RWTH Aachen

Department of Computer Science
Technical Report

An L-Attributed Grammar for Adjoint Code

Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-12

RWTH Aachen · Department of Computer Science · June 2007

An L-Attributed Grammar for Adjoint Code

Uwe Naumann

LuFG Informatik 12, Department of Computer Science, RWTH Aachen University
52056 Aachen, Germany
naumann@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 simple imperative language (a subset of C). Our ideas can easily be applied to any programming language that is suitable for syntax-directed translation. Moreover the conceptual insights are useful in the context of multi-pass generation of adjoint code. Our focus is on correctness. The necessary domain-specific code optimizations are beyond the scope of this paper. We give references to corresponding work in this area.

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 [1–4] for an impressive collection of such applications.

Consider an implementation of a multivariate nonlinear function $f : I\!R^n \rightarrow I\!R$ 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

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

for an appropriate (small) value $h \in I\!R$. 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 [8] 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 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 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. We conclude with an outlook to potential areas of application of the proposed technology in Section 5.

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 $t \in \mathbb{R}$ 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 Figure 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) [6]. 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 $\mathbb{R}^n$. 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 $t \in \mathbb{R}$ 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 Figure 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.

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*.

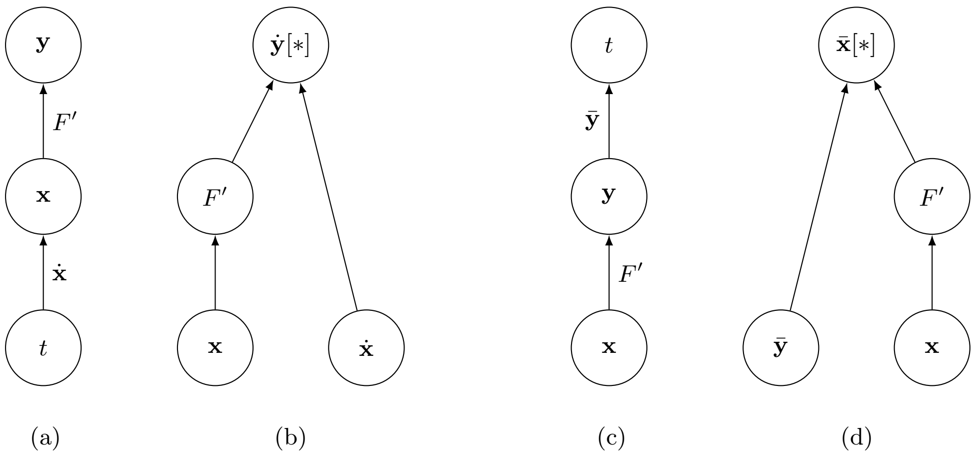

Fig. 1. Linearized $F(\mathbf{x}(t))$ (a), $\dot{F}$ (b) linearized $t(F(\mathbf{x}))$ (c), $\bar{F}$ (d)

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 Equations 5–7. 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 Equations 8–11. 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

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

followed by the backward code

```
pop_v(x); v2_=x_; x_=0;
pop_v(v2); v0_=v2_*v1; v1_=v2_*v0;
pop_v(v1); y_+=v1_;
pop_v(v0); 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 [5]. 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).

The input code is a sequence of one or more statements (**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.

$(P0)$ **code** ::

$$s.k = 0$$

s

$$
\begin{aligned}
\mathbf{code}.c^f = s.c^f & \\
& + \text{"}\texttt{int i;}\text{"} \\
& + \text{"}\texttt{while(pop\_c(i)) \{}\text{"} \\
& + \quad \text{"}\texttt{if(i == 1) \{}\text{"} \\
& + \qquad s.c^b_1 \\
& + \quad \text{"}\texttt{\} else if(i == 2) \{}\text{"} \\
& + \qquad s.c^b_2 \\
& \ \vdots \\
& + \quad \text{"}\texttt{\} else if(i ==}\text{"} + s.k + \text{"}\texttt{) \{}\text{"} \\
& + \qquad s.c^b_{s.k} \\
& + \quad \text{"}\texttt{\}}\text{"}
\end{aligned}
$$

$(P1)$ s ::

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

a

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

The vector assignment $s.c^b = a.c^b$ operates at the elemental level, that is, $s.c^b_i = a.c^b_i$ 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).

$(P1a)$ s ::

$$b.k = s.k$$

b

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

$(P1b) \qquad s ::$

$$l.k = s.k$$

l

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

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

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

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, $s^l.c^b_i = s^r.c^b_i + a.c^b_i$ for $i = 1, \ldots, \alpha$.

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

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

b

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

s^r

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

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

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

l

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

s^r

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

$$
\begin{aligned}
(P3)\qquad & a ::\\
& \qquad\qquad e.k = a.k; \quad e.j = 0\\
& V = e;\\
& \qquad\qquad a.c^f = e.c^f + \text{"}\texttt{push\_c(}\text{"} + a.k + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{push\_v(}\text{"} + V.c^f + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + V.c^f + \text{"}\texttt{= v0;}\text{"}\\
& \qquad\qquad a.c^b_{a.k} = \text{"}\texttt{pop\_v(}\text{"} + V.c^f + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{v0\_ =}\text{"} + V.c^f + \text{"}\texttt{\_;}\text{"}\\
& \qquad\qquad\quad + V.c^f + \text{"}\texttt{\_ = 0;}\text{"}\\
& \qquad\qquad\quad + e.c^b_{a.k}
\end{aligned}
$$

The root of the syntax tree of the expression of the right-hand side has fixed code list variable index 0.

$$
\begin{aligned}
(P4)\qquad & e ::\ V \qquad\qquad e.n = 1\\
& \qquad\qquad a.c^f = \text{"}\texttt{push\_v(v}\text{"} + e.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{v}\text{"} + e.j + \text{"}\texttt{=}\text{"} + V.c^f + \text{"}\texttt{;}\text{"}\\
& \qquad\qquad a.c^b_{e.k} = \text{"}\texttt{pop\_v(v}\text{"} + e.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + V.c^f + \text{"}\texttt{\_+ = v}\text{"} + e.j + \text{"}\texttt{\_;}\text{"}
\end{aligned}
$$

$$
\begin{aligned}
(P5)\qquad & e ::\ C \qquad\qquad e.n = 1\\
& \qquad\qquad a.c^f = \text{"}\texttt{push\_v(v}\text{"} + e.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{v}\text{"} + e.j + \text{"}\texttt{=}\text{"} + C.c^f + \text{"}\texttt{;}\text{"}\\
& \qquad\qquad a.c^b_{e.k} = \text{"}\texttt{pop\_v(v}\text{"} + e.j + \text{"}\texttt{);}\text{"}
\end{aligned}
$$

$$
\begin{aligned}
(P6)\qquad & e^l ::\\
& \qquad\qquad e^r.j = e^l.j + 1; \quad e^r.k = e^l.k\\
& F(e^r)\\
& \qquad\qquad e^l.n = e^r.n + 1\\
& \qquad\qquad e^l.c^f = e^r.c^f\\
& \qquad\qquad\quad + \text{"}\texttt{push\_v(v}\text{"} + e^l.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{v}\text{"} + e^l.j + \text{"}\texttt{=}\text{"} + F.c^f\\
& \qquad\qquad\quad + \text{"}\texttt{(v}\text{"} + e^r.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad e^l.c^b_{e.k} = \text{"}\texttt{pop\_v(v}\text{"} + e^l.j + \text{"}\texttt{);}\text{"}\\
& \qquad\qquad\quad + \text{"}\texttt{v}\text{"} + e^r.j + \text{"}\texttt{\_ =}\text{"} + F_{e^r.j}\\
& \qquad\qquad\quad + \text{"}\texttt{*v}\text{"} + e^l.j + \text{"}\texttt{\_;}\text{"} + e^r.c^b_{e.k}
\end{aligned}
$$

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.

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

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

e^{r_1}

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

Oe^{r_2}

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

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_1})

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

$$s.k = b.k$$

$\{s\}$

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

$(P9) \qquad l :: WHILE(r)$

$$s.k = l.k$$

$\{s\}$

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

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

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

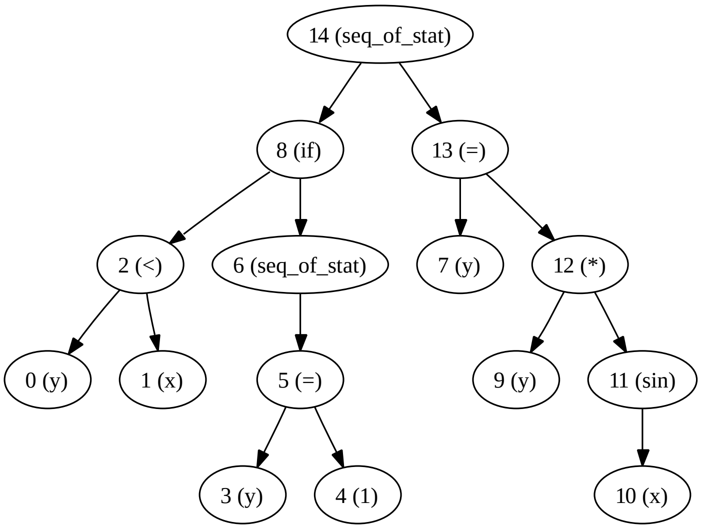

Fig. 2. Parse Tree (generated with `dot`; see `www.graphviz.org`)

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

```
if (y<x) {
  y=1;
}
y=y*sin(x);
```

The parse tree is depicted in Figure 2. Sequences of statement 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:
   - $v_0.c^f = "\texttt{y}"$ (Scanner)
   - $v_1.c^f = "\texttt{x}"$ (Scanner)

– $v_2.c^f = \text{”}\texttt{y < x}\text{”}$ (P10)
– $v_3.c^f = \text{”}\texttt{y}\text{”}$ (Scanner)
– $v_4.c^f = \text{”}\texttt{push\_v(v0); v0 = 1;}\text{”}$ (P5, Scanner)
– $v_5.c^f = v_4.c^f + \text{”}\texttt{push\_c(1); push\_v(y); y = v0;}\text{”}$ (P3, Scanner)
– $v_6.c^f = v_5.c^f$ (P1)
– $v_8.c^f = \text{”}\texttt{if(y < x)\{}\text{”} + v_5.c^f + \text{”}\texttt{\}}\text{”}$ (P8)
– $v_7.c^f = \text{”}\texttt{y}\text{”}$ (Scanner)
– $v_9.c^f = \text{”}\texttt{push\_v(v1); v1 = y;}\text{”}$ (P4, Scanner)
– $v_{10}.c^f = \text{”}\texttt{push\_v(v3); v3 = x;}\text{”}$ (P4, Scanner)
– $v_{11}.c^f = v_{10}.c^f + \text{”}\texttt{push\_v(v2); v2 = sin(v3);}\text{”}$ (P6)
– $v_{12}.c^f = v_9.c^f + v_{11}.c^f + \text{”}\texttt{push\_v(v0); v0 = v1 * v2;}\text{”}$ (P7)
– $v_{13}.c^f = v_{12}.c^f + \text{”}\texttt{push\_c(2); push\_v(y); y = v0;}\text{”}$ (P3, Scanner)
– $v_{14}.c^f = v_8.c^f + v_{13}.c^f$ (P2a, P1)

5. Inherited backward code:
– $v_4.c^b_1 = \text{”}\texttt{pop\_v(v0);}\text{”}$ (P5)
– $v_5.c^b_1 = \text{”}\texttt{pop\_v(y); v0\_ = y\_; y\_ = 0;}\text{”} + v_4.c^b_1$ (P3)
– $v_8.c^b_1 = v_6.c^b_1 = v_5.c^b_1$ (P1, P8)
– $v_9.c^b_2 = \text{”}\texttt{pop\_v(v1); y\_+ = v1\_;}\text{”}$ (P4)
– $v_{10}.c^b_2 = \text{”}\texttt{pop\_v(v3); x\_+ = v3\_;}\text{”}$ (P4)
– $v_{11}.c^b_2 = \text{”}\texttt{pop\_v(v2); v3\_ = cos(v3) * v2\_;}\text{”} + v_{10}.c^b_2$ (P6)
– $v_{12}.c^b_2 = \text{”}\texttt{pop\_v(v0); v2\_+ = v1 * v0\_; v1\_+ = v2 * v0\_}\text{”} + v_9.c^b_2 + v_{11}.c^b_2$ (P7)
– $v_{13}.c^b_2 = \text{”}\texttt{pop\_v(y); v0\_ = y\_; y\_ = 0;}\text{”} + v_{12}.c^b_2$ (P3)
– $v_{14}.c^b_1 = v_8.c^b_1 (P2a); v_{14}.c^b_2 = v_{13}.c^b_2$ (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:

$v_{14}.c^f$
`int i;`
`while (pop_c(i)) {` `if (i==1) {` $v_{14}.c^b_1$ `}` `else if (i==2) {` $v_{14}.c^b_2$ `} }`

4 Implementation and Case Study

We have developed a proof-of-concept implementation based on the scanner and parser generators `flex` and `bison`. The `bison` input has the following structure.

```
...

%token V F IF WHILE
%left '*'

%%

code : s
s : a
  | b
  | l
s : a s { ... };
```

```
13    | b s { ... };
14    | l s { ... };
15 b  : IF '(' c ')' '{' s '}' { ... };
16 l  : WHILE '(' c ')' '{' s '}' { ... };
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%spush_v(v%d); v%d=v%d*v%d;\n",$1.cf,$3.cf,$$.j,$$.j,$1.j,$3.j);
23      sprintf($$.cr[k],"pop_v(v%d); v%d_=v%d_*v%d; v%d_=v%d_*v%d;\n%s%s",$$.j,$1.j,$$.j,$3.j,$3.j,$$.j,$1.j,$3.cr[k],$1.cr[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, WHILE, V, F, as well as the remaining seven single-character tokens are delivered by the lexical analyzer. 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 ($$.cf) is generated on line 22. The adjoint code ($$.cr[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 C-code fragment.

```
if(x<y) {
  x=x*y;
  while (y<x) {
   x=sin(x*y);
  }
}
```

With x=-5.0 and y=-0.5 as inputs the while-loop is traversed twice 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.

```
if (x<y) {
  push_c(1);
  push_v(v0); v0=x;
```

```
4   push_v(v1); v1=y;
5   push_v(v2); v2=v0*v1;
6   push_v(x); x=v2;
7   while (y<x) {
8     push_c(2);
9     push_v(v0); v0=x;
10    push_v(v1); v1=y;
11    push_v(v2); v2=v0*v1;
12    push_v(v3); v3=sin(v2);
13    push_v(x); x=v3;
14  }
15 }
16 int i;
17 while (pop_c(i)) {
18   if (i==1) {
19     pop_v(x); v2_=x_; x_=0;
20     pop_v(v2); v0_=v2_*v1; v1_=v2_*v0;
21     pop_v(v1); y_+=v1_;
22     pop_v(v0); x_+=v0_;
23   }
24   else if (i==2) {
25     pop_v(x); v3_=x_; x_=0;
26     pop_v(v3); v2_=cos(v2)*v3_;
27     pop_v(v2); v0_=v2_*v1; v1_=v2_*v0;
28     pop_v(v1); y_+=v1_;
29     pop_v(v0); x_+=v0_;
30   }
31 }
```

The two assignment statements are enumerated by pushing unique indexes onto the control stack (lines 2 and 8; grammar rule P3). Assignment-level code list are built and the values of all overwritten variables are saved on the value stack (lines 3 – 6 and 9–13; 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 16, 17, 18, and 24); 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 25–29; grammar rules P3–P7). For example, the adjoint corresponding to the assignment on line 12 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 26).

5 Conclusion

We are actively involved in the development of the adjoint code compilers OpenAD [10] and of the differentiation-enabled NAGWare Fortran compiler [9]. Conceptually, the semantic actions that are performed on the respective internal representations are similar to those proposed in this paper.

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 or the differentiation-enabled NAGWare Fortran compiler. Due to the lack of static program analysis and the corresponding optimizations (see, for example, [7]) 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. Currently we are investigating the syntax-directed compilation of adjoint Matlab code. Matlab is certainly not a single-pass language. However we expect to find a suitable large enough subset having this property.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
2. 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.
3. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
4. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.
5. 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.
6. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
7. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
8. 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.
9. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
10. 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*, 2006. To appear.