

Multiple opening and closing assignments are separated by commas. Independent variables are listed in the "calls" to **TANGENT** on the left-hand sides of the opening assignments and are given the specified tangent values. Dependent variables appear in the "calls" to **TANGENT** on the right-hand sides of the closing assignments and the corresponding tangent values are assigned to the indicated destination variables.

The **ADF** construct uses forward AD to compute the directional derivative of the dependent variables at the point specified by the vector of independent variables in the direction specified by the vector of tangent values for the independent variables and assigns it to the destination variables.

An analogous Farfel construct supports reverse AD:

```
ADR(COTANGENT(var) = expr ...)
statements
END ADR(var = COTANGENT(var) ...)
```

*Dependent* variables are listed in the "calls" to **COTANGENT** on left-hand sides of the opening assignments and are given the specified cotangent values as inputs to the reverse phase. *Independent* variables appear in the "calls" to **COTANGENT** on the right-hand sides of the closing assignments and the corresponding cotangent values at the end of the reverse phase are assigned to the indicated destination variables. The **ADR** construct uses reverse AD to compute the gradient with respect to the independent variables at the point specified by the vector of independent variables induced by the specified gradient with respect to the dependent variables, and assigns it to the destination variables. The expressions used to initialize the cotangent inputs to the reverse phase are evaluated at the end of the forward phase, even though they appear textually prior to the statements specifying the forward phase. This way, the direction input to the reverse phase can depend on the result of the forward phase.

For both **ADF** and **ADR**, implied-**DO** syntax is used to allow arrays in the opening and closing assignments. By special dispensation, the statement **ADF**(*var*) is interpreted as **ADF**(**TANGENT**(*var*)=1) and **ADR**(*var*) as **ADR**(**COTANGENT**(*var*)=1).

## 2.2 Extension 2: Nested Subprograms

In order to conveniently support distinctions between different variables of differentiation for distinct invocations of AD, as in the example below, we borrow from Algol 60 [1] and generalize the Fortran "statement function" construct by allowing subprograms to be defined inside other subprograms, with lexical scope. As in Algol 60, the scope of parameters and declared variables is the local subprogram, and these may shadow identifiers from the surrounding scope. Implicitly declared variables have the top-level subprogram as their scope.

## 2.3 Concrete Example

In order to describe our implementation, we employ a concrete example. The task is to find an equilibrium $(a^*, b^*)$ of a two-player game with continuous scalar strategies $a$ and $b$ and given payoff functions $A$ and $B$. The method is to find roots of

$$a^* = \underset{a}{\operatorname{argmax}}\, A(a, \underset{b}{\operatorname{argmax}}\, B(a^*, b)) \tag{1}$$

The full program is given, for reference, in Listing 7. The heart of the program is the implementation EQLBRM of (1). Note that this whole program is only 63 lines of code, with plenty of modularity boundaries. This code is used as a running example for the remainder of the paper.

## 3 Implementation

Farfel is implemented by the Farfallen preprocessor. The current version is merely a proof of concept, and not production quality: it does not accept the entire Fortran77 language, and does not scale. However, its principles of operation will be unchanged in a forthcoming production-quality implementation. Here we describe the reduction of Farfel constructs to Fortran77, relying on existing Fortran-based AD tools for the actual derivative transformations.

Farfel introduces two new constructs into Fortran77: nested subprograms and syntax for requesting AD. We implement nested subprograms by extracting them to the top level, and communicating the free variables from the enclosing subprogram by passing them as arguments into the new top-level subprogram. This is an instance of *closure conversion*, a standard class of techniques for converting nested subprograms to top-level ones [4]. In order to accommodate passing formerly-free variables as arguments, we must adjust all the call sites of the formerly-nested subprogram; we must specialize all the subprograms that accept that subprogram as an external to also accept the extra closure parameters; and adjust all call sites to all those specialized subprograms to pass those extra parameters.

We implement the AD syntax by constructing new subroutines that correspond to the statements inside each **ADF** or **ADR** block, arranging to call the AD tool of choice on each of those new subroutines, and transforming the block itself into a call to the appropriate tool-generated subroutine.

### 3.1 Nested Subprograms in Detail

Let us illustrate closure conversion on our example. Recall ARGMAX:

```
      FUNCTION ARGMAX(F, X0, N)
        FUNCTION FPRIME(X)
        FPRIME = DERIV1(F, X)
        END
      ARGMAX = ROOT(FPRIME, X0, N)
      END
```

**Listing 7** Complete example Farfel program: equilibria of a continuous-strategy game.

```
C     ASTAR & BSTAR: GUESSES IN, OPTIMIZED VALUES OUT
      SUBROUTINE EQLBRM(BIGA, BIGB, ASTAR, BSTAR, N)
      EXTERNAL BIGA, BIGB
        FUNCTION F(ASTAR)
          FUNCTION G(A)
            FUNCTION H(B)
            H = BIGB(ASTAR, B)
            END
          BSTAR = ARGMAX(H, BSTAR, N)
          G = BIGA(A, BSTAR)
          END
        F = ARGMAX(G, ASTAR, N)-ASTAR
        END
      ASTAR = ROOT(F, ASTAR, N)
      END

      FUNCTION ROOT(F, X0, N)
      X = X0
      DO 1669 I=1,N
      CALL DERIV2(F, X, Y, YPRIME)
 1669 X = X-Y/YPRIME
      ROOT = X
      END

      SUBROUTINE DERIV2(F, X, Y, YPRIME)
      EXTERNAL F
      ADF(X)
      Y = F(X)
      END ADF(YPRIME = TANGENT(Y))
      END

      FUNCTION ARGMAX(F, X0, N)
        FUNCTION FPRIME(X)
        FPRIME = DERIV1(F, X)
        END
      ARGMAX = ROOT(FPRIME, X0, N)
      END

      FUNCTION DERIV1(F, X)
      EXTERNAL F
      ADF(X)
      Y = F(X)
      END ADF(DERIV1 = TANGENT(Y))
      END

      FUNCTION GMBIGA(A, B)
      PRICE = 20-0.1*A-0.1*B
      COSTS = A*(10-0.05*A)
      GMBIGA = A*PRICE-COSTS
      END

      FUNCTION GMBIGB(A, B)
      PRICE = 20-0.1*B-0.0999*A
      COSTS = B*(10.005-0.05*B)
      GMBIGB = B*PRICE-COSTS
      END

      PROGRAM MAIN
      READ *, ASTAR
      READ *, BSTAR
      READ *, N
      CALL EQLBRM(GMBIGA, GMBIGB, ASTAR, BSTAR, N)
      PRINT *, ASTAR, BSTAR
      END
```

This contains the nested function FPRIME. We closure convert this as follows. First, extract FPRIME to the top level:

```
      FUNCTION ARGMAX_FPRIME(X, F)
      ARGMAX_FPRIME = DERIV1(F, X)
      END

      FUNCTION ARGMAX(F, X0, N)
      ARGMAX = ROOT_1(ARGMAX_FPRIME, F, X0, N)
      END
```

Note the addition of a closure argument for F since it is freely referenced in FPRIME, and the addition of the same closure argument at the call site, since FPRIME is passed as an external to ROOT. Then we specialize ROOT to create a version that accepts the needed set of closure arguments (in this case one):

```
      FUNCTION ROOT_1(F, F1, X0, N)
      X = X0
      DO 1669 I=1,N
      CALL DERIV2_1(F, F1, X, Y, YPRIME)
1669  X = X-Y/YPRIME
      ROOT_1 = X
      END
```

Since ROOT contained a call to DERIV2, passing it the external passed to ROOT, we must also specialize DERIV2:

```
      SUBROUTINE DERIV2_1(F, F1, X, Y, YPRIME)
      EXTERNAL F
      ADF(X)
      Y = F(X, F1)
      END ADF(YPRIME = TANGENT(Y))
      END
```

We must, in general, copy and specialize the portion of the call graph where the nested subprogram travels, which in this case is just two subprograms. During such copying and specialization, we propagate external constants (e.g., FPRIME through the call to ROOT_1, the call to DERIV2_1, and the call site therein) allowing the elimination of the **EXTERNAL** declaration for these values. This supports AD tools that do not allow taking derivatives through calls to external subprograms.

That is the process for handling one nested subprogram. In our example, the same is done for F in EQLBRM, G in F, and H in G. Doing so causes the introduction of a number of closure arguments, and the specialization of a number of subprograms to accept those arguments; including perhaps further specializing things that have already been specialized. The copying also allows a limited form of subprogram reentrancy: even if recursion is disallowed (as in traditional Fortran77) our nested

uses of ARGMAX will cause no difficulties because they will end up calling two different specializations of ARGMAX.

Note that we must take care to prevent this process from introducing spurious aliases. For example, in EQLBRM, the internal function F that is passed to ROOT closes over the iteration count N, which is also passed to ROOT separately. When specializing ROOT to accept the closure parameters of F, we must not pass N to the specialization of ROOT twice, lest we run afoul of Fortran's prohibition against assigning to aliased values. Fortunately, such situations are syntactically apparent.

Finally, specialization leaves behind unspecialized (or underspecialized) versions of subprograms, which may now be unused, and if so can be eliminated. In this case, that includes ROOT, DERIV2, ARGMAX, FPRIME, and DERIV1 from the original program, as well as some intermediate specializations thereof.

## 3.2 AD Syntax in Detail

We implement the AD syntax by first canonicalizing each **ADF** or **ADR** block to be a single call to a (new, internal) subroutine, then extracting those subroutines to the top level, then rewriting the block to be a call to an AD-transformed version of the subroutine, and then arranging to call the AD tool of choice on each of those new subroutines to generate the needed derivatives.

Returning to our example program, closure conversion of nested subprograms produced the following specialization of DERIV1:

```
      FUNCTION DERIV1_1(ASTAR, BIGA, BIGB, BSTAR, N, X)
      ADF(X)
      Y = EQLBRM_F_G(X, ASTAR, BIGA, BIGB, BSTAR, N)
      END ADF(DERIV1_1 = TANGENT(Y))
      END
```

which contains an **ADF** block. We seek to convert this into a form suitable for invoking the AD preprocessor. We first canonicalize by introducing a new subroutine to capture the statements in the **ADF** block, producing the following:

```
      FUNCTION DERIV1_1(ASTAR, BIGA, BIGB, BSTAR, N, X)
        SUBROUTINE ADF1()
        Y = EQLBRM_F_G(X, ASTAR, BIGA, BIGB, BSTAR, N)
        END
      ADF(X)
      CALL ADF1()
      END ADF(DERIV1_1 = TANGENT(Y))
      END
```

Extracting the subroutine `ADF1` to the top level as before yields the following:

```
      SUBROUTINE DERIV1_1_ADF1(X, ASTAR, BIGA, BIGB, BSTAR, N, Y)
      Y = G(X, ASTAR, BIGA, BIGB, BSTAR, N)
      END

      FUNCTION DERIV1_1(ASTAR, BIGA, BIGB, BSTAR, N, X)
      ADF(X)
      CALL DERIV1_1_ADF1(X, ASTAR, BIGA, BIGB, BSTAR, N, Y)
      END ADF(DERIV1_1 = TANGENT(Y))
      END
```

Now we are properly set up to rewrite the **ADF** block into a subroutine call—specifically, to a subroutine that will be generated from `DERIV1_1_ADF1` by AD. The exact result depends on the AD tool that will be used to construct the derivative of `DERIV1_1_ADF1`; for Tapenade, the generated code looks like this:

```
      FUNCTION DERIV1_1(ASTAR, BIGA, BIGB, BSTAR, N, X)
      X_G1 = 1
      ASTAR_G1 = 0
      BSTAR_G1 = 0
      CALL DERIV1_1_ADF1_G1(X, X_G1, ASTAR, ASTAR_G1, BIGA, BIGB,
     +BSTAR, BSTAR_G1, N, Y, DERIV1_1)
      END
```

Different naming conventions are used for `DERIV1_1_ADF1_G1` when generating code for ADIFOR; the parameter passing conventions of Tapenade and ADIFOR agree in this case. Farfallen maintains the types of variables in order to know whether to generate variables to hold tangents and cotangents (which are initialized to zero if they were not declared in the relevant opening assignments.)

The same must be repeated for each **ADF** and **ADR** block; in our example there are five in all: two in specializations of `DERIV1` and three in specializations of `DERIV2`. We must also specialize `EQLBRM` and its descendants in the call graph, by the process already illustrated, to remove external calls to the objective functions.

Finally, we must invoke the user's preferred AD tool to generate all the needed derivatives. Here, Farfallen might invoke Tapenade as follows:

```
#! /bin/sh
tapenade -root deriv1_2_adf2 -d -o eqlbrm42 -diffvarname "_g2"\
    -difffuncname "_g2" eqlbrm42.f
tapenade -root deriv2_1_2_adf4 -d -o eqlbrm42 -diffvarname "_g4"\
    -difffuncname "_g4" eqlbrm42{,_g2}.f
tapenade -root deriv1_1_adf1 -d -o eqlbrm42 -diffvarname "_g1"\
    -difffuncname "_g1" eqlbrm42{,_g2,_g4}.f
tapenade -root deriv2_1_1_adf3 -d -o eqlbrm42 -diffvarname "_g3"\
    -difffuncname "_g3" eqlbrm42{,_g2,_g4,_g1}.f
tapenade -root deriv2_2_adf5 -d -o eqlbrm42 -diffvarname "_g5"\
    -difffuncname "_g5" eqlbrm42{,_g2,_g4,_g1,_g3}.f
```

We must take care that multiple invocations of the AD tool to generate the various derivatives occur in the proper order, which is computable from the call graph of

the program, to ensure that generated derivative codes that are used in subprograms to be differentiated are available to be transformed. For example, all the derivatives of `EQLBRM_F_G_H` needed for its optimizations must be generated before generating derivatives of (generated subprograms that call) `EQLBRM_F_G`. We must also take care to invoke the AD tool with different prefixes/suffixes, so that variables and subprograms created by one differentiation do not clash with those made by another.

### *3.3 Performance*

We tested the performance of code generated by Farfallen in conjunction with Tapenade and with ADIFOR. We executed Farfallen once to generate Fortran77 source using Tapenade for the automatic differentiation, and separately targeting ADIFOR. In each case, we compiled the resulting program (`gfortran` 4.6.2-9, 64-bit Debian sid, `-Ofast -fwhole-program`, single precision) with $N = 1,000$ iterations at each level and timed the execution of the binary on a 2.93 GHz Intel i7 870. For comparison, we translated the same computation into vlad [6] (Fig. 1), compiled it with Stalingrad [8], and ran it on the same machine. Stalingrad has the perhaps unfair advantage of being an optimizing compiler with integrated support for AD, so we are pleased that Farfallen was able to achieve performance that was nearly competitive.

| Tapenade | ADIFOR | Stalingrad |
|---|---|---|
| 6.97 | 8.92 | 5.83 |

The vlad code in Fig. 1 was written with the same organization, variable names, subprogram names, and parameter names and order as the corresponding Farfel code in Listing 7 to help a reader unfamiliar with Scheme, the language on which vlad is based, understand how Farfel and vlad both represent the same essential notions of nested subprograms and the AD discipline of requesting derivatives precisely where they are needed. Because functional-programming languages, like Scheme and vlad, prefer higher-order functions (i.e., operators) over the block-based style that is prevalent in Fortran, AD is invoked via the $\overrightarrow{\mathbf{J}}$ and $\overleftarrow{\mathbf{J}}$ operators rather than via the **ADF** and **ADR** statements. However, there is a one-to-one correspondence between

$$\overrightarrow{\mathbf{J}} : f \times x \times \acute{x} \to y \times \acute{y}$$

an operator that takes a function $f$ as input, along with a primal argument $x$ and tangent argument $\acute{x}$ and returns both a primal result $y$ and a tangent result $\acute{y}$, and:

```
ADF (TANGENT (x) = ẋ)
y = f (x)
END ADF (ẏ = TANGENT (y) )
```

$$
\begin{aligned}
&\textsc{eqlbrm}(A,B,a_0,b_0,n) \triangleq \textbf{let } f(a') \triangleq \textbf{let } g(a) \triangleq \textbf{let } h(b) \triangleq B(a',b)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } A(a,\textsc{argmax}(h,b_0,n))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \textsc{argmax}(g,a',n) - a'\\
&\qquad\qquad\qquad\qquad\qquad\qquad a^* = \textsc{root}(f,a_0,n)\\
&\qquad\qquad\qquad\qquad\qquad \textbf{in let } h(b) \triangleq B(a^*,b)\\
&\qquad\qquad\qquad\qquad\qquad\qquad b^* = \textsc{argmax}(h,b_0,n)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } a^*,b^*\\
&\textsc{root}(f,x,n) \triangleq \textbf{if } n = 0\\
&\qquad\qquad\qquad\qquad\quad \textbf{then } x\\
&\qquad\qquad\qquad\qquad\quad \textbf{else let } y,y' = \textsc{deriv2}(f,x)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } \textsc{root}(f,x-\tfrac{y}{y'},n-1)\\
&\textsc{deriv2}(f,x) \triangleq \overrightarrow{\mathbf{J}}(f,x,1)\\
&\textsc{argmax}(f,x_0,n) \triangleq \textbf{let } f'(x) \triangleq \textsc{deriv1}(f,x)\\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \textsc{root}(f',x_0,n)\\
&\textsc{deriv1}(f,x) \triangleq \textbf{let } x,\acute{y} = \overrightarrow{\mathbf{J}}(f,x,1)\\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \acute{y}\\
&\textsc{a}(a,b) \triangleq \textbf{let } \mathit{price} = 20 - 0.1\times a - 0.1\times b\\
&\qquad\qquad\qquad\quad \mathit{costs} = a\times(10 - 0.05\times a)\\
&\qquad\qquad\qquad \textbf{in } a\times\mathit{price} - \mathit{costs}\\
&\textsc{b}(a,b) \triangleq \textbf{let } \mathit{price} = 20 - 0.1\times b - 0.0999\times a\\
&\qquad\qquad\qquad\quad \mathit{costs} = b\times(10.005 - 0.05\times b)\\
&\qquad\qquad\qquad \textbf{in } b\times\mathit{price} - \mathit{costs}\\
&\textbf{let } a_0 = \textsc{readReal}()\\
&\quad b_0 = \textsc{readReal}()\\
&\quad n = \textsc{readReal}()\\
&\quad a^*,b^* = \textsc{eqlbrm}(\textsc{a},\textsc{b},a_0,b_0,n)\\
&\textbf{in } \textsc{writeReal}(a^*)\\
&\quad \textsc{writeReal}(b^*)
\end{aligned}
$$

**Figure 1** Complete vlad program for our concrete example with the same organization and functionality as the Farfel program in Listing 7

Similarly, there is a one-to-one correspondence between

$$\overleftarrow{\mathbf{J}} : f \times x \times \grave{y} \to y \times \grave{x}$$

an operator that takes a function $f$ as input, along with a primal argument $x$ and cotangent $\grave{y}$ and returns both a primal result $y$ and a cotangent $\grave{x}$, and

```
      ADR(COTANGENT(y) = ẏ)
      y = f(x)
      END ADR(ẋ = COTANGENT(x))
```

The strong analogy between how the callee-derives AD discipline is represented in both Farfel and vlad serves two purposes: it enables the use of the Stalingrad compiler technology for compiling Farfel and facilitates migration from legacy imperative languages to more modern, and more easily optimized, pure languages.

## 4 Conclusion

We have illustrated an implementation of the Farfel extensions to Fortran—nested subprograms and syntax for AD [7]. These extensions enable convenient, modular programming using a callee-derives paradigm of automatic differentiation. Our implementation is a preprocessor that translates Farfel Fortran77 extensions into input suitable for an existing AD tool. This strategy enables modular, flexible use of AD in the context of an existing legacy language and tool chain, without sacrificing the desirable performance characteristics of these tools: in the concrete example, only 20–50% slower than a dedicated AD-enabled compiler, depending on which Fortran AD system is used.

**Acknowledgements** This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/I2637, National Science Foundation grant CCF-0438806, Naval Research Laboratory Contract Number N00173-10-1-G023, and Army Research Laboratory Cooperative Agreement Number W911NF-10-2-0060. Any views, opinions, findings, conclusions, or recommendations contained or expressed in this document or material are those of the authors and do not necessarily reflect or represent the views or official policies, either expressed or implied, of SFI, NSF, NRL, ONR, ARL, or the Irish or U.S. Governments. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

## References

1. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language ALGOL 60. The Computer Journal **5**(4), 349–367 (1963). DOI 10.1093/comjnl/5.4.349
2. Bischof, C.H., Carle, A., Corliss, G.F., Griewank, A., Hovland, P.D.: ADIFOR: Generating derivative codes from Fortran programs. Scientific Programming **1**(1), 11–29 (1992)
3. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL http://www.inria.fr/rrrt/rt-0300.html
4. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional Programming Languages and Computer Architecture. Springer Verlag, Nancy, France (1985)

5. Naumann, U., Riehme, J.: A differentiation-enabled Fortran 95 compiler. ACM Transactions on Mathematical Software **31**(4), 458–474 (2005). URL http://doi.acm.org/10.1145/1114268.1114270
6. Pearlmutter, B.A., Siskind, J.M.: Using programming language theory to make automatic differentiation sound and efficient. In: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (eds.) Advances in Automatic Differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 64, pp. 79–90. Springer, Berlin (2008). DOI 10.1007/978-3-540-68942-3_8
7. Radul, A., Pearlmutter, B.A., Siskind, J.M.: AD in Fortran, Part 1: Design (2012). URL http://arxiv.org/abs/1203.1448
8. Siskind, J.M., Pearlmutter, B.A.: Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code. Tech. Rep. TR-ECE-08-01, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA (2008). URL http://docs.lib.purdue.edu/ecetr/367