OpenAD/F: User Manual

J. Utke
U. Naumann

Draft compile date November 2, 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The basic principles of automatic differentiation (AD) (see also Sec. **??**) have been known for several decades [35], but only during the past 15 years have the tools implementing AD found significant use in optimization, data assimilation, and other applications in need of efficient and accurate derivative information. As a consequence of the wider use of AD, various tools have been developed that address specific application requirements or programming languages. The AD community's website `www.autodiff.org` provides a comprehensive overview of the tools that are currently available. One can categorize two user groups of AD tools. On one side are casual users with small-scale problems applying AD mostly in a black-box fashion and demanding minimal user intervention. This category also includes users of AD tools in computational frameworks such as NEOS [25]. On the other side are experienced AD users aiming for highly efficient derivative computations. Their need for efficiency is dictated by the computational complexity of models that easily reaches the limits of current supercomputers. In turn this group is willing to accept some limitation in the support of language features.

1.1 Motivation

One of the most demanding applications of AD is the computation of gradients for data assimilation on large-scale models used in oceanography and climate research. This application clearly falls in the category of experienced users. An evaluation of the available tools revealed some shortcomings from the perspectives of the tool users as well as the tool developers and was the rationale for designing a new tool with particular emphasis on

- flexibility,
- the use of open source components, and
- modularity.

From the AD tool *users* point of view there is a substantial need for flexibility of AD tools. The most demanding numerical models operate at the limit of the computing capacity of state-of-the-art facilities. Usually the model code itself is specifically adapted to fit certain hardware characteristics. Therefore AD tool code generation ideally should be adaptable in a similar fashion. Since some of these adaptations may be too specific for a general-purpose tool, the AD tool should offer *flexibility* at various levels of the transformation – from simple textual preprocessing of the code down to the changes to the generic code transformation engine. This is the rationale for developing an *open source* tool where all components are accessible and may be freely modified to suit specific needs. A *modular* tool design with clearly defined interfaces supports such user interventions. Since this design instigates a staged transformation, each transformation stage presents a opportunity to check and modify the results.

From the AD tool *developers* point of view many AD tools share the same basic algorithms, but there is a steep hurdle to establish a transformation environment consisting of a front-end that turns the textual program into a compilerlike internal representation, an engine that allows the transformations of this internal representation, and an unparser that turns the transformed internal representation back into source code. A *modular, open-source* tool facilitating the integration of new transformations into an existing environment allows for a quick implementation and testing of algorithmic ideas. Furthermore, a modular design permits the reuse of transformation algorithms across multiple target languages, provided the parsing front-ends can translate to and from the common internal representation.

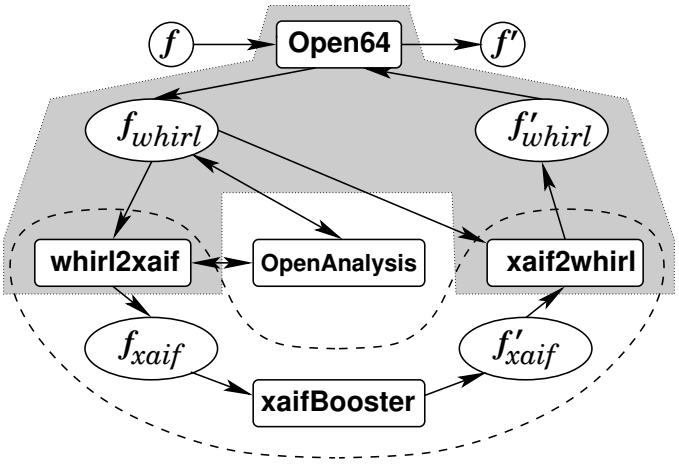


Figure 1.1: OpenAD/F components and pipeline

These considerations motivated the Adjoint Compiler Technology & Standards [4] project, a research and development collaboration of MIT, Argonne National Laboratory, The University of Chicago, and Rice University. OpenAD/F is one of its major results.

1.2 Overview

OpenAD/F[27] is the Fortran incarnation of the AD framework OpenAD. The C/C++ oriented tool ADIC v2.0 [3] is based on the same framework but is not subject of this article. OpenAD/F has a modular design. The collaboration of the OpenAD/F components is illustrated in Fig. 1.1. Our input is some numerical model given as a Fortran program $\boldsymbol{f}$. The Open64[26] front-end performs a lexical, syntactic, and semantic analysis and produces an intermediate representation of $\boldsymbol{f}$, here denoted by $\boldsymbol{f}_{whirl}$, in the so-called whirl format. OpenAnalysis is used to build call and control flow graphs and perform code analyses such as alias, activity, side-effect analysis. This information is used by whirl2xaif to construct a representation of the numerical core of $\boldsymbol{f}$ in XAIF format shown as $\boldsymbol{f}_{xaif}$. A differentiated version of $\boldsymbol{f}_{xaif}$ is derived by an algorithm that is implemented in xaifBooster and is again represented XAIF as $\boldsymbol{f}'_{xaif}$. The information in $\boldsymbol{f}'_{xaif}$ and the original $\boldsymbol{f}_{whirl}$ are used by xaif2whirl to construct a whirl representation $\boldsymbol{f}'_{whirl}$ of the differentiated code. The unparser of Open64 transforms $\boldsymbol{f}'_{whirl}$ into Fortran90, thus completing the semantic transformation of a program $\boldsymbol{f}$ into a differentiated program $\boldsymbol{f}'$. The gray shaded area encloses the language specific front-end that can potentially be replaced by front-ends for languages other than Fortran. For instance, the new version of ADIC [19] couples a C/C++ front-end based on the EDG parser [12] and uses ROSE in combination with SAGE 3 [30] as internal representation in combination with language independent components of OpenAD.

In Sec. 2 we discuss the basic concepts of AD as relevant for the description of OpenAD, Sec. 3 discusses the components that make up OpenAD/F, and Sec. 4 details the usage of the tool. Two applications further illustrate the tool usage in Sec. 5 and we conclude with a section on future developments.

Chapter 2

AD Concepts

In this section we present the terminology and basic concepts that we will refer to throughout this paper. A detailed introduction to AD can be found in [15]. The interested reader should also consider the proceedings of AD conferences [11, 7, 10, 9].

We present the concepts and resulting transformations with respect to the input source code in a bottom up fashion. We first consider elemental numerical operations, then their control flow context within a subroutine and finally the entire program consisting of several subroutines in a call graph.

We view a given numerical model as a vector valued function $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : I\!R^n \mapsto I\!R^m$ that is implemented as a computer program in a language such as Fortran, C, or C++ and the objective is to compute products of Jacobians with see matrices $\boldsymbol{S}$.

$$\boldsymbol{JS} \quad \text{and} \quad \boldsymbol{J}^T\boldsymbol{S} \tag{2.1}$$

2.1 Computational Graphs

Without loss of generality we can simply assume that an evaluation of $\boldsymbol{f}(\boldsymbol{x})$ for a specific value of $\boldsymbol{x}$ can be represented by a sequence of elemental operations $v_j = \phi_j(\ldots, v_i, \ldots)$. The v_i represent the vertices $\in V$ in the corresponding computational graph $G = (V, E)$. The edges $(i, j) \in E$ in this graph are the direct dependencies $v_i \prec v_j$ implied by the elemental $v_j = \phi_j(\ldots, v_i, \ldots)$. The elemental operations ϕ are differentiable on open subdomains. Each edge $(i, j) \in E$ has an attached local partial derivative $c_{ji} = \frac{\partial v_j}{\partial v_i}$. The central principle of AD is the application of the chain rule to the elemental ϕ, that is multiplications and additions of the c_{ji}.

Like most of the AD literature we follow a specific numbering scheme for the vertices v_i. We presume q intermediate values $v_j = \phi_j(\ldots, v_i, \ldots), v_j \in Z$ for $j = 1, \ldots, q+m$ and $h, i = 1-n, \ldots, q,\ j > h, i$. The n *independent* variables $x_1, \ldots, x_n$ correspond to $v_{1-n}, \ldots, v_0, v_i \in X$. We consider the computation of derivatives of the *dependent* variables $y_1, \ldots, y_m$ represented by m variables $v_{q+1}, \ldots, v_{q+m}, v_j \in Y$ with respect to the independents. The dependency $v_i < v_j$ implies $i < j$. The *forward mode* of AD propagates directional derivatives as

$$\dot{v}_j = \sum_i \frac{\partial \phi_j}{\partial v_i}\dot{v}_i \quad \text{for} \quad j = 1, \ldots, q+m. \tag{2.2}$$

In *reverse mode* we compute adjoints of the arguments of the ϕ_j as a function of local partial derivatives and the adjoint of the variable on the left-hand side

$$\overline{v}_i = \sum_j \frac{\partial \phi_j}{\partial v_i}\overline{v}_j \quad \text{for} \quad j = 1, \ldots, q+m. \tag{2.3}$$

In practice, the sum in (2.3) is often split into individual increments associated with each statement in which v_i occurs as an argument $\overline{v}_i = \overline{v}_i + \overline{v}_j * \frac{\partial \phi_j}{\partial v_i}$.

Equations (2.2) and (2.3) can be used to accumulate the (local) Jacobian $\boldsymbol{J}(G)$ of G, see also Sec. 2.2.

In a source transformation context we want to generate code for all $\boldsymbol{f}(\boldsymbol{x})$ in the domain and because the above construction disregards control flow it is impractical here. Instead we simply consider the statements contained in a basicblock as a section of code below the granularity of control flow and construct our computational (sub) graph for a basicblock.

2.2 Elimination Methods

Let ***f*** represent a single basicblock that is subject to preaccumulation. For notational simplicity and without loss of generality we assume that the dependent variables are mutually independent. This situation can always be reached by introducing auxiliary assignments. Consider the small example in Fig. 2.1. Reformulating the example in terms

```
t1   = x(1) + x(2)
t2   = t1 + sin(x(2)
y(1) = cos(t1 * t2)
y(2) = -sqrt(t2)
```

Figure 2.1: Example of code contained in a basicblock

of results of elemental operations ϕ assigned to unique intermediate variables v we have

$$\begin{aligned} &v_1 = v_{-1} + v_0;\ v_2 = \sin(v_0);\ v_3 = v_1 + v_2;\ v_4 = v_1 * v_3; \\ &v_5 = \sqrt{v_3};\ v_6 = \cos(v_4);\ v_7 = -v_5 \quad . \end{aligned} \tag{2.4}$$

In the tool this modified representation is created as part of the linearization transformation, see Sec. 3.1.3.2. In Fig. 2.2 (a) we show the computational graph G for this representation. The edges $(i,j) \in E$ are labeled with partial derivatives c_{ji}, for instance, in the example we have $c_{64} = -\sin(v_4)$. In the tool, this graph is generated as part of the algorithm described in Sec. 3.1.3.3. Jacobian preaccumulation can be interpreted as eliminations in G. The graph-based elimination steps are categorized in vertex, edge, and face eliminations. In G a vertex $j \in V$ is eliminated by connecting its predecessors with its successors [16]. An edge (i,k) with $i \prec j$ and $j \prec k$ is labeled with $c_{ki} + c_{kj} \cdot c_{ji}$ if it existed before the elimination of j. We say that *absorption* takes place. Otherwise, (i,k) is generated as *fill-in* and labeled with $c_{kj} \cdot c_{ji}$ The vertex j is removed from G together with all incident edges. Fig. 2.2 (b) shows the result of eliminating vertex 3 from the graph in Fig. 2.2 (a).

An edge (i,j) is *front eliminated* by connecting i with all successors of j, followed by removing (i,j) [20]. The corresponding structural modifications of the c-graph in Fig. 2.2 (a) are shown in Fig. 2.2 (c) for front elimination of $(1,3)$. The new edge labels are given as well. Edge-front elimination eventually leads to intermediate vertices in G becoming *isolated*; that is, these vertices no longer have predecessors. Isolated vertices are simply removed from G together with all incident edges.

Back elimination of an edge $(i,j) \in E$ results in connecting all predecessors of i with j [20]. The edge (i,j) itself is removed from G. The back elimination of $(3,4)$ from the graph in Fig. 2.2 (a) is illustrated in Fig. 2.2 (d). Again, vertices can become isolated as a result of edge-back elimination because they no longer have successors. Such vertices are removed from G.

Numerically the elimination is the application of the chain rule, that is, a sequence of *fused-multiply-add* (fma) operations

$$c_{ki} = c_{ji} * c_{kj} \ \ (+c_{ki}) \tag{2.5}$$

where the additions in parenthesis take place only in the case of absorption or otherwise fill-in is created as described above.

Aside from special cases a single vertex or edge elimination will result in more than one fma. *Face elimination* was introduced as the elimination operation with the finest granularity of exactly one multiplication[1] per elimination step.

Vertex and edge elimination steps have an interpretation in terms of vertices and edges of G, whereas face elimination is performed on the corresponding directed line graph $\mathcal{G}$. Following [21], we define the directed line graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ corresponding to $G = (V, E)$ as follows:

$$\mathcal{V} = \left\{ \boxed{i,j} : (i,j) \in E \right\} \cup \left\{ \boxed{\oplus, j} : v_j \in X \right\} \cup \left\{ \boxed{i, \ominus} : v_i \in Y \right\}$$

and

$$\begin{aligned} \mathcal{E} = &\left\{ \left(\boxed{i,j}, \boxed{j,k} \right) : (i,j),(j,k) \in E \right\} \\ &\cup \left\{ \left(\boxed{\oplus, j}, \boxed{j,k} \right) : v_j \in X \wedge (j,k) \in E \right\} \\ &\cup \left\{ \left(\boxed{i,j}, \boxed{j, \ominus} \right) : v_j \in Y \wedge (i,j) \in E \right\} \quad . \end{aligned}$$

[1] Additions are not necessarily directly coupled.

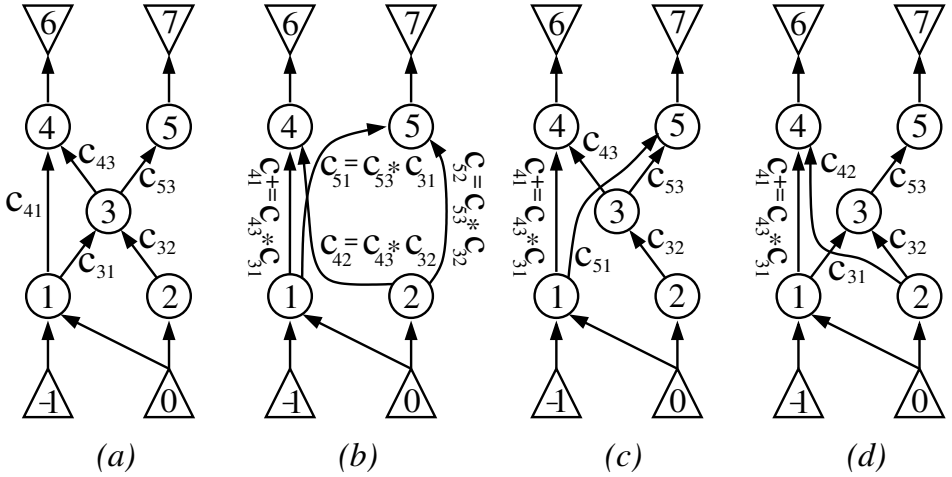


Figure 2.2: (a) Computational graph G for (2.4), (b) eliminate vertex 3 from G, (c) front eliminate edge $(1,3)$ from G, (d) back eliminate edge $(3,4)$ from G

That is, we add a source vertex $\oplus$ and a sink vertex $\ominus$ to G connecting all independents to $\oplus$ and all dependents to $\ominus$. $\mathcal{G}$ has a vertex $v \in \mathcal{V}$ for each edge in the extended G, and $\mathcal{G}$ has an edge $e \in \mathcal{E}$ for each pair of adjacent edges in G. Fig. 2.3 gives an example of constructing the directed line graph in (b) from the graph in (a) which is the graph from Fig. 2.2(a) extended by the source and sink vertex. All intermediate vertices $\boxed{i,j} \in \mathcal{V}$ inherit the labels c_{ji}. In order to formalize face elimination, it is advantageous to move away from the double-index notation and use one that is based on a topological enumeration of the edges in G. Hence, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ becomes a DAG with $\mathcal{V} \subset I\!N$ and $\mathcal{E} \subset I\!N \times I\!N$ and certain special properties. The set of all predecessors of $j \in \mathcal{V}$ is denoted as P_j. Similarly, S_j denotes the set of its successors in $\mathcal{G}$. A vertex $j \in \mathcal{V}$ is called *isolated* if either $P_j = \emptyset$ or $S_j = \emptyset$. Face elimination is defined in [21] between two incident intermediate vertices i and j in $\mathcal{G}$ as follows:

1. If there exists a vertex $k \in \mathcal{V}$ such that $P_k = P_i$ and $S_k = S_j$, then set $c_k = c_k + c_j c_i$ *(absorption)*; else $\mathcal{V} = \mathcal{V} \cup \{k'\}$ with a new vertex k' such that $P_{k'} = P_i$ and $S_{k'} = S_j$ *(fill-in)* and labeled with $c_{k'} = c_j c_i$.

2. Remove (i,j) from $\mathcal{E}$.

3. Remove $i \in \mathcal{V}$ if it is isolated. Otherwise, if there exists a vertex $i' \in \mathcal{V}$ such that $P_{i'} = P_i$ and $S_{i'} = S_i$, then

   - set $c_i = c_i + c_{i'}$ *(merge)*;
   - remove i'.

4. Repeat Step 3 for $j \in \mathcal{V}$.

In Fig. 2.3 (c) we show the elimination of $(i,j) \in \mathcal{E}$, where $i = \boxed{1,3}$ and $j = \boxed{3,4}$.

A complete face elimination sequence σ_f yields a tripartite directed line graph $\sigma_f(\mathcal{G})$ that can be transformed back into the bipartite graph representing the Jacobian $\boldsymbol{f'}$. We note that any G can be transformed into the corresponding $\mathcal{G}$ but that a back transformation generally is not possible once face elimination steps have been applied. Therefore, face eliminations can generally not precede vertex and edge eliminations. In OpenAD these eliminations are implemented in the algorithms described in Sec. 3.1.3.4 and Sec. 3.1.3.5.

In a source transformation context of OpenAD/F the operations (2.5) are expressed as actual code, the Jacobian accumulation code. For our example code from Fig. 2.1 the code computing the local partials in conjunction with the function value is shown in Fig. 2.4. [2] In OpenAD/F the operations in Fig. 2.4 are generated by the transformation algorithm discussed in Sec. 3.1.3.2. The operations induced by the eliminations on the graph can be expressed in terms of the auxiliary variables c_{ji}. For our example, a forward vertex elimination in the order (1,2,3,4,5) in G (Fig. 2.2), leads to the following Jacobian accumulation code. In the tool the operations shown in Fig. 2.5 are generated by the transformation algorithm discussed in Sec. 3.1.3.3.

[2] For better readability we write the indices of the c_{ji} with commas.

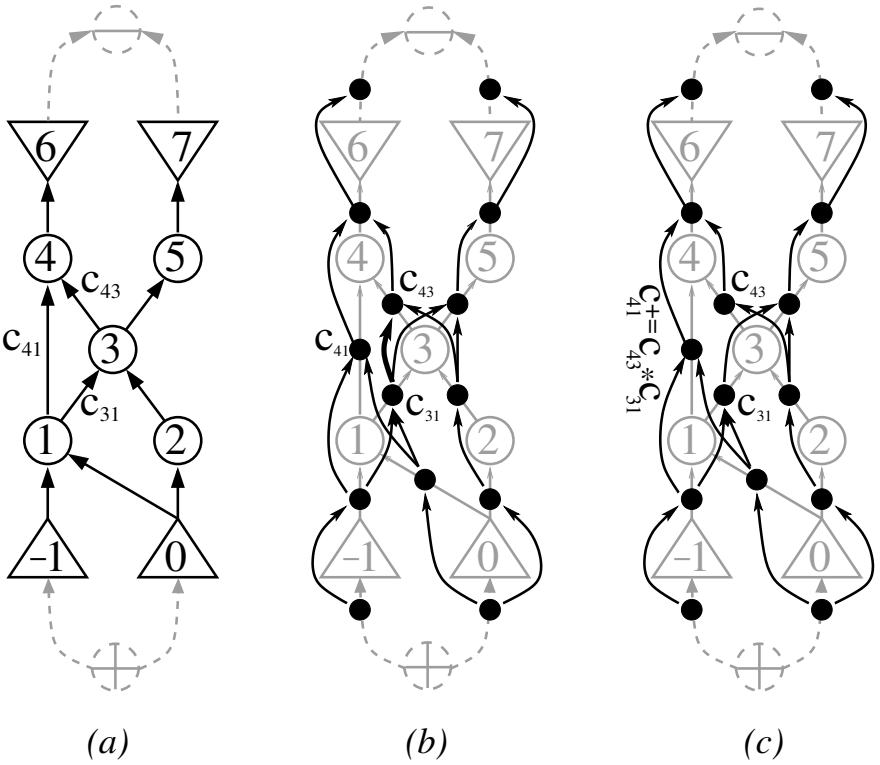


Figure 2.3: (a) G extended, (b) $\mathcal{G}$ overlaid, (c) face elimination

$v_1 = v_{-1} + v_0;$	$c_{1,-1} = 1;$	$c_{1,0} = 1;$
$v_2 = \sin(v_0);$	$c_{2,0} = \cos(v_0);$	
$v_3 = v_1 + v_2;$	$c_{3,1} = 1;$	$c_{3,2} = 1;$
$v_4 = v_1 * v_3;$	$c_{4,1} = v_3;$	$c_{4,3} = v_1;$
$v_5 = \sqrt{v_3};$	$c_{5,3} = (2\sqrt{v_3})^{-1};$	
$v_6 = \cos(v_4);$	$c_{6,4} = -\sin(v_4);$	
$v_7 = -v_5;$	$c_{7,5} = -1;$	

Figure 2.4: Pseudo code for (2.4) and the computation of the c_{ji}

1: $c_{3,-1} = c_{3,1} * c_{1,-1}$; $c_{3,0} = c_{3,1} * c_{1,0}$; $c_{4,-1} = c_{4,1} * c_{1,-1}$;
$c_{4,0} = c_{4,1} * c_{1,0}$;
2: $c_{3,0} = c_{3,2} * c_{2,0} + c_{3,0}$;
3: $c_{4,-1} = c_{4,3} * c_{3,-1} + c_{4,-1}$; $c_{4,0} = c_{4,3} * c_{3,0} + c_{4,0}$; $c_{5,-1} = c_{5,3} * c_{3,-1}$;
$c_{5,0} = c_{5,3} * c_{3,0}$;
4: $c_{6,-1} = c_{6,4} * c_{4,-1}$; $c_{6,0} = c_{6,4} * c_{4,0}$;
5: $c_{7,-1} = c_{7,5} * c_{5,-1}$; $c_{7,0} = c_{7,5} * c_{5,0}$;

Figure 2.5: Pseudo code for vertex eliminations for (2.4)

```
00    y(k) = sin(x(1)*x(2))
01    k = k+1
02    if (mod(k,2) .eq.  1) then
03       y(k) = 2*y(k-1)
04    else
05       do i=1,k
06          t1 = x(1)+x(2)
07          t2 = t1+sin(x(1))
08          x(1) = cos(t1*t2)
09          x(2) = -sqrt(t2)
10       end do
11    end if
12    y(k) = y(k)+x(1)*x(2)
```

Figure 2.6: Toy example code with control flow

2.3 Control Flow Reversal

Because the code for a $\boldsymbol{f}$ generally contains control flow constructs there is no single computational graph G that represents the computation of $\boldsymbol{f}$ for all possible values of $\boldsymbol{x}$. We explained in Sec. 2.1 that OpenAD/F considers subgraphs constructed from the contents of a basicblock. In the example shown in Fig. 2.6 we put the basicblock code shown in Fig. 2.1 into a control flow context, see lines 06–09. The control flow graph (CFG) [5] resulting from the above code is depicted in Fig. 2.7(a). The assignment statements are contained in the basicblocks B(2,4,6,9). For instance, the statements from Fig. 2.1 now in lines 06–09 form the loop body, basicblock B(6). As B(6) is executed `k` times it may be worth putting additional effort into the optimization of the derivative code generated for B(6) by optimizing the elimination sequence as illustrated in Sec. 2.2. For B(6) the corresponding computational graph G see Fig. 2.2(a).

For a sequence of l basicblocks that are part of a path through the CFG for a particular value of $\boldsymbol{x}$ the equations (2.2) and (2.3) can be generalized as follows:

$$\dot{\boldsymbol{y}}_j = \boldsymbol{J}_j \dot{\boldsymbol{x}}_j \quad \text{for} \quad j = 1, \ldots, l \tag{2.6}$$

and

$$\bar{\boldsymbol{x}}_j = \boldsymbol{J}_j^T \bar{\boldsymbol{y}}_j \quad \text{for} \quad j = l, \ldots, 1 \quad , \tag{2.7}$$

where $\boldsymbol{x}_j = (x_i^j \in V : i = 1, \ldots, n_j)$ and $\boldsymbol{y}_j = (y_i^j \in V : i = 1, \ldots, m_j)$ are the inputs and outputs of the basicblocks respectively. In *forward mode* a sequence of products of the local Jacobians $\boldsymbol{J}_j$ with the directions $\dot{x}_j$ are propagated forward in the direction of the flow of control, for instance simultaneously to the computation of $\boldsymbol{f}$. In our example basicblock B(6) is the third basicblock ($j = 3$) and we have $\boldsymbol{x}_3 = \boldsymbol{y}_j = $ (`x(1)`,`x(2)`) and consequently have the operations for the Jacobian vector product shown in Fig. 2.8. Note that the code overwrites `x(1)` and `x(2)` and therefore we have to preserve the original derivatives in temporaries t_1 and t_2.

In *reverse mode* products of the transposed Jacobians $\boldsymbol{J}_j^T$ with adjoint vectors $\overline{\boldsymbol{y}}_j$ are propagated reverse to the direction of the flow of control. The $\boldsymbol{J}_j^T$ can be computed by augmenting the original code with linearization and Jacobian accumulation statements, see Sec. 2.2. The preaccumulated $\boldsymbol{J}_j^T$ are stored during the forward execution which is commonly called the *tape*, see Fig. 2.9(a) for an example. They are retrieved from the tape for computing (2.7) during the reverse execution, see Fig. 2.9(b) for an example. It is always possible to organize the store and retrieve such that the tape can be implemented as a stack.

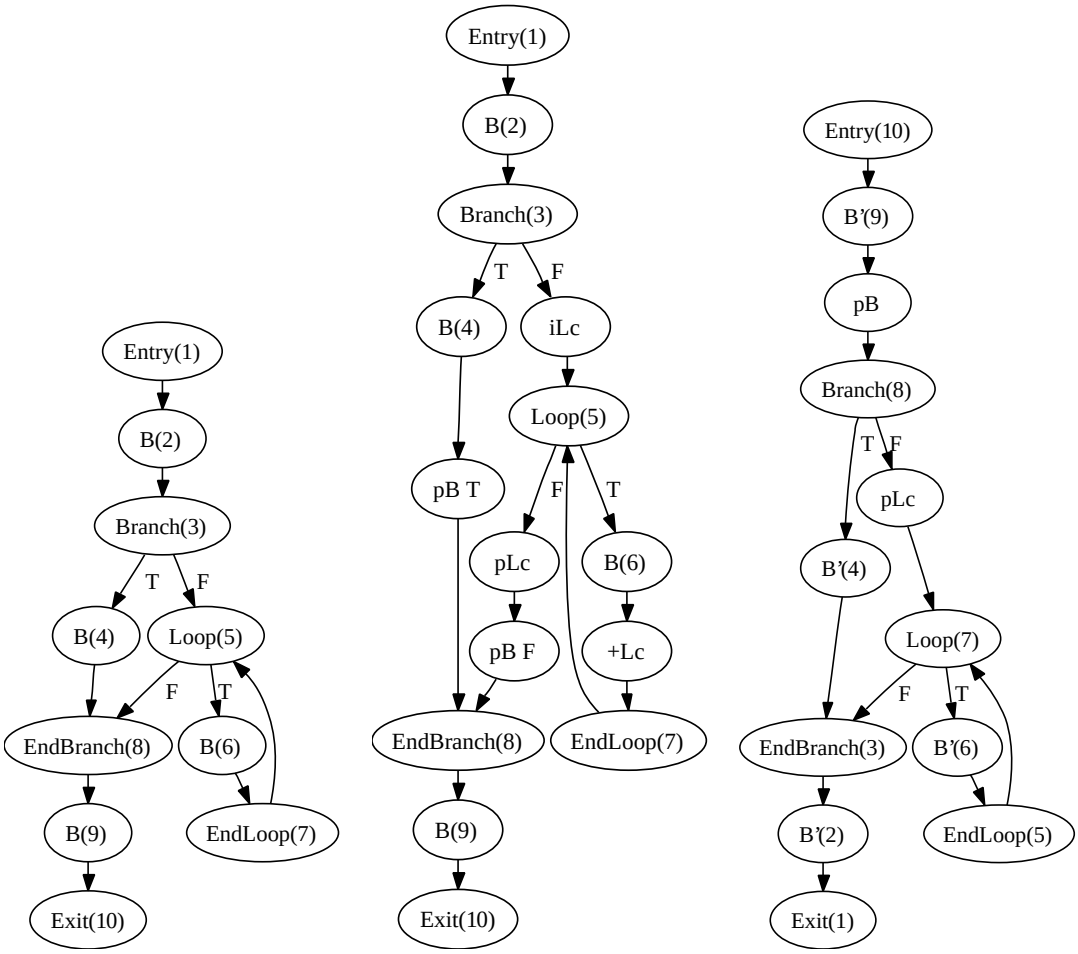


Figure 2.7: CFG of Fig. 2.6 (a) original, (b) trace generating, (c) reversed

In order to find the corresponding path to the reversed control flow graph we also have to generate a trace which is done with an augmented CFG, for our toy example see Fig. 2.7(b). This augmented CFG keeps track of which branch was taken and counts how often a loop was executed. This information is pushed on a stack and popped from that stack during the reverse sweep see also [24]. Because the control flow trace adheres to the stack model it often is also considered part of the tape. In the example in Fig. 2.7(b) the extra basicblocks pBT and pBF push a boolean (T or F) onto the stack depending on the branch. In iLc we initialize a loop counter, increment the loop counter in +Lc, and push the final count in pLc.

Fig. 2.7(c) shows the reversed CFG for our toy example. The parenthesized numbers in the node labels align the node transformation to Fig. 2.7(a). The exit node becomes the entry, loop becomes endloop, branch becomes endbranch, and vice versa. Each basicblock B is replaced with its reversed version B'. Finally, to find the proper path through this reversed CFG we need to retrieve the information recorded in Fig. 2.7(b). The extra nodes pB and pLc pop the branch information and the loop counter respectively. We enter the branch and execute the loop as indicated by the recorded information. The process of the control flow reversal is described in detail in [24].

2.4 Call Graph Reversal

Generally, the computer program induces a *call graph* (CG) [5] whose vertices are subroutines and whose edges represent calls potentially made during the computation of $\boldsymbol{y}$ for all values of $\boldsymbol{x}$ in the domain of $\boldsymbol{f}$.

For a large number of problems it is possible to statically predetermine either *split* or *joint* reversal [15] for any subroutine in the call graph . These concepts are easier understood with the help of the dynamic call tree, see also [23]. where each vertex represents an actual invocation of a subroutine for a given execution of the program, see Fig. 2.10 and Table 2.1 for an explanation of the symbols. The order of calls is implied by following the edges in

$$
\begin{aligned}
t_1 &= \dot{\mathtt{x}}(1);\\
t_2 &= \dot{\mathtt{x}}(2);\\
\dot{\mathtt{x}}(1) &= c_{6,-1} * t_1;\\
\dot{\mathtt{x}}(1) &= \dot{\mathtt{x}}(1) + c_{6,0} * t_2;\\
\dot{\mathtt{x}}(2) &= c_{7,-1} * t_1;\\
\dot{\mathtt{x}}(2) &= \dot{\mathtt{x}}(2) + c_{7,0} * t_2;
\end{aligned}
$$

Figure 2.8: Pseudo code for $\boldsymbol{J}_3\dot{\boldsymbol{x}}_3$ for the loop body in Fig. 2.6

$$
\begin{aligned}
&\mathtt{push}(c_{6,-1});\\
&\mathtt{push}(c_{6,0});\\
&\mathtt{push}(c_{7,-1});\\
&\mathtt{push}(c_{7,0});
\end{aligned}
$$

(a)

$$
\begin{aligned}
t_2 &= \mathtt{pop}() * \bar{\mathtt{x}}(2);\\
t_1 &= \mathtt{pop}() * \bar{\mathtt{x}}(2);\\
t_2 &= t_2 + \mathtt{pop}() * \bar{\mathtt{x}}(1);\\
t_1 &= t_1 + \mathtt{pop}() * \bar{\mathtt{x}}(1);\\
\bar{\mathtt{x}}(2) &= t_2;\\
\bar{\mathtt{x}}(1) &= t_1;
\end{aligned}
$$

(a)

Figure 2.9: Pseudo code for writing the tape (a) and consuming the tape for $\boldsymbol{J}_3^T\bar{\boldsymbol{y}}_3$ (b) for the loop body in Fig. 2.6

left to right order. Using split reversal for all subroutines in the program means that first the tape for the entire program is written. Then we follow with the reverse steps that read the tape, see Fig. 2.11.

Using joint reversal for all subroutines in a program means that the tape, see Sec. 2.3 for a each subroutine invocation is written immediately before the reverse execution for that invocation. In our example this implies that we have to generate a tape for $\mathtt{C}^2$ while the caller $\mathtt{B}^2$ is being reversed, i.e. this is not the proper context to simply reexecute $\mathtt{C}^2$. We can either reexecute the entire program up to the $\mathtt{C}^2$ call and then start taping, or (preferably) we store the arguments while running forward and restore them before starting the taping. The ensuing dynamic call tree for our example is shown in Fig. 2.12. For many applications neither an all split nor all joint reversal is efficient. Often a mix of split and joint reversals statically applied to subtrees of the call tree is suitable.

```
subroutine A()
  call B(); call D(); call B();
end subroutine A
subroutine B()
  call C()
end subroutine B
subroutine C()
  call E()
end subroutine C
```

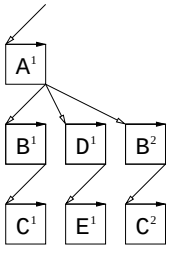


Figure 2.10: Dynamic call tree of a simple calling hierarchy

S^n	n-th invocation of subroutine S		subroutine call
	run forward	→	order of execution
	store checkpoint		restore checkpoint
	run forward and tape		run adjoint

Table 2.1: Symbols for call tree reversal

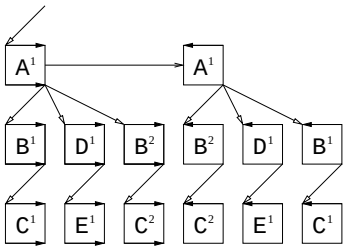


Figure 2.11: Dynamic call tree for split reversal

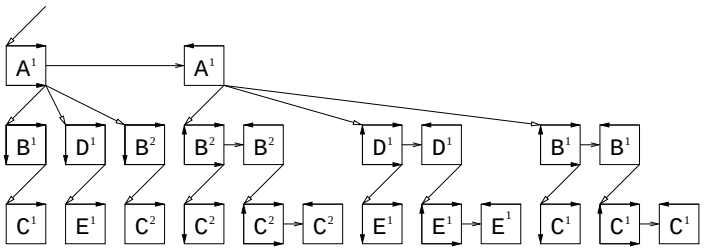


Figure 2.12: DCT of adjoint obtained by joint reversal mode

Chapter 3

Components of OpenAD/F

OpenAD/F is built on components that belong to a framework designed for code transformation of numerical programs. The components are tied together either via programmatic interfaces or by communication using the XAIF language. The transformation of the source code follows the pipeline shown in Fig. 1.1. In Sec. 3.1 we describe the language-independent components of OpenAD framework and Sec. 3.2 provides details in the Fortran front-end. The regular setup procedure for OpenAD/F, see also Sec. 4.1, will retrieve all components into an `OpenAD/` directory to which we refer from here on.

3.1 Language Independent Components (OpenAD)

The component design of the tool aims for reuse of the different components for different types of source transformation of numerical codes, for different programming languages in which these tools are written and finally also for the reuse of the individual components in different contexts. A second, equally important concern is the flexibility of the tool. This section covers the language independent components that make up the core OpenAD framework.

3.1.1 Static Code Analyses (OpenAnalysis)

The OpenAnalysis toolkit, see [28], separates program analysis from language-specific or front-end specific intermediate representations. This separation enables a single implementation of domain-specific analyses such as activity analysis, to-be-recorded analysis, and linearity analysis in OpenAD/F. Standard analyses implemented within OpenAnalysis such as CFG construction, call graph construction, alias analysis, reaching definitions, ud- and du-chains, and side-effect analysis are also available via OpenADFortTk.

OpenADFortTk interfaces with OpenAnalysis as a producer and a consumer. A description of Alias analysis illustrates this interaction. XAIF requires an alias map data structure, in which each variable reference is mapped to a set of virtual locations that it may or must reference. For example, if a global variable `g` is passed into subroutine `foo` through the reference parameter `p`, variable references `g` and `p` will reference the same location within the subroutine `foo` and therefore be aliases. OpenAnalysis determines the aliasing relationships by querying an abstract interface called the alias IR interface, which is a language-independent interface between OpenAnalysis and any intermediate representation for an imperative programming language. An implementation of the alias IR interface for the Fortran 90 intermediate representation is part of OpenADFortTk. The interface includes queries for an iterator over all the procedures, statements in those procedures, memory references in each statement, and memory reference expression and location abstractions that provide further information about memory references and symbols. The results of the alias analysis are then provided back to OpenADFortTk through an alias results interface.

Using language-independent interfaces between OpenAnalysis and the intermediate representation will enable alias analysis for multiple language front-ends without requiring XAIF to include the union of all language features that affect aliasing (ie. pointer arithmetic and casting in C/C++ and equivalence in Fortran 90). Instead OpenAnalysis has analysis-specific interfaces for querying language-specific intermediate representations.

OpenAnalysis also performs activity analysis. For activity analysis the independent and dependent variables of interest are communicated to the front-end through the use of pragmas, see Sec. 3.2.2. The results of the analysis are then encoded by the Fortran 90 front-end into XAIF. The analysis indicates which variables are active at any time, which memory references are active, and which statements are active.

The activity analysis itself is based on the formulation in [17]. The main difference is that the data-flow framework in OpenAnalysis does not yet take advantage of the structured data-flow equations. Activity analysis is implemented in a context-insensitive, flow-sensitive interprocedural fashion.

All sources for OpenAnalysis can be found under `OpenAD/OpenAnalysis/`.

3.1.2 Representing the Numerical Core (XAIF)

To obtain a language independent representation of programs across multiple programming languages one might choose the union of all language features. On the other hand one can observe that the majority of differences between languages does not lie with the elemental numerical operations that are at the heart of AD transformations. This more narrow representation is a compromise permitting just enough coverage to achieve language independence for the numerical core across languages. Consequently, certain program features are not represented and have to be retained by the language specific front-end to reassemble the complete program from the (transformed) numerical core. Among the generic language features not considered part of the numerical core are:

- user type definitions and member access, see also Sec. 3.2.1
- pointer arithmetic
- I/O operations
- memory management, see also Sec. 6
- preprocessor directives

For a more general discussion regarding this compromise see also [34]. It is apparent that certain aspects of the adjoint code such as checkpointing, see Sec. 2.4, and taping, see Sec. 2.3, can involve memory allocation and various I/O schemes and therefore are not amenable to representation in the XAIF. At the same time it is also clear that the way one has to handle the memory and I/O for taping and checkpointing is primarily determined by the problem size at runtime and not primarily by the code we transform. Therefore in OpenAD such transformation results are handled by specific code expansion for subroutine specific templates and inlinable subroutine calls in the post processor, see Sec. 3.2.4. This not only avoids the typically language specific I/O and memory management aspects, it also affords additional flexibility.

The format of choice in OpenAD is an XML-based [13] hierarchy of directed graphs, referred to as XAIF[18]. Using XML is motivated by the existence of XML parsers and the ability to specify the XAIF specific XML contents with a schema which the parser can use to validate a given XAIF representation. The current XAIF schema is documented at [36]. The basic building blocks are the same data structures commonly found in compilers from top down call graph with scopes and symbol tables, control flow graphs as call graph vertices, basic blocks as control flow graph vertices, statement lists contained in basic blocks, assignments as a statement with expression graphs, and variable references and intrinsic operations as expression graph vertices. The role of the respective elements in the XAIF schema is fairly self evident. Elements are associated by containment. In the graph structures edges refer to source and target vertices by vertex ids. Variable references contain references to symbols which in turn are associated to symbol table elements via a scope and a symbol id. An example can be found in Sec. 3.2.3, Fig. 3.7. Further documentation for individual elements can be found directly in the schema annotations.

The XAIF also contains the results of the code analyses provided by OpenAnalysis, see Sec. 3.1.1. Some are expressed either as additional attributes on certain XAIF elements, e.g. for activity information. Side-effect analysis provides lists of variable references per subroutine, i.e. a call graph vertex element. DuUd chains are expressed as list of ids found in the assignment XAIF element. Alias information is expressed as set of virtual addresses. DuUd chains and alias information is collected in maps indexed by keys associated with the call graph. References to individual entries held in these maps are expressed via foreign key attributes in the elements.

The source transformation at the code of OpenAD potentially changes and augments all elements of the XAIF. While it would in principle be possible to express the result entirely in the plain XAIF format we already mentioned the code expansion approach. Therefore the transformed XAIF adheres to a schema that is extended by a construct to represent inlinable subroutine calls and a specific list of control flow graphs that the post processor places in predefined locations in the subroutine template.

The XAIF schema and examples can be found under `OpenAD/xaif/`.

3.1.3 Transforming the Numerical Core (xaifBooster)

The transformation engine that differentiates the XAIF representation of $\boldsymbol{f}$ is called xaifBooster. It is implemented in C++ based on a data structure that represents all information supplied in the XAIF input together with collection of algorithms that operate on this data structure, modify it and produce transformed XAIF output as the result.

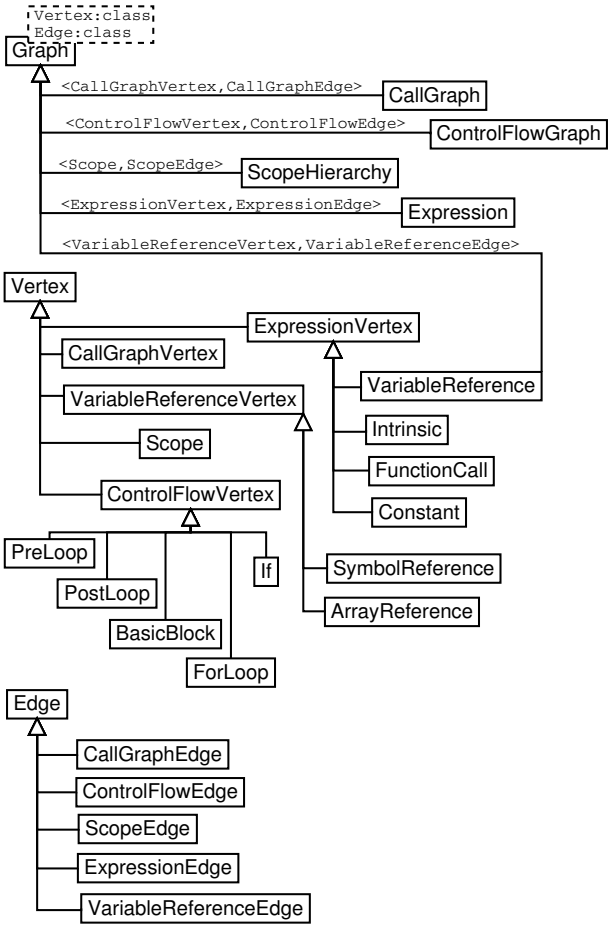


Figure 3.1: Simplified class inheritance in xaifBooster

All sources for xaifBooster can be found under `OpenAD/xaifBooster/`. The principal setup of the source tree is shown in Table 6.1. The xaifBooster data structure closely resembles the information one would find in a compiler's high level internal representation. the boost graph library [8] and the Standard C++ Library[14]. Figure3.1 and Fig. 3.2 show simplified subsets of the classes occurring in the xaifBooster data structure in the inheritance as well as the composition hierarchy. A doxygen generated documentation of the entire data structure can be found on the OpenAD website [27]. The class hierarchy is organized top down with a single `CallGraph` instance as the top element. The top down structure is also imposed on the ownership of dynamically allocated elements. Wherever possible, the class interfaces encapsulate dynamic allocation of members. Only in cases of containment of polymorphic elements is explicit dynamic allocation outside of the owning class' members appropriate. In these cases the container class interface naming and documentation indicates the assumption of ownership of the dynamically allocated elements being supplied to the container class. An example is the graph class `Expression` accepting vertex instances that can be `Constant`, `Intrinsic` , etc.

The transformation algorithms are modularized to enable reuse in different contexts. Fig. 3.3 shows some implemented algorithms with dependencies. To avoid conflicts the transformation algorithms the data structure representing the input code is never directly modified. Instead, any data representing modifications or augmentations of the original representation element in a class `<name>` are held in algorithm specific instances of class `<name>Alg`. The association is done via mutual references accessible through `get<name>AlgBase()` and `getContaining<name>()` respectively. The instantiation of the algorithm specific classes follows the factory design pattern. The factory instances in turn are controlled by a transformation algorithm specific `AlgFactoryManager` classes. Further details can be found in [33], however, the code code for this mechanism is fairly self-explanatory.

In the following sections we want to concentrate on the transformation the algorithms execute while deferring to the generated code documentation for most technical details.

Each algorithm has a driver `t.pp` (compiled into a binary `t`) found in `algorithms/<the_algorithm_name>/test/` that encapsulates the algorithm in a stand-alone binary which provides the functionality described in the following

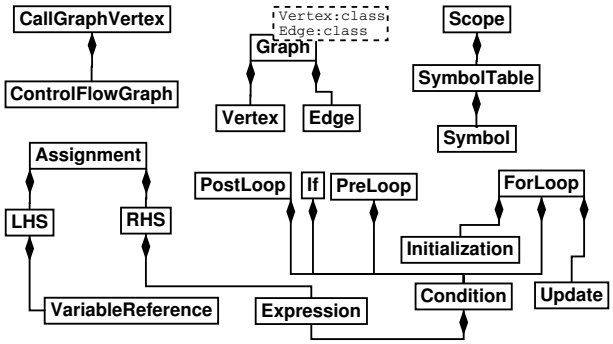


Figure 3.2: Simplified class composition in xaifBooster

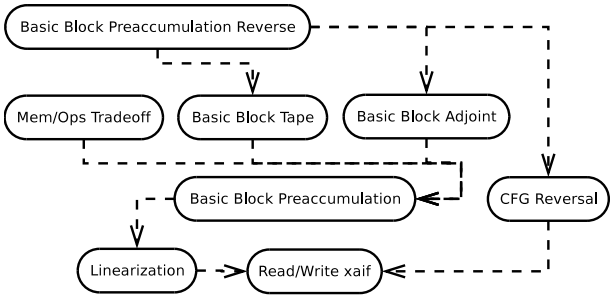


Figure 3.3: xaifBooster algorithms

sections. For details on the invocation and command line options refer to Sec. 4.4.

3.1.3.1 Reading and Writing XAIF

Reading and Writing the XAIF is part of basic infrastructure found in the sources in `system/`. Parsing is done through the Xerces C++ XML parser [37] such that the XML element handler implementations, see `system/src/XAIFBaseParserHandlers.cpp`, build the xaifBooster data structure from the top down. As an additional consistency check all components that read XAIF data have the validation according to the schema enabled. Beyond the schema validation these components perform validity checks. Therefore, manual modifications of XAIF data , while possible, should be done judiciously.

The unparsing of the transformed data structure into XAIF is performed through a series of that traverses the data structure and the respective algorithm specific data. For information of the files containing the XAIF representation refer to Sec. 4.4.

Aside from the parsing of the actual input XAIF there is also the so called *catalog of inlinable intrinsics* supplied as an XML following a specialized schema in XAIF, see Sec. 3.1.3.2 and Sec. 3.2.3. There is also a driver at this level found in `system/test/t.cpp` used to verify reading and writing functionality. It can be used to establish that the tool pipeline preserves the semantics of the original program when no transformation is involved.

3.1.3.2 Linearization

Sec. 2 explained the computation of the local partial derivatives c_{ji} that can be thought of as edge labels in the computational graph G. Per canonicalization (see Sec. 3.2.1) all elemental ϕ occur only in the right-hand side of an assignment. For each ϕ we look up the definition of the respective partials in the intrinsics catalog. [not sure how much detail is necessary] The partials are defined in terms of positional arguments, see Fig. 3.4.

Because of this, the right-hand-side expression may have to be split up into subexpressions to assign intermediate values to ancillary variables that can be referenced in the partial computation, for an example see the code shown in Fig. 2.4. In cases of the left-hand-side variable occurring on the right-hand-side (or being may-aliased to a right-hand-side variable, see Sec. 3.1.1) we also require an extra assignment to delay the (potential) overwrite until after the partials depending on the original variable value have been computed. The result of the Linearization is a representation for code containing the potentially split assignments along with assignments for each non-zero edge label c_{ji}. These representations are contained in the `xaifBoosterLinearization::AssignmentAlg` instances associated

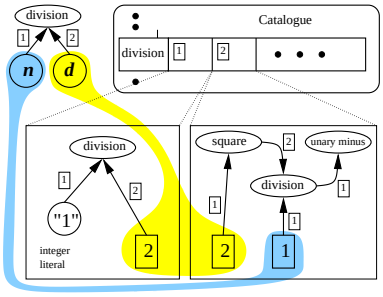


Figure 3.4: Partial expressions for the division operator

with each assignment in the XAIF. The generated code after unparsing to Fortran is compilable but does by itself not compute useful derivative information at the level for the target function $\boldsymbol{f}$. The transformation driver is used to verify the results of the linearization transformation.

3.1.3.3 Basic Block Preaccumulation

This transformation generates a code representation that can be used to compute derivatives in forward mode. It builds upon the Linearization done in Sec. 3.1.3.2. The first step constructs the computational graphs G for contiguous assignment sequences in a given basicblock. To ensure semantic correctness of the graph being constructed in the presence of aliasing it relies on alias analysis and define-use/use-define chains supplied by OpenAnalysis, see Sec. 3.1.1. The algorithm itself is described in detail in [31]. Because the analysis results supplied by OpenAnalysis are always conservatively correct it may not be possible to flatten all assignments into a single computational graph. In such cases a sequence of graphs is created. Likewise, the occurrence of a subroutine call leads to a split in the graph construction. In the context of Sec. 2 one may think of the sets of assignments forming each of these graphs as a separate basicblock. The driver for the algorithm allows to disable the graph construction across assignments and restrict it to single right-hand sides by adding the `-S` command line flag.

Based on the constructed G an elimination sequence has to be determined. To allow a choice for the computation of the elimination sequence the code uses the interface coded in `algorithms/CrossCountryInterface/` and by default calls the angel library [2, 6, 22]. angel determines an elimination sequence and returns it as fused multiply add expressions in terms of the edge references. There are several heuristics implemented within angel that control the selection of elimination steps and thereby the preaccumulation code that is generated. The algorithm code calls a default set of heuristics. However, all heuristics use the `CrossCountryInterface` and therefore different heuristics can be selected with minimal changes in algorithm code.

The second step in this transformation is the generation of preaccumulation code. First it turns the abstract expression graphs returned by angel into assignments and resolves the edge references into the labels c_{ji}. The resulting code resembles what we show in Fig. 2.5. Then it generates the code that eventually performs the saxpy operations shown in (2.6). Considering the input and output variables $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ of a basicblock the code generation also ensures proper propagation of $\dot{x}_i^j$ of variables $x_i^j \in \boldsymbol{x}_j \cap \boldsymbol{y}_j$ by saving the $\dot{x}_i^j$ in temporaries. The example in Fig. 2.8 illustrates this case. The detection of the intersection elements relies on the alias analysis provided by OpenAnalysis. To reduce overhead the generated saxpy operations we generate saxpy calls following the interface specified in `algorithms/DerivativePropagator/` for the following four cases:

$$\text{(a): } \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x}, \text{ (b): } \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x} + \dot{y}, \text{ (c): } \dot{y} = \dot{x}, \text{ (d): } \dot{y} = 0 \quad . \tag{3.1}$$

The generated code is executable and represents an overall forward mode according to (2.6) with basicblocklocal preaccumulation in cross-country fashion.

3.1.3.4 Memory/Operations Tradeoff

This algorithm can be seen as an alternative to the angel library. Like angel it uses the `CrossCountryInterface`. In its implementation it replaces the call to angel with one to its own internal routines that determine an elimination sequence according to a selectable set of heuristics. In difference to the angel heuristics they aim for a tradeoff between the number of operations required to complete an elimination sequence on the one hand and the temporal locality of the c_{ji} in memory on the other hand. The rationale for these heuristics is the observation that in many modern computer architectures the performance is memory bound, i.e. a few additional operations can easily be

absorbed if we keep all the necessary data in cache. All heuristics take as an input a set $\Theta \neq \emptyset$ of target elements, that is a set of vertices or edges from G, or faces from $\mathcal{G}$. The heuristic selects a nonempty subset $\Theta' \subseteq \Theta$ from this set. In order to determine a single elimination target a sequence of heuristics may be applied that successively shrink the target set concluding with a tie breaker such as selecting the next target that would be eliminated in forward or reverse mode. Table 3.1 describes the selection criterion of a heuristics with respect to an elimination technique. If the selection criterion is not met by any target in Θ, then $\Theta' = \Theta$. The driver allows a sequence of heuristics to be

	VERTEX	EDGE	FACE
SIBLING	select vertices that share at least one predecessor and successor with the most recently eliminated vertex.	N/A	N/A
SIBLING2	select vertices with the maximal product of the number of predecessors and the number of successors shared with the most recently eliminated vertex	select edges with the same source / target and the maximal number of successors / predecessors shared with the successors / predecessors of the most recently front / back eliminated edge	N/A
SUCCPRED	select vertices that were either predecessors or successors of the most recently eliminated vertex	N/A	N/A
ABSORB	N/A	N /A	select faces that are absorbed
MARKOWITZ	select vertices with the lowest Markowitz degree	select edges with the lowest Markowitz degree	N/A
FORWARD	select the target next in forward mode		
REVERSE	select the target next in reverse mode		

Table 3.1: Heuristics selection criteria

selected via string supplied as an argument to the commandline switch `-H`. The string needs to contain the target selection, one of `Vertex`, `EDGE`, or `FACE` followed by a sequence of heuristics that should include at least on of the tie breakers `FORWARD` or `REVERSE`. Obviously the data locality criteria still are rather simplistic but the code is easily extensible for more elaborate strategies.

The generated code is executable and represents an overall forward mode according to (2.6) with basicblocklocal preaccumulation in cross-country fashion.

3.1.3.5 Using the ANGEL Library

[JU: This was a placeholder to talk about the angel lib but I think I will remove this unless somebody strongly objects]

3.1.3.6 CFG Reversal

Sec. 2.3 explains the principal approach to the reversal of the CFG. The CFG reversal as implemented in this transformation is by itself not useful as unparsed code other than for checking the correctness without interference from other transformations. It is a major building block for the adjoint code generator described in Sec. 3.1.3.8. The loop counters and branch identifiers are stored the same stack data structure that is used for the *tape* (introduced in Sec. 2.3 and also used in Sec. 3.1.3.7. The reversal of loops and branches as detailed in [24] assumes CFGs to be well-structured, that is, essentially to be free of arbitrary jump instructions such as `GOTO` or `CONTINUE`. It is of course possible to reverse such graphs, for instance by enumerating all basicblocks, recording the execution sequence and invoking them according to their recorded identifier in large `SWITCH` statement in reverse order. Such a reversal is obviously less efficient than a code that, by employing proper control flow constructs, aids compiler optimization. For the same reason well tuned codes implementing the target function $\boldsymbol{f}$ will avoid arbitrary jumps and therefore we have not seen sufficient demand to implement a CFG reversal for arbitrary jumps.

The reversal of loop constructs such as `do i=1,10` replaces the loop variable `i` with a generated variable name, say `t` and we loop up to the stored execution count which we will call `c` here. Then the reversed loop is `do t=1,c`. Quite often the loop body contains array dereferences such as `a(i)` but `i` is no longer available in the reversed loop. We call this kind of loop reversal *anonymous*. To access the proper memory location `i` will have to be stored along with the loop counters and branch identifiers in the tape stack. To avoid this overhead the loop reversal may be declared *explicit* by prepending `!$openad xxx simple loop` to the loop in question. With this directive the original loop variable will be preserved, the reversed loop in our example constructed as `do i=10,1,-1` and no index values for the array references in the loop body are stored. In general the decision when an array index needs to be stored is

forward	adjoint
(a): $\dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x}$	$\overline{x} = \frac{\partial y}{\partial x} \cdot \overline{y} + \overline{x}, \;\; \overline{y} = 0$
(b): $\dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x} + \dot{y}$	$\overline{x} = \frac{\partial y}{\partial x} \cdot \overline{y} + \overline{x}$
(c): $\dot{y} = \dot{x}$	$\overline{x} = \overline{y}, \;\; \overline{y} = 0$
(d): $\dot{y} = 0$	$\overline{y} = 0$

Table 3.2: saxpy operations from (3.1) and their corresponding adjoints

better answered with a code analysis similar to TBR analysis [17]. Currently we do not have such analysis available and instead as a compromise define the *simple* loop which can reversed explicitly as follows.

- loop variables are not updated within the loop,
- the loop condition does not use `.ne.`,
- the loop condition's left-hand side consists only of the loop variable,
- the stride in the update expression is fixed,
- the stride is the right-hand side of the top level `+` or `-` operator,
- the loop body contains no index expression with variables that are modified within the loop body.

While these conditions can be relaxed in theory, in practice the effort to implement the transformation will rise sharply. Therefore they represent a workable compromise for the current implementation. Because often multidimensional arrays are accessed with nested loops the loop directive when specified for the outermost loop will assume the validity of the above conditions for everything within the loop body including nested loop and branch constructs. More details on this aspect can be found in [32].

3.1.3.7 Writing and Consuming the Tape

Sec. 2 explains the need to store the $\frac{\partial \phi_j}{\partial v_i}$ on the tape. The writing transformation[1] stores the nonzero elements of local Jacobians $\boldsymbol{J}_j$. It is implemented as an extension of the preaccumulation in Sec. 3.1.3.3 but instead of using the Jacobian elements in the forward saxpy operations as in (2.6) we store them on a stack as shown for the example code in Fig. 2.8(a). The tape consuming transformation algorithm[2] reinterprets the saxpy operations generated in Sec. 3.1.3.3 according to Table 3.2. The tape writing and consumption implemented in these transformations are by themselves not useful as unparsed code other than for checking the correctness without interference from other transformations. They are, however, major building blocks for the adjoint code generator described in Sec. 3.1.3.8.

3.1.3.8 Basic Block Preaccumulation Reverse

This transformation[3] represents the combination of the various transformation into a coherent representation that, unparsed into code and post-processed, compiles as an adjoint model. For the postprocessing steps refer to Sec. 3.2.4. Additional functionality is the generation of code that is able write and read checkpoints at a subroutine level, see also Sec. 2.4. This part of the transformation relies heavily on the results of side-effect analysis, see Sec. 3.1.1 and the inlinable subroutine call mechanism of the postprocessor, see Sec. 3.2.4.2, to accomplish the checkpointing. The driver offers command line options

- to change subroutine argument intents such that checkpointing can take place; while checkpointing will generally be needed this option is useful for certain application scenarios where the intent change can be avoided.
- to validate the XAIF input against the schema; the validation takes considerable time for large XAIF files
- to specify a list of subroutines that have a wrappers which should be called in its place
- to force the renaming of all non-external subroutines which may be necessary for applications which expose only portions of the code to OpenAD/F.

[1] see `algorithms/BasicBlockPreaccumulationTape/`
[2] see `algorithms/BasicBlockPreaccumulationTapeAdjoint/`
[3] see `algorithms/BasicBlockPreaccumulationReverse`

```
y = x * foo(a,b)
```

(a)

```
! type matching foo return
real oad_ctmp0
! and for the assignment
call oad_s_foo(a,b,oad_ctmp0)
y = x * oad_ctmp0
```

(b)

Table 3.3: Canonicalizing a function(a) to a subroutine(b) call

```
real function foo(a,b)
! declarations, body etc...
  foo = ...
end
```

(a)

```
subroutine oad_s_foo(a,b,oad_ctmp0)
! type matches foo return
  real oad_ctmp0
! old declarations, body etc...
  oad_ctmp0 =...
end
```

(b)

Table 3.4: Canonicalizing a function(a) to a subroutine(b) definition

3.2 Language Dependent Components (OpenADFortTk)

For simplicity we consider all language dependent components part of the OpenAD Fortran Tool Kit (OpenADFortTk). The following sections provide details for the various subcomponents that are used in transformation pipeline in the following sequence.

1. The *canonicalizer* converts programming constructs into a canonical form described in Sec. 3.2.1.
2. The compiler front-end mfef90 parses Fortran and generates an intermediate representation (IR) in the whirl format, see Sec. 3.2.2
3. whirl2xaif is a bridge component that
   - drives the various program analyses (see Sec. 3.1.1),
   - translates the numerical core of the program and the results of the program analyses from whirl to XAIF.

   see also Sec. 3.2.3
4. xaif2whirl is bridge component that translates the differentiated numerical core represented in XAIF into the whirl format. see Sec. 3.2.3.
5. whirl2f is the "unparser" that converts whirl to Fortran, see Sec. 3.2
6. The *postprocessor* is the final part of the transformation that performs template expansion as well as inlining substitutions, see Sec. 3.2.4

3.2.1 Canonicalization

In Sec. 3.1.2 we explain how the restriction to the numerical core contributes to the language independence of the transformation engine. Still, even for a single programming language, the numerical core often exhibits a large variability in expressing semantically identical constructs. To streamline the transformation engine we reduce this variability by *canonicalizing* the numerical core. Because it is done automatically, the canonicalization does *not* restrict the expressiveness of the input programs supplied by the user. Rather it is a means to reduce the development effort of the transformation engine. In the following we describe the canonical form. The canonicalizer mechanism is written in Python. All sources can be found under `OpenADFortTk/tools/canonicalize/`.

Canonicalization 1 *All function calls are canonicalized into subroutine calls, see Table 3.3. For the transformations, in particular the basicblock level preaccumulation we want to ensure that an assignment effects a single variable on the left-hand side. Therefore the right-hand-side expression ought to be side-effect free. While often not enforced by compilers, this is a syntactic requirement for Fortran programs. Rather than determining which user-defined functions have side effects, we pragmatically hoist* all *user-defined functions. Consequently the right-hand-side expression of an assignment consists only of elemental operations ϕ typically defined in a programming language as built-in operators and intrinsics. The canonicalizer also performs the accompanying transformation of the function definition Table 3.4(a) into a subroutine definition Table 3.4(b). The* `oad_s_` *prefix can be adjusted by modifying* `__call_prefix`

```
call foo(x*y)
```

(a)

```
real ad_ctmp0
! ...
ad_ctmp0 = x*y
call foo(ad_ctmp0)
```

(b)

Table 3.5: Before(a) and after(b) hoisting a non-variable parameter

```
integer,paramter :: n=10
real :: a,b
common /foo/ a(n),b
```

(a)

```
module oad_m_foo
private n
integer,paramter :: n=10
real :: a(n),b
end module
```

(b)

Table 3.6: Converting a common block (a) to a module (b)

in `Lib/canon.py`. *A particular canonicalization of calls without canonicalization of definitions is applied to the* `max` *and* `min` *intrinsics because in Fortran they do not have closed form expressions for the partials. OpenAD/F provides a run time library containing definitions for the respective subroutines called instead.*

Canonicalization 2 *Non-variable actual parameters are hoisted to temporaries. Any value passed to a routine could conceivably need augmentation. Furthermore, only variables can be augmented. Consequently, OpenAD hoists all non-variable actual parameters into temporaries.*

In the Fortran context there is also the need to canonicalize certain constructs to support the implementation with a specific active type.

Canonicalization 3 *Common blocks are converted to modules. The purpose of this canonicalization is to ensure proper initialization of active global variables. The method of conversion is to simply declare the elements of the common block as module variables. Care must be taken to privatize and declare any symbolic size parameters for elements of the common block. See Table 3.6 for an example.*

Because none of the above canonicalizations are intended to produce manually maintainable code we prefer simplicity over more sophisticated transformations e.g. a module generator which abstracts dimension information shared between common blocks.

3.2.2 Compiler Front-End Components (from Open64)

The choice of Open64 for some of the programming-language-dependent components ensures some initial robustness of the tool that is afforded by an industrial-strength compiler. The Center for High Performance Software Research (HiPerSoft) at Rice University develops Open64 [26] as a multi-platform version of the SGI Pro64/Open64 compiler suite, originally based on SGI's commercial MIPSPro compiler.

OpenAD/F uses the parser, an internal representation and the unparser of the Open64 project. The classical compiler parser mfef90 produces a representation of the Fortran input in a format known as very high level or source level whirl . The whirl representation can unparsed into Fortran using the tool whirl2f. The source level whirl representation resembles a typical abstract syntax tree with the addition of machine type deductions. The original design of whirl in particular the descent to lower levels closer to machine code enables good optimization for high performance computing in Fortran, C, and C++. HiPerSoft's main contribution to the Open64 community has been the source level extension to whirl which is geared towards supporting source-to-source transformations and it has invested significant effort in the whirl2f unparser.

For the purpose of AD, user-supplied hints and required input is typically not directly representable in programming languages such as Fortran. For example, an AD tool must know which variables in the code for $\boldsymbol{f}$ are independent and which are dependent. While it is possible to supply this information externally, for instance with a configuration file, we introduced a special pragma facility, encoded within Fortran comments. While pragmas are intrusive they have the added advantage to be parsed by the front-end and be associated with a given context in the code. Thereby code and AD information is easier kept in sync. For OpenAD/F we extended the Open64 components to generate and unparse these pragma nodes represented in whirl. The behavior is similar to many other special-purpose Fortran pragma systems such as OpenMP [29]. To specify a variable y as a dependent, the user writes

```
subroutine daerfj(x,fvec)
  double precision x(4), fvec(11)
  ! declare and initialize local variables
  do i = 1, 11
    ! more computations
    fvec(i) = y(i) x(1)*t emp1/temp2
  end do
  ! print some results (PASSIVE statements)
end subroutine daerfj
```

Figure 3.5: Example Fortran code

```
FUNC_ENTRY <1,20,daerfj_>
   ...
   F8ISTORE 0 T<33,anon_ptr.,8>
    F8SUB
     F8F8ILOAD 0 T<11,.predef_F8,8>...
      U8ARRAY 1 8
       U8LDA 0 <2,7,Y> T<32,anon_ptr.,8>
       I4INTCONST 11 (0xb)
       I4I4LDID 0 <2,3,I> T<4,.predef_I4,4>
     F8DIV
      F8MPY
       F8F8ILOAD 0 T<11,.predef_F8,8>...
        U8ARRAY 1 8
         U8U8LDID 0 <2,1,X>...
         I4INTCONST 4 (0x4)
         I4INTCONST 1 (0x1)
       F8F8LDID 0 <2,4,TEMP1>...
      F8F8LDID 0 <2,5,TEMP2>...
    # lefthandside (elided)
...
```

Figure 3.6: Part of whirl for Fig. 3.5

`!$openad dependent(y)`, where `$openad` is the special prefix that identifies OpenAD pragmas. To provide flexibility we introduced a generic `!$openad xxx <some text>` pragma[4] that can communicate arbitrary pieces of text through the pipeline. These generic pragmas can be associated with whole procedures, single statements, or groups of statements. They provides an easy way to implement additional user hints while eliminating the significant development costs associated with modifying Open64.

3.2.3 Translating between whirl and XAIF

The translation of whirl into XAIF (whirl2xaif), feeding it to the transformation engine, and then backtranslating the differentiated XAIF into whirl (xaif2whirl) are crucial parts of the tool pipeline. Two distinguishing features of XAIF shape the contours of whirl2xaif and xaif2whirl.

First, because XAIF represents only the numerical core of a program, many whirl statements and expressions are not translated into XAIF. For instance, XAIF does not represent dereferences for user-defined types because numerical operations simply will not involve the user defined type as such but instead always the numerical field that eventually is a member of the user defined type (hierarchy). Derived type references are therefore *scalarized*. This consists of converting the derived type reference into a canonically named scalar variable. To ensure correctness, this scalarization must be undone upon backtranslating to whirl. The effect can be observed in the generated XAIF for `v%f` where the dereference shows up for example as `<xaif:SymbolReference vertex_id="1" scope_id="4" symbol_id="scalarizedref0">` and in the XAIF symbol table we would find `scalarizedref0` as a scalar variable with a type that matches that of `f`. However, variable references of user defined type can still show up in the XAIF for instance as subroutine parameters. Such references are listed with an *opaque* type. Statements in the original code that do not have an explicit representation in the XAIF, such as I/O statements, take the form of annotated markers that retain their position in the representation during the transformation of the XAIF. Given the original whirl and the differentiated XAIF (with the scalarized objects, opaque types and annotated markers intact), xaif2whirl is able generates new whirl representing the differentiated code while restoring the statements and types not shown in the XAIF.

Second, XAIF provides a way to represent the results of common compiler analyses. To provide these to the transformation engine whirl2xaif acts as a driver for the analyses provided by the OpenAnalysis package, see Sec. 3.1.1. In particular it implements the abstract OpenAnalysis interface to the whirl IR. The results returned by OpenAnalysis are then translated into a form consistent with XAIF.

[4] The mnemonic behind the name is that as x is the typical variable name, so `!$openad xxx` is the *variable* pragma.

```
<xaif:CallGraph ...>
  <!--Scope and symbol tables >
  <xaif:ControlFlowGraph symbol_id="daerfj_"...>
    <!-- BBs for 1) CFG Entry, 2) init statements, 3) DO loop... -->
    <!-- statements inside DO loop -->
    <xaif:BasicBlock vertex_id="4">
      ...
      <xaif:Assignment statement_id="3">
        <xaif:AssignmentLHS ... />
        <xaif:AssignmentRHS>
           <xaif:Intrinsic vertex_id="1" name="sub_scal_scal"/>
           <xaif:VariableReference vertex_id="2">
             <xaif:SymbolReference vertex_id="1" symbol_id="Y"/>
             <xaif:ArrayElementReference vertex_id="2">
                <xaif:Index>
                  <xaif:VariableReference vertex_id="1" ... "I" />
                </xaif:Index>
             </xaif:ArrayElementReference>
             <xaif:VariableReferenceEdge source="1" target="2"/>
           </xaif:VariableReference>
           <xaif:Intrinsic vertex_id="3" name="div_scal_scal"/>
           <!-- x(1)*temp1/temp2 -->
  <xaif:ExpressionEdge source="2" target="1" position="1"/>
           <xaif:ExpressionEdge source="3" target="1" position="2"/>
        </xaif:AssignmentRHS>
        ...
  <!-- PASSIVE statements after DO loop to print results -->
  <xaif:BasicBlock vertex_id="6">
  <xaif:Marker statement_id="1" annotation="{WHIRL_Id#201}"/>
  </xaif:BasicBlock>
  <!--- ControlFlowGraph edges -->
```

Figure 3.7: Part of XAIF for Fig. 3.5

The companion tool xaif2whirl backtranslates XAIF into whirl. As indicated above it has to take care of restoring filtered out statements and type information. Because the differentiated XAIF relies on postprocessing, see Sec. 3.2.4, its other major challenge is the creation of whirl that contains the postprocessor directives related to three tasks to be accomplished by the postprocessor.

- The declaration and use of the active variables;
- The placement of inlinable subroutine calls;
- The demarcation of the various alternative subroutine bodies used in the subroutine template replacements.

3.2.4 Postprocessing

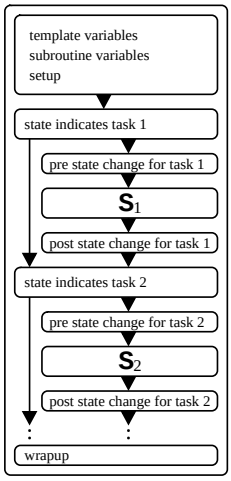
The postprocessor performs the three tasks outlined at the end of Sec. 3.2.3.

3.2.4.1 Use of the Active Type

The simplest postprocessing task is the concretization of the active variable declarations and uses. The main rationale for postponing the concretization of the active type is flexibility with respect to the actual active type implementation. The current postprocessor is written in Perl[5] and therefore is much easier to adapt to a changing active type implementation than to find find the proper whirl representation and modify xaif2whirl to create it. However, it should be noted right away, that the ease of adaptation is clearly correlated to the simplicity and in particular the locality of the transformation. The advantage disappears with increased complexity of the transformation. For an active variable, for example `v`, the representation created by xaif2whirl in whirl and then unparsed to Fortran, shows up as `TYPE (OpenADTy_active) v`. In whirl the type remains abstract because the accesses to the conceptual value and derivative components are represented as function calls `__value__(v)` and `__deriv__(v)` respectively. The concretized versions created by the postprocessor for the current active type implementation, see `runTimeSupport/simple/OpenAD_active.f90` are `type(active) v` for the declaration and simply `v%v` for the value `v%d` for the derivative component respectively and each subroutine will also receive an additional `USE` statement which makes the type definition in `OpenAD_active` known.

[5] The source code can be found under `OpenADFortTk/tools/multiprocess/`. A rewrite in Python reusing the same Fortran parsing functionality of the canonicalizer is underway.

(a)



(b)

```
subroutine template()
  use OpenAD_tape ! tape storage
  use OpenAD_rev  ! state structure
!$TEMPLATE_PRAGMA_DECLARATIONS

  if (rev_mode%tape) then
! the state component
! 'taping' is true
!$PLACEHOLDER_PRAGMA$ id=2
  end if

  if (rev_mode%adjoint) then
! the state component
! 'adjoint' run is true
!$PLACEHOLDER_PRAGMA$ id=3
  end if


end subroutine template
```

Figure 3.8: Subroutine template components (a), split-mode Fortran90 template (b)

3.2.4.2 Inlinable Subroutine Calls

The second task, the expansion of inlinable subroutine calls, is more complex because any call expansion has now the scope of a subroutine body. The calls unparsed from whirl to Fortran are regular subroutine call statements. They are however preceded by an inline pragma `!$openad inline <name(parameters)>` that directs the postprocessor to expand the following call according to a definition found in an input file[6], see also `runTimeSupport/simple/ad_inline.f`. For example, pushing a preaccumulated sub-Jacobian value as in Fig. 2.9(a) might appear in the code as

```
C$openad inline push(subst)
CALL push(OpenAD_Symbol_5)
```

for which we have a definition in `ad_inline.f` as

```
subroutine push(x)
C$openad$ inline DECLS
  use OpenAD_tape
  implicit none
  double precision :: x
C$openad$ end DECLS
  double_tape(double_tape_pointer)=x
  double_tape_pointer=double_tape_pointer+1
end subroutine
```

The postprocessor ignores the `DECLS` section and expands this to

```
double_tape(double_tape_pointer) = OpenAD_Symbol_5
double_tape_pointer = double_tape_pointer+1
```

Note, that for flexibility any calls with inline directives for which the postprocessor cannot find an inline definition remain unchanged. For example we may instead compile the above definition for `Push` and link it instead.

3.2.4.3 Subroutine Templates

The third task, the subroutine template expansion is somewhat related the inlining. In our example above, the tape storage referred to in the `Push` need to be defined and in the design the subroutine template is the intended place for these definitions, in our example achieved through including the `use` statement in the template code, see Fig. 3.9. The main purpose of the subroutine template expansion however is to orchestrate the call graph reversal. The reversal schemes introduced in Sec. 2.4 can be realized by carrying state through the call tree.

The basic building blocks from the transformations in Sec. 3.1.3 are variants $\mathtt{S}_i$ of the body of an original subroutine body $\mathtt{S}_0$, each accomplishing one of the tasks shown as one of the squares with arrows in Table 2.1. For instance, the taping variant is created by the transformation in Sec. 3.1.3.7 or the checkpointing by the transformation in Sec. 3.1.3.8. To integrate the S_i into a particular reversal scheme, we need to be able to make all subroutine calls in the same fashion as in the original code and, at the same time, control which task each subroutine call accomplishes. We replace the original subroutine body with a branch structure in which each branch contains one S_i. The execution of each branch is determined by a global control structure whose members represent the state of execution in the reversal scheme. The branches contain code for pre- and post-state transitions enclosing the respective S_i. This ensures that the transformations producing the S_i do not depend on any particular reversal scheme. The postprocessor inserts the $\mathtt{S}_i$ into a subroutine template, schematically shown in Fig. 3.8(a). The template is written in Fortran. Each

[6] specified with command line option `-i` which defaults to `ad_inline.f`

subroutine in the postprocessor Fortran input is transformed according to either a default subroutine template found in a `ad_template.f` file or in a file specified in a `!$openad XXX Template <file name>` pragma to be located in the subroutine body. The input Fortran also contains `!$openad begin replacement <`i`>` paired with pragmas `!$openad end replacement`. Each such pair delimits a code variant $\mathtt{S}_i$ and the postprocessor matched the respective identifier i (an integer) with the identifier given in the template `PLACEHOLDER_PRAGMA`.

Split reversal is the simplest static call graph reversal. We first execute the entire computation with the augmented forward code (S_2) and then follow with the adjoint (S_3). From the task pattern shown in Fig. 2.11 it is apparent that, aside from the top-level routine, there is no change to the state structure within the call tree. Therefore, there is no need for state changes within the template. Since no checkpointing is needed either, we have only two tasks: producing the tape and the adjoint run. Fig. 3.8(b) shows a simple split-mode template, see also `runTimeSupport/simple/ad_template.split.f`. The state is contained in `rev_mode`, a static Fortran90 variable, see `runTimeSupport/simple/OpenAD_rev.f90` of type `modeType` also defined in this module. In order to perform a split-mode reversal for the entire computation, a driver routine calls the top-level subroutine first in taping mode and then in adjoint mode.

Fig. 2.12 illustrates the task pattern for a joint reversal scheme that requires state changes in the template and requires more code alternatives. Fig. 3.9 shows a simplified joint mode template, see also `runTimeSupport/simple/ad_template.joi` The state transitions in the template directly relate to the pattern shown in Fig. 2.12. Each prestate change applies to the callees of the current subroutine. Since the argument store (S_4) and restore (S_6) do not contain any subroutine calls they do not need state changes. Looking at Fig. 2.12, one realizes that the callees of any subroutine executed in plain forward mode (S_1) never store the arguments (only callees of subroutines in taping mode do). This explains lines 18, 25, and 30. Furthermore, all callees of a routine currently in taping mode are not to be taped but instead run in plain forward mode, as reflected in lines 27 and 28. Joint mode in particular means that a subroutine called in taping mode (S_2) has its adjoint (S_3) executed immediately after S_2. This is facilitated by line 33, which makes the condition in line 35 true, and we execute S_3 without leaving the subroutine. Any subroutine executed in adjoint mode has its direct callees called in taping mode, which in turn triggers their respective adjoint run. This is done in lines 37–39. Finally, we have to account for sequence of callees in a subroutine; that is, when we are done with this subroutine, the next subroutine (in reverse order) needs to be adjoined. This process is triggered by calling the subroutine in taping mode, as done in lines 41–43. The respective top-level routine is called by the driver with the state structure having both `tape` and `adjoint` set to `true`.

```
 1:subroutine template()
 2:  use OpenAD_tape
 3:  use OpenAD_rev
 4:  use OpenAD_checkpoints
 5: !$TEMPLATE_PRAGMA_DECLARATIONS
 6:  type(modeType) :: orig_mode
 7:
 8:  if (rev_mode%arg_store) then
 9: ! store arguments
10: !$PLACEHOLDER_PRAGMA$ id=4
11:  end if
12:  if (rev_mode%arg_restore) then
13: ! restore arguments
14: !$PLACEHOLDER_PRAGMA$ id=6
15:  end if
16:  if (rev_mode%plain) then
17:    orig_mode=rev_mode
18:    rev_mode%arg_store=.FALSE.
19: ! run the original code
20: !$PLACEHOLDER_PRAGMA$ id=1
21:    rev_mode=orig_mode
22:  end if
23:  if (rev_mode%tape) then
24: ! run augmented forward code
25:    rev_mode%arg_store=.TRUE.
26:    rev_mode%arg_restore=.FALSE.
27:    rev_mode%plain=.TRUE.
28:    rev_mode%tape=.FALSE.
29: !$PLACEHOLDER_PRAGMA$ id=2
30:    rev_mode%arg_store=.FALSE.
31:    rev_mode%arg_restore=.FALSE.
32:    rev_mode%plain=.FALSE.
33:    rev_mode%adjoint=.TRUE.
34:  end if
35:  if (rev_mode%adjoint) then
36: ! run the adjoint code
37:    rev_mode%arg_restore=.TRUE.
38:    rev_mode%tape=.TRUE.
39:    rev_mode%adjoint=.FALSE.
40: !$PLACEHOLDER_PRAGMA$ id=3
41:    rev_mode%plain=.FALSE.
42:    rev_mode%tape=.TRUE.
43:    rev_mode%adjoint=.FALSE.
44:  end if
45:end subroutine template
```

Figure 3.9: Joint mode Fortran90 template with argument checkpointing

Chapter 4

Tool Usage

The following contains brief instructions how to obtain and use OpenAD/F. While the principal approach will remain the same, future development may introduce slight changes. The reader is encouraged to refer to the up to date instructions on the OpenAD/F website [27].

4.1 Download and Build

All components are open source and readily available for download from the HiPerSoft CVS server at Rice University. Instructions to set up for anonymous CVS access are found at `http://hipersoft.cs.rice.edu/cvs/index.html#anonymous`. The download is a 2 stage process where initially a skeleton repository is retrieved via `cvs co OpenAD`. This contains a few scripts and setup to retrieve and build all components. Then proceed as follows.

1. go into the OpenAD/F download/build environment which was retrieved as described above:
   `cd OpenAD`
2. retrieve all components from their various repositories by invoking
   `./bin/goad`
3. set the environment for building and using OpenAD/F for
   - shell/ksh/bash etc users with
     `source ./setenv.sh`
   - csh/tcsh etc. users with
     `source ./setenv.csh`
4. We assume the GNU c/c++/f77 compilers versions 3.3.x to 3.4.4[1] and GNU make (gmake) to be in your `PATH`. The complete set of all components can be build by invoking
   `make`

To update the sources to the latest version simply repeat the above steps 1 through 4 which executes CVS updates and performs a mostly incremental build.

4.2 Code Prepaparation

Pragma's etc.

4.3 Automatic Pipeline

The components of OpenAD/F transform the code in a predetermined sequence of steps, the *pipeline*. Depending on the particular problem there are certain variations to the pipeline that achieve a better performance of the generated code. The most common pipeline setups are encapsulated in a Perl script. The script is part of the skeleton environment that is used to download and build OpenAD/F and relies on the same environment setup. Therefore

[1] Open64 currently does not compile with gcc 4.x.

user starts with steps 1 and 2 from Sec. 4.1. The script is contained in the file `OpenAD/tools/openad/openad`. Invoking it with the `-h` option displays the script usage of which the mode choices are the most important.

`--bb-rev` the default mode that produces adjoint code as explained in Sec. 3.1.3.8,

`--bb` the tangent-linear mode as explained in Sec. 3.1.3.3, and

`--memopstradeoff` the tangent-linear mode with heuristics addressing data locality and operations count for the elimination sequence as explained in Sec. 3.1.3.4.

As an example we use a code assumed to be in a file called head.f and transform this code by invoking `openad head.f` which produces the adjoint version of `head.f` emitting the following messages.

```
Preprocess Fortran: 'head.f' -> 'head.pre.f'
Fortran to WHIRL: 'head.pre.f' -> 'head.pre.B'
WHIRL to XAIF: 'head.pre.B' -> 'head.xaif'
xaifBooster: 'head.xaif' -> 'head.xb.xaif'
XAIF to WHIRL: ('head.pre.B' 'head.xb.xaif') -> 'head.xb.x2w.B'
WHIRL to Fortran: 'head.xb.x2w.B' -> 'head.xb.x2w.w2f.f'
Postprocess Fortran: 'head.xb.x2w.w2f.f' -> 'head.xb.x2w.w2f.pp.f'

OpenAD: 'head.f' --> 'head.xb.x2w.w2f.pp.f'
```

For larger projects it is obviously appropriate to customize the sequence by adding the steps outlined in Sec. 4.4 to a `Makefile`.

4.4 Manual Pipeline

The following illustrates the pipeline that could be implemented in a `Makefile` where we use successive file name postfixes to retain the transformation results at each stage.

1	Canonicalization, see Sec. 3.2.1	
	in (Fortran)	`head.f`
	out (Fortran)	`head.canon.f`
	command	`python canon.v1.py head.f > head.canon.f`
	The script is located at `OpenADFortTk/tools/canonicalize/canon.v1.py`.	
2	Parsing, see Sec. 3.2.2	
	in (Fortran)	`head.canon.f`
	out (whirl)	`head.canon.B`
	command	`mfef90 -F -ffixed head.canon.f`
	The binary is located at `Open64/osprey1.0/targ_ia32_ia64_linux/crayf90/sgi/mfef90`. The `-F` flag enables C preprocessing if needed, the `-ffixed` indicates fixed format input.	
3	Translating to XAIF, see Sec. 3.2.3	
	in (whirl)	`head.canon.B`
	out (XAIF)	`head.canon.xaif`
	command	`whirl2xaif -o head.canon.xaif head.canon.B`
	The binary is located at `OpenADFortTk/OpenADFortTk-i686-Linux/bin/whirl2xaif`. The `-o` flag allows to specify an output file, the default is `stdout`.	
4	Transformation, see Sec. 3.1.3	
	in (XAIF)	`head.canon.xaif`
	out (XAIF)	`head.canon.xb.xaif`
	command	`<trans> -i head.canon.xaif -c <iCat> -o head.canon.xb.xaif`
	The binary `<trans>` is one of `xaifBooster/algorithms/BasicBlockPreaccumulation/test/t`, see Sec. 3.1.3.3 `xaifBooster/algorithms/MemOpsTradeoffPreaccumulation/test/t`, see Sec. 3.1.3.4 `xaifBooster/algorithms/BasicBlockPreaccumulationReverse/test/t`, see Sec. 3.1.3.8 Each of the binaries prints a self explanatory usage message when invoked without any arguments. The intrinsics catalogue `<iCat>` can be found in `xaif/schema/examples/inlinable_intrinsics.xaif`	
5	Translating to whirl, see Sec. 3.2.3	
	in (whirl and XAIF)	`head.canon.B head.canon.xb.xaif`
	out (whirl)	`head.canon.xb.x2w.B`
	command	`xaif2whirl head.canon.B head.canon.xb.xaif`

	The binary is located at `OpenADFortTk/OpenADFortTk-i686-Linux/bin/xaif2whirl`. The optional `--structured` flag indicates structured control flow, as required for the control flow reversal employed by the adjoint code transformation, see Sec. 2.3.	
6	Unparse to Fortran, see Sec. 3.2.2	
	in (whirl)	`head.canon.xb.x2w.B head.canon.xb.xaif`
	out (Fortran)	`head.canon.xb.x2w.w2f.f`
	command	`whirl2f head.canon.xb.B`
	The binary is located at `Open64/osprey1.0/targ_ia32_ia64_linux/whirl2f/whirl2f`. It internally loads `whirl2f_be` located in the same directory.	
7	Postprocessing, see Sec. 3.2.4	
	in (Fortran)	`head.canon.xb.x2w.w2f.f`
	out (Fortran)	`head.canon.xb.x2w.w2f.pp.f`
	command	`perl multi-pp.pl head.f head.canon.xb.x2w.w2f.f`
	The script is located at `OpenADFortTk/tools/multi-process/multi-pp.pl`. For tangent linear models the postprocessor requires the command line flag `-f` while for inlining the default file name `ad_inline.f` can be changed with the `-i` flag and for template expansion the default template file name `ad_template.f` can be changed with the `-t` flag.	

The binaries used in steps 3, 5, and 6 all rely on the `libbe.so` Open64 backend library. It is located in `Open64/osprey1.0/targ_ia32_ia64_linux/whirl2f/`.

4.5 Compiling and Linking

All Fortran produced by `whirl2f` needs definitions for `kind` variables that occur within the `whirl2f`-generated code. These definitions can be found in `runTimeSupport/all/w2f__types.f90`. The code produced by transformation pipeline requires implementations (OpenAD/F supplies samples) for the following aspects.

- active type (see `runTimeSupport/simple/OpenAD_active.f90`)
- checkpointing (only for adjoint models, see `runTimeSupport/simple/OpenAD_checkpoints.f90`)
- taping (only for adjoint models, see `runTimeSupport/simple/OpenAD_tape.f90`)
- state for call graph reversal (only for adjoint models, see `runTimeSupport/simple/OpenAD_rev.f90`)

The compilation order for these various modules follows exactly the order given here. The provided sample implementations work with the subroutine inlining and templates found in the same directory.

Finally, we need a *driver* that invokes the transformed routines and seeds and retrieves the derivatives. Examples for such drivers can be found in Sec. 5.

Chapter 5

Application

This applications section intends to augment the explanations given so far. First we use a toy example to show how to embed the transformed code into a driver. The following sections illustrate practical concerns for real life applications.

5.1 Toy Example

Consider a toy example code Fig. 5.1(a) where we already inserted the dependent and independent declararations, see Sec. 3.2.2. Transformed into a tangent linear model `head` turns into a subroutine that has active parameters and the calling code, i.e. the driver, is written to seed (`x%d`) and extract (`y%d`) the derivatives according to (2.1). A very simple driver for the tangent-linear model is Due to the simplicity of the example, the adjoint model version does not provide much insight other than the reversal of seeding (`y%d`) and extraction (`x%d`)of the derivatives, see Fig. 5.3.

5.2 Shallow Water Model

In this section we will use a practical application to highlight advanced aspects arising for more complicated applications. The model is a time stepping scheme which eventually computes a scalar valued cost function. We generate an adjoint model to compute the gradient.

5.2.1 Collect and Prepare Source Files

The entire model consists of many subroutines distributed over various source files and the existing build sequence involves C preprocessing. To perform the static code analysis as explained in Sec. 3.1.1 all code that takes part in computation of the model has to be visible to the tool which means it has to be concatenated into a single file. It is possible to do this for all source files of the model but in many cases this will include code for ancillary tasks such as diagnostics and data processing not directly related to the model computation. Often it is better to filter out such ancillary code.

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
c$openad INDEPENDENT(x)
  y=sin(x*x)
c$openad DEPENDENT(y)
end subroutine
```

(a)

```
SUBROUTINE head(X, Y)
use w2f__types
use OpenAD_active
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y

! function body etc...

END SUBROUTINE
```

(b)

Figure 5.1: A toy example(a) and the modified signature for the tangent-linear model(b)

```
program driver
  use OpenAD_active
  external head
  type(active):: x, y
  read *, x%v
  x%d=1.0
  call head(x,y)
  write (*,*) "J(1,1)=",y%d
end program driver
```

(a)

```
prompt> ./a.out
1.0
 J(1,1)=   1.0806046117362795
prompt> ./a.out
2.0
 J(1,1)=  -2.6145744834544478
prompt>
```

(b)

Figure 5.2: A toy example tangent-linear driver(a) and output(b)

```
program driver
  use OpenAD_active
  use OpenAD_rev
  external head
  type(active):: x, y
  read *, x%v
  y%d=1.0
  our_rev_mode%tape=.TRUE.
  our_rev_mode%adjoint=.TRUE.
  call head(x,y)
  write (*,*) "J(1,1)=",x%d
end program driver
```

(a)

```
prompt> ./a.out
1.0
 J(1,1)=   1.0806046117362795
prompt> ./a.out
2.0
 J(1,1)=  -2.6145744834544478
prompt>
```

(b)

Figure 5.3: A toy example adjoint driver(a) and output(b)

- The static code analysis and subsequently the code transformation has to make conservative assumptions to ensure correctness, e.g. for alias analysis this means an overestimate of the memory locations that can alias each other. One of the effects of these potential aliases are additional assignments in the generated code which lead to a less efficient adjoint. Including ancillary sources may cause more conservative assumptions to be made and therefore lead to an unnecessary loss in efficiency.
- While the numerical portions frequently have been tuned and made platform neutral the ancillary portions often are platform dependent and may contain Fortran constructs that the language dependent components handle improperly or not at all. While all tools in principle strive for complete language coverage the limited development resources can often not be spared to cover infrequently used language aspects and rather need to be focused on features that actually benefit capabilities and efficiency for a wide range of applications.

As for all AD tools in existence today the above concerns also apply to OpenAD/F and users are kindly asked to keep them in mind when preparing the source code.

Sec. 5.1 indicates the need for a modification to the code that drives the model computation to at least preform the seeding and extraction of the derivatives. The easiest approach to organize the driver is to identify (or create) a top level subroutine that computes the model with a single call. This routine and all code it requires to compute the model become the contents of the single file to be processed by the tool pipeline. The independent and dependent variables should be identified in the top level routine.

5.2.2 Orchestrate a Reversal and Checkpointing Scheme

Joint and split reversal, see Sec. 2.4 are two special cases of a large variety of reversal schemes. The model here involved a time stepping scheme controlled by a main loop. OpenAD/F supports automatic detection of the data set to be checkpointed at a subroutine level. To use this feature the loop body is encapsulated into a inner loop subroutine `I`. To realize a nested checkpointing scheme we select a number `i` for the inner checkpoints, divide the original loop bound `t` by `i` and encapsulate the inner loop into an outer loop subroutine `O` schematically shown in Fig. 5.4 which is invoked `o` times[1] Now we can describe the reversal scheme with the call tree shown in Fig. 5.5. The state changes can be encapsulated in four templates, one joint mode template for `top` and all its callees except `O`, one for all callees of `I` and one each for `O` and `I`. The collection of downloadable test problems contains the model and the four subroutine templates. Fig. 5.4(b) shows the `cost` subroutine called from `I` as well as from `top`. However, according to Fig. 5.4 we would need two versions of `cost`, one that as callee of `top` is reversed in joint mode and one as callee of `I` is reversed in split mode. In order to maintain the static reversal approach[2] one needs to duplicate

[1] for simplicity disregarding remainders `o=t/i`.

[2] A dynamic reversal scheme is forthcoming.

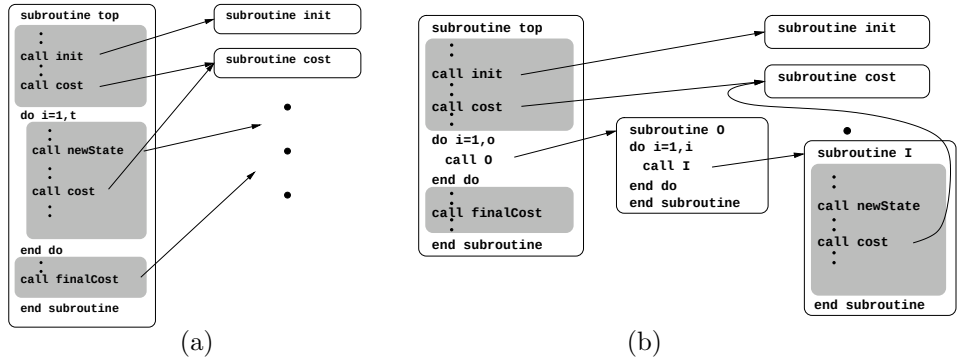


Figure 5.4: Modification of the original code (a) to allow 2 checkpointing levels (b)

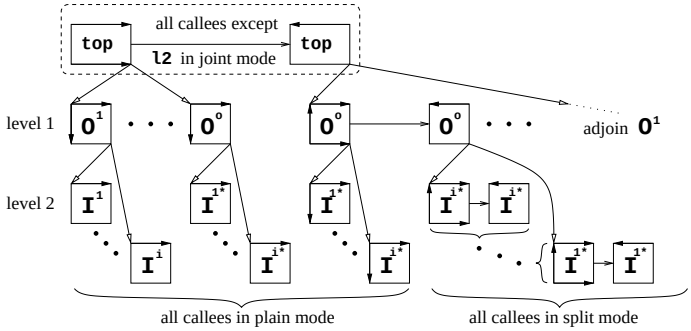


Figure 5.5: Checkpointing scheme, the `.*` indicating `.+(o-1)i`

`cost`.

5.2.3 File I/O and Simple Loops

The model code uses both the NetCDF library as well as the built in Fortran I/O during the initialization and output of results. Because in the model computation no intermediate values are written and read during the model computation there is no loss of dependency information. However, the I/O can lead to problems, for instance when an activated array is initialized. The prevalent lack of type checking in Fortran may lead to setting the first half of the `%v` and `%d` values instead of setting all of the `%v` values. This is a well known consequence of the active type implementation. While one could argue that the code should be generated to avoid reading or writing the derivative information this is not always the actually desired behavior, in particular not if one reads or writes active intermediate variables. A simple and effective measure to circumvent this problem is let the initialization remain an external routine in which case OpenAD/F will insert conversion routines for external subroutine parameters that are active at the call site. It should be noted that this approach does not work when instead of passing a parameter the external routine refers to active global variables.

Early tests showed a considerable amount of runtime and memory spent on taping array indices used in loops. The *simple* loop concept introduced in Sec. 3.1.3.6 is designed to eliminate much of this overhead. Not all loops within the given model code satisfy the conditions so as an additional step throughout the model code we identified the conformant loop constructs to the tool using the simple loop pragma. The resulting efficiency gain was about a factor 4 in run time and more than a factor 10 in memory use.

5.2.4 Results

Fig. 5.6 shows as an example output a map of sensitivities of zonal volume transport through the Drake Passage to changes in bottom topography everywhere in a barotropic ocean model computed from the shallow water code by P. Heimbach. The adjoint model generated with the current version of OpenAD/F applied to the shallow water code achieves a run time that is only about 8 times that of plain model computation. We expect the ongoing development of OpenAD/F, see also Sec. 6 to yield further efficiency gains.

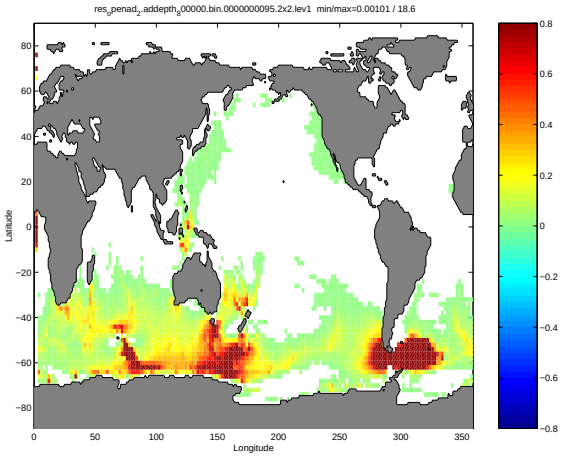


Figure 5.6: Sensitivity (gradient) map for 2×2 degree resolution

5.3 A Second Order Example

candidate for removal, unless I get the fix in for the structure stuff

Chapter 6

Summary and Future Work

OpenAD/F is an AD tool built on a language independent infrastructure with well-separated components. It allows developers to focus on various aspects of source-to-source transformation AD, including parsing and unparsing of different programming languages, data and control flow analysis, and (semantic) transformation algorithms. The components have well defined interfaces and intermediate stages are retained as either Fortran or XML sources.

OpenAD/F allows users a great amount of flexibility in the use of the code transformation and permits interventions at various stages of the transformation process. We would like to emphasize the fact that for large scale applications the efficiency of checkpointing and taping can be improved merely by modifying the implementation of the run time support, the template and inlining code. They are not conceived to be just static deliverables of OpenAD/F but rather are part of the interface accessible to the user. It is not the intention to stop with a few prepackaged solutions as one would expect from a monolithic, black-box tool. True to the nature of an open source design, the interface is instead conceived as a wide playground for experimentation and improvement. Expanding on the schematic depiction of the tool workings in Fig. 1.1 we want to highlight the options to modify the transformation and the tool itself in Fig. 6.1 at different levels of complexity reaching from the casual user to actual coding work in the tool's components. As part of using OpenAD/F for different applications we see a growing number of variations to the transformation and run time support implementations available to the user.

Aside from the plain AD tool aspect the intention of the underlying OpenAD framework is to provide the AD community with an open, extensible, and easy-to-use platform for research and development that can be applied across programming languages. Tools that have a closer coupling with a language-specific, internal representation have the potential to make the exploitation of certain language features easier. Consequently we do not expect OpenAD/F to obsolete existing source transformation tools such as the differentiation-enabled NAG Fortran 95 compiler, [1] TAF,[2] or TAPENADE.[3] Rather it is to complement these tools by providing well-defined APIs to an open internal representation that can be used by a large number of AD developers. Users of AD technology will benefit from the expected variety of combinations of front-ends and algorithms that is made possible by OpenAD/F.

As with any software project there is ample room for improvement. The robustness of the tool, in particular the coverage of some specific language features, often is of concern to first time users. While robustness is not to be disregarded, it is clearly not a research subject and as such cannot be made the major objective of a development project in an academic setting. Robustness issues affect mostly the language dependent components and the contributing parties undertake a considerable effort to address concerns common to many applications. Many issues specific to a particular input code can be addressed by minor adjustments which often happen to reflect good coding practices anyway. Take for example a change away from `goto - label` to a well structured control flow. While we plan to implement code that handles unstructured control flow at some point, the corresponding adjoint will always be less efficient than the respective structured equivalent and an automatic transformation to structured control flow is somewhat beyond the scope of an AD tool.

We are concerned with changes that affect many applications and yield improved efficiency of the adjoint code. Currently the most important items on the development list are the support for vector intrinsics and the handling of allocation/deallocation cycles during the model computation for the generation of an adjoint model. Because the tool provides a variety of options to the user we are also working on collecting data for efficiency estimates that permit an informed choice between the code transformation options. Ongoing research in AD algorithms, in particular dynamic call graph reversal, more efficient control flow reversal and improved elimination techniques in the computational

[1] `http://www.nag.co.uk/nagware/research/ad_overview.asp`
[2] `http://www.FastOpt.de`
[3] `http://tapenade.inria.fr:8080/tapenade/index.jsp`

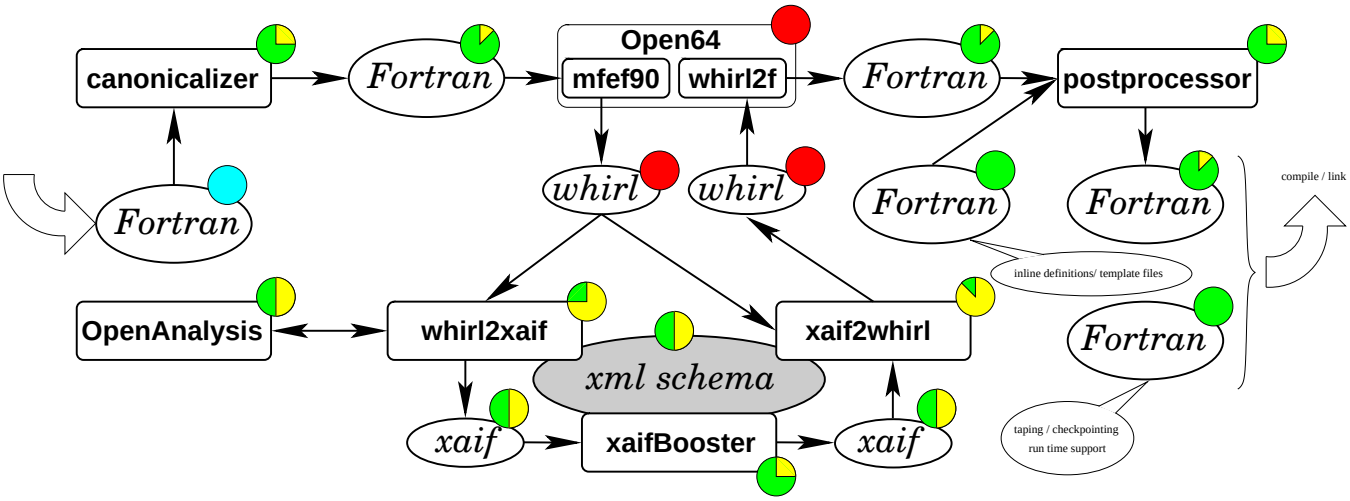


Figure 6.1: Levels of complexity for modifications

graphs will be incorporated into OpenAD.

Appendix

`Makefile`	the top level `Makefile`
`utils/`	utility classes (debugging, generic traversal, etc.)
`tools/`	code generator supporting XAIF parser
`boostWrapper/`	wrapper classes for the boost graph library
`system/`	all basic data structures, XAIF (un)parsing, Sec. 3.1.3.1
`algorithms/`	see the subdirectories below
`CodeReplacement`	support library for subroutine templates
`CrossCountryInterface`	support library for elimination strategies, Sec. 3.1.3.3
`DerivativePropagator`	support library for Jacobian vector products
`InlinableXMLRepresentation`	support library for inlinable subroutine calls
`Linearization`	Linearization transformation, Sec. 3.1.3.2
`BasicBlockPreaccumulation`	elimination with angel and preaccumulation at the basicblock level, Sec. 3.1.3.3
`MemOpsTradeoffPreaccumulation`	as above but with different heuristics than angel
`ControlFlowReversal`	control flow graph reversal
`BasicBlockPreaccumulationReverse`	adjoint code
`BasicBlockPreaccumulationTape`	taping code supporting adjoint, Sec. 3.1.3.7
`BasicBlockPreaccumulationTapeAdjoint`	reverse sweep portion supporting adjoint,Sec. 3.1.3.7

Table 6.1: Directory structure in xaifBooster

Bibliography

[1] AD community website. `http://www.autodiff.org`.

[2] AD Nested Graph Elimination Library (angel). `http://angellib.sourceforge.net`.

[3] ADIC. `http://www.mcs.anl.gov/adicserver`.

[4] Adjoint Compiler Technology & Standards (ACTS). `http://www.autodiff.org/ACTS`.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

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

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

[8] Boost. `http://www.boost.org`.

[9] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory and Implementations*, volume 50 of *LNCS*, New York, 2006. Springer.

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

[11] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.

[12] Edison Design Group (EDG). `http://www.edg.com`.

[13] Extensible Markup Language (XML). `http://www.w3.org/XML`.

[14] GNU C++ Standard Library. `http://gcc.gnu.org/libstdc++`.

[15] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.

[16] A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[11]*, pages 126–135, 1991.

[17] L. Hascoët, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Comp. Syst.*, 21(8):1401–1417, 2005.

[18] P. Hovland, U. Naumann, and B. Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications (SEA 2002)*, pages 530–538. ACTA Press, 2002.

[19] P. Hovland and B. Norris. Users' guide to ADIC 1.1. Technical Memorandum ANL/MCS-TM-225, Mathematical and Computer Science Division, Argonne National Laboratory, 2001.

[20] U. Naumann. Elimination techniques for cheap Jacobians. In G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors, *Automatic Differentiation of Algorithms – From Simulation to Optimization*, pages 247–253, New York, 2002. Springer.

[21] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 2003. Published online at `http://www.springerlink.com`.

[22] U. Naumann and P. Gottschling. Simulated annealing for optimal pivot selection in Jacobian accumulation. In A. Albrecht and K. Steinhöfel, editors, *Stochastic Algorithms: Foundations and Applications*, volume 2827 of *LNCS*, pages 83–97. Springer, 2003.

[23] U. Naumann and J. Utke. Source templates for the automatic generation of adjoint code through static call graph reversal. In V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2005*, volume 3514 of *LNCS*, pages 338–346, Berlin, 2005. Springer. also as ANL preprint ANL/MCS-P1226-0205.

[24] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, 2004. IEEE Computer Society.

[25] Network Enhance Optimization Server (NEOS). `http://www-neos.mcs.anl.gov/`.

[26] Open64. `http://www.hipersoft.rice.edu/open64`.

[27] OpenAD. `http://www.mcs.anl.gov/OpenAD`.

[28] OpenAnalysis. `http://www-unix.mcs.anl.gov/OpenAnalysisWiki/moin.cgi`.

[29] OpenMP. `http://www.openmp.org`.

[30] ROSE. `http://www.llnl.gov/CASC/rose/`.

[31] J. Utke. Flattening basic blocks. In *[9]*, pages 121–133, 2006.

[32] J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the interprocedural flow of control in adjoint computations. *Journal of Systems and Software*. to appear in a special edition.

[33] J. Utke and U. Naumann. Software technological issues in automatizing the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications (SEA 2003)*, pages 417–422, Anaheim, Calgary, Zurich, 2003. ACTA Press.

[34] J. Utke and U. Naumann. Separating language dependent and independent tasks for the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications (SEA 2004)*, pages 552–558, Anaheim, Calgary, Zurich, 2004. ACTA Press.

[35] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.

[36] XAIF. `http://www.mcs.anl.gov/xaif`.

[37] Xerces C++ XML parser. `http://xml.apache.org/xerces-c`.