Control Flow Reversal for Adjoint Code Generation

Uwe Naumann and Jean Utke
Mathematics and Computer Science Division, Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60438, USA
{naumann,utke}@mcs.anl.gov

Andrew Lyons
Vanderbilt University
2201 West End Avenue, Nashville, TN 37235, USA
andrew.m.lyons@vanderbilt.edu

Abstract

We describe an approach to the reversal of the control flow of structured programs. It is used to automatically generate adjoint code for numerical programs by semantic source transformation. After a short introduction to applications and the implementation tool set, we describe the building blocks using a simple example. We then illustrate the code reversal within basic blocks. The main part of the paper covers the reversal of structured control flow graphs. We show the algorithmic steps for simple branches and loops and give a detailed algorithm for the reversal of arbitrary combinations of loops and branches in a general control flow graph. [1]

1. Introduction

Research into the reversal of the control flow of structured programs is part of the "Adjoint Compiler Technology and Standards"[2] (ACTS) project, a collaboration between MIT, Rice University, and Argonne National Laboratory/University of Chicago. It has a twofold goal. The first is to provide an open platform, called OpenAD, for developing of algorithms for the automatic semantic transformation of numerical programs. The second is to use this platform to create an adjoint compiler for generating efficient adjoint code for the MIT General Circulation Model[3] [8]. A control flow reversal is implemented as one of the fundamental algorithms provided by OpenAD. The principal setup of OpenAD and its interaction with other software is illustrated in Fig. 1.

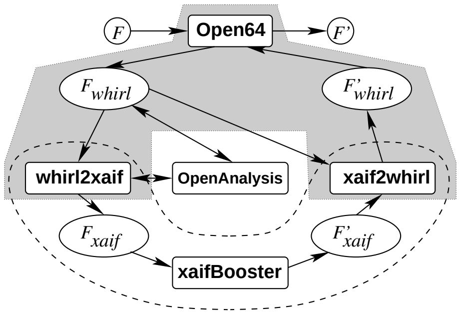


Figure 1. OpenAD setup

OpenAD focuses on the semantic transformation of numerical codes. In order to achieve language independence of the transformation algorithms, the numerically relevant core is extracted and represented in an intermediate XML format called xaif.[4]. The algorithmic component called xaifBooster accepts xaif and transforms it. The modified xaif is then back-translated into the original programming language. In Fig. 1 all language-dependent parts are shown in the shaded area. The core OpenAD components are encircled by the dashed line. Currently OpenAD uses two front ends for the translation to and from xaif. For C/C++ codes we use the EDG[5] front-end in combination with Sage 3.[6]. In this paper we concentrate on the Fortran front-

[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38 and by NSF under ITR contract OCE-0205590.

[2] See `www.autodiff.org/ACTS`.

[3] See `mitgcm.org`.

[4] See `www.mcs.anl.gov/xaif`

[5] See `www.edg.com`

[6] See `www.llnl.gov/CASC`

end Open64.[7]

Consider a Fortran code implementing a vector function F as in Eqn. (1). The Open64 front-end performs a lexical, syntactic, and semantic analysis and produces an intermediate representation of F in the *whirl* format.[8] OpenAnalysis[9] is a framework that provides typical compiler analyzes and can be used to implement domain-specific analyzes. The whirl2xaif component creates a representation of the numerical core of F in xaif format, F_{xaif}, including call graphs and control flow graphs built by using OpenAnalysis. The transformation algorithms implemented in xaifBooster then modify F_{xaif} and return F'_{xaif}. In the case of adjoint code generation F'_{xaif} represents the differentiated code. F'_{xaif} and F_{whirl} are combined by xaif2whirl to construct a *whirl* representation of the differentiated code. The unparser of Open64 then transforms the *whirl* representation back into Fortran.

The paper is structured as follows. In Sec. 2 we briefly introduce the elements of the "big picture" that our control flow reversal algorithm fits in. A method for building adjoint basic blocks is discussed in Sec. 3. In Sec. 4 we derive an algorithm for the reversal of structured control flow graphs. We draw conclusions in Sec. 5.

2. The Big Picture

Automatic differentiation (AD) [2, 3, 5] is a set of techniques for transforming numerical programs into derivative code that can be used to compute derivatives of vector functions such as Jacobians, Hessians, or higher-order Taylor coefficients. A detailed description of the mathematical foundations of AD is beyond the scope of this paper. Refer to [4] for a discussion of the theory.

The adjoint of a program implementing a vector function

$$\mathbf{y} = F(\mathbf{x}), \quad \mathbf{x} \in I\!R^n, \mathbf{y} \in I\!R^m \quad , \tag{1}$$

is obtained by the reverse mode of AD. It represents a semantically modified version of the original program. Given values for $\mathbf{x}$ and the adjoints of the original outputs $\bar{\mathbf{y}}$, the adjoint program computes the transposed Jacobian vector product

$$(F')^T \cdot \bar{\mathbf{y}} \equiv \bar{\mathbf{x}} \quad . \tag{2}$$

This process is best illustrated with the help of a simple example. Consider the following Fortran implementation of a vector function $\mathbf{y} = F(\mathbf{x})$, where $\mathbf{x} \in I\!R^2$, $\mathbf{y} \in I\!R^3$.

```
i:=1
do while (i<3)
  if (i<2) then
    y(2):=sin(x(1))
  else
    y(1):=cos(x(2))
  end if
  i:=i+1
end do
y(3):=y(1)*y(2)
```

The example has been crafted to illustrate some important features of control flow reversal for adjoint codes. We assume the availability of a control flow graph (CFG) for the code for F. A forward run is required to store information regarding the specific path taken through the CFG as well as numerical values needed for the adjoint computation during the following reverse sweep. Domain-specific data flow analyzes for reverse-mode AD have been developed to determine these sets of numerical values while minimizing the conservative overestimate (see [7]). For our simple example no numerical values need to be stored as none of the variables gets overwritten. However, we require additional statements to store the path through the CFG. To this end we push a unique identifier onto a stack for each branch inside the if-statement and bu counting the number of iterations performed by the loop.

```
i:=1
ctr:=0
do while (i<3)
  if (i<2) then
    y(2):=sin(x(1))
    push(1)
  else
    y(1):=cos(x(2))
    push(0)
  end if
  i:=i+1
  ctr:=ctr+1
end do
push(ctr)
y(3):=y(1)*y(2)
```

We call this code version the augmented forward code. The adjoint code is obtained by applying Eqn. (2) to each statement $y = \phi(x)$ in the original code yields $\bar{x} = \frac{\partial \phi}{\partial x}\bar{y}$. Alternatively Eqn. (2) can be applied to whole basic blocks as described in Sec. 3. The control flow is reversed by executing all statements in reverse order. The number of iterations of the adjoint loop is equal to the number of iterations performed by the original loop for the current set of inputs. The decision about which branch to take is based on what has been stored during the forward sweep. The adjoint version of variables in the original code is marked by the suffix `_adj`.

```
y_adj(1):=y(2)*y_adj(3)
y_adj(2):=y(1)*y_adj(3)
rBound:=pop()
do rCtr:=1,rBound
  branchId:=pop()
  if (branchId=1) then
    x_adj(1):=cos(x(1))*y_adj(2)
  else
    x_adj(2):=-sin(x(2))*y_adj(1)
```

[7]See hipersoft.cs.rice.edu/open64.

[8]This is the Open64-specific internal representation.

[9]See www.hipersoft.rice.edu/openanalysis.

```
  end if
end do
```

The local partial derivatives used in the adjoint statements are the results of applying the well-known differentiation rules. For example, $\frac{\partial(a \cdot b)}{\partial b} = a$ and $\frac{\partial \cos(a)}{\partial a} = -\sin(a)$. The fact that Eqn. (2) can be applied per statement is an immediate consequence of the chain rule.

Adjoint codes permit the accumulation of the Jacobian at a cost proportional to the number of outputs m. Gradients ($m = 1$) can be obtained at a small constant multiple of the cost of evaluating the function itself. This property is of particular interest for the ACTS project. For example, in the context of data assimilation in oceanography the gradient of some objective with respect to the grid points of a very fine discretization of the ocean is required. Potentially, the number of input variables n is on the order of 10^9. Neither forward-mode AD nor approximation by finite difference quotients represents a feasible approach, as each has a complexity that is $O(n)$.

From now on we will consider a representation of the CFG as the directed graph G. Figure 2 (a) shows G for the example code introduced earlier.[10] The basic block represented by vertex 3 is the result of a canonicalization step performed by the front-end that substitutes a Boolean variable for the loop condition. Formally, we define structured CFGs as follows.

Definition 1 *A structured CFG $G = (V, E)$ is a directed graph consisting of a list of integer vertices V and a list of edges $E \subseteq V \times V$. Vertices $i \in V$ have a type, where $type(i) \in$ {*ENTRY, BASICBLOCK, LOOP, ENDLOOP, BRANCH, ENDBRANCH, EXIT*}. G has a unique entry $i \in V : type(i) =$* ENTRY, *$|P(i)| = 0$ and a unique exit $j \in V : type(i) =$* EXIT, *$|S(j)| = 0$.*[11] BASICBLOCK *and* ENDLOOP *vertices have one predecessor and one successor each. Both the number of predecessors and successors of* LOOP *vertices is equal to two. The number of predecessors of* BRANCH *and* EXIT *vertices as well as the number of successors of* ENDBRANCH *and* ENTRY *vertices is equal to one.*

Edges $e \in E$ can have labels. All edges emanating from a BRANCH *vertex are labeled with mutually distinct identifiers. Edges whose source is a* LOOP *vertex that lead into the loop body carry the label "1".*

In this paper we follow a unified approach for all kinds of loops. A formal distinction between for-loops, pre-loops, and post-loops is unnecessary, as shown in Sec. 4.4. We use the vertex type LOOP for all these constructs.

[10] `graphviz` is used to visualize the graphs as part of xaifBooster's debug output (see `www.research.att.com/sw/tools/graphviz`).

[11] We denote the set of predecessors and successors of a vertex i by $P(i)$ and $S(i)$, respectively. The cardinality of a set A is denoted by $|A|$.

Basic blocks can contain assignments and subroutine calls. The latter require the code reversal to be extended to the call graph. This is a research topic in itself, and a variety of solutions have been proposed (see, for example, [4, Chapter 12]. The techniques proposed in this paper can be used by all of them.

We present a method for transforming a subroutine that implements a vector function as in Eqn. (1) into a semantically modified version that computes the product of the transposed Jacobian with a vector. If the subroutine is given in form of a CFG, then the transformation $G \rightarrow (\overrightarrow{G}, \overleftarrow{G})$ consists of two parts.

$\overrightarrow{G} = (\overrightarrow{V}, \overrightarrow{E})$ represents the augmented forward code and is equivalent to G with all basic blocks semantically modified as in Eqn. (4) and Eqn. (5). Additional basic blocks are inserted that contain instructions for storing the flow of control as shown in Sec. 4.

The CFG of the adjoint code $\overleftarrow{G} = (\overleftarrow{V}, \overleftarrow{E})$ is built by using the information stored during the execution of the augmented forward code. It executes the adjoint basic blocks, constructed as in Eqn. (6), in reverse order.

3. Adjoint Basic Blocks

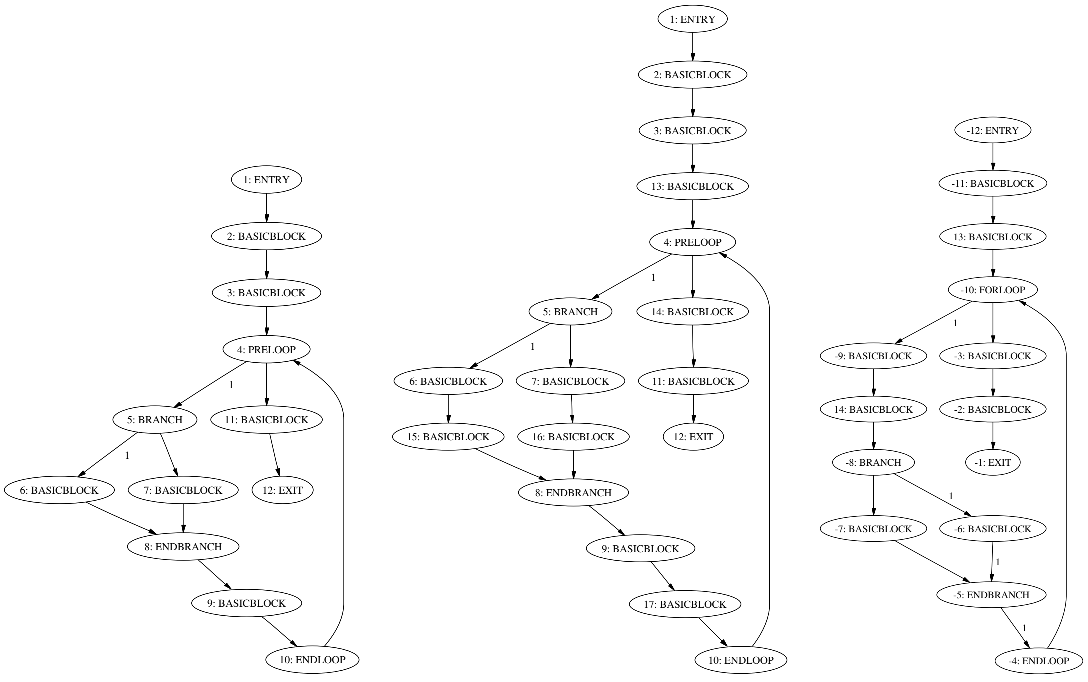
A potential improvement of the adjoint code can be achieved by preaccumulating Jacobians of basic blocks using elimination techniques in linearized computational graphs [9]. Basic blocks can be viewed as local vector functions as in Eqn. (1). The preaccumulation algorithm implemented in xaifBooster generates optimized code for the computation of F' in the following manner. We consider the simple example of a basic block in Eqn. (3).

$$\begin{aligned} &v_3 := v_1 * v_2;\ v_4 := v_1 * v_3 \\ &v_6 := cos(v_4);\ v_7 := (v_3 * v_2) * v_4 \end{aligned} \tag{3}$$

Linearization implies augmenting the code to include the computation of local partial derivatives $c_{ji} = \frac{\partial v_j}{\partial v_i}$ for each elemental operation as in Eqn. (4).

$$\begin{aligned} v_3 &:= (v_1 * v_2);\ c_{31} := v_2;\ c_{32} := v_1 \\ v_4 &:= (v_1 * v_3);\ c_{41} := v_3;\ c_{43} := v_1 \\ t &:= (v_2 * v_3);\ v_7 := (v_4 * t) \\ c_{t3} &:= v_2;\ c_{t2} := v_3;\ c_{7t} := v_4;\ c_{74} := t \\ v_6 &:= \sin(v_4);\ c_{64} := \cos(v_4) \end{aligned} \tag{4}$$

This may require the assignment of intermediates as in the expression $(v_3 * v_2) * v_4$ to temporary variables. A linearized computational graph (LCG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is derived from the linearized code, in the form of a directed acyclic graph. The vertices $v \in \mathcal{V} = \mathcal{X} \cup \mathcal{Z} \cup \mathcal{Y}$ represent input variables $\in \mathcal{X}$, intermediates $\in \mathcal{Z}$, and output variables $\in \mathcal{Y}$. The edges $(v_i, v_j) \in \mathcal{E}$ are labeled with their respective local partial derivatives c_{ji}. Fig. 3 (a) shows $\mathcal{G}$ represented by our linearized vector function.



(a) (b) (c)

Figure 2. G, $\overrightarrow{G}$, **and** $\overleftarrow{G}$.

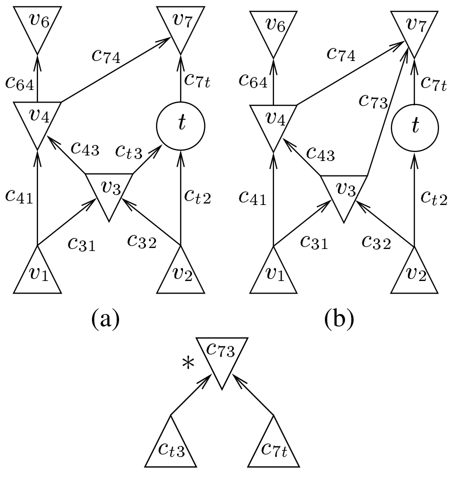


Figure 3. $\mathcal{G}$, $\mathcal{G}'$, **and JAE of** $(3,t)_f$

3.1. Edge Elimination

The collection of algorithms in xaifBooster provides three elimination techniques for the accumulation of Jacobian matrices, known as vertex [6], edge, and face elimination [9]. For our example, we will use edge elimination. An edge (v_i, v_j), (i, j) for short, can be either *front* or *back* eliminated, denoted by $(i,j)_f$ or $(i,j)_b$, respectively. Front elimination of (i, j) is executed by connecting all vertices in the predecessor set $P_{(i,j)_f} = \{v_i\}$ with all vertices in the successor set $S_{(i,j)_f} = S_{v_j}$. These new edges are $(i,k) : v_k \in S_{v_j}$. Only edges whose target is not an output vertex can be front eliminated.

Back elimination of (j, k) is executed by connecting all vertices in the predecessor set $P_{(j,k)_b} = P_{v_j}$ with all vertices in the successor set $S_{(j,k)_b} = \{v_k\}$. The new edges are $(i,k) : v_i \in P_{v_j}$. Only edges whose source is not an input variable can be back eliminated.

In both cases the new edges are labeled with the values $c_{ki} := c_{ji} * c_{kj}$, and the edge (i,j) is removed. If an edge elimination $(i,j)_f$ or $(j,k)_b$ would create an already existing edge (i,k), the label of (i,k) is incremented $c_{ki} := c_{ki} + c_{ji} * c_{kj}$. This is referred to as *absorption*, as opposed to the creation of new edges that represent *fill-in*.

If at any point during the elimination process an intermediate vertex has no more in- or out-edges, the vertex and all incident edges are removed from the graph. Thereby, a complete sequence of edge eliminations reduces $\mathcal{G}$ to a bipartite graph consisting only of vertices $\in \mathcal{X} \cup \mathcal{Y}$ and edges whose labels represent the Jacobian entries.

Each multiplication or combined increment / multiplication on the edge labels implies a Jacobian accumulation expression (JAE), which is stored in a list. In our example the elimination $(3,t)_f$ implies a single JAE, shown in Fig. 3 (c). Fig. 3 (b) shows $\mathcal{G}$ after edge $(3,t)_f$ has been eliminated. To understand why $(3,t)_f$ was eliminated first, we must examine our method of choosing eliminations.

3.2. Heuristics

Use of the chain rule in preaccumulation yields a computationally complex search space when attempting to determine the optimal sequence of edge eliminations. We use two different types of heuristics to determine our elimination sequences. One group attempts to minimize the number of operations, that is, the number of JAEs implied by a complete elimination sequence. The other group attempts to maximize data locality in the generated code. In order to narrow the choice to a single elimination target, it may be necessary to successively apply several heuristics.

For our example, we are primarily interested in maximizing data locality and therefore choose a heuristic from the second group as the first heuristic in the sequence. This heuristic is called *highest sibling degree*, or simply HS.

HS, like all edge elimination heuristics, is a mapping from a set of elimination targets Θ to a subset $\Theta' \subseteq \Theta$. An elimination target consists of an edge $e \in \mathcal{E}$ and an elimination direction that can be either front or back.

HS will choose elimination targets that have the maximum *sibling degree* denoted by sd_{max}. $\forall \theta \in \Theta$, the sibling degree of θ with respect to the previous elimination θ^-, denoted $sd_{\theta^-}(\theta)$, is defined by

$$sd_{\theta^-}(\theta) = |S_\theta \cap S_{\theta^-}| * |P_\theta \cap P_{\theta^-}| \quad .$$

The maximum sibling degree is defined as follows:

$$sd_{max}(\theta^-) = \max_{\forall \theta \in \Theta} \{sd_{\theta^-}(\theta)\} \quad .$$

HS selects $\Theta' = \{\theta : sd_{\theta^-}(\theta) = sd_{max}(\theta^-)\}$. In the case when $sd_{max} = 0$, $\Theta' = \Theta$. The elimination of a target θ^+ directly following the elimination of a target θ with $sd_\theta(\theta^+) > 0$ creates code that stipulates the immediate absorption of a new edge, which should still be resident in fast memory.

Note that if the last elimination was a front (back) elimination, any edge being considered for back (front) elimination must have a sibling degree of 1. Thus, HS can choose front (back) eliminations following a back (front) elimination only when the maximum sibling degree is 1.

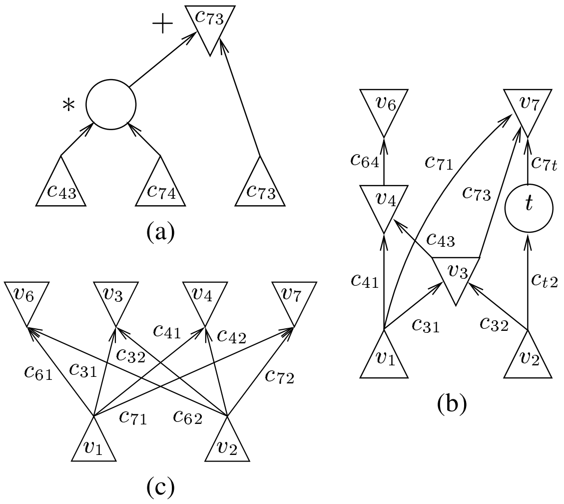


Figure 4. JAE of $(4,7)_b$, $\mathcal{G}''$, and F'

When HS determines a sibling set with $|\Theta'| > 1$, we need to narrow the choice by applying another heuristic as a tiebreaker. For our example, we use a lowest Markowitz degree heuristic (LM). The Markowitz degree for any front or back elimination is defined as $|S|$ or $|P|$, respectively; see [1] for an in-depth description of Markowitz-type heuristics for edge elimination on an LCG.

If more than one edge exists in the most favorable equivalence class for LM, reverse mode is used to acquire a unique elimination. Because forward and reverse mode are implemented based on a single topological sort on $\mathcal{G}$, they will always return a unique selection.

3.3. Preaccumulation

Our first elimination on the graph shown in Fig. 3 must be decided by LM because there isn't any data in memory yet. LM chooses both $(3,t)_f$ and $(2,t)_f$ because they are in the same equivalence class, with Markowitz degree 1. Reverse mode chooses $(3,t)_f$ because vertex 3 occurs after vertex 2 in a topological sort of our LCG. Now that we have made an elimination and we have some data in fast memory, we can make use of data locality in order to expedite our accumulation.

Inspection of Fig. 3 (b) reveals that both $(3,4)_f$ and $(4,7)_b$ are siblings (of sibling degree 1) of $(3,t)_f$. However, we cannot eliminate edge $(3,4)_f$ because vertex 4 is a dependent vertex. Without better analyzes any vertex with more than 1 out-edge must be considered an output variable, in case it appears in some right-hand side later in the code. Hence, every vertex in our graph except for `t` will be treated as an output. One of the two JAE graphs generated by the elimination of $(4,7)_b$ is shown in Fig. 4 (a); the resulting LCG appears in Fig. 4 (b).

Fig. 4 (c) shows the complete bipartite graph that represents the Jacobian F'. After a complete sequence of edge eliminations, the JAE graphs that represent the remaining edges are identified as Jacobian entries J_{lk} referenced in Eqn. (5) and Eqn. (6). Finally, xaifBooster, when generating the modified xaif output F'_{xaif}, iterates through the list of JAE graphs and generates the corresponding xaif representation that is unparsed back into Fortran. For our example a representation of this code is listed in the center column of Eqn. (5). The left column of Eqn. (5) shows the respective elimination steps. The right column indicates which labels are Jacobian entries that are pushed onto the stack. For $(1,3)$ and $(2,3)$ the original edge labels are already Jacobian entries as defined in Eqn. (4).

$$\begin{array}{rll}
 & c_{31} & \text{push}(J_{31}) \\
 & c_{32} & \text{push}(J_{32}) \\
(v_3,t)_f & c_{73} := c_{t3} * c_{7t} & \\
(v_4,v_7)_b & c_{71} := c_{41} * c_{74} & \\
 & c_{73} := c_{73} + c_{74} * c_{43} & \\
(v_3,v_7)_b & c_{71} := c_{71} + c_{73} * c_{31} & \text{push}(J_{71}) \\
 & c_{72} := c_{73} * c_{32} & \\
(t,v_7)_b & c_{72} := c_{72} + c_{7t} * c_{t2} & \text{push}(J_{72}) \\
(v_4,v_6)_b & c_{61} := c_{64} * c_{41} & \\
 & c_{63} := c_{64} * c_{43} & \\
(v_3,v_6)_b & c_{61} := c_{61} + c_{63} * c_{31} & \text{push}(J_{61}) \\
 & c_{62} := c_{63} * c_{32} & \text{push}(J_{62}) \\
(v_3,v_4)_b & c_{41} := c_{41} + c_{43} * c_{31} & \text{push}(J_{41}) \\
 & c_{42} := c_{43} * c_{32} & \text{push}(J_{42})
\end{array} \qquad (5)$$

The subsequent reverse sweep pops these values and performs a (sparse) transposed Jacobian vector product with the vector of the adjoint variables $\bar{v}_i$ that correspond to the original variables v_i.

$$\begin{array}{ll}
 & \text{pop() returns:} \\
\bar{v}_2 := pop() * \bar{v}_4 & J_{42} \\
\bar{v}_1 := pop() * \bar{v}_4 & J_{41} \\
\bar{v}_2 := \bar{v}_2 + pop() * \bar{v}_6 & J_{62} \\
\bar{v}_1 := \bar{v}_1 + pop() * \bar{v}_6 & J_{61} \\
\bar{v}_2 := \bar{v}_2 + pop() * \bar{v}_7 & J_{72} \\
\bar{v}_1 := \bar{v}_1 + pop() * \bar{v}_7 & J_{71} \\
\bar{v}_2 := \bar{v}_2 + pop() * \bar{v}_3 & J_{32} \\
\bar{v}_1 := \bar{v}_1 + pop() * \bar{v}_3 & J_{31} \\
\bar{v}_7 := \bar{v}_6 := \bar{v}_4 := \bar{v}_3 := \bar{0} &
\end{array} \qquad (6)$$

The adjoints of certain basic block outputs need to be set to zero explicitly if the basic block appears in the context of a larger program. The discussion of the conditions is beyond the scope of this paper.

4. Adjoint Control Flow

In this section we describe our approach to the automatic reversal of the control flow. The method is based on a topologically sorted vertex list V obtained by an algorithm described in Sec. 4.1. The transformation $V \to (\overrightarrow{V}, \overleftarrow{V})$ is defined in Sec. 4.2. Specifics of the reversal for branches and loops are discussed in Sections 4.3 and 4.4, respectively. The heart of this section is an algorithm for for reversing structured CFGs defined by Def. (1).

4.1. Topological Sort

Algorithm 1 **(**$topsort(i) : i \in V$**)** *We start with an empty vertex stack S. The functions push, pop, and top are the usual stack access routines. The visited flag of all vertices is assumed to be* `false`*. A temporary list V' is used to hold all vertices initially. V is emptied. The algorithm is called with the ENTRY vertex as argument.*

$i \in V'$, $type(i)$ =ENTRY*:*
1 **If** *(*$visited(i)$*)* **Return**
2 $visited(i) :=$ `true`
3 **If** *(*$type(i) =$ ENDBRANCH*)* **Then**
4 $S.push(i)$; **Return**
5 **Endif**
6 $V.append(i)$
7 **If** *(*$type(i) =$ LOOP*)* **Then**
8 $topsort(j) : e \equiv (i,j) \in E \wedge label(e) = 1$
9 $topsort(k) : e' \equiv (i,k) \in E \wedge label(e') \neq 1$
10 **Else**
11 $\forall (i,j) \in E : topsort(j)$
12 **Endif**
13 **If** *(*$type(i) =$ BRANCH*)* **Then**
14 $V.append(S.top())$
15 $\forall (S.top(), j) \in E : topsort(j)$
16 $S.pop()$
17 **Endif**
18 **Return**

We require that any given vertex succeeds its dominators and it precedes its post-dominators. In particular, this requirement implies that a ENDLOOP vertex succeeds any vertex in the corresponding loop body. Furthermore, it is ensured that any ENDBRANCH vertex succeeds the vertices in either of the branches.

Every vertex is visited once (lines 1..2). For BRANCH vertices, the corresponding ENDBRANCH is appended to the sorted vertex list only after all vertices inside the branches have been processed. The algorithm is then applied recursively to the successor of the ENDBRANCH vertex (lines 3..5 and 13..17). Loop bodies are sorted prior to the successor of a LOOP vertex (lines 7..10). By default, the algorithms always proceeds to the successors of the given vertex (line 11).

4.2. Vertex Transformation

Alg. 2 defines the transformation of vertices in G into vertices in $\overrightarrow{G}$ and $\overleftarrow{G}$.

Algorithm 2 ($V \rightarrow (\overrightarrow{V}, \overleftarrow{V})$ **)**

$\forall i \in V$:

$i \rightarrow (\overrightarrow{i} \in \overrightarrow{V}, \overleftarrow{i} \in \overleftarrow{V})$:

If $type(i)$ = ENTRY **Then**

  $type(\overrightarrow{i})$:= ENTRY; $type(\overleftarrow{i})$:= EXIT

ElseIf $type(i)$ = EXIT **Then**

  $type(\overrightarrow{i})$:= EXIT; $type(\overleftarrow{i})$:= ENTRY

ElseIf $type(i)$ = BASICBLOCK **Then**

  $type(\overrightarrow{i})$:= BASICBLOCK *(as in Equations (4, 5))*

  $type(\overleftarrow{i})$:= BASICBLOCK *(as in Equation (6))*

ElseIf $type(i)$ = BRANCH **Then**

  $type(\overrightarrow{i})$:= BRANCH

  $type(\overleftarrow{i})$:= ENDBRANCH

ElseIf $type(i)$ = ENDBRANCH **Then**

  $type(\overrightarrow{i})$:= ENDBRANCH

  $type(\overleftarrow{i})$:= BRANCH

  $cond(\overleftarrow{i})$:= $\boxed{\mathit{mkr(i)=branchId}}$

ElseIf $type(i)$ = LOOP **Then**

  $type(\overrightarrow{i})$:= LOOP

  $type(\overleftarrow{i})$:= ENDLOOP

ElseIf $type(i)$ = ENDLOOP **Then**

  $type(\overrightarrow{i})$:= ENDLOOP

  $type(\overleftarrow{i})$:= FORLOOP

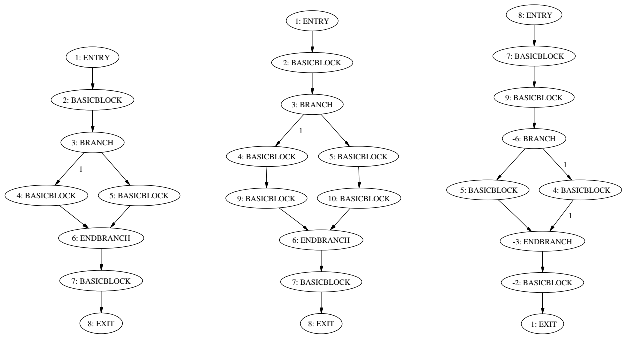
  $iter(\overleftarrow{i})$:= $\boxed{\mathit{rCtr(j)\!:=\!1,rBound(j)}}$: $(i, j) \in E$

Endif

One possible approach to building augmented and adjoint basic blocks has been discussed in Sec. 3. The adjoint of an ENDBRANCH vertex i is a BRANCH vertex. The decision on which branch to execute during the adjoint sweep is made by matching the restored $branchId$ with the $mkr(i)$, which are unique for all branches that are merged at i. The $mkr(i)$ represent the labels of the corresponding edges. Adjoint loops are FORLOOPs performing $rBound(j)$ iterations of the adjoint of the original loop body, where $rBound(j)$ is the result of counting the number of iterations during the augmented forward sweep. Further details can be found in Sec. 4.3 and Sec. 4.4.

4.3. Branches

Consider the CFG G in Figure 5 (a). It shows a two-way branch preceded and succeeded by a basic block and results



G $\overrightarrow{G}$ $\overleftarrow{G}$

```
2: <B2>
3: if (...)
4:   <B4>
   else
5:   <B5>
6: end if
7: <B7>
```

(a)

```
 2: <B2>
 3: if (...)
 2:   <B4>
 9:   push(1)
    else
 5:   <B5>
10:   push(0)
 6: end if
 7: <B7>
```

(b)

```
-7: <B7>
 9: branchId:=pop()
-6: if (branchId=1)
-4:   <B4>
    else
-5:   <B5>
-3: end if
-2: <B2>
```

(c)

Figure 5. Branch Reversal

from an `IF-THEN-ELSE` statement. The edge leading into the `true` branch is labeled with 1. For multi-way branches all edges are labeled with a unique identifier. $\overrightarrow{G}$ is shown in Figure 5 (b). Two new basic blocks (9 and 10) are inserted that contain a single statement each. A call to `push` stores the value of the corresponding edge label on the stack. To ensure the correctness of the value that is pushed, we require the edges leading into the ENDBRANCH vertex be marked by the identifier of the corresponding edge emanating from the matching BRANCH vertex. This is achieved by a simple traversal algorithm based on Algorithm 1.

$\overleftarrow{G}$ is shown in Figure 5 (c). Vertices that correspond to some vertex in G are marked with the respective negative index. The ENDBRANCH vertex in G becomes a BRANCH vertex in $\overleftarrow{G}$. The latter is preceded by a new basic block (9) that `pop`s the identifier of the branch taken during the forward run from the stack. The corresponding adjoint branch is then executed.

Algorithm 3 (Adjoint of Simple Branch) *The adjoint code uses a stack S to store the control flow (push) and to restore it in reversed order (pop). Branches*[12] *are marked with a unique identifier $mkr(i)$,*[13] *where i is the*

[12]These are the paths connecting a given BRANCH vertex with its given ENDBRANCH vertex.

[13]Only one branch is executed for given inputs of the original program. Therefore, a second index for addressing the single branches explicitly is not required.

corresponding ENDBRANCH vertex.

$\forall e \equiv (i,j) \in E$:

1 **If** $type(j)$ =ENDBRANCH **Then**
2 $\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} : \ k \ni \boxed{\textit{S.push(mkr(j))}}$
3 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k), (k, \overrightarrow{j})\}$
4 **Else**
5 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, \overrightarrow{j})\}$
6 **Endif**

7 **If** $type(i)$ =ENDBRANCH **Then**
8 $\quad \overleftarrow{V} := \overleftarrow{V} \cup \{k'\} : \ k' \ni \boxed{\textit{branchId:=S.pop()}}$
9 $\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(k', \overleftarrow{i}), (\overleftarrow{j}, k')\}$
10 **Else**
11 $\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j}, \overleftarrow{i})\}$
12 **Endif**

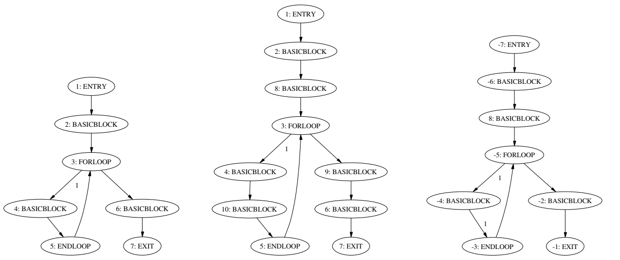
$\overrightarrow{V}$ and $\overleftarrow{V}$ are constructed as in Sec. 4.2. Edges $(\overrightarrow{i}, \overrightarrow{j}) \in \overrightarrow{G}$ are constructed similar to their originals $(i,j) \in G$ (line 5) with an exception if the target j is an ENDBRANCH vertex. In this case a new basic block containing a statement that pushes the unique marker of the current branch onto S needs to precede $\overrightarrow{j}$ (lines 1..3).

If the source i of an edge $(i,j) \in G$ is not an ENDBRANCH vertex, then its adjoint is obtained by switching its direction (line 11). Otherwise, the new basic block that restores the *branchId* value needs to precede $\overleftarrow{i}$ (lines 7..9).

4.4. Loops

The CFG of a simple loop is shown in Fig. 6 (a). We reverse loops by counting the executions of the loop body performed during the forward sweep. Therefore, a loop counter variable `ctr` is initialized before the loop statement. After the end of the loop body `ctr` is incremented by one. The final value is pushed onto the stack right after the end of the loop. This procedure results in the augmented forward code and the corresponding augmented CFG that are shown in Fig. 6 (b). The adjoint code is displayed in Fig. 6 (c). Any type of loop is transformed into a FORLOOP with loop index `rCtr` that executes the adjoint of the loop body exactly `ctr` times, where `ctr` is the number of executions of the loop body while running the forward code. This value is restored from the stack as `rBound`. In Fig. 6 (c) all adjoint CFG vertices are marked with the negative of the index of their corresponding original vertex.

Algorithm 4 (Adjoint of Simple Loop) *Again, a stack S is used to store the control flow (push) and to restore it in reversed order (pop). The number of executions of the loop body corresponding to a LOOP vertex i are counted during the augmented forward sweep. The counters $ctr(i)$ are equal to zero initially.*



G $\quad\quad \overrightarrow{G}$ $\quad\quad \overleftarrow{G}$

2: `<B2>`
3: `do ...`
4: `  <B4>`
5: `end do`
6: `<B6>`

(a)

2: $\overrightarrow{\texttt{<B2>}}$
8: `ctr:=0`
3: `do ...`
4: $\quad\overrightarrow{\texttt{<B4>}}$
10: `  ctr:=ctr+1`
5: `end do`
9: `push(ctr)`
6: $\overrightarrow{\texttt{<B6>}}$

(b)

-6: $\overleftarrow{\texttt{<B6>}}$
8: `rBound:=pop()`
-5: `do rCtr:=1,rBound`
-4: $\quad\overleftarrow{\texttt{<B4>}}$
-3: `end do`
-2: $\overleftarrow{\texttt{<B2>}}$

(c)

Figure 6. Loop Reversal

$\forall e \equiv (i,j) \in E$:

If $type(j)$ =LOOP $\wedge \ i < j$ **Then**
$\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} : \ k \ni \boxed{\textit{ctr(j):=0}}$
$\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k), (k, \overrightarrow{j})\}$
ElseIf $type(j)$ =ENDLOOP **Then**
$\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} : \ k \ni \boxed{\textit{ctr(j):=ctr(j)+1}}$
$\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k), (k, \overrightarrow{j})\}$
ElseIf $type(i)$ =LOOP $\wedge \ label(e) \neq 1$ **Then**
$\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} : \ k \ni \boxed{\textit{S.push(ctr(i))}}$
$\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k), (k, \overrightarrow{j})\}$
Else
$\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, \overrightarrow{j})\}$
Endif

If $type(j)$ =LOOP $\wedge \ i < j$ **Then**
$\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{k}, \overleftarrow{i})\} : \ (k,j) \in E$
ElseIf $type(i)$ =LOOP $\wedge \ label(e) \neq 1$ **Then**
$\quad \overleftarrow{V} := \overleftarrow{V} \cup \{k'\} : \ k' \ni \boxed{\textit{rBound(i):=S.pop()}}$
$\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j}, k'), (k', \overleftarrow{k})\} : \ (k,i) \in E$
Else
$\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j}, \overleftarrow{i})\}$
Endif

The construction of vertices in $\overrightarrow{G}$ and $\overleftarrow{G}$ is described in Sec. 4.2. The forward sweep needs to be augmented by statements for initializing [$ctr(i) := 0$], incrementing [$ctr(i) := ctr(i) + 1$], and storing the final value [$S.push(ctr(i))$] of the counter associated with a LOOP vertex i. The initialization is done right before the LOOP vertex (lines 1..3), and the final value is stored right after the execution of the loop (lines 7..9). The incrementation of $ctr(i)$ is performed right before the ENDLOOP vertex that corresponds to i (lines 4..6). The transformation of any remaining edges is straightforward (line 11).

If the source i of an edge $(i,j) \in G$ is a LOOP vertex and i is not leading into the loop body, then the value of $ctr(i)$ needs to be restored right before the adjoint loop as $rBound(i)$ (lines 15..17). If j is a LOOP vertex that follows i in the topological order, then the adjoint edge leads into the corresponding adjoint FORLOOP (lines 13..14). Otherwise, the adjoint of an edge is obtained by switching its direction in $\overleftarrow{G}$ (line 19).

Note that this algorithm does not explicitly refer to any index or condition that could be part of the original LOOP construct in G. Differing from the example used in Sec. 2, any dependencies on such loop indices have to be resolved by other means, for example, as introduced in [7].

4.5. Nesting Branches and Loops

Obviously, Algorithms 3 and 4 do not cover the general case where loops and branches can be nested arbitrarily. The following issues need to be considered.

Vertices need to be visited in the topological order defined by Algorithm 1. In particular, this implies that loop bodies are processed before the statements following loop.

A stack is required for storing the loop counter symbols as they are required for inserting the correct incrementation statement right before the matching ENDLOOP vertex in $\overrightarrow{G}$.

Algorithm 5 (Adjoint Structured Control Flow Graphs) *This algorithm combines Algorithms 3 and 4. The starting conditions are as follows: S is empty, $ctr(i) = 0$ for all LOOP vertices i, and $mkr(j)$ is used to mark the branches merged by an ENDBRANCH vertex j.*

$\forall e \equiv (i,j) \in E$:

1 **If** $type(i)$ =LOOP $\wedge\ label(e) \neq 1$ **Then**
2 $\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} :\ k \ni \boxed{S.push(ctr(i))}$
3 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k)\}$
4 $\quad$ **If** $type(j)$ =LOOP **Then**
5 $\quad\quad \overrightarrow{V} := \overrightarrow{V} \cup \{l\} :\ l \ni \boxed{ctr(j){:=}0}$
6 $\quad\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(k,l),(l,\overrightarrow{j})\}$
7 $\quad$ **ElseIf** $type(j)$ =ENDLOOP **Then**
8 $\quad\quad \overrightarrow{V} := \overrightarrow{V} \cup \{l\} :\ l \ni \boxed{ctr(j){:=}ctr(j)+1} \ \wedge\ (j,j') \in E$
9 $\quad\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(k,l),(l,\overrightarrow{j})\}$
10 $\quad$ **ElseIf** $type(j)$ =ENDBRANCH **Then**
11 $\quad\quad \overrightarrow{V} := \overrightarrow{V} \cup \{l\} :\ l \ni \boxed{S.push(mkr(j))}$
12 $\quad\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(k,l),(l,\overrightarrow{j})\}$
13 $\quad$ **Else**
14 $\quad\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(k,\overrightarrow{j})\}$
15 $\quad$ **Endif**
16 **ElseIf** $type(j)$ =LOOP $\wedge\ i < j$ **Then**
17 $\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} :\ k \ni \boxed{ctr(j){:=}0}$
18 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k),(k,\overrightarrow{j})\}$
19 **ElseIf** $type(j)$ =ENDLOOP **Then**
20 $\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} :\ k \ni \boxed{ctr(j){:=}ctr(j)+1}$
21 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k),(k,\overrightarrow{j})\}$
22 **ElseIf** $type(j)$ =ENDBRANCH **Then**
23 $\quad \overrightarrow{V} := \overrightarrow{V} \cup \{k\} :\ k \ni \boxed{S.push(mkr(j))}$
24 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i}, k),(k,\overrightarrow{j})\}$
25 **Else**
26 $\quad \overrightarrow{E} := \overrightarrow{E} \cup \{(\overrightarrow{i},\overrightarrow{j})\}$
27 **Endif**

28 **If** $type(i)$ =LOOP $\wedge\ label(e) \neq 1$ **Then**
29 $\quad \overleftarrow{V} := \overleftarrow{V} \cup \{k'\} :\ k' \ni \boxed{rBound(i){:=}S.pop()}$
30 $\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(k',\overleftarrow{k})\} :\ (k,i) \in E$
31 $\quad$ **If** $type(j)$ =LOOP **Then**
32 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{l},k')\} :\ (l,j) \in E$
33 $\quad$ **Else**
34 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j},k')\}$
35 $\quad$ **Endif**
36 **ElseIf** $type(i)$ =ENDBRANCH **Then**
37 $\quad \overleftarrow{V} := \overleftarrow{V} \cup \{k'\} :\ k' \ni \boxed{branchId{:=}S.pop()}$
38 $\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(k',\overleftarrow{i})\}$
39 $\quad$ **If** $type(j)$ =LOOP **Then**
40 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{k},k')\} :\ (k,j) \in E$
41 $\quad$ **Else**
42 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j},k')\}$
43 $\quad$ **Endif**
44 **ElseIf** $type(j)$ =LOOP $\wedge\ i < j$ **Then**
45 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{k},\overleftarrow{i})\} :\ (k,j) \in E$
46 **Else**
47 $\quad\quad \overleftarrow{E} := \overleftarrow{E} \cup \{(\overleftarrow{j},\overleftarrow{i})\}$
48 **Endif**

The decision about how to augment a given edge in $\overrightarrow{G}$ is based either on its source (if $\vec{i}$ needs to be succeeded by a new basic block to store the control flow) or on its target (if $\vec{j}$ needs to be preceded by such a new basic block). The first situation occurs if i is a LOOP and (i, j) does not lead into the loop body (lines 1..3). Then $ctr(i)$ needs to be stored. There are three cases in which a new basic block needs to be inserted right before $\vec{j}$. They are handled separately (lines 16..23). Special care must be taken if two new successive basic blocks need to be inserted between $\vec{i}$ and $\vec{j}$ (lines 4..12).

The construction of $\overleftarrow{E}$ is based on the fact that each $e \equiv (i, j) \in E$ has a matching $\overleftarrow{e} \in \overleftarrow{E}$. There are two cases where new basic blocks need to be inserted to restore the control flow. If i is a LOOP vertex and e does not lead into the loop body, then $ctr(i)$ needs to be restored prior to the corresponding FORLOOP vertex in $\overleftarrow{G}$ (lines 28..30). Similarly, the adjoint of an ENDBRANCH vertex must be preceded by a new basic block to restore $mkr(i)$ (lines 36..38). A special treatment is required j is a LOOP vertex and i is not the matching ENDLOOP vertex. The source of the adjoint edge must then be the FORLOOP vertex of the adjoint loop (lines 31..32, 39..40, and 44..45). All other cases are covered by the simple reversal of the edge (line 47).

The result of applying Alg. 5 to the CFG in Fig. 2 (a) is shown in Fig. 2 (b) and (c). In $\overleftarrow{G}$ the labels of the edges emanating from LOOP and BRANCH vertices have been propagated to the matching ENDLOOP and ENDBRANCH vertices. Again, adjoint vertices corresponding to vertices in G carry the respective negative index. Vertices whose index is greater than 12 (the index of the EXIT vertex in G) contain statements for storing and restoring the flow of control.

5. Conclusions

The strategy presented in this paper is not the only possible method to reverse the control flow of a subroutine. For example, instead of storing independent identifiers for branches, one could store the value of the condition. A corresponding approach can be taken for multi-way branches and loops. In doing so, however, one introduces additional dependencies between the original code and the adjoint code, for example, the requirement for additional canonicalization. The present approach allows the algorithms to be formulated purely in terms of the CFG.

We realize that the repeated insertion of new basic blocks for storing and retrieving the flow of control is not necessary. The corresponding statements could be merged with already existing basic blocks. In any case, the final unparsed codes are equivalent.

The algorithms introduced in this paper have been implemented in the OpenAD framework. There the adjoining of the basic blocks is decoupled from the reversal of the CFGs. Such a decoupling favors the insertion of new basic blocks for storage and retrieval of control flow information, as suggested in this paper. However, the examples in this paper cover only the most simple cases. The adjoining of more complex codes requires more advanced analyzes to guarantee semantical correctness. We mentioned the issue of variable address computation depending on loop variables in Sec. 4.4. Another issue is the adjoining of unstructured CFGs. The most general reversal algorithm assigns unique identifiers to all basic blocks and stores them at execution during the augmented forward sweep. The adjoint code restores the identifiers in reverse order and executes the corresponding adjoint basic blocks. Exploitation of partial structuredness is the subject of future work.

References

[1] A. Albrecht, P. Gottschling, and U. Naumann. Markowitz-type heuristics for computing Jacobian matrices efficiently. In *ICCS 2003*, volume 2658 of *LNCS*, pages 575 –584, Berlin, 2003. Springer.

[2] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996.

[3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[4] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.

[5] A. Griewank and G. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, 1991.

[6] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markowitz rule. In A. Griewank and G. Corliss, editors, *[5]*, pages 126–135. SIAM, Philadelphia, 1991.

[7] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint MCS-P936-0202, Argonne National Laboratory, 2002.

[8] J. Marshall, C. Hill, L. Perelman, and A. Adcroft. Hydrostatic, quasi-hydrostatic and nonhydrostatic ocean modeling. *J. Geophysical Research*, 102, C3:5,733–5,752, 1997.

[9] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 2003. To appear.