Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 99–111
www.elsevier.com/locate/entcs

Term Graphs for Computing Derivatives in Imperative Languages

Paul D. Hovland[1] Boyana Norris[2]

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave, Argonne, IL 60439, USA

Michelle Mills Strout[3]

Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA

Jean Utke[4]

Department of Computer Science
University of Chicago
Chicago, IL 60637, USA

Abstract

Automatic differentiation is a technique for the rule-based transformation of a subprogram that computes some mathematical function into a subprogram that computes the derivatives of that function. Automatic differentiation algorithms are typically expressed as operating on a weighted term graph called a linearized computational graph. Constructing this weighted term graph for imperative programming languages such as C/C++ and Fortran introduces several challenges. Alias and definition-use information is needed to construct term graphs for individual statements and then combine them into one graph for a collection of statements. Furthermore, the resulting weighted term graph must be represented in a language-independent fashion to enable the use of AD algorithms in tools for various languages. We describe the construction and representation of weighted term graphs for C/C++ and Fortran, as implemented in the ADIC 2.0 and OpenAD/F tools for automatic differentiation.

Keywords: automatic differentiation, computational graph, term graph

[1] Email: hovland@mcs.anl.gov
[2] Email: norris@mcs.anl.gov
[3] Email: mstrout@cs.colostate.edu
[4] Email: utke@mcs.anl.gov

doi:10.1016/j.entcs.2006.10.029

1 Introduction

Automatic differentiation is a technique for the rule-based transformation of a subprogram that computes some mathematical function into a subprogram that computes the derivatives of that function [4, 5, 3]. Derivatives have a variety of uses in scientific computing, including the solution of nonlinear partial differential equations, function minimization, parameter identification, data assimilation, sensitivity analysis, and uncertainty quantification. Automatic differentiation algorithms typically operate on a directed acyclic graph referred to as a *computational graph* or, after edge weights corresponding to partial derivatives have been added, a *linearized computational graph.*

The computational graph represents each value in a computation as a vertex and represents value dependences between values as directed edges. Formally, the computation graph is a graph $G(V, E)$, where V is the set of vertices and E is the set of edges. Each vertex represents a value $\mathrm{val}(v)$ and is labeled with an ordered pair $< \mathrm{op}, \mathrm{var} >$ corresponding to the operation that generates the value (null in the case of leaf vertices) and the variable that contains the value (null in the case of anonymous intermediate values). For example, Figure 2 shows the computational graph for the simple function defined by the program segment in Figure 1. This computational graph is essentially an acyclic term graph [16], with the orientation of the directed edges reversed from the usual convention.

```
a = cos(x);         // statement 1
b = sin(y)*y*y;     // statement 2
f = exp(a*b);       // statement 3
```

Fig. 1. Pseudocode for a simple example.

The computational graph is transformed into a linearized computational graph by adding edge weights that correspond to partial derivatives. Formally, each edge e_{ij} is associated with a partial derivative $p(e_{ij}) = \partial \mathrm{val}(v_i)/\partial \mathrm{val}(v_j)$. For example in Figure 2, the partial derivative of `cos(x)` with respect to `x` is `-sin(x)`. The linearized computational graph can be interpreted as a *weighted, acyclic term graph.* Figure 3 shows the linearized computational graph for the simple example. Given this linearized computational graph, the derivative of a dependent variable v_j with

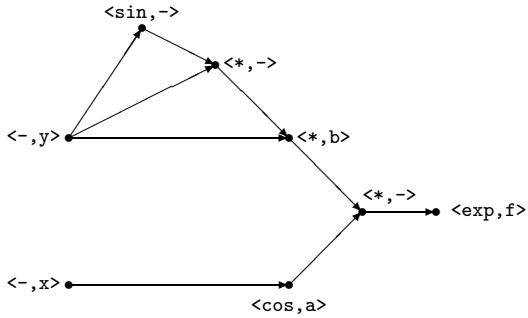

Fig. 2. Computational graph for the simple example.

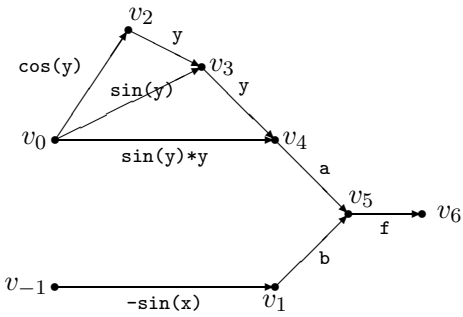

Fig. 3. Linearized computational graph for the simple example.

respect to an independent variable v_i is the sum over all paths from v_i to v_j of the product of the edge weights along that path [2].

Equivalently, the linearized computational graph can be transformed via a sequence of vertex or edge eliminations (rewrites) into a bipartite graph whose edge weights correspond to these derivatives. A vertex is eliminated by multiplying the weight of each input edge by that of each output edge and adding the product to the edge whose source is that of the input edge and whose sink is that of the output edge, creating new edges from predecessor vertices to successor vertices where necessary. More formally, a vertex v_k is eliminated using the rule: $\forall v_i \in \text{Pred}(v_k), v_j \in \text{Succ}(v_k)$, if $e_{ij} \in E$ then $p(e_{ij}) = p(e_{ij}) + p(e_{ik})p(e_{kj})$ else $E = E \cup e_{ij}$ and $p(e_{ij}) = p(e_{ik})p(e_{kj})$, where E is the set of all edges, e_{ij} denotes the edge from vertex v_i to vertex v_j, $p(e)$ is the weight of edge e, $\text{Pred}(v_i)$ denotes the set of all predecessors to vertex v_i, and $\text{Succ}(v_i)$ denotes the set of all successors to vertex v_i. Figure 4 shows the linearized computational graph after eliminating the vertices in the sequence v_5, v_2, v_3, v_4, v_1. Figure 5 shows pseudocode corresponding to this elimination order. The associativity of the chain rule of differential calculus implies that vertices may be eliminated in any order. Because of fill-in, the elimination order impacts the computational cost. For example, while the cost of the given elimination order is six multiplications and two additions, the cost of the elimination order v_5, v_4, v_3, v_2, v_1 is nine multiplications and two additions. Finding an order that minimizes the number of multiplications is conjectured to be NP-hard [13, 10]. Many heuristics are used, however, including topological order (called the forward mode), reverse topological order (called the reverse mode), minimum Markowitz degree[5] [7], and relative Markowitz degree [5, 11]. The number of multiplications can be further reduced by eliminating individual edges [11] or pairs of edges [12], rather than entire vertices, but such techniques are beyond the scope of this paper. We note that for all types of elimination, the rewrite system is confluent and always terminates in a unique bipartite graph. It is easy to see this for vertex elimination: after each elimination step,

(i) the number of vertices is reduced by one and

(ii) the sum over all paths from v_i to v_j of the product of the edge weights along

[5] The Markowitz degree is the product of the in degree and the out degree.

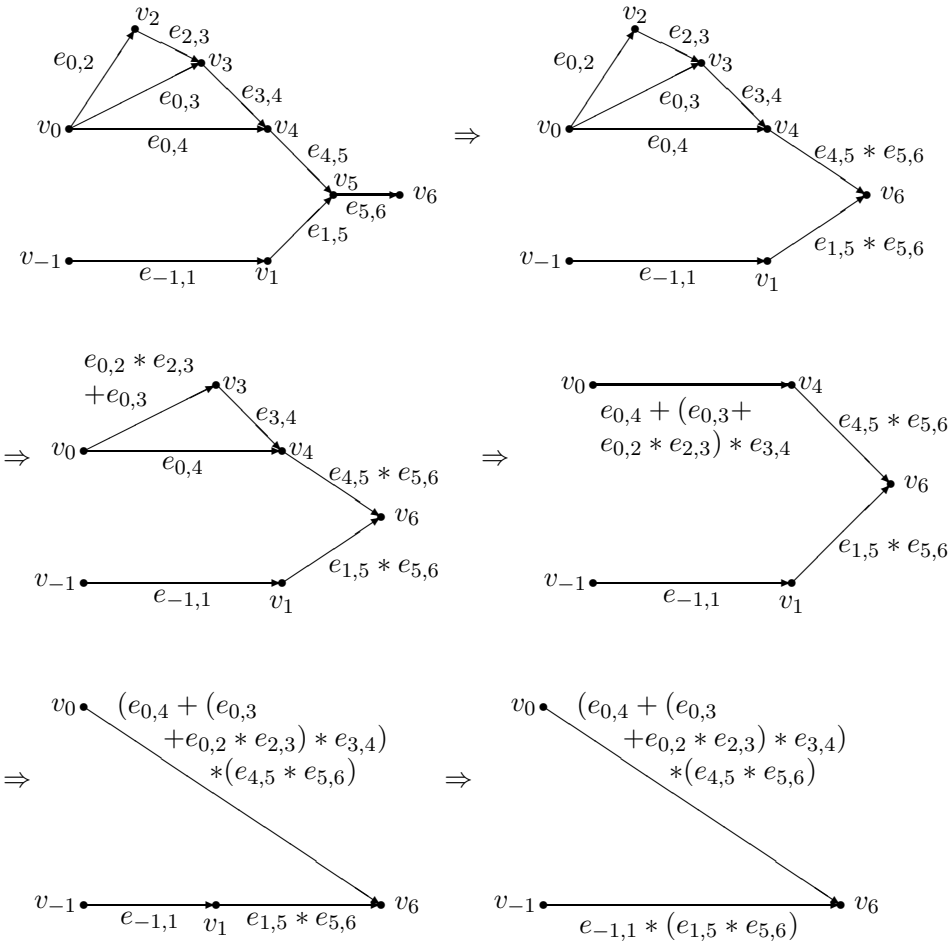

Fig. 4. Linearized computational graph of Figure 3, following the vertex elimination sequence v_5, v_2, v_3, v_4, v_1.

the path (the Baur-Strassen derivative [2]) remains unchanged.

A more detailed termination proof, including the cases of edge and face elimination, can be found in [12].

This paper represents a reinterpretation of existing work together with an exposition of implementation details relevant to the use of term graphs for imperative programming languages. The rest of the paper is organized as follows. The next section sketches the construction of weighted term graphs for imperative programming languages. Section 3 describes some of the static analyses used in automatic differentiation tools. Section 4 discusses a procedure for merging the term graphs for individual statements into larger term graphs. Section 5 describes an XML representation for term graphs. In Section 6, we speculate on how ideas from term graphs could benefit the automatic differentiation community (and vice versa). We conclude with a brief summary.

```
a = cos(x);         // statement 1
temp_v2 = sin(y);
temp_v3 = temp_v2*y;
b = temp_v3*y;      // statement 2
f = exp(a*b);       // statement 3

ex1 = -sin(x);      // da/dx
ey2 = cos(y);       // dv2/dy
ey3 = temp_v2;      // dv3/dy
e23 = y;            // dv3/dv2
ey4 = temp_v3;      // dv4/dy
e34 = y;            // dv4/dv3
e45 = a;            // d(a*b)/db
e15 = b;            // d(a*b)/da
e56 = f;            // df/d(a*b)

e46 = e45*e56;      // eliminate vertex v5
e16 = e15*e56;      // eliminate vertex v5
ey3 += ey2*e23;     // eliminate vertex v2
ey4 += ey3*e34;     // eliminate vertex v3
ey6 = ey4*e46;      // eliminate vertex v4
ex6 = ex1*e16;      // eliminate vertex v1

dfdx = ex6
dfdy = ey6
```

Fig. 5. Pseudocode for derivative code, using vertex elimination order v_5, v_2, v_3, v_4, v_1. Temporary variables have been introduced for anonymous intermediate values needed in the derivative computation. Trivial assignments are included for clarity.

2 Weighted Term Graphs for Imperative Programming Languages

Automatic differentiation is used primarily in the domain of scientific computing, where the vast majority of programs is implemented in an imperative programming language such as C/C++ or Fortran. Because automatic differentiation algorithms operate on linearized computational graphs, mechanisms are needed for the construction of these weighted term graphs from programs written in imperative languages. We briefly describe two strategies, one suitable for the runtime construction of a weighted term graph and one suitable for compile-time construction, which requires static analysis and transformation of source code. The remainder of this paper discusses the second strategy in greater detail.

2.1 Runtime Construction of Weighted Term Graphs

In programming languages that support operator overloading, including C++, one can construct a term graph for a particular execution history by overloading the

operation	cos	sin	*	*	*	exp
input value 0	1.00	2.00	0.91	1.82	0.54	1.97
input value 1	–	–	2.00	2.00	3.64	–
output value	0.54	0.91	1.82	3.64	1.97	7.17
input address 0	0	1	5	6	2	5
input address 1	–	–	1	1	3	–
output address	2	5	6	3	5	4

Fig. 6. A possible tape for the simple example of Figure 1.

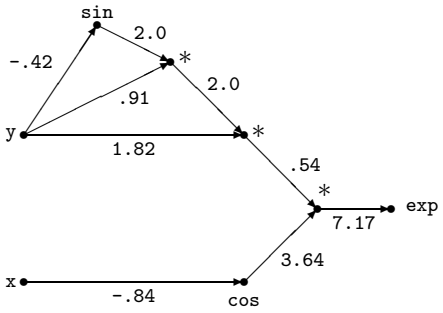

Fig. 7. Linearized computational graph constructed from the tape of Figure 6.

operators and intrinsic functions to record ("tape") the operation type as well as the values and addresses of the input and output arguments. When subprogram execution completes, the computation graph is constructed from this "tape," using each definition of an address as a vertex in the computational graph. The vertex identifier for any input address can be obtained by searching for the latest definition of that address in the tape. Given the operation type and input values, partial derivatives can be computed and assigned as edge weights to create a linearized computational graph. Figure 6 shows one possible tape for our simple example, using input values `x=1.0` and `y=2.0`. Figure 7 shows the weighted term graph constructed from this tape. Because the graph structure is not known until runtime, the vertex elimination order must be determined online, necessitating the use of simple, linear time heuristics. The ADOL-C automatic differentiation tool [6] uses runtime taping to construct weighted term graphs for C++ programs.

2.2 Static Construction of Weighted Term Graphs

One can also construct the term graph using static analysis of the source code. In this case, the edge weights are not known until runtime, but the structure of the term graph is known at compile time. Since the analysis is performed offline, polynomial time algorithms can be used to select a vertex elimination order. Global search algorithms such as simulated annealing are also tractable [14]. Static source transformation offers several other opportunities for efficiency improvements in automatic differentiation. However, it also presents many challenges: a robust compiler infrastructure is needed for parsing and unparsing; many types of static

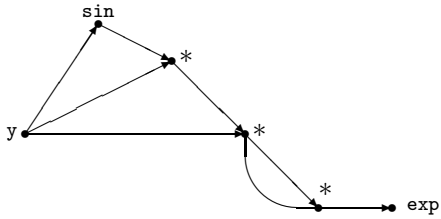

Fig. 8. Computational graph when a is aliased to b.

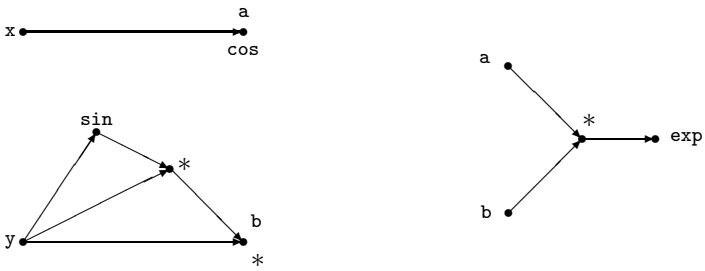

Fig. 9. Term graphs for individual statements.

analysis are required; and a mechanism for handling control structures such as loops and branches is essential.

For programs with complex control flow, static construction of the complete and correct term graph is not possible. Instead, one typically constructs a term graph for each basic block (a sequence of statements with no intervening control constructs) and applies arbitrary vertex elimination strategies only within individual basic blocks. The derivatives of basic blocks are combined using either the forward mode or reverse mode. The forward mode requires no additional runtime information, since the derivative computation follows the same control flow as the original function evaluation. The reverse mode must reverse the control flow. Therefore, at runtime a record of control flow decisions (such as basic block identifiers or branch conditions and loop bounds) must be stored.

Even constructing a single term graph for each basic block may not be possible. Consider the simple example of Figure 1. If a is aliased to b, then the computational graph is the term graph shown in Figure 8. If one cannot statically determine whether a and b are aliased, separate term graphs are needed for each statement, as depicted in Figure 9. In practice, separate term graphs are used by default and these separate graphs are merged into larger term graphs only when static analysis guarantees correctness. This process, called flattening, is described in more detail in Section 4.

3 Representation-Independent Static Analysis

Implementation of efficient automatic differentiation tools requires various types of static analysis. Rather than implement these analyses twice (once for the Open64/SL infrastructure used by OpenAD/F and again for the ROSE/Sage infrastructure used by ADIC 2.0), we have implemented them within the OpenAnalysis framework [17]. OpenAnalysis seeks to decouple compiler analyses from specific intermediate representations by introducing analysis-specific interfaces. This facilitates the use of multiple analysis algorithms with a single compiler infrastructure as well as the use of a single analysis implementation with multiple compiler infrastructures. OpenAnalysis provides algorithms for call graph construction, control flow graph construction, alias analysis, and interprocedural data-flow analysis. We have implemented OpenAnalysis interfaces for the Open64/SL (for Fortran) and ROSE/Sage (for C/C++) infrastructures.

One of the most important analyses implemented in OpenAnalysis is alias analysis. Alias analysis is used to identify whether two inputs to a statement should share a vertex or be treated as separate variables. In addition, alias analysis is needed for other static analyses, including data-flow analyses such as reaching definitions. Reaching definitions is a static analysis used to determine which definitions of a variable can possibly reach a particular use of a variable. It can be used to construct the du- and ud-chains used in the flattening procedure described in Section 4.

While the precise flow of information cannot always be determined statically, one can often determine that certain variables, no matter what control path is taken, will never lie along the paths between the independent variables and the dependent variables of interest. Such variables are called passive and do not need to have their derivatives computed. Thus, these variables can be ignored in the construction of term graphs. The OpenAnalysis infrastructure implements an interprocedural data-flow analysis called activity analysis to identify the set of passive variables.

4 Merging Term Graphs by Flattening

In Section 1 we mentioned the possibility of minimizing the cost of computing derivatives using automatic differentiation by searching for an optimal elimination order in the term graphs. For many existing tools the default scope for constructing term graphs is the assignment statement as explained in Section 2.2. It is clear that this limits the improvements one can gain from optimizing the elimination order. Consequently we prefer to construct term graphs that cover a larger scope. On the other hand, due to the complexity of the optimization problem, we must avoid graphs that grow proportionally with the run time of the program as done in Section 2.1. The unrolling of loop bodies in this fashion is a good example of bloating the term graph with repetitive structures that should be avoided. Furthermore, if the control flow contains branches, a unified term graph for these branches requires a transformation that makes the computations in the branches mutually independent.

Therefore, we consider consecutive sequences of assignment statements within

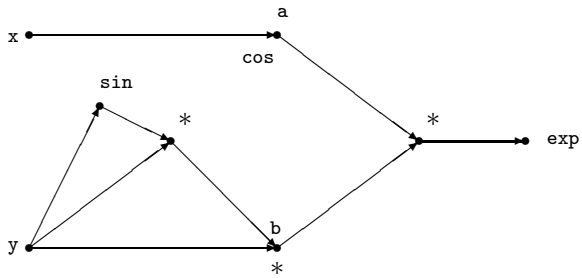

Fig. 10. Merging vertices `a` and `b`

basic blocks to be reasonable scopes for constructing term graphs. The example in Section 2.2 illustrates the principal problem that arises due to aliasing. The most familiar form of aliasing occurs with arrays when for example we consider `v[i]` and `v[j]` and we cannot tell at compile time if `i` and `j` will always or never be the same, i.e. `v[i]` and `v[j]` refer to the same address in memory. We can use refined code analyses for the purpose of constructing semantically correct term graphs in the presence of aliasing. The so called *use/define-* or *ud-chains* are a suitable representation for the combined results of alias and dependency analysis. In essence, each use of a variable in the code is associated with a ud-chain that contains a location list of possible definitions. We start out with the term graphs of the individual assignment statements. The left-hand side defined in the assignment is represented by the maximal vertex (we assume side-effect-free expressions) in the term graph. We iterate through all statements in execution order. For each statement we consider each variable use in the right-hand side. If the ud-chain associated with that use contains exactly one element then we can merge the vertex representing the use with the vertex representing the definition. If there is no definition within the scope of the merging process than we retain the vertex as is. If none of the variables from the example in Figure 1 are aliased then the ud-chain for the use of `a` in statement 3 will contain only statement 1 as the definition point. Similarly, the ud-chain for the use of `b` in statement 3 will point only to statement 2 as the definition point and we merge the respective vertices as shown in Figure 10. If `a` is aliased to `b` then their respective ud-chains both point to `b` defined by statement 2 and we obtain the graph as shown in Figure 8.

We already mentioned that quite often the alias analysis does not yield such clear cut results. If we replace `b` in statement 2 by `b[i]` and in statement 3 by `b[j]` then in many cases the ud-chain for the use of `b[j]` in statement 3 will not only point to statement 2 but also to some preceding statement 0, e.g. `b[k]=2*x`, as a possible point of definition. While the `a` vertices might still be merged the proper definition point of `b[j]` will only be known at run time. The easiest (but not the only) way to enforce semantical correctness is to organize the merging such that a given statement term graph can either be merged completely or in the case of ambiguous defines a new merge is started. This results in a sequence of merged graphs. Semantical correctness is ensured if the derivative accumulation is executed in the order implied by the statement subsequences that make up the merged graphs.

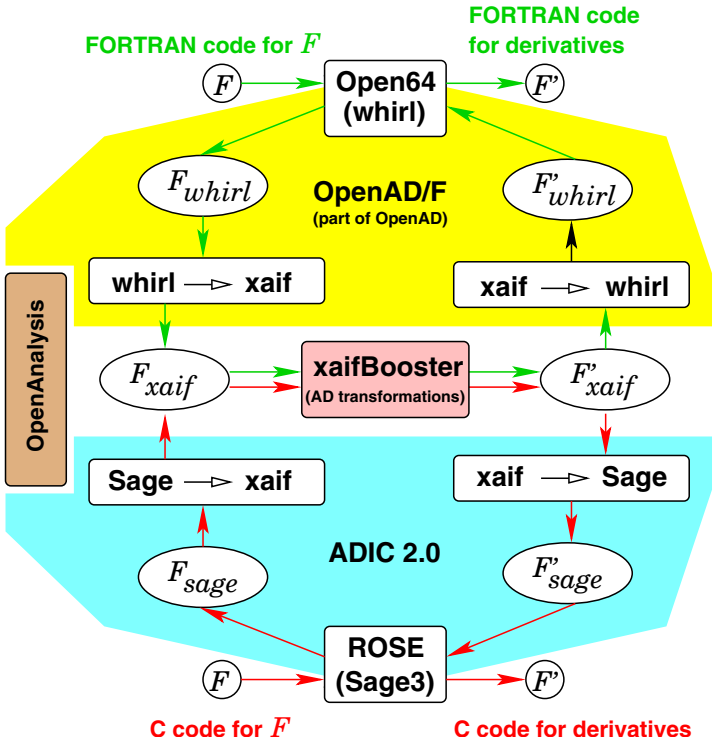

Fig. 11. Schematic of the OpenAD architecture.

In practical applications we observe graph sizes with a few hundred vertices which is reasonable for the elimination heuristics we employ.

5 An XML Schema for Term Graphs

To facilitate software reuse, the ADIC 2.0 [15] and OpenAD/F [18] automatic differentiation tools are constructed in a modular fashion, as depicted in Figure 11. Language-specific frontends communicate with a differentiation module using an XML representation of the mathematically-relevant elements of a program. The XML Abstract Interface Form (XAIF) [8] provides a language-independent representation of constructs common in imperative languages, such as C, C++, and Fortran. The program is represented as a sequence of nested graphs: a call graph that contains a scope tree and one or more control flow graphs, whose vertices are basic blocks. Each basic block contains assignment statements whose right-hand sides are expression term graphs. At the assignment statement level, imperative languages are not very different from other types of languages, making the XAIF useful for representing term graphs for expressions in non-imperative languages. The XAIF schemas are also designed with extensibility in mind, allowing easy customization of the contents of graph, vertex, and edge elements.

Figure 12 shows an XAIF fragment describing the computational graph for the second statement in the simple example in Figure 1. Each assignment statement consists of a `AssignmentLHS` and `AssignmentRHS` elements. The expression graph

```
<xaif:Assignment statement_id="139561992">
  <xaif:AssignmentLHS>
   <xaif:SymbolReference annotation="139557224" scope_id="4"
                         symbol_id="b" vertex_id="1"/>
   </xaif:SymbolReference>
  </xaif:AssignmentLHS>
  <xaif:AssignmentRHS>
    <xaif:VariableReference vertex_id="1">
      <xaif:SymbolReference annotation="139554184" scope_id="3"
                            symbol_id="y" vertex_id="1"/>
    </xaif:VariableReference>
    <xaif:Intrinsic name="sin_scal" vertex_id="2"/>
    <xaif:Intrinsic name="mul_scal_scal" vertex_id="3"/>
    <xaif:Intrinsic name="mul_scal_scal" vertex_id="4"/>
    <xaif:VariableReference vertex_id="2">
      <xaif:SymbolReference annotation="139554184" scope_id="3"
                            symbol_id="y" vertex_id="1"/>
    </xaif:VariableReference>
    <xaif:ExpressionEdge edge_id="1" position="1" source="1" target="2"/>
    <xaif:ExpressionEdge edge_id="2" position="1" source="2" target="3"/>
    <xaif:ExpressionEdge edge_id="3" position="2" source="1" target="3"/>
    <xaif:ExpressionEdge edge_id="4" position="1" source="3" target="4"/>
    <xaif:ExpressionEdge edge_id="5" position="2" source="1" target="4"/>
  </xaif:AssignmentRHS>
</xaif:Assignment>
```

Fig. 12. XAIF representation of the statement `b=sin(y)*y*y`.

in the `AssignmentRHS` element can contain vertices corresponding to variable references, constants, binary, and unary operators, as illustrated in Figure 9 (excluding the edge weights). In the automatic differentiation context, first the XAIF representation of a program is generated by a language-specific frontend, then the XAIF is transformed by a language-independent differentiation module, and finally the resulting new XAIF is parsed by the language-specific backend and merged with the original language-specific AST representation.

In addition to acyclic term graphs for expressions, the XAIF representation includes elements for expressing scope hierarchies as trees whose vertices are the symbol tables for each scope. Each symbol reference vertex contains `scope_id` and `symbol_id` attributes, which refer to the scope and symbol element definitions contained in the scope hierarchy. This provides the connection between the abstract expression term graph representation and the actual program elements.

6 Future Directions

There are several possible advantages to interpreting automatic differentiation as a rewrite system for weighted term graphs. First, although linearized computational graphs are always acyclic, the introduction of cycles, following the example of cyclic term graphs, might facilitate the development of more sophisticated differentiation algorithms for code with loops and/or recursion. Currently, automatic differentiation tools use either the forward mode or reverse mode at scopes larger than basic blocks. It also seems likely that the transformation of a computational graph into a linearized computational graph can be recast as a graph rewrite system. Finally, term graphs are the most natural way to express and reason about automatic differentiation of functional programming languages.

Conversely, it appears that the automatic differentiation community may be able to contribute technologies to the term graph rewriting community. While the

latter community focuses on functional and declarative programming languages, automatic differentiation tools typically target imperative and object-oriented languages. It may be possible to use much of the existing infrastructure to implement term graph rewrite techniques, thus making them available to a broader user community. Furthermore, the XAIF, used to represent computational graphs in XML, seems well-suited, with minimal modifications, for the representation of generic terms graphs. Having a portable, standard representation for term graphs would facilitate the development of common infrastructure, such as tools for the graphical display of term graphs. To our knowledge, no such standard representation exists, nor are there alternative XML representations from automatic differentiation that could be readily adapted.

7 Conclusions

We have presented automatic differentiation as a rewrite system for weighted term graphs. The automatic differentiation rewrite rules guarantee termination at a unique, bipartite graph. We believe that this interpretation may lead to new algorithms for recursive functions and support the theoretical analysis of automatic differentiation algorithms. We have described an infrastructure for the static construction of term graphs from imperative programming languages and an XML representation for term graphs.

While we have focused on static techniques in this paper, our automatic differentiation tools often employ hybrid static-dynamic techniques. For example, the determination of whether a variable is active or passive can be deferred until runtime, at least for those variables where static analysis is inconclusive [9]. Future work will examine similar techniques for the situation where static alias analysis is ambiguous.

There is evidence to suggest that rewrite rules that introduce new vertices into a linearized computational graph may reduce the cost of computing derivatives. However, adding such rules to the rewrite system removes the guarantee of termination. Because these new rules are a modified form of copying, we believe that the work of Ariola et al. [1] on bisimilarity in term graph rewriting may provide insight into efficient ways of computing derivatives under such a system. We also note that the NP-hardness proof of [13] relies on the addition of collapsing operations to the differentiation rewrite rules.

Although we have speculated on some ways that the automatic differentiation and term graph rewriting research communities can learn from one another, it seems likely that many other opportunities for technology transfer remain to be discovered.

8 Acknowledgments

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. Dept. of Energy under Contract DE-AC02-06CH11357

and by the National Science Foundation under Grant No. OCE-020559. Uwe Naumann contributed to the design and implementation of the XAIF.

References

[1] Ariola, Z. M., J. W. Klop and D. Plump, *Bisimilarity in term graph rewriting*, Inf. Comput. **156** (2000), pp. 2–24.

[2] Baur, W. and V. Strassen, *The complexity of partial derivatives*, Theoretical Computer Science **22** (1983), pp. 317–330.

[3] Bischof, C. H., P. D. Hovland and B. Norris, *Implementation of automatic differentiation tools*, Higher-Order and Symbolic Computation (2006), to appear.

[4] Griewank, A., *On automatic differentiation*, in: M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publishers, Dordrecht, 1989 pp. 83–108.

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

[6] Griewank, A., D. Juedes and J. Utke, *ADOL–C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software **22** (1996), pp. 131–167.

[7] Griewank, A. and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in: A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, PA, 1991 pp. 126–135.

[8] Hovland, P. D., U. Naumann and B. Norris, *An XML-based platform for semantic transformation of numerical programs*, in: M. Hamza, editor, *Software Engineering and Applications* (2002), pp. 530–538.

[9] Kreaseck, B., L. Ramos, S. Easterday, M. Strout and P. Hovland, *Hybrid static/dynamic activity analysis*, in: V. Alexandrov, G. van Albada, P. Sloot and J. Dongarra, editors, *Computational Science - ICCS 2006, Proceedings of the Sixth International Conference on Computational Science, Reading, UK, May 28-31, 2006, Part IV*, Lecture Notes in Computer Science **3994** (2006), pp. 582–590, also as ANL preprint ANL/MCS-P1226-0205.

[10] Naumann, U., *Min-ops derivative computation is NP-complete*, Presentation at 2nd European Workshop on Automatic Differentiation.

[11] Naumann, U., "Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs," Ph.D. thesis, Technical University of Dresden (1999).

[12] Naumann, U., *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph*, Math. Program. **99** (2004), pp. 399–421.

[13] Naumann, U., *The complexity of derivative computation*, Technical Report AIB-2005-15, RWTH Aachen (2005).

[14] Naumann, U. and P. Gottschling, *Simulated annealing for optimal pivot selection in Jacobian accumulation*, in: A. Albrecht and K. Steinhöfel, editors, *Stochastic Algorithms: Foundations and Applications*, Lecture Notes in Computer Science **2827** (2003), pp. 83–97.

[15] Norris, B. and S. Melfi, *ADIC Web Server* (2000), http://www.mcs.anl.gov/adicserver.

[16] Plump, D., *Term graph rewriting*, Technical Report CSI-R9822, Computing Science Institute, Catholic University of Nijmegen (1998).

[17] Strout, M. M., J. Mellor-Crummey and P. Hovland, *Representation-independent program analysis*, in: *Proceedings of the Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2005, pp. 67–74.

[18] Utke, J., *OpenAD web site*, http://www.mcs.anl.gov/openad.