OpenAD: Algorithm Implementation User Guide

by

J. Utke

Mathematics and Computer Science Division

Technical Memorandum No. 274

04/16/2004

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.
Argonne National Laboratory, a U.S. Department of Energy Office of Science Laboratory, is operated by The University of Chicago under Contract W-31-109-ENG-38.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor The University of Chicago, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof.

Available electronically at http://www.doe.gov/bridge
Available for a processing fee to U.S. Dept. of Energy and its contractors, in paper, from:
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov
Contents

Abstract 1

1 Introduction 1

2 Current State of OpenAD Development 3
  2.1 Representation of the Numerical Kernel ................. 3
  2.2 Basic Functionality and Higher-Level Algorithms .......... 4
    2.2.1 Linearization ..................................... 5
    2.2.2 Basic Block Preaccumulation .......................... 6
    2.2.3 Taping the Basic Block Preaccumulation ............... 7
    2.2.4 Adjoint of the Taped Basic Block Preaccumulation .... 7
    2.2.5 Reversing the Control Flow Graph ........................ 8
    2.2.6 Reversing the Basic Block Preaccumulation .......... 8
    2.2.7 Heuristics for Minimizing Operations and Data Locality 9
  2.3 Parsing and Unparsing .................................. 9

3 User Extensions to Algorithms 10
  3.1 Algorithm Objects ..................................... 10
  3.2 Algorithm Invocation ..................................... 11
  3.3 Algorithm Factories ...................................... 11
  3.4 Algorithm Factory Instantiation .......................... 12
  3.5 Algorithms and Inheritance ................................ 13
  3.6 Algorithm Inheritance and Unparsing ....................... 13
  3.7 Algorithm Object Interactions ............................. 14
  3.8 Linearization as Case Study ............................... 14
    3.8.1 Expression Edge ..................................... 15
    3.8.2 Expression Vertices .................................. 15
    3.8.3 Expression .......................................... 15
    3.8.4 Assignment .......................................... 16
    3.8.5 Invocation .......................................... 17

4 Conclusion 17

References 17
OpenAD: Algorithm Implementation User Guide

J. Utke

Abstract

Research in automatic differentiation has led to a number of tools that implement various approaches and algorithms for the most important programming languages. While all these tools have the same mathematical underpinnings, the actual implementations have little in common and mostly are specialized for a particular programming language, compiler internal representation, or purpose. This specialization does not promote an open test bed for experimentation with new algorithms that arise from exploiting structural properties of numerical codes in a source transformation context.

OpenAD is being designed to fill this need by providing a framework that allows for relative ease in the implementation of algorithms that operate on a representation of the numerical kernel of a program. Language independence is achieved by using an intermediate XML format and the abstraction of common compiler analyses in Open-Analysis. The intermediate format is mapped to concrete programming languages via two front/back end combinations. The design allows for reuse and combination of already implemented algorithms.

We describe the set of algorithms and basic functionality currently implemented in OpenAD and explain the necessary steps to add a new algorithm to the framework.

1 Introduction

This paper focuses on the implementation of automatic differentiation (AD) via source transformation, with the specific goal of implementing advanced schemes for the calculation of derivatives using checkpointing for various reversal schemes. We briefly discuss the theoretical aspects of AD as they relate to the algorithms introduced. For a detailed introduction to the theory and practice of AD, the interested reader should particularly consider [5] but also [6, 2, 3].

The basic approaches for the implementation of AD tools can be categorized as operator overloading and source code transformation. The AD community generally agrees that many advanced AD algorithms should use information typically available to compilers. Consequently, source transformation tools built in a compiler context should be the implementation environment of choice.

The code representing the computation of a numerical function \( f : \mathbb{R}^n \to \mathbb{R}^m \) is analyzed and augmented to compute not only \( f \) itself but also derivative information, which in typical applications includes gradients and Jacobian or Hessian projections. The analyses needed for the code augmentation are similar to analyses performed by compilers for code optimization.

Source-to-source transformation tools usually have a tight coupling to a compiler- and language-specific internal representation (IR). This has obvious benefits in allowing direct access to compiler- and language-specific optimizations and shortcuts. Little emphasis is placed on providing an interface for algorithm extensions and implementation of new algorithms to the AD community. Consequently, any such attempt to implement new algorithms
Figure 1: Dependencies between the front ends, OpenAnalysis, and OpenAD components

in different languages or compilers requires familiarity with the specifics of a compiler’s IR. In an academic setting this proved too steep a hurdle for contributors outside a tool’s original developer group.

The Adjoint Compiler Technology & Standards\(^1\) (ACTS) project has a twofold goal. One goal is to provide a framework that allows relatively easy algorithm implementation and testing. This framework is called OpenAD. The other goal is to implement an adjoint compiler advanced enough to handle the computationally challenging applications of the MIT general circulation model\(^2\). This compiler will be derived as one instantiation of the algorithms implemented in the OpenAD framework. The main components as of this time are shown in Figure 1 in the gray box. Language-specific front ends (Open64\(^3\) for Fortran and EDG\(^4\)/Sage 3\(^5\) for C/C++) parse the code into their specific IRs.

The OpenAD framework contains conversion components that extract the numerical kernel of the source code computing \(f\) and translates the kernel to a language-independent XML representation called xaif\(^6\). The translation includes the results of common analyses performed in OpenAnalysis\(^7\). The AD-specific analysis and transformation are encapsulated in the xaifBooster component. On return, the transformed xaif is sent back to the translator components, which create source code in the original programming language in conjunction with the compiler front ends. So far the development of OpenAD has concentrated on a group of techniques that center on the preaccumulation of local Jacobians of all basic blocks by using a near-minimal number of arithmetic operations. Consider a basic block to be represented by the associated computational graph, a directed acyclic graph.

\[
\begin{align*}
x_1 &= x_1 \cdot \cos(x_1 \cdot x_2) & A_1 \\
y_1 &= \sin(x_1) & A_2 \\
y_2 &= x_1 \cdot x_2 & A_3
\end{align*}
\]

Figure 2: Code for \(f\)

\(^1\) See www.autodiff.org/ACTS.

\(^2\) See mitgcm.org.

\(^3\) See hipersoft.cs.rice.edu/open64.


\(^5\) See www.llnl.gov/CASC.

\(^6\) See www.mcs.anl.gov/xaif.

\(^7\) See www.hipersoft.rice.edu/openanalysis.
For example, the basic block with three assignments $A_{1,2,3}$ in Figure 2 implements the vector function $y = f(x) : \mathbb{R}^2 \rightarrow \mathbb{R}^2$, with the corresponding computational graph shown in Figure 3. The linearization of $G$ associates local partial partial derivative with its edges; see [5]. Reuse of intermediate values leads to a breakup of the assignments original right-hand sides; see Figure 4. Subsequent application of eliminations $E_1 - E_7$ in $G$ (see [7]) imply chain rule operations on the local partial derivatives until all elements of the Jacobian $J = \frac{\partial y_i}{\partial x_j} \bigg|_{i,j=1,2}$ are computed. In Figure 4 we employ vertex elimination according to the lowest Markowitz degree.

The OpenAD framework provides access to various elimination techniques on vertices, edges, and faces; see [9]. The order of elimination steps in $G$ is crucial for the number of operations incurred and thereby contributes to the efficiency of the generated derivative code. Minimizing the operations count presents a hard combinatorial problem. Through heuristics implemented in OpenAD and access to external libraries, we provide a set of algorithms for computing near-optimal orderings for a given code [1, 10].

![Figure 3: G](image)

![Figure 4: Code for $f$ and $J$](image)

2 Current State of OpenAD Development

The development of the OpenAD framework is in progress. The IR at its core and some fundamental algorithms are stable and are briefly described in the following sections. In Section 3 we provide a short guide on extending the available functionality with new algorithms. For downloading all necessary components and build instructions, please contact the authors. The software design decisions made during the development are explained in more detail in [11]. The following contains numerous references to the code base of OpenAD. For the sake of brevity in this paper we use some abbreviations that have their full name listed in Table 1.

2.1 Representation of the Numerical Kernel

For the transformation of numerical codes it is sufficient to represent the numerically relevant portions and their connecting structures. The intermediate format xaif represents these necessary elements.
The xaifBooster IR in turn closely resembles the respective structures one would find in a compiler’s IR. It is implemented as a hierarchy of C++ classes using the boost graph library\(^8\) and the Standard C++ Library\(^9\). Figure 5 shows a simplified inheritance hierarchy. The classes representing IR elements are named in a fairly self-explanatory fashion. The class hierarchy is most easily browsed through the doxygen\(^10\) html pages that can be generated through make doc and can be found in xaifBooster/system/doc/html. The IR class hierarchy uses composition of its elements from top down. The top element is a single CallGraph instance. Figure 6 shows a simplified composition hierarchy. This defines ownership of any dynamically allocated elements as well. Where possible, the interfaces avoid the need for explicit dynamic allocation of members outside the methods of the containing class. Only in cases of containment of polymorphic elements is explicit dynamic allocation necessary. In such cases the container class interface 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, and so forth; see also Section 3.5.

### 2.2 Basic Functionality and Higher-Level Algorithms

The basic functionality provided in the IR consists of the following:

---

\(^8\) See [www.boost.org](http://www.boost.org).


\(^10\) See [www.doxygen.org](http://www.doxygen.org).
• Parsing from and unparsing to xaif; see also Section 2.3.

• Building the IR as a C++ class hierarchy with hooks for user implementable algorithms; see also Section 3.

• Passes through the IR in a compiler-like fashion.

• Parsing and building of a catalogue of intrinsics (InlinableIntrinsicsCatalogue) along with their respective local partial derivatives. The catalogue can be extended by users to include user-defined intrinsics.

• Basic debugging and logging facilities.

Based on the IR, we have built the following algorithms.

2.2.1 Linearization

We already referred to linearization in Section 1 and will use it as an example for algorithm implementation in Section 3.8. For each Assignment we create assignments for the respective local partial derivatives using the information in the intrinsics catalogue. This approach may lead to splitting the original assignment into multiple assignments in order to reuse intermediate values needed for the derivatives. The linearization also includes the reduction of the assignment’s right-hand side expression graph to the active portions. This localized activity analysis is usually not done by the compiler analysis. The linearization also handles possible aliasing of the right-hand side to any input in the left-hand side by delaying the assignment to the left-hand side until after all partials have been computed.
2.2.2 Basic Block Preaccumulation

The basic block preaccumulation algorithm uses linearization through inheritance. The application of elaborate elimination techniques in computational graphs offers an advantage only if the computational graph contains more than a single right-hand side of an assignment.\(^\text{11}\) The combination of multiple right-hand sides in a computational graph (also known as flattening) uses alias and def/use chain information provided by OpenAnalysis through the front end.

A simple example is shown in Figure 7. Ambiguities are resolved through segmentation of the computational graph into unambiguous subgraphs. Each subgraph is a structural representation with back references to its generating assignments. Once a sequence of unambiguous subgraphs is built, the algorithm invokes an external mechanism that, using various heuristics, produces an efficient elimination sequence in terms of the edges of the subgraph; see Figure 8. Based on this elimination sequence, a series of assignments is generated whose right-hand sides each represent an elimination step. Finally, the resulting Jacobian entries are the factors in saxpy operations that implement a (sparse) Jacobian/vector or vector/Jacobian product. In the example we show three elimination steps where the second step represents the Jacobian entry \(\frac{\partial y}{\partial x}\) and the third step \(\frac{\partial y}{\partial z}\). Currently the front-end-generated code uses a special type that has a tangential derivative component \(\dot{v}\) for each active variable \(v\).

The corresponding initialization and saxpy\((a, x, y)\) operations \((y = a \ast x + y)\) in the example are shown in Figure 9. The algorithm handles possible aliasing between inputs and outputs of the respective subgraph through explicit assignment to temporaries. These temporary assignments and all initialization and saxpy operations are contained in special xaii elements under DerivativePropagator. A support library implementing the derivative type and all related operations allows for a direct forward mode using the preaccumulated local Jacobians.

A different driver version, which pushes these local Jacobian entries along with the saxpy operations onto a stack, allows for a split reverse mode [5] and, in particular, a direct combination of the forward/reverse run using the same code base. The reversal relies on popping the saxpy operations from the stack and executing an adjoint interpretation. This algorithm uses the libraries created in \(A_{dte}/CrossCountryInterface\) for the subgraph rep-

\(^{11}\) The single expression use property of stand-alone right-hand sides leads to a known optimal elimination sequence.
representation and the library created in $A_{dir}/\text{DerivativePropagator}$ for the representation of the \texttt{DerivativePropagator} elements. The elimination sequence is computed by the external ANGEL (\texttt{Automatic Differentiation Nested Graph Elimination}) library.\footnote{See angellib.sourceforge.net.} The code can be found in $A_{dir}/\text{BasicBlockPreaccumulation}$.\footnote{We note that constant Jacobian entries can be generated directly in the adjoint code.}

### 2.2.3 Taping the Basic Block Preaccumulation

Our third algorithm is similar to that described in Section 2.2.2, with the difference that, instead of letting the run-time support library reinterpret certain \texttt{saxpy} operations as pushes to a stack (also called \textit{tape}), we insert an explicit push for all variable Jacobian entries.\footnote{See angellib.sourceforge.net.}

The respective push calls for the example in Section 2.2.2 are shown in Figure 10. To allow for a variety of taping and checkpointing storage formats and to maintain language independence, we require the front end to handle inlinable subroutine calls. Doing so avoids the overhead associated with subroutine calls. The actual code to push and pop values from a stack representation is language specific. In the xaij the inlinable subroutine calls have partial argument lists that contain only the information related to the numerical kernel. The taping itself is useful only in the context of a reversal scheme; see Section 2.2.6. One possible reversal scheme uses an enclosing code template to control the invocation of taping, checkpointing, and adjoint versions of any given subroutine within the code for $f$; see Figure 13. This wrapping code contains logic providing access to the stack. Thus, the stack-related parameters of the inlinable subroutines are handled by the front end without involvement of the xaij. This algorithm extends the functionality of the basic block preaccumulation through inheritance and uses the library created in $A_{dir}/\text{InlinableXMLRepresentation}$ for representing inlinable subroutine calls. The code can be found in $A_{dir}/\text{BasicBlockPreaccumulationTapeAdjoint}$.

### 2.2.4 Adjoint of the Taped Basic Block Preaccumulation

A fourth algorithm, built on top of the algorithms in Sections 2.2.2 and 2.2.3 through inheritance, creates adjoint code for a given basic block in terms of the preaccumulated Jacobians. This is done by reversing the order of all \texttt{DerivativePropagators} and their contained operations and replacing each operation with its respective adjoint equivalent in the shape of an inlinable subroutine call. Each push created in Section 2.2.3 is replaced by its equivalent pop inlinable subroutine element. The respective operations for the example in Section 2.2.2 are shown in Figure 11. Again the implementation uses a special type that has an adjoint derivative component $\bar{a}$ for each active variable $a$. The code can be found in $A_{dir}/\text{BasicBlockPreaccumulationTapeAdjoint}$.
2.2.5 Reversing the Control Flow Graph

Our fifth algorithm takes the control flow graph (CFG) of a given subroutine and produces its reverse representation. For a structured CFG [8] the reversal essentially requires a redirection of the edges and a reinterpretation of the nodes such that, for instance, a “for” loop end turns into a “for” loop begin and a branch node becomes a merge node (and vice versa). The algorithm augments the code created in Section 2.2.3 to include new basic blocks containing statements that record a particular path through the CFG by pushing condition values and loop bounds onto a stack. Consequently, the reverse CFG has the respective statements that pop those values from the stack and thereby find the corresponding reverse path through the reversed CFG. In Figure 12 we show a sample CFG in its augmented and reversed versions.\textsuperscript{14} The augmenting basic blocks are marked with “0”; all other vertices have a number marker. Their equivalents in the reversed CFG have the respective negative marker. The edge labels indicate condition values. The code can be found in \texttt{A\_dir/ControlFlowReversal}.

2.2.6 Reversing the Basic Block Preaccumulation

Our sixth algorithm involves the integration of the transformations produced by the algorithms in Sections 2.2.3, 2.2.4, and 2.2.5 representing a particular version from the set of possible adjoint codes. We note that this is not the reverse mode in the usual strict definition of elemental partials but rather a reverse mode through the preaccumulated Jacobians. Whether this approach yields increased efficiency depends on the application. A comparison of this reversal mode with other derivative computing schemes in AD and a characterization of favorable scenarios are beyond the scope of this paper.

\textsuperscript{14} This is debugging output produced by OpenAD using \texttt{boost} together with the \texttt{graphviz} utility; see \url{www.research.att.com/sw/tools/graphviz/}.
As mentioned in Section 2.2.3, we rely on code templates to enclose the different transformations of the original code that each underlying algorithm produces. For brevity we show some C-style pseudo-code in Figure 13 that roughly resembles the output to be expected if the example code from Section 2.2.2 is wrapped in some subroutine foo(x,z,y) and transformed to R_foo(x,z,y,control) and active types have a value component v and a derivative component d. For a “split” mode, R_foo would be called with control=tape and subsequently with control=reverse. These calls would be part of the adjoint code generated for the caller of R_foo. The adjoint code generation for subroutine calls as part of this algorithm is coupled with checkpointing schemes and is also beyond the scope of this paper. The code can be found in \texttt{A}\textsubscript{dir}/BasicBlockPreaccumulationReverse.

2.2.7 Heuristics for Minimizing Operations and Data Locality

Our last algorithm replaces the call to the ANGEL library used in Section 2.2.2 to generate an elimination sequence by its own implementation of some elimination techniques and heuristics. It reuses the algorithm in Section 2.2.2 and the interface defined by the library in \texttt{A}\textsubscript{dir}/CrossCountryInterface. The emphasis is on heuristics that not only minimize the operations count but also consider data locality. The latter is obviously significant for the run time of the generated code and is a step toward similar optimizations a compiler might perform.

The hierarchy of algorithms implemented so far and the ability to reuse algorithms indicate that the chosen design represents a workable solution. Assuming relative stability of the interfaces of these algorithms and the IR the same ability to reuse algorithms should extend to the greater AD community. In Section 3 we briefly introduce the essential coding steps for algorithm implementation.

2.3 Parsing and Unparsing

Parsing is done through the Xerces C++ XML parser\footnote{See xml.apache.org/xerces-c.} such that the XML element handler implementations build the IR from the top down. XAIFBaseParser implements the parsing. It has to be initialized using initialize(), and a subsequent call to parse() supplying the name of the input xaif file builds the IR. According to the schema we expect a single, not necessarily connected, call graph. The call graph instance is associated with the singleton ConceptuallyStaticInstances from which we can access it via getCallGraph(), which is used by the parser and any driver code. InlinableIntrinsicsParser is the equivalent...
3 User Extensions to Algorithms

An algorithm is implemented in its own namespace by a set of algorithm classes associated with classes in the IR. In the following sections we describe the necessary steps to implement a user algorithm in abstract terms, with examples referring to existing implementations.

Note: By convention the user creates a new directory $A_{dir}/A_{name}$ and chooses $A_{ns}$ as the namespace for the algorithm.

3.1 Algorithm Objects

Every relevant class named $c_{IR}$ in the xaifBooster IR has a pointer to an algorithm instance of $c_{IR}$AlgBase, which is the binding naming convention for all algorithm base classes. All constructors of $c_{IR}$ have a bool parameter makeAlgorithm, which defaults to true. The respective constructor definitions contain code that instantiates the algorithm object via a factory (see Section 3.3), as in the following example.

```cpp
Assignment::Assignment(bool theActiveFlag,
    bool makeAlgorithm)
    : myRHS(makeAlgorithm), myAssignmentAlgBase_p(0), myActiveFlag(theActiveFlag)
{ if (makeAlgorithm)
    myAssignmentAlgBase_p=AssignmentAlgFactory::instance()->makeNewAlg(*this);
}
```

The $c_{IR}$ destructor is responsible for deleting the instance. All $c_{IR}$AlgBase objects have a back reference to their containing $c_{IR}$ instance accessible via getContaining(). As a rule, all objects in the IR created during the parsing of the input-xaif are created with their respective algorithm instance. A $c_{IR}$ instance may not need an associated algorithm instance when it is instantiated in an algorithm itself. For example the member myDelayedLHSAssignment_p of xaifBoosterLinearization::AssignmentAlg is used to represent an additional assignment as the result of the transformation steps performed by the linearization process and is not itself subject to the transformation process applied to the original assignments. This mechanism avoids potentially costly instantiations. It does not directly determine which object instances take part in an algorithm invocation; see also Section 3.2.

The naming convention for all specific algorithm classes is $c_{IR}$Alg, and they are distinguished by namespace. The namespace xaifBooster is reserved for OpenAD utilities and the IR.

Note: In namespace $A_{ns}$ the user creates a new $c_{IR}$Alg for each $c_{IR}$ in the IR for which analysis and transformations need to be implemented; see also Section 3.2.

This approach limits the implementation overhead for any modification, since we require only new code for the $c_{IR}$ to be modified.
### 3.2 Algorithm Invocation

We expect algorithms to operate in a compiler-like fashion that involves analysis and modifications during one or more passes through the IR and the associated algorithm objects. This process is facilitated by virtual methods in class `GTI`. All relevant `cIR` objects in the IR inherit from `GTI` and all leaf classes implement at least the `traverseToChildren` method, which determines the traversal through the IR. As a rule, the traversal proceeds first to the algorithm object associated with `cIR`, followed by its direct children, namely, the elements of the IR that are members of `cIR`, as in the following example:

```cpp
void Assignment::travTo(const GenericAction::GenericAction_E anAction_c) {
    getAssignmentAlgBase().genTrav(anAction_c); // traversal to algorithm
    myLHS.genTrav(anAction_c); // traversal to right hand side
    myLHS.genTrav(anAction_c); // traversal to left hand side
}
```

Each unit of work is to be accomplished by a particular pass; the passes are enumerated `ALGORITHM_ACTION_1[1,2,...]` in `GenericAction` and stand for the invocation of the respective virtual methods `GTI::algorithm_action_1[1,2,...]`. The default implementation for these methods is empty, and as a rule no `cIR` or `cIRAlgBase` reimplements any action. The diagram in Figure 14 illustrates the setup. While the `cIRAlgBase` instances are mere placeholders, the derived `cIRAlg` classes implement the action appropriate for each pass.

**Note:** The user reimplements `algorithm_action_1[1,2,...]` where needed as entry points for the algorithms analyses and transformations, as explained in Section 3.1.

### 3.3 Algorithm Factories

All `cIRAlg` objects are instantiated through their respective factories, by convention named `cIRAlgFactory`. Each `cIR` with a `cIRAlgBase` has a corresponding `cIRAlgFactory`. This is the root base class for all derivations `A_base::cIRAlgFactory`; see also Section 3.5. The declaration and definition of these factory classes are highly repetitive. At the moment we use macros for the declaration and definition of these classes, as in the following example.
The algorithm class declaration refers to its super class and the algorithm-specific factory manager (see Section 3.4), which necessitates the two includes. The class definition is similar.

The algorithm class definition refers to $c_{IR}$, $c_{IR}$AlgBase, $A_{vs} :: c_{IR}$AlgFactory and its own declaration, which necessitates the four includes. All macro definitions can be found in xaifBooster/system/inc/AlgFactory.hpp.

Note: Following the above patterns, the user implements $A_{vs} :: c_{IR}$AlgFactory for each $c_{IR}$Alg created in Section 3.1.

Figure 15: Algorithm inheritance IR

### 3.4 Algorithm Factory Instantiation

Initially all algorithm objects are instantiated through factories during the parsing of the xaif. The factories instantiating the $c_{IR}$AlgBase objects represent the default factory set. Any algorithm supplying a $c_{IR}$Alg needs to replace the default factory with its own respective factory. This is done in an algorithm-specific AlgFactoryManager class, which needs to reimplement the init method, as in the following example.

```cpp
void AlgFactoryManager::init() {
  // default factory set
  xaifBooster::AlgFactoryManager::init();
  // replace default for ArgumentAlg
  resetArgumentAlgFactory(new ArgumentAlgFactory());
  // replace default for AssignmentAlg
  resetAssignmentAlgFactory(new AssignmentAlgFactory());
  // etc.
}
```
As a rule, the reimplementation calls the `init` of the respective parent(s) (see also Section 3.5) and then replaces its specific factories. Any code using the algorithm needs to instantiate the `AlgFactoryManager` and call `init` before the parsing of the input `xaif`.

**Note:** Following the above pattern, the user implements $A_{rel}::AlgFactoryManager$.

### 3.5 Algorithms and Inheritance

Algorithms should be reusable in different contexts. For example, the linearization of a given code is needed for the preaccumulation of Jacobians in a basic block, which in turn can be reused for a reverse mode that operates on preaccumulated Jacobians. The primary vehicle for reuse is inheritance of the algorithm objects, which implies inheritance in the associated factories and the factory manager class. In Figure 15 we show the inheritance and the partial replacement in an example. The same figure also illustrates another issue, mirroring inheritance of $c_{IR}$ objects the IR in the respective $c_{IRAlg}$ objects. An example is the hierarchy of all expression vertices, which can be constants, intrinsics, and the like; see Figure 16.

![Diagram of inheritance structure](image)

**Figure 16:** Mirroring inheritance

### 3.6 Algorithm Inheritance and Unparsing

As algorithms replace certain portions of the IR, the unparsing has to refer to an algorithm-specific version. Therefore, each $c_{IR}$ in IR implements `printXMLHierarchy` according to the pattern in the following example.

```cpp
void
Assignment::printXMLHierarchy(std::ostream& os) const {
  if (myAssignmentAlgBase.p)
    getAssignmentAlgBase().printXMLHierarchy(os);
  else
    printXMLHierarchyImpl(os);
}
```

The template implementation for all $c_{IRAlgBase}$ simply refers to the code that contains the unparse logic in $c_{IR}::printXMLHierarchyImpl$.

The reimplementation in $A_{rel}::c_{IRAlg}::printXMLHierarchy` typically refers to the algorithm-specific entity that represents the transformation, as in the following example.

```cpp
void
AssignmentAlg::printXMLHierarchy(std::ostream& os) const {
  if (mySSAReplacementAssignmentList.size()) {
    for (std::list<Assignment>::const_iterator li=mySSAReplacementAssignmentList.begin();
      li!=mySSAReplacementAssignmentList.end();
      ++li)
      (*li)->printXMLHierarchy(os);
  // ... etc.
  }
}```
Note: Any \( c_{IR}^{Alg} \) representing a transformation of its corresponding \( c_{IR} \) instance needs to reimplement \( \text{printXMLHierarchy} \).

An issue with algorithm inheritance and unparsing arises when one wants to unpars multiple representations of the same IR element but at different levels of the algorithm inheritance hierarchy. In this case one has to break the virtual method invocation, which would always use the leaf class implementation, and code a scheme with invocations via explicit function pointers instead. A practical example can be found in the implementation described in Section 2.2.6.

3.7 Algorithm Object Interactions

The traversal mechanism introduced in Section 3.2 strictly follows top down or bottom up in the IR hierarchy bound to the virtual methods in \( \text{GTI} \). Any algorithm code requiring access from \( c_{IR}^{A}^{Alg} \) to \( c_{IR}^{B}^{Alg} \) uses the back references from \( c_{IR}^{A}^{Alg} \) to \( c_{IR}^{A} \) via \( \text{getContaining()} \). With few exceptions, the respective \( \text{getContaining()} \) methods return \( \text{const} \) references. This is intended to emphasize the rule that no algorithm should modify the objects in the original IR but only data encapsulated in the algorithm objects. The objective is to lessen the dependencies between algorithm implementations. Once we have the \( c_{IR}^{A} \) reference, we traverse the IR hierarchy explicitly to \( c_{IR}^{B} \), obtain the algorithm base class handle to \( c_{IR}^{B}^{AlgBase} \) via \( \text{getc}_{IR}^{B}^{AlgBase}() \), and cast to the proper subclass \( A_{\text{alg}}::c_{IR}^{B}^{Alg} \). For most \( c_{IR} \) objects the top-down hierarchy provides explicit traversal top down only. Back references to the containing elements are the exception. As a logical consequence, transformations on a hierarchy level resulting in the creation or deletion of IR elements at the same level need to be rooted at the higher level of the containing element. For example, an algorithm seeking to remove certain vertices from an expression graph based on a vertex specific activity flag needs to be rooted at the containing expression.

The generic traversal mechanism itself does not support the passing of data between \( c_{IR}^{Alg} \) instances. If different \( c_{IR}^{Alg} \) instances have to communicate across different \( \text{algorithm\_action} [1,2,...] \), one can use established programming patterns [4] such as external (thread safe) containers to hold the data.

3.8 Linearization as Case Study

In Sections 1 and 2.2.1 we explained the purpose of the linearization of a given code. In the following we use linearization as a case study for an algorithm implementation. First we define the set of \( c_{IR} \) that will be subject to the linearization transformation.

- We attach partial derivatives to the edges in expression graphs representing right-hand sides of assignments \( \rightarrow \) modifies \( \text{ExpressionEdge} \).

- We analyze right-hand-side expressions to identify passive subexpressions by attaching an activity flag to expression vertices. The analyzing methods operate on expressions. \( \rightarrow \) modifies \( \text{ExpressionVertex} \) and all its subclasses and implements analysis on \( \text{Expression} \).
• We create a copy of the right-hand-side expression and remove the passive subexpressions. The copy is attached to the algorithm object associated with the assignment in question. \( \rightarrow \) modifies Assignment.

This describes the set of the \( c_{IR Alg} \) that need to be implemented; see also Figure 15.

### 3.8.1 Expression Edge

The ExpressionEdgeAlg class has four additional data members:

• A reference to the formal expression for the partial derivative as provided by the intrinsics catalogue; see Section 2.2 (myPartialDerivative_p)

• A list mapping formal arguments in the expression for the partial derivative to actual arguments in this expression (myConcreteArgumentInstancesList)

• The assignment that has the concrete expression for the partial derivative as the right-hand side (myConcretePartialAssignment_p)

• A flag indicating the category of the partial ranging from NONLINEAR to PASSIVE (myConcretePartialDerivativeKind)

Aside from get/set methods, this class contains no additional logic pertaining to the algorithm.

### 3.8.2 Expression Vertices

We need a common algorithm base class for all expression vertices (ExpressionVertexAlg).

• Activity analysis inside a right-hand side requires a flag on each vertex (myActiveFlag).

• The computation of the partials may require some intermediate values as inputs, and therefore those intermediate values need to be assigned to temporary variables (myAuxiliaryVariable_p). While this applies only to Intrinsic vertices, the transformation output is generated via the base class.

• We eventually need to build the entire assignment of which myAuxiliaryVariable_p is the left-hand side of myReplacementAssignment_p.

In ExpressionVertexAlg we encapsulate the allocation/deallocation of the last two data members. All derived algorithm classes differ only in their initialization with respect to the activity flag.

### 3.8.3 Expression

In ExpressionAlg we encapsulate the following.
• We track the usage of Argument vertices in partial derivative computations in a list called myPartialUsageList. This is used to determine exactly when we need a delay in assigning the left-hand side of the parent assignment in cases where the left-hand side aliases any argument in the right-hand side

• We create expressions for the partial derivative computation in a method called createPartialExpressions(). This happens in conjunction with tracking the argument use and determining which intermediates need to be saved in a two-pass fashion through the expression graph after activity analysis is performed. The activity analysis marks passive subgraphs that we can ignore. The first pass determines which intermediates have to be saved for the partial computation based on the argument usage information in the InlinableIntrinsicsCatalogue. The second pass generates a concrete expression from the formal expression for the respective partial derivative by replacing the formal arguments with the concrete arguments as determined in the previous pass and assigning the result to another temporary.

• The activity analysis is implemented as a bottom-up pass through the expression graph in activityAnalysis().

3.8.4 Assignment

AssignmentAlg is the entry point for the linearization algorithm. This class illustrates a few issues with the algorithm implementation. From the previous it is apparent that new assignments are created, and therefore one might expect that the entry point should be the basic block according to Section 3.7. However, all new assignments being created have a unique parent assignment in the original IR. This association allows lowering the entry point to the assignment level. For the sake of reuse in other algorithms, the linearization is split in two phases.

1. algorithm_action_1(): First we copy the original right-hand side into the data member myLinearizedRightHandSide. Given the implementation of algorithm objects for the entire right-hand side, we would not have to make this copy if we were to stop with linearization. Obviously the linearization is only a building block for other algorithms. These reusing algorithms have to change the structure of the right-hand side, and therefore the linearization has to be performed on a copy because as a rule we do not modify the original IR; see Section 3.7. On the copy we run the activity analysis provided by ExpressionAlg. This requires the copy to be created with the associated algorithm objects. If the entire right-hand side is determined to be passive, then the assignment’s left-hand side is marked passive as well. Note that most compiler-style analysis will simply propagate activity based on the activity of the arguments. Here we allow in particular for passivating intrinsics.

2. algorithm_action_2(): This method iterates through the right-hand side and creates the respective replacement assignments associated with the expression vertices and the partial derivative assignments associated with the expression edges. The creation of the replacement assignments is somewhat complex as the algorithm has to determine subgraphs in the expression that have a given maximal node and has minimal nodes
that are determined by the maximal nodes of preceding subgraphs. This is done by
nested recursions; see localRHSExtractionOuter and localRHSExtractionInner

The unparsing of the transformed representation is done in printXMLHierarchy, which refers
to myLinearizedRightHandSide. It first unparses the replacement assignments for expression
edges, followed by the assignments that compute the partial derivatives, potentially followed
by the delayed assignment to the right-hand side in case of aliasing to the left-hand-side
arguments.

3.8.5 Invocation

The factory manager and factory classes are written following the patterns shown in Sec-
tions 3.3 and 3.4. The linearization can be run as illustrated in the associated test example.
The factory manager is instantiated and initialized with

taxifBoosterLinearization::AlgFactoryManager::instance()->init();
The parsing is done as explained in Section 2.3. The algorithm invocation is simply two
passes from the call graph object.

```
CallGraph& Cg(ConceptuallyStaticInstances::instance()->getCallGraph());
Cg.genericTraversal(GenericAction::ALGORITHM_ACTION_1); // analyze
Cg.genericTraversal(GenericAction::ALGORITHM_ACTION_2); // generate code
```

This is followed by the unparsing, as explained in Section 2.3.

4 Conclusion

Implementing an entire program’s IR, including parsing and unparsing, is a considerable
effort that poses a steep hurdle to anybody interested in manipulating low-level numerical
structures. With OpenAD we remove this obstacle and provide the possibility for a com-
paratively quick implementation of source transformation algorithms. It is not intended
as a replacement for fully compiler-integrated AD that can use the results of algorithmic
research done in OpenAD but would, as a matter of commercial product development,
ensure complete language coverage and the like. Instead, OpenAD is geared toward an ac-
dademic audience with an emphasis on algorithmic experimentation and handling of specific
computationally challenging applications that require finely tuned advanced algorithms.

In presenting the implemented algorithms and the linearization as a use case, we show
that OpenAD fulfills the needs of a test bed for experimentation. The application to the
MIT general circulation model is subject to further algorithm development.

References

puting Jacobian matrices efficiently. In Proceedings of International Conference on
Computational Science, volume 2658 of Lecture Notes in Computer Science, pages 575


