UDC 004.421

On Syntax-Directed Adjoint Fortran Code

Uwe Naumann[1 2] and Jan Riehme[2]

[1] LuFG Informatik 12, Department of Computer Science, RWTH Aachen University
52056 Aachen, Germany
[2] Department of Computer Science, University of Hertfordshire
Hatfield, AL10 9AB, United Kingdom
{naumann, riehme}@stce.rwth-aachen.de

Abstract. Gradients of high-dimensional functions can be computed efficiently and with machine accuracy by so-called adjoint codes. We present an L-attributed grammar for the single-pass generation of intraprocedural adjoint code for a subset of Fortran. Our aim is to integrate the syntax-directed approach into the front-end of the NAGWare Fortran compiler. Research prototypes of this compiler that build adjoint code based on an abstract intermediate representation have been developed for several years. We consider the syntax-directed generation of adjoint code as a low development cost alternative to more sophisticated algorithms. The price to pay comes in form of a very limited set of code optimizations that can be performed in a single-pass setting.

1. Motivation

Numerical simulation plays a central role in computational science and engineering. Derivatives (gradients, Jacobians, Hessians or even higher derivatives) are required in order to make the highly desirable transition from pure simulation to optimization of the numerical model or its parameters. Refer to [2],[3], [4],[5] for an impressive collection of such applications.

Consider an implementation of a multivariate nonlinear function $f : \mathbb{R}^n \to \mathbb{R}$ as a computer program where $y = f(\mathbf{x})$. Suppose that we are interested in the sensitivities of the objective y with respect to changes in the parameter vector $\mathbf{x}$, for example, in the context of an unconstrained optimization algorithm. Such derivatives (the gradient of y with respect to $\mathbf{x}$) can be approximated by centered (or forward, or backward) finite difference quotients

$$\frac{\partial y}{\partial x_i} \approx \frac{f(x_0, \ldots, x_i + h, \ldots, x_{n-1}) - f(x_0, \ldots, x_i - h, \ldots, x_{n-1}, u)}{2h} \tag{1}$$

for an appropriate (small) value $h \in \mathbb{R}$. Choosing the right value for h for a given function evaluated in a given floating-point number system can be problematic. Cancellations can lead to very poor approximations of the derivatives. More importantly, the accumulation

of the whole gradient requires $2n$ function evaluations which may be infeasible for high-dimensional problems. See [15] for an application in oceanography where n can be of the order of 10^{12} and higher.

Even for very simple representatives of equation (1) (for example, $y = x_0 * \ldots * x_{n-1}$) the finite difference approximation of the gradient can take several hours for $n \geq 10^6$. In this paper we present an L-attributed grammar [13] for transforming the implementation of f into an adjoint code during a single pass compilation process. The adjoint code computes the same gradient in only a few seconds.

In section 2 we outline the basic structure of adjoint codes. An L-attributed grammar for transforming programs written in imperative programming languages that are suitable for single-pass compilation ([1],[17]) is presented in section 3. A simple proof-of-concept implementation based on `flex` and `bison` as well as a case study are discussed in section 4. Links with the ongoing development of the NAGWare Fortran compiler are established in section 5. We conclude with an outlook to potential areas of application of the proposed technology in section 6.

2. Adjoint Code

The problem of determining an appropriate value for h can be eliminated by considering a tangent-linear model $\dot{F}$ of F. Let therefore $\mathbf{x} = \mathbf{x}(t)$ with $t \in I\!R$ and set

$$\frac{\partial \mathbf{x}}{\partial t} = \dot{\mathbf{x}} \quad .$$

By the chain rule we get

$$\frac{\partial \mathbf{y}}{\partial t} = \dot{\mathbf{y}} = \dot{F}(\mathbf{x}, \dot{\mathbf{x}}) = F' \cdot \dot{\mathbf{x}} \quad . \tag{2}$$

Refer to Fig. 1 (a) and (b) for a graphical illustration. It can be regarded as a transformation of the parser tree of $F(\mathbf{x}(t))$ (see (a)) into one for $\dot{F}(\mathbf{x}, \dot{\mathbf{x}})$. The technique is known as the forward mode of automatic differentiation (AD) [9]. The parse tree is linearized by attaching partial derivatives to the corresponding edges. The chain rule of differentiation is interpreted as the chained product of all edge labels along the path from t to $\mathbf{y}$. Assuming that we have an implementation of $\dot{F}$ we can compute the columns of F' by letting $\dot{\mathbf{x}}$ range over the Cartesian basis vectors in $I\!R^n$. The computational complexity of this approach is of the same order as that of finite differences.

To eliminate the dependence of the computational complexity on the potentially very large value of n we consider adjoint codes that can be generated by the reverse mode of AD. Let therefore $t = t(\mathbf{y})$ with $t \in I\!R$ and set

$$\frac{\partial t}{\partial \mathbf{y}} = \bar{\mathbf{y}} \quad .$$

Exploiting the associativity of the chain rule we get

$$\bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) = \bar{\mathbf{y}} \cdot F' \quad . \tag{3}$$

Refer to Fig. 1 (c) and (d) for illustration. All barred vectors ($\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$) are row vectors. Assuming that we have an implementation of $\bar{F}$ we can compute the rows of F' by letting $\bar{\mathbf{y}}$ range over the Cartesian basis vectors in $\mathbb{R}^m$. Gradients of scalar functions in particular can be obtained at a (hopefully) small constant multiple of the computational complexity of F. The realization of this theoretical result in practice is the subject of numerous ongoing research and development efforts world-wide. See `http://www.autodiff.org` for links and further information.

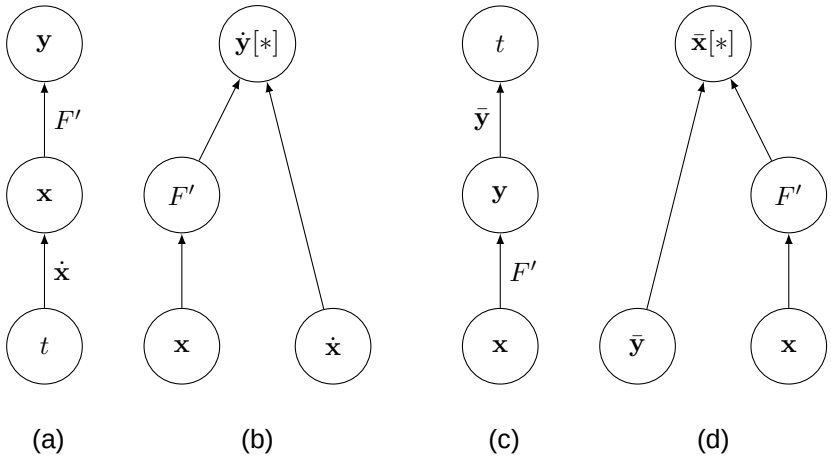

Fig. 1. Linearized $F(\mathbf{x}(t))$ (a), $\dot{F}$ (b) linearized $t(F(\mathbf{x}))$ (c), $\bar{F}$ (d)

Any execution of the program F is expected to decompose into a sequence of elemental assignments

$$v_j = \varphi_j(v_i)_{i \prec j} \tag{4}$$

for $j = 1, \ldots, p+m$ and $i \prec j$ if and only if v_i is an argument of φ_j. equation (4) is also referred to as the *code list* of F at the given point that fixes the flow of control. We set $v_{i-n} = x_i$ for $i = 1, \ldots, n$ and $v_{p+j} = y_j$ for $j = 1, \ldots, m$. The v_k, $k = 1-n, \ldots, p+m$, are called *code list variables*.

The *elemental functions* φ_j are assumed to be continuously differentiable in a neighborhood of the current argument. The corresponding local partial derivatives are denoted by

$$c_{j,i} = \frac{\partial \varphi_j}{\partial v_i} \quad .$$

Adjoints are propagated backwards with respect to the data flow in the code list. Hence, the values of the intermediate variables are not used in their original order of computation. In (incremental) reverse mode AD the local partial derivatives are computed during

the adjoint evaluation.

$$v_{i-n} = x_i \qquad \text{for } i = 1, \ldots, n \tag{5}$$
$$v_j = \varphi_j(v_i)_{i \prec j} \qquad \text{for } j = 1, \ldots, p+m \tag{6}$$
$$y_k = v_{p+k} \qquad \text{for } k = 1, \ldots, m \tag{7}$$
$$\bar{v}_{p+k} = \bar{y}_k \qquad \text{for } k = 1, \ldots, m \tag{8}$$
$$\bar{v}_j = 0 \qquad \text{for } j = 1-n, \ldots, p \tag{9}$$
$$c_{j,i} = \frac{\partial \varphi_j}{\partial v_i}; \quad \bar{v}_i = \bar{v}_i + c_{j,i} \cdot \bar{v}_j \qquad \text{for } i \prec j \text{ and } j = q, \ldots, 1 \tag{10}$$
$$\bar{x}_i = \bar{v}_{i-n} \qquad \text{for } i = 1, \ldots, n \quad . \tag{11}$$

Adjoint assignments are generated for all assignments in the original code. We build assignment-level code lists as in equation (5)–equation (7). The data-flow reversal requires arguments of nonlinear operations to be persistent. Conservatively we account for this by storing all overwritten values on a value stack (`push_v`). The resulting code is referred to as *augmented forward code*. Adjoints are propagated backwards according to equation (8)–equation (11). The previously stored values are restored from the stack (`pop_v`). The resulting code is referred to as *backward code*. The compiler-generated intermediate variables v_j represent subexpressions of the right-hand side. Hence, they are read exactly once, thus eliminating the need for the initialization in equation (9) as well as that for the incrementation in equation (10). Adjoints of compiler-generated intermediate variables are simply overwritten. Only adjoints of program variables need to be initialized and incremented.

Example In our proof-of-concept implementation (see section 4) the single assignment

```
x=x*y
```

is transformed into the augmented forward code

```
call push_v(v0); v0=x
call push_v(v1); v1=y
call push_v(v2); v2=v0*v1
call push_v(x); x=v2
```

followed by the backward code

```
call pop_v(x); v2_=x_; x_=0
call pop_v(v2); v0_=v2_*v1; v1_=v2_*v0
call pop_v(v1); y_=y_+v1_
call pop_v(v0); x_=x_+v0_
```

For notational simplicity code list variables are enumerated starting from 0 rather than $1-n=-1$. Adjoint variables are marked by a trailing underscore. The gradient of x as an output with respect to x as an input and y at a given point (x,y) is equal to (y,x). It can be computed numerically by a single run of the adjoint code. Note that, while being conservatively correct, this adjoint code is far from optimal. Optimization of adjoint code

is a major issue in ongoing research and development in the field of AD. It is beyond the scope of this paper that aims to relate the fundamental concept of attribute grammars to correct automatically generated adjoint code.

The reversal of the data-flow implies the necessity to reverse the flow of control. A simple, conservatively correct approach is to enumerate all assignments and to remember their order of execution in the augmented forward code by pushing their respective indexes onto a control stack (`push_c`). The backward code loops over the indexes in reverse (`pop_c`) thus executing the backward codes of all assignments in the correct order. Refer to section 4 for an example.

3. L-Attributed Grammar for Adjoint Code

A syntax-directed approach to the automatic generation of tangent-linear code has been presented in [8]. In this section we extend these ideas to adjoint codes by the definition of an appropriate L-attributed grammar. Recall that a grammar is called *L-attributed* if the values of all inherited attributes are functions of non-terminals to the left in the given production rule (includes the parent on the left-hand side). A formal proof of the correctness of our approach is straight forward. All one has to show is that the augmented production rules (P0)–(P10) imply equations (5)–(11).

The input code is a sequence of one or more statements ($\mathbf{code} :: s$). In addition to assignments (a) we introduce simple branch (b) and loop (l) constructs causing a potentially nontrivial intraprocedural flow of control. Five attributes are associated with each grammar symbol: The integer attribute j represents the assignment-level code list variable indexes. A second integer attribute n is used to synthesize the sizes of subtrees (parse trees) in right-hand sides of assignments. A third integer attribute k serves as an enumerator of the assignments in the input code. There are two text attributes to hold the forward (c^f) and backward (c^b) sections of the adjoint code. The text vector c^b has length α, where α denotes the number of assignment statements in the input code. The whole adjoint code is synthesized into $\mathbf{code}.c^f$ during a successful compilation. The complete augmented forward code $s.c^f$ is followed by the reverse loop over the adjoints of all executed assignments. The Fortran syntax is sensitive with respect to `newline` characters. Below we assume that the $+$ at the beginning of a line implies a `newline` in the generated code. Moreover we assume that the input code is syntactically correct. For example, syntactic (and semantic) correctness can be verified by running the commercial version of the NAGWare Fortran compiler prior to using our research prototype.

$(P0)$ $\quad \mathbf{code} ::$

$$s \qquad \begin{array}{l} s.k = 0 \\ \\ \mathbf{code}.c^f = \text{"}\texttt{integer i}\text{"} \\ \qquad\qquad + s.c^f \end{array}$$

$$
\begin{aligned}
&+ \text{"}\texttt{do while(pop\_c(i))}\text{"}\\
&+ \quad \text{"}\texttt{if (i == 1) then}\text{"}\\
&+ \qquad s.c_1^b\\
&+ \quad \text{"}\texttt{else if (i == 2) then}\text{"}\\
&+ \qquad s.c_2^b\\
&\quad \vdots\\
&+ \quad \text{"}\texttt{else if (i ==}\text{"} + s.k\\
&+ \text{"}\texttt{) then}\text{"}\\
&+ \qquad s.c_{s.k}^b\\
&+ \quad \text{"}\texttt{end if}\text{"}
\end{aligned}
$$

$$
\begin{array}{lll}
(P1) & s :: & \\
 & & a.k = s.k + 1\\
 & a & \\
 & & s.k = a.k; \quad s.c^f = a.c^f; \quad s.c^b = a.c^b
\end{array}
$$

The vector assignment $s.c^b = a.c^b$ operates at the elemental level, that is, $s.c_i^b = a.c_i^b$ for $i = 1, \ldots, \alpha$. We chose a split way of presenting the production rules together with their associated semantic actions that resembles the implementation in section 4. For example, the attribute k of a is set prior to parsing the assignment itself. The forward and backward codes of the nonterminal symbol on the left-hand side of the production rule are synthesized at the time of reduction (in the context of a shift-reduce parser).

$$
\begin{array}{lll}
(P1a) & s :: & \\
 & & b.k = s.k\\
 & b & \\
 & & s.k = b.k; \quad s.c^f = b.c^f; \quad s.c^b = b.c^b
\end{array}
$$

$$
\begin{array}{lll}
(P1b) & s :: & \\
 & & l.k = s.k\\
 & l & \\
 & & s.k = l.k; \quad s.c^f = l.c^f; \quad s.c^b = l.c^b
\end{array}
$$

$(P2)$ $\quad s^l ::$

$$a.k = s^l.k + 1$$

$$a$$

$$s^r.k = a.k$$

$$s^r$$

$$s^l.k = s^r.k; \quad s^l.c^f = a.c^f + s^r.c^f$$
$$s^l.c^b = s^r.c^b + a.c^b$$

The vector sum $s^l.c^b = s^r.c^b + a.c^b$ is also elemental, that is, $s^l.c_i^b = s^r.c_i^b + a.c_i^b$ for $i = 1, \ldots, \alpha$.

$(P2a)$ $\quad s^l ::$

$$b.k = s^l.k$$

$$b$$

$$s^r.k = b.k$$

$$s^r$$

$$s^l.k = s^r.k; \quad s^l.c^f = b.c^f + s^r.c^f$$
$$s^l.c^b = s^r.c^b + b.c^b$$

$(P2b)$ $\quad s^l ::$

$$l.k = s^l.k$$

$$l$$

$$s^r.k = l.k$$

$$s^r$$

$$s^l.k = s^r.k; \quad s^l.c^f = l.c^f + s^r.c^f$$
$$s^l.c^b = s^r.c^b + l.c^b$$

$(P3)$ $\quad a ::$

$$e.k = a.k; \quad e.j = 0$$

$$V = e$$

$$a.c^f = e.c^f + \text{"}\texttt{call push\_c}(\text{"} + a.k + \text{"})\text{"}$$
$$+ \text{"}\texttt{call push\_v}(\text{"} + V.c^f + \text{"})\text{"}$$
$$+ V.c^f + \text{"} = \texttt{v0}\text{"}$$

$$
\begin{aligned}
a.c^b_{a.k} = &\text{"}\texttt{call pop\_v(}\text{"} + V.c^f + \text{"}\texttt{)}\text{"} \\
&+ \text{"}\texttt{v0\_ =}\text{"} + V.c^f + \text{"}\texttt{\_}\text{"} \\
&+ V.c^f + \text{"}\texttt{\_ = 0}\text{"} \\
&+ e.c^b_{a.k}
\end{aligned}
$$

The root of the syntax tree of the expression e of the right-hand side has fixed code list variable index 0. Variable references are stored in $V.c^f$ by the scanner or some preparser depending on the regularity of the syntax for variable references.

$(P4)$ $\quad e :: V$

$$
\begin{aligned}
e.n &= 1 \\
a.c^f &= \text{"}\texttt{call push\_v(v}\text{"} + e.j + \text{"}\texttt{)}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e.j + \text{"}\texttt{=}\text{"} + V.c^f \\
a.c^b_{e.k} &= \text{"}\texttt{call pop\_v(v}\text{"} + e.j + \text{"}\texttt{)}\text{"} \\
&\quad + V.c^f + \text{"}\texttt{\_ =}\text{"} + V.c^f \\
&\quad + \text{"}\texttt{\_ + v}\text{"} + e.j + \text{"}\texttt{\_}\text{"}
\end{aligned}
$$

$(P5)$ $\quad e :: C$

$$
\begin{aligned}
e.n &= 1 \\
a.c^f &= \text{"}\texttt{call push\_v(v}\text{"} + e.j + \text{"}\texttt{)}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e.j + \text{"}\texttt{=}\text{"} + C.c^f \\
a.c^b_{e.k} &= \text{"}\texttt{call pop\_v(v}\text{"} + e.j + \text{"}\texttt{)}\text{"}
\end{aligned}
$$

$(P6)$ $\quad e^l ::$

$$
\begin{aligned}
& e^r.j = e^l.j + 1; \quad e^r.k = e^l.k \\
& \qquad F(e^r) \\
& e^l.n = e^r.n + 1 \\
& e^l.c^f = e^r.c^f \\
&\quad + \text{"}\texttt{call push\_v(v}\text{"} + e^l.j + \text{"}\texttt{)}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e^l.j + \text{"}\texttt{=}\text{"} + F.c^f \\
&\quad + \text{"}\texttt{(v}\text{"} + e^r.j + \text{"}\texttt{)}\text{"} \\
& e^l.c^b_{e.k} = \text{"}\texttt{call pop\_v(v}\text{"} + e^l.j + \text{"}\texttt{)}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e^r.j + \text{"}\texttt{\_ =}\text{"} + F_{e^r.j} \\
&\quad + \text{"}\texttt{*v}\text{"} + e^l.j + \text{"}\texttt{\_}\text{"} \\
&\quad + e^r.c^b_{e.k}
\end{aligned}
$$

F is an arbitrary unary function, such as $\sin$ or $\exp$. $F_{e^r.j}$ denotes the partial derivative of F with respect to the code list variable holding the value of the expression e^r.

$(P7)$ $e^l ::$

$$e^{r_1}.j = e^l.j + 1; \quad e^{r_i}.k = e^l.k \text{ for } i = 1, 2$$

e^{r_1}

$$e^{r_2}.j = e^{r_1}.j + e^{r_1}.n + 1$$

Oe^{r_2}

$$\begin{aligned}
e^l.n &= e^{r_1}.n + e^{r_2}.n + 1 \\
e^l.c^f &= e^{r_1}.c^f + e^{r_2}.c^f \\
&\quad + \text{"}\texttt{call push_v(v}\text{"} + e^l.j + \text{"}\texttt{);}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e^l.j + \text{"}\texttt{= v}\text{"} + e^{r_1}.j + O.c^f \\
&\quad + \text{"}\texttt{v}\text{"} + e^{r_2}.j \\
e^l.c^b_{e.k} &= \text{"}\texttt{call pop_v(v}\text{"} + e^l.j + \text{"}\texttt{)}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e^{r_2}.j + \text{"}\texttt{_ =}\text{"} + O_{e^{r_2}.j} \\
&\quad + \text{"}\texttt{*v}\text{"} + e^l.j + \text{"}\texttt{_}\text{"} \\
&\quad + \text{"}\texttt{v}\text{"} + e^{r_1}.j + \text{"}\texttt{_ =}\text{"} + O_{e^{r_1}.j} \\
&\quad + \text{"}\texttt{*v}\text{"} + e^l.j + \text{"}\texttt{_}\text{"} \\
&\quad + e^{r_2}.c^b_{e.k} + e^{r_1}.c^b_{e.k}
\end{aligned}$$

O is an arbitrary binary operator, such as $+$ or $*$. $O_{e^{r_1}.j}$ denotes the partial derivative of O with respect to the code list variable holding the value of the expression e^{r_1}. (similarly e^{r_2})

$(P8)$ $b :: IF\ (r)$

$$s.k = b.k$$

$THEN$

s

$ENDIF$

$$\begin{aligned}
b.k &= s.k \\
b.c^f &= \text{"}\texttt{if}\text{"} + \text{"}\texttt{(}\text{"} + r.c^f + \text{"}\texttt{)}\text{"} + \text{"}\texttt{then}\text{"} \\
&\quad + s.c^f \\
&\quad + \text{"}\texttt{end if}\text{"} \\
b.c^b &= s.c^b
\end{aligned}$$

$(P9)$ $\quad l :: DO\ WHILE\ (r)$

$$s.k = l.k$$

s

$ENDDO$

$$\begin{aligned} l.k &= s.k \\ l.c^f &= \text{"}\texttt{do while}\text{"} + \text{"(" } + r.c^f + \text{")"} \\ &\quad + s.c^f \\ &\quad + \text{"}\texttt{end do}\text{"} \\ l.c^b &= s.c^b \end{aligned}$$

$(P10)$ $\quad r :: V^{r_1} < V^{r_2}$

$$r.c^f = V^{r_1}.c^f + \text{"<"} + V^{r_2}.c^f$$

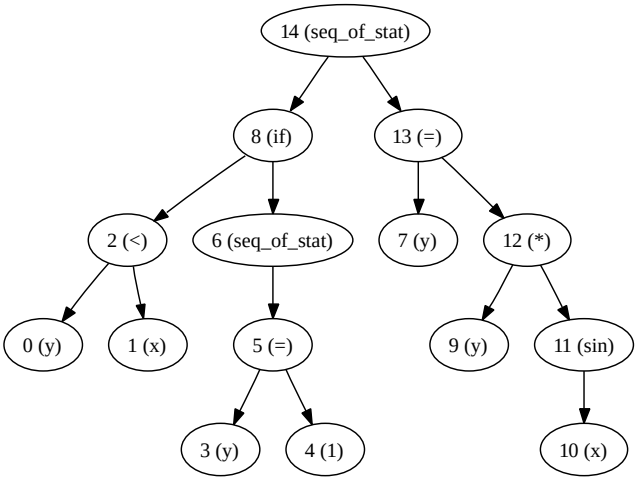

Fig. 2. Parse Tree (generated with `dot`; see `www.graphviz.org`)

Example We investigate the development of the values of all five attributes when parsing

```
if (y<x) then
  y=1
end if
y=y*sin(x)
```

The parse tree is depicted in Fig. 2. Sequences of statements have been flattened into a single vertex with the corresponding statements as immediate successors (v_6 and v_{14}). The i-th vertex is referenced as v_i.

1. Synthesized subtree sizes in right-hand sides of assignments: $v_4.n = 1$, $v_9.n = 1$, $v_{10}.n = 1$, $v_{11}.n = v_{10}.n + 1 = 2$, $v_{12}.n = v_9.n + v_{11}.n + 1 = 4$.
2. Inherited code list variable indexes: $v_4.j = 0$ (P3), $v_{12}.j = 0$ (P3), $v_9.j = v_{12}.j + 1 = 1$ (P7), $v_{11}.j = v_{12}.j + v_9.n + 1 = 2$ (P7), $v_{10}.j = v_{11}.j + 1 = 3$ (P6).
3. Inherited assignment counter: $v_{14}.k = 0$ (P0) $v_8.k = v_{14}.k = 0$ (P2a), $v_6.k = v_8.k = 0$ (P8), $v_5.k = v_6.k + 1 = 1$ (P1), $v_4.k = v_5.k = 1$ (P3), $v_8.k = v_6.k = v_5.k = 1$ (P8, P2a), $v_{13}.k = v_8.k + 1 = 2$ (P2, P3), $v_{12}.k = v_{13}.k = 2$ (P3), $v_9.k = v_{10}.k = v_{11}.k = v_{12}.k = 2$ (P4–P7), $v_{14}.k = v_{13}.k = 2$ (P1, P2a).
4. Synthesized augmented forward code:
   - $v_0.c^f =$ "y" (Scanner)
   - $v_1.c^f =$ "x" (Scanner)
   - $v_2.c^f =$ "y < x" (P10)
   - $v_3.c^f =$ "y" (Scanner)
   - $v_4.c^f =$ "call push_v(v0); v0 = 1" (P5, Scanner)
   - $v_5.c^f = v_4.c^f +$ "call push_c(1); call push_v(y); y = v0" (P3, Scanner)
   - $v_6.c^f = v_5.c^f$ (P1)
   - $v_8.c^f =$ "if(y < x)then" $+ v_5.c^f +$ "endif" (P8)
   - $v_7.c^f =$ "y" (Scanner)
   - $v_9.c^f =$ "call push_v(v1); v1 = y" (P4, Scanner)
   - $v_{10}.c^f =$ "call push_v(v3); v3 = x" (P4, Scanner)
   - $v_{11}.c^f = v_{10}.c^f +$ "call push_v(v2); v2 = sin(v3)" (P6)
   - $v_{12}.c^f = v_9.c^f + v_{11}.c^f +$ "call push_v(v0); v0 = v1 * v2" (P7)
   - $v_{13}.c^f = v_{12}.c^f +$ "call push_c(2); call push_v(y); y = v0" (P3, Scanner)
   - $v_{14}.c^f = v_8.c^f + v_{13}.c^f$ (P2a, P1)
5. Inherited backward code:
   - $v_4.c_1^b =$ "call pop_v(v0)" (P5)
   - $v_5.c_1^b =$ "call pop_v(y); v0_ = y_; y_ = 0" $+ v_4.c_1^b$ (P3)
   - $v_8.c_1^b = v_6.c_1^b = v_5.c_1^b$ (P1, P8)
   - $v_9.c_2^b =$ "call pop_v(v1); y_ = y_ + v1_" (P4)
   - $v_{10}.c_2^b =$ "call pop_v(v3); x_ = x_ + v3_" (P4)
   - $v_{11}.c_2^b =$ "call pop_v(v2); v3_ = cos(v3) * v2_" $+ v_{10}.c_2^b$ (P6)
   - $v_{12}.c_2^b =$ "call pop_v(v0); v2_ = v1 * v0_; v1_ = v2 * v0_" $+ v_9.c_2^b + v_{11}.c_2^b$ (P7)
   - $v_{13}.c_2^b =$ "call pop_v(y); v0_ = y_; y_ = 0" $+ v_{12}.c_2^b$ (P3)
   - $v_{14}.c_1^b = v_8.c_1^b (P2a); v_{14}.c_2^b = v_{13}.c_2^b$ (P2a, P1)

Finally, the augmented forward code and the backward codes of both assignments are synthesized according to production rule P0 to obtain the whole adjoint code:

```
integer i;
v14.c^f
do while (pop_c(i))
if (i==1) then
v14.c_1^b
else if (i==2) then
v14.c_2^b
end if
end do
```

4. Implementation and Case Study

We have developed a proof-of-concept implementation based on the scanner and parser generators `flex` and `bison`. The inherited attributes are evaluated very conveniently as a combination of synthesized and global variables. Alternatively we could have used an attribute grammar processing tool such as LISA [16] or JastAdd [12]. This research is part of the CompAD project – a larger R&D effort aiming at the integration of adjoint code generation capabilities into the NAGWare Fortran compiler [20]. The NAGWare compiler uses `yacc` / `bison` as a parser generator. Hence, our focus is on `bison` in order to make our results directly exploitable in the context of CompAD. See section 5 for further information on the CompAD project.

The `bison` input has the following structure.

```
1  ...
2
3  %token V F IF THEN DO WHILE END
4  %left '*'
5
6  %%
7
8  code : s
9  s : a
10   | b
11   | l
12 s : a s { ... };
13   | b s { ... };
14   | l s { ... };
15 b : IF '(' c ')' THEN s END IF { ... };
16 l : DO WHILE '(' c ')' s END DO { ... };
17 c : V '<' V { ... };
18 a : V '=' { ... } e ';' { ... };
19 e : e '*' e {
```

```
20      $$.j=clc++;
21      get_memory_f(&$$); get_memory_r(&$$,k);
22      sprintf($$.cf,"%s%scall push_v(v%d); v%d=v%d*v%d\n",$1.cf,$3
            .cf,$$.j,$$.j,$1.j,$3.j);
23      sprintf($$.cb[k],"call pop_v(v%d); v%d_=v%d_*v%d; v%d_=v%d_*
            v%d\n%s%s",$$.j,$1.j,$$.j,$3.j,$3.j,$$.j,$1.j,$3.cb[k],
            $1.cb[k]);
24      free_memory_f(&$1); free_memory_r(&$1,k); free_memory_f(&$3)
            ; free_memory_r(&$3,k);
25    };
26    | F '(' e ')' { ... };
27    | V { ... };
28    | C { ... };
29
30 %%
```

IF, THEN, DO, WHILE, END, V, F, as well as the remaining single-character tokens are delivered by the lexical analyzer. Assuming syntactic correctness of the input code we can ignore the sensitivity with respect to `newline` characters. As an example we include the treatment of the product of two expressions. The inherited attribute j in grammar rule P7 is implemented as a global counter variable (line 20). While the order of the enumeration is changed, the necessary properties (uniqueness, correct dependences among the code list variables) are preserved. New dynamic memory is allocated for the augmented forward and adjoint codes corresponding to the nonterminal symbol e on the left-hand side of the production rule (line 21). The augmented forward code ($$.cf) is generated on line 22. The adjoint code ($$.cb[k]) is generated on line 23. Finally, the dynamic memory associated with the two nonterminal symbols on the right-hand side of the production rule is deallocated (line 24). The entire code is open source. It can be obtained by sending an email to the first author.

As a final case study consider the following simple Fortran code fragment.

```
if (x<y) then
  x=x*y
  do while (y<x)
   x=sin(x*y)
  end do
}
```

With x=-5.0 and y=-0.5 as inputs the while-loop is traversed once to compute x=-0.949. The corresponding gradient (0.079,1.577) is computed by a single run of the following automatically generated adjoint code.

```
1 integer i;
2 if (x<y) then
3   call push_c(1)
4   call push_v(v0); v0=x
5   call push_v(v1); v1=y
```

```
6    call push_v(v2); v2=v0*v1
7    call push_v(x); x=v2
8    do while (y<x)
9      call push_c(2)
10     call push_v(v0); v0=x
11     call push_v(v1); v1=y
12     call push_v(v2); v2=v0*v1
13     call push_v(v3); v3=sin(v2)
14     call push_v(x); x=v3
15   end do
16 end if
17 do while (pop_c(i))
18   if (i==1) then
19     call pop_v(x); v2_=x_; x_=0
20     call pop_v(v2); v0_=v2_*v1; v1_=v2_*v0
21     call pop_v(v1); y_=y_+v1_
22     call pop_v(v0); x_=x_+v0_
23   else if (i==2) then
24     call pop_v(x); v3_=x_; x_=0
25     call pop_v(v3); v2_=cos(v2)*v3_
26     call pop_v(v2); v0_=v2_*v1; v1_=v2_*v0
27     call pop_v(v1); y_=y_+v1_
28     call pop_v(v0); x_=x_+v0_
29   end if
30 end do
```

The two assignment statements are enumerated by pushing unique indexes onto the control stack (lines 3 and 9; grammar rule P3). Assignment-level code lists are built and the values of all overwritten variables are saved on the value stack (lines 4 – 7 and 10–14; grammar rules P3–P7). The flow of control remains unchanged in the augmented forward code.

The adjoint code executes the adjoint code lists in reverse order by restoring the indexes of the corresponding original assignments (lines 1, 17, 18, and 23; grammar rule P0). All adjoint code list statements are preceded by pop accesses to the value stack to restore the old value of the variables of the left-hand side of the original code list statement (lines 19 – 22 and 24–28; grammar rules P3–P7). For example, the adjoint corresponding to the assignment on line 13 first restores the value of v3 followed by the evaluation of the product of cos(v2) (derivative of sin(v2)) with the adjoint of v3 to get the adjoint of v2 (line 25).

5. The CompAD Project

The aim of the **Comp**iler for **A**utomatic **D**ifferentiation project is to integrate Automatic Differentiation (AD) [9] capabilities into an industrial-strength compiler – the NAGWare Fortran compiler. The assessment of the individual grant review of CompAD-I (EPSRC

grant GR/R55252/01) was “tending to outstanding.” We are currently at the end of the first year of CompAD-II – a two year project funded by EPSRC under grant number EP/D062071/1.

The detailed description of all issues considered during CompAD so far would be inappropriate for this manuscript. We focus on some major points. Additional material can be found on the project's Internet site.[3] It is impossible to present all the background of the following statements. Good references are [9] and the proceedings of the past four international conferences on automatic differentiation [2],[3],[4], [5].

Adjoint code is generated automatically for programs written in Fortran We have developed a C++-API for the compiler's native C interface to the internal representation (IR). The API is used to perform modifications of the IR that result in adjoint C code after unparsing. See [14] for details. A number of successful tests are documented on our Internet site. The interprocedural flow of control is currently reversed in split mode [9]. Building on [18] we can show that the combinatorial problem of optimal call graph reversal is NP-complete. Near-optimal trade-offs between various reversal modes are the subject of ongoing research.

Tangent-linear code is generated automatically for programs written in Fortran An elegant approach to the automatic generation of tangent-linear code is to inline implementations of overloaded operators and intrinsic functions. The static resolution of overloading in Fortran 90 makes this solution possible. Our inliner replaces calls to subroutines from the tangent-linear module with the corresponding code to obtain clean tangent-linear code. Refer to our Internet site for examples.

Tangent-linear code is generated automatically for programs written in Assembler Our tool for generating tangent-linear versions of assembler programs is called ADAC (“Automatic Differentiation of Assembler Code”). It parses the input into a custom IR that is then converted and unparsed. The efficient handling of various issues related to addressing memory (including registers and stacks) poses several problems. Moreover the corresponding solutions are often hardware-dependent (Intel 80386 in our case). Nevertheless, our pioneering work in this field is likely to make the potential transition to other architectures much easier. We consider assembler as an IR that is independent of the high-level programming language. Refer to [7] and to our Internet site for various proof-of-concept examples, where Assembler is generated from C/C++ and Fortran programs followed by the application of ADAC.

Second-order adjoint code is generated automatically for programs written in Fortran Given an adjoint code the obvious and very robust approach to the generation of second-order adjoint code [9] is to overload the given first-order adjoint code in tangent-linear mode [19]. The computational overhead due to the subroutine calls resulting from resolved overloading is usually acceptable. Otherwise the previously mentioned inliner can be used to gain efficiency. Corresponding tests are underway. Alternatively,

[3] `http://wiki.stce.rwth-aachen.de/bin/view/Projects/CompAD/WebHome`

the compiler can produce adjoint assembler code that is fed into ADAC. Refer to our Internet site for illustration.

Automatically generated adjoint code is used in a time-dependent optimal control formulation of the compressible Navier-Stokes equations In collaboration with colleagues at the Technical University Dresden, Germany, the compiler has been coupled with the checkpointing tool `revolve` [10] to solve a large-scale optimal control problem (10^5 controls). The adjoints are computed by overloading using the built-in support for the necessary type changes provided by the differentiation-enabled NAGWare Fortran compiler.

Automatically generated second-order adjoint code is used to obtain a matrix-free version of the Truncated Newton (TN) algorithm by Dembo and Steihaug [6] The basic approach has been described in [19]. The availability of exact (up to machine accuracy) derivative information proves to be crucial for the convergence behavior of TN. We are currently working on a set of different stopping criteria. Larger test cases will are targeted as the robustness of the adjoint code generator and the efficiency of the generated code increase.

6. Conclusion

The syntax-directed compilation of adjoint codes for numerical programs written in suitable (subsets of) programming languages represents a low-development-cost alternative to full-size adjoint code compilers such as OpenAD [21] or the differentiation-enabled NAGWare Fortran compiler. Due to the lack of static program analysis and the corresponding optimizations (see, for example, [11]) the output of a single-pass adjoint compiler should not be expected to have the same level of efficiency. However the conceptual insight provided by the formulation of adjoint code generation rules in form of an L-attributed grammar represents a good entry point into the subject. For example, we follow this approach in our course on “Adjoint Compilers” taught at the Department of Computer Science at RWTH Aachen University.

The proposed method can be modified and extended to decrease the computational complexity through a decrease of the memory requirement. The use of compiler-generated variables in the context of assignment-level code lists can be reduced drastically. Arithmetic expressions (required for the local partial derivatives) need to be synthesized instead of indexes of the corresponding code list variables. This work is to be continued on the basis of graduate- and undergraduate-level projects.

7. Acknowledgments

The ongoing support provided by the Numerical Algorithms Group[4] and QinetiQ[5] is instrumental to the CompAD project whose principal investigator is Prof. Bruce Christianson of the University of Hertfordshire, UK.

We would like to thank the anonymous referees for their helpful comments on the manuscript.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
3. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2005. Springer.
4. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
5. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.
6. R. Dembo and T. Steihaug. Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26:190–212, 1983.
7. D. Gendler, U. Naumann, and B. Christianson. Automatic differentiation of assembler code. In N. Guimarães and P. Isaias, editors, *Proceedings of the IADIS International Conference on Applied Computing*, pages 431–436, Salamanca, Spain, 2007. IADIS.
8. D. Gendler, U. Naumann, and E. Varnik. Syntax-Directed Tangent-Linear Code. In *Proceedings of the International IADIS Conference on Applied Computing 2007 (IADIS AC07)*, pages 425–430, February 2007.
9. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
10. A. Griewank and A. Walther. Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, mar 2000. Also appeared as Technical University of Dresden, Technical Repor t IOKOMO-04-1997.
11. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.

[4] `http://www.nag.com`
[5] `http://www.qinetiq.com`

12. G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.
13. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
14. M. Maier and U. Naumann. Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In *Proceedings of 5th International Conference on Engineering Computational Technology (ECT 2006)*, pages 1–19. Civil-Comp Press, 2006.
15. J. Marotzke, R. Giering, Q. K. Zhang, D. Stammer, C. N. Hill, and T. Lee. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity. *J. Geophys. Res.*, 104:29,529 – 29,548, 1999.
16. M. Mernik, V. Žumer, M. Lenič, and E. Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool lisa. *SIGPLAN Not.*, 34(6):68–75, 1999.
17. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.
18. U. Naumann. Optimal dag reversal is NP-complete. Technical Report AIB-2007-05, RWTH Aachen, 2007. Submitted.
19. U. Naumann, M. Maier, J. Riehme, and B. Christianson. Automatic first- and second-order adjoints for Truncated Newton. In *Proceedings of the Workshop on Computer Aspects of Numerical Algorithms (CANA'07)*, Wisla, Poland, 2007.
20. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
21. J. Utke, U. Naumann, C. Wunsch, C. Hill, P. Heimbach, M. Fagan, N. Tallent, and M. Strout. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 2007. To appear.

Uwe Naumann is associate professor for Computer Science at RWTH Aachen University, Germany, and a member of Aachen's Center for Computational Engineering Science. He has obtained his Ph.D. in Mathematics from the Technical University Dresden, Germany, in 1999. Prior to his current affiliation he has held research positions at INRIA Sophia Antipolis, France (post-doctoral researcher, 1999-2000), University of Hertfordshire, UK (senior lecturer, 2000-2002; visiting researcher since 2002), and Argonne National Laboratory, USA (post-doctoral researcher / assistant computer scientist, 2002-2004). His research is inspired by problems in automatic differentiation (AD) with particular focus on adjoint code compilers, efficient algorithms and data structures for AD, and complexity of differentiation algorithms.

Jan Riehme is research associate at the Department of Computer Science, University of Hertfordshire (UH), UK. He holds an M.Sc. in Mathematics from the Technical University Dresden (TUD), Germany (1994). Since then he has worked as a researcher / scientific software developer at TUD (1994-2001 and 2003-2004), UH (2002-2003 and since 2006), and Humboldt Universität zu Berlin, Germany (2004-2006). Mr. Riehme is the main developer of the differen-tiation-enabled NAGWare Fortran compiler. In addition to his research and development activities in the area of derivative code compilers he is interested in the application of AD to a wide range of numerical simulation codes for real-world applications.