OpenAD/F: User Manual

J. Utke
U. Naumann
A. Lyons

draft vers. hg:65776b2019f3+:118+

compiled on March 15, 2014 with

../OpenAD	svn:247
Open64	svn:917
OpenADFortTk	svn:1127
OpenAnalysis	svn:454
xercesc	svn:48
xaifBooster	svn:156
xaif	svn:62
angel	svn:90
boost/boost	svn:66620
RevolveF9X	hg:20:56046883e55e
Examples	hg:107:8626ab6e2128
Regression	hg:483:c48c7a716a50
OpenADFortTk/Regression	hg:240:1b4c6f62a461
OpenADFortTk/tools/SourceProcessing/Regression	hg:223:18a00f156e5c

This is hyperref'ed PDF and should be viewed in a PDF reader instead of being printed!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A general introduction to the aims of the OpenAD/F tool and the underlying principles was given in an ACM TOMS paper [46]. Because of the ongoing development of the tool a number of changes have occurred since finalizing this paper. The most significant changes are listed in sec. 8.1. This manual concentrates the technical details of using OpenAD/F and introduces the theoretical principles of automatic automatic differentiation (AD) only briefly. For more in-depth discussions the reader is referred to [22], the series of AD conference proceedings [19, 10, 14, 13, 11], and the AD community's website www.autodiff.org.

For a quick test with small-scale problems one may proceed directly to sec. 1.3.

1.1 Motivation for the OpenAD/F Design

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 [33]. 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.

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

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

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

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

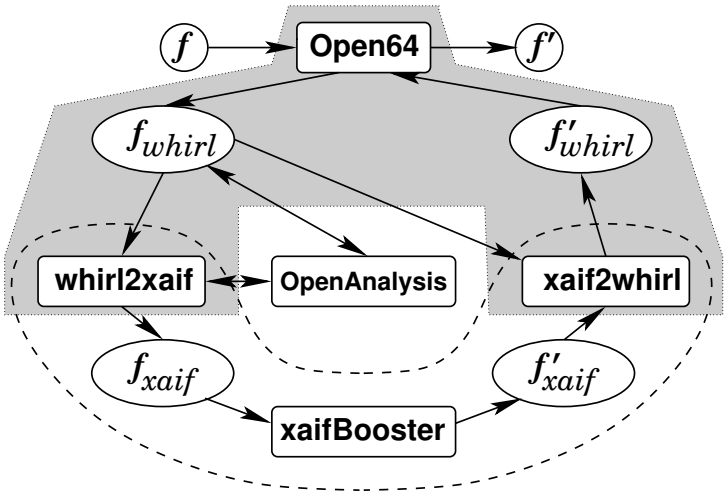


Figure 1.1: OpenAD/F components and tool chain

These considerations motivated the Adjoint Compiler Technology & Standards [5] project, a research and development collaboration of MIT, Argonne National Laboratory, The University of Chicago, and Rice University. OpenAD/F is one of its major results.

1.2 Overview

OpenAD/F[35] is the Fortran incarnation of the AD framework OpenAD. The C/C++ oriented tool ADIC v2.0 [4] is based on the same framework but is not subject of this article. OpenAD/F has a modular design. The collaboration of the OpenAD/F components is illustrated in fig. 1.1. Our input is some numerical model given as a Fortran program $\boldsymbol{f}$. The Open64[34] front-end performs a lexical, syntactic, and semantic analysis and produces an intermediate representation of $\boldsymbol{f}$, here denoted by $\boldsymbol{f}_{whirl}$, in the so-called whirl format. OpenAnalysis is used to build call and control flow graphs and perform code analyses such as alias, activity, side-effect analysis. This information is used by whirl2xaif to construct a representation of the numerical core of $\boldsymbol{f}$ in XAIF format shown as $\boldsymbol{f}_{xaif}$. A differentiated version of $\boldsymbol{f}_{xaif}$ is derived by an algorithm that is implemented in xaifBooster and is again represented XAIF as $\boldsymbol{f}'_{xaif}$. The information in $\boldsymbol{f}'_{xaif}$ and the original $\boldsymbol{f}_{whirl}$ are used by xaif2whirl to construct a whirl representation $\boldsymbol{f}'_{whirl}$ of the differentiated code. The unparser of Open64 transforms $\boldsymbol{f}'_{whirl}$ into Fortran90, thus completing the semantic transformation of a program $\boldsymbol{f}$ into a differentiated program $\boldsymbol{f}'$. The gray shaded area encloses the language specific front-end that can potentially be replaced by front-ends for languages other than Fortran. For instance, the new version of ADIC [24] couples a C/C++ front-end based on the EDG parser [15] and uses ROSE in combination with SAGE 3 [40] as internal representation in combination with language independent components of OpenAD.

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

1.3 A One-Minute Example

This section provides a quick illustration of the tool without going into any detail of the inner workings. It may be useful to run the examples as a sanity test.

Users familiar with source transformation AD tools may want to skip this section and proceed to sec. 1.4.

To obtain the source code for the examples in this section please refer to sec. 8.3. To run the examples one needs to install OpenAD/F following the instructions in sec. 2.1. We assume a simple routine, see fig. 1.2 (left), computing $y = f(x) = \tan(x)$ implemented by `head`. We need to prepare the code for the activity analysis by specifying the independent variables (here `x`) and the dependent variables (here `y`) shown in lines 4 and 6 of fig. 1.2 (right). In the following we assume the OpenAD/F environment has been set up as described in sec. 2.2.

```
1  subroutine head(x,y)
2    double precision :: x
3    double precision :: y
4
5    y=tan(x)
6
7  end subroutine
```

```
1  subroutine head(x,y)
2    double precision :: x
3    double precision :: y
4  !$openad INDEPENDENT(x)
5    y=tan(x)
6  !$openad DEPENDENT(y)
7  end subroutine
```

Figure 1.2: Simple example code (left, see file `$OPENADROOT/Examples/OneMinute/head.f90`), prepared for differentiation (right, see file `$OPENADROOT/Examples/OneMinute/head.prepped.f90`)

```
openad -c -m f head.prepped.f90
openad log: openad.2014-03-15_01:33:34.log~
preprocessing fortran
parsing preprocessed fortran
analyzing source code and translating to xaif
tangent linear transformation
 getting runtime support file OAD_active.f90
 getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
```

Figure 1.3: Progress messages from `openad` for forward mode.

1.3.1 Forward Mode

For this simple example we can use the following wrapper script.

`$OPENADROOT/bin/openad`

The environment setup adds this directory to the `PATH`. All script options are shown when it is invoked with the `-h` flag. It provides a simple recipe for the AD transformations that we can use in a straightforward case like this by calling

```
cd $OPENDADROOT/Examples/OneMinute
openad -c -m f head.prepped.f90
```

to generate a code version for forward mode. The transformation is selected by the flag `-m f`. The script produces progress progress messages like the ones shown in fig. 1.3. The resulting Fortran output is written to a file called

`$OPENDADROOT/Examples/OneMinute/head.prepped.pre.xb.x2w.w2f.post.f90`

and its contents is shown in fig. 1.4. The formal parameters `x` and `y` are active and consequently their type is changed. Lines 24 and 25 contain the derivative of $\tan(x)$ which is $\tan(x)^2 + 1$. Because of potential aliasing between the formal parameters of `head` a temporary variable `OADsym_0` is introduced to hold the result before it is assigned to the output's value component `y%v`. The call to `sax` performs the forward propagation operation

$$\dot{y} = \frac{\partial \tan(x)}{\partial x} \cdot \dot{x} \quad ,$$

see also (3.6) on pg. 25. A simple driver that calls the transformed `head(x,y)` is shown in fig. 1.5. It initializes `x%d`$\equiv \dot{x} \equiv 1$ and therefore `y%d`$\equiv \dot{y}$ will contain just that partial derivative. Aside from running the transformation tool chain the `openad` script also copies the appropriate run time support files into the working directory. These support files need to be compiled and linked with the driver and the transformed code. The example includes a `Makefile`, see fig. 1.6, and the entire example can be built by executing the following commands.

```
cd $OPENDADROOT/Examples/OneMinute
make
```

This creates an executable called `driver`. The `Makefile` tests if `F90C` is defined in the environment and if not assumes `gfortran` is installed and should be used. The output generated by running `driver` is shown in fig. 1.7.

An approach without using the `openad` script is explained in sec. 2.5.1.

```
1
2  MODULE oad_intrinsics
3  use OAD_active
4  use w2f__types
5  IMPLICIT NONE
6  SAVE
7  !
8  ! **** Statements ****
9  !
10 END MODULE
11
12 SUBROUTINE head(X, Y)
13 use OAD_active
14 use w2f__types
15 use oad_intrinsics
16 IMPLICIT NONE
17 !
18 ! **** Parameters and Result ****
19 !
20 type(active) :: X
21 type(active) :: Y
22 !
23 ! **** Local Variables and Functions ****
24 !
25 REAL(w2f__8) OpenAD_aux_0
26 REAL(w2f__8) OpenAD_lin_0
27 !
28 ! **** Top Level Pragmas ****
29 !
30 !$OPENAD INDEPENDENT(X)
31 !$OPENAD DEPENDENT(Y)
32 !
33 ! **** Statements ****
34 !
35 OpenAD_aux_0 = TAN(X%v)
36 OpenAD_lin_0 = (OpenAD_aux_0*OpenAD_aux_0+INT(1_w2f__i8))
37 Y%v = OpenAD_aux_0
38 CALL sax(OpenAD_lin_0,X,Y)
39 END SUBROUTINE
```

Figure 1.4: Forward mode transformed code for fig. 1.2 (left)

```
1  program driver
2    use OAD_active
3    implicit none
4    external head
5    type(active) :: x, y
6    x%v=.5D0
7    x%d=1.0D0
8    call head(x,y)
9    print *, 'driver running for x =',x%v
10   print *, '          yields y =',y%v,' dy/dx =',y%d
11   print *, '    1+tan(x)^2-dy/dx =',1.0D0+tan(x%v)**2-y%d
12 end program driver
```

Figure 1.5: Forward mode `driver` (see file `$OPENADROOT/Examples/OneMinute/driver.f90`) for the transformed code shown in fig. 1.4

```
ifndef F90C
F90C=gfortran
endif
RTSUPP=w2f__types OAD_active
driver: $(addsuffix .o, $(RTSUPP)) driver.o head.prepped.pre.xb.x2w.w2f.post.o
        ${F90C} -o $@ $^
head.prepped.pre.xb.x2w.w2f.post.f90 $(addsuffix .f90, $(RTSUPP)) : toolChain
toolChain : head.prepped.f90
        openad -c -m f $<
%.o : %.f90
        ${F90C} -o $@ -c $<
clean:
        rm -f ad_template* OAD_* w2f__*  iaddr*
        rm -f head.prepped.pre* *.B *.xaif *.o *.mod driver driverE *~
.PHONY: clean toolChain
# the following include has explicit rules that could replace the openad script
include MakeExplRules.inc
```

Figure 1.6: Portion of `$OPENADROOT/Examples/OneMinute/Makefile` needed compile and link the forward mode example using the `openad` script

.

```
 driver running for x =  0.50000000000000000
             yields y =  0.54630248984379048       dy/dx =   1.2984464104095248
     1+tan(x)^2-dy/dx =   0.0000000000000000
```

Figure 1.7: Output from forward mode `driver` shown in fig. 1.5.

1.3.2 Reverse Mode

There is a slightly different `Makefile` and `driver` for the reverse mode. As in sec. 1.3.1 we can use the `openad` script by calling

```
cd $OPENDADROOT/Examples/OneMinuteReverse
openad -c -m rj head.prepped.f90
```

The reverse mode code generation is triggered by setting the flag `-m rj` where `rj` stands for *reverse joint* mode, see also sec. 3.4. The script produces progress output like the one shown in fig. 1.8. The resulting Fortran output is written to a file called

```
$OPENADROOT/Examples/OneMinuteReverse/head.prepped.pre.xb.x2w.w2f.post.f90
```

and sections of it are shown in fig. 1.9. In particular in fig. 1.9(top) one can observe the writing of the local partial derivative to the tape on line 6 which subsequently is retrieved in the adjoint sweep shown in fig. 1.9(bottom) on line 3 and used on line 4. The latter represents the actual propagation operation

$$\bar{x} += \frac{\partial \tan(x)}{\partial x} \cdot \bar{y} \quad ,$$

see also (3.7) on pg. 25. A simple driver that calls the transformed `head(x,y)` is shown in fig. 1.10. Note, that here we initialize `y%d` $\equiv \bar{(y)} = 1.0$ which implies that `x%d` $\equiv \bar{x}$ holds the value of the partial derivative. As in sec. 1.3.1, the `openad` script copies the appropriate run time support files into the working directory. These support files need to be compiled and linked with the driver and the transformed code. The example includes a `Makefile` (see fig. 1.11, in addition to testing if

```
openad -c -m rj head.prepped.f90
openad log: openad.2014-03-15_01:33:37.log~
preprocessing fortran
parsing preprocessed fortran
analyzing source code and translating to xaif
adjoint transformation
 getting runtime support file OAD_active.f90
 getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
 getting runtime support file ad_inline.f
 getting runtime support file OAD_cp.f90
 getting runtime support file OAD_rev.f90
 getting runtime support file OAD_tape.f90
 getting template file
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
```

Figure 1.8: Progress messages from `openad` for reverse mode.

```
1  ! taping
2  OpenAD_aux_0 = TAN(X%v)
3  OpenAD_lin_0 = (OpenAD_aux_0*OpenAD_aux_0+INT(1_w2f__i8))
4  Y%v = OpenAD_aux_0
5  double_tape(double_tape_pointer) = OpenAD_lin_0
6  double_tape_pointer = double_tape_pointer+1
7
8  ! taping end
```

```
1  ! adjoint
2  double_tape_pointer = double_tape_pointer-1
3  OpenAD_Symbol_0 = double_tape(double_tape_pointer)
4  X%d = X%d+Y%d*(OpenAD_Symbol_0)
5  Y%d = 0.0d0
6
7  ! adjoint end
```

Figure 1.9: Reverse mode transformed code sections for taping (top) and adjoint (bottom) for the code from fig. 1.2 (left)

```
1  program driver
2    use OAD_active
3    use OAD_rev
4    implicit none
5    external head
6    type(active) :: x, y
7    x%v=.5D0
8    y%d=1.0D0
9    our_rev_mode%tape=.TRUE.
10   call head(x,y)
11   print *, 'driver running for x = ',x%v
12   print *, '           yields y = ',y%v,' dy/dx = ',x%d
13   print *, '    1+tan(x)^2-dy/dx = ',1.0D0+tan(x%v)**2-x%d
14 end program driver
```

Figure 1.10: Reverse mode `driver` routine for the transformed code shown in fig. 1.9

```
ifndef F90C
F90C=gfortran
endif
ifndef CC
CC=gcc
endif
RTSUPP=w2f__types OAD_active OAD_cp OAD_tape OAD_rev
driver: $(addsuffix .o, $(RTSUPP) iaddr) driver.o head.prepped.pre.xb.x2w.w2f.post.o
        ${F90C} -o $@ $^
head.prepped.pre.xb.x2w.w2f.post.f90 $(addsuffix .f90, $(RTSUPP)) iaddr.c : toolChain
toolChain : head.prepped.f90
        openad -c -m rj $<
%.o : %.f90
        ${F90C} -o $@ -c $<
%.o : %.c
        ${CC} -o $@ -c $<
clean:
        rm -f ad_template* ad_inline.f OAD_* w2f__*  iaddr*
        rm -f head.prepped.pre.* *.B *.xaif *.o *.mod driver driverE *~
.PHONY: clean toolChain
# the following include has explicit rules that could replace the openad script
include MakeExplRules.inc
```

Figure 1.11: Portion of `$OPENADROOT/Examples/OneMinuteReverse/Makefile` needed to compile and link the reverse mode example using the `openad` script.

.

```
 driver running for x =  0.50000000000000000
              yields y =  0.54630248984379048       dy/dx =   1.2984464104095248
     1+tan(x)^2-dy/dx =   0.0000000000000000
```

Figure 1.12: Output from reverse mode `driver` shown in fig. 1.10.

`F90C` is defined in the environment it also tests for `CC` and if not assumes `gcc` is installed and should be used), The output generated by the driver is shown in fig. 1.12.

1.4 Deciding on OpenAD/F Usage Patterns

The small example in sec. 1.3 already shows two variants of applying the OpenAD/F tool. In this section we step through a number of decisions that will determine how one might use OpenAD/F for a given application. We will just give a cursory explanation of the concepts involved and refer to the detailled explanation in other sections.

The following sections may not be very relevant for small-scale applications and one may skip to sec. 5 to see if any of the examples discussed there is a good template for the application in question.

1.4.1 When is AD via source transformation appropriate?

AD via operator overloading is available through various tools such as AD02 [2], Rapsodia [39], and Adol-C [6] (for C and C++).

Language Support Because operator overloading is not available within the Fortran standard prior to Fortran90 there is no alternative for older Fortran programs unless they are migrated to the newer standards.

Type Change Another concern is the actual type change from Fortran built-in numerical types to the user defined type that triggers the execution of the AD logic. As is the case in C++ the typechange affects also formatted I/O operations, library calls, memory allocation. Consequently, additional changes other than just a global change of declarations may be required to obtain a semantically correct program. Fortran lacks a template-like language construct to easily obtain a type-changed version of the original Fortran source code while keeping the use of built-in types untouched. One can workaround this problem by extensive use of the C preprocessor or a source transformation tool to perform the type change.

Activity Analysis Unless source transformation is used, the type change will generally be applied globally to all floating point variables. Source transformation tools like OpenAD/F are able to identify the subset of program variables for which derivative computations have to be performed. This is known as *activity analysis*, see sec. 4.1.1. Identifying the active variables and performing the type change only to these variables can save a substantial amount of computational overhead.

Reverse Mode Operator overloading for reverse mode typically relies on tracing each call to an overloaded operation and saving values required for the adjoint computations. The size of such naive traces severly limits the complexity of computations for which an operator overloading based reverse mode is feasible. Source transformation tools can use program context information and data flow analysis to significantly reduce the memory requirements.

Derivative Order For the computation of higher order derivatives operator overloading is often viewed as a reasonable approach because the complexity encapsulated in each operator outweighs the overhead implied by making an extra call to the overloaded operator. Unlike for gradients and Hessians there are no large practical applications where conquering the technical difficulties of the reverse mode is justified. The first two issues may be tackled by applying a source transformation tool to perform a selective and semantically correct type change.

OpenAD/F has a type change algorithm encapsulated as one step in its transformation chain, see also sec. 4.1.3.2. This transformation step can be adapted to the use of a specific type that has overloaded operators defined, for instance as in [39].

1.4.2 When should the source code be split?

This and the following section typically apply only to models with a large code base. For small models one can proceed to sec. 1.4.3.

Splitting the source is an option to be considered for numerical models that have large codes base and have extensive debugging, I/O, monitoring, communication and other non-numerical logic. The source transformation use data dependency information in the creation of the derivative code. The built-in analysis has to be conservatively correct and therefore will likely have a conservative overestimate, for instance, of the set of active program variables for which derivative computations have to be generated. The use of common buffers, e.g. in communication or I/O logic can significantly increase the overestimate. Semantically complicated I/O constructs or other logic that is not numerical in nature may be something that is not or only incompletely covered by the source transformation tool. One should keep in mind that for a tool like OpenAD/F that is funded by research grants, there is little academic benefit to be had from covering esoteric language features as opposed to concentrating on the efficiency of the code generated for the typical numerical computations. Consequently, filtering out logic that is not part of the numerical core not only will increase the likelihood that the tool can transform the code but may also increase the efficiency of the generated code by shrinking the conservative overestimates of the code analysis.

Often the natural way of splitting the code is to identify the source files that pertain to the numerical core and transform only the code in these files. The analysis then optimistically assumes the calls to the "external" methods will not impact the relevant data dependencies.

1.4.3 When Should One Use Reverse Mode Instead of Forward Mode?

The simplest rule of thumb is to use reverse mode when the number n of inputs is significantly larger than the number m of outputs as is typically the case with gradients and Hessians. Because of the technical hurdles of the reverse mode one should also consider if the model has characteristics such as sparsity or partial separability that would permit using the forward mode even for a nominally large number of independents. For large scale models one can also consider to separate the computation if the dimension of an intermediate interface is significantly smaller or larger than n and m and selectively apply forward and reverse mode to the respective parts. For higher order derivative computations n is typically low and consequently one will tend toward forward mode. While in principle a reverse mode sweep could be injected there too at any order o the efficiency improvement remains at n while the memory requirements for the reverse sweep grow linearly with o.

1.4.4 When Should One Use Checkpointing?

This section applies only to reverse mode computations of models with a long runtime or large nonlinear parts; otherwise proceed to sec. 2.

The memory requirements for the reverse mode depend on the extent of nonlinearity in the model and are roughly proportional to the run time of the numerical core of the model computation. The memory is used to trace the forward computation and retain values needed for the computation of the adjoints. For large computations the memory requirements for the entire forward trace and all needed values are prohibitive and the common tradeoff is to restart the forward run from some checkpoint and tracing only small sections with manageable memory requirements. Various checkpointing schemes have been devised and examples can be found in sec. 5.2 and sec. 1.4.5.

1.4.5 When should `make` rules be used instead of the `openad` script?

The `openad` script is intended only as wrapper for the transformation tool chain of simple models. Complicated models will have a predefined build process using `makefile`s and consequently it is more appropriate to integrate the stages of the transformation tool chaine explicitly into that build process. This gives access to all the options at each transformation stage and this extra flexibility will be needed for large models. An example for the setup with `make` is given in sec. 5.2.

Chapter 2

Usage Details

This section will explain the tool usage in detail. following contains brief instructions how to obtain and use OpenAD/F. While the principal approach will remain the same, future development may introduce slight changes. The reader is encouraged to refer to the up to date instructions on the OpenAD/F website [35].

2.1 Download and Build

All components are open source and readily available for download as `tar` files or via version control checkout. The website

http://www.mcs.anl.gov/openad

provides details on downloading and building the tool chain. The build process essentially consists setting up the environment as described in sec. 2.2 followed invoking

```
make
```

2.2 OpenAD/F Environment

Building and running the OpenAD/F tool chain as well as optional updates from version control repositories requires certain environment variables to be set. This is done as follows.

1. change directory into the OpenAD/F source tree:
   `cd OpenAD`
2. set the environment
   - for `shell`/`ksh`/`bash` users with
     `source ./setenv.sh`
   - for `csh`/`tcsh` users with
     `source ./setenv.csh`

From now on we assume the environment variables have been set and we will refer to them as needed.

2.3 Code Preparation with Pragmas (incompl.)

The tool chain recognizes pragmas of the format

```
$openad <pragma argument>
```

and the specific *<pragma argument>* options are listed below.

`independent(`*<variable name>*`)` At least one of these pragmas is required in the source code to be transformed to identify program variables that are considered independent. The pragma is used to initialize the activity analysis (see also sec. 4.1.1). The *¡variable name¿* must be a program variable visible in the context in which the pragma is placed. The front-end and the subsequent tool chain components will parse *¡variable name¿* as a properly scoped variable. Index

expressions for *¡variable name¿* such as `X[2]` are currently not meaningful for the activity analysis. An example is shown in fig. 1.2(right).

`dependent(<variable name>)` At least one of these pragmas is required in the source code to be transformed to identify program variables that are considered dependent. The pragma functions similarly to the above mentioned `independent` pragma. An example is shown in fig. 1.2(right).

`xxx template <template file>` This is an optional pragma. The postprocessor will recognize the procedure in which the pragma occurs (or which immediately follows) as the target to which the template should be applied, see also sec. 4.2.4. The *¡template file¿* needs to be a file either with an absolute path or a path relative to the working directory from which the postprocesor is invoked. The tool chain has to parse and pass through the pragma to the postprocessor while retaining the pragma location. Note that currently the Open64 front-end will within a module `contains` block move all such pragmas into the beginning of the `contains` block and therefore placing the pragma *inside* the procedure to which it should be applied.

`xxx simple loop` This is an optional pragma. The transformation recognizes the loop immediately following this declaration and all loops nested within as *simple loops*, see also sec. 3.3.

2.4 Running the tool chain with the `openad` script

This section applies only to very simple models. For reasonably complicated computations proceed to sec. 2.5.

The components of OpenAD/F transform the code in a predetermined sequence of steps. Depending on the particular problem there are certain variations to the tool chain execution that achieve a better performance of the generated code. The most common setups are encapsulated in the Python script `$OPENADROOT/bin/openad`. The script is part of the skeleton environment that is used to download and build OpenAD/F and relies on the same environment setup that also puts the script into the `PATH`. Therefore user starts with steps 1 and 2 from sec. 2.1. Invoking the script with the `-h` option displays the following command-line options.

```
Usage: openad [options] <fortran-file>

Options:
  -h, --help            show this help message and exit
  -m MODE, --mode=MODE  basic transformation mode with MODE being one of: t =
                        tracing; rs = reverse split; rj = reverse joint; fv =
                        forward vector; f = forward; (default is forward)
  -d DEBUG, --debug=DEBUG
                        the debugging level
  -i, --interactive     requires to confirm each command
  -k, --keepGoing       keep going despite errors
  -c, --copy            copy run time support files instead of linking them
  -n, --noAction        display the pipeline commands, do not run them
```

The most important of these options is the mode specification. The default is the forward (or tangent-linear) mode, which is described in sec. 4.1.3.4. The reverse (or adjoint) mode is described in sec. 4.1.3.9; the "split" and "joint" variants refer to two different schemes for call graph reversal (see sec. 3.4).

As an example, assume we wish to create a tangent-linear version of a code in a file named `head.f`. Invoking the command `openad head.f` will create the transformed file `head.pre.xb.x2w.w2f.post.f`, where the following messages appear as output during the transformation process.

```
openad log: openad.2014-03-15_01:34:01.log~
preprocessing fortran
parsing preprocessed fortran
analyzing source code and translating to xaif
tangent linear transformation
 getting runtime support file OAD_active.f90
 getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
```

The script also links (or copies with `-c`) a simple version of the necessary support files. As with the tool chain itself, a computationally complex application will likely want to adapt the support files. For larger projects it is obviously appropriate to customize the sequence by adding the steps outlined in sec. 2.5 to a `Makefile`. The steps performed by the `openad` script can serve as an initial guideline for the manual invocations. The script dumps the commands without executing them when the `-n` flag is passed. For the above example the output is shown in fig. 2.1.

```
# preprocessing fortran
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/preProcess.py -m f -o head.pre.f head.f
# parsing preprocessed fortran
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/crayf90/sgi/mfef90 -z -F -N132 head.pre.f
# analyzing source code and translating to xaif
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/whirl2xaif -o head.pre.xaif -n head.pre.B
# tangent linear transformation
/sandbox/utke/CronTest/OpenAD/xaifBooster/../xaifBooster/algorithms/BasicBlockPreaccumulation/driver/oadDriver -c \
        /sandbox/utke/CronTest/OpenAD/xaif/schema/examples/inlinable_intrinsics.xaif -s /sandbox/utke/CronTest/OpenAD/xaif/schema -i \
        head.pre.xaif -o head.pre.xb.xaif
#  getting runtime support file OAD_active.f90
ln -sf /sandbox/utke/CronTest/OpenAD/runTimeSupport/scalar/OAD_active.f90 ./
#  getting runtime support file w2f__types.f90
ln -sf /sandbox/utke/CronTest/OpenAD/runTimeSupport/all/w2f__types.f90 ./
#  getting runtime support file iaddr.c
ln -sf /sandbox/utke/CronTest/OpenAD/runTimeSupport/all/iaddr.c ./
# translating transformed xaif to whirl
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/xaif2whirl head.pre.B head.pre.xb.xaif
# unparsing transformed whirl to fortran
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/whirl2f/whirl2f -openad head.pre.xb.x2w.B
# postprocessing transformed fortran
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/postProcess.py --infoUnitFile=w2f__types.f90 -m f -o \
        head.pre.xb.x2w.w2f.post.f head.pre.xb.x2w.w2f.f
```

Figure 2.1: Actions to be performed for the default invocation of the `openad` script; we folded long lines.

2.5 Explicitly invoking the tool chain elements

As explained in sec. 2.4, the steps performed by the `openad` script can serve as an initial guideline for the manual invocation of the tool chain components. Rather than an abstract explanation of the steps we will just show the explicit `make` rules for the examples in sects. 1.3.1,1.3.2 where we have a prepared source file named `head.prepped.f90`.

2.5.1 Forward Mode

For forward mode the rules are shown in fig. 2.2. By changing directory to `$OPENADROOT/Examples/OneMinute` and invoking

```
make driverE
```

one triggers the explicit rules shown in fig. 2.2 and the output including compile and link steps will look like the output shown in fig. 2.3 and among various extra compiler invocation one will recognize the steps listed in fig. 2.1.

Regarding the individual steps, details may be found in the following references.

1. **Canonicalization:** The optional canonicalization step not considered in the rules in fig. 2.2 is discussed in sec. 4.2.1.

2. **Parsing:** The input source is parsed by the Open64 front-end, see also sec. 4.2.2 and one obtains a binary file with a `.B` extension contain the representation in `whirl`.

3. **Translating to XAIF:** The `whirl` is analyzed and the results are translated into XAIF, see also sec. 4.2.3.

4. **Transforming the XAIF:** The XAIF representation is transformed by the tangent linear transformation algorithm. The driver for the transformation is located at

   ```
   $XAIFBOOSTERROOT/xaifBooster/algorithms/BasicBlockPreaccumulation/driver/oadDriver
   ```

   see also sec. 4.1.3.4.

5. **Back - translating the transformed XAIF:** From the transformed XAIF a corresponding transformed `whirl` representation is created, see also sec. 4.2.3.

6. **Unparsing to transformed Fortran:** From the transformed `whirl` we unparse to Fortran using the back-end uparser provided by Open64, see also sec. 4.2.2.

7. **Post processing:** Various constructs that are ancillary to the transformation, such as the actual name of the value and the derivative components are done at this stage, see also sec. 4.2.4. **Note** the command-line flag `-f` passed to the post processor to indicate forward mode. This flag is required to be consistent with the chosen transformation.

8. **Compiling/Linking:** The transformed sources, the driver and the runtime support files need to be compiled and linked. This is briefly discussed in sec. 1.3.1 but see also sec. 2.6.

```
# explicit make rules
# preprocess
head.prepped.pre.f90: head.prepped.f90
	${OPENADFORTTK_BASE}/tools/SourceProcessing/preProcess.py -m f --inputFormat=free -o $@ $<
# fortran -> whirl
head.prepped.pre.B: head.prepped.pre.f90
	${OPEN64ROOT}/crayf90/sgi/mfef90 -z -F -N132 $<
# whirl -> xaif
head.prepped.pre.xaif : head.prepped.pre.B
	${OPENADFORTTKROOT}/bin/whirl2xaif -n -o $@ $<
# xaif -> xaif'
head.prepped.pre.xb.xaif : head.prepped.pre.xaif
	${XAIFBOOSTERROOT}/xaifBooster/algorithms/BasicBlockPreaccumulation/driver/oadDriver \
	-c ${XAIFSCHEMAROOT}/schema/examples/inlinable_intrinsics.xaif \
	-s ${XAIFSCHEMAROOT}/schema -i $< -o $@
# xaif' -> whirl'
head.prepped.pre.xb.x2w.B : head.prepped.pre.xb.xaif
	${OPENADFORTTKROOT}/bin/xaif2whirl head.prepped.pre.B $<
# whirl' -> fortran'
head.prepped.pre.xb.x2w.w2f.f: head.prepped.pre.xb.x2w.B
	${OPEN64ROOT}/whirl2f/whirl2f -openad $<
# postprocess
head.prepped.pre.xb.x2w.w2f.post.E.f90: head.prepped.pre.xb.x2w.w2f.f
	${OPENADFORTTK_BASE}/tools/SourceProcessing/postProcess.py -m f --outputFormat=free \
	--infoUnitFile=w2f__types.E.f90 -o $@ $<
# we add the .E extension  here to distinguish the targets from the
# rules using the openad script
driverE: $(addsuffix .E.o, $(RTSUPP)) driver.o head.prepped.pre.xb.x2w.w2f.post.E.o
	${F90C} -o $@ $^
w2f__types.E.f90: ${OPENADROOT}/runTimeSupport/all/w2f__types.f90
	cp -f $< $@
OAD_active.E.f90: ${OPENADROOT}/runTimeSupport/scalar/OAD_active.f90
	cp -f $< $@
```

Figure 2.2: Contents of `$OPENADROOT/Examples/OneMinute/MakeExplRules.inc` included in the example's `Makefile` providing explicit rules to replace the actions of the `openad` script for forward mode.

```
make[1]: Entering directory `/sandbox/utke/CronTest/OpenAD/Examples/OneMinute'
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/all/w2f__types.f90 w2f__types.E.f90
gfortran -o w2f__types.E.o -c w2f__types.E.f90
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/scalar/OAD_active.f90 OAD_active.E.f90
gfortran -o OAD_active.E.o -c OAD_active.E.f90
gfortran -o driver.o -c driver.f90
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/preProcess.py -m f --inputFormat=free -o head.prepped.pre.f90 \
	head.prepped.f90
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/crayf90/sgi/mfef90 -z -F -N132 head.prepped.pre.f90
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/whirl2xaif -n -o head.prepped.pre.xaif head.prepped.pre.B
/sandbox/utke/CronTest/OpenAD/xaifBooster/../xaifBooster/algorithms/BasicBlockPreaccumulation/driver/oadDriver -c \
	/sandbox/utke/CronTest/OpenAD/xaif/schema/examples/inlinable_intrinsics.xaif -s /sandbox/utke/CronTest/OpenAD/xaif/schema -i \
	head.prepped.pre.xaif -o head.prepped.pre.xb.xaif
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/xaif2whirl head.prepped.pre.B head.prepped.pre.xb.xaif
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/whirl2f/whirl2f -openad head.prepped.pre.xb.x2w.B
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/postProcess.py -m f --outputFormat=free --infoUnitFile=w2f__types.E.f90 -o \
	head.prepped.pre.xb.x2w.w2f.post.E.f90 head.prepped.pre.xb.x2w.w2f.f
gfortran -o head.prepped.pre.xb.x2w.w2f.post.E.o -c head.prepped.pre.xb.x2w.w2f.post.E.f90
gfortran -o driverE w2f__types.E.o OAD_active.E.o driver.o head.prepped.pre.xb.x2w.w2f.post.E.o
make[1]: Leaving directory `/sandbox/utke/CronTest/OpenAD/Examples/OneMinute'
```

Figure 2.3: Output of invoking `make driverE` in `$OPENADROOT/Examples/OneMinute` .

```
# explicit make rules
# preprocess
head.prepped.pre.f90: head.prepped.f90
        ${OPENADFORTTK_BASE}/tools/SourceProcessing/preProcess.py -m r --inputFormat=free -o $@ $<
# fortran -> whirl
head.prepped.pre.B: head.prepped.pre.f90
        ${OPEN64ROOT}/crayf90/sgi/mfef90 -z -F -N132 $<
# whirl -> xaif
head.prepped.pre.xaif : head.prepped.pre.B
        ${OPENADFORTTKROOT}/bin/whirl2xaif -n -o $@ $<
# xaif -> xaif'
head.prepped.pre.xb.xaif : head.prepped.pre.xaif
        ${XAIFBOOSTERROOT}/xaifBooster/algorithms/BasicBlockPreaccumulationReverse/driver/oadDriver \
        -c ${XAIFSCHEMAROOT}/schema/examples/inlinable_intrinsics.xaif \
        -s ${XAIFSCHEMAROOT}/schema -i $< -o $@
# xaif' -> whirl'
head.prepped.pre.xb.x2w.B : head.prepped.pre.xb.xaif
        ${OPENADFORTTKROOT}/bin/xaif2whirl head.prepped.pre.B $<
# whirl' -> fortran'
head.prepped.pre.xb.x2w.w2f.f: head.prepped.pre.xb.x2w.B
        ${OPEN64ROOT}/whirl2f/whirl2f -openad $<
# postprocess
head.prepped.pre.xb.x2w.w2f.post.E.f90: head.prepped.pre.xb.x2w.w2f.f
        ${OPENADFORTTK_BASE}/tools/SourceProcessing/postProcess.py -m r \
        --infoUnitFile=w2f__types.E.f90 \
        -i ${OPENADROOT}/runTimeSupport/simple/ad_inline.f              \
        -t ${OPENADROOT}/runTimeSupport/simple/ad_template.joint.f      \
        --outputFormat=free -o $@ $<
# we add the .E extension  here to distinguish the targets from the
# rules using the openad script
driverE: $(addsuffix .E.o, $(RTSUPP)) driver.o head.prepped.pre.xb.x2w.w2f.post.E.o
        ${F90C} -o $@ $^
w2f__types.E.f90: ${OPENADROOT}/runTimeSupport/all/w2f__types.f90
        cp -f $< $@
%.E.f90: ${OPENADROOT}/runTimeSupport/scalar/%.f90
        cp -f $< $@
%.E.f90: ${OPENADROOT}/runTimeSupport/simple/%.f90
        cp -f $< $@
ad_template.f: ${OPENADROOT}/runTimeSupport/simple/ad_template.joint.f
        cp -f $< $@
```

Figure 2.4: Contents of `$OPENADROOT/Examples/OneMinuteReverse/MakeExplRules.inc` included in the example's `Makefile` providing explicit rules to replace the actions of the `openad` script for reverse mode.

2.5.2 Reverse Mode

This section follows the same model as sec. 2.5.1, only for reverse mode. A more elaborate reverse mode example can be found in sec. 5.2 and the experienced user may want to jump to that example.

For reverse mode the rules are shown in fig. 2.4. By changing directory to `$OPENADROOT/Examples/OneMinuteReverse` and invoking

```
make driverE
```

one triggers the explicit rules shown in fig. 2.4 and the output including compile and link steps will look like the output shown in fig. 2.5 and among various extra compiler invocation one will recognize the steps listed in fig. 2.1.

Regarding the individual steps, details may be found in the same references given in sec. 2.5.1. The following differences are noteworthy.

4. **Transforming the XAIF:** The XAIF representation is transformed by the adjoint transformation algorithm. The driver for the transformation is located at

   `$XAIFBOOSTERROOT/xaifBooster/algorithms/BasicBlockPreaccumulationReverse/driver/oadDriver`

   see also sec. 4.1.3.9.

7. **Post processing:** Various constructs that are ancillary to the transformation, such as the actual name of the value and the derivative components are done at this stage, see also sec. 4.2.4. **Note** that unlike in forward mode the command-line flag `-f` is not passed to the post processor.

8. **Compiling/Linking:** There are additional runtime support files need to be compiled and linked beyond the set of files for forward mode. This is briefly discussed in sec. 1.3.2 but see also sec. 2.6.

```
make[1]: Entering directory '/sandbox/utke/CronTest/OpenAD/Examples/OneMinuteReverse'
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/all/w2f__types.f90 w2f__types.E.f90
gfortran -o w2f__types.E.o -c w2f__types.E.f90
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/scalar/OAD_active.f90 OAD_active.E.f90
gfortran -o OAD_active.E.o -c OAD_active.E.f90
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/simple/OAD_cp.f90 OAD_cp.E.f90
gfortran -o OAD_cp.E.o -c OAD_cp.E.f90
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/simple/OAD_tape.f90 OAD_tape.E.f90
gfortran -o OAD_tape.E.o -c OAD_tape.E.f90
cp -f /sandbox/utke/CronTest/OpenAD/runTimeSupport/simple/OAD_rev.f90 OAD_rev.E.f90
gfortran -o OAD_rev.E.o -c OAD_rev.E.f90
gfortran -o driver.o -c driver.f90
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/preProcess.py -m r --inputFormat=free -o head.prepped.pre.f90 \
       head.prepped.f90
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/crayf90/sgi/mfef90 -z -F -N132 head.prepped.pre.f90
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/whirl2xaif -n -o head.prepped.pre.xaif head.prepped.pre.B
/sandbox/utke/CronTest/OpenAD/xaifBooster/../xaifBooster/algorithms/BasicBlockPreaccumulationReverse/driver/oadDriver -c \
       /sandbox/utke/CronTest/OpenAD/xaif/schema/examples/inlinable_intrinsics.xaif -s /sandbox/utke/CronTest/OpenAD/xaif/schema -i \
       head.prepped.pre.xaif -o head.prepped.pre.xb.xaif
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/xaif2whirl head.prepped.pre.B head.prepped.pre.xb.xaif
/sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/whirl2f/whirl2f -openad head.prepped.pre.xb.x2w.B
/sandbox/utke/CronTest/OpenAD/OpenADFortTk/tools/SourceProcessing/postProcess.py -m r --infoUnitFile=w2f__types.E.f90 -i \
       /sandbox/utke/CronTest/OpenAD/runTimeSupport/simple/ad_inline.f -t \
       /sandbox/utke/CronTest/OpenAD/runTimeSupport/simple/ad_template.joint.f --outputFormat=free -o head.prepped.pre.xb.x2w.w2f.post.E.f90 \
       head.prepped.pre.xb.x2w.w2f.f
gfortran -o head.prepped.pre.xb.x2w.w2f.post.E.o -c head.prepped.pre.xb.x2w.w2f.post.E.f90
gfortran -o driverE w2f__types.E.o OAD_active.E.o OAD_cp.E.o OAD_tape.E.o OAD_rev.E.o driver.o head.prepped.pre.xb.x2w.w2f.post.E.o
rm OAD_rev.E.f90 OAD_active.E.f90 OAD_cp.E.f90 OAD_tape.E.f90
make[1]: Leaving directory '/sandbox/utke/CronTest/OpenAD/Examples/OneMinuteReverse'
```

Figure 2.5: Output of invoking `make driverE` in `$OPENADROOT/Examples/OneMinuteReverse` ; we folded long lines.

2.6 Compiling and Linking

Examples for compiling and linking the transformed source code can be found in sec. 1.3 and sec. 5. Compared to the original model build process the following changes have to be accomodated.

- To facilitate the AD code analysis and comprehensive transformation one has to extract the numerical core into a single file, transform that file and reintegrate the transformed source into the build process.
- At various stages in the tool chain temporary variables may be introduced and one has to be aware of assumptions on the default precision of variables, typically specified during the compile step with flags such as `-r8` or `-i4`. The individual `make` rules mentioned in sects. 2.5.1,2.5.2 may have to be adjusted accordingly.
- There are simple implementations for runtime support files. These simple implementations for checkpointing and taping etc. are properly working but may not be the most efficient solution for a given application or hardware. Consequently, one should consider modifying or reimplementing the runtime support files for these aspects.

2.6.1 Runtime Support Files

All support files discussed in this section can be found in the subdirectories under

`$OPENADROOT/runTimeSupport`

and are the source for the examples used in this Manual.

2.6.1.1 Front-End Definitions

All Fortran produced by `whirl2f`, see sec. 4.2.2, needs definitions for values supplied as `kind` parameters. These values have standard names within the `whirl2f`-generated code. The definitions can be found in

`$OPENADROOT/runTimeSupport/all/w2f__types.f90`

Modifications to this file should be done judiciously to avoid cases where identical `kind` values lead to duplicate definitions of module routines for the OpenAD active type that differ only by that kind value.

2.6.1.2 Active Type

The active type definitions supplied here cover simple examples of active types. They can easily be extended for other experiments, e.g. an on-demand allocation of derivative data at runtime, runtime activity flags etc. The definitions cover the data type, conversion routines from active to passive, and propagation operations.

```
saxpy_d0_a0_a0(a,x,y)
saxpy_l0_a0_a0(a,x,y)
saxpy_i0_a0_a0(a,x,y)
saxpy_d0_a0_a1(a,x,y)
saxpy_d0_a1_a1(a,x,y)
saxpy_d0_a2_a2(a,x,y)
saxpy_d1_a0_a1(a,x,y)
saxpy_d1_a1_a1(a,x,y)
saxpy_l1_a1_a1(a,x,y)
saxpy_i1_a1_a1(a,x,y)
saxpy_a1_a1_a1(a,x,y)
saxpy_d2_a0_a2(a,x,y)
saxpy_d2_a2_a2(a,x,y)
saxpy_r0_a0_a0(a,x,y)
saxpy_r0_a1_a1(a,x,y)
saxpy_r1_a0_a1(a,x,y)
saxpy_r1_a1_a1(a,x,y)
```

```
sax_d0_a0_a0(a,x,y)
sax_l0_a0_a0(a,x,y)
sax_i0_a0_a0(a,x,y)
sax_d0_a0_a1(a,x,y)
sax_d0_a0_a2(a,x,y)
sax_d0_a1_a1(a,x,y)
sax_d0_a2_a2(a,x,y)
sax_d0_a3_a3(a,x,y)
sax_d1_a0_a1(a,x,y)
sax_d1_a1_a1(a,x,y)
sax_l1_a1_a1(a,x,y)
sax_i1_a1_a1(a,x,y)
sax_d2_a0_a2(a,x,y)
sax_d2_a2_a2(a,x,y)
sax_r0_a0_a0(a,x,y)
```

```
setderiv_a0_a0(y,x)
setderiv_a1_a0(y,x)
setderiv_a1_a1(y,x)
setderiv_a2_a0(y,x)
setderiv_a2_a2(y,x)
setderiv_a3_a3(y,x)
set_neg_deriv_a0_a0(y,x)
set_neg_deriv_a1_a1(y,x)
inc_deriv_a0_a0(y,x)
inc_deriv_a1_a1(y,x)
inc_deriv_a2_a2(y,x)
dec_deriv_a0_a0(y,x)
dec_deriv_a1_a1(y,x)
dec_deriv_a2_a2(y,x)
zero_deriv_a0(x)
zero_deriv_a1(x)
zero_deriv_a2(x)
zero_deriv_a3(x)
zero_deriv_a4(x)
```

Figure 2.6: Propagation routines extracted from `$OPENADROOT/runTimeSupport/scalar/OAD_active.f90`

- conversion routines between active and passive data overloaded as module procedures for various precisions and shapes with the following variations.

  `convert_[d|r|a][0-7]_[d|r|a][0-7](convertTo, convertFrom)`

- propagation routines overloaded as module procedures for various precisions and shapes listed in fig. 2.6

In the above we abbreviate

`i`	integer (4 byte)
`l`	inetger (8 byte)
`r`	float (4 btye)
`d`	float (8 byte)
`a`	active type

and the following digit $\in [0-7]$ indicates the respective shape.

2.6.1.2.1 Scalar An active type for propagation of scalar derivative components is defined in[1]

`$OPENADROOT/runTimeSupport/scalar/OAD_active.f90`

2.6.1.2.2 Vector An active type for propagation of a vector of length `max_deriv_vec_len` of derivative components is defined in[1]

`$OPENADROOT/runTimeSupport/vector/OAD_active.f90`

For the sake of simplicity with the demonstration examples we have here hard coded the value of `max_deriv_vec_len`. This value should be adjusted for a given application context either to the number of independent variables or a suitable slice size when repeated computation over slices (aka "strip mining") is to be done.
This file is created from `$OPENADROOT/runTimeSupport/genBase/OAD_active.F90` by preprocessing.

2.6.1.3 Taping

A simple implementation for tape storage can be found in

`$OPENADROOT/runTimeSupport/simple/OAD_tape.f90`

which goes together with the subroutines defined in

`$OPENADROOT/runTimeSupport/simple/ad_inline.f`

[1] This file is created from `$OPENADROOT/runTimeSupport/genBase/OAD_active.F90` by preprocessing.

which are inlined by the post processor, see sec. 4.2.4. For dynamically allocated taping space see the respective files in the `cpToFile` subdirectory.

2.6.1.4 Reversal State

A module containing global reversal scheme state definitions defined in

`$OPENADROOT/runTimeSupport/simple/OAD_rev.f90`

The split and joint reversal schemes using the state are enabled via the templates

`$OPENADROOT/runTimeSupport/simple/ad_template.joint.f`

and

`$OPENADROOT/runTimeSupport/simple/ad_template.split.f`

respectively.

2.6.1.5 Checkpointing (incompl.)

checkpointing (only for adjoint models, see `runTimeSupport/simple/OpenAD_checkpoints.f90`)

2.6.1.6 PostProcessor - Inlining (incompl.)

2.6.1.7 PostProcessor - Templates (incompl.)

2.6.1.8 Trace (incompl.)

2.6.1.9 Reversal with Revolve

A implementation of the *Revolve* reversal scheme [21] as a Fortran library is provided in the `$OPENADROOT/RevolveF9X` component that is part of the optional components (see 'extras' in sec. 4.3.1). The library interface is self explanatory and its use is illustrated in the examples `ADMrevolve` and `ADMrevolve2` found under `$OPENADROOT/Examples` (see sec. 5).

To adapt a given loop to revolve one should use the templates given in the above examples. Care should be taken with the interpretation of the `iteration` member of an `rvAction` instance. Within the Revolve implementation the loop iterations are assumed to run from 0 to $s-1$ where s is the argument value given to `rvInit` as `steps`. The value i of the `iteration` member is to be interpreted as follows with respect to a *current* iteration state c that is initialized to 0.

- for `rvRestore`: the argument checkpoint for iteration i is being restored; set $c = i$
- for `rvForward`: perform iterations $[c, i-1]$ with c as the loop variable, that is, after this loop the value of c should be $i-1$.

Using this recipe c will attain all the iteration values in $[0, s-1]$ until the completion of the Revolve loop and i can be interpreted as the respective "next" iteration not yet executed. For values of `rvAction%actionFlag` other than the two values listed above no meaning is assigned to `iteration`.

Chapter 3

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 [22]. The interested reader should also consider the proceedings of AD conferences [19, 10, 14, 13, 11].

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

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

$$\boldsymbol{J}\boldsymbol{S} \quad \text{and} \quad \boldsymbol{J}^T\boldsymbol{S} \tag{3.1}$$

3.1 Computational Graphs

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

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

$$\dot{v}_j = \sum_i \frac{\partial \phi_j}{\partial v_i} \dot{v}_i \quad \text{for} \quad j = 1, \ldots, q+m. \tag{3.2}$$

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

$$\overline{v}_i = \sum_j \frac{\partial \phi_j}{\partial v_i} \overline{v}_j \quad \text{for} \quad j = 1, \ldots, q+m. \tag{3.3}$$

In practice, the sum in (3.3) is often split into individual increments associated with each statement in which v_i occurs as an argument $\overline{v}_i = \overline{v}_i + \overline{v}_j * \frac{\partial \phi_j}{\partial v_i}$.

Equations (3.2) and (3.3) can be used to accumulate the (local) Jacobian $\boldsymbol{J}(G)$ of G, see also sec. 3.2.

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

3.2 Elimination Methods

Let $\boldsymbol{f}$ represent a single basicblock that is subject to preaccumulation. For notational simplicity and without loss of generality we assume that the dependent variables are mutually independent. This situation can always be reached by introducing auxiliary assignments. Consider the small example in fig. 3.1. Reformulating the example in terms of results of elemental

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

Figure 3.1: Example of code contained in a basicblock

operations ϕ assigned to unique intermediate variables v we have

$$\begin{aligned} &v_1 = v_{-1} + v_0;\ v_2 = \sin(v_0);\ v_3 = v_1 + v_2;\ v_4 = v_1 * v_3; \\ &v_5 = \sqrt{v_3};\ v_6 = \cos(v_4);\ v_7 = -v_5 \quad . \end{aligned} \tag{3.4}$$

In the tool this modified representation is created as part of the linearization transformation, see sec. 4.1.3.3. In fig. 3.2 (a) we show the computational graph G for this representation. The edges $(i,j) \in E$ are labeled with partial derivatives c_{ji}, for instance, in the example we have $c_{64} = -\sin(v_4)$. In the tool, this graph is generated as part of the algorithm described in sec. 4.1.3.4. 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 [20]. An edge (i,k) with $i \prec j$ and $j \prec k$ is labeled with $c_{ki} + c_{kj} \cdot c_{ji}$ if it existed before the elimination of j. We say that *absorption* takes place. Otherwise, (i,k) is generated as *fill-in* and labeled with $c_{kj} \cdot c_{ji}$ The vertex j is removed from G together with all incident edges. fig. 3.2 (b) shows the result of eliminating vertex 3 from the graph in fig. 3.2 (a).

An edge (i,j) is *front eliminated* by connecting i with all successors of j, followed by removing (i,j) [29]. The corresponding structural modifications of the c-graph in fig. 3.2 (a) are shown in fig. 3.2 (c) for front elimination of $(1,3)$. The new edge labels are given as well. Edge-front elimination eventually leads to intermediate vertices in G becoming *isolated*; that is, these vertices no longer have predecessors. Isolated vertices are simply removed from G together with all incident edges.

Back elimination of an edge $(i,j) \in E$ results in connecting all predecessors of i with j [29]. The edge (i,j) itself is removed from G. The back elimination of $(3,4)$ from the graph in fig. 3.2 (a) is illustrated in fig. 3.2 (d). Again, vertices can become isolated as a result of edge-back elimination because they no longer have successors. Such vertices are removed from G.

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

$$c_{ki} = c_{ji} * c_{kj} \ \ (+c_{ki}) \tag{3.5}$$

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

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

Vertex and edge elimination steps have an interpretation in terms of vertices and edges of G, whereas face elimination is performed on the corresponding directed line graph $\mathcal{G}$. Following [30], we define the directed line graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ corresponding to $G = (V, E)$ as follows:

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

and

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

That is, we add a source vertex $\oplus$ and a sink vertex $\ominus$ to G connecting all independents to $\oplus$ and all dependents to $\ominus$. $\mathcal{G}$ has a vertex $v \in \mathcal{V}$ for each edge in the extended G, and $\mathcal{G}$ has an edge $e \in \mathcal{E}$ for each pair of adjacent edges in G. fig. 3.3 gives an example of constructing the directed line graph in (b) from the graph in (a) which is the graph from fig. 3.2(a) extended by

[1] Additions are not necessarily directly coupled.

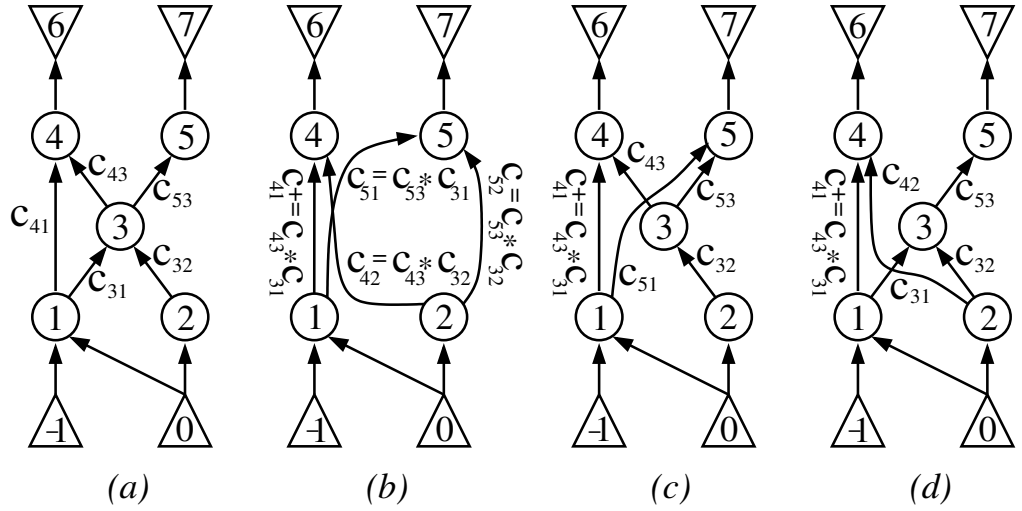


Figure 3.2: (a) Computational graph G for (3.4), (b) eliminate vertex 3 from G, (c) front eliminate edge $(1,3)$ from G, (d) back eliminate edge $(3,4)$ from G

the source and sink vertex. All intermediate vertices $\boxed{i,j} \in \mathcal{V}$ inherit the labels c_{ji}. In order to formalize face elimination, it is advantageous to move away from the double-index notation and use one that is based on a topological enumeration of the edges in G. Hence, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ becomes a DAG with $\mathcal{V} \subset I\!N$ and $\mathcal{E} \subset I\!N \times I\!N$ and certain special properties. The set of all predecessors of $j \in \mathcal{V}$ is denoted as P_j. Similarly, S_j denotes the set of its successors in $\mathcal{G}$. A vertex $j \in \mathcal{V}$ is called *isolated* if either $P_j = \emptyset$ or $S_j = \emptyset$. Face elimination is defined in [30] between two incident intermediate vertices i and j in $\mathcal{G}$ as follows:

1. If there exists a vertex $k \in \mathcal{V}$ such that $P_k = P_i$ and $S_k = S_j$, then set $c_k = c_k + c_j c_i$ *(absorption)*; else $\mathcal{V} = \mathcal{V} \cup \{k'\}$ with a new vertex k' such that $P_{k'} = P_i$ and $S_{k'} = S_j$ *(fill-in)* and labeled with $c_{k'} = c_j c_i$.
2. Remove (i,j) from $\mathcal{E}$.
3. Remove $i \in \mathcal{V}$ if it is isolated. Otherwise, if there exists a vertex $i' \in \mathcal{V}$ such that $P_{i'} = P_i$ and $S_{i'} = S_i$, then
   - set $c_i = c_i + c_{i'}$ *(merge)*;
   - remove i'.
4. Repeat Step 3 for $j \in \mathcal{V}$.

In fig. 3.3 (c) we show the elimination of $(i,j) \in \mathcal{E}$, where $i = \boxed{1,3}$ and $j = \boxed{3,4}$.

A complete face elimination sequence σ_f yields a tripartite directed line graph $\sigma_f(\mathcal{G})$ that can be transformed back into the bipartite graph representing the Jacobian $\boldsymbol{f}'$. We note that any G can be transformed into the corresponding $\mathcal{G}$ but that a back transformation generally is not possible once face elimination steps have been applied. Therefore, face eliminations can generally not precede vertex and edge eliminations. In OpenAD these eliminations are implemented in the algorithms described in sec. 4.1.3.5 and sec. 4.1.3.6.

In a source transformation context of OpenAD/F the operations (3.5) are expressed as actual code, the Jacobian accumulation code. For our example code from fig. 3.1 the code computing the local partials in conjunction with the function value is shown in fig. 3.4. [2] In OpenAD/F the operations in fig. 3.4 are generated by the transformation algorithm discussed in sec. 4.1.3.3. The operations induced by the eliminations on the graph can be expressed in terms of the auxiliary variables c_{ji}. For our example, a forward vertex elimination in the order (1,2,3,4,5) in G (fig. 3.2), leads to the following Jacobian accumulation code. In the tool the operations shown in fig. 3.5 are generated by the transformation algorithm discussed in sec. 4.1.3.4.

3.3 Control Flow Reversal

Because the code for a $\boldsymbol{f}$ generally contains control flow constructs there is no single computational graph G that represents the computation of $\boldsymbol{f}$ for all possible values of $\boldsymbol{x}$. We explained in sec. 3.1 that OpenAD/F considers subgraphs constructed from the contents of a basicblock. In the example shown in fig. 3.6 we put the basicblock code shown in fig. 3.1 into a control flow context, see lines 06–09. The control flow graph (CFG) [7] resulting from the above code is depicted in fig. 3.7(a). The

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

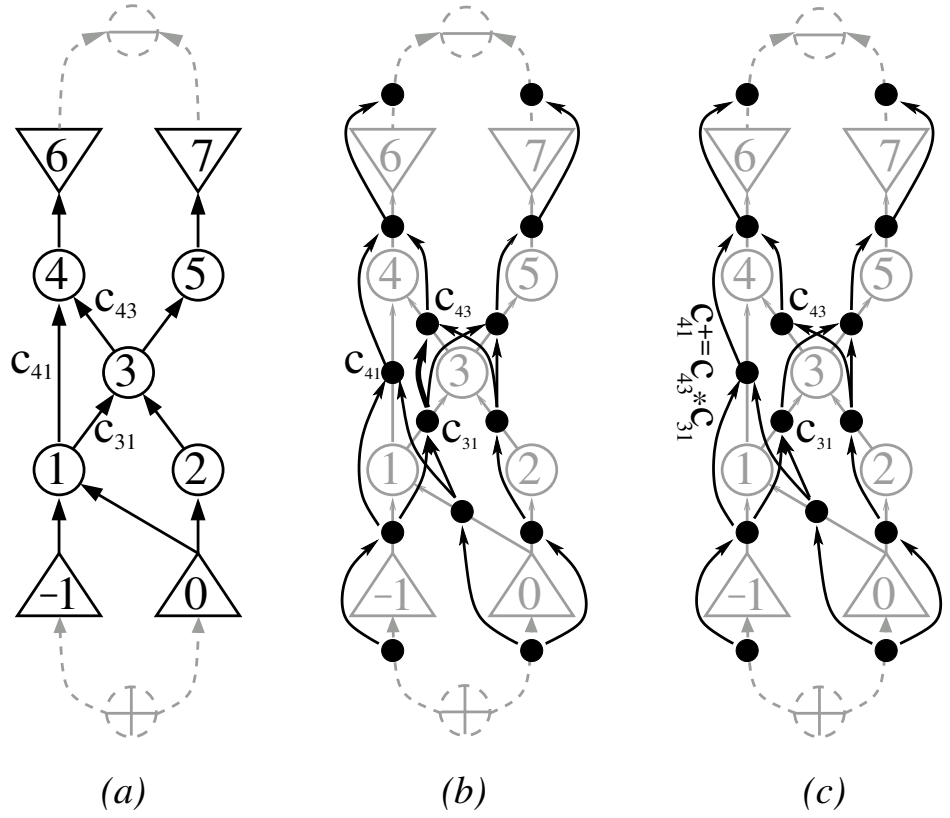


Figure 3.3: (a) G extended, (b) $\mathcal{G}$ overlaid, (c) face elimination

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

Figure 3.4: Pseudo code for (3.4) and the computation of the c_{ji}

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

Figure 3.5: Pseudo code for vertex eliminations for (3.4)

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

Figure 3.6: Toy example code with control flow

assignment statements are contained in the basicblocks B(2,4,6,9). For instance, the statements from fig. 3.1 now in lines 06–09 form the loop body, basicblock B(6). As B(6) is executed `k` times it may be worth putting additional effort into the optimization of the derivative code generated for B(6) by optimizing the elimination sequence as illustrated in sec. 3.2. For B(6) the corresponding computational graph G see fig. 3.2(a).

For a sequence of l basicblocks that are part of a path through the CFG for a particular value of $\boldsymbol{x}$ the equations (3.2) and (3.3) can be generalized as follows:

$$\dot{\boldsymbol{y}}_j = \boldsymbol{J}_j \dot{\boldsymbol{x}}_j \quad \text{for} \quad j = 1, \ldots, l \tag{3.6}$$

and

$$\bar{\boldsymbol{x}}_j = \boldsymbol{J}_j^T \bar{\boldsymbol{y}}_j \quad \text{for} \quad j = l, \ldots, 1 \quad , \tag{3.7}$$

where $\boldsymbol{x}_j = (x_i^j \in V : i = 1, \ldots, n_j)$ and $\boldsymbol{y}_j = (y_i^j \in V : i = 1, \ldots, m_j)$ are the inputs and outputs of the basicblocks respectively. In *forward mode* a sequence of products of the local Jacobians $\boldsymbol{J}_j$ with the directions $\dot{x}_j$ are propagated forward in the direction of the flow of control, for instance simultaneously to the computation of $\boldsymbol{f}$. In our example basicblock B(6) is the third basicblock ($j = 3$) and we have $\boldsymbol{x}_3 = \boldsymbol{y}_j =$ (`x(1),x(2)`) and consequently have the operations for the Jacobian vector product shown in fig. 3.8. Note that the code overwrites `x(1)` and `x(2)` and therefore we have to preserve the original derivatives in temporaries t_1 and t_2.

In *reverse mode* products of the transposed Jacobians $\boldsymbol{J}_j^T$ with adjoint vectors $\bar{\boldsymbol{y}}_j$ are propagated reverse to the direction of the flow of control. The $\boldsymbol{J}_j^T$ can be computed by augmenting the original code with linearization and Jacobian accumulation statements, see sec. 3.2. The preaccumulated $\boldsymbol{J}_j^T$ are stored during the forward execution which is commonly called the *tape*, see fig. 3.9(a) for an example. They are retrieved from the tape for computing (3.7) during the reverse execution, see fig. 3.9(b) for an example. It is always possible to organize the store and retrieve such that the tape can be implemented as a stack.

In order to find the corresponding path to the reversed control flow graph we also have to generate a trace which is done with an augmented CFG, for our toy example see fig. 3.7(b). This augmented CFG keeps track of which branch was taken and counts how often a loop was executed. This information is pushed on a stack and popped from that stack during the reverse sweep see also [43]. Because the control flow trace adheres to the stack model it often is also considered part of the tape. In the example in fig. 3.7(b) the extra basicblocks pBT and pBF push a boolean (T or F) onto the stack depending on the branch. In iLc we initialize a loop counter, increment the loop counter in +Lc, and push the final count in pLc.

fig. 3.7(c) shows the reversed CFG for our toy example. The parenthesized numbers in the node labels align the node transformation to fig. 3.7(a). The exit node becomes the entry, loop becomes endloop, branch becomes endbranch, and vice

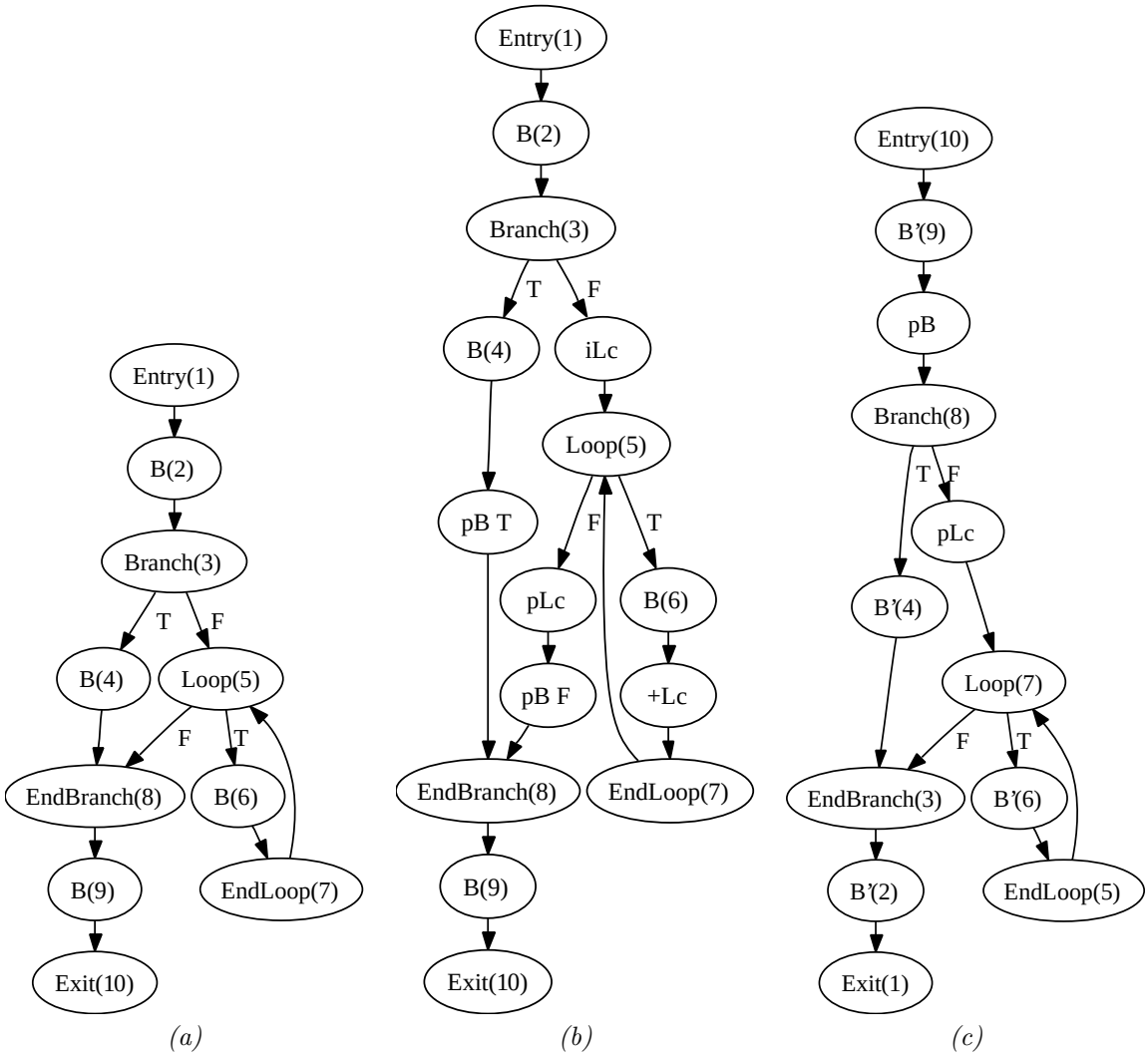


Figure 3.7: CFG of fig. 3.6 (a) original, (b) trace generating, (c) reversed

versa. Each basicblock B is replaced with its reversed version B'. Finally, to find the proper path through this reversed CFG we need to retrieve the information recorded in fig. 3.7(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 [43].

3.4 Call Graph Reversal

Generally, the computer program induces a *call graph* (CG) [7] 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 [22] for any subroutine in the call graph . These concepts are easier understood with the help of the dynamic call tree, see also [32]. where each vertex represents an actual invocation of a subroutine for a given execution of the program, see fig. 3.10 and table 3.1 for an explanation of the symbols. The order of calls is implied by following the edges in left to right order. Using split reversal for all subroutines in the program means that first the tape for the entire program is written. Then we follow with the reverse steps that read the tape, see fig. 3.11.

Using joint reversal for all subroutines in a program means that the tape, see sec. 3.3 for a each subroutine invocation is written immediately before the reverse execution for that invocation. In our example this implies that we have to generate a tape for $\mathtt{C}^2$ while the caller $\mathtt{B}^2$ is being reversed, i.e. this is not the proper context to simply reexecute $\mathtt{C}^2$. We can either reexecute the entire program up to the $\mathtt{C}^2$ call and then start taping, or (preferably) we store the arguments while running forward and restore them before starting the taping. The ensuing dynamic call tree for our example is shown in fig. 3.12.

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

Figure 3.8: Pseudo code for $\boldsymbol{J}_3\dot{\boldsymbol{x}}_3$ for the loop body in fig. 3.6

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

(a)

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

(a)

Figure 3.9: Pseudo code for writing the tape (a) and consuming the tape for $\boldsymbol{J}_3^T\bar{\boldsymbol{y}}_3$ (b) for the loop body in fig. 3.6

For many applications neither an all split nor all joint reversal is efficient. Often a mix of split and joint reversals statically applied to subtrees of the call tree is suitable.

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

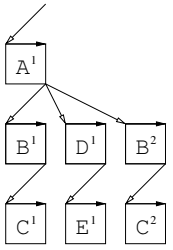


Figure 3.10: Dynamic call tree of a simple calling hierarchy

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

Table 3.1: Symbols for call tree reversal

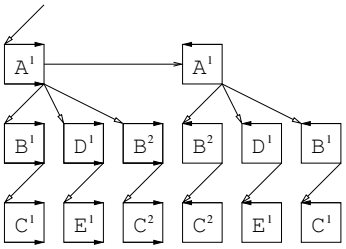


Figure 3.11: Dynamic call tree for split reversal

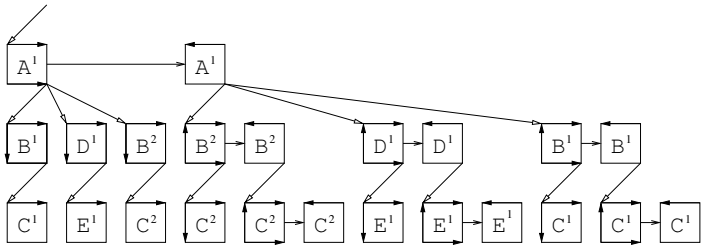


Figure 3.12: DCT of adjoint obtained by joint reversal mode

Chapter 4

Components of OpenAD/F

OpenAD/F is built on components that belong to a framework designed for code transformation of numerical programs. The components are tied together either via programmatic interfaces or by communication using the XAIF language. The transformation of the source code follows the tool chain shown in fig. 1.1. In sec. 4.1 we describe the language-independent components of OpenAD framework and sec. 4.2 provides details in the Fortran front-end. The regular setup procedure for OpenAD/F, see also sec. 2.1, will retrieve all components into an `OpenAD/` directory to which we refer from here on.

4.1 Language Independent Components (OpenAD)

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

4.1.1 Static Code Analyses (OpenAnalysis)

The OpenAnalysis toolkit, see [36], separates program analysis from language-specific or front-end specific intermediate representations. This separation enables a single implementation of domain-specific analyses such as activity analysis, to-be-recorded analysis, and linearity analysis in OpenAD/F. Standard analyses implemented within OpenAnalysis such as CFG construction, call graph construction, alias analysis, reaching definitions, ud- and du-chains, and side-effect analysis are also available via OpenADFortTk.

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

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

OpenAnalysis also performs activity analysis. For activity analysis the independent and dependent variables of interest are communicated to the front-end through the use of pragmas, see sec. 4.2.2. The results of the analysis are then encoded by the Fortran 90 front-end into XAIF. The analysis indicates which variables are active at any time, which memory references are active, and which statements are active.

The activity analysis itself is based on the formulation in [23]. The main difference is that the data-flow framework in OpenAnalysis does not yet take advantage of the structured data-flow equations. Activity analysis is implemented in a context-insensitive, flow-sensitive interprocedural fashion.

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

4.1.2 Representing the Numerical Core (XAIF)

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

- user type definitions and member access, see also sec. 4.2.1
- pointer arithmetic
- I/O operations
- memory management, see also sec. 7
- preprocessor directives

For a more general discussion regarding this compromise see also [45]. It is apparent that certain aspects of the adjoint code such as checkpointing, see sec. 3.4, and taping, see sec. 3.3, can involve memory allocation and various I/O schemes and therefore are not amenable to representation in the XAIF. At the same time it is also clear that the way one has to handle the memory and I/O for taping and checkpointing is primarily determined by the problem size at runtime and not primarily by the code we transform. Therefore in OpenAD such transformation results are handled by specific code expansion for subroutine specific templates and inlinable subroutine calls in the post processor, see sec. 4.2.4. This not only avoids the typically language specific I/O and memory management aspects, it also affords additional flexibility.

The format of choice in OpenAD is an XML-based [16] hierarchy of directed graphs, referred to as XAIF[25]. Using XML is motivated by the existence of XML parsers and the ability to specify the XAIF specific XML contents with a schema which the parser can use to validate a given XAIF representation. The current XAIF schema is documented at [47]. The basic building blocks are the same data structures commonly found in compilers from top down call graph with scopes and symbol tables, control flow graphs as call graph vertices, basic blocks as control flow graph vertices, statement lists contained in basic blocks, assignments as a statement with expression graphs, and variable references and intrinsic operations as expression graph vertices. The role of the respective elements in the XAIF schema is fairly self evident. Elements are associated by containment. In the graph structures edges refer to source and target vertices by vertex ids. Variable references contain references to symbols which in turn are associated to symbol table elements via a scope and a symbol id.

An snippet of the XAIF representation of a part of the example code from fig. 1.2(right) can be found in fig. 4.1. When

```
119     <xaif:Assignment statement_id="11" lineNumber="8" do_chain="1">
120       <xaif:AssignmentLHS du_ud="2" alias="2">
121         <xaif:SymbolReference vertex_id="1" scope_id="3" symbol_id="Y_2"/>
122       </xaif:AssignmentLHS>
123       <xaif:AssignmentRHS>
124         <xaif:Intrinsic vertex_id="1" name="tan_scal" type="***" annotation="{IntrinsicKey#0_TAN}"/>
125         <xaif:VariableReference vertex_id="2" du_ud="2" alias="3">
126           <xaif:SymbolReference vertex_id="1" scope_id="3" symbol_id="X_1"/>
127         </xaif:VariableReference>
128         <xaif:ExpressionEdge edge_id="1" source="2" target="1" position="1"/>
129       </xaif:AssignmentRHS>
130     </xaif:Assignment>
```

Figure 4.1: Snippet of XAIF representation for line 5 of fig. 1.2(right). Note that first the preprocessor is invoked to create `$OPENDADROOT/Examples/OneMinute/head.prepped.pre.f` which adds additional lines such that the line number is given as `lineNumber="8"`.

`make` is invoked in `$OPENDADROOT/Examples/OneMinute/`the full XAIF is written to a file called `$OPENDADROOT/Examples/OneMinute/head.prepped.pre.xaif`.

Further documentation for individual elements can be found directly in the schema annotations.

The XAIF also contains the results of the code analyses provided by OpenAnalysis, see sec. 4.1.1. Some are expressed either as additional attributes on certain XAIF elements, e.g. for activity information. Side-effect analysis provides lists of

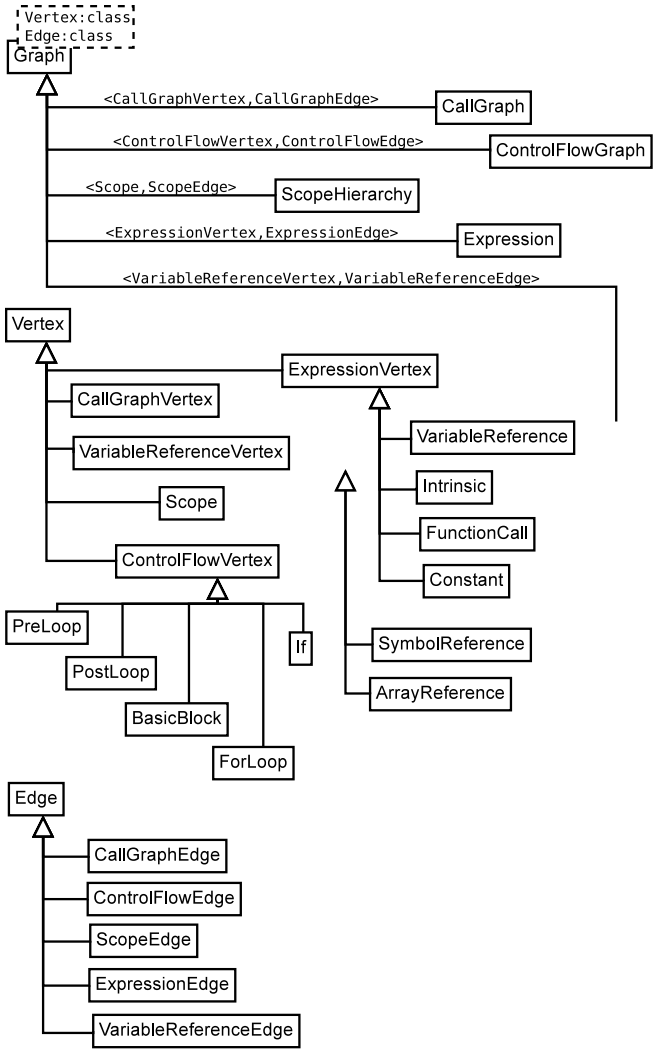


Figure 4.2: Simplified class inheritance in xaifBooster

variable references per subroutine, i.e. a call graph vertex element. DuUd chains are expressed as list of ids found in the assignment XAIF element. Alias information is expressed as set of virtual addresses. DuUd chains and alias information is collected in maps indexed by keys associated with the call graph. References to individual entries held in these maps are expressed via foreign key attributes in the elements.

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

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

4.1.3 Transforming the Numerical Core (xaifBooster)

The transformation engine that differentiates the XAIF representation of $\boldsymbol{f}$ is called xaifBooster. It is implemented in C++ based on a data structure that represents all information supplied in the XAIF input together with collection of algorithms that operate on this data structure, modify it and produce transformed XAIF output as the result. All sources for xaifBooster can be found under `OpenAD/xaifBooster/`. The principal setup of the source tree is shown in table 8.1. The xaifBooster data structure closely resembles the information one would find in a compiler's high level internal representation. the boost graph library [12] and the Standard C++ Library[18]. Figure4.2 and fig. 4.3 show simplified subsets of the classes occurring in the xaifBooster data structure in the inheritance as well as the composition hierarchy. A doxygen generated documentation of the entire data structure can be found on the OpenAD website [35]. The class hierarchy is organized top down with a single

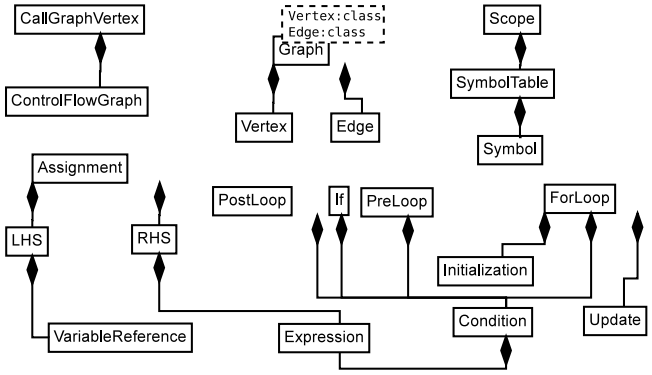


Figure 4.3: Simplified class composition in xaifBooster

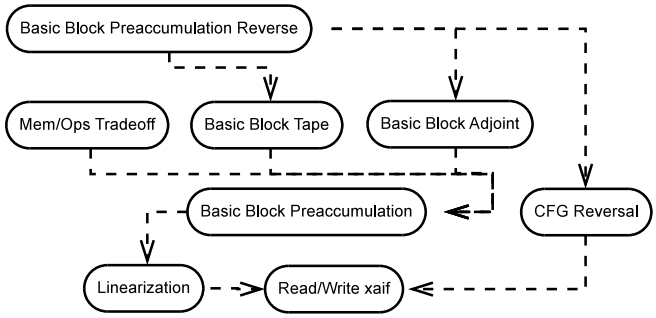


Figure 4.4: xaifBooster algorithms

`CallGraph` instance as the top element. The top down structure is also imposed on the ownership of dynamically allocated elements. Wherever possible, the class interfaces encapsulate dynamic allocation of members. Only in cases of containment of polymorphic elements is explicit dynamic allocation outside of the owning class' members appropriate. In these cases the container class interface naming and documentation indicates the assumption of ownership of the dynamically allocated elements being supplied to the container class. An example is the graph class `Expression` accepting vertex instances that can be `Constant`, `Intrinsic` , etc.

The transformation algorithms are modularized to enable reuse in different contexts. fig. 4.4 shows some implemented algorithms with dependencies. To avoid conflicts the transformation algorithms the data structure representing the input code is never directly modified. Instead, any data representing modifications or augmentations of the original representation element in a class `<name>` are held in algorithm specific instances of class `<name>Alg`. The association is done via mutual references accessible through `get<name>AlgBase()` and `getContaining<name>()` respectively. The instantiation of the algorithm specific classes follows the factory design pattern. The factory instances in turn are controlled by a transformation algorithm specific `AlgFactoryManager` classes. Further details can be found in [44], however, the code code for this mechanism is fairly self-explanatory.

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

Each algorithm has a driver `oadDriver.cpp` (compiled into a binary `oadDriver`) found in `algorithms/<the_algorithm_name>/test/` that encapsulates the algorithm in a stand-alone binary which provides the functionality described in the following sections. For details on the invocation and command line options refer to sec. 2.5.

4.1.3.1 Reading and Writing XAIF

Reading and Writing the XAIF is part of basic infrastructure found in the sources in `system/`. Parsing is done through the Xerces C++ XML parser [48] such that the XML element handler implementations, see `system/src/XAIFBaseParserHandlers.cpp`, build the xaifBooster data structure from the top down. As an additional consistency check all components that read XAIF data have the validation according to the schema enabled. Beyond the schema validation these components perform validity checks. Therefore, manual modifications of XAIF data , while possible, should be done judiciously.

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

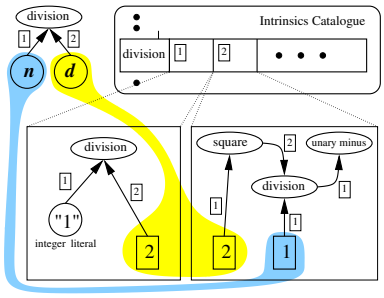


Figure 4.5: Partial expressions for the division operator

sec. 2.5.

Aside from the parsing of the actual input XAIF there is also the so called *catalog of inlinable intrinsics* supplied as an XML following a specialized schema in XAIF, see sec. 4.1.3.3 and sec. 4.2.3. There is also a driver at this level found in `system/test/t.cpp` used to verify reading and writing functionality. It can be used to establish that the tool chain preserves the semantics of the original program when no transformation is involved.

4.1.3.2 Type Change

4.1.3.3 Linearization

sec. 3 explained the computation of the local partial derivatives c_{ji} that can be thought of as edge labels in the computational graph G. Per canonicalization (see sec. 4.2.1) all elemental ϕ occur only in the right-hand side of an assignment. For each ϕ we look up the definition of the respective partials in the intrinsics catalog. [not sure how much detail is necessary] The partials are defined in terms of positional arguments, see fig. 4.5.

Because of this, the right-hand-side expression may have to be split up into subexpressions to assign intermediate values to ancillary variables that can be referenced in the partial computation, for an example see the code shown in fig. 3.4. In cases of the left-hand-side variable occurring on the right-hand-side (or being may-aliased to a right-hand-side variable, see sec. 4.1.1) we also require an extra assignment to delay the (potential) overwrite until after the partials depending on the original variable value have been computed. The result of the Linearization is a representation for code containing the potentially split assignments along with assignments for each non-zero edge label c_{ji}. These representations are contained in the `xaifBoosterLinearization::AssignmentAlg` instances associated with each assignment in the XAIF. The generated code after unparsing to Fortran is compilable but does by itself not compute useful derivative information at the level for the target function $\boldsymbol{f}$. The transformation driver is used to verify the results of the linearization transformation.

4.1.3.4 Basic Block Preaccumulation

This transformation generates a code representation that can be used to compute derivatives in forward mode. It builds upon the Linearization done in sec. 4.1.3.3. The first step constructs the computational graphs G for contiguous assignment sequences in a given basicblock. To ensure semantic correctness of the graph being constructed in the presence of aliasing it relies on alias analysis and define-use/use-define chains supplied by OpenAnalysis, see sec. 4.1.1. The algorithm itself is described in detail in [42]. Because the analysis results supplied by OpenAnalysis are always conservatively correct it may not be possible to flatten all assignments into a single computational graph. In such cases a sequence of graphs is created. Likewise, the occurrence of a subroutine call leads to a split in the graph construction. In the context of sec. 3 one may think of the sets of assignments forming each of these graphs as a separate basicblock. The driver for the algorithm allows to disable the graph construction across assignments and restrict it to single right-hand sides by adding the `-S` command line flag.

Based on the constructed G an elimination sequence has to be determined. To allow a choice for the computation of the elimination sequence the code uses the interface coded in `algorithms/CrossCountryInterface/` and by default calls the angel library [1, 8, 31]. angel determines an elimination sequence and returns it as fused multiply add expressions in terms of the edge references. There are several heuristics implemented within angel that control the selection of elimination steps and thereby the preaccumulation code that is generated. The algorithm code calls a default set of heuristics. However, all heuristics use the `CrossCountryInterface` and therefore different heuristics can be selected with minimal changes in algorithm code.

The second step in this transformation is the generation of preaccumulation code. First it turns the abstract expression graphs returned by angel into assignments and resolves the edge references into the labels c_{ji}. The resulting code resembles

what we show in fig. 3.5. Then it generates the code that eventually performs the saxpy operations shown in (3.6). Considering the input and output variables $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ of a basicblock the code generation also ensures proper propagation of $\dot{x}_i^j$ of variables $x_i^j \in \boldsymbol{x}_j \cap \boldsymbol{y}_j$ by saving the $\dot{x}_i^j$ in temporaries. The example in fig. 3.8 illustrates this case. The detection of the intersection elements relies on the alias analysis provided by OpenAnalysis. To reduce overhead the generated saxpy operations we generate saxpy calls following the interface specified in `algorithms/DerivativePropagator/` for the following four cases:

$$\text{(a):}\ \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x},\ \text{(b):}\ \dot{y} = \frac{\partial y}{\partial x} \cdot \dot{x} + \dot{y},\ \text{(c):}\ \dot{y} = \dot{x},\ \text{(d):}\ \dot{y} = 0 \quad . \tag{4.1}$$

The generated code is executable and represents an overall forward mode according to (3.6) with basicblocklocal preaccumulation in cross-country fashion.

4.1.3.5 Memory/Operations Tradeoff

This algorithm can be seen as an alternative to the angel library. Like angel it uses the `CrossCountryInterface`. In its implementation it replaces the call to angel with one to its own internal routines that determine an elimination sequence according to a selectable set of heuristics. In difference to the angel heuristics they aim for a tradeoff between the number of operations required to complete an elimination sequence on the one hand and the temporal locality of the c_{ji} in memory on the other hand. The rationale for these heuristics is the observation that in many modern computer architectures the performance is memory bound, i.e. a few additional operations can easily be absorbed if we keep all the necessary data in cache. All heuristics take as an input a set $\Theta \neq \emptyset$ of target elements, that is a set of vertices or edges from G, or faces from $\mathcal{G}$. The heuristic selects a nonempty subset $\Theta' \subseteq \Theta$ from this set. In order to determine a single elimination target a sequence of heuristics may be applied that successively shrink the target set concluding with a tie breaker such as selecting the next target that would be eliminated in forward or reverse mode. table 4.1 describes the selection criterion of a heuristics with respect to an elimination technique. If the selection criterion is not met by any target in Θ, then $\Theta' = \Theta$. The driver allows

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

Table 4.1: Heuristics selection criteria

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

The generated code is executable and represents an overall forward mode according to (3.6) with basicblocklocal preaccumulation in cross-country fashion.

4.1.3.6 Using the ANGEL Library

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

4.1.3.7 CFG Reversal

sec. 3.3 explains the principal approach to the reversal of the CFG. The CFG reversal as implemented in this transformation is by itself not useful as unparsed code other than for checking the correctness without interference from other transformations. It is a major building block for the adjoint code generator described in sec. 4.1.3.9. The loop counters and branch identifiers

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

Table 4.2: saxpy operations from (4.1) and their corresponding adjoints

are stored the same stack data structure that is used for the *tape* (introduced in sec. 3.3 and also used in sec. 4.1.3.8. The reversal of loops and branches as detailed in [43] assumes CFGs to be well-structured, that is, essentially to be free of arbitrary jump instructions such as `GOTO` or `CONTINUE`. It is of course possible to reverse such graphs, for instance by enumerating all basicblocks, recording the execution sequence and invoking them according to their recorded identifier in large `SWITCH` statement in reverse order. Such a reversal is obviously less efficient than a code that, by employing proper control flow constructs, aids compiler optimization. For the same reason well tuned codes implementing the target function $\boldsymbol{f}$ will avoid arbitrary jumps and therefore we have not seen sufficient demand to implement a CFG reversal for arbitrary jumps.

The reversal of loop constructs such as `do i=1,10` replaces the loop variable `i` with a generated variable name, say `t` and we loop up to the stored execution count which we will call `c` here. Then the reversed loop is `do t=1,c`. Quite often the loop body contains array dereferences such as `a(i)` but `i` is no longer available in the reversed loop. We call this kind of loop reversal *anonymous*. To access the proper memory location `i` will have to be stored along with the loop counters and branch identifiers in the tape stack. To avoid this overhead the loop reversal may be declared *explicit* by prepending `!$openad xxx simple loop` to the loop in question. With this directive the original loop variable will be preserved, the reversed loop in our example constructed as `do i=10,1,-1` and no index values for the array references in the loop body are stored. In general the decision when an array index needs to be stored is better answered with a code analysis similar to TBR analysis [23]. Currently we do not have such analysis available and instead as a compromise define the *simple* loop which can reversed explicitly as follows.

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

While these conditions can be relaxed in theory, in practice the effort to implement the transformation will rise sharply. Therefore they represent a workable compromise for the current implementation. Because often multidimensional arrays are accessed with nested loops the loop directive when specified for the outermost loop will assume the validity of the above conditions for everything within the loop body including nested loop and branch constructs. More details on this aspect can be found in [43].

4.1.3.8 Writing and Consuming the Tape

sec. 3 explains the need to store the $\frac{\partial \phi_j}{\partial v_i}$ on the tape. The writing transformation[1] stores the nonzero elements of local Jacobians $\boldsymbol{J}_j$. It is implemented as an extension of the preaccumulation in sec. 4.1.3.4 but instead of using the Jacobian elements in the forward saxpy operations as in (3.6) we store them on a stack as shown for the example code in fig. 3.8(a). The tape consuming transformation algorithm[2] reinterprets the saxpy operations generated in sec. 4.1.3.4 according to table 4.2. The tape writing and consumption implemented in these transformations are by themselves not useful as unparsed code other than for checking the correctness without interference from other transformations. They are, however, major building blocks for the adjoint code generator described in sec. 4.1.3.9.

4.1.3.9 Basic Block Preaccumulation Reverse

This transformation[3] represents the combination of the various transformation into a coherent representation that, unparsed into code and post-processed, compiles as an adjoint model. For the postprocessing steps refer to sec. 4.2.4. Additional

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

functionality is the generation of code that is able write and read checkpoints at a subroutine level, see also sec. 3.4. This part of the transformation relies heavily on the results of side-effect analysis, see sec. 4.1.1 and the inlinable subroutine call mechanism of the postprocessor, see sec. 4.2.4.2, to accomplish the checkpointing. The driver offers command line options

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

4.2 Language Dependent Components (OpenADFortTk)

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

1. The *canonicalizer* converts programming constructs into a canonical form described in sec. 4.2.1.
2. The compiler front-end mfef90 parses Fortran and generates an intermediate representation (IR) in the whirl format, see sec. 4.2.2
3. whirl2xaif is a bridge component that
   - drives the various program analyses (see sec. 4.1.1),
   - translates the numerical core of the program and the results of the program analyses from whirl to XAIF.

   see also sec. 4.2.3
4. xaif2whirl is bridge component that translates the differentiated numerical core represented in XAIF into the whirl format. see sec. 4.2.3.
5. whirl2f is the "unparser" that converts whirl to Fortran, see sec. 4.2
6. The *postprocessor* is the final part of the transformation that performs template expansion as well as inlining substitutions, see sec. 4.2.4

4.2.1 Canonicalization with `preProcess.py` (incompl.)

In sec. 4.1.2 we explain how the restriction to the numerical core contributes to the language independence of the transformation engine. Still, even for a single programming language, the numerical core often exhibits a large variability in expressing semantically identical constructs. To streamline the transformation engine we reduce this variability by *canonicalizing* the numerical core. This is done using the following Python script.

```
$OPENADFORTTK_BASE/tools/SourceProcessing/preProcess.py
```

Invoking the script with the `-h` option displays the following command-line options.

```
Usage: preProcess.py [options] <input_file> [additional input files]

Options:
  -h, --help            show this help message and exit
  --inputFormat={ fixed | free }
                        input file format
  --outputFormat={ fixed | free }
                        output file format
  --infoUnitFile=INFOUNITFILE
                        <infoUnitFile> contains modules which have type information needed for parsing, but should not be processed
                        themselves
  --inputLineLength=INT
                        sets the max line length of the input file. The default line lengths based on the format are fixed:72 free:132
  --outputLineLength=INT
                        sets the max line length of the output file. The default line lengths based on the format are fixed:72 free:132
  -o <output_file>, --output=<output_file>
                        redirect output to  file OUTPUT (default output is stdout); If the "--outputFormat" option is not used, the output
                        format is taken from the extension of this filename
  -v, --verbose         turns on verbose debugging output
  --check               turns on sanity checking which can slow down execution
  --noWarnings          suppress warning messages (defaults to False)
  -n, --noCleanup       do not remove the output file if an error was encountered (defaults to False)
  -m MODE, --mode=MODE  set default options for transformation mode with MODE being one of: r = reverse; f = forward;   reverse mode  implies
                        -H but not -S; specific settings override the mode defaults.
  --pathPrefix=PATHPREFIX
                        for use with --separateOutput: prepend this prefix to the directory name of the corresponding input file (defaults to
                        an empty string)
  --pathSuffix=PATHSUFFIX
                        for use with --separateOutput: append this suffix to the directory name of the corresponding input file (defaults to
                        "OAD")
  --filenameSuffix=FILENAMESUFFIX
                        for use with --separateOutput: append this suffix to the name of the corresponding input file (defaults to an empty
                        string)
  --recursionLimit=INT  recursion limit for the python interpreter (default: 1500; setting it too high may permit a SEGV in the interpreter)
  --timing              simple timing of the execution
  --separateOutput      split output into files corresponding to input files (defaults to False; conflicts with --output; if --outputFormat
                        is specified the extension will be adjusted to .f for fixed format and .f90 for free formar)
  -I PATH               directory to be added to the search path for Fortran INCLUDE directives; (default is the current directory)
  --warn=WARN           issue warning messages only for the specified type which is one of ( implicit | hoisting | ifStmtToIfConstr |
                        controlFlow | nesting | noDefinition | activeIO ); conflicts with --noWarnings
  --keepGoing           try to continue despite error messages; this is intended only to find trouble spots for the canonicalization, if
                        problems occur the output may contain invalid code (defaults to False)
  --removeFunction      remove original function definition when it is transformed to a subroutine definitions
  --r8                  set default size of REAL to 8 bytes
  -H, --hoistNonStringConstants
                        enable the hoisting of non-string constant arguments to subroutine calls (defaults to False)
  -S, --hoistStringConstants
                        enable the hoisting of string constant arguments to subroutine calls (defaults to False)
  --nonStandard=NONSTANDARD
                        allow non-standard intrinsics: ( etime | free | getuid | getpid | hostnam | hostnm | loc | malloc | sleep | time ) ;
                        can be specified multiple times  (defaults to None).
  --subroutinizeIntegerFunctions
                        should integer function calls be subroutinized (defaults to False)
  --progress            issue progress message to stderr per opened input file (default is False)
  --overloading         prepare to handle the type conversion in a manner suitable for overloading, e.g. with Rapsodia generated libraries;
                        this skips the canonicalization that turns non-inlinable intrinsics into subroutines; not usable for source
                        transformation with OpenAD; defaults to false
```

Because it is done automatically, the canonicalization does *not* restrict the expressiveness of the input programs supplied by the user. Rather it is a means to reduce the development effort of the transformation engine. In the following we describe the canonical form.

Canonicalization 1 *All function calls are canonicalized into subroutine calls, see fig. 4.6 line 9 on the left vs. lines 22,23 on the right. For the transformations, in particular the basicblock level preaccumulation we want to ensure that an assignment effects a single variable on the left-hand side. Therefore the right-hand-side expression ought to be side-effect free. While often not enforced by compilers, this is a syntactic requirement for Fortran programs. Rather than determining which user-defined functions have side effects, we pragmatically hoist* all *user-defined functions. Consequently the right-hand-side expression of an assignment consists only of elemental operations ϕ typically defined in a programming language as built-in operators and intrinsics. The canonicalization of the respective definitions for the function in fig. 4.6 is shown in fig. 4.7.*

Canonicalization 2 *A particular canonicalization of calls without canonicalization of definitions is applied to certain intrinsics, e.g. the* `max` *and* `min` *intrinsics because in Fortran they do not have closed form expressions for the partials. OpenAD/F provides a run time library containing definitions for the respective subroutines called instead. An example is shown in fig. 4.8.*

Canonicalization 3 *Non-variable actual parameters are hoisted to temporaries. Any value passed to a routine could conceivably need augmentation. Furthermore, only variables can be augmented. Consequently, OpenAD hoists all non-variable actual parameters into temporaries. An example is shown in fig. 4.9.*

```
6  program p
7    real x,y,a,b
8    x=1.0; a=2.0; b=3.0
9    y = x * foo(a,b)
10 end program
```

```
15 program p
16   use OAD_intrinsics
17   real x,y,a,b
18   real :: oad_ctmp0
19   x=1.0
20   a=2.0
21   b=3.0
22   call oad_s_foo(a,b,oad_ctmp0)
23   y = x*oad_ctmp0
24 end program
```

Figure 4.6: Canonicalizing a function call (left, see file `$OPENADROOT/Examples/SRCanonical/func.f90`), to a subroutine call (right, see file `$OPENADROOT/Examples/SRCanonical/func.pre.f90` after running `make`)

```
1  real function foo(a,b)
2    real a,b
3    foo = a*b
4  end
```

```
8  subroutine oad_s_foo(a,b,foo)
9    use OAD_intrinsics
10   real a,b
11   real,intent(out) :: foo
12   foo = a*b
13 end subroutine oad_s_foo
```

Figure 4.7: Canonicalizing a function definition (left, see file `$OPENADROOT/Examples/SRCanonical/func.f90`), to a subroutine definition (right, see file `$OPENADROOT/Examples/SRCanonical/func.pre.f90` after running `make`) for the accompanying call change shown in fig. 4.6

```
1  program p
2    real x,y,z
3    x=1.0; y=2.0; z=3.0
4    z=max(x,y,z)
5  end program
```

```
17 program p
18   use OAD_intrinsics
19   real x,y,z
20   real :: oad_ctmp0
21   real :: oad_ctmp1
22   real :: oad_ctmp2
23   x=1.0
24   y=2.0
25   z=3.0
26   call oad_s_max(y,z,oad_ctmp2)
27   call oad_s_max(x,oad_ctmp2,oad_ctmp1)
28   z = oad_ctmp1
29 end program
```

```
1  module OAD_intrinsics
2    interface oad_s_max
3      module procedure oad_s_max_r
4    end interface
5    contains
6      subroutine oad_s_max_r(a0,a1,r)
7        real,intent(in) :: a0
8        real,intent(in) :: a1
9        real,intent(out) :: r
10       if (a0>a1) then
11         r = a0
12       else
13         r = a1
14       end if
15     end subroutine
16 end module
```

Figure 4.8: Canonicalizing a call to `max` (top left, see file `$OPENADROOT/Examples/MaxCanonical/func.f90`), to two subroutine calls (bottom left) and the accompanying definition (right, see file `$OPENADROOT/Examples/MaxCanonical/func.pre.f90` after running `make`)

```
6  program p
7    real x
8    x=1.0;
9    call foo(x*3.0)
10 end program
```

```
9  program p
10   use OAD_intrinsics
11   real x
12   real :: oad_ctmp0
13   x=1.0
14   oad_ctmp0 = x*3.0
15   call foo(oad_ctmp0)
16 end program
```

Figure 4.9: Canonicalizing a subroutine argument (left, see file `$OPENADROOT/Examples/ArgExprCanonical/func.f90`), into a temporary variable (right, see file `$OPENADROOT/Examples/ArgExprCanonical/func.pre.f90` after running `make`)

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

4.2.1.1 Error Conditions

For simplicity the Fortran parser use here does not validate the input. All input is expected to be syntactically correct, otherwise an exception may be raised and the `preProcess.py` or `postProcess.py` script may abort with an error message.

`UserError :` a simple user error (such as specifying a non-existant input file); the details should be explained by the accompanying message.

`FunToSubError :` an error specific to the function to subroutine conversion; if the message does not spell out a specific problem with the user input it may indicate a problem in the implementation and should be reported to the bug tracking system via the OpenAD/F website.

`CanonError :` an error specific to the canonicalization phase; if the message does not spell out a specific problem with the user input it may indicate a problem in the implementation and should be reported to the bug tracking system via the OpenAD/F website.

other errors: error messages of the following categories:

`ScanError`

`ParseError`

`AssemblerException`

`ListAssemblerException`

`SymtabError`

`InferenceError` Note that an error message relating the Open64 specific `kind` parameters `w2f__4`, `w2f__8` etc. typically indicates an unresolved reference in unparsed code and can be resolved by supplying the `w2f__types.f90` file that contains the defining statements using the `--infoUnitFile` flag.

`LogicError`

thrown either because the input has incorrect syntax or there is a bug in the parser. If you are convinced the input is syntactically correct please submit the input along with the error message to the bug tracking system via the OpenAD/F website.

4.2.2 Compiler Front-End Components (from Open64)

The choice of Open64 for some of the programming-language-dependent components ensures some initial robustness of the tool that is afforded by an industrial-strength compiler. The Center for High Performance Software Research (HiPerSoft) at Rice University develops Open64 [34] as a multi-platform version of the SGI Pro64/Open64 compiler suite, originally based on SGI's commercial MIPSPro compiler.

OpenAD/F uses the parser, an internal representation and the unparser of the Open64 project. The classical compiler parser mfef90 produces a representation of the Fortran input in a format known as very high level or source level whirl .

The whirl representation can unparsed into Fortran using the tool whirl2f. The source level whirl representation resembles a typical abstract syntax tree with the addition of machine type deductions. The original design of whirl in particular the descent to lower levels closer to machine code enables good optimization for high performance computing in Fortran, C, and C++. HiPerSoft's main contribution to the Open64 community has been the source level extension to whirl which is geared towards supporting source-to-source transformations and it has invested significant effort in the whirl2f unparser.

For the purpose of AD, user-supplied hints and required input is typically not directly representable in programming languages such as Fortran and therefore represented by pragmas. For example, an AD tool must know which variables in the code for $\boldsymbol{f}$ are independent and which are dependent, see also sec. 2.3. 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 [37].

4.2.2.1 Parser

The parser binary can be found in the following file.

```
$OPEN64ROOT/crayf90/sgi/mfef90
```

The full set of options can be retrieved with the command line flag `-h`. The options of interest for the OpenAD/F context are the following.

```
The following options are available in the Open64 frontend:
-f: souce format (-ffree or -ffixed)
-z: clean up whirl (required for use with OpenAD/OpenAnalysis)
-F: fortran macro expansion
-N: fixed line size
```

As indicated above, the flag **`-z` has to be specified**. The majority of the other options are not useful in the OpenAD context. There are however some debugging and tracing options that can be specified with the `-u` flag.

4.2.2.2 Unparser

The unparser binary can be found in the following file.

```
$OPEN64ROOT/whirl2f/whirl2f
```

and the full set of options can be retrieved by invoking it without any arguments. The options of interest in shown in fig. 4.10. As indicated, the flag `-openad` **has to be specified**. The name for the abstract active type that can be supplied with the `-openadType` flag **must be identical** to the name passed to the post processor with the `-a` flag, see sec. 4.2.4.

4.2.3 Translating between whirl and XAIF

Two features of XAIF shape the contours of whirl2xaif (translate whirl to XAIF) and xaif2whirl (translate XAIF to whirl). Both are located in the following directory.

```
$OPENADFORTTKROOT/bin/
```

Invoking the respective component with the `-h` option displays command-line options. The options of the whirl2xaif and xaif2whirl component are shown in fig. 4.11 and fig. 4.12, respectively.

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

Furthermore, XAIF provides a way to represent the results of common compiler analyses. To provide these to the transformation engine whirl2xaif acts as a driver for the analyses provided by the OpenAnalysis package, see sec. 4.1.1. In

```
USAGE: in EBNF notation, where '|' indicates choice and '['
indicates an optional item:

        /sandbox/utke/CronTest/OpenAD/Open64/osprey1.0/targ_ia64_ia64_linux/whirl2f/whirl2f [-FLIST:<opts>] [-TARG:<t>] [-TENV:<e>]
        [-openad] [-openadType <name>]  <inp_files>

        <inp_files> ::= [-fB,<Whirl_File_Name>] <File_Name>
        <opts> ::= <single_opt>[:<opts>]

We recommend always using the common option -TARG:abi=[32|64].

The <File_Name> is a mandatory command-line argument, which may
denote either a (Fortran) source filename or a WHIRL file.
In the abscense of a -fB option, the <Whirl_File_Name> will be
derived from the <File_Name>
-openad required within the OpenAD tool, see
        http://www.mcs.anl.gov/OpenAD .
-openadType <name> unparses a specially named active type <name>; default is 'oadactive'
        <name> cannot be longer than 26 characters; requires the -openad flag

Each -FLIST:<single_opt> is described below:

-FLIST:show
        Indicate the input/output file-names to stderr.
-FLIST:linelength=<n>
        Specifies an upper limit on the number of characters we allow
        on each line in the output file.  For tab-formatting, a tab
        is counted as one character
-FLIST:old_f77
        Prevents emission of calls to intrinsic functions that are not
        in compilers earlier than version v7.00.  The generated source
        will include <whirl2f.h>
-FLIST:ansi_format
        Format the output according to Fortran 77 rules, with at most
        72 columns per line.  Without this option, tab formatting is
        employed without any limit on the line-length.
-FLIST:src_file=<Src_File_Name>
        The name of the original source program.  When not given,
        the <Src_File_Name> is derived from the <Whirl_File_Name>.
-FLIST:ftn_file=<Ftn_OutFile_Name>
        The file into which program units will be emitted.  When not
        given, <Ftn_OutFile_Name> is derived from <Src_File_Name>.
```

Figure 4.10: Subset of whirl2f options that are relevant for OpenAD/F.

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.

The companion tool xaif2whirl backtranslates XAIF into whirl. As indicated above it has to take care of restoring filtered out statements and type information. Because the differentiated XAIF relies on postprocessing, see sec. 4.2.4, its other major challenge is the creation of whirl that contains the postprocessor directives related to three tasks to be accomplished by the postprocessor.

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

4.2.4 Postprocessing with `postProcess.py` (incompl.)

The postprocessor performs the three tasks outlined at the end of sec. 4.2.3. This is done using the following Python script.

`$OPENADFORTTK_BASE/tools/SourceProcessing/postProcess.py`

Invoking the script with the `-h` option displays the command-line options shown in fig. 4.13.

4.2.4.1 Use of the Active Type

The simplest postprocessing task is the concretization of the active variable declarations and uses. The main rationale for postponing the concretization of the active type is flexibility with respect to the actual active type implementation. This is done via the `-t` flag. Using the text replacement in the postprocessor it is much easier to adapt to a changing active type implementation than to find find the proper whirl representation and modify xaif2whirl to create it. However, it should be noted, that the ease of adaptation is clearly correlated to the simplicity and in particular the locality of the transformation.

```
Usage: /sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/whirl2xaif [options] <whirl-file>

Given a WHIRL file, translates the 'numerical core' into XAIF.  By default,
output is sent to stdout.

Options:
  -h, --help          print this help and exit
  -n, --noFilter      do not filter ud/du chains by current basic block
  -N, --noTimeStamp   do not print a time stamp into the output
  -o, --output <file> send output to <file> instead of stdout
      --prefix <pfx>  Set the temporary variable prefix to <pfx>. Default
                      is 'OpenAD_'
  -s, --simpleLoop    force simple loop property on all loop constructs
  -v, --variedOnly    do not require active data to also be 'useful'
      --uniformCBact  if any variable in a given common block is active
                      then activate all of them
      --allActive     all floating point variables are made active
      --debug [lvl]   debug mode at level `lvl'
```

Figure 4.11: Options of whirl2xaif.

```
Usage: /sandbox/utke/CronTest/OpenAD/OpenADFortTk/OpenADFortTk-x86_64-Linux/bin/xaif2whirl [options] <whirl-file> <xaif-file>

Given a WHIRL file and a *corresponding* XAIF file, generates new WHIRL.
By default, the output is sent to the filename formed by replacing the
extension of <xaif-file> with 'x2w.B'.

Algorithms:
  --bb-patching     TEMPORARY: use basic-block patch algorithm

Options:
  -o, --output <file> send output to <file> instead of default file
      --i4            make integers 4 byte where not specified
                      (default 8 bytes)
      --u4            make unsigned integers 4 byte where not specified
                      (default 8 bytes)
      --r4            make reals 4 byte where not specified (default 8 bytes)
  -t, --type <name>   abstract active type name (default oadactive), no longer
                      than 26 characters
  -V, --version       print version information and exit
  -v, --validate      validate agains schema
  -h, --help          print this help and exit
  -n, --noCleanUp     only for development: do not perform whirl cleanup
                      needed for OpenAD
      --debug [lvl]   only for development: debug mode at level `lvl'
```

Figure 4.12: Options of xaif2whirl.

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 by default as `TYPE (oadactive)v`.[4] 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/scalar/OAD_active.f90` are by default `type(active)v` for the declaration and simply `v\%v` for the value `v\%d` for the derivative component respectively and each subroutine will also receive an additional `USE` statement which makes the type definition in `OAD_active` known. The abstract active type name can be changed with the `--abstractType` option and the concrete active type name can be changed with the `--concreteType` option, respectively.

4.2.4.2 Inlinable Subroutine Calls

The second task, the expansion of inlinable subroutine calls, is more complex because any call expansion has now the scope of a subroutine body. The calls unparsed from whirl to Fortran are regular subroutine call statements. They are however preceded by an inline pragma `!$openad inline <name(parameters)>` that directs the postprocessor to expand the following call according to a definition found in an input file[5], see also `runTimeSupport/simple/ad_inline.f`. Pushing a preaccumulated sub-Jacobian value as in fig. 3.9(a) might appear in the code for example from sec. 1.3.2 as[6]

[4] See also the `-t` flag in sec. 4.2.3 to change the name of the abstract active type.
[5] specified with command line option `-i` which defaults to `ad_inline.f`
[6] see file `$OPENADROOT/Examples/OneMinuteReverse/head.prepped.pre.xb.x2w.w2f.f` after running `make`)

```
43  C $OpenAD$ INLINE push_s0(subst)
44        CALL push_s0(OpenAD_lin_0)
```

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

```
48
49
50
51
52  C taping
        ------------------------------------------
53
54
55        subroutine push_s0(x)
56  C $OpenAD$ INLINE DECLS
```

The postprocessor ignores the `DECLS` section and expands this to[7]

```
102  Y%v = OpenAD_aux_0
103  double_tape(double_tape_pointer) = OpenAD_lin_0
```

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

4.2.4.3 Subroutine Templates

The third task, the subroutine template expansion is somewhat related the inlining. In our example above, the tape storage referred to in the `Push` need to be defined and in the design the subroutine template is the intended place for these definitions, in our example achieved through including the `use` statement in the template code, see fig. 4.15. The main purpose of the subroutine template expansion however is to orchestrate the call graph reversal. The reversal schemes introduced in sec. 3.4 can be realized by carrying state through the call tree.

The basic building blocks from the transformations in sec. 4.1.3 are variants $\mathtt{S}_i$ of the body of an original subroutine body $\mathtt{S}_0$, each accomplishing one of the tasks shown as one of the squares with arrows in table 3.1. For instance, the taping variant is created by the transformation in sec. 4.1.3.8 or the checkpointing by the transformation in sec. 4.1.3.9. To integrate the S_i into a particular reversal scheme, we need to be able to make all subroutine calls in the same fashion as in the original code and, at the same time, control which task each subroutine call accomplishes. We replace the original subroutine body with a branch structure in which each branch contains one S_i. The execution of each branch is determined by a global control structure whose members represent the state of execution in the reversal scheme. The branches contain code for pre- and post-state transitions enclosing the respective S_i. This ensures that the transformations producing the S_i do not depend on any particular reversal scheme. The postprocessor inserts the $\mathtt{S}_i$ into a subroutine template, schematically shown in fig. 4.14(a). The template is written in Fortran. Each subroutine in the postprocessor Fortran input is transformed according to either a default subroutine template found in a `ad_template.f` file or in a file specified in a `!$openad XXX Template <file name>` pragma to be located in the subroutine body. The input Fortran also contains `!$openad begin replacement <i>` paired with pragmas `!$openad end replacement`. Each such pair delimits a code variant $\mathtt{S}_i$ and the postprocessor matched the respective identifier i (an integer) with the identifier given in the template `PLACEHOLDER_PRAGMA`.

Split reversal is the simplest static call graph reversal. We first execute the entire computation with the augmented forward code (S_2) and then follow with the adjoint (S_3). From the task pattern shown in fig. 3.11 it is apparent that, aside from the top-level routine, there is no change to the state structure within the call tree. Therefore, there is no need for state changes within the template. Since no checkpointing is needed either, we have only two tasks: producing the tape and the adjoint run. fig. 4.14(b) shows a simple split-mode template, see also `runTimeSupport/simple/ad_template.split.f`. The state is contained in `rev_mode`, a static Fortran90 variable, see `runTimeSupport/simple/OpenAD_rev.f90` of type `modeType` also defined in this module. In order to perform a split-mode reversal for the entire computation, a driver routine calls the top-level subroutine first in taping mode and then in adjoint mode.

fig. 3.12 illustrates the task pattern for a joint reversal scheme that requires state changes in the template and requires more code alternatives. fig. 4.15 shows a simplified joint mode template, see also `runTimeSupport/simple/ad_template.joint.f`. The state transitions in the template directly relate to the pattern shown in fig. 3.12. Each pre-state change applies to the callees of the current subroutine. Since the argument store (S_4) and restore (S_6) do not contain any subroutine calls they do not need state changes. Looking at fig. 3.12, one realizes that the callees of any subroutine executed in plain forward

[7] see file `$OPENADROOT/Examples/OneMinuteReverse/head.prepped.pre.xb.x2w.w2f.post.f90` after running `make`)

mode (S_1) never store the arguments (only callees of subroutines in taping mode do). This explains lines 18, 25, and 30. Furthermore, all callees of a routine currently in taping mode are not to be taped but instead run in plain forward mode, as reflected in lines 27 and 28. Joint mode in particular means that a subroutine called in taping mode (S_2) has its adjoint (S_3) executed immediately after S_2. This is facilitated by line 33, which makes the condition in line 35 true, and we execute S_3 without leaving the subroutine. Any subroutine executed in adjoint mode has its direct callees called in taping mode, which in turn triggers their respective adjoint run. This is done in lines 37–39. Finally, we have to account for sequence of callees in a subroutine; that is, when we are done with this subroutine, the next subroutine (in reverse order) needs to be adjoined. This process is triggered by calling the subroutine in taping mode, as done in lines 41–43. The respective top-level routine is called by the driver with the state structure having both `tape` and `adjoint` set to `true`.

4.2.4.4 Error Conditions

In addition to what is mentioned in sec. 4.2.1.1 the post processing phase may abort with:

`PostProcessError :` an error specific to the post processing logic; if the message does not spell out a specific problem with the user input it may indicate a problem in the implementation and should be reported to the bug tracking system via the OpenAD/F website.

4.3 Ancillary Tools

We provide a number of helper scripts to manage the OpenAD/F installation.

4.3.1 The `openadUpdate` and `openadStatus` Scripts

This section is useful only for OpenAD/F installation made from source code repositories. The binary distribution does not include the scripts and source tar ball does not include the repository data. To get OpenAD/F components and update the revision from the source code repositories it is easiest to use the `openadUpdate` script which is written in Python [38] and can be found at

```
$OPENDADROOT/bin/openadUpdate
```

It provides the following command line flags.

```
Usage: openadUpdate [options]
        get or update OpenAD repositories

Options:
  -h, --help            show this help message and exit
  -e, --extras          include repositories for revolve and examples
                        referenced in the User Manual (requires Mercurial)
  -f, --force           do all actions, no confirmations (even when
                        repositories are deleted), implies -k
  -i, --interactive     requires to confirm each command
  -k, --keepGoing       keep going despite errors
  -t, --tests           include repositories for test cases (requires
                        Mercurial)
  --revisionFile=<file_name>
                        takes the output of 'opendadStatus -t' as  an input
                        file to update to specific component revisions
  -v, --verbose         extra output
  -d, --development     only for developers!: adjust updates for Mercurial
                        development repositories
```

Typically it will be invoked without any flags. To obtain the examples for this manual one will, however, need the Mercurial tool [27] and use the `-t` flag; see also sec. 8.3. The `openadUpdate` script uses the default locations for the component repositories given in `$OPENADROOT/openad_config.py`. A log named `openadUpdate.`*timestamp*`.log~` will contain output and potential error messages.

The `openadStatus` script is primarily for development purposes to obtain an overview of the status of the various repositories. It can also be used to see if an update from the source repositories is ready. That would be indicated by an `I` in the `Remote` column of `openadStatus` output such as the following.

```
local directory                                 |Kind|R/W|Loc.|Rem.|URL
------------------------------------------------------------------------
../OpenAD                                       | svn| W |    |    |https://svn.mcs.anl.gov/repos/OpenAD
Open64                                          | svn| W |    |    |https://svn.mcs.anl.gov/repos/Open64
OpenADFortTk                                    | svn| W |    |    |https://svn.mcs.anl.gov/repos/OpenADFortTk
OpenAnalysis                                    | svn| W |    |    |https://svn.mcs.anl.gov/repos/OpenAnalysis
xercesc                                         | svn| R |    |    |http://hpc.svn.rice.edu/r/xercesc
xaifBooster                                     | svn| W |    |    |https://svn.mcs.anl.gov/repos/xaifBooster
xaif                                            | svn| W |    |    |https://svn.mcs.anl.gov/repos/xaif
angel                                           | svn| W |    |    |https://svn.code.sf.net/p/angellib/code
boost/boost                                     | svn| R |    |    |http://svn.boost.org/svn/boost
RevolveF9X                                      |  hg| R |    |    |http://mercurial.mcs.anl.gov//ad/RevolveF9X
Examples                                        |  hg| R |    |    |http://mercurial.mcs.anl.gov//ad/OpenADExamples
Regression                                      |  hg| R |    |    |http://mercurial.mcs.anl.gov//ad/RegressionOpenAD
OpenADFortTk/Regression                         |  hg| R |    |    |http://mercurial.mcs.anl.gov//ad/RegressionOpenADFortTk
OpenADFortTk/tools/SourceProcessing/Regression|  hg| R |    |    |http://mercurial.mcs.anl.gov//ad/RegressionSourceProcessing
```

The script provides the following command line flags.

```
Usage: openadStatus [options]
        displays OpenAD repository status
        Kind:   hg, svn or cvs
        R/W:    R=read-only, W=writeable, L=symbolic link
        Local:  C=locally changed , U=locally pending updates (hg only)
        Remote: O=outgoing changes , I=incoming changes
        A '?' is shown when the network/server takes too long.

Options:
  -h, --help            show this help message and exit
  -l, --localTag        inlcude the local revision tag where applicable
                        (implies -t)
  -q, --quick           don't check for uncommited changes or pending pushes
  -s, --skipNetworkTest
                        skip the network test (ping www.rice.edu) that we do
                        to avoid waiting for repository commands when the
                        network is unavailable, but sometimes pings are being
                        blocked.
  -t, --tagsOnly        only list the repository and a version tag
  -v, --verbose         extra output
  -d, --development     only for developers!: consider SourceProcessing as a
                        separate repository
```

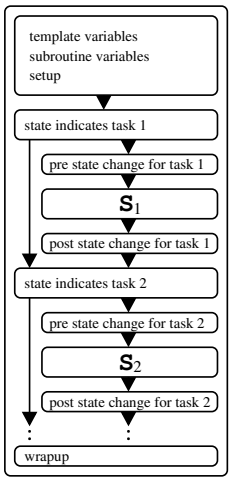
Unlike `openadUpdate`, the `openadStatus` script determines the repository type and URL not based on the defaults in `$OPENADROOT/openad_config.py` but instead uses the actual data found in the respective directories.

```
Usage: postProcess.py [options] <input_file> [additional input files]

Options:
  -h, --help            show this help message and exit
  --inputFormat={ fixed | free }
                        input file format
  --outputFormat={ fixed | free }
                        output file format
  --infoUnitFile=INFOUNITFILE
                        <infoUnitFile> contains modules which have type information needed for parsing, but should not be processed
                        themselves
  --inputLineLength=INT
                        sets the max line length of the input file. The default line lengths based on the format are fixed:72 free:132
  --outputLineLength=INT
                        sets the max line length of the output file. The default line lengths based on the format are fixed:72 free:132
  -o <output_file>, --output=<output_file>
                        redirect output to  file OUTPUT (default output is stdout); If the "--outputFormat" option is not used, the output
                        format is taken from the extension of this filename
  -v, --verbose         turns on verbose debugging output
  --check               turns on sanity checking which can slow down execution
  --noWarnings          suppress warning messages (defaults to False)
  -n, --noCleanup       do not remove the output file if an error was encountered (defaults to False)
  -m MODE, --mode=MODE  set default options for transformation mode with MODE being one of: r = reverse; f = forward;   reverse mode  implies
                        -H but not -S; specific settings override the mode defaults.
  --pathPrefix=PATHPREFIX
                        for use with --separateOutput: prepend this prefix to the directory name of the corresponding input file (defaults to
                        an empty string)
  --pathSuffix=PATHSUFFIX
                        for use with --separateOutput: append this suffix to the directory name of the corresponding input file (defaults to
                        "OAD")
  --filenameSuffix=FILENAMESUFFIX
                        for use with --separateOutput: append this suffix to the name of the corresponding input file (defaults to an empty
                        string)
  --recursionLimit=INT  recursion limit for the python interpreter (default: 1500; setting it too high may permit a SEGV in the interpreter)
  --timing              simple timing of the execution
  --separateOutput      split output into files corresponding to input files (defaults to False; conflicts with --output; if --outputFormat
                        is specified the extension will be adjusted to .f for fixed format and .f90 for free formar)
  -d, --deriv           appends %d to deriv types instead of removing __deriv__
  -i INLINE, --inline=INLINE
                        file with definitions for inlinable routines (defaults to none for forward mode and ad_inline.f for reverse mode)
  -p, --progress        progress message to stdout per processed unit
  -t TEMPLATE, --template=TEMPLATE
                        file with subroutine template for reverse mode post processing (defaults to ad_template.f) for subroutines that do
                        not have a template file specified via the template pragma; requires reverse mode ( -m r )
  --abstractType=ABSTRACTTYPE
                        change the abstract active type name to be replaced  (see also --concreteType ) to ABSTRACTTYPE; defaults to
                        'oadactive')
  --activeVariablesFile=ACTIVEVARIABLESFILE
                        write all active variable declarations into file ACTIVEVARIABLESFILEFILE.
  --concreteType=CONCRETETYPE
                        replace abstract active string (see also --abstractType ) with concrete active type CONCRETETYPE; defaults to
                        'active'
  --overloading         handle the type conversion in a manner suitable for overloading, e.g. with Rapsodia generated libraries, by not
                        making explicit and value references in active variables with the exception of I/O statements and assignments with a
                        passive righ-hand side; this suppresses the injection of "use oad_active", see also the --extraReference option; not
                        usable for source transformation with OpenAD; defaults to false
  --extraReference=EXTRAREFERENCE
                        if specified inject this string as an additional line after the standard "use w2f__types" statement
  --noInline            no inlining; overrides the defaults of the reverse mode settings; (defaults to False)
  --explicitInit        create subroutines for the explicit initialization of active variables in common blocks and modules
  --width=WIDTH         write one compile unit per output file with WIDTH digits prepended to the extension of <input_file>, e.g. for -n 2
                        and three compile units in an input file named 'a.f' we create 'a.00.f', a.01.f', 'a.02.f'; also creates a file named
                        'postProcess.make' for reference within a makefile; cannot be specified together with -o
  --whitespace          inserts whitespaces between tokens
```

Figure 4.13: Options of the post processor

(a)



(b)

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

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

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


end subroutine template
```

Figure 4.14: Subroutine template components (a), split-mode Fortran90 template (b)

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

Figure 4.15: Joint mode Fortran90 template with argument checkpointing

Chapter 5

Application

This applications section intends to augment the explanations given so far. We use two small examples to further illustrate the explanations given in sec. 1.3 and discuss a more complicated example in sec. 5.2. To obtain the source code for the examples in this section please refer to sec. 8.3. To run the examples one needs to install OpenAD/F following the instructions in sec. 2.1. As in sec. 1.3 one can change the default Fortran and C compilers for these examples by setting the environemt variables `F90C` and `CC` respectively. We have tested the examples with `gfortran`, NAG `f95` and Intel's `ifort`.

5.1 Two Small Examples

Both examples follow the same prinicipal steps as the example in sec. 1.3 :

- write a `Makefile` that uses the `openad` script,
- identify independent and dependent variables, and
- write driver logic to setup the derivative computation and obtain the derivative values.

We do not explain these fundamental steps again but instead concentrate on various aspects specific to the examples.

- select the source code to be transformed,
- compute a whole Jacobian either iteratively or in vector mode, and
- control the call graph reversal mode.

5.1.1 Flow in a Driven Cavity

This example code was taken from the Minpack test problem "flow in a driven cavity" (FDC) [9]. The source code for the problem can be found under `$OPENADROOT/Examples/FDC/` in file `head.f` and a simple driver in `driver.f90` and an executable named `driver` can be produced by running

```
cd $OPENADROOT/Examples/FDC
make
```

To illustrate the use of OpenAD/F and provide a comparison with finite differences we have setups with driver source files and `Makefiles` recursively invoked by the above command in the following subdirectories

`FD` - for a finite difference approximation to the Jacobian

`TLM` - for the OpenAD/F tangent linear model, i.e. forward mode

`ADMsplit` - for the OpenAD/F adjoint model in split mode (see also, sec. 3.4)

`ADMjoint` - for the OpenAD/F adjoint model in joint mode (see also, sec. 3.4)

The respective executables are named

`driver`*subDir*

where *subDir* is one of the four choices given above. One can compare the produced output and observe matching values for

the Jacobian entries. We suggest to compare also the driver source files and the `Makefile`s to see the differences between the variants.

What to transform: Note that part of the computation is an initialization done in `init.f`. All driver files call the `init` routine but it is *not part of the transformation* because we want to differentiate only with the respect to the initial value in variable `x` and not with respect to the parameters that are used in `init`. Because the `init` routine is not transformed, its floating point argument will retain the original type while the argument type of `x` in `head` has changed to the active type. Consequently, we use a variable `x0` to compute the initial value then copy it from there into the active variable `x`.

How to obtain the whole Jacobian: In these examples we use a single scalar derivative associated with each scalar program variable. Consequently, in one sweep we can obtain in forward mode one projection $\boldsymbol{J}\dot{\boldsymbol{x}}$ or in reverse mode one projection $\bar{\boldsymbol{y}}^T\boldsymbol{J}$. This means we need either n forward sweeps or m reverse sweeps for the whole Jacobian[1]. This is reflected in the respective loops in the driver routines that wrap the calls to the differentiated `head` routine where in each loop itertation we initialize $\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}$ to a column of the $n \times n$ or $m \times m$ identity matrix and compute the respective column or row of $\boldsymbol{J}$.

Controling split and joint mode in the adjoint model: The principal behavior (split or joint mode) of the call graph reversal is encoded in the template file that the `openad` script uses which is controled by the `--mode` flag which for reverse mode is set either to `rs` or `rj`. Aside from that difference, which is evident in the respective `Makefile`s, one can see in the driver logic that the split reaches up to the top level routine `head` which in the driver is first invoked for the taping sweep (the code variant is selected by calling `OAD_rev_tape()`) and then again for the adjoint sweep (the code variant is selected by calling `OAD_rev_adjoint()`). In joint mode, on the other hand, the implicit behavior encoded in the template is that the adjoint sweep follows immediately after the taping sweep and consequently we see only one call to `head` initially for the taping variant and the template internally then selects and invokes the adjoint sweep variant of the code.

5.1.2 Box Model

This code was provided by Patrick Heimbach[2]. It is used for the solution of a (generalized) eigenvalue problem arising in physical oceanography, where substantial effort in understanding the ocean circulation's role in the variability of the climate system, on time scales of decades to millennia and beyond, is being directed at investigating the so-called 'thermohaline' circulation (THC). This refers to the contribution to the ocean circulation which is driven by density gradients and thus controlled by temperature and salinity properties and its associated fluxes. It plays a crucial role in connecting the surface to the deep ocean through deep-water formation which occurs at some isolated convection sites at high latitudes mainly in the subpolar Atlantic ocean, such as the Labrador Sea and the Greenland-Irminger-Norwegian (GIN) Seas.

The source code for the problem can be found under `$OPENADROOT/Examples/BoxModel/` in file `head.f` and a simple driver in `driver.f90` and an executable named `driver` can be produced by running

```
cd $OPENADROOT/Examples/BoxModel
make
```

and followin the same approach as in sec. 5.1.1 we have variants in the following 5 subdirectories:

`FD` - for a finite difference approximation to the Jacobian

`TLM` - for the OpenAD/F tangent linear model, i.e. forward mode

`TLMvec` - for the OpenAD/F forward vector mode

`ADMsplit` - for the OpenAD/F adjoint model in split mode (see also, sec. 3.4)

`ADMjoint` - for the OpenAD/F adjoint model in joint mode (see also, sec. 3.4)

In this example we have an additional variant for *vector mode*, see below under "How to obtain the whole Jacobian". The noteworthy differences to the example in sec. 5.1.1 are explained in the following.

What to transform: This example stores the computed data as module variables (see `all_globals_mod.f`. The module variables are accessed by the top level routine that computes the model (see `box_forward` in `head.f`) but also by initializations (for example `box_ini_fields`). Because of the common access to the global data we put all the code in `all_globals_mod.f` and `head.f` together into one file called `numCore.f` (see the respective `Makefile`s in the OpenAD/F variants of the example) and transform the entire code. After running `make` one can inspect the transformed code in `numCore.pre.xb.x2w.w2f.post.f` and find that the module variables `T` and `S` have had their type changed to `type(active)` and the transformed `box_ini_fields` correctly references them with the value component of the active type.

[1] Because one can see that the Jacobian is in fact sparse one could employ various compression techniques to reduce the effort but we do not do this here to keep the example simple.

[2] http://heimbach.wordpress.com/

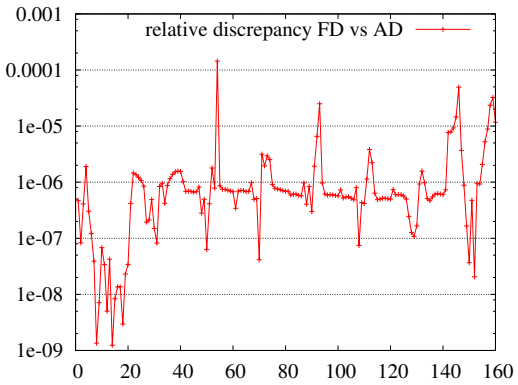


Figure 5.1: Relative discrepancy between finite differences and AD derivatives for a small shallow water model setup with 160 gradient elements and a uniform perturbation factor

Compile order: Note, that the transformed source code in `numCore.pre.xb.x2w.w2f.post.f` contains the modified module which is referenced by the driver routine so that we can access the derivative components. Therefore the compilation of `numCore.pre.xb.x2w.w2f.post.f` has to happen before the compilation of the driver which is reflected in the `Makefile`.
How to obtain the whole Jacobian: This example has variant in subdirectory `TLMvec` which computes all needed directions in one sweep by propagating an array of derivative components. The `openad` script links a different active type definition (see also sec. 2.6.1.2) when the `--mode` flag is set to `fv`, see the `Makefile`. In the example module the length of the derivative component array is hard coded but obviously one should change that definition to adapt it to the application context.

5.2 Shallow Water Model

In this section we will use a practical application to highlight advanced aspects arising for more complicated applications. The application at hand is a shallow water model used in a study [26] on bottom topography as a control variable in ocean models. This code was originally written in Fortran77 but there have been a few Fortran90 extensions to the source. It contains some of the basic language features also used in the much larger MIT general circulation model (MITgcm) [28, 3]. The model is a time stepping scheme which eventually computes a scalar valued cost function. We generate an adjoint model to compute the gradient. For a consistency check we looked at the relative discrepancy between finite differences and AD-derivatives. Even for a small setup with only 160 gradient elements the discrepancies as plotted in fig. 5.1 are significant.

5.2.1 Collect and Prepare Source Files

The entire model consists of many subroutines distributed over various source files and the existing build sequence involves C preprocessing. To perform the static code analysis as explained in sec. 4.1.1 all code that takes part in computation of the model has to be visible to the tool which could be done by concatenating into a single file. It is possible to do this for all source files of the model but in many cases this will include code for ancillary tasks such as diagnostics and data processing not directly related to the model computation. Often it is better to filter out such ancillary code.

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

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

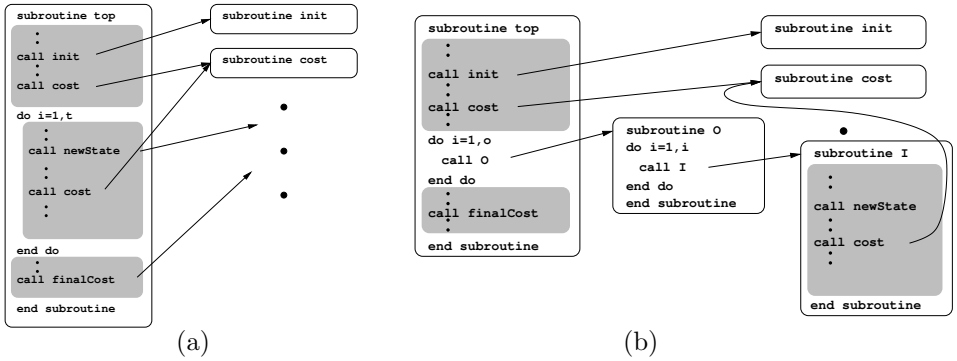


Figure 5.2: Modification of the original code (a) to allow 2 checkpointing levels (b)

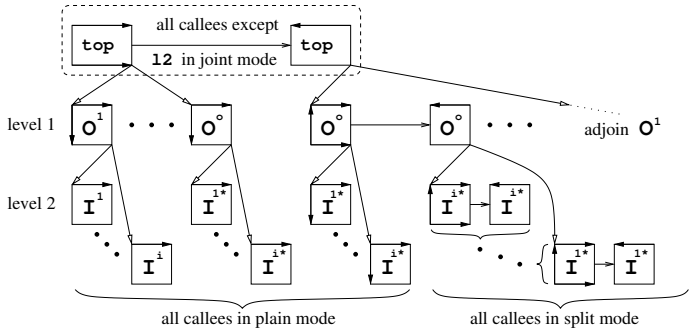


Figure 5.3: Checkpointing scheme, the `.*` indicating `.+(o-1)i`

Section 5.1 explains the need for augmenting the "driver" logic for the model computation such that it carries out the seeding and extraction of the derivative values. 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 chain. The independent and dependent variables should be identified in the top level routine.

5.2.2 Orchestrate a Reversal and Checkpointing Scheme

Joint and split reversal, see sec. 3.4 are two special cases of a large variety of reversal schemes. The model here involves a time stepping scheme controlled by a main loop. To facilitate explanations about the OpenAD/F mechanisms in this example we will disregard here the optimal Revolve scheme explained in sec. 2.6.1.9. OpenAD/F supports automatic detection of the data set to be checkpointed at a subroutine level. To use this feature the loop body is encapsulated into a inner loop subroutine denoted here for brevity `I`, see `loop_body_wrapper_inner.F`. 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` (for brevity; see `loop_body_wrapper_outer.F`) schematically shown in fig. 5.2 which is invoked `o` times[3]. Now we can describe the reversal scheme with the call tree shown in fig. 5.3. The state changes can be encapsulated in four templates.

`OADrts/ad_template.joint.f`: joint mode template for `top` and all its callees except `O`

`OADrts/ad_template.joint_split_oif.f`: for `O`

`OADrts/ad_template.joint_split_iif.f`: for `I`

`OADrts/ad_template.split.f`: for all callees of `I`

The collection of downloadable test problems contains the model and the four subroutine templates. Figure5.2(b) shows the `cost` subroutine called from `I` as well as from `top`. However, according to fig. 5.2 we would need two versions of `cost`, one that as callee of `top` is reversed in joint mode and one as callee of `I` is reversed in split mode. In order to maintain the static reversal approach one needs to duplicate `cost`.

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

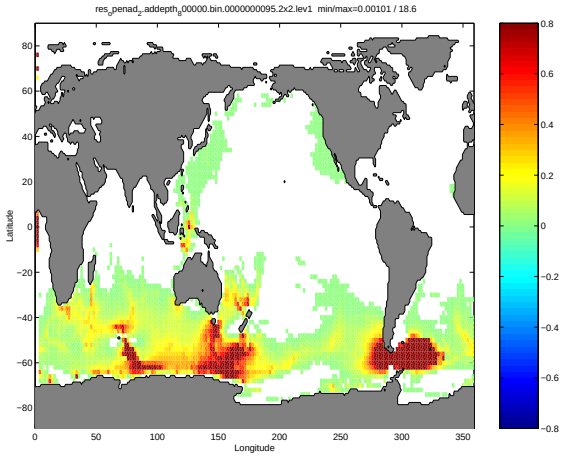


Figure 5.4: Sensitivity (gradient) map for 2×2 degree resolution

5.2.3 File I/O and Simple Loops

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

Early tests showed a considerable amount of runtime and memory spent on taping array indices used in loops. The *simple* loop concept introduced in sec. 4.1.3.7 is designed to eliminate much of this overhead. Not all loops within the given model code satisfy the conditions so as an additional step throughout the model code we identified the conformant loop constructs to the tool using the simple loop pragma. The resulting efficiency gain was about a factor 4 in run time and more than a factor 10 in memory use.

5.2.4 Results

fig. 5.4 shows as an example output a map of sensitivities of zonal volume transport through the Drake Passage to changes in bottom topography everywhere in a barotropic ocean model computed from the shallow water code by P. Heimbach. The adjoint model generated with the current version of OpenAD/F applied to the shallow water code achieves a run time that is only about 8 times that of plain model computation. We expect the ongoing development of OpenAD/F, see also sec. 7 to yield further efficiency gains.

Chapter 6

Recepies

6.1 Wrapping Higher-Level Calls using Stubs and Templates

Portions of the code that implement well defined higher-level mathematical mappings, such as linear solves, it is generally preferable to formulate the mapping for the derivatives directly rather than differentiating the code that implements the mapping. This section we provides a few examples how to orchestrate the wrapping.

6.1.1 Self Adjoint Solves

In `$OPENADROOT/Examples/SelfAdjoint/` one can find a simple example for a selfadjoint solve of the following system

$$\underbrace{\begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix}}_{=\boldsymbol{A}} \boldsymbol{x} = \boldsymbol{b} \quad .$$

The function in question is $\boldsymbol{f} : \boldsymbol{b} \mapsto \boldsymbol{x}$. Because the matrix is symmetric and fixed we can write the adjoint simply as $\bar{\boldsymbol{b}} = \bar{\boldsymbol{b}} + A^{-1}\bar{\boldsymbol{x}}$. In other words this is just another solve with the same matrix. Let the solver be implemented as a simple LU-factorization, see `lu.f90`. One can compile the original example in this directory by invoking

```
make
```

and then invoking the resulting binary

```
./driver
```

which produces the output shown in fig. 6.1. The correctness test is also implemented in `lu.f90`.

```
driver running for b =  0.50000000000000000        12.000000000000000
  test: max of |Ax-b|:    0.0000000000000000
            yields x =   7.0000000000000000       -4.5000000000000000
```

Figure 6.1: Output from `driver` of the self-adjoint example.

For the reverse mode setup we assume a simple split scheme (see also, sec. 3.4) implemented in `ADMsplit/`. For the differentiation we replace the original code from `lu.f90` with stubs provided in `ADMsplit/luStubs.f90`. The purpose of the stub is simply to establish the data dependency for the code analysis while hiding the complexity of the actual code or dealing with cases where the source code is not available in the first place. In this file we simply establish a dependency from the right-hand-side vector `b` to the solution `x`. The code analysis now can trace the path from the independent to the dependent declared in `head.f90`. The rules in `ADMsplit/Makefile` show how the `ADMsplit/luStubs.f90` and `head.f90` are concatenated, the module name is changed to the `LUstubs` module, and then the sources are transformed using the `openad` wrapper script.

During the post processing phase we use the default split mode template copied in by the `openad` script. The one exception is the transformed `ADMsplit/luStubs.f90` that contains the template pragma directing the post processor to use the template files `oad_template_solve.f90` (and also `oad_template_test.f90`) in `ADMsplit` instead. Inspecting the `ADMsplit /oad_template_solve.f90` file, see also fig. 6.2 shows that we avoid a name clash by `USE`ing the original `solve` renamed as `trueSolve` and implement the forward and reverse sweeps simply as the linear system solves indicated above.

The output created by the `ADMsplit/driverADMsplit` variant can be compared with the output created for finite differences with `FD/driverFD`, see also fig. 6.3.

```
1         subroutine template()
2           use OAD_tape
3           use OAD_rev
4           use lu , trueSolve => solve
5   !$TEMPLATE_PRAGMA_DECLARATIONS
6
7           integer iaddr
8           external iaddr
9
10          double precision , dimension(size(x)) :: passiveX, passiveB
11
12          if (our_rev_mode%plain) then
13            passiveB=b%v
14            call trueSolve(A,passiveX,passiveB)
15            x%v=passiveX
16          end if
17          if (our_rev_mode%tape) then
18            passiveB=b%v
19            call trueSolve(A,passiveX,passiveB)
20            x%v=passiveX
21          end if
22          if (our_rev_mode%adjoint) then
23            passiveX=x%d
24            call trueSolve(A,passiveB,passiveX)
25            b%d=b%d+passiveB
26          end if
27        end subroutine template
```

Figure 6.2: Split mode template for LU solve (see file `$OPENADROOT/Examples/SelfAdjoint/ADMsplit/oad_template_solve.f90`)

```
 driver running for b =  0.50000000000000000        12.000000000000000
   test: max of |Ax-b|:    0.0000000000000000
   test: max of |Ax-b|:   3.47661440246873872E-309
   test: max of |Ax-b|:    0.0000000000000000
   test: max of |Ax-b|:   3.47661440246873872E-309
AD Jacobian:     -0.4000000000000E+00     0.6000000000000E+00     0.6000000000000E+00    -0.4000000000000E+00
```

```
   test: max of |Ax-b|:    0.0000000000000000
   test: max of |Ax-b|:   1.77635683940025046E-015
   test: max of |Ax-b|:   1.77635683940025046E-015
FD Jacobian:     -0.4000000000559E+00     0.5999999999950E+00     0.5999999999062E+00    -0.3999999999671E+00
```

Figure 6.3: Output from `ADMsplit/driverADMsplit` (top) and `FD/driverFD` for the self-adjoint example.

```
1  subroutine template()
2    use OAD_tape
3    use OAD_rev
4  !$TEMPLATE_PRAGMA_DECLARATIONS
5    double precision , dimension(size(x)) :: passiveX, passiveB
6    if (our_rev_mode%plain) then
7        passiveB=b%v
8        call solve(n,A,passiveX,passiveB,trans)
9        x%v=passiveX
10   end if
11   if (our_rev_mode%tape) then
12       passiveB=b%v
13       call solve(n,A,passiveX,passiveB,trans)
14       x%v=passiveX
15   end if
16   if (our_rev_mode%adjoint) then
17       passiveX=x%d
18       call solve(n,A,passiveB,passiveX, .not. trans) ! <---- flipping the transpose flag
19       b%d=b%d+passiveB
20       x%d=0.0
21   end if
22 end subroutine template
```

Figure 6.4: Split mode template for passive UMFPACK solve (see file `$OPENADROOT/Examples/LibWrappers/UmfPack_2.2_passive/ADMsplit/Templates/solve.f90`)

6.1.2 Linear Solve with UMFPACK

Because UMFPACK [41] is not included in the OpenAD/F downloads you need to make sure to have it installed and have the environment variable `UMFPACK_LID_DIR` point to the installation directory to compile and run the examples. We use this example to also illustrate the difference between an active and a passive system matrix.

6.1.2.1 Passive System Matrix (UMFPACK v. 2.2)

This example refers to the old UMFPACK implementation v. 2.2. All related files can be found in `$OPENADROOT/Examples/LibWrappers/UmfPack_2.2_passive`. Similarly to sec. 6.1.1 we consider a linear solve but now with an asymmetric system:

$$\underbrace{\begin{bmatrix} 2 & 4 \\ 3 & 2 \end{bmatrix}}_{=\boldsymbol{A}} \boldsymbol{x} = \boldsymbol{b} \quad .$$

As opposed to sec. 6.1.1, the corresponding adjoint solve has to be done with the transpose $\bar{\boldsymbol{b}} = \bar{\boldsymbol{b}} + A^{-T}\bar{\boldsymbol{x}}$. For the solve with UMFPACK a sequence of three calls has to be issued to the routines `UMD21I` (initialization), `UMD2FA` (factorization), `UMD2SO` (backsubstitution). The last two calls are typically followed by error checks. Considering this calling sequence it is apparent that

- a naive reversal of the sequence will let the error checks precede the calls to be checked, and
- the calls themselves do not have an obvious adjoint equivalent.

The entire calling sequence as a linear solve however does. Therefore we consider the solve as implemented in `solve.f90` the unit to be wrapped. Note the `trans` parameter to quickly switch to the transposed version. Similarly to sec. 6.1.1 we use a stub, see `ADMsplit/Stubs/solve.f90` for the transformation which then is replaced in the post process step by the template indicated in the stub's template pragma, see `ADMsplit/Templates/solve.f90` which is also shown in fig. 6.4. Note the switch in the transpose parameter on the `solve` call in the adjoint branch. In general it is not guaranteed that the values of $\boldsymbol{A}$ are available without taping and restoring (see sec. 3.3). In this example the values of $\boldsymbol{A}$ happen to be set with a `data` statement (see `head.f90`) which allows their reuse in the adjoint without any extra measures.

Also, note that in difference to sec. 6.1.1 we disambiguate between the stub for `solve` and the actual `solve` wrapper by forcing a rename on all routines during the transformation with the `-r` switch, see `ADMsplit/Makefile` for the command line

flags passed to `oadDriver`. Because all routines are renamed, the calls to `head` in the driver have to be changed to a calls to `OpenAD_head`, see `ADMsplit/driverADMsplit.f90`.

The output created by the `ADMsplit/driverADMsplit` variant can be compared with the output created for finite differences with `FD/driverFD`

6.1.2.2 Active System Matrix (UMFPACK v. 2.2)

This example refers to the old UMFPACK implementation v. 2.2. All related files can be found in `$OPENADROOT/Examples/LibWrappers/UmfPack_2.2_active`. In difference to sec. 6.1.2.1 the matrix values are computed using the independents $\boldsymbol{b}$:

$$\underbrace{\left(\begin{bmatrix} 2 & 4 \\ 3 & 2 \end{bmatrix} + \boldsymbol{I}\boldsymbol{b}\right)}_{=\boldsymbol{A}} \boldsymbol{x} = \boldsymbol{b} \quad .$$

We follow [17] in deriving the respective adjoint statements as

$$\bar{\boldsymbol{b}} = \bar{\boldsymbol{b}} + A^{-T}\bar{\boldsymbol{x}} \tag{6.1}$$

and

$$\bar{\boldsymbol{A}} = \bar{\boldsymbol{A}} - \bar{\boldsymbol{b}}\boldsymbol{x}^T \quad . \tag{6.2}$$

While the setup of the example follows the one in sec. 6.1.2.1, there are a few notable differences.

- The values of $\boldsymbol{A}$ are set in the code (not with a `data` statement) and that means for (6.1) we need to tape and restore $\boldsymbol{A}$. This is done manually in the template, see fig. 6.5 lines 23-25 and lines 31-32 respectively.
- The values of the solution $\boldsymbol{x}$ are needed in (6.2). We could recompute them if we knew without more effort the values of $\boldsymbol{b}$ or alternatively we tape and restore $\boldsymbol{x}$. This is done manually in the template, see fig. 6.5 lines 20-21 and lines 37-38 respectively. Note that the order of taping (first $\boldsymbol{x}$, then $\boldsymbol{A}$) implies the inverse order for restore (first $\boldsymbol{A}$, then $\boldsymbol{x}$).

The output created by the `ADMsplit/driverADMsplit` variant can be compared with the output created for finite differences with `FD/driverFD`

```
1  subroutine template()
2    use OAD_tape
3    use OAD_rev
4  !$TEMPLATE_PRAGMA_DECLARATIONS
5    double precision , dimension(size(x)) :: passiveX, passiveB
6    double precision , dimension(size(x),size(x)) :: passiveA
7    integer :: i,j
8    if (our_rev_mode%plain) then
9        passiveB=b%v
10       passiveA=A%v
11       call solve(n,passiveA,passiveX,passiveB,trans)
12       x%v=passiveX
13   end if
14   if (our_rev_mode%tape) then
15       passiveB=b%v
16       passiveA=A%v
17       call solve(n,passiveA,passiveX,passiveB,trans)
18       x%v=passiveX
19       ! save x (do by hand what the inliner does)
20       double_tape(double_tape_pointer:double_tape_pointer+size(passiveX)-1)=passiveX(:)
21       double_tape_pointer=double_tape_pointer+size(passiveX)
22       ! save A (do by hand what the inliner does)
23       double_tape(double_tape_pointer:double_tape_pointer+ &
24            (size(passiveA,1)*size(passiveA,2))-1)=reshape(passiveA,(/size(passiveA,1)*size(passiveA,2)/))
25       double_tape_pointer=double_tape_pointer+(size(passiveA,1)*size(passiveA,2))
26   end if
27   if (our_rev_mode%adjoint) then
28       ! for b:
29       passiveX=x%d
30       ! restore A: (do by hand what the inliner does)
31       double_tape_pointer=double_tape_pointer-(size(passiveA,1)*size(passiveA,2))
32       passiveA(:,:)=reshape(double_tape(double_tape_pointer:),shape(passiveA))
33       call solve(n,passiveA,passiveB,passiveX, .not. trans) ! <---- flipping the transpose flag
34       b%d=b%d+passiveB
35       ! for the matrix: decrement by outer product (\bar{b}, x^T)
36       ! restore x (do by hand what the inliner does)
37       double_tape_pointer=double_tape_pointer-size(passiveX)
38       passiveX(:)=double_tape(double_tape_pointer:double_tape_pointer+size(passiveX)-1)
39       do j=1,n
40          do i=1,n
41             A(i,j)%d=A(i,j)%d-passiveB(i)*passiveX(j)
42          end do
43       end do
44       ! reset
45       x%d=0.0
46   end if
47 end subroutine template
```

Figure 6.5: Split mode template for active UMFPACK solve (see file `$OPENADROOT/Examples/LibWrappers/UmfPack_2.2_active/ADMsplit/Templates/solve.f90`)

Chapter 7

Modifying OpenAD/F

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

OpenAD/F allows users a great amount of flexibility in the use of the code transformation and permits interventions at various stages of the transformation process. We would like to emphasize the fact that for large scale applications the efficiency of checkpointing and taping can be improved merely by modifying the implementation of the run time support, the template and inlining code. They are not conceived to be just static deliverables of OpenAD/F but rather are part of the interface accessible to the user. It is not the intention to stop with a few prepackaged solutions as one would expect from a monolithic, black-box tool. True to the nature of an open source design, the interface is instead conceived as a wide playground for experimentation and improvement. Expanding on the schematic depiction of the tool workings in fig. 1.1 we want to highlight the options to modify the transformation and the tool itself in fig. 7.1 at different levels of complexity reaching from the casual user to actual coding work in the tool's components. As part of using OpenAD/F for different applications we see a growing number of variations to the transformation and run time support implementations available to the user.

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

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

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

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

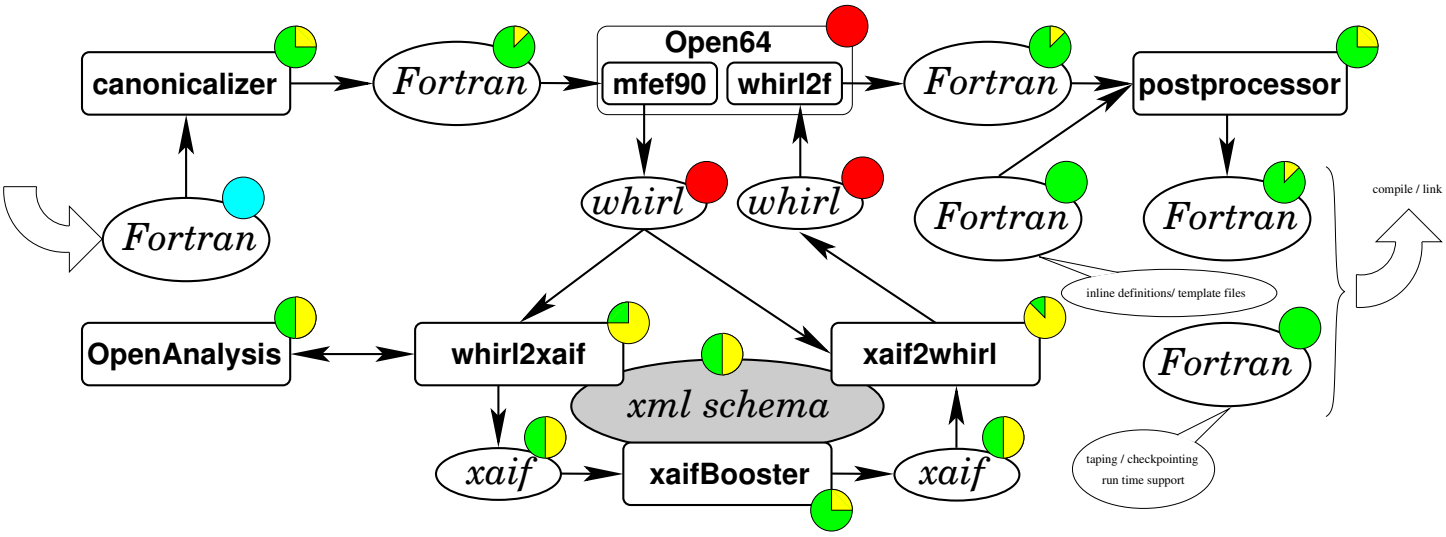


Figure 7.1: Levels of complexity for modifications

Chapter 8

Miscellaneous

8.1 Changes relative to the ACM TOMS paper

Because of continued development since [46] was finalized we maintain the following list of significant changes.

Renamed files and directories In a number of cases the names and locations of files and tool components turned out to be inappropriate or misleading. Consequently a number of changes have been introduced to rectify the situation.

- The binaries in `test/t` under `$OPENADROOT/xaifBooster/system` and the algorithm directories have been renamed to `driver/oadDriver` and the underlying source files from `t.cpp` to `oadDriver.cpp`.
- the script to get the components has been renamed from `goad` to `openadUpdate` and command line flags have been added.

Changed command line flags Because of the ongoing development new command line flags have been added. In particular the replacement for the `goad` script which is called `openadUpdate` has command line flags that were not present for `goad`. Details can be found in sec. 4.3.

Refactorized Algorithms

8.2 Regression Tests

All regression tests can be added to the OpenAD source tree using the `-t` flag with the `openadUpdate` script, see sec. 4.3.1. The test setups are in the following locations.

- `$OPENAD/Regression/`which exercises the entire tool chain (requires `gnuplot` [1])
- `$OPENAD/OpenADFortTk/Regression/`which exercises the translation between whirl and XAIF, see sec. 4.2.3
- `$OPENAD/OpenADFortTk/tools/SourceProcessing/Regression/`which exercises the Fortran parser that is used in the pre and post processing, see sects. 4.2.1,4.2.4
- `$OPENAD/Open64/osprey1.0/tests/`which is included in the Open64 repository and exercises the Fortran frontend, see sec. 4.2.2

In each directory is a main driver script called `testAll.py`. The `-h` option shows all available options which are self explanatory. Note that the test sets include test cases that have been included to illustrate certain problems with the tool and not all of these problems have been fixed. The expected outcome is kept in the respective reference outputs and the latest update is listed in tables on the OpenAD website. With the `-i` option the `testAll.py` script will skip test cases expected to fail and otherwise prompt for execution. All test sets except for the first one have a simple boolean outcome. Exercising the entire tool chain permits a consistency check against finite differences. We use a default tolerance to accommodate variations in hardware, compilers and compiler optimization but for some problems because of cancellations etc the tolerance is exceeded either for the absolute or relative discrepancy. To facilitate maximal use of the test cases we compare the intermediate stages of the transformation in the tool chain are kept as references for forward and reverse mode (joint and split) setup. To illustrate the numerical comparison we use a test called `boxmodel` that can in the `$OPENAD/OpenADFortTk/Regression/`invoke

[1] see `http://www.gnuplot.info/`

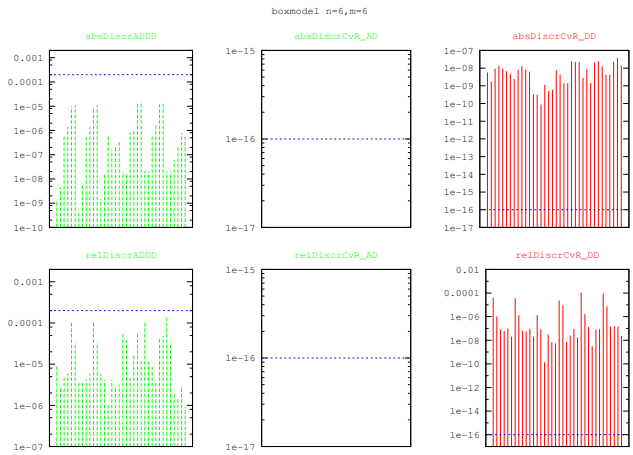


Figure 8.1: Example for numerical discrepancy shown for test case `boxmodel` for forward mode.

like this

```
./testAll.py boxmodel
```

Hitting `enter` on all subsequent prompts will execute the test case with the default settings (forward mode using the `ifort` compiler without optimization but extra checking enabled). The numerical discrepancies are shown using gnuplot with output similar to the the one shown in fig. 8.1.

8.3 Compiling and Contributing to this Manual

The LaTeX sources of this manual are kept under mercurial revision control. Details how to download the manual can be found on the OpenAD/F website under "How to Contribute". In addition to LaTeX(in particular `pdflatex`) one also needs `dot`, `fig2dev`, and `dia` for the conversion of figures. The manual's `Makefile` requires a source installation of OpenAD/F from the source code repository with all extra tests and examples using

```
openadUpdate -et
```

including the prerequisites to run regression tests, see sec. 8.2. The environment must be setup as described in sec. 2.2. The manual refers to a **set of code examples** which are linked from the OpenAD/F installation by the `Makefile`. It can be build in one step as follows.

```
cd OpenADF_Manual/Manual
make
```

Appendix

`Makefile`	the top level `Makefile`
`utils/`	utility classes (debugging, generic traversal, etc.)
`tools/`	code generator supporting XAIF parser
`boostWrapper/`	wrapper classes for the boost graph library
`system/`	all basic data structures, XAIF (un)parsing, sec. 4.1.3.1
`algorithms/`	see the subdirectories below
`CodeReplacement`	support library for subroutine templates
`CrossCountryInterface`	support library for elimination strategies, sec. 4.1.3.4
`DerivativePropagator`	support library for Jacobian vector products
`InlinableXMLRepresentation`	support library for inlinable subroutine calls
`Linearization`	Linearization transformation, sec. 4.1.3.3
`BasicBlockPreaccumulation`	elimination with angel and preaccumulation at the basicblock level, sec. 4.1.3.4
`MemOpsTradeoffPreaccumulation`	as above but with different heuristics than angel
`ControlFlowReversal`	control flow graph reversal
`BasicBlockPreaccumulationReverse`	adjoint code
`BasicBlockPreaccumulationTape`	taping code supporting adjoint, sec. 4.1.3.8
`BasicBlockPreaccumulationTapeAdjoint`	reverse sweep portion supporting adjoint,sec. 4.1.3.8

Table 8.1: Directory structure in xaifBooster

Bibliography

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

[2] AD02 (part of the Harwell Subroutine Library). http://www.hsl.rl.ac.uk/archive/specs/hsl_ad02.pdf.

[3] Alistair Adcroft, Jean-Michel Campin, Patrick Heimbach, Chris Hill, and John Marshall. The MITgcm. Online documentation, Massachusetts Institute of Technology, USA, 2002.

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

[5] Adjoint Compiler Technology & Standards (ACTS). http://www.nsf.gov/awardsearch/showAward?AWD_ID=0205590.

[6] ADOL-C. https://projects.coin-or.org/ADOL-C.

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

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

[9] Brett M. Averick, Richard G. Carter, Jorge J. Moré, and Guo-Liang Xue. The MINPACK-2 test problem collection. Technical Report Preprint MCS-P152-0694, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[10] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors. *Computational Differentiation: Techniques, Applications and Tools*, Philadelphia, PA, 1996. SIAM.

[11] Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and Jean Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008.

[12] Boost. http://www.boost.org.

[13] H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, 2005.

[14] George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, New York, NY, 2002. Springer.

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

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

[17] Mike B. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In Bischof et al. [11], pages 35–44.

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

[19] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.

[20] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Griewank and Corliss [19], pages 126–135.

[21] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, mar 2000. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[22] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[23] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

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

[25] Paul D. Hovland, Uwe Naumann, and Boyana Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 530–538, Anaheim, CA, 2002. ACTA Press.

[26] M. Losch and C. Wunsch. Bottom topography as a control parameter in an ocean circulation model. *Journal Of Atmospheric And Oceanic Technology*, 20(11):1685–1696, 2003.

[27] Mercurial. http://www.selenic.com/mercurial/.

[28] MIT general circulation model (MITgcm). http://mitgcm.org.

[29] Uwe Naumann. Elimination techniques for cheap Jacobians. In Corliss et al. [14], chapter 29, pages 247–253.

[30] Uwe Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming, Ser. A*, 99(3):399–421, 2004.

[31] Uwe Naumann and Peter Gottschling. Simulated annealing for optimal pivot selection in Jacobian accumulation. In Andreas Albrecht and Kathleen Steinhöfel, editors, *Stochastic Algorithms: Foundations and Applications*, number 2827 in Lecture Notes in Computer Science, pages 83–97. Springer, 2003.

[32] Uwe Naumann and Jean 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, Proceedings of the International Conference on Computational Science, Atlanta, GA, USA, May 22-25, 2005, Part I*, volume 3514 of *Lecture Notes in Computer Science*, pages 338–346, Berlin, 2005. Springer. also as ANL preprint ANL/MCS-P1226-0205.

[33] Network Enhance Optimization Server (NEOS). http://www.neos-server.org/.

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

[35] OpenAD. http://www.mcs.anl.gov/OpenAD.

[36] OpenAnalysis. http://openanalysis.berlios.de/.

[37] OpenMP. http://www.openmp.org.

[38] Python. http://www.python.org/.

[39] Rapsodia. http://www.mcs.anl.gov/Rapsodia/.

[40] ROSE. http://rosecompiler.org/.

[41] UMFPACK. http://www.cise.ufl.edu/research/sparse/umfpack/.

[42] Jean Utke. Flattening basic blocks. In Bücker et al. [13], pages 121–133.

[43] Jean Utke, Andrew Lyons, and Uwe Naumann. Efficient reversal of the interprocedural flow of control in adjoint computations. *Journal of Systems and Software*, 79:1280–1294, 2006.

[44] Jean Utke and Uwe Naumann. Software technological issues in automating the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications, Proceedings of the Seventh IASTED International Conference*, pages 417–422. ACTA Press, 2003.

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

[46] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.

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

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