RWTH Aachen

Department of Computer Science
Technical Report

OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-14

RWTH Aachen · Department of Computer Science · June 2007

OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

Jean Utke[1], Uwe Naumann[2], Mike Fagan, Nathan Tallent[3], Michelle Strout[4] and Patrick Heimbach, Chris Hill, Carl Wunsch[5]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
[2] Dept. of Computer Science, RWTH Aachen University, Aachen, Germany
[3] Dept. of Computer Science, Rice University, Houston, TX, USA
[4] Dept. of Computer Science, Colorado State University, Fort Collins, CO, USA
[5] Dept. of Earth, Atmospheric, and Planetary Sciences, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract. The OpenAD/F tool allows the evaluation of derivatives of functions defined by a Fortran program. The derivative evaluation is performed by a Fortran code resulting from the analysis and transformation of the original program that defines the function of interest. OpenAD/F has been designed with a particular emphasis on modularity, flexibility, and the use of open source components. While the code transformation follows the basic principles of automatic differentiation, the tool implements new algorithmic approaches at various levels, for example, for basic block preaccumulation and call graph reversal. Unlike most other automatic differentiation tools, OpenAD/F uses components provided by the OpenAD framework, which supports a comparatively easy extension of the code transformations in a language-independent fashion. It uses code analysis results implemented in the OpenAnalysis component. The interface to the language-independent transformation engine is an XML-based format, specified through an XML schema. The implemented transformation algorithms allow efficient derivative computations using locally optimized cross-country sequences of vertex, edge, and face elimination steps. Specifically, for the generation of adjoint codes, OpenAD/F supports various code reversal schemes with hierarchical checkpointing at the subroutine level. As an example from geophysical fluid dynamics a nonlinear time-dependent scalable, yet simple, barotropic ocean model is considered. OpenAD/F's reverse mode is applied to compute sensitivities of some of the model's transport properties with respect to gridded fields such as bottom topography as independent (control) variables.

1 Introduction

The basic principles of automatic differentiation (AD) (see also Section 2) have been known for several decades [36], 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 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 A Large-Scale Example Application

One of the most demanding applications of AD is the computation of gradients for sensitivity analysis and state estimation (sometimes referred to as data assimilation) on large-scale models in oceanography and climate research. This application clearly falls in the category requiring experienced users.

To demonstrate what can be achieved today with the gradient computed in this manner, and to expose how AD has made a specific large-scale optimization problem practical, we elaborate on this application. It takes advantage of the reverse mode of AD (see Section 2), which yields the transpose of the tangent linear model, the adjoint model. It has long been recognized that for scalar-valued objective functions (such as energy, property transports, property content, or least-squares model data misfits) the sensitivity or gradient of such a determining function with respect to a large suite of n independent (or control) variables can be calculated efficiently with a single adjoint integration, whereas as many as n separate perturbation integrations would be required with the original forward model. In the context of ocean and climate modeling the latter computation quickly becomes prohibitive (n being of the order 10^4 to 10^8). On the other hand, the generation of an adjoint model to a given fully fledged nonlinear time-dependent general circulation model (GCM) is a major undertaking, similar in complexity to the GCM development itself.

A major step forward was the implementation at MIT of a new ocean general circulation model [18] and simultaneous development of the AD tool TAMC [10] to fully support the GCM's coding structures and to render efficient adjoint code in the context of a nonlinear time-dependent problem. This led to the first sensitivity application based on fully AD-derived adjoint code in which, for the first time, one could comprehensively address the question of how North Atlantic heat transport (a scalar-valued objective function) depended on changes, separately, in every element of gridded two- or three-dimensional fields, such as sea-surface temperature *everywhere in the domain* (i.e., a 10^4-dimensional control space) [17]. At the same time, the Estimating the Circulation and Climate of the Ocean (ECCO) Consortium set out to derive an ocean state estimate (OSE) covering the period of the World Ocean Circulation Experiment and the beginning of the satellite altimetric record (1992) [28]. The availability of the AD-derived adjoint made practical the application of the method of Lagrange multipliers in a gradient-based iterative optimization that brings the numerical model to consistency with a plethora of observations (today on the order of 100 million) by varying elements of a 10^8-dimensional control space [37]. Since then, both underlying code and the type of observations available have evolved significantly. AD has permitted maintenance of an up-to-date adjoint code for the forward model undergoing vigorous model development [14]. OSE is today developing in a number of different directions, (see, e.g., the overview by [38]), but challenges remain, among others, with limitations in computational resources and with limitations in the AD tools flexibility, code handling, derivation of efficient code, and trade-offs between efficiency and approximations made to the exactness of *tangent* linearity. Despite these limitations it has come as a surprise to ocean and climate modelers that the potential of the reverse mode of AD has not found much wider application in the large-scale optimization community.

1.2 Motivation

An evaluation of the available AD tools revealed 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 view point of AD tool *users*, 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 changes in 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 an opportunity to check and modify the results.

From the view point of AD tool *developers*, 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 compiler-like internal representation, an engine that supports 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.

These considerations motivated the Adjoint Compiler Technology & Standards [1] 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.

1.3 Overview

OpenAD/F is the Fortran incarnation of the OpenAD framework [26]. OpenAD/F has a modular design, illustrated in Figure 1. Given as input a Fortran program $\boldsymbol{f}$, the Open64 (see `www.hipersoft.rice.edu/open64`) front-end performs

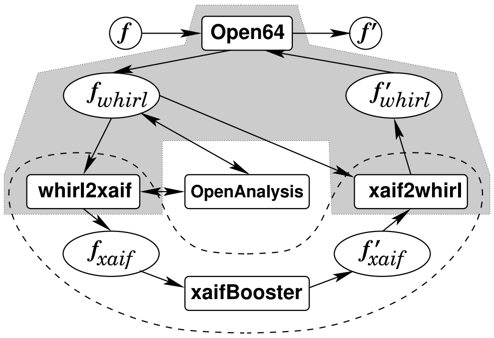


Fig. 1. Simplified overview of the OpenAD/F components and the pipeline of transformation steps. The input is a numerical model $\boldsymbol{f}$ given as a Fortran program. The Open64 compiler front-end parses it into whirl, which is translated into the language-independent XML-based Xaif format. The Xaif representation then is translated back into a Fortran program $\boldsymbol{f}'$ that computes derivatives.

a lexical, syntactic, and semantic analysis and produces an intermediate representation of $\boldsymbol{f}$, here denoted by $\boldsymbol{f}_{whirl}$, in the whirl format. OpenAnalysis is used to build call and control flow graphs and perform code analyses such as alias, activity, and 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, represented in Xaif as $\boldsymbol{f}'_{xaif}$. The information in $\boldsymbol{f}'_{xaif}$ and the original $\boldsymbol{f}_{whirl}$ is used by xaif2whirl to construct a whirl representation $\boldsymbol{f}'_{whirl}$ of the differentiated code. The unparser of Open64 transforms $\boldsymbol{f}'_{whirl}$ into Fortran, thus completing the semantic transformation of a program $\boldsymbol{f}$ into a differentiated program $\boldsymbol{f}'$. The shaded area encloses the language-specific front-end that can potentially be replaced by front-ends for languages other than Fortran. The C/C++ tool ADIC v2.0 (see [2]) is also based on OpenAD but is not discussed here.

Our focus is on the design rationale and major features of OpenAD/F. Technical details are left to the user manual [35]. In Section 2 we cover the concepts of AD relevant to the description of OpenAD, in Section 3 the components that make up OpenAD/F, and in Section 4 tool usage. Two applications in Section 5 illustrate the tool usage. We conclude in Section 6 with future developments.

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 [11]. The interested reader should also consult the proceedings of AD conferences [8, 5, 7, 6]. We consider first consecutive sequences of elemental numerical operations in Section 2.1, then their control flow context within a subroutine in Section 2.2, and the entire program consisting of several subroutines in a call graph in Section 2.3. The given numerical model is a vector-valued function

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : I\!R^n \mapsto I\!R^m$$

implemented as a computer program, and the objective is to compute derivatives such as products of Jacobians $\boldsymbol{J} = \left[\frac{\partial y_j}{\partial x_i}\right]$ with *seed* matrices $\boldsymbol{S}$:

$$\boldsymbol{JS} \quad \text{and} \quad \boldsymbol{J}^T\boldsymbol{S} \quad . \tag{1}$$

2.1 Elemental Operations in Computational Graphs

Without loss of generality 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

$$v_j = \phi_j(\ldots, v_i, \ldots) \quad . \tag{2}$$

The $v_i \in V$ represent the vertices 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 assignment $v_j = \phi_j(\ldots, v_i, \ldots)$. The elemental operations ϕ_j are assumed to be differentiable on open subdomains. Each edge $(i,j) \in E$ is labeled with a 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 partials of the elementals ϕ, that is, multiplications and additions of the c_{ji}.

Using a specific numbering scheme for the vertices v_i we presume q intermediate values $v_j = \phi_j(\ldots, v_i, \ldots)$ for $j = 1, \ldots, q+m$ and $i = 1-n, \ldots, q$, $j > i$. The n *independent* variables $x_1, \ldots, x_n$ correspond to $v_{1-n}, \ldots, v_0$. We consider the computation of derivatives of the *dependent* variables $y_1, \ldots, y_m$ represented by m variables $v_{q+1}, \ldots, v_{q+m}$ with respect to the independents. The dependency $v_i \prec v_j$ implies $i < j$. Using the edge labels $c_{ji} = \frac{\partial v_j}{\partial v_i}$, the *forward mode* of AD propagates directional derivatives as

$$\dot{v}_j = \sum_i c_{ji}\dot{v}_i \quad \text{for} \quad j = 1, \ldots, q+m. \tag{3}$$

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:

$$\overline{v}_i = \sum_j c_{ji}\overline{v}_j \quad \text{for} \quad j = q, \ldots, 1-n. \tag{4}$$

In practice, the sum in (4) is often split into individual increments associated with each statement in which v_i occurs as an argument $\overline{v}_i = \overline{v}_i + \overline{v}_j \cdot c_{ji}$. Equations (3) and (4) can be used to accumulate the (local) Jacobian $\boldsymbol{J}(G)$ of G; see also Section 2.2.

In a source transformation context we want to generate code for *all* possible inputs $\boldsymbol{x}$. In general, there is no single representative G because of the presence of control flow. Instead we simply consider the statements contained in a basic block as a section of code below the granularity of control flow and construct our computational (sub)graph for a basic block.

2.2 Elimination Methods and Preaccumulation

Let $\boldsymbol{f}$ represent a single basic block for which we compute a local Jacobian (preaccumulation). For notational simplicity and w.l.o.g. we assume that the dependent

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

Fig. 2. Code example of a sequence of assignment statements that can form a basic block. Here `x(1)` and `x(2)` are the independent, `y(1)` and `y(2)` the dependent variables.

variables are mutually independent (can always be enforced by introducing auxiliary assignments). Consider the small example in Figure 2. Expressing it in terms of results of elemental operations ϕ assigned to unique intermediate variables v, we have

$$\begin{aligned} &v_1 = v_{-1} + v_0;\ v_2 = \sin(v_0);\ v_3 = v_1 + v_2;\ v_4 = v_1 * v_3; \\ &v_5 = \sqrt{v_3};\ v_6 = \cos(v_4);\ v_7 = -v_5 \quad . \end{aligned} \tag{5}$$

In OpenAD/F this modified representation is created as part of the linearization transformation; see Section 3.1. In Figure 3(a) we show the computational graph G for this representation. The edge labels c_{ji} are the local partial derivatives, for

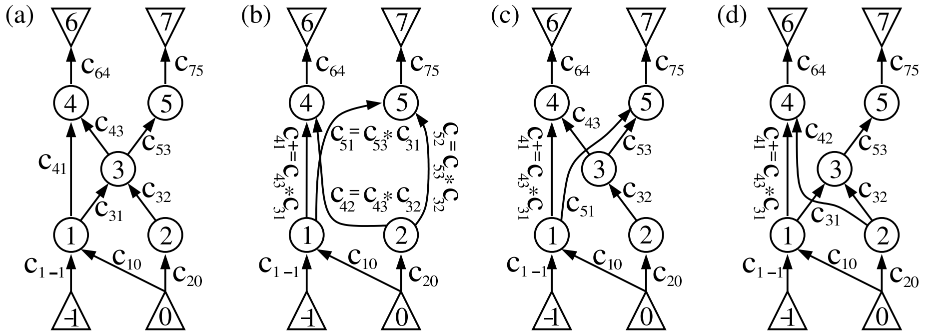


Fig. 3. (a) Computational graph G for the representation in equation (5). Elimination steps are categorized as follows: (b) eliminate vertex 3 from G, (c) front eliminate edge $(1,3)$ from G with $c_{51} = c_{53} * c_{31}$, and (d) back eliminate edge $(3,4)$ from G with $c_{42} = c_{43} * c_{32}$

example, $c_{64} = -\sin(v_4)$. With the tool, this graph is generated as part of the algorithm described in Section 3.1. 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 [12]. An edge (i,k) with $i < j$ and $j < k$ is labeled with $c_{ki} + c_{kj} \cdot c_{ji}$ if it existed before the elimination of j (absorption). Otherwise, (i,k) is generated (fill-in) and labeled with $c_{kj} \cdot c_{ji}$ The vertex j is removed from G together with all incident edges. Figure 3(b) shows the result of eliminating vertex 3 from the graph in Figure 3(a).

An edge (i,j) is *front eliminated* by connecting i with all successors of j, followed by removing (i,j) [19]. The corresponding structural modifications of G in Figure 3(a) are shown in Figure 3(c) for front elimination of $(1,3)$ together with the new edge labels. 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 (see Figure 3(d)) is symmetric to front elimination.

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

$$c_{ki} = c_{ji} * c_{kj} \ \ (+c_{ki}) \quad , \tag{6}$$

where the additions in parentheses take place only in the case of absorption; 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 (additions are not necessarily directly coupled) 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 $\mathcal{G}$. A face elimination is the elimination of an edge in $\mathcal{G}$. The result may be absorbed, as with vertex and edge eliminations, or may generate fill-in. A complete face elimination sequence yields a tripartite directed line graph that can be transformed back into the bipartite graph representing the Jacobian $\boldsymbol{f}'$. Any G can be transformed into the corresponding $\mathcal{G}$, but a back transformation generally is not possible once face elimination steps have been applied. A detailed

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

Fig. 4. Pseudo code for computing $\boldsymbol{f}$ from Figure 2 as done in (5) together with the computation of the c_{ji} for each v_j.

description of face elimination is given in [20]. In OpenAD all these eliminations are implemented in the algorithms described in Section 3.1. The accumulation of a Jacobian with a minimal arithmetic complexity has only recently been shown to be NP-complete [21].

In the context of source transformations the operations (6) are expressed as code (the Jacobian accumulation code). For our example code from Figure 2 the code computing the local partials in conjunction with the function value is shown in Figure 4 (for readability we write the c_{ji} with commas). In OpenAD/F the operations in Figure 4 are generated by the linearization algorithm discussed in Section 3.1. The operations induced by the eliminations on the graph can be expressed in terms of the edge labels c_{ji}. For our example, a forward vertex elimination in G (Figure 3), leads to the Jacobian accumulation code shown in Figure 5. In the tool these operations are generated by the preaccumulation algorithms (see Section 3.1).

v_1: $c_{3,-1} = c_{3,1} * c_{1,-1}$; $c_{3,0} = c_{3,1} * c_{1,0}$; $c_{4,-1} = c_{4,1} * c_{1,-1}$;
$c_{4,0} = c_{4,1} * c_{1,0}$;
v_2: $c_{3,0} = c_{3,2} * c_{2,0} + c_{3,0}$;
v_3: $c_{4,-1} = c_{4,3} * c_{3,-1} + c_{4,-1}$; $c_{4,0} = c_{4,3} * c_{3,0} + c_{4,0}$; $c_{5,-1} = c_{5,3} * c_{3,-1}$;
$c_{5,0} = c_{5,3} * c_{3,0}$;
v_4: $c_{6,-1} = c_{6,4} * c_{4,-1}$; $c_{6,0} = c_{6,4} * c_{4,0}$;
v_5: $c_{7,-1} = c_{7,5} * c_{5,-1}$; $c_{7,0} = c_{7,5} * c_{5,0}$;

Fig. 5. Pseudo code for eliminations of $v_1 \ldots v_5$ for (5) in forward order.

2.3 Control Flow Reversal and Taping

The code for $\boldsymbol{f}$ generally contains control flow constructs. Therefore, in general, no single computational graph G represents the computation of $\boldsymbol{f}$ for all possible values of $\boldsymbol{x}$. Section 2.1 considers computational graphs constructed from the contents of a basic block. In the example shown in Figure 6 we put the basic block code shown in Figure 2 into control flow context. A representation of the

```
 1  y(k) = sin(x(1)*x(2))
 2  k = k+1
 3  if(mod(k,2) .eq. 1) then
 4    y(k) = 2*y(k-1)
 5  else
 6    do i=1,k
 7      t1 = x(1)+x(2) }
 8      t2 = t1+sin(x(1))
 9      x(1) = cos(t1*t2)
10      x(2) = -sqrt(t2)
11    end do
12  end if
13  y(k) = y(k)+x(1)*x(2)
```

Fig. 6. Toy example code with control flow. Assignment statements are contained in basic blocks, B(2) (lines 1–2), B(4) (line 4), B(6) (lines 7–10), and B(9) (line 13); see also Figure 7(a). The sequence of assignment statements from Figure 2 forms B(6) except that instead of assigning `y(1)` and `y(2)` we overwrite `x(1)` and `x(2)`.

control flow graph (CFG) [3] resulting from the code in Figure 6 is depicted in Figure 7(a). All assignment statements are contained in basic blocks. Because the loop body B(6) is executed `k` times, the additional (compile-time) effort of optimizing the derivative code by optimizing the elimination sequence (as illustrated in Section 2.2) is justified. For a sequence of l basic blocks that are part of a path through the CFG for a particular value of $\boldsymbol{x}$, the equations (3) and (4) can be generalized as follows:

$$\dot{\boldsymbol{y}}_j = \boldsymbol{J}_j \dot{\boldsymbol{x}}_j \quad \text{for} \quad j = 1, \ldots, l \tag{7}$$

and

$$\bar{\boldsymbol{x}}_j = \boldsymbol{J}_j^T \bar{\boldsymbol{y}}_j \quad \text{for} \quad j = l, \ldots, 1 \quad , \tag{8}$$

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 basic blocks, respectively. In *forward mode* a sequence of

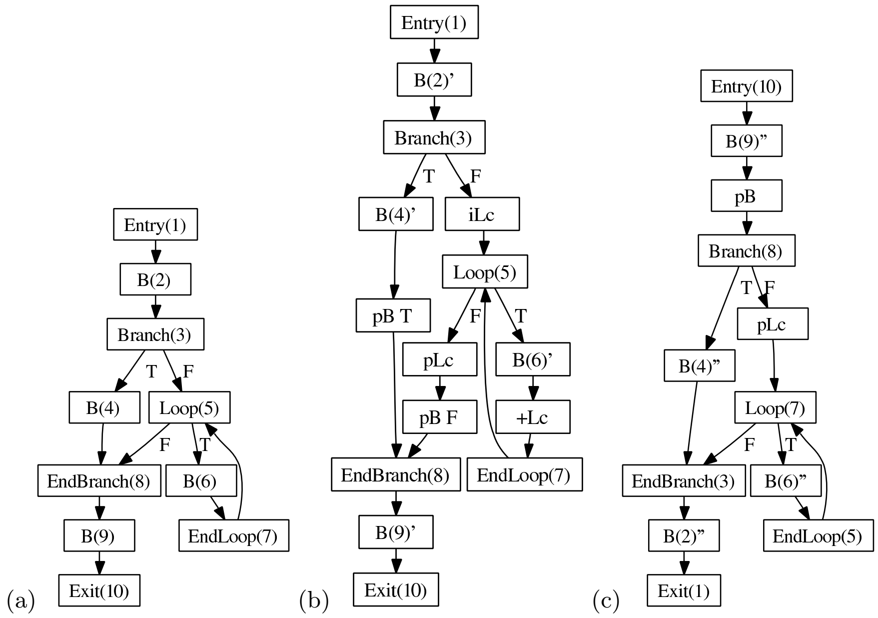


Fig. 7. CFG of Figure 6 (a) original, (b) trace generating, (c) for reversed control flow

products of the local Jacobians $\boldsymbol{J}_j$ with the directions $\dot{x}_j$ is propagated forward in the direction of the flow of control together with the computation of $\boldsymbol{f}$. In our example basic block B(6) is the third basic block ($j = 3$), and we have $\boldsymbol{x}_3 = \boldsymbol{y}_j = (\texttt{x(1)},\texttt{x(2)})$ and consequently have the operations for the Jacobian vector product shown in Figure 8.

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

Fig. 8. Pseudo code for the (sparse) product $\boldsymbol{J}_3\dot{\boldsymbol{x}}_3$ for the loop body in Figure 6. This follows the statements from Figure 4 and Figure 5. Note that `x(1)` and `x(2)` are overwritten, 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 Section 2.2. The preaccumulated $\boldsymbol{J}_j^T$ are stored during the forward execution on a stack, also called the *tape*; see Figure 9(a) for an example. In order to compute (8), they are retrieved during the reverse execution; see Figure 9(b) for an example. To find the proper path through the reversed control flow, we also have to generate a trace. We do so with an augmented CFG; for our toy example, see Figure 7(b). This augmented CFG

```
push(c_{6,-1});        t_2 = pop() * x̄(2);
push(c_{6,0});         t_1 = pop() * x̄(2);
push(c_{7,-1});        t_2 = t_2 + pop() * x̄(1);
push(c_{7,0});         t_1 = t_1 + pop() * x̄(1);
                     x̄(2) = t_2;
                     x̄(1) = t_1;
```

Fig. 9. Pseudo code for writing the tape (left) and consuming the tape for $\boldsymbol{J}_3^T \bar{\boldsymbol{y}}_3$ (right) for the loop body B(6) in Figure 6. Writing values to the tape follows the statements in Figure 4 and Figure 5, and together they constitute B(6)' in Figure 7(b). Consuming the tape constitutes B(6)" in Figure 7(c).

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 considered part of the tape. In the example in Figure 7(b) the extra basic blocks 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.

Figure 7(c) shows the CFG for the reversed control flow for our toy example. The parenthesized numbers in the node labels relate the nodes across the three graphs. The exit node becomes the entry, loop becomes endloop, branch becomes endbranch, and vice versa. Each basic block B is replaced with its reversed version B'. Finally, all control flow conditions are decided with the information recorded in Figure 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 Reversal and Checkpointing

Generally, the computer program induces a *call graph* (CG) [3] 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 [11] for any subroutine in the call graph. These concepts are more easily understood with the help of a dynamic call tree (see also [23]), where each vertex represents an actual invocation of a subroutine for a given execution of the program; see Figure 10(a) and (b) and Figure 11 for an explanation of the symbols. The order of calls is implied by following the edges left to right. Split reversal for all subroutines in the program implies we first write the tape for the entire program, followed by the reverse sweep that consumes the tape; see Figure 10(c).

Joint reversal as introduced in [9] for all subroutines in a program implies that the tape for each subroutine invocation is written immediately before the reverse sweep for that particular invocation. In our example we have to generate a tape for $\mathtt{C}^2$ while the caller $\mathtt{B}^2$ is being reversed. Because in the reversal we have no guarantee that the data used by $\mathtt{C}^2$ is correct (variables may have

```
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
```

(a)

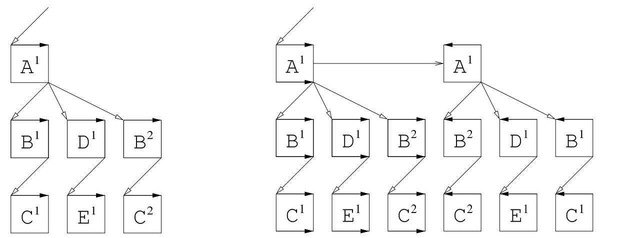


Fig. 10. Simple example code (a) with a (static) calling hierarchy, the corresponding DCT (b), and the DCT for the adjoint via split reversal (c).

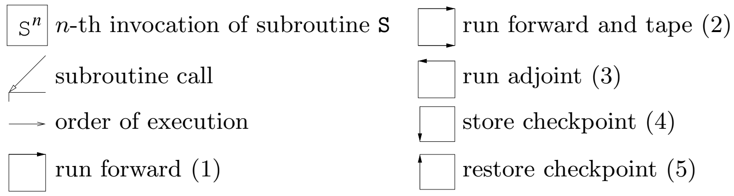


Fig. 11. Symbols for call tree reversal

been overwritten), we cannot simply reexecute $\mathtt{C}^2$. We could reexecute the entire program up to the $\mathtt{C}^2$ call and start taping, or we could store the input of $\mathtt{C}^2$ during the regular forward execution and restore it for the taping sweep. The latter approach is called checkpointing. The ensuing dynamic call tree for our example is shown in Figure 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; see Section 5.2.

3 Components of OpenAD/F

OpenAD/F is built of components that belong to a framework designed for code transformation of numerical programs. The components are tied together either by programmatic interfaces or by communication using the Xaif language. The modular design of the tool aims to reuse the components for different types of source transformation of numerical codes and for different programming languages in which these tools are written. Uses of some components outside of OpenAD/F further improve their utility and reliability. The flexibility of the tool afforded by the modular design is of equal importance. The transformation of the source code follows the pipeline shown in Figure 13. A fundamental design decision is the separation of programming-language-independent components. The pipeline as shown here and the use of a canonicalizer and postprocessor are a consequence of the design. Section 3.1 motivates the language-independent component design followed by sections on the language-independent transformation and analysis engine. The language-dependent components introduced in

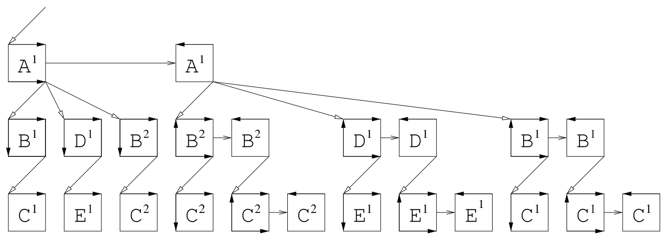


Fig. 12. DCT of adjoint obtained by joint reversal mode

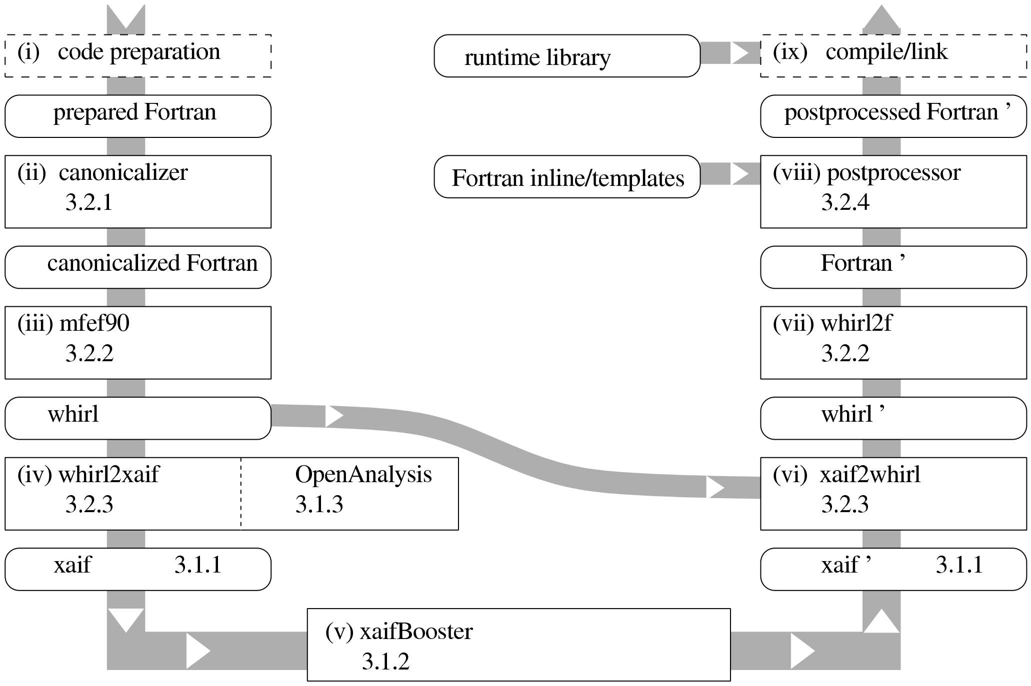


Fig. 13. OpenAD pipeline of components and relevant section numbers

Section 3.2 are to be seen in the context of this design. The components are connected in multiple ways. For instance, the activity analysis is performed within whirl2xaif, but the independent and dependent variable designation is done via a pragma mechanism implemented in the mfef90 front-end, and the results of the analyses are used with the transformation engine. The following sections contain numerous cross-references to ease following these connections. Figure 13 should be used as a reference for positioning any of the components within the tool pipeline. The regular setup procedure for OpenAD/F (see also Section 4) will retrieve all components into an `OpenAD/` directory, to which we refer from here on.

3.1 Language-Independent Components (OpenAD)

This section describes OpenAD's language-independent components. They are also used by ADIC v2.0.

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 most differences between languages do not lie with the elemental numerical operations that are at the heart of AD or other numerical 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 filtered and preserved by the language-specific front-end to reassemble the complete program after the transformation. Among these are

- user type definitions and member access (see also Section 3.2),
- pointer arithmetic,
- I/O operations,
- memory management, and
- preprocessor directives.

A more detailed discussion regarding this compromise can be found in [34]. Certain aspects of the adjoint code, such as checkpointing (see Section 2.4) and taping (see Section 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, the concrete handling of memory and I/O for taping and checkpointing is typically determined by the problem size at run-time and not by static information available during the transformation. Therefore, OpenAD handles these transformation aspects by special code expansion for subroutine-specific templates and inlinable subroutine calls in the postprocessor; see Section 3.2. Not only does this approach avoid any language-specific I/O and memory management constructs, it also affords additional flexibility.

The format of choice in OpenAD is an XML-based hierarchy of directed graphs, referred to as Xaif [15]. This is motivated by the ability to describe the Xaif with an XML schema and the existence of XML parsers that can validate any given Xaif representation against the schema. The annotated Xaif schema is documented at `www.mcs.anl.gov/xaif`. The building blocks of Xaif are structures commonly found in compilers, starting from the top with a call graph containing scopes and symbol tables, CFGs as call graph vertices, basic blocks as CFG vertices, statement lists contained in basic blocks, assignments statements with expression graphs, and variable references and intrinsic operations as expression graph vertices. Xaif elements are associated by containment. In the graph, edges refer to source and target vertices by vertex ids. Variable references contain symbol references that are associated to symbol table entries via a scope and a symbol id. An example can be found in Section 3.2, Figure 20.

The Xaif contains the results of the code analyses provided by OpenAnalysis; see Section 3.1, for instance, for activity information as additional attributes on certain Xaif elements. Side-effect analysis provides lists of variable references per subroutine; du/ud-chains are expressed as lists of identifiers of assignment elements. Alias information is expressed as sets of virtual addresses. Du/ud-chains and alias information are collected in maps. Individual entries held in these maps are referenced via foreign key attributes.

The transformation algorithms change the Xaif contents at almost all levels. In principle it would be possible to express the result entirely in plain Xaif. However, we already mentioned the code expansion approach introduced for added flexibility. The transformed Xaif adheres to a schema extended by elements representing inlinable subroutine calls as well as groups of CFGs that the postprocessor places into predefined locations in subroutine templates. The Xaif schema and examples can be found in subdirectory `xaif/`.

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 a collection of algorithms that operate on this data structure, modify it, and produce transformed Xaif output as the result. All sources for xaifBooster can be found under `xaifBooster/`. The xaifBooster data

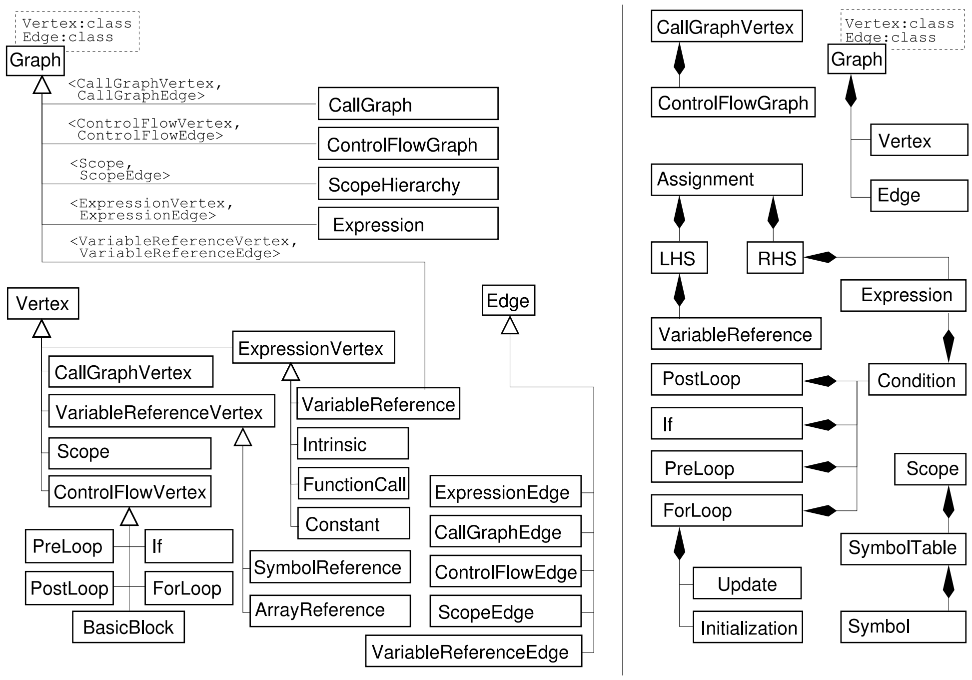


Fig. 14. Simplified class inheritance (left) and composition (right) in xaifBooster

structure resembles the information one would find in a compiler's high-level internal representation using the boost graph library (see `www.boost.org`) and the GNU Standard C++ Library (see `gcc.gnu.org/libstdc++`). Figure 14 shows simplified subsets of the classes occurring in the xaifBooster data structure in the inheritance as well as the composition hierarchy. A doxygen-generated documentation (see `www.doxygen.org`) of all data structures can be found in [26]. The class hierarchy is organized top down with a single `CallGraph` instance as the top element.

The transformation algorithms are modularized to enable reuse in different contexts. Figure 15 shows dependencies between some of the implemented algorithms. In order to avoid conflicts between transformations, all data representing the input are preserved. The data representing modifications or augmentations of the original input element in a class `<name>` (e.g., `Variable`) are held in algorithm-specific instances of class `<name>Alg` (e.g., `VariableAlg`). They are associated via mutual references accessible through `get<name>AlgBase()` and `getContaining<name>()`,

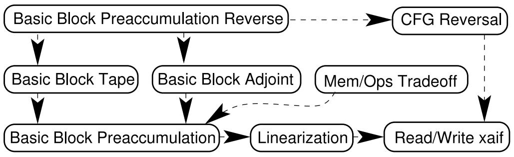


Fig. 15. Schematic dependencies of the transformation algorithms implemented in xaifBooster. Note that the inheritance hierarchy of the individual classes constituting the algorithms may skip some of these dependencies depicted here.

respectively. The instantiation of the algorithm-specific classes follows the factory design pattern. The factory instances in turn are controlled by transformation-specific `AlgFactoryManager` classes that ensure instantiation of the `<name>Alg` subclass appropriate for a given transformation. Further details can be found in [33].

In the following sections we highlight the role of particular algorithms. Each algorithm has a driver `t.cpp` that is compiled into a binary `t`. Both can be found in `algorithms/<algorithm_name>/test/`. The driver encapsulates the algorithm in a stand-alone binary that provides the functionality described in the following sections.

Reading and Writing Xaif Reading and writing the Xaif are part of the basic infrastructure found in the source code in `system/`. The Xerces C++ XML parser (see `xml.apache.org/xerces-c`) uses XML element handlers implemented in `system/src/XAIFBaseParserHandlers.cpp` to populate the xaifBooster data structures from the top down. All OpenAD/F components that read Xaif data can perform validation according to the schema. Beyond the schema-based validation are additional consistency checks; therefore, manual modifications of Xaif data should be done judiciously. The transformed data is unparsed into Xaif through a traversal of the data structure and calls to virtual functions implemented by the transformation algorithms.

The *catalog of inlinable intrinsics*, supplied as a separate XML file following a specialized schema in Xaif (see Sections 3.1 and 3.2), is parsed prior to the Xaif program representation. It contains the declarations of intrinsics and defines the partial derivative expressions. The driver found in `system/test/t.cpp` provides only parsing and unparsing functionality. It can be used to establish that the tool preserves the semantics of the original program when no transformation is involved.

Linearization Section 2.1 explained the computation of the local partial derivatives c_{ji}, the edge labels in the computational graph G. For each elemental ϕ (see (2)) we find definitions of the respective partials in the inlinable intrinsics catalog (see above). An example for division is given in Figure 16.

Because partials are defined in terms of positional arguments, the right-hand-side expression may have to be split into subexpressions to assign intermediate values (positional arguments for the partial computation) to temporary variables, for example, v_3 and v_4 in Figure 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 Section 3.1), conservative correctness requires an additional assignment to delay the (potential) overwrite until after the partials depending on

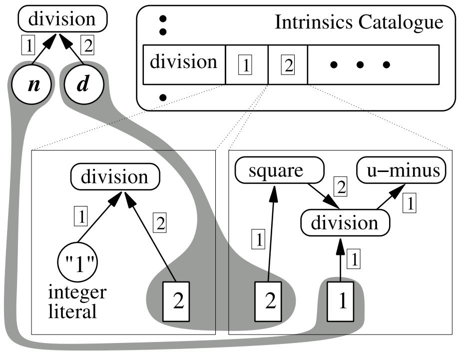


Fig. 16. Partial expressions for the division operator. The expression is given in terms of argument positions, here 1 for the numerator and 2 for the denominator. Applied to `z=n/d` this yields $\frac{\partial z}{\partial n}= 1/d$ and $\frac{\partial z}{\partial d}= -n/(d**2)$. The format also permits the reuse of the intrinsic result in the partial computation, resulting in savings for some intrinsics such as `exp()`.

the variable's original value have been computed. The result of the linearization is a representation for code containing the potentially split assignments along with assignments for each nonzero edge label c_{ji}. These representations are contained in the `xaifBoosterLinearization::AssignmentAlg` instances associated with each assignment in the Xaif. The generated code, after unparsing to Fortran, is compilable, but by itself does not compute useful derivative information for $\boldsymbol{f}$.

Basic Block Preaccumulation Basic block preaccumulation generates a code representation that can be used to compute derivatives in forward mode. It builds on the linearization done in Section 3.1. The first step constructs the computational graphs G for contiguous assignment sequences in any given basic block. To ensure semantic correctness of the graph being constructed in the presence of aliasing, it relies on alias analysis and du/ud-chains supplied by OpenAnalysis; see Section 3.1. The algorithm itself is described in detail in [31]. Because of aliasing as analyzed by OpenAnalysis, it may not be possible to cover all assignments by the construction of a single G. 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 Section 2 one may think of the sets of assignments forming each of these graphs as a separate basic block. The driver for the algorithm allows one to disable the graph construction across assignments and restrict it to single right-hand sides.

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 [4, 22]. Angel determines an elimination sequence and returns it as fused multiply-add expressions in terms of the edge references. Several heuristics implemented within Angel control the selection of elimination steps and thereby the preaccumulation code that is generated. The algorithm code calls a default set of heuristics. All heuristics use the `CrossCountryInterface` and therefore different heuristics can be selected.

The second step in this transformation is the generation of preaccumulation code. First the algorithm concretizes 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 Figure 5. Then the transformation generates the code that performs the saxpy operations $(a \cdot x + y)$ shown in (7). Considering the input and output variables $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ of a basic block, 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 Figure 8 illustrates this case. The detection of the intersection elements relies on the alias analysis provided by OpenAnalysis. To reduce overhead, we generate saxpy calls following the interface specified in `algorithms/DerivativePropagator/` for four cases:

$$\text{(a): } \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x}, \text{ (b): } \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x} + \dot{y}, \text{ (c): } \dot{y} = \dot{x}, \text{ (d): } \dot{y} = 0 \quad . \tag{9}$$

The generated code is executable and represents an overall forward mode according to (7) with basic block-local preaccumulation in cross-country fashion.

Control Flow Reversal Section 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 to check the correctness without interference from other transformations. It is a building block for the adjoint code generator described in Section 3.1. The loop counters and branch identifiers are stored in the same stack data structure that is used for the *tape* (introduced in Section 2.3, see Figure 9, and also used in Section 3.1). 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`. Of course it is possible to reverse such graphs, for instance by enumerating all basic blocks, recording the execution sequence, and invoking them according to their recorded identifier in reverse order. Such a reversal is less efficient than a code that, by employing proper control flow constructs, aids compiler optimization and yields much more efficient adjoint code. For the same reason well-tuned codes implementing the target function $\boldsymbol{f}$ avoid arbitrary jumps. 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, for example, `t`, and we iterate up to the stored execution count, for example, `c`. Then the reversed loop is `do t=1,c`. 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` must 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 is 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 better answered with a code analysis similar to to-be-recorded analysis [13]. Currently we do not have such analysis available and instead, as a compromise, define the *simple* loop that can be 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, and
– 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 multidimensional arrays often 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. Details can be found in [32].

Writing and Consuming the Tape Section 2 explains the need to store the c_{ji} on the tape. The writing transformation[6] stores the nonzero elements of local Jacobians $\boldsymbol{J}_j$. It is implemented as an extension of the preaccumulation in Section 3.1, but instead of using the Jacobian elements in the forward saxpy operations as in (7), we store them on a stack as shown for the example code in Figure 8(a). The tape consuming transformation algorithm[7] reinterprets the saxpy operations generated in Section 3.1 according to Figure 17. The tape writing and consumption

	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)	$\dot{y} = 0$	$\overline{y} = 0$

Fig. 17. Saxpy operations from (9) and their corresponding adjoints

are used for the adjoint code generator in Section 3.1.

Basic Block Preaccumulation Reverse This transformation[8] represents the combination of the various transformations into a coherent output that, unparsed into code and postprocessed, compiles as an adjoint model. For the postprocessing steps, see Section 3.2. Additional functionality is the generation of code that is able to write and read checkpoints at a subroutine level; see also Section 2.4, which relies heavily on the results of side-effect analysis (see Section 3.1) and the inlinable subroutine call mechanism of the postprocessor (see Section 3.2) to accomplish the checkpointing. The transformation has various options that control subroutine argument intents (needed to accomplish checkpointing) and the renaming of subroutines. Details can be found in [35].

Static Code Analyses (OpenAnalysis) The OpenAnalysis toolkit (see [30]) 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 [13], and linearity analysis [29] in OpenAD/F. Standard analyses provided by OpenAnalysis

[6] See `algorithms/BasicBlockPreaccumulationTape/` .
[7] See `algorithms/BasicBlockPreaccumulationTapeAdjoint/` .
[8] See `algorithms/BasicBlockPreaccumulationReverse/` .

include CFG and call graph construction, alias analysis, reaching definitions, ud- and du-chains, and side effects [3]. Because these analyses require lower-level information (e.g., pointer dereferences) not represented in the numerical core (see Section 3.1), OpenAnalysis primarily interacts with the language-dependent OpenADFortTk component (see Section 3.2).

A brief description of alias analysis illustrates this interaction. The alias map data structure in Xaif maps each variable reference to a set of virtual memory 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 aliased to one another. OpenAnalysis determines the aliasing relationships by querying the front-end's intermediate representation of the program through an abstract, analysis-specific interface called the *alias IR interface*. This is a language-independent interface between OpenAnalysis and any intermediate representation for an imperative programming language. Within OpenADFortTk the whirl2xaif subcomponent implements the alias IR interface for the Fortran intermediate representation given in whirl. The interface includes iterators 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 analysis result (i.e., the alias map needed to create Xaif) is returned to whirl2xaif through an alias results interface.

For the activity analysis performed by OpenAnalysis, the independent and dependent variables of interest are communicated via the front-end through the use of pragmas; see Section 3.2. The analysis indicates which variables are *active* (i.e., have nonzero derivatives) at any time, which memory references are active, and which statements are active. The current activity analysis is based on the formulation in [13], but the implemented data-flow engine does not take advantage of structured data-flow equations yet. It can handle unstructured as well as structured programs. The source code can be found in subdirectory `OpenAnalysis/`.

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 used in the transformation pipeline in the following sequence.

1. The *canonicalizer* converts programming constructs into a canonical form described in Section 3.2.
2. The compiler front-end mfef90 parses Fortran and generates an intermediate representation (IR) in the whirl format; see Section 3.2.
3. The bridge component whirl2xaif
   - drives the various program analyses (see Section 3.1) and
   - translates the numerical core of the program and the results of the program analyses from whirl to Xaif; see also Section 3.2.
4. The bridge component xaif2whirl translates the differentiated numerical core represented in Xaif into the whirl format; see Section 3.2.
5. The unparser whirl2f converts whirl to Fortran; see Section 3.2.

6. The *postprocessor* is the final part of the transformation that performs template expansion as well as inlining substitutions; see Section 3.2.

Canonicalization In Section 3.1 we explained 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 the canonicalization is done automatically, it 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 is written in Python and can be found under `OpenADFortTk/tools/canonicalize/`.

Canonicalization 1 *All function calls are canonicalized into subroutine calls.*

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

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

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

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

Fig. 18. Canonicalizing a function call (a) inside an assignment statement into a subroutine call (b). The function definition (c) is turned into a subroutine definition (d).

For the transformations, in particular the basic block-level preaccumulation, we want to ensure that an assignment affects a single variable on the left-hand side. Therefore, the right-hand-side expression must to be side-effect free. While rarely enforced by compilers, this is also a requirement for Fortran programs. Rather than determining side effects of user-defined functions, we pragmatically hoist *all* user-defined functions; see Figure 18(a) and (b). Subsequently, assignment right-hand-side expressions consist only of elemental operations typically defined in the programming language as built-in operators and intrinsics. The canonicalizer also performs the accompanying transformation of the function definitions (Figure 18(c)) into subroutine definitions (Figure 18(d)). The `oad_s_` prefix is adjustable. A particular canonicalization of calls without canonicalization of definitions is applied to the `max` and `min` intrinsics because Fortran cannot express their partials in closed form. OpenAD/F provides a run-time library containing definitions for the respective subroutines called instead.

Canonicalization 2 *Nonvariable actual parameters are hoisted.*

Expressions as actual parameters may need to be differentiated, but we differentiate only assignment right-hand sides. Consequently, we hoist all nonvariable actual parameters into temporaries; see Figure 19(a) and (b).

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

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

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

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

Fig. 19. Before (a) and after (b) hoisting a nonvariable parameter and canonicalizing a common block (c) into a module (d).

Canonicalization 3 *Common blocks are converted to modules.*

To ensure proper initialization of active global variables, the elements of the common block (Figure 19(c)) are declared as module variables (Figure 19(d)). Care must be taken to privatize and declare any symbolic size parameters for elements of the common block. None of the above canonicalizations are intended to produce manually maintainable code. Therefore we prefer simplicity to more sophisticated transformations, for example, a common block converter that modularizes dimension information shared between common blocks.

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 as a multiplatform 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 be unparsed to Fortran with whirl2f. The source-level whirl representation is 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. HiPerSoft's main contribution to the Open64 community has been the source-level extension to whirl that is geared toward source-to-source transformations. It has invested significant effort in the whirl2f unparser.

For the purpose of AD, user-supplied hints and required input are typically not directly representable in the programming language. For example, an AD tool must know which variables in the code for $\boldsymbol{f}$ are independent and which are dependent; see Section 3.1. While one can supply this information externally, for instance with a configuration file, we introduced a special pragma facility, encoded within Fortran comments. Pragmas are intrusive, but they have the advantage of being parsed by the front-end and being associated with a given context in the code. Thus, code and AD information are easily 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 [27]. To specify a variable y as a dependent, the user writes `!$openad dependent(y)`, where `$openad` is the special prefix that identifies OpenAD pragmas. To provide flexibility, we introduced a generic `!$openad xxx <some text>` pragma,[9] which 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 provide an easy way to implement additional user hints while eliminating the significant development costs associated with modifying Open64.

```
<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>
```

Fig. 20. Section of Xaif representing an array dereference `Y(I)`

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*; that is, the derived type reference is converted into a uniquely named scalar variable in Xaif. For example, `u%v` may be represented 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 `v`. This scalarization is reversed upon backtranslating the transformed Xaif representation into whirl. Variable references of user-defined type can appear in the Xaif as subroutine parameters and in these cases are given a special *opaque* type attribute. 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 type, and annotated markers preserved), xaif2whirl generates a new whirl representation for the differentiated code while restoring user-defined types, dereferences, and state-

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

ments not shown in the Xaif. The differentiated Xaif relies on postprocessing; see Section 3.2. Therefore the second major challenge for xaif2whirl is the creation of whirl containing the postprocessor directives related to three tasks to be accomplished by the postprocessor:

– Declaration and use of the active variables
– Placement of inlinable subroutine calls
– Demarcation of the various subroutine bodies used in the subroutine template replacements

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 Section 3.1. 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.

Postprocessing The postprocessor performs the three tasks outlined in Section 3.2. The main rationale for its existence is the added flexibility it affords the tool, which would otherwise be achieved only at a substantially higher implementation effort.

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[10] and therefore is much easier to adapt to a changing active type implementation than to find the proper whirl representation and modify xaif2whirl to create it. However, 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 receives an additional `USE` statement that makes the type definition in `OpenAD_active` known.

Inlinable Subroutine Calls The second task, the expansion of inlinable subroutine calls, is more complex because any call expansion now has 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)>
```

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

that directs the postprocessor to expand the following call according to a definition found in an input file[11]; see also `runTimeSupport/simple/ad_inline.f`. For example, pushing a preaccumulated local Jacobian value as in Figure 9(a) might appear in the code as shown in Figure 21(b), for which we have a definition in

(a)
```
1 subroutine push(x)
2 !$openad$ inline DECLS
3   use OpenAD_tape
4   implicit none
5   double precision :: x
6 !$openad$ end DECLS
7   dTape(dTapePtr)=x
8   dTapePtr=dTapePtr+1
9 end subroutine
```

(b)
```
!$openad inline push(subst)
call push(OpenAD_Symbol_5)
```

(c)
```
dTape(dTapePtr) = OpenAD_Symbol_5
dTapePtr = dTapePtr+1
```

Fig. 21. Inlinable subroutine definition(a), a call (b), and the expanded call (c).

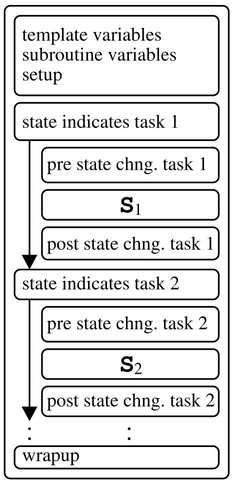
`ad_inline.f` as for instance in Figure 21(a). The postprocessor ignores the `DECLS` section (lines 2–6) and expands this to what is shown in Figure 21(c). Note that for flexibility any calls with inline directives for which the postprocessor cannot find an inline definition remain unchanged. For example, we may compile the above definition for `push` and link it instead.

Subroutine Templates The third task, the subroutine template expansion, is somewhat related to the inlining. In Figure 21, the tape storage referred to in the `push` routine needs to be defined, and in our design the subroutine template is the intended place for such definitions. In our example this is achieved through including a `use` statement in the template code (see Figure 21(a), line 3) where a module provides the taping space. The main purpose of the subroutine template expansion, however, is to orchestrate the call graph reversal. The simple reversal schemes introduced in Section 2.4 can be realized by carrying state while traversing the call tree.

The basic building blocks from the transformations in Section 3.1 are variants $\mathtt{s}_i$ of the body of an original subroutine body $\mathtt{s}_0$, each accomplishing one of the tasks (i) denoted with (1)...(5) in Figure 11. For instance, the taping variant S_2 is created by the transformation in Section 3.1 or the checkpointing by the transformation in Section 3.1. 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 body contains one S_i. The execution of each branch is determined by a global structure whose members represent the state of execution in the reversal scheme. The branches contain code for pre- and poststate 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 Figure 22(a). The template is written in Fortran. Each subroutine in the postprocessor Fortran input is transformed according to a default subroutine template found in a `ad_template.f` file or in a

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

(a)



(b)

```
subroutine template()
  use OpenAD_tape ! tape
       storage
  use OpenAD_rev ! state
       structure
!$TEMPLATE_PRAGMA_DECLARATIONS
  if (rev_modetape) then
! the state component
! 'taping' is true
!$PLACEHOLDER_PRAGMA$ id=2
  end if

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

end subroutine template
```

Fig. 22. Subroutine template components (a), split-mode Fortran template (b)

file specified in a `!$openad XXX Template <file name>` pragma to be located within the subroutine body. The input Fortran also contains `!$openad begin replacement` <*i*> paired with pragmas `!$openad end replacement`. Each such pair delimits a code variant $\mathtt{S}_i$ and the postprocessor matches the respective integer identifier i 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 Figure 10(c) 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 performing the adjoint run. Figure 22(b) shows a simple split-mode template; see also `runTimeSupport/simple/ad_template.split.f`. The state is contained in `rev_mode`, a static Fortran variable; see `runTimeSupport/simple/OpenAD_rev.f90` of type `modeType` also defined in this module. 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.

Figure 12 illustrates the task pattern for a joint reversal scheme that requires state changes in the template and requires more code alternatives. Figure 23 shows a simplified joint mode template together with a detailed explanation; see also `runTimeSupport/simple/ad_template.joint.f`.

4 Tool Usage

The following contains brief remarks about how to obtain and use OpenAD/F. While the principal approach will remain the same, future development may introduce slight changes.

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.`

```
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=5
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.; rev_mode%arg_restore=.false.
26     rev_mode%plain=.true.; rev_mode%tape=.false.
27  !$PLACEHOLDER_PRAGMA$ id=2
28     rev_mode%arg_store=.false.; rev_mode%arg_restore=.false.
29     rev_mode%plain=.false.; rev_mode%adjoint=.true.
30   end if
31   if (rev_mode%adjoint) then
32  ! run the adjoint code
33     rev_mode%arg_restore=.true.; rev_mode%tape=.true.
34     rev_mode%adjoint=.false.
35  !$PLACEHOLDER_PRAGMA$ id=3
36     rev_mode%plain=.false.; rev_mode%tape=.true.
37     rev_mode%adjoint=.false.
38   end if
39 end subroutine template
```

Fig. 23. Joint-mode Fortran template with argument checkpointing. The state transitions in the template directly relate to the pattern shown in Figure 12. Each prestate change applies to the callees of the current subroutine. Since the argument store (S_4) and restore (S_5) variants do not contain any subroutine calls, they do not need state changes. Looking at Figure 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 fact 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`.

html#anonymous. The reader is encouraged to refer to the up-to-date instructions in [26].

The components of OpenAD/F transform the code in a predetermined sequence of steps, the *pipeline*. Depending on the particular problem, certain variations to the pipeline achieve better performance of the generated code. The most common pipeline setups are encapsulated in a Perl script `OpenAD/tools/openad/openad`. Invoking it with the `-h` option displays the script usage, of which the mode choices are the most important. For large projects it will be more appropriate to integrate the sequence of customized transformation steps into a `Makefile`. The technical details can be found in [35].

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

- Active type (see `OpenAD_active.f90`)
- Checkpointing (only for adjoint models, see `OpenAD_checkpoints.f90`)
- Taping (only for adjoint models, see `OpenAD_tape.f90`)
- State for call graph reversal (only for adjoint models; see `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.

We also require a *driver* that invokes the transformed routines and seeds and retrieves the derivatives. Examples for such drivers can be found in Section 5.1 and [35].

5 Applying OpenAD/F

The following examples illustrate some of the implementation strategies chosen for OpenAD/F. The first, simple example demonstrates the general embedding approach of the transformed code within an overall driver. The second example is taken from a complex real-life application.

5.1 Toy Example

Consider as a toy example the function $y = \sin(x^2)$ whose code is depicted in Figure 24(a), along with the user-defined dependent and independent declarations; see Section 3.2. 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 Eqn. (1). A simple driver for the tangent-linear model is shown in Figure 25(a). Because of 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 Figure 25(b).

(a)

```
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
```

(b)

```
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
```

Fig. 24. Toy example(a) and the modified signature for the tangent-linear model(b)

(a)

```
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
```

(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
```

Fig. 25. Toy example tangent-linear (a) and adjoint (b) driver

5.2 Shallow Water Model

Implementation details rapidly become critical determinants for the practicality of complex real-life applications. We illustrate some aspects by way of a model common in geophysical fluid dynamics. The shallow-water model for a homogeneous inviscid fluid on a rotating sphere provides useful insights into time-dependent large-scale flow regimes in the atmosphere and ocean. While simpler than a fully-fledged, vertically stratified atmosphere or ocean general circulation model, it retains many of its complexities, including, from a physical point of view, the nonlinear momentum equations, and, from a computational point of view, the scaling of the computational problem with domain size and resolution. The model is thus well suited as an example for applications where (i) the control space scales with the dimensionality of the model grid, and (ii) the model state is nonlinear and time-evolving, requiring the full state for derivative evaluations. The present implementation was adapted from work by [16] in which they study the feasability of using bottom topography as a control variable for ocean state estimation. It is here extended to a global configuration at 2×2 degree horizontal resolution with realistic bathymetry. The scalar-valued dependent variable (objective function) is the volume transport through Drake Passage in the Southern Ocean, the control space spanned by the horizontal bathymetry field is 180*80-dimensional. We seek the sensitivity of the Drake Passage transport to changes in bottom topography everywhere (independently) in the domain, which we obtain by means of the adjoint model (reverse mode AD)

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 Section 3.1, all code that takes part in computing the model has to be visible to the tool, which means it has to be concatenated into a single file. One can do so for all source files of the model, but in many cases the result 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.

- The static code analysis and subsequently the code transformation has to make conservative assumptions to ensure correctness; for example, for alias analysis this means an overestimate of the memory locations that can alias each other. One of the effects of these potential aliases is 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 often cannot 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.

Section 5.1 indicates the need for a modification to the code that drives the model computation to at least perform 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.

Orchestrate a Reversal and Checkpointing Scheme Joint and split reversal (see Section 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 an 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 Figure 26, which is invoked `o` times.[12] 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`. Figure 26(b) shows the `cost` subroutine called from `I` as well as from `top`. According to Figure 26, however, we would need two versions of `cost`, one that as callee of `top` is reversed in joint

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

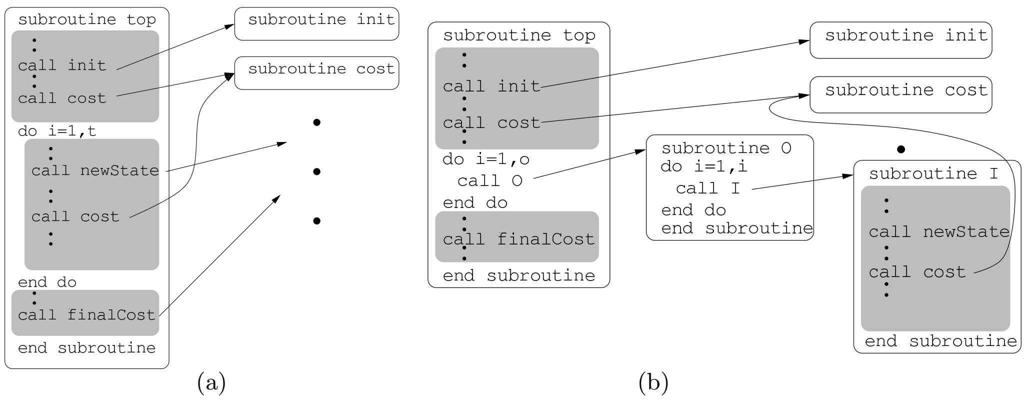


Fig. 26. Modification of the original code (a) to allow 2 checkpointing levels (b)

mode and one that as callee of `I` is reversed in split mode. To maintain the static reversal approach,[13] one needs to duplicate `cost`.

File I/O and Simple Loops The model code uses both the NetCDF library and the built-in Fortran I/O during the initialization and output of results. Because during the model computation no intermediate values are written and read, there is no loss of dependency information. The I/O can lead to problems, however, for instance when an activated array is initialized, effectively setting the `%v` and `%d` values in the first half of the array instead of setting only the `%v` values in the entire array. This is a well-known consequence of the active type implementation and the lack of type checks in Fortran compilers. While one could argue that the code should be generated to avoid reading or writing the derivative information, this is not always the desired behavior, in particular not if one reads or writes active intermediate variables. A simple and effective measure to circumvent this problem is to encapsulate initializations in routines excluded from the transformation. OpenAD/F creates conversion code for parameters to such external subroutine that are active at the call site. However, 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 run-time and memory spent on taping array indices used in loops. The *simple* loop concept introduced in Section 3.1 is designed to eliminate much of this overhead. Not all loops within the given model code satisfy the corresponding conditions. Hence, 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.

Results Figure 27 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. The integration period spans an interval of 50 years. Enhanced sensitivities are manifest both locally in the vicinity of the Drake Passage as well as remotely,

[13] A dynamic reversal scheme is forthcoming.

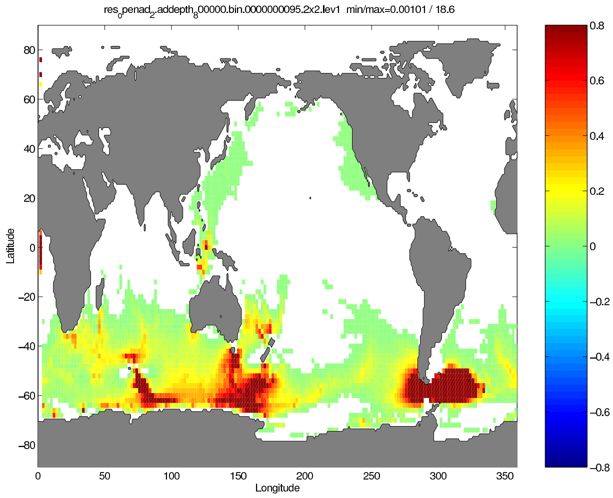


Fig. 27. Sensitivity (gradient) map for 2×2 degree resolution

e.g. over the Kerguelen Plateau, the South Pacific Ridge and in the Indonesian Throughflow. Sensitivities are mediated through the adjoint variables (i.e. the Lagrange multipliers of the) flow field represented by the model dynamics. Instead of one single adjoint model integration $180*80 = 14{,}400$ forward integrations would have been necessary to produce this map. 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 normal model computation. We expect the ongoing development of OpenAD/F(see also Section 6) to yield further efficiency gains.

6 Summary and Future Work

OpenAD/F is an automatic differentiation 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.

The largest portion of the ongoing work on OpenAD/F is devoted to improvements needed for the application of the tool to a fully-fledged parallel nonlinear ocean general circulation model used for ocean circulation and climate studies. The currently applied configurations of the model yield adjoint codes with an overhead factor of about 10 over the normal model computation.

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 emphasize 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.

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 make obsolete existing source transformation tools such as the differentiation-enabled NAG Fortran compiler,[14] TAF,[15] or TAPENADE.[16] 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 rarely considered a research subject and as such cannot be made the major objective of a tool 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 (e.g., using well-structured versus unstructured control flow).

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 users, we are also working on collecting data for efficiency estimates that permit an informed choice between the code transformation options. The results of ongoing research into AD algorithms – in particular dynamic call graph reversal, more efficient control flow reversal, and improved elimination techniques in the computational graphs – will be incorporated into OpenAD.

Acknowledgements

Funding for the first phase of the OpenAD project was provided by NSF under ITR contract OCE-0205590. The current development is supported in part by NSF under ITR contract OCE-0530867, by NASA, Modeling, Analysis and Prediction (MAP), Earth-Sun Division, Science Mission Directorate, award number NNG06GC28G and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

[14] `http://www.nag.co.uk/nagware/research/ad_overview.asp`
[15] `http://www.FastOpt.de`
[16] `http://tapenade.inria.fr:8080/tapenade/index.jsp`

References

1. ACTS. http://www.autodiff.org/ACTS, 2007. Adjoint Compiler Technology & Standards project.
2. ADIC. http://www.mcs.anl.gov/adicserver, 2007.
3. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
4. 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.
5. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.
6. 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.
7. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
8. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.
9. C. Faure and Y. Papegay. Odyssée version 1.6: The user's reference manual. Technical Report available at http://www.inria.fr/recherche/equipes/safir.en.html, Sophia Antipolis, Projet SAFIR, 1997.
10. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24:437–474, 1998.
11. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.
12. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markovitz rule. In *[8]*, pages 126–135, 1991.
13. 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.
14. P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
15. 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, Anaheim,CA, 2002. ACTA Press.
16. M. Losch and C. Wunsch. Bottom topography as a control variable in an ocean model. *J. Atmospheric and Oceanic Technology*, 20:1685–1696, 2003.
17. J. Marotzke, R. Giering, K.Q. Zhang, D. Stammer, C. Hill, and T. Lee. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport variability. *J. Geophysical Research*, 104, C12:29,529–29,547, 1999.
18. J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *J. Geophysical Research*, 102, C3:5,753–5,766, 1997.
19. U. Naumann. Elimination techniques for cheap Jacobians. In *[7]*, pages 247–253, 2002.
20. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004.
21. U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Prog.*, 2007. To appear.
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.
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. NEOS. `http://www-neos.mcs.anl.gov/`, 2007. Network Enhanced Optimization Server.
26. OpenAD. `http://www.mcs.anl.gov/OpenAD`, 2007.
27. OpenMP. `http://www.openmp.org`, 2007.
28. D. Stammer, C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C.N. Hill, and J. Marshall. The global ocean circulation and transports during 1992 – 1997, estimated from ocean observations and a general circulation model. *J. Geophysical Research*, 107(C9):3118, 2002.
29. M. Strout and P. Hovland. Linearity analysis for automatic differentiation. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 574–581. Springer, 2006.
30. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Proceedings of the Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 5-6 2005.
31. J. Utke. Flattening basic blocks. In *[6]*, 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*, (79):1280–1294, 2006.
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, CA, 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, CA, 2004. ACTA Press.
35. J. Utke and U. Naumann. OpenAD/F: User manual. Technical Report available at `http://www.mcs.anl.gov/openad/`, Argonne National Laboratory, 2006.
36. R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.
37. C. Wunsch. *Discrete Inverse and State Estimation Problems: With Geophysical Fluid Applications*. Cambridge (UK), 2006.
38. C. Wunsch and P. Heimbach. Practical global oceanic state estimation. *Physica D*, in press, 2006.