RWTH Aachen

Department of Computer Science
Technical Report

Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

Michael Maier and Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2006-03

RWTH Aachen · Department of Computer Science · March 2006

Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

Michael Maier and Uwe Naumann

LuFG Informatik 12 (Software and Tools for Computational Engineering)
Department of Computer Science
RWTH Aachen University, D-52056 Aachen Germany
WWW: `http://www.stce.rwth-aachen.de`, Email:
{`maier`|`naumann`}`@stce.rwth-aachen.de`

Abstract. In this paper we report on recent advances made in the development of the first Fortran compiler that provides intrinsic support for computing derivatives. We focus on the automatic generation of intraprocedural adjoint code. Technical details of the modifications made to the internal representation as well as case studies are presented. For example, the new feature allows for the computation of large gradients at a computational cost that is independent of their sizes. Numerous numerical algorithms – derivative-based optimization algorithms in particular – will benefit both from the convenience of the approach and from the efficiency of the intrinsic derivative code.

1 Introduction

We start with an example to motivate the need for adjoints in numerical computing. Consider the problem of minimizing the following function $f : I\!R^n \rightarrow I\!R$ that is due to Griewank [7].

$$y = f(\mathbf{x}) = \sum_{i=1}^{n} \frac{x_i^2}{400} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \quad . \tag{1}$$

It usually serves as a test problem for global optimization algorithms. Interval arithmetic can be used in combination with a branch-and-bound method and with an interval Newton step for local refinement of the enclosure as described in [4] to compute a verified solution to this problem. The algorithm requires the repeated evaluation of the gradient $f'(\mathbf{x}) \in I\!R^n$ at various points $\mathbf{x}$. Moreover, the Hessian $f''(\mathbf{x}) \in I\!R^{n\times n}$ is required to check for concavity of f over a given subinterval. The interval method is merely one way to approach this optimization problem. A large number of algorithms for both local and global optimization have been proposed in the literature. See, for example, [6] and [5] for a comprehensive introduction to these fields. Most algorithms use derivative information to determine descent directions in the context of some kind of search procedure. This is where we come in.

The classical approach to numerical differentiation uses finite difference quotients to approximate the values of derivatives. Centered finite differences defined by

$$f'(\mathbf{x}) \approx \frac{f(\mathbf{x}+\epsilon) - f(\mathbf{x}-\epsilon)}{2\epsilon}$$

are known to exhibit second-order accuracy as opposed to only first order accuracy obtained when using forward or backward differences. Tangent-linear code that can be generated automatically according to the principles of forward mode automatic differentiation [8] produces accurate[1] directional derivatives by computing the inner product

$$\dot{y} =< f'(\mathbf{x}), \dot{\mathbf{x}} >$$

[1] up to machine precision

where $\dot{\mathbf{x}}$ is a direction in the input space $I\!R^n$ and $\dot{y} \in I\!R$. This approach has been taken by the first prototype of the differentiation-enabled NAGWare Fortran compiler [10].

The computational complexity of the accumulation of the full gradient by either finite difference approximation or tangent-linear code is $O(n)$. Every single input needs to be perturbed or, similarly, the derivative in the direction of each Cartesian basis vector needs to be computed. It is well known that for large n neither of the two approaches is feasible. The solution comes in the form of reverse mode automatic differentiation [8]. It builds the basis for semantic source transformation algorithms that generate adjoint code $\bar{f}$ for the computation of

$$\bar{\mathbf{x}} = \bar{f}(\mathbf{x}, \bar{y}) = \bar{y} \cdot f'(\mathbf{x})$$

automatically. The adjoint input vector $\bar{\mathbf{x}}$ becomes equal to the gradient $f'(\mathbf{x})$ when setting $\bar{y} = 1$. A single evaluation of the adjoint code is sufficient (compared with n evaluations of the tangent-linear code).

The theoretical computational complexity suggests that adjoint codes are the method of choice when it comes to the accumulation of large gradients. Unfortunately, writing adjoint codes, and, even more so, generating them automatically, is everything but a trivial task. We will see that the computation of adjoints requires the reversal of the flow of data (and, hence, also the reversal of the flow of control) of the original program that implements $f(\mathbf{x})$. Combine this with the requirement to develop a fully functional compiler front-end for the programming language of your choice and the scope of the development becomes apparent. While many of the basic underlying techniques have been well known for some time now the development of an adjoint code compiler still remains a highly complex task. In particular the generation of efficient adjoint code requires a high level of understanding of the original code resulting in static data-flow analyses tailored toward the specific semantic source transformation [11].

The work described in this paper builds on previous prototypes of the differentiation-enabled NAGWare Fortran compiler. We focus on the transformation of single subroutines $F(\mathbf{x}, \mathbf{y})$ that compute outputs $\mathbf{y} \in I\!R^m$ as functions of given inputs $\mathbf{x} \in I\!R^n$ into adjoint subroutines $\bar{F}(\mathbf{x}, \bar{\mathbf{y}}, \bar{\mathbf{x}})$ for computing $\bar{\mathbf{x}} = F'(\mathbf{x})^T \cdot \bar{\mathbf{y}}$, where $F'(\mathbf{x})$ is the Jacobian of F and $F'(\mathbf{x})^T$ is its transposed.

The structure of this paper is as follows: In Section 2 we review the basic principles of tangent-linear and adjoint codes in the shape of the forward and reverse modes of automatic differentiation. We summarize the features of previous prototypes of the differentiation-enabled NAGWare Fortran compiler that provide both tangent-linear and adjoint capabilities. The heart of this paper is represented by Section 3. Therein we discuss the novel extension of the compilers adjoint capabilities. Technical details about how the control- and data-flow reversals are realized are illustrated with the help of intuitive examples. Case studies are presented in Section 4. We draw conclusions in Section 5 and we give a brief outlook to future work.

2 Preliminaries and Summary of Prior Work

We consider computer programs that evaluate vector functions $\mathbf{y} = F(\mathbf{x})$ with

$$F : I\!R^n \to I\!R^m \quad .$$

2.1 Automatic Differentiation

Automatic differentiation (AD) expects the evaluation of F to decompose into a sequence of elemental computational operations

$$v_j = \varphi_j(v_i)_{i \prec j} \qquad \text{for } j = 1, \ldots, q \quad . \tag{2}$$

We adopt Griewank's notation [8]. Hence, $i \prec j$ if v_i is an argument of φ_j and $q = p + m$. We set $v_j = x_{n-j}$ for $j = n-1, \ldots, 0$ and $y_j = v_{p+j}$ for $j = 1, \ldots, m$. Refer to [8] for a comprehensive discussion of AD.

Under the usual assumptions about differentiability of the elemental functions we can compute local partial derivatives

$$c_{ji} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \qquad \text{for } j = 1, \ldots, q \quad . \tag{3}$$

Forward mode AD computes $\dot{\mathbf{y}} = F' \cdot \dot{\mathbf{x}}$ where $F' \equiv F'(\mathbf{x})$ is the Jacobian matrix of F as

$$\dot{v}_j = \sum_{i \prec j} c_{ji} \cdot \dot{v}_i \qquad \text{for } j = 1, \ldots, q \quad . \tag{4}$$

Alternatively, the transposed product $\bar{\mathbf{x}} = (F')^T \cdot \bar{\mathbf{y}}$ can be computed by *incremental reverse mode AD* (see Section 3.2) after initializing $\bar{v}_i = 0$ for $i = 1 - n, \ldots, p$ as

$$\bar{v}_i += c_{ji} \cdot \bar{v}_j \qquad \text{for } j = p, \ldots, 1 - n, \; i \prec j \quad . \tag{5}$$

Such adjoints are of particular interest in large-scale optimization since they allow for gradients to be computed with a computational complexity that is independent of n.

Example Consider the simple function $y = \sin(x_1 \cdot x_2)$. The equivalent of Equation (2) becomes

$$v_{-1} = x_1; \quad v_0 = x_2; \quad v_1 = v_{-1} \cdot v_0; \quad v_2 = \sin(v_1); \quad y = v_2 \quad .$$

The generation of tangent-linear and adjoint codes is based on such *statement-level single assignment code*. Applying Equation (3) and Equation (4) we get the tangent-linear code

$$
\begin{aligned}
\dot{v}_{-1} &= \dot{x}_1 && // \quad \frac{\partial v_{-1}}{\partial x_1} = 1 \\
v_{-1} &= x_1 \\
\dot{v}_0 &= \dot{x}_2 && // \quad \frac{\partial v_0}{\partial x_2} = 1 \\
v_0 &= x_2 \\
\dot{v}_1 &= \dot{v}_{-1} \cdot v_0 + v_{-1} \cdot \dot{v}_0 && // \quad c_{1,-1} \equiv \frac{\partial v_1}{\partial v_{-1}} = v_0 \qquad c_{1,0} \equiv \frac{\partial v_1}{\partial v_0} = v_{-1} \\
v_1 &= v_{-1} \cdot v_0 \\
\dot{v}_2 &= \cos(v_1) \cdot \dot{v}_1 && // \quad c_{2,1} \equiv \frac{\partial v_2}{\partial v_1} = \cos(v_1) \\
v_2 &= \sin(v_1) \\
\dot{y} &= \dot{v}_2 && // \quad \frac{\partial y}{\partial v_2} = 1 \\
y &= v_2 && .
\end{aligned}
$$

We use C-style // to marks comments. Specifically we list the local partial derivatives of the left-hand side of each elementary assignment with respect to the variables on the right-hand side.

According to Equation (5) the adjoint code requires the values of intermediate variables in reverse order. Hence, it is preceded by the evaluation of these values as follows:

$$
\begin{aligned}
v_{-1} &= x_1 && // \quad \text{begin augmented forward code} \\
v_0 &= x_2 \\
v_1 &= v_{-1} \cdot v_0 \\
v_2 &= \sin(v_1) \\
y &= v_2 && // \quad \text{end augmented forward code} \\
\bar{v}_2 &= \bar{y} && // \quad \text{begin adjoint code} \qquad \frac{\partial y}{\partial v_2} = 1 \\
\bar{v}_1 &= \cos(v_1) \cdot \bar{v}_2 && // \quad c_{2,1} = \cos(v_1) \\
\bar{v}_0 &= v_{-1} \cdot \bar{v}_1 && // \quad c_{1,0} = v_{-1} \\
\bar{v}_{-1} &= v_0 \cdot \bar{v}_1 && // \quad c_{1,-1} = v_0 \\
\bar{x}_2 &= \bar{v}_0 && // \quad \frac{\partial v_0}{\partial x_2} = 1 \\
\bar{x}_1 &= \bar{v}_{-1} && // \quad \text{end adjoint code} \qquad \frac{\partial v_{-1}}{\partial x_1} = 1
\end{aligned}
$$

In a larger context we need to set $\bar{y} = 0$ after the first adjoint statement and we need to increment the adjoints of x_i (see Section 3.2).

2.2 Previous Prototypes of the Compiler

The current research prototype of the differentiation-enabled NAGWare Fortran 95 compiler is based on three previous prototypes.

In [9] we describe an approach to the generation of tangent-linear code that is based on operator overloading. The compiler provides an intrinsic Fortran module that contains redefined arithmetic operators and intrinsic functions for variables of a new derived data type. The latter contains variables for the function value as well as for the directional derivative. The advantage of this method is its high degree of robustness that can be achieved quite easily by simply relying on the compilers resolution method for overloaded operations.

The efficiency of the tangent-linear code can be improved by preaccumulation of local gradients at the level of scalar assignments. The operations count is reduced considerably if a large number of directional derivatives is computed. Refer to [10] for further details on this approach.

The statement-level gradients can be accumulated and stored during an evaluation of correspondingly augmented forward code. In order to propagate adjoints these gradients can be restored and used in reverse order inside the adjoint code. This method has been proposed in [14]. Its main disadvantage is the requirement of hashing memory addresses in order to access the correct adjoint memory. This address translation is typically slow. Hence, we launched the next stage of the CompAD project in order to overcome this problem by generating intraprocedural adjoint code based on explicit data- and control-flow reversal as described below.

3 Intrinsic Intraprocedural Adjoint Code

Our algorithm transforms a given abstract syntax tree (AST) of the original numerical program into one that represents the adjoint program. The unparser walks the transformed AST to generate the adjoint code in C. Thus we are in line with the NAGWare Fortran 95 compiler's strategy that transforms the given Fortran code into optimized C and then uses a native C compiler to generate efficient machine code for the respective target architecture. The transformation is best illustrated by an example.

3.1 Example

Consider a subroutine $\mathrm{FUNC}(\texttt{C},\texttt{A},\texttt{B})$ that contains the single statement

$$\texttt{C} = \mathrm{SIN}(\texttt{A} \cdot \texttt{B}) \quad .$$

Original AST The internal representation of this code inside the NAGWare Fortran compiler is as follows.

```
SUBROUTINE FUNC |Node [Id:43] [Internal Subprogram]
                |+- Node [Id:24] [Subroutine]
                ||   +- Leaf [Id:18] ("FUNC") [Symbol]
...             ||   +- (Argument List and Type Declarations)
C = SIN(A · B)  |+- Node [Id:41] [Assignment]
                ||   +- Leaf [Id:35] ("C") [Ident.]
                ||   +- Node [Id:40] [Function Reference]
                ||   |   +- Leaf [Id:36] [Procedure] ("SIN") [Symbol]
                ||   |   +- Node [Id:39] [Multiplication (*)]
                ||   |   |   +- Leaf [Id:37] ("A") [Ident.]
                ||   |   |   +- Leaf [Id:38] ("B") [Ident.]
END SUBROUTINE  |+- Leaf [Id:42] [Program Subtree End]
```

We use our own tree printer routine to generate the above output. The main purpose of this mechanism lies in debugging. We use a custom exception type that generates a printout of the AST each time it is raised. Thus we have full access to the present state of the AST. This support has turned out as extremely useful for past and ongoing development.

Transformed AST We generate statement-level single assignment code as described in the previous section. The values of certain auxiliary variables are then required in reverse order by the adjoint code to evaluate the values of the local partial derivatives of the elementary arithmetic operators and intrinsic functions. Note that required values may be overwritten by succeeding statement-level single assignment codes unless the program consists of only a single statement. Hence these values need to be stored by the augmented forward code and restored later by the adjoint code. Lacking data-flow analysis one needs to assume conservatively that the values of all auxiliary variables are required by the adjoint code. Therefore the statement-level single assignment code is augmented with `PUSH` statements preceding each of its assignments. The corresponding `POP`'s occur in the adjoint code.

```
SUBROUTINE FUNC     | Node [Id:43] [Internal Subprogram]
                    | +- Node [Id:24] [Subroutine]
                    | |  +- Leaf [Id:18] ("FUNC") [Symbol]
...                 | |  +- (Argument List and Type Declarations)
--------------------+------------------------------------------------
                    | Augmented Subroutine Body:
PUSH(v0)            | +- Node [Id:49] [Generic (Overloaded) CALL]
                    | |  +- Leaf [Id:48] ("PUSH") [Symbol]
                    | |  +- Leaf [Id:47] ("v0") [Ident.]
v0 = B              | +- Node [Id:51] [Assignment]
                    | |  +- Leaf [Id:46] ("v0") [Ident.]
                    | |  +- Leaf [Id:38] ("B") [Ident.]
PUSH(v1)            | +- Node [Id:55] [Generic (Overloaded) CALL]
                    | |  +- Leaf [Id:54] ("PUSH") [Symbol]
                    | |  +- Leaf [Id:53] ("v1") [Ident.]
v1 = A              | +- Node [Id:57] [Assignment]
                    | |  +- Leaf [Id:52] ("v1") [Ident.]
                    | |  +- Leaf [Id:37] ("A") [Ident.]
PUSH(v2)            | +- Node [Id:63] [Generic (Overloaded) CALL]
                    | |  +- Leaf [Id:62] ("PUSH") [Symbol]
                    | |  +- Leaf [Id:61] ("v2") [Ident.]
v2 = v1 × v0        | +- Node [Id:65] [Assignment]
                    | |  +- Leaf [Id:60] ("v2") [Ident.]
                    | |  +- Node [Id:39] [Multiplication (*)]
                    | |  |  +- Leaf [Id:58] ("v1") [Ident.]
                    | |  |  +- Leaf [Id:59] ("v0") [Ident.]
PUSH(v3)            | +- Node [Id:70] [Generic (Overloaded) CALL]
                    | |  +- Leaf [Id:69] ("PUSH") [Symbol]
                    | |  +- Leaf [Id:68] ("v3") [Ident.]
v3 = SIN(v2)        | +- Node [Id:72] [Assignment]
                    | |  +- Leaf [Id:67] ("v3") [Ident.]
                    | |  +- Node [Id:40] [Function Reference]
                    | |  |  +- Leaf [Id:36] [Procedure] ("SIN") [Symbol]
                    | |  |  +- Leaf [Id:66] ("v2") [Ident.]
PUSH(C)             | +- Node [Id:76] [Generic (Overloaded) CALL]
                    | |  +- Leaf [Id:75] ("PUSH") [Symbol]
                    | |  +- Leaf [Id:74] ("C") [Ident.]
C = v3              | +- Node [Id:41] [Assignment]
                    | |  +- Leaf [Id:35] ("C") [Ident.]
                    | |  +- Leaf [Id:73] ("v3") [Ident.]
```

	Reversed & Adjoint Subroutine Body:
$\bar{v}_3 = \bar{c}$	+- Node [Id:89] [Assignment]
	│ +- Leaf [Id:87] ("$\bar{v}_3$") [Ident.]
	│ +- Leaf [Id:88] ("$\bar{c}$") [Ident.]
$\bar{c} = 0$	+- Node [Id:92] [Assignment]
	│ +- Leaf [Id:91] ("$\bar{c}$") [Ident.]
	│ +- Leaf [Id:90] [Constant] [INTEGER Constant]
POP(C)	+- Node [Id:95] [Generic (Overloaded) CALL]
	│ +- Leaf [Id:94] ("POP") [Symbol]
	│ +- Leaf [Id:93] ("C") [Ident.]
$\bar{v}_2$ = COS(v_2) $\times$ $\bar{v}_3$	+- Node [Id:103] [Assignment]
	│ +- Leaf [Id:98] ("$\bar{v}_2$") [Ident.]
	│ +- Node [Id:102] [Multiplication (*)]
	│ │ +- Node [Id:101] [Function Reference]
	│ │ │ +- Leaf [Id:100] ("COS") [Symbol]
	│ │ │ +- Leaf [Id:99] ("v_2") [Ident.]
	│ │ +- Leaf [Id:97] ("$\bar{v}_3$") [Ident.]
POP(v_3)	+- Node [Id:106] [Generic (Overloaded) CALL]
	│ +- Leaf [Id:105] ("POP") [Symbol]
	│ +- Leaf [Id:104] ("v_3") [Ident.]
$\bar{v}_1 = \bar{v}_2 \times v_0$	+- Node [Id:113] [Assignment]
	│ +- Leaf [Id:108] ("$\bar{v}_1$") [Ident.]
	│ +- Node [Id:112] [Multiplication (*)]
	│ │ +- Leaf [Id:110] ("$\bar{v}_2$") [Ident.]
	│ │ +- Leaf [Id:111] ("v_0") [Ident.]
$\bar{v}_0 = \bar{v}_2 \times v_1$	+- Node [Id:119] [Assignment]
	│ +- Leaf [Id:114] ("$\bar{v}_0$") [Ident.]
	│ +- Node [Id:118] [Multiplication (*)]
	│ │ +- Leaf [Id:116] ("$\bar{v}_2$") [Ident.]
	│ │ +- Leaf [Id:117] ("v_1") [Ident.]
POP(v_2)	+- Node [Id:122] [Generic (Overloaded) CALL]
	│ +- Leaf [Id:121] ("POP") [Symbol]
	│ +- Leaf [Id:120] ("v_2") [Ident.]
$\bar{a}$ += $\bar{v}_1$	+- Node [Id:130] [Assignment]
	│ +- Leaf [Id:124] ("$\bar{a}$") [Ident.]
	│ +- Node [Id:129] [Addition (+)]
	│ │ +- Leaf [Id:127] ("$\bar{a}$") [Ident.]
	│ │ +- Leaf [Id:128] ("$\bar{v}_1$") [Ident.]
$\bar{v}_1 = 0$	+- Node [Id:133] [Assignment]
	│ +- Leaf [Id:132] ("$\bar{v}_1$") [Ident.]
	│ +- Leaf [Id:131] [Constant] [INTEGER Constant]
POP(v_1)	+- Node [Id:136] [Generic (Overloaded) CALL]
	│ +- Leaf [Id:135] ("POP") [Symbol]
	│ +- Leaf [Id:134] ("v_1") [Ident.]
$\bar{b}$ += $\bar{v}_0$	+- Node [Id:144] [Assignment]
	│ +- Leaf [Id:138] ("$\bar{b}$") [Ident.]
	│ +- Node [Id:143] [Addition (+)]
	│ │ +- Leaf [Id:141] ("$\bar{b}$") [Ident.]
	│ │ +- Leaf [Id:142] ("$\bar{v}_0$") [Ident.]
$\bar{v}_0 = 0$	+- Node [Id:147] [Assignment]
	│ +- Leaf [Id:146] ("$\bar{v}_0$") [Ident.]
	│ +- Leaf [Id:145] [Constant] [INTEGER Constant]
POP(v_0)	+- Node [Id:150] [Generic (Overloaded) CALL]
	│ +- Leaf [Id:149] ("POP") [Symbol]
	│ +- Leaf [Id:148] ("v_0") [Ident.]
END SUBROUTINE	+- Leaf [Id:42] [Program Subtree End]

A runtime support module is provided that contains implementations for `PUSH` and `POP`.

3.2 Data-Flow Reversal

The adjoint code uses certain intermediate values in reverse order to determine the values of the local partial derivatives. Hence, the values of these variables need to be made available somehow. The two fundamental approaches to this problem are the so-called "store-all" and "recompute-all" strategies. While the latter results in an unacceptable quadratic computational complexity the former may lead to infeasible memory requirements for very large programs. In order to be able to generate adjoints for large-scale numerical simulation programs one has to follow a hybrid strategy that stores some values in so-called *checkpoints* while recomputing others from these checkpoints. The implementation of such a checkpointing scheme inside the differentiation-enabled NAGWare Fortran 95 compiler is the subject of future work. For the time being we pursue a "store-all" strategy. Lacking AD-specific data-flow analysis [11] we have implemented a "copy-on-def" method that saves the values of all program variables prior to them getting overwritten. For example,

$$
\begin{aligned}
z &= \sin(x) \\
x &= y * z
\end{aligned}
$$

becomes

$$
\begin{aligned}
&\mathbf{call}\ PUSH(z) \\
&z = \sin(x) \\
&\mathbf{call}\ PUSH(x) \\
&x = x * z \\
&\mathbf{call}\ POP(x) \\
&\bar{z} = \bar{z} + x \cdot \bar{x}; \quad \bar{x} = z \cdot \bar{x} \\
&\mathbf{call}\ POP(z) \\
&\bar{x} = \bar{x} + \cos(x) \cdot \bar{z}; \quad \bar{z} = 0
\end{aligned}
$$

Let us have a closer look at this code in order to understand the way the adjoint code is generated.

Incremental Adjoint Code In incremental reverse mode the local partial derivatives are computed during the adjoint evaluation (second line in Equation (6)).

$$
\begin{array}{l}
v_j = \varphi_j(v_i)_{i \prec j}; \;\; \bar{v}_j = 0 \quad \text{for } j = 1, \ldots, q \\
c_{j,i} = \dfrac{\partial \varphi_j}{\partial v_i}; \;\; \bar{v}_i = \bar{v}_i + c_{j,i} \cdot \bar{v}_j \quad \text{for } i \prec j \text{ and } j = q, \ldots, 1
\end{array} . \tag{6}
$$

Consider an arbitrary assignment of the form

$$
v_j = \varphi_j(v_i)_{i \prec j} \tag{7}
$$

where, possibly, $\&v_j \in \{\&v_i : i \prec j\}$ as well as $\&v_{i_1} = \&v_{i_2}$ for $i_1 \prec j$ and $i_2 \prec j$. We use $\&v$ to denote the address of a variable v in memory.

In general, it is undecidable at compile time if $\&v_{i_1} = \&v_{i_2}$ as this may depend on parameters that are only available at run time. Alias analysis [15] may provide some insight. Lacking any program analysis one needs to assume conservatively that $\&v_j = \&v_i$ for all $i \prec j$ to ensure the correctness of the adjoint code.

As a consequence of overwriting a given location in memory may represent different code list variables. Note that the adjoints of all code list variables need to be initialized with zero to ensure the correctness of the incremental reverse mode. The adjoint of v_j in Equation (7) dies (its value is no longer used) once it has been used to increment the adjoints of all arguments of φ_j. However, the memory location $\&v_j$ may well be incremented by some succeeding adjoint statement for the reasons stated above. Hence, adjoint variables need to be reset to zero immediately after their death.

If we can prove that some $\&v_j$ is always read only once after its initialization and before getting overwritten, then its adjoint does not need to be initialized with zero and all adjoint statements that have $\bar{v}_j$ on the left-hand side simply overwrite its value. Building the statement-level single assignment code as described in the Example in Section 2.1 ensures that the above holds for all auxiliary non-program variables. Hence, adjoints of auxiliary variables are always overwritten and need neither be initialized nor reset to zero at their death. It is guaranteed that they are defined before being used.

3.3 Control-Flow Reversal

The reversal of the flow of data implies the reversal of the flow of control. We need to provide a method that allows us to follow the original flow of control in reverse in the adjoint code. W.l.o.g., we assume that variable flow of control is caused only by branch statements (`IF-THEN`, `IF-THEN-ELSE`) and loops (`DO`, `DO-WHILE`). Their representations in the abstract syntax tree are then handled as described in the following. Similar approaches are used by the AD tools TAPENADE [12] and OpenAD [17].

Branches

Original `IF-THEN` statements are always transformed into `IF-THEN-ELSE` statements with an empty `ELSE` branch. Hence, the resulting AST looks as follows:

```
IF (···) THEN |+- Node [Id:40] [Block IF Statement]
              ||  +- (Branch Condition)
···           |+- (Branch(true)-Body)
ELSE          |+- Leaf [Id:46] [ELSE]
···           |+- (Branch(false)-Body)
ENDIF         |+- Leaf [Id:52] [END IF]
```

Augmented Forward In order to memorize the flow through the `IF-THEN-ELSE` statement we augment the original forward computation with statements that push a branch id (1 for `IF` branch; 0 for `ELSE` branch) onto a special control stack.

```
IF (···) THEN        |+- Node [Id:40] [Block IF Statement]
                     |  +- (Branch Condition)
···                  |+- (Branch(true)-Body)
CONTROLPUSH(1)       |+- Node [Id:67] [Generic (Overloaded) CALL]
                     |   +- Leaf [Id:68] ("CONTROLPUSH") [Specific Name]
                     |   +- Leaf [Id:66] ("CONTROLPUSH") [Ident./Sym.Name]
                     |   +- Leaf [Id:65] [Constant] [INTEGER Constant]
ELSE                 |+- Leaf [Id:46] [ELSE]
···                  |+- (Branch(false)-Body)
CONTROLPUSH(0)       |+- Node [Id:79] [Generic (Overloaded) CALL]
                     |   +- Leaf [Id:80] ("CONTROLPUSH") [Specific Name]
                     |   +- Leaf [Id:78] ("CONTROLPUSH") [Ident./Sym.Name]
                     |   +- Leaf [Id:77] [Constant] [INTEGER Constant]
ENDIF                |+- Leaf [Id:52] [END IF]
```

The subroutines CONTROLPUSH and CONTROLPOP are part of the runtime support module.

Adjoint The reverse section of the adjoint code retrieves the branch id from the control stack. Depending on its value it then executes the adjoint of the IF or ELSE branches.

```
CONTROLPOP(_b)         |+- Node [Id:93] [Generic (Overloaded) CALL]
                       |   +- Leaf [Id:94] ("CONTROLPOP") [Specific Name]
                       |   +- Leaf [Id:92] ("CONTROLPOP") [Ident./Sym.Name]
                       |   +- Leaf [Id:91] ("_b") [Ident./Sym.Name]
IF (0 .EQ. _b) &       |+- Node [Id:102] [Block IF Statement]
THEN                   |   +- Node [Id:100] [.EQ.]
                       |   |   +- Leaf [Id:99] [Constant] [INTEGER Constant]
                       |   |   +- Leaf [Id:101] ("_b") [Ident./Sym.Name]
···                    |+- (Adjoint Branch(false)-Body)
ELSE                   |+- Leaf [Id:119] [ELSE]
···                    |+- (Adjoint Branch(true)-Body)
ENDIF                  |+- Leaf [Id:134] [END IF]
```

The explicit conversion of IF−THEN statements into IF−THEN−ELSE statements is necessary to ensure a balanced control stack.

Loops

Original DO−WHILE loops result in an AST as the following:

```
DO WHILE (···)       |+- Node [Id:31] [DO]
                     |    +- Node [Id:30] [WHILE]
                     |    |    +- (Loop Condition)
  ···                |+- (Loop Body)
                     |+- ···
END DO               |+- Leaf [Id:44] [END DO]
```

DO loops are analogous.

Augmented Forward One way to reverse the flow of control is to enumerate all basic blocks and to simply push their indexes onto the control stack during the execution of the augmented forward code [16]. This approach may lead to unacceptably high memory requirements in the presence of loops as the number of values pushed onto the stack is multiplied with the number of loop iterations. To overcome this problem one may simply count the number of iterations performed by the loop in the augmented forward section and store this value for later use by the reverse section of the adjoint code. This idea results in the following augmented forward code:

```
_i = 0              |+- Node [Id:57] [Assignment]
                    ||   +- (Initial Assignment)
DO WHILE (···)      |+- Node [Id:31] [DO]
  ···               ||   +- (Loop Condition + Body)
  _i = _i + 1       |+- Node [Id:66] [Assignment]
                    ||   +- Leaf [Id:62] ("_i") [Ident.]
                    ||   +- Node [Id:64] [Addition (+)]
                    ||   |   +- Leaf [Id:63] ("_i") [Ident.]
                    ||   |   +- Leaf [Id:65] [INTEGER Constant]
END DO              |+- Leaf [Id:44] [END DO]
CONTROLPUSH (_i)    |+- Node [Id:69] [Generic (Overloaded) CALL]
                    ||  +- Leaf [Id:70] ("CONTROLPUSH") [Specific Name]
                    ||  +- Leaf [Id:68] ("CONTROLPUSH") [Ident./Sym.Name]
                    ||  +- Leaf [Id:67] ("_i") [Ident./Sym.Name]
```

Adjoint The adjoint loop is implemented as a DO loop that performs as many iterations as the original loop while executing the adjoint of the loop body.

```
CONTROLPOP (_i)    |+- Node [Id:85] [Generic (Overloaded) CALL]
                   ||   +- Leaf [Id:86] ("CONTROLPOP") [Specific]
                   ||   +- Leaf [Id:84] ("CONTROLPOP") [Ident./Sym.Name]
                   ||   +- Leaf [Id:83] ("_i") [Ident./Sym.Name]
DO _c=1, _i        |+- Node [Id:91] [DO]
                   ||   +- Node [Id:90] [Do Specification]
                   ||   |   +- Leaf [Id:87] ("_c") [Ident./Sym.Name]
                   ||   |   +- Leaf [Id:88] [INTEGER Constant]
                   ||   |   +- Leaf [Id:89] ("_i") [Ident./Sym.Name]
    ···            |+- (Adjoint Loop Body)
END DO             |+- Leaf [Id:44] [END DO]
```

3.4 Software Development Issues

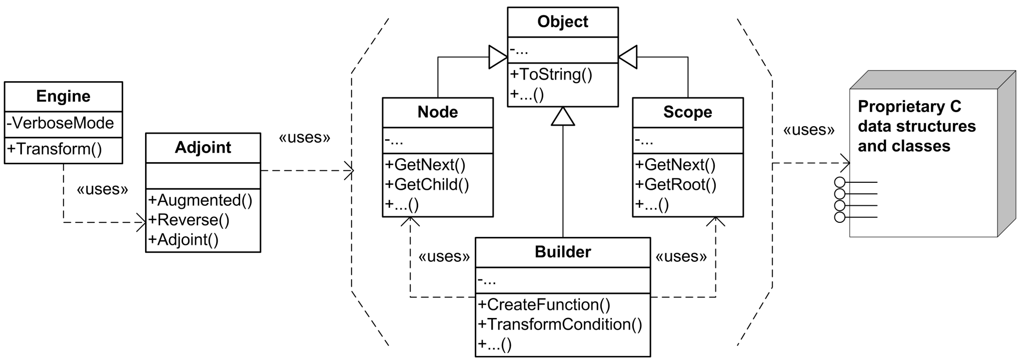

Fig. 1. C++-Interface to the NAGWare Fortran Compilers C-API

The described parse tree transformations are handled by an adjoint transformation engine based on an object-oriented interface to the original compiler API provided by the NAGWare compiler. Figure 1 summarizes the three layers of this interface.

The first layer is the low-level API containing proprietary C data structures, macros, and functions that are used by the compiler itself in the standard compilation process. The second layer is based on these data structures and classes. This layer is a C++

interface introducing an object-oriented model to the parse tree providing access to and the ability to modify parse tree nodes, scopes, and entries in associated string tables. Therefore the layer itself is organized in a hierarchical class structure starting with the basic `Object` class that every other class is derived from to offer consistency in the use of objects. A `Scope` class instance can move through every single scope using the `GetNext()` method. `GetRoot()` returns the root node of the associated parse tree as an instance of the `Node` class. A `Node` object also offers a `GetNext()` method to get the following node in the parse tree or an associated child node using `GetChild()`. Every class instance has a `ToString()` method that returns a description of the object. For example, all parse tree visualizations in this paper are rendered by the `Node` class implementation of `ToString()`.

By an own set of exception classes Exceptions are handled by a set of exception classes. Debugging of complex parse tree structures is simplified considerably. The user is informed through detailed error messages and parse tree dumps are generated to isolate erroneous structures.

Builder-classes provide support for parse tree modifications. They handle complex operations, such as adding and removing nodes and variables, with very few method calls. This is where the third layer starts. It contains the two classes `Adjoint` and `Engine`. The latter is a driver-like class an instance of which is created by the compiler whenever adjoint code generation is requested. The `Engine` object then iterates through all scopes searching for subroutines and applies adjoint code transformation to them. The latter is implemented by the `Adjoint` class in two stages: First the original subroutine is augmented for the subsequent data- and control-flow-reversal and, second, the adjoint code is generated.

Though there are several advantages and disadvantages in choosing an object-oriented API, compatibility to other AD projects that use an object-oriented interface is the greatest advantage. We are primarily interested in compatibility with OpenAD [2] and, in particular, with its algorithmic core called `xaifBooster` [18].

4 Case Studies

4.1 Griewank

Extensive tests were performed with Griewank's function implemented in Fortran 90 as follows:

```
...

c = 1
d = 1.D0
DO i = 1, s
     c = c + (a(i)**2) / 400
     d = d * COS (a(i) / SQRT (DBLE (i)))
END DO
c = c - d
...
```

In Table 2 we list the results obtained by running the adjoint code vs. finite differences and we show the absolute error for $n = 5$ and $x_i = 1$ for $i = 1, \ldots, 5$.. The latter is given in single precision whereas the former are in double precision. The following driver was used to generate the results.

[2] `http://www.mcs.anl.gov/openad`

```
...

! Compute gradient with intrinsic adjoint
c_   = 1.D0
CALL griewank_adj (s, a, a_, c, c_)

! Approximate gradient by forward finite differences
CALL griewank (s, a, c)
DO i = 1, s
    ! Perturb current input vector element
    ae(i) = a(i) + e
    CALL griewank (s, ae, c1)
    ! Restore input vector
    ae(i) = a(i)
    ! Compute divided difference
    b (i) = (c1 - c) / e
END DO
...
```

Adjoint	Finite Differences	Absolute Error
0.4291500041640747	0.4291500188546848	$0.0000000e^{\pm 00}$
0.1695581977732323	0.1695582052541056	$-1.4901161e^{-08}$
0.1074226956746736	0.1074227018449392	$-7.4505806e^{-09}$
$7.9390989486530394e^{-02}$	$7.9390993779071550e^{-02}$	$-7.4505806e^{-09}$
$6.3416055468902388e^{-02}$	$6.3416059070675601e^{-02}$	$0.0000000e^{\pm 00}$

Table 2. Gradient by Adjoints vs. Finite Differences for Griewank's test problem

The runtime behavior is documented in Figure 2. We observe that the overhead due to storing and restoring values of intermediate variables is negligible for the considered problem sizes. The adjoint code performs very well. The computational complexity of the finite difference approximation grows with the number of inputs. The vertical axis shows relative CPU times. The actual values depend heavily on the hardware platform.

4.2 Bratu

We consider a variant of the Bratu problem from the MINPACK test problem collection [1] given as the following Fortran code.

```
...

h = 2.0 / (dim + 1)

F(1) = -2 * x(1) + h * h * prm(1) / &
      12.0 * (1 + 10 * exp (x(1) / (1.0 + prm(2) * x(1))))
F(2) =      x(1) + h * h * prm(1) / &
      12.0 *          exp (x(1) / (1.0 + prm(2) * x(1)))

DO i = 2, dim - 1
    F(i-1) = F(i-1) +     x(i) + h * h * prm(1) / &
            12.0 * exp (x(i) / (1.0 + prm(2) * x(i)))
    F(i)   = F(i)   - 2 * x(i) + h * h * prm(1) / &
             1.2 * exp (x(i) / (1.0 + prm(2) * x(i)))
    F(i+1) =                x(i) + h * h * prm(1) / &
```

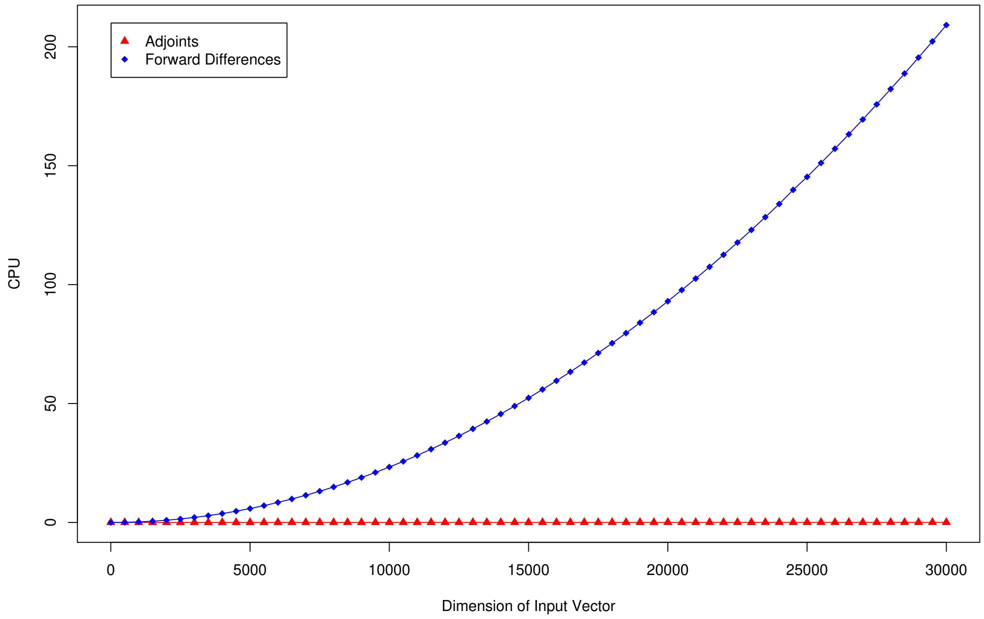

Fig. 2. Adjoints vs. Finite Differences on Griewank's Test Problem

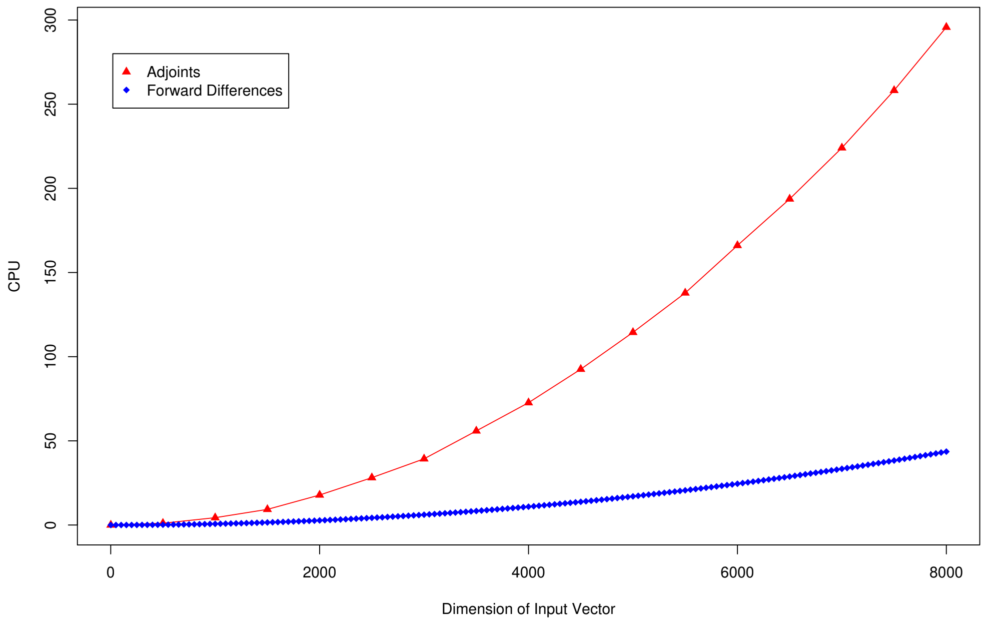

Fig. 3. Adjoints vs. Finite Differences on Bratu Problem

```
            12.0 * exp (x(i) / (1.0 + prm(2) * x(i)))
END DO

F(dim-1) = F(dim-1) + x(dim) + h * h * prm(1) / &
           12.0 * exp (x(dim) / (1.0 + prm(2) * x(dim)))
F(dim)   = F(dim) - 2 * x(dim)
F(dim)   = F(dim) + h * h * prm(1) / 12.0 * (1 + 10 * &
           exp (x(dim) / (1.0 + prm(2) * x(dim))))
...
```

Our objective is to compute the Jacobian matrix of `F` with respect to both `x` and `prm`. In our example we set `dim`=7. Using the intrinsic adjoint code that is generated automatically by the differentiation-enabled NAGWare Fortran 95 compiler we get the following 7×9 Jacobian:

```
-1.89  1.01  0.00  0.00  0.00  0.00  0.00  0.21 -0.48
 1.01 -1.87  1.01  0.00  0.00  0.00  0.00  0.40 -1.78
 0.00  1.01 -1.87  1.01  0.00  0.00  0.00  0.49 -2.70
 0.00  0.00  1.01 -1.87  1.01  0.00  0.00  0.56 -3.50
 0.00  0.00  0.00  1.01 -1.87  1.01  0.00  0.49 -2.70
 0.00  0.00  0.00  0.00  1.01 -1.87  1.01  0.40 -1.78
 0.00  0.00  0.00  0.00  0.00  1.01 -1.89  0.21 -0.48
```

It is obtained by initializing the adjoints of the dependent variables `y` as a 9×9 identity matrix. Notice, that F' is sparse. A column-compressed version can be computed by applying Curtis-Powell-Reid seeding [2] to the tangent-linear code (or, equivalently, the finite difference approximation) as, for example, described in [3]. With the eighth and ninth column of the Jacobian being dense row compression is not possible in adjoint mode.

With $m = n - 2$ the Bratu problem is not our typical application for adjoint mode. It must be expected that the runtime performance of the adjoint code lacks behind that of finite differences due to the overhead resulting from storing and restoring all intermediate values. The correctness of this conjecture is illustrated in Figure 3.

5 Conclusions and Outlook

We have presented a research prototype of the differentiation-enabled NAGWare Fortran 95 compiler that produces intraprocedural adjoint code. Numerical tests show a considerable speedup of medium to large scale gradient computations due to the independence of the number of adjoint function evaluations on the number of inputs of the underlying numerical model.

Ongoing research and development aims toward the intrinsic generation of interprocedural adjoint code inside the compiler. Our objective is to be able to handle numerical simulation programs written in Fortran 95 that result from a large variety of real-world applications in science and engineering.

Acknowledgments

Michael Maier was supported via a subcontract by the University of Hertfordshire, Hatfield, UK.
Malcolm Cohen from NAG Ltd., Oxford, UK has provided his expertise on technical issues while we made ourselves familiar with the compiler's internals.

References

1. B. Averik, R. Carter, and J. Moré, *The MINPACK-2 Test Problem Collection (Preliminary Version)*, Argonne Technical Report ANL/MCS-TR-150, Argonne National Laboratory, 1991.
2. A. Curtis, M. Powell, and J. Reid, *On the Estimation of Sparse Jacobian Matrices*, J. Inst. Math. Appl. 13 (1974), 117-119.
3. J. Riehme and U. Naumann, *Using the Differentiation-Enabled NAGWare Fortran 95 Compiler – A Guided Tour*, Proceedings of ECCOMAS 2004, Jyväskylä, Finland.
4. R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, *Numerical Toolbox for Verified Computing I*, Springer Series in Computational Mathematics, Springer, 1993.
5. R. Horst and P.M. Pardalos (eds.), *Handbook of Global Optimization*, Kluwer, 1995.
6. J. Nocedal and S. Wright, *Numerical Optimization*, Springer, 1999.
7. A. Griewank, *Generalized descent for global optimization*, Journal of Optimization Theory and Applications 34, 11–39, 1981.
8. A. Griewank, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, Frontiers in Applied Mathematics, SIAM, 2000.
9. M. Cohen, U. Naumann and J. Riehme, *Toward Differentiation-Enabled Fortran 95 Compiler Technology*, Proceedings of the 2003 ACM Symposium on Applied Computing, 143–147, 2003,
10. U. Naumann and J. Riehme, *A Differentiation-Enabled Fortran 95 Compiler*, ACM Transactions on Mathematical Software, 31(4), 458–474, 2005.
11. L. Hascoët, Uwe Naumann, and V. Pascual, *TBR Analysis in Reverse Mode Automatic Differentiation*, Future Generation Computer Systems 21, 1401–1417, Elsevier, 2005.
12. V. Pascual and L. Hascoët: *Extension of TAPENADE towards Fortran 95*, in [13], Springer, 2006.
13. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.): *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, vol. 50, Springer, 2006.
14. U. Naumann and J. Riehme, *Computing Adjoints with the NAGWare Fortran 95 Compiler*, in [13], Springer, 2006.
15. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.
16. J. Utke, A. Lyons, and U. Naumann, *Efficient Reversal of the Intraprocedural Flow of Control in Adjoint Computations*, to appear in Journal of Systems and Software, Elsevier 2006.
17. URL: `http://www.mcs.anl.gov/openad`
18. J. Utke and U. Naumann, *Software Technological Issues in Automatizing the Semantic Transformation of Numerical Programs*, Software Engineering and Applications, M. Hamza, ed., ACTA Press, 417–422, 2003.