ADIC: An Extensible Automatic Differentiation Tool for ANSI-C

Christian Bischof and Lucas Roh
Andrew Mauer

CRPC-TR97676-S
January 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Also available as Argonne Preprint ANL/MCS-P626-1196

ADIC: An Extensible Automatic Differentiation Tool for ANSI-C*

Christian Bischof and Lucas Roh

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA
{bischof,roh}@mcs.anl.gov

Andrew Mauer†

Department of Mathematics
University of Illinois at Urbana-Champaign
Urbana, IL 61802
mauer@math.uiuc.edu

Argonne Preprint ANL/MCS-P626-1196

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Orders L25935D and L64948D, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†The work of this author was performed while he was a student research associate with Argonne National Laboratory.

Abstract. In scientific computing, we often require the derivatives $\partial f/\partial x$ of a function f expressed as a program with respect to some input parameter(s)x, say. Automatic differentiation (AD) techniques augment the program with derivative computation by applying the chain rule of calculus to elementary operations in an automated fashion. Due to the associativity of the chain rule, there are many ways for accumulating these derivatives. This article introduces ADIC (Automatic Differentiation of C), a new AD tool for ANSI-C programs. ADIC is currently the only tool for ANSI-C that employs a source-to-source program transformation approach; that is, it takes a C code and produces a new C code that computes the original results as well as the derivatives.

We first present ADIC "by example" to illustrate the functionality and ease of use of ADIC and then describe in detail the architecture of ADIC. ADIC encorporates a modular design that provides a foundation for both rapid prototyping of better AD algorithms and their sharing across AD tools for different languages. A component architecture called AIF (AD Interface Form) separates core AD concepts from their language-specific implementation and allows the development of generic AD modules that can be directly be reused in other AIF-based AD tools. The language-specific ADIC front-end and backend canonicalize C programs to make them fit for semantic augmentation and manage, for example, the association of a program variable with its associated derivative object.

We also report on applications of ADIC to a semiconductor device simulator, 3-D CFD grid generator, vehicle simulator, and neural network code.

Key words. Automatic differentiation, gradient, Jacobian, Hessian, Sage++, compiler, source transformation, AIF, ADIC.

1 Introduction

Given a complex computational model of physical phenomena (such as interconnect properties in semiconductors, air flow around a wing, or chemical reactions in the atmosphere), we are often interested in *sensitivity analysis*, in other words, assessing the impact of changes in model input values on the model outputs. One way to do this systematically is to compute the derivatives of output variables with respect to input variables. If y is an output variable of the model, and x an input variable, the availability of $\partial y/\partial x$ allows us to predict to first order the impact that changes in x will have on y. Thus, derivative information can be used to test the robustness of the model or to adjust, typically with the help of numerical optimization algorithms, certain model parameters so that the model agrees with experimental results (this is typically called *parameter identification* or *data assimilation*). Derivatives are also essential in other areas of nonlinear modeling, for example in nonlinear equation solving and design optimization [22, 1].

In general, given a code C that computes a function $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$ with n inputs and m outputs, we may then require the derivatives of some of the outputs y with respect to some of the inputs x. Thus, we would like to create from C a new code C' that computes $f' = \partial y/\partial x$. Ideally, C' should be accurate and fast and should require little development time.

The derivative code C' can be produced in a number of different ways. It can be developed *by hand*, which typically is laborious and error prone. On the other hand, a skilled user can take advantage of domain-specific knowledge, which can result in very efficient code. C' can also be developed with the help of a symbolic-mathematics package such as Mathematica, Maple, or Reduce. While this approach works well for domain-specific languages (see, e.g., [27, 38]), it is not directly applicable to large computer codes in languages such as Fortran 77 or C. Another alternative, *divided differences*, does not directly produce a derivative code but rather approximates the derivatives by evaluating f at multiple input points. For the simplest case, one-sided differences, the derivative of f with respect to the ith input x_i is approximated by

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x \pm \Delta x_i) - f(x)}{\pm \Delta x_i}.$$

The main drawback of the divided differences technique is that inherent errors make it difficult to determine the accuracy of the approximation [26, 40]. In addition, the computation of n partial derivatives requires $n + 1$ function evaluations.

Recently, automatic differentiation (AD) has been gaining popularity with the emergence of software tools such as ADIFOR [8, 9], ODYSSEE [44], or ADOL-C [30]. Given a code C, these tools can automatically produce an accurate and reasonably fast derivative code C'. AD works by systematically applying the chain rule of differential calculus at the elementary operator level and thus does not incur the errors inherent in divided difference

approximations. Also, if C changes, which is often the case during code development, an up-to-date C' is produced by simply rerunning the tools. An overview of currently available AD tools can be found at `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools`, as well as in the articles in [5].

AD technology is still in its infancy; the development of better algorithms for exploiting chain rule associativity and their incorporation into AD tools promise significant improvement in the performance of generated derivative codes. This article focuses on the design issues in building an extensible AD tool that

- fully supports the ANSI-C language,
- lays the foundation for the rapid assimilation of algorithmic improvements, and
- enables code reuse across different AD tools.

ADIC (Automatic Differentiation of C) is our AD tool that addresses these design issues. A prototype of ADIC was successfully applied to CSCMDO [16], a 3-D volume grid generator specifically designed for use in multidisciplinary design optimization. The current ADIC has been employed to generate derivative-enhanced versions of semiconductor device simulators, a 3-D motion control simulator, and a neural network model. These applications range in size from several hundred to more than 10,000 lines of code.

In the following paragraphs, we set the stage for the remainder of the article by briefly reviewing the algorithmic underpinnings of AD, highlighting the salient points of implementation strategies for AD, and summarizing the novel aspects of our work on ADIC.

1.1 The Algorithmic Aspect of Automatic Differentiation

We first define certain terms that are commonly used in the automatic differentiation context:

- *Independent variables* are program input variables with respect to which derivatives are desired.
- *Dependent variables* are output variables whose derivatives are desired.
- A *derivative object* represents some derivative information, such as a vector of partial derivatives $(\partial z/\partial x_1, \ldots, \partial z/\partial x_n)$ of some variable z with respect to a vector x.
- Any program variable with which a derivative object is associated is called an *active* variable. A conservative strategy is to consider all program variables to be active, but only variables that are on a computational path from an independent to a dependent variable need to be active.

The execution of any program, no matter how complex, boils down to a series of elementary operations such as an arithmetic operator (e.g., add, multiply) or an intrinsic function (e.g., sine, cosine). Thus, a particular set of input values to the program induces an execution path that transforms input values into the output values. The derivatives are computed by repeatedly applying the chain rule to combine the local partial derivatives of each executed operator.

For example, let a and b be intermediate values that depend on some independent variables x, and let $g = f(a, b)$. Then, by using the chain rule, $\nabla_x g$, the derivative of g with respect to x, is computed as

$$g = f(a,b) \quad \Rightarrow \quad \nabla_x g = \frac{\partial f}{\partial a} * \nabla_x a + \frac{\partial f}{\partial b} * \nabla_x b, \tag{1}$$

and a familiar incarnation is the product rule

$$g = a * b \quad \Rightarrow \quad \nabla_x g = b * \nabla_x a + a * \nabla_x b. \tag{2}$$

Note that this approach can easily be generalized to compute derivatives of arbitrary order [4, 11].

Due to the associativity of the chain rule, there are many ways to combine local derivatives, and each different accumulation path may exhibit different time and memory complexity. The classic *forward mode* of automatic differentiation accumulates derivatives as the computation proceeds from the inputs to outputs. Another classic method, the *reverse mode*, accumulates the derivatives in the opposite direction—from the outputs to inputs. The reverse mode requires a reversal of the order of the original program execution, but is attractive when we desire the computation of the derivatives of few output values with respect to many input values. These issues are discussed in more detail in [28, 31, 42]. For general problems, the strategy to be chosen depends on the time and memory constraints and also the particular program structure. The development of heuristics to exploit algorithmic and program structure, and hence to better exploit chain rule associativity, is the subject of current research (see, for example, [15, 17, 23, 29, 32]).

1.2 Implementation Strategies for Automatic Differentiation Tools

Unlike compilers or most program transformation systems, *AD tools modify the semantics* of the underlying program by inserting, in a rule-based fashion, code for computing derivatives. In this section, we give a brief overview of two main approaches taken by automatic differentiation tools to implement this semantic augmentation process.

Operator overloading: This approach overloads the basic arithmetic operators and intrinsic calls with special routines that carry out the propagation of derivatives in addition to the original operations or, alternatively, record information that allows

a subsequent code reversal. The source program itself is only minimally changed, and most of the complexity of derivative computations is embedded in a library. The operator overloading mechanism properly invokes these library routines as the execution proceeds. This approach works with languages that support operator overloading such as C++ or Fortran 90.[‡] Despite the elegance of this approach, its major disadvantage is the inability to exploit chain rule associativity at compile time because of the lack of context inherent in operator overloading [9, 15]. In addition, some overhead is associated with the operator overloading itself. Examples of this approach are ADOL-C [30], ADOL-F [45], ADO1 [41] and OPTIMA [20, 3].

Source-to-source transformation: This approach employs compiler techniques to transform a program source code into a new source code that explicitly carries out the derivative computation; hence, it is applicable to any language. The advantages of this approach are that the entire program context is available at compile time to exploit chain rule associativity and to gather information that could tighten conservative assumptions underlying AD algorithms. The disadvantage is the major effort required in implementing such a tool. ADIC, ADIFOR [8, 9], ODYSSEE [44], and TAMC [25] are examples of this approach.

1.3 What's New in ADIC

Our goal was to build an efficient AD tool for ANSI-C geared primarily toward the computation of first- and second-order derivatives. For efficiency, we employ the source transformation approach.

Figure 1 shows the logical stages of such a tool for AD.

- *Parsing:* The source is parsed into an intermediate representation.
- *Canonicalization:* The intermediate format is transformed into a semantically equivalent form more suitable for automatic differentiation.
- *Analysis (optional):* This stage may, for example, determine which variables are active and hence require associated derivatives (versus making the conservative assumption that all variable have this requirement).
- *Derivative Augmentation:* The strategy for applying the chain rule is determined and the source augmented with derivative computations.
- *Optimization (optional):* For example, certain low-level computational kernels may be instantiated in an architecture-specific fashion. Traditional scalar and loop optimizations may also be performed here.

[‡]It is possible to take an ANSI-C program and compile it as a C++ program. However, since ANSI-C is not a strict subset of C++, this approach is not infallible.

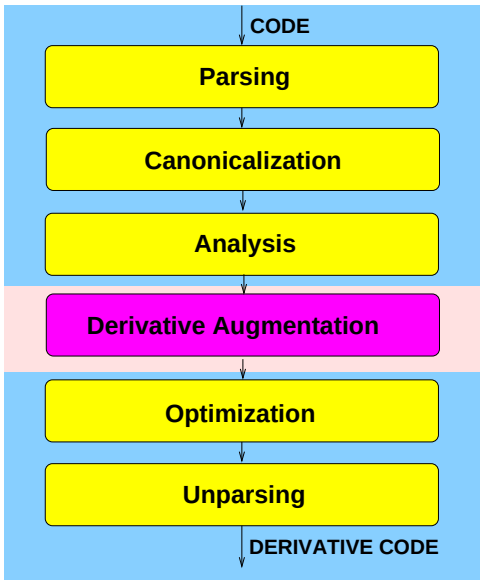

Figure 1: Anatomy of an AD translator

- *Unparsing:* The transformed source is unparsed back into legal code in the original language.

Most existing AD tools operate on Fortran 77 or Fortran 90 programs. However, ANSI-C is also widely used in scientific and engineering applications, and thus robust AD tools that handle ANSI-C programs are needed. Two main features distinguish ADIC:

- It is the first source transformation-based AD tool for ANSI-C. Developing a source transformation-based tool for C is challenging because a number of issues arise (for example, pointers and dynamically allocated memory) that do not occur in Fortran 77.

- It employs a component architecture that insulates the *derivative augmentation* stage from the peculiarities of a particular input language and allows for the development and incorporation of language-independent "plug-in" AD modules. This is achieved through AIF (automatic differentiation intermediate form), which provides a language-independent abstraction of program fragments, as well as abstractions for common AD concepts.

While we are at this point primarily interested in derivatives, we note that ADIC actually provides an infrastructure for arbitrary semantic augmentation of ANSI-C programs, for example, with interval arithmetic.

The rest of this article is organized as follows. The next section gives a short example that illustrates automatic differentiation in general, as well as ADIC's particular approach. Section 3 discusses in detail the issues that arise when one wants to associate additional information and computation with an ANSI-C program, and how ANSI-C code needs to be canonicalized to that end. In Section 4 we describe the AIF component architecture as well as the philosophy behind it. Four applications that used ADIC to compute first-order derivatives are presented in Section 5. In Section 6, we summarize the benefits of ADIC, and we mention areas for future development.

2 ADIC in a Nutshell

ADIC aims to provide AD functionality for general ANSI-C programs. ADIC can also handle many features of C++, such as classes and methods, but cannot handle arbitrary C++ programs yet (e.g., those containing templates).

2.1 An Example

ADIC can be used as a "black box" to generate derivative code with minimal human effort. As an illustration, Figure 2 shows the listing of a simple function `func` contained in a file `foo.c` that we would like to differentiate.

```
 1  #include <math.h>
 2  typedef struct {
 3    double *value_ptr;
 4    char tag;
 5  } footype;
 6
 7  double func(footype *a, double b, double x[], int n)
 8  {
 9    double r,t; int i;
10    r = 1.0;
11    for (i = 0; i < n; i++) {
12       t = (*a->value_ptr)*x[i]*b;
13       if (t != 0.0) {
14          r *= sqrt(t);
15       }
16    }
17    return r;
18  }
```

Figure 2: File `foo.c` containing function `func`.

```
 1  #include "ad_deriv.h"
 2  #include <math.h>
 3  #include "adintrinsics.h"
 4
 5  typedef struct {
 6    DERIV_TYPE  *value_ptr;
 7    char tag;
 8  } footype;
 9
10  void  ad_func(DERIV_TYPE  *ad_var_, footype  *a,
11                DERIV_TYPE b, DERIV_TYPE x[], int n) {
12
13    double     ad_loc_0, ad_loc_1;
14    double     ad_adj_0, ad_adj_1, ad_adji_0;
15    DERIV_TYPE ad_var_0, r, t;
16    int i;
17
18    static int g_filenum = 0;
19    if (g_filenum == 0) {
20       adintr_ehsfid(&g_filenum, __FILE__, "func");
21    }
22
23    ad_grad_axpy_0(DERIV_grad(r));
24    DERIV_VAL(r) = 1.0;
```

Figure 3: First part of file `func.ad.c` containing function `ad_func` as generated by ADIC.

The invocation of ADIC with the command

```
% adiC -d gradient foo.c
```

leads to the generation of file `func.ad.c` shown in Figures 3 and 4.[§] The new function `ad_func` generated by ADIC computes first-order derivatives in addition to the values originally computed.

Derivative Objects. We first note that all double-precision variables and type declarations that were present in `foo.c` have been redeclared to be of type `DERIV_TYPE`. `DERIV_VAL` denotes the original `double` value for each variable of type `DERIV_TYPE`. `DERIV_GRAD` denotes the vector of total derivatives with respect to the chosen independent variables (we will call it a gradient in the sequel) that is associated with an original `double` variable. ADIC also

[§]The code has been reformatted slightly for inclusion here.

```
 1    for (i = (0); i < n; i++) {
 2      ad_loc_0 = DERIV_VAL( *a->value_ptr) * DERIV_VAL(x[i]);
 3      ad_loc_1 = ad_loc_0 * DERIV_VAL(b);
 4      ad_adj_0 = DERIV_VAL( *a->value_ptr) * DERIV_VAL(b);
 5      ad_adj_1 = DERIV_VAL(x[i]) * DERIV_VAL(b);
 6      ad_grad_axpy_3(DERIV_grad(t), ad_adj_1, DERIV_grad(*a->value_ptr),
 7                     ad_adj_0, DERIV_grad(x[i]),
 8                     ad_loc_0, DERIV_grad(b));
 9      DERIV_VAL(t) = ad_loc_1;
10      if (DERIV_VAL(t) != 0.0) {
11        DERIV_VAL(ad_var_0) = sqrt(DERIV_VAL(t));
12        if (DERIV_VAL(t) > 0.0) {
13          ad_adji_0 = 1.0 / (2.0 * DERIV_VAL(ad_var_0));
14        }
15        else {
16          adintr_sqrt(1, g_filenum, __LINE__, & ad_adji_0);
17        }
18        ad_grad_axpy_1(DERIV_grad(ad_var_0), ad_adji_0, DERIV_grad(t));
19        ad_loc_0 = DERIV_VAL(r) * DERIV_VAL(ad_var_0);
20        ad_grad_axpy_2(DERIV_grad(r), DERIV_VAL(ad_var_0), DERIV_grad(r),
21                       DERIV_VAL(r), DERIV_grad(ad_var_0));
22        DERIV_VAL(r) = ad_loc_0;
23      }
24    }
25    ad_grad_axpy_1(DERIV_grad(*ad_var_), 1.0, DERIV_grad(r));
26    DERIV_VAL(*ad_var_) = DERIV_VAL(r);
27    return;
28  }
```

Figure 4: Second part of file `func.ad.c` containing function `ad_func` as generated by ADIC.

generates an include file ad_deriv.h that instantiates these macros as shown in Figure 5. Here ad_GRAD_MAX is a compile-time constant indicating the maximum number of derivatives to be computed (the actual number of derivatives desired up to the maximum is determined at runtime). ADIC can generate a different instantiation of these macros depending on the mode of operation. The foo.ad.c code shows that ADIC also changed the call interface

```
typedef struct {
  double value;
  double  grad[ad_GRAD_MAX];
} DERIV_TYPE;
#define DERIV_VAL(a)  ((a).value)
#define DERIV_grad(a) ((a).grad)
```

Figure 5: Partial listing of file ad_deriv.h generated by ADIC.

to return the original double result through a new parameter; this step avoids unnecessary copying of data due to the return value having been turned into a structure. Calls to func in other files submitted to ADIC would have been changed accordingly.

Derivative Code Generation. Conceptually, the strategy for computing first-order derivatives as shown is the same one that currently underlies ADIFOR 2.0 [8, 9] and thus allows the same flexibility with respect to "derivative seeding" to compute selected derivatives, chain derivatives, or exploit derivative sparsity that is described in these references.

In particular, we use the forward mode overall at the level of statements, but use the reverse mode within an assignment statement. Control flow of the derivative program thus mirrors that of the original code. For each assignment we compute the derivatives of the left-hand side of the assignment with respect to a particular variable on the right-hand side of the assignment. For example, ad_loc_0, ad_adj_0, and ad_adj_1 computed in lines 2, 4, and 5 of Figure 4 corresponds to $\frac{\partial\, \texttt{t}}{\partial\, \texttt{b}}$, $\frac{\partial\, \texttt{t}}{\partial\, \texttt{x[i]}}$, and $\frac{\partial\, \texttt{t}}{\partial\, \texttt{*a->value\_ptr}}$, respectively. The call to the ad_grad_axpy_* routines denotes a vector linear combination; for example, the grad_axpy_3 invocation on lines 6 to 8 of Figure 4 corresponds to

```
DERIV_GRAD(t) = ad_adj_1 * DERIV_GRAD(*a->value_ptr)
              + ad_adj_0 * DERIV_GRAD(x[i])
              + ad_loc_0 * DERIV_GRAD(b);
```

This is a particular instantiation of the chain rule, namely,

$$\nabla t = \frac{\partial\, t}{\partial\, z} * \nabla z + \frac{\partial\, t}{\partial\, x[i]} * \nabla x[i] + \frac{\partial\, t}{\partial\, b} \nabla b. \tag{3}$$

The use of DERIV_GRAD, DERIV_VAL, DERIV_TYPE, and grad_axpy_* provides abstraction as well as considerable flexibility in how to associate a gradient with a memory location containing a double (this issue is explored in more detail in Section 3.1), what data structures to use for implementing a derivative object, and how to implement the vector linear combination with the chosen data structure.

For example, we can provide the ad_grad_axpy_3 functionality through either a macro or a function call. This is shown in Figures 6 and 7. The ad_grad.h file included in ad_deriv.h supplies the appropriate macro definitions or external declarations. There may also be several implementations of the functions, each tailored for a particular type of problems. The decision on which approach to use can be deferred to compile or link time.

```
#define ad_grad_axpy_3(gz, ca, ga, cb, gb, cc, gc) {\
                int g_i_;\
                for (g_i_ = 0; g_i_ < DERIV_SIZE; g_i_++) {\
                    gz[g_i_] =  + (ca)*ga[g_i_] + (cb)*gb[g_i_] \
                                + (cc)*gc[g_i_];\
                }\
            }
```

Figure 6: Macro instantiation of ad_grad_axpy_3.

```
void grad_axpy_3(double* dest, double adj_1, double* grad_1,
                 double adj_2, double* grad_2,
                 double adj_3, double *grad_3)
{
  int i;
  for (i = 0; i < DERIV_SIZE; i++) {
    dest[i] = adj_1 * grad_1[i] + adj_2 * grad_2[i]
            + adj_3 * grad_3[i];
  }
}
```

Figure 7: Subroutine instantiation of ad_grad_axpy_3.

Handling Intrinsics. The sqrt(x) intrinsic is not differentiable when x equals zero. To alert the user to such an occurrence, ADIC checks for this occurrence and prints a warning message. The file adintrinsics.h that is included in line 3 in Figure 3 provides definitions for the ad_intr_ehsfid (line 22 in Figure 3) and adintr_sqrt (line 20 in Figure 4) functions

that set up an error handler for this file and report the occurrence of `sqrt(0)`, respectively. The latter function also provides a reasonable default value for the local partial derivatives (e.g., `ad_adji_0`) so that the execution can proceed. These functions are part of an ANSI-C instantiation of the ADIntrinsics subsystem [37]. While most of the time the evaluation of an intrinsic at a point of nondifferentiability does not compromise the overall result, subtle issues may arise whose satisfactory solution does depend on the particular application context [10, 6].

2.2 The ADIC Process

In this subsection, we further expand the internal aspect of an AD tool briefly outlined in Section 1.3 for the case of ADIC. The automatic differentiation process with ADIC is shown in Figure 8. The user submits the code to be differentiated, as well as optional control files. The control files may indicate optional configuration items such as the prefix that is used to generate new file and function names (all the examples in this article use "ad_" as the prefix) or decide whether to inline certain utility functions to improve performance but at the expense of code expansion. For more details, see [13].

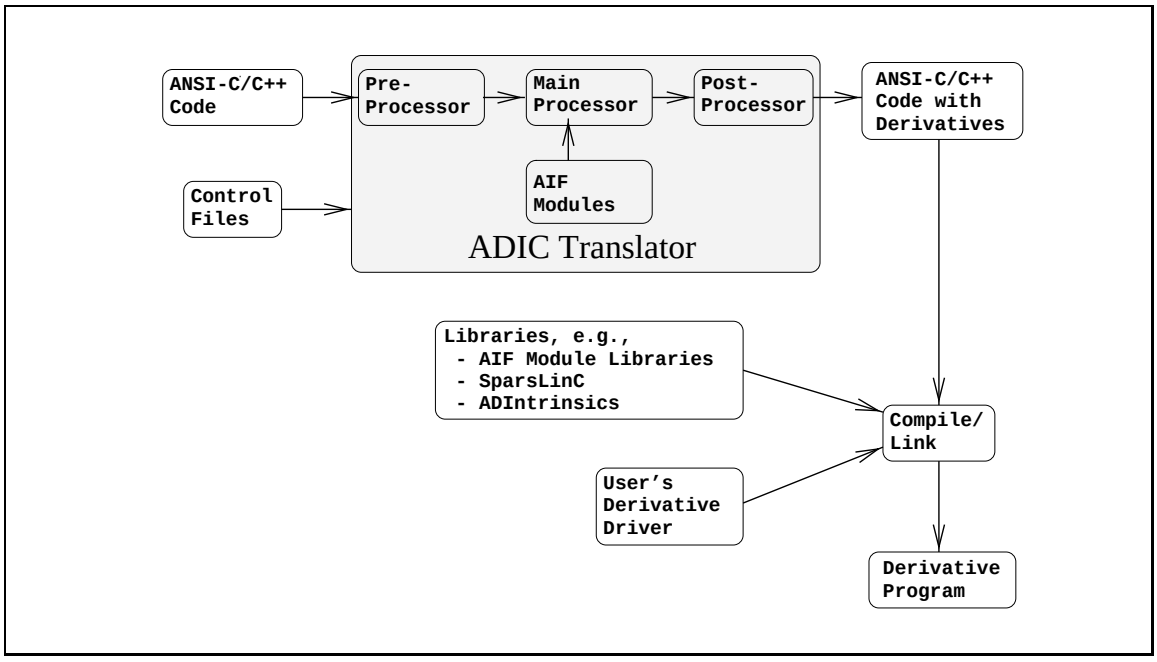

Figure 8: Generating derivative code with ADIC.

Transforming Using ADIC. The ADIC translator generates the derivative code from the code submitted by the user and consists of four components:

- *Preprocessor:* The preprocessor processes the C preprocessor directives and expands macros embedded in the source code. It also marks up the source code so that some of the original C preprocessor directives and macros can later be recovered. These issues are further discussed in Section 3.4.

- *Main Processor and AIF Modules:* This part of the system is our specific instantiation of the generic AD translator shown in Figure 1. We discuss it in more detail below.

- *Postprocessors:* A postprocessor provides the ability to perform a further textual transformation on the generated source. For example, one postprocessor may inline certain calls using templates. Another postprocessor that is routinely used is `purse`, a component of the ADIntrinsics system.

The ADIC main processor and the AIF modules perform the following functions:

- *Parsing:* The marked-up source files are parsed into an intermediate form. To this end, we employ the Sage++ [18] parser.

- *Canonicalization:* At this stage, we canonicalize the intermediate form by addressing ANSI-C-specific issues such as side-effects and pointers. The subtle issues arising in this context are described in Section 3.

- *Analysis:* ADIC currently does not employ data flow or dependence analysis for improving derivative generation. In particular, unlike ADIFOR, ADIC does not perform an interprocedural data flow analysis to determine which variables need to be active, but makes the correct (but conservative) assumption that *all* variables are active. Pointer analysis of ANSI-C programs remains an active research area (see, e.g., [39, 43, 49]); in the future we hope to able to assimilate emerging tools from the compiler community to provide some of these capabilities.

- *Derivative Augmentation:* ADIC's default strategy currently is the forward mode. However, as shown in [15, 17, 34], considerable improvements can be obtained by varying strategies at lower levels within a code. To enable this, we identify code fragments that can be mapped to the simple language underlying the AIF abstraction. As will be shown in Section 4, at the very least assignment statements can be abstracted into AIF, allowing, for example, the statement-based hybrid derivative generation approach underlying the example in Section 2.1. AIF code fragments are augmented with derivative computations based on the strategies present in the AIF modules.

- *Optimization:* Since ADIC inserts derivative computations into the code, we know a lot about the resulting code. Thus, we are in a position to potentially communicate a fair amount of information to code restructuring systems aimed at generating codes tuned for a particular architecture.
- *Unparsing:* We unparse the intermediate form (represented as a combination of AIF and Sage++ intermediate form) to legal ANSI-C code.

Linking with ADIC Runtime Libraries. After the derivative code is generated, it may need to be linked with the following runtime libraries.

- *Libraries Invoked by AIF Modules:* Automatic differentiation programs rely heavily on kernels that implement the vector or matrix operations that are implied by the chain rule. Typically these kernels are provided both as macros and as library functions (e.g., `ad_grad_axpy_3` from Figure 7).
- *SparsLinC:* SparsLinC (Sparse Linear Combination Library) provides implementations of vector linear combinations such as `ad_grad_axpy_3` with data structures that are suitable when the gradients are large and contain many nonzero entries. In this case, SparsLinC provides a much more suitable implementation than the dense loops shown in Figures 6 and 7. SparsLinC, which is written in ANSI-C, was originally developed in the context of the ADIFOR project and has been successfully employed in large-scale nonlinear modeling [7, 12, 17]. Since SparsLinC employs dynamic data structures, from a user's perspective it allows the exploitation of derivative sparsity without any *a priori* knowledge of the sparsity structure in a transparent fashion.
- *ADIntrinsics:* The ADIntrinsics system provides (1) a reasonable default behavior for all cases where the derivative of a standard C intrinsic is not defined and (2) an error-reporting mechanism that gives users control over the amount of detail reported when exceptions do occur. To compute the elementary derivative of an intrinsic function, the ADIC main processor inserts a call to an *intrinsic template* function. The `purse` postprocessor expands the template call into an appropriate C code, depending upon the error-reporting level desired. A set of user-extensible templates files provide the blueprints for expanding the template calls. Finally, the error handler library provides a collection of routines used to record and report runtime errors and to change certain default values.

Writing and Linking the Driver. The user provides a driver that specifies, at runtime, the input variables with respect to which derivatives actually need to be computed. In fact, with proper initialization, we can compute *directional* derivatives (this process is termed "derivative seeding"; see [8, 9])).

```
 1  #include <stdio.h>
 2  #include "ad_deriv.h"
 3
 4  /* definition for footype type is the same as in Figure 3. */
 5  typedef struct {
 6    DERIV_TYPE *value_ptr;
 7    char tag;
 8  } footype;
 9
10  int main() {
11    struct footype foo;
12    DERIV_TYPE bias;
13    DERIV_TYPE x[ad_GRAD_MAX];
14    DERIV_TYPE result;
15    int        size;
16
17    /*read in size, bias, and x from the stdin, and initialize foo*/
18    scanf("%d", &size);
19    scanf("%lf", &DERIV_VAL(bias));
20    for (i = 0; i < size; i++) scanf("%lf", &DERIV_VAL(x[i]));
21    foo->value_ptr = x + size - 1;
22
23    /*initialize*/
24    ad_ad_init();
25
26    /*set up seed matrix (independent variables)*/
27    ad_SET_INDEPENDENT_ARRAY(x, size);
28    ad_grad_clear(bias);
29
30    /*call the derivative function*/
31    ad_func(&result, foo, bias, x, size);
32
33    /*print the result and its gradient*/
34    printf("%f\n", DERIV_VAL(result));
35    for (int i = 0; i < 30 ; i++) {
36      printf("%f\n", DERIV_GRAD(result)[j]);
37    }
38  }
39
```

Figure 9: A driver for `func.ad.c` from Figures 3 and 4.

We show a simple driver for `func.ad.c` (from Figures 3 and 4) in Figure 9. Variables that used to be of type `double` are now declared to be of type `DERIV_TYPE`, and their values are referred to via the `DERIV_VAL` macro. The `ad_SET_INDEPENDENT_ARRAY` utility function sets up `size` elements of array `x` as independent variables. The gradient of `bias` is initialized to zero. The `foo->value_ptr` variable is set to point to the last element of `x`; hence there is no need to explicitly initialize its gradient (this initialization was chosen to demonstrate the fact that aliasing does not present a problem). The call to `ad_func` then computes the derivatives of `result`, the result of the original function, with respect to the entries of `x`. In particular $\frac{mboxresult}{x(j)}$ is contained in `DERIV_GRAD(result)[j]`.

3 Handling the C Language

Automatic differentiation is a particular instantiation of a semantic augmentation process that inserts additional computations related to the part of the program that deals with floating-point numbers. We can view the "derivative space" as an additional address space containing *derivative objects* (e.g., first, second, or higher-order derivatives). The access patterns of the derivative space conceptually mirror those of the original program, in that *whenever a floating-point value is changed, we need to update the derivative associated with that value in an analogous fashion.* If we let the term a "float object" denote a memory location that holds a floating-point variable, then the above informal statement suggests that we need to be concerned about three issues:

Derivative Object Association: We need to be able to find, for a given float object, its associated derivative object.

Side Effects: To be able to do analogous actions in the derivative world, we need to be able to repeatedly refer to subexpressions representing indices and addresses. Thus, we need to isolate side effects to make this possible.

Expressivity: Complex operations may be represented in a compact form (e.g., `if ((c=a) || (d=c=b)) { ... }`). When augmenting the code, the derivative generation process must respect various syntactic limitations; for example, we cannot simply insert statements to update derivatives inside the control expression.

We will address these issues in the context of the ANSI-C language in the next subsections.

An important practical issue is the portability of the code generated by ADIC. Many implementation details of the ANSI-C standard are platform dependent; see, for example, the function and data structures in `<stdio.h>`. Thus, for ADIC to be usable as a cross-translator, we need to retain some of the C preprocessor directives and macros embedded in the original source code whose expansions, however, are necessary to parse (understand) the program. In the last subsection we briefly discuss issues that arise from C's preprocessing.

3.1 Derivative Object Association

In C, float objects may be created either explicitly (through variable declarations or dynamic memory allocations) or implicitly (through function returns or by casting). An *lvalue* is simply an expression that, when evaluated, describes an explicitly created float object; examples from Figure 2 are `*a->value_ptr`, `x[i]`, and `b`.

For automatic differentiation to work, we must be able to associate a unique derivative object with each float object used in a computation. The lifetime of the derivative object must be at least that of its associated float object; so if two float objects have non-overlapping lifetimes, the same derivative object might be reused to save space.

This association is rendered difficult by *aliasing*, that is, two different lvalue expressions may refer to the same float object. Thus, choices of association schemes are fundamentally determined by our ability to resolve aliasing. If we can statically resolve aliasing, as, for example, in Fortran 77, we can trace each lvalue expression back to the point of declaration of the float object denoted by it, and thus we may associate a float object and its derivative object *by name*; for example, we associate a vector `g_x(:)` with the variable `x`. If `x` and `y` are aliases of each other, we also alias `g_x` and `g_y`. Such an approach is taken by ADIFOR [8, 9].

This approach is generally not feasible for C, however, because of the unrestricted use of pointers. Nevertheless, since an lvalue expression evaluates to a unique address, *we can use the address as the basis of an association scheme.* That is, for any float object x and its associated derivative object ∇x, we choose $\&\nabla x = \mathcal{F}(\&x)$, where $\mathcal{F}$ specifies a mapping function and & represents the address of an object. Implicit here is the ability to use $\&x$ twice—once in the context of the original program, and another time to determine the location of its associated derivative object. This is a problem for implicitly created float objects, which are accessible only once at their creation. ADIC handles this during the canonicalization stage by creating a temporary and copying the implicit float object to it. Figure 10 shows an example.

The easiest way to implement the *by-address* association scheme was illustrated by the example in Section 2.1, where the floating-point variables were changed to structures containing the original value and a fixed-size array for the associated derivative object (see Figure 5). Here the mapping is very simple: $\mathcal{F}(\&x) = \&x + c$ for some constant c.

As a variant of this method, instead of storing the derivative object directly in the structure itself, we may store only a pointer to the derivative object, as shown in Figure 11. This approach is necessary if we want to use dynamic data structures such as those used by SparsLinC. In addition, this scheme allows memory savings through lazy allocation of derivative objects, and computational savings through special representations of vectors that are all zeros (e.g., through a `NULL` pointer). The major difficulty with this approach lies in the prevention of memory leaks—all allocated derivative objects must be deallocated when the associated float objects are deallocated or goes out of scope. This is possible if we can statically determine the scope of a float object, but is generally impossible to resolve

Original Code:

```
unsigned long get_information (int key);
double x,y;
int key;

y = x * (double) get_information (key);
```

Canonicalized Code:

```
unsigned long get_information (int key);
double tmp,x,y;
int key;

tmp = (double) get_information (key);
y = x * tmp;
```

Figure 10: Implicit float object represented by the cast expression is made explicit by saving it to a temporary.

```
struct DERIV_TYPE {
    double value;
    derivative_structure* grad;
};
```

Figure 11: Different methods of associating a gradient vector by modifying float types.

automatically for dynamically allocated memory which can potentially be freed anywhere in the program. In this context, garbage collection mechanisms such as the one described in [19] may play a useful role.

This approach of redefining `double` in the derivative code cannot be used, however, in the following circumstances:

- We cannot statically determine that a particular memory location will be used to house a `double`. While we expect that a programmer would write `malloc(k*sizeof(double))`, which allows us to adjust the memory allocation, a call to `malloc(1024)` and a cast to a `double*` somewhere later in the program are also possible.
- A data structure containing a `double` cannot be changed for a variety of reasons—it may be a memory-mapped I/O port, or it may be used by an additional code linked with the derivative code. This issue may be resolved by retaining both the original and augmented data structures and copying values back and forth as needed.

At this point, ADIC employs the schemes shown in Figures 5 and 11, but without fully addressing the memory leak question associated with the latter. We are also working on an associative scheme that does not modify the original data structure, to avoid the need for duplication of data structures. The associative map is based on a dynamic data structure that tries to take advantage of the locality of data accesses. This is the most expensive approach, but it is always feasible. Our goal is to develop an infrastructure for ANSI-C that allows us to *correctly* augment arbitrary programs and to use a combination of static analysis and user directives to adaptively choose the least expensive approach. We also note that a lot of the issues that are troublesome would not arise in a language like Java which has built-in garbage collection and strict type conversion rules.

3.2 Handling Side Effects

For the derivative augmentation process to work, float objects and their associated derivative objects must be accessed in the same fashion. Side effects obstruct this "parallel" behavior. For a simple case, consider `x[i++] = y[i]`. We need to reference the expression `x[i++]` several times (for example `DERIV_VAL(x[i++])`); however, each such reference then refers to a different variable. Thus, we need to rewrite the code to ensure that lvalues referring to float objects are free of side effects.

To this end we evaluate all lvalues that may cause side effects only once by hoisting them out of expressions during the canonicalization stage. We make sure that the transformations do not change the semantic meaning of the program. Figure 12 shows an example for an autoincrementing address.

Figure 13 shows another example where an lvalue expression on the left-hand side contains a function call (with potential side effects). The function returns a pointer to a

Original Code:

```
data[i++] *= scale;
```

Canonicalized Code:

```
data[i] *= scale;
i++;
```

Figure 12: Handling side effects

`double`, which is then dereferenced. Essentially any action can occur in a function, so there is no hope of "understanding" the side effects, as in the previous cases. Since the function's immediate result (before being dereferenced) is being used only as a value, not as a storage location, this value can be hoisted to a temporary.

Original Code:

```
(*f(x)) /= y;
```

Canonicalized Code:

```
t1 = f(x);
(*t1) = (*t1) / y;
```

Figure 13: Example of hoisting a function call with potential side effects

3.3 Expressivity

The C language provides a rich set of operators and syntactic constructs to compactly describe computations. The compactness makes the automatic differentiation process more difficult by hiding access to certain values. For example, in statement y `*=` x, the variable y plays two roles: its original value appears on the right-hand side of the assignment, and it is a lvalue that will be modified after the operation. In this particular case, rewriting it into y `=` $y * x$ is sufficient.

However, things may not be as simple. Consider, for example, the code fragment labeled "original code" in Figure 14. The upper part of Figure 14 shows a loop that contains a `continue` statement. When the `continue` statement is executed, the execution immediately skips to the next iteration by executing the iterative expression (`z += 2.0`) and then performs the loop test. Since we cannot add statements that perform derivative compu-

Original Code:

```
for (z = 0.0; func(z) > 1.0; z += 2.0) {
    [...]
    if (k) {
        continue;
    }
    [...]
}
```

Canonicalized Code:

```
z = 0.0;
for (; func(z) > 1.0;) {
    [...]
    if (k) {
        goto label;
    }
    [...]
label:
    z += 2.0;
}
```

Figure 14: The loop is rewritten before the derivative augmentation step.

tations inside the expression part of the loop statement, we hoist the initial expression of the `for` loop out of the loop and the iterative expression to the bottom of the loop. To preserve the original semantic, we change the `continue` into a `goto`. Hence, ADIC rewrites the computation to make all operations explicitly visible as is shown in the part labeled "canonicalized code" in Figure 14. If the lower bound or increment expressions contains side effects, they would be further treated as previously described.

Original Code:

```
if (((a = b) == x) || ((a = c) == y)) {
    x = k;
}
```

Canonicalized Code:

```
int  flag;
[...]
a = b;
if (a == x) {
    flag = TRUE;
}
else {
    flag = FALSE;
    a = c;
}
if (flag == TRUE || a == y) {
    x = k;
}
```

Figure 15: Implicit control flow operations are made explicit.

Another potential problem arises from C operators that have implicit control flow. For example, the logical OR operator in `(a=x) || (b=x)` will short-circuit (i.e., not execute the second assignment), if the first assignment expression is nonzero. When propagating derivatives, the derivative code must behave accordingly. As shown in Figure 15, ADIC rewrites the code during the canonicalization phase to make the control flow explicit and isolates the `a = c` side effect.

3.4 Preprocessor Issues

The C preprocessor expands macros and handles other directives embedded in the source. The portability and flexibility of C programs stem in part from this preprocessing facility. Unfortunately, this flexibility can significantly complicate source-to-source transformation systems, since directives and macro usage will be lost in the preprocessed source file.

In most instances, no problem results. However, some of the portability and flexibility provided by the preprocessor facility is lost. Generally, it is either impossible or does not make sense to recreate all the original preprocessor directives and macro uses after transformation. However, in some instances such a capability is very useful. To this end the pre-processor component of ADIC provides the following facilities:

- `include` directives typically are used to include the contents of the standard or user header files into the source. Since the included standard headers (like `<stdio.h>` or `<math.h>` are determined at transformation time, the transformed source is not portable across platforms. To handle the `include` directives, ADIC marks up the locations of any included text and stores the names of the original header files. When ADIC generates the augmented source, the entire contents of the standard headers (which are needed during the augmentation process) are replaced with the original directives. ADIC also provides an option to restore the user include directives.

- A macro can represent any text. A function-like macro can also take arguments, performing argument substitutions during the preprocessing stage. Wherever the macro name occurs in the source, it gets expanded. The expanded macros may not be portable across machines or even across compilers, since they may be system dependent (e.g., FILE defined in `stdio.h`). ADIC allows the user to specify macros that should *not* be expanded through its control file mechanism. This facility can also be used to handle function-like macros and type definitions.

These preprocessor issues are discussed in more detail in [13].

4 AIF – The Automatic Differentiation Intermediate Form

The preceding sections discussed how to prepare an ANSI-C code for derivative augmentation. In this section, we discuss the mechanism for achieving this augmentation. As mentioned previously, automatic differentiation is a field in its infancy. Hand-translation schemes (for small codes) have shown the promise of going further beyond either the traditional forward and reverse modes or the static hybrid scheme used in ADIFOR. However, changing the underlying AD approach typically implied considerable effort, since the implementations of AD algorithms in existing AD tools are deeply tied to the implementation infrastructure of a particular tool. Thus, these implementations of AD algorithms suffer from two limitations.

Lack of Abstraction: From the perspective of an AD algorithm, the two assignment statements

```
z = 2.0 + x * y
foo->struc.z = 2.0 + bar->x * q[c]
```

are identical in that both represent

```
var_1 = const + var_2 * var_3
```

where `var_*` are floating-point variables, and `const` is a constant.

Lack of Portability: AD algorithms augment only the floating-point computations of a program and are, except for the purpose of static analysis, oblivious of the remainder. Thus, as suggested by the algorithmic commonality of ADIFOR and ADIC, there is much scope for bringing the very same algorithms to bear on both Fortran and C codes, for example. However, previously this meant replication of the coding effort in both language contexts, although libraries such as SparsLinC could be shared.

Concurrent with the development of ADIC, we have developed AIF (automatic differentiation intermediate form) to provide an infrastructure that allows experimentation with AD algorithms in a language-independent fashion and to promote software reuse. To this end, we define a simple language for AD to which we can map program fragments written in languages such as C and Fortran. The AIF language has the usual notion of functions, statements, and expressions. It defines canonical forms of various control flow constructs such as loops and conditionals available in most high-level languages. In addition, it defines several data types (e.g., integer, double-precision float) and the usual arithmetic operators. Our goal is not to make it so feature-rich that it ends up being a full C/Fortran intermediate form but, rather, provide a useful set of canonical language features to enable us to map at least fragments of programs. Since many constructs are not relevant in the AD context, the AIF language also defines a NO-OP expression and statement, which are not interpreted and serve only as placeholders (useful when we need to map back into the original language). Hence, many low-level notions are either abstracted away (e.g., `foo->struc.z` becomes `VAR_T`, denoting the reference to a variable) and code fragments such as `a | b` that do not affect automatic differentiation are hidden inside a NO-OP placeholder.

For a given input C or Fortran program, then, we can map it into a set of AIF code fragments. The size of each fragment depends on the particular structure and well-behavedness of the program; but at the least, we can convert a single statement at a time (e.g., a function call, assignment) into AIF. A *derivative module* takes an AIF code fragment and augments it with the derivative computations according to the strategy built into the module. Hence the derivative module works at the abstract level of the program fragments. The main tool

(called the host) then glues these augmented fragments together to form the final derivative code. A simpler derivative module may handle only a statement fragment at a time, whereas a more sophisticated module may handle a basic block or an even larger fragment. Hence, depending on the sophistication of a particular derivative module, we may need to tailor the size of AIF fragments to be no greater than what the module can handle. Generally, the larger the code fragment, the more sophisticated the AD algorithms that are potentially applicable.

The mechanism just described is used by ADIC to perform the derivative augmentation. We have built a `gradient` module for first-order derivatives, and we are working on a `hessian` module for second-order derivatives. These modules process one statement fragment at a time. We are also upgrading the `gradient` module that can decide on the best strategy to use on a basic block level. Each derivative module is a separate executable program that communicates with the host via files. In addition to the AIF code fragments, the standard application programming interface (API) also defines a control mechanism between the the derivative modules and the host. For example, the host may pass along some information about the program, or the derivative module may notify the host that certain additional files (e.g., headers or libraries) are required to compile the generated AIF code fragment.

With regard to implementation, both the AIF code fragments and controls are represented as annotated abstract syntax trees (AST's). The AST is represented in a child-sibling relationship—the down link of a node represents its first child, and the right link of the node represents its right sibling. Each node may contain one or more attributes that specify additional information about the node. AIF defines a standard set of these attributes but is fully extensible. ADIC also uses additional attributes to store low-level information associated with a node (for example, a pointer to a symbol table entry), which are ignored by the derivative modules. The details of AIF representation are given in [14].

As an illustration, an assignment

```
a = b * c;
```

is converted into AIF to be sent to the `gradient` module. Figure 16 shows the AIF in graphical format. In the figure, the italicized words represent attributes and their values. The `BIND_T` node has various attributes that represent the API requests from the host. In this case, we specify the AIF language version, the AD transformation desired (first-order derivatives), the fact that we do not want to inline gradient calls, that we want to compute at most five directional derivatives, and that the maximum number of variables on the right-hand side of an assignment statement is nine. The `ASSGN_T` denotes an assignment, `VAL_T` denotes the reference to a value, `VAR_T` denotes a floating-point variable, and `MUL_T` denotes the multiplication. The `NAME_A` attributes of `VAR_T` are actually pointers to an internal data structure that identifies the variable.

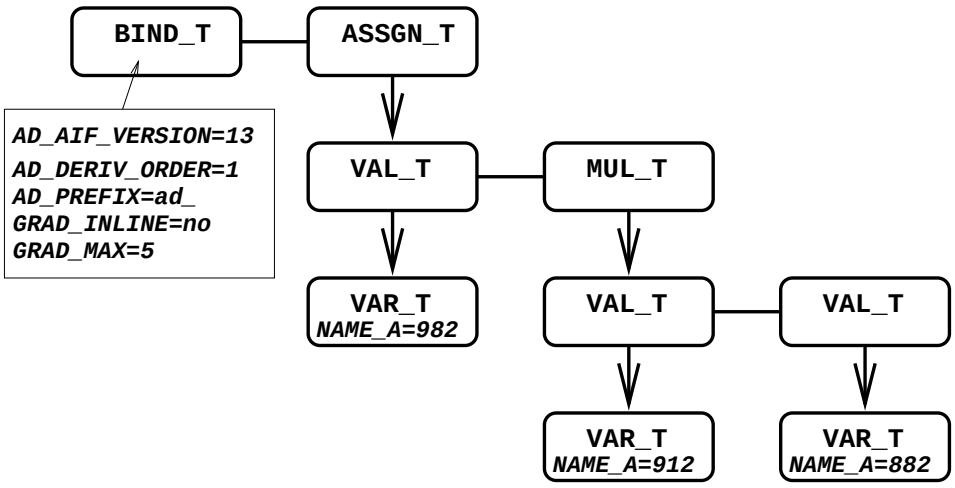

Figure 16: Input AST representing an assignment to be processed by the `gradient` module.

The `gradient` module then transforms this representation into a new AIF fragment that also contains derivative computations, as shown in Figure 17. The new AIF fragment represents the statements

```
ad_loc_0 = DERIV_VAL(b) * DERIV_VAL(c);
ad_grad_axpy_2(DERIV_grad(a), DERIV_VAL(c), DERIV_grad(b),
               DERIV_VAL(b), DERIV_grad(c));
DERIV_VAL(a) = ad_loc_0;
```

The `BIND_T` node specifies various declarations and also the return API requests from the module. The attribute-value pair "`AD_CONST_ad_GRAD_MAX = type=integer default=5 shape=scalar`" specifies that an integer constant named "ad_GRAD_MAX" be declared with the default value of 5. The attribute-value pair "`AD_TYPE_loc = scope=local type=float shape=scalar`" represents a type declaration that specifies that every local name beginning with `loc` is to be properly declared as a double and has the local scope. The `LOC_T` node corresponds to the use of such a temporary variable (identified with the name `!!loc_0`). The attribute-value pair "`AD_DERIV_grad = type=float shape=array,ad_GRAD_MAX`" specifies that a derivative object (associated with all floating-point variables) named `grad` be defined as an array of `ad_GRAD_MAX` doubles. The `STMTS_T` node represents a statement block containing the two assignments and a call. A `DERIV_T` node indicates the reference to a derivative object (named `grad`) associated with a particular float variable. The `VARS_A` attribute of `STMTS_T` summarizes the new local variables that have been requested by the `gradient` module in a statement block. The `NO_SIDE_EFFECT_A` attribute of `CALL_T` spec-

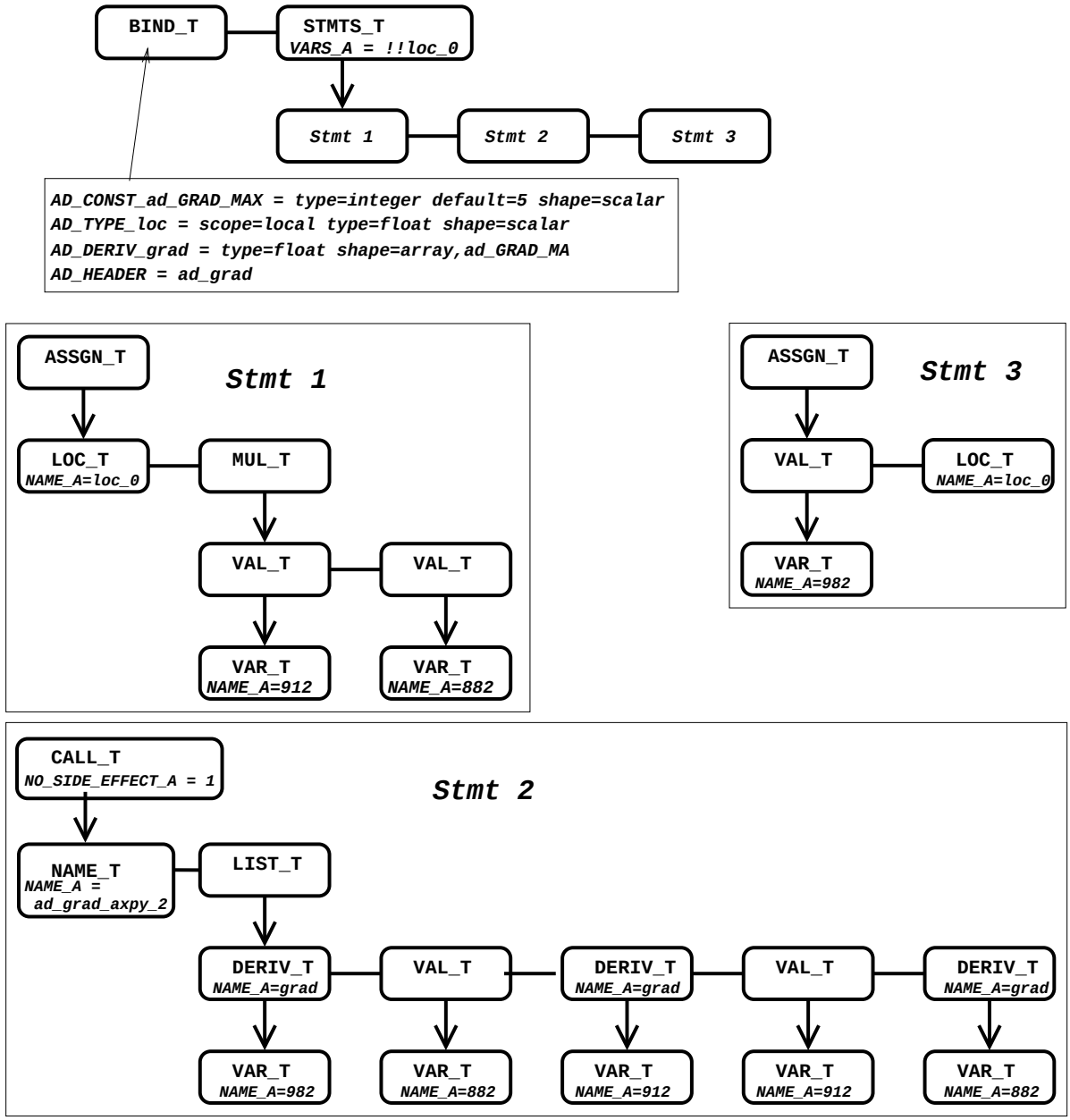

Figure 17: Output AST after processed by the `gradient` module.

ifies that we know the call to `ad_grad_axpy_3` to be free of side effects. These attributes provide useful information when further transformations, such as optimizations, are to be performed.

This example illustrated the following features of AIF:

Abstraction: By abstracting away language-specific ways for referring to float objects, we arrived at a much simpler representation of the program. Also, we can easily refer to "gradients associated with a particular float value" or "local variables" without concern for how these concepts need to be implemented.

Language Independence: The AIF language does not resolve how variables need to be allocated or how derivative associations need to be maintained. Also, the use of standard API provides a platform-independent ways to request information or services between modules and hosts. Thus, the AIF representation tries to provide a language-independent platform for experimentation with AD algorithms, and the same AIF derivative module could easily interface to different AD front-ends.

Flexibility: Instead of the `gradient` module, we could have easily invoked a different module, for example, the `hessian` module, in which case the augmented AIF would compute the second-order derivatives. In fact, a given module may perform *context-sensitive* transformations rather than the same transformation algorithm to every statement, but the change of algorithms is transparent to the surrounding tool layers.

To help developers in writing AIF modules, we have created a toolkit. This toolkit provides a library of C++ classes that insulates the developer from various interfacing issues and provides the AST node definitions, the attribute mechanism, a set of tree manipulation routines, and various utilities. The library takes care of encoding and decoding the API and transparently handles communication with the invoking tool via files. Even though the derivative generation speed is not really a factor, the toolkit does provide the facility so that a module may be directly linked to the host and thus eliminate the latency associated with file-based communication. However, this change would be transparent to users of the AIF toolkit. In our experience, this additional interface layer provided considerable stability to developers of transformation tools, even though the AIF representation itself changed several times. We are doubtful that such development stability could have been achieved if an AIF-like representation had been directly manipulated in Scheme [24] or CAML [48].

In summary, AIF allows us to decouple the world of the language-dependent AD front-end and the world of the AD transformation developer (which sees a simple, language-independent representation of program fragments). In this fashion, we hope to accelerate progress in AD algorithms and accelerate the incorporation of new ideas into robust tools.

5 Applications

We present four applications that show that ADIC can be used to reliably augment ANSI-C codes with derivatives. The variety of application domains attests to the generality of automatic differentiation as well as ADIC. All experiments were performed on a Sun SPARCstation 20 running Solaris 2.5, compiled with `gcc`. For each problem, we report the runtime of the original (unmodified) code, the ratio between the derivative computation and the original code, (labeled $\frac{\text{time}(\nabla f)}{\text{time}(f)}$), and the ratio of the runtime of the ADIC-generated code to central divided differences (labeled $\frac{\text{time}(AD)}{\text{time}(CD)}$). Central differences, which usually deliver acceptable derivative approximations, would have required $2p + 1$ function evaluations. Central difference approximations with varying stepsizes were also used to verify the ADIC-generated derivatives. Since ADIC currently augments all `double` variables with an array for the gradient object, memory requirements scale linearly with the number of derivatives computed.

5.1 The CSCMDO 3-D Volume Grid Generator

CSCMDO is a general-purpose, multi-block, three-dimensional, structured volume grid generator with specialized features for grid modifications that occur in multidisciplinary design optimization contexts [35]. It has been used, for example, with the RAPID 2-D surface grid generator [46] and the TLNS3D 3-D CFD solver [47] in design optimization studies at NASA for the high-speed planes.

CSCMDO consists of 16,500 lines of ANSI-C; the unmodified code runs for 49 seconds. As shown in Table 1, ADIC-generated code is faster than central differences, and its advantage improves as the number of derivatives increases. This result is not surprising because the hybrid forward/reverse AD approach used in ADIC incurs a fixed overhead for the scalar reverse mode computations, which it then tries to recoup by amortizing over loops that compute all desired derivatives at once.

# Derivatives	1	2	3	4	5	6
$\frac{\text{time}(\nabla f)}{\text{time}(f)}$	2.4	3.1	4.0	5.2	6.3	7.4
$\frac{\text{time}(CD)}{\text{time}(AD)}$	1.3	1.6	1.7	1.7	1.7	1.7

Table 1: Timing Results for CSCMDO

5.2 The FCAP2 Circuit Interconnect Simulator

The FCAP (Fast Capacitance Extraction) suite of codes has been under development by Hewlett-Packard Laboratory since the 1980's [21]. These codes are used in the context of simulating capacitance and thermal properties of devices as well as on-chip/off-chip interconnects. They were also incorporated into the Raphael^{TM} capacitance extraction software that is marketed by TMA (Technology Modeling Associate) Inc. FCAP2 consists of 7,680 lines of ANSI-C code.

For our experiments, we use two input models that compute (1) potentials of a couple of trace lines sandwiched between layers of metal planes and dielectrics, and (2) capacitance of five parallel traces between two planes and dielectrics. The runtime for the original code was 6.2 seconds for Model 1 and 7.5 seconds for Model 2.

# Derivatives	1	2	3	4	5	6	7	8	9	10
Model 1										
$\frac{\text{time}(\nabla f)}{\text{time}(f)}$	2.9	5.2	7.4	9.2	11.7	13.2	15.0	16.6	18.7	20.7
$\frac{\text{time}(CD)}{\text{time}(AD)}$	1.03	0.97	0.94	0.98	0.95	1.00	1.02	1.01	1.01	1.01
Model 2										
$\frac{\text{time}(\nabla f)}{\text{time}(f)}$	2.3	3.6	5.0	6.2	7.5	8.4	9.6	10.6	11.9	13.1
$\frac{\text{time}(CD)}{\text{time}(AD)}$	1.31	1.38	1.40	1.46	1.46	1.55	1.55	1.60	1.60	1.60

Table 2: Timing Results for FCAP2

The results in Table 2 show that for the first case, ADIC-generated code is not faster than central difference approximations, while for the second case it is up to 1.6 times faster. These results are due to different execution paths through the code in the two cases (since the derivative code used is identical in both cases). The example shows that, in general, it is somewhat difficult to predict the speedup to be gained from automatic differentiation. The example also corroborates the need for more context-sensitive differentiation strategies, since the same AD strategy seems to be working better in one case than in the other.

Nevertheless, the main benefit is that AD is guaranteed to deliver correct derivatives. With central differences, for FCAP2, smaller grid sizes (and thus more computations) are often required to achieve the desired level of accuracy, and then ADIC generated code is still a faster solution.

5.3 Stewart Platform

In the design of vehicle simulators, which move around a fair amount to simulate actual driving conditions, one is interested in determining their operational envelope, namely, the set of points in space that could be occupied by the simulator. In the context of such work, we were presented with a model for the so-called Stewart platform (the model derivation is described in [2]). The Stewart platform consisted of 763 lines of C++ code, and the unmodified code took 1.7 seconds to complete. The results in Table 3 show that ADIC-generated code is roughly two times faster than central difference approximations throughout.

# Derivatives	5	10	15	20	25	30
$\frac{\text{time}(\nabla f)}{\text{time}(f)}$	5.5	9.7	14.4	19.2	22.8	28.3
$\frac{\text{time}(CD)}{\text{time}(AD)}$	2.0	2.1	2.2	2.1	2.2	2.1

Table 3: Timing Results for Stewart Platform

5.4 Neural Network Model

Our last example is a generic neural network with n inputs, k hidden layers, a single output, and sigmoidal activation functions (as described on p. 279 in [36]). The model consists of 73 lines of ANSI-C. The training of these networks gives rise to an optimization problem that requires a gradient of the model for its solution. The data in Table 4 are based on the time for 1000 executions of the model. For this problem, ADIC-generated code is on average 2.4 times faster than central differences.

6 Conclusions

The need for accurate and fast derivatives for models presented as computer codes is ubiquitous in computational science. Automatic differentiation provides a mechanism for computing those derivatives accurately with minimal human effort. In this article, we presented the design and workings of ADIC, a tool for augmenting ANSI-C programs with derivative computations.

Automatic differentiation is a field very much in its infancy. Recent work has shown that AD tools can reliably augment large computer codes, but much work still needs to be done to realize the algorithmic speedups promised by the associativity of the underlying chain rule of differential calculus. The design of ADIC is geared toward accelerating this progress. It combines an ANSI-C-specific frontend with a language-independent AD transformation

# Inputs	# Hidden Layers	# Derivatives	time(f)	$\frac{\text{Time}(\nabla f)}{\text{Time}(f)}$	$\frac{\text{Time}(CD)}{\text{Time}(AD)}$
3	1	15	0.01	16.3	1.9
3	2	24	0.03	18.8	2.6
3	3	33	0.04	24.5	2.7
4	1	24	0.02	22.8	2.1
4	2	40	0.03	32.5	2.5
4	3	56	0.05	42.4	2.7
5	1	35	0.02	31.4	2.3
5	2	60	0.04	50.4	2.4
5	3	85	0.06	74.1	2.3

Table 4: Timing Results for Neural Network Model

engine. The ADIC frontend address issues such as language canonicalization to make a C code "augmentable" and provide language-specific implementations for abstractions such as "the derivative object associated with a particular variable." The automatic differentiation intermediate form (AIF), a canonical representation of computer programs useful for AD, together with the corresponding toolkit, enables programming of the AD transformation engine at a high level of abstraction easily and independent of a particular platform. In addition, this paradigm promotes software reuse.

Application examples have shown that, in its current form, AD for ANSI-C programs as implemented by ADIC already delivers competitive performance. Enhancements in underlying AD algorithms will further improve this performance, either through algorithmic improvements in the application of the chain rule or through backend optimizations such as loop unrolling.

Automatic differentiation tools also offer promising avenues for the application of research typically done in the compiler and runtime system communities, for instance, in flow analysis and performance prediction. As mentioned, for example, in [9, 17], AD provides ample opportunities for exploiting parallelism, from threads in shared-memory programming models to the typically coarser-grained communication paradigms (e.g., MPI [33]) used in distributed-memory paradigms.

Acknowledgments

We thank Jason Abate of the University of Texas at Austin and Alan Carle of Rice University for their insightful comments and Po-Ting Wu of Argonne National Laboratory for his help in debugging ADIC. We are also indebted to William Jones of Nasa Langley, Norman

Chang of Hewlett-Packard Research Labs, Frederick Adkins of the University of Iowa, and David Juedes of Ohio University for providing us with application problems and assisting us in their operation and verification.

References

[1] Proceedings of the 5th AIAA/NASA/USAF/ISSMO symposium on multidisciplinary analysis and optimization, Panama City, Florida, American Association of Aeronautics and Aerospace Engineers, 1994.

[2] Frederick A. Adkins and Edward J. Haug. Operational envelope of a Spatial stewart platform. Technical Brief, May 1996.

[3] M. Bartholomew-Biggs. OPFAD - a users guide to the OPtima Forward Automatic Differentiation tool. Technical report, Numerical Optimization Centre, University of Hertfordsshire, 1995.

[4] M. Berz. *High-Order Computation and Normal Form Analysis of Repetitive Systems*, volume AIP 249, page 456. American Institute of Physics, Woodbury, NY, 1991.

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

[6] Christian Bischof. Automatic differentiation and numerical software design. In Ronald Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 287–299, London, 1997. Chapman & Hall.

[7] Christian Bischof, Ali Bouaricha, Peyvand Khademi, and Jorge Moré. Computing gradients in large-scale optimization using automatic differentiation. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in ORSA Journal of Computing.

[8] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[9] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[10] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Report ANL/MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[11] Christian Bischof, George Corliss, and Andreas Griewank. Structured second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, 2:211–232, 1993.

[12] Christian Bischof, Peyvand Khademi, Ali Bouaricha, and Alan Carle. Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7(1):1–39, July 1996.

[13] Christian Bischof and Lucas Roh. ADIC user guide, 1996. Unpublished information, Argonne National Laboratory.

[14] Christian Bischof and Lucas Roh. The automatic differentiation intermediate form (AIF), 1996. Unpublished Information.

[15] Christian H. Bischof and Mohammad R. Haghighat. On hierarchical differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94, Philadelphia, 1996. SIAM.

[16] Christian H. Bischof, William T. Jones, Andrew Mauer, and Jamshid Samareh. Experiences with the application of the ADIC automatic differentiation tool to the CSCMDO 3-D volume grid generation code. In *Proceedings of the 34th AIAA Aerospace Sciences Meeting, AIAA Paper 96-0716*. American Institute of Aeronautics and Astronomics, 1996.

[17] Christian H. Bischof and Po-Ting Wu. Exploiting intermediate sparsity in computing derivatives of a leapfrog scheme. Preprint ANL/MCS-P572-0396, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[18] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Goutwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*. IEEE, 1994.

[19] H.-J. Boehm. Space efficient conservative garbage collection. *SIGPLAN Notices*, 28(6):197–206, June 1993.

[20] S. Brown. OPRAD - a users guide to the OPtima Reverse Automatic Differentiation tool. Technical report, Numerical Optimization Centre, University of Hertfordsshire, 1995.

[21] K. M. Cham, S.-Y. Oh, D. Chin, J. Moll, K.Lee, and P. V. Voorde. *Parasitic Elements Simulation*, pages 129–140. Kluwer Academic Publishers, Boston, 2nd edition, 1988.

[22] H. Engl and J. McLaughlin. Proceedings of the Symposium on Inverse Problems and Optimal Design in Industry, Teubner Verlag, Stuttgart, 1994.

[23] Christèle Faure. Splitting of algebraic expressions for automatic differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 117–128, Philadelphia, 1996. SIAM.

[24] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, Cambridge, 4th edition, 1996.

[25] Ralf Giering. Tangent linear and adjoint model compiler, users manual. Unpublished Information, 1996.

[26] Phillip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, London, 1981.

[27] Victor V. Goldman and Gerard Cats. Automatic adjoint modeling within a program generation framework: A case study for a weather forecasting grid-point model. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996. To appear.

[28] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[29] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[30] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[31] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, 1991.

[32] José Grimm, Löc Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space time product for reversing a certain class of programs. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation, Techniques, Applications, and Tools*, pages 95–106, Philadelphia, 1996. SIAM.

[33] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI – Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, 1994.

[34] Paul Hovland, Christian Bischof, Donna Spiegelman, and Mario Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. Preprint MCS-P491-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in SIAM J. Scientific Computing, 18(4) (July 1997).

[35] William T. Jones and Jamshid Samareh-Abolhassani. A grid generation system for multidisciplinary design optimization. In *Proceedings of the Workshop on Surface Modeling, Grid Generation, and Related Issues in CFD Solutions*, pages 11–21, 1995. NASA-CP3291.

[36] David W. Juedes and Karthik Balakrishnan. Generalized neural networks, computational differentiation, and evolution. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 273–286, Philadelphia, 1996. SIAM.

[37] Andrew Mauer, Christian Bischof, and Alan Carle. The ADIntrinsics system for handling automatic differentiation of Fortran 77 intrinsics, 1996. Unpublished information, Argonne National Laboratory.

[38] Michael Monagan and Rene R. Rodoni. An implementation of the forward and reverse mode of automatic differentiation in Maple. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 353–362. SIAM, Philadelphia, 1996.

[39] Heman Pande. *Compile-Time Analysis of C and C++ Systems*. PhD thesis, Dept. of Computer Science, Rutgers University, 1996. Technical Report LCSR-TR-260.

[40] William H. Press and Saul A. Teukolsky. Numerical calculation of derivatives. *Computers in Physics*, 5(1):88–89, Jan./Feb. 1991.

[41] J. D. Pryce and J. K. Reid. AD01 – a Fortran 90 code for automatic differentiation. Unpublished information, 1996.

[42] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[43] Martin Rinard and Pedro Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 54–67, New York, 1996. ACM.

[44] Nicole Rostaing, Stephane Dalmas, and Andre Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[45] Dimitri Shiriaev and Andreas Griewank. ADOL-F: Automatic differentiation of Fortran codes. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 375–384, Philadelphia, 1996. SIAM.

[46] Robert E. Smith, Malcolm G. I. Bloor, Michael Wilson, and Almuttil M. Thomas. Rapid airplane parametric input design (RAPID). In *Proceedings of the 12th AIAA Computational Fluid Dynamics Conference, San Diego, AIAA 95-1687*. American Institute of Aeronautics and Astronautics, 1995.

[47] V. N. Vatsa, M. D. Sanetrik, and E. B. Parlette. Development of a flexible and efficient multigrid-based multiblock flow solver. In *Proceedings of the 31st AIAA Aerospace Sciences Meeting, AIAA 93-0677*. American Institute of Aeronautics and Astronautics, 1993.

[48] P. Weis, M. Mauny, A. Laville, and A. Suarez. *The CAML Reference Manual*, 1990. See also http://pauillac.inria.fr/caml.

[49] Robert P. Wilson and Monica S. Lam. Efficient context-sensitivity pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, New York, 1995. ACM Press.