RWTH Aachen

Department of Computer Science
Technical Report

Second-Order Adjoints by Source Code Manipulation of Numerical Programs

Uwe Naumannn, Michael Maier, Jan Riehme,
and Bruce Christianson

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-13

RWTH Aachen · Department of Computer Science · June 2007

Second-Order Adjoints by Source Code Manipulation of Numerical Programs

Uwe Naumann and Michael Maier[1] and Jan Riehme and Bruce Christianson[2]

[1] LuFG Informatik 12, Department of Computer Science, RWTH Aachen University
D-52056 Aachen, Germany. naumann@stce.rwth-aachen.de
[2] Department of Computer Science, University of Hertfordshire, Hatfield, AL10 9AB, UK

Abstract. The analysis and modification of numerical programs in the context of generating and optimizing adjoint code automatically probably ranges among the technically and theoretically most challenging source transformation algorithms known today. A complete compiler for the target language (Fortran in our case) is needed to cover the technical side. This amounts to a mathematically motivated semantic transformation of the source code that involves the reversal of the flow of data through the program. Both the arithmetic complexity and the memory requirement can be substantial for large-scale numerical simulations. Finding the optimal data-flow reveral schedule turns out to be an NP-complete problem. The same complexity result applies to other domain-specific peephole optimizations.
In this paper we present a first research prototype of the NAGWare Fortran compiler with the ability to generate adjoint code automatically. Moreover, we discuss an approach to generating second-order adjoint code for use in Newton-type algorithms for unconstrained nonlinear optimization. While the focus of this paper is mostly on the compiler issues some information on the mathematical background will be found helpful for motivational purposes.

1 Motivation

Suppose that we are interested in the gradient of a given Fortran implementation

```
real function f(n,x)
  integer n
  real, dimension(n), intent(in) :: x
  ...
end function
```

of a multivariate scalar function $y = f(\mathbf{x})$, $f : I\!R^n \rightarrow I\!R$, with respect to the elements of its input vector $\mathbf{x}$. Classical numerical differentiation by finite difference quotients computes approximations to the gradient entries by perturbing the corresponding elements in $\mathbf{x}$. The following example illustrates this method.

```
  ...
  real, dimension(n) :: x, g
  real, parameter :: h=1e-5
  real :: y, t
  integer i
  ...
  y=f(n,x)
  do i=1,n
    t=x(i)
    x(i)=x(i)+h
```

```
    g(i)=(f(n,x)-y)/h
    x(i)=t
  end do
  ...
```

There are two major problems with this approach.

1. In infinite precision arithmetic we would make the perturbation parameter h as small as necessary in order to get the desired accuracy of the approximation. Unfortunately, floating-point arithmetic is everything but infinite in precision making the right choice of h a matter of trial and error. Moreover, in most cases we do not even know what to aim for when trying out different values of h. It may be dangerous to trust the derivatives obtained by finite difference approximation.
2. Unidirectional (forward or backward) finite differences require $n+1$ function evaluations. Even for a very simple nonlinear computation such as

```
  ...
  integer i
  f=x(1)
  do i=2,n
    f=f*x(i)
  end do
  ...
```

   the runtime can amount to several hours for $n \geq 10^7$. Some of our target applications are in the range of $n \geq 10^9$.

Second-order accuracy can be obtained by centered finite differences. The computational cost though is twice that of the unidirectional approaches.

A semantic source transformation technique known as automatic differentiation [6] can be used to transform numerical codes into tangent-linear and adjoint codes. Adjoint codes compute the gradient with machine accuracy and at a computational cost exceeding that of the original function by merely a constant factor. Accurate second derivatives can be computed by tangent-linear versions of adjoint codes.

In this paper we present a research prototype of the NAGWare Fortran compiler that can generate adjoint code automatically by modifying the original program at the level of the compiler's intermediate representation. The benefit of this approach in the context of numerical algorithms is illustrated in Figure 1 where we compare the runtimes (vertical axis in seconds) of a Truncated Newton algorithm for unconstrained minimization of the Elastic Plastic Torsion problem from the Minpack test problem collection [1] based on the finite difference (FD) and automatic differentiation (AD) approaches for increasing problem sizes n (horizontal axis). In FD mode both the gradients and the Hessian vector products are approximated by finite difference quotients. An adjoint code for the efficient evaluation of accurate gradient information is generated by the compiler automatically in AD mode. The Hessian vector products are evaluated by a tangent-linear version of the adjoint code obtained by operator overloading with automatic type changes performed by the compiler. Obviously, finite differences do not scale well. For $n = 300$ the optimization took approximately 15 minutes

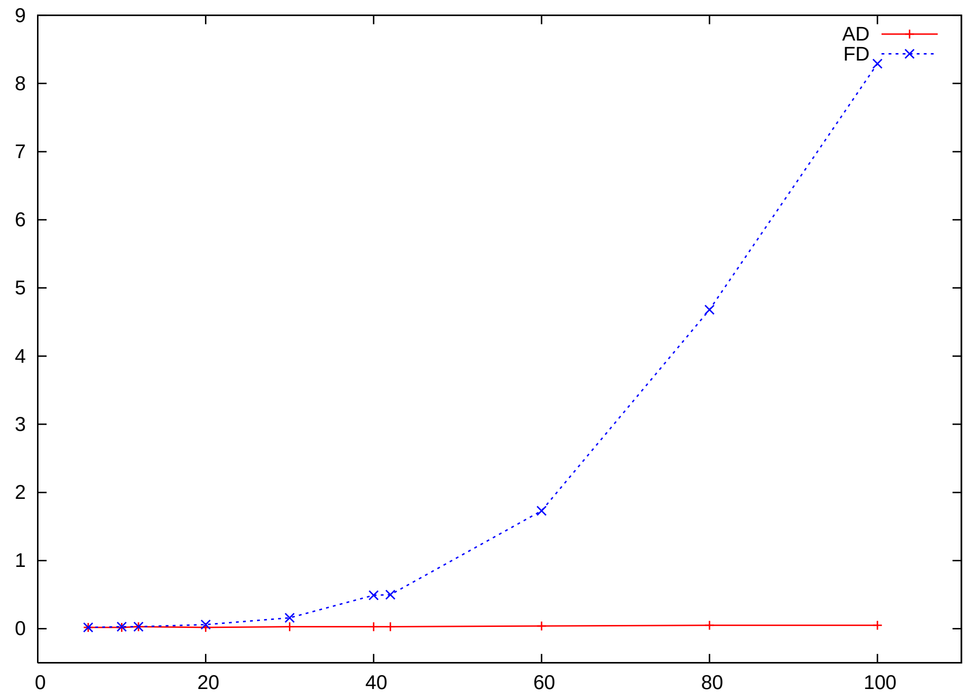

Fig. 1. Runtime of Truncated Newton algorithm for unconstrained minimization of the Elastic Plastic Torsion problem.

on a state-of-the art high-end PC whereas the compiler-generated derivative code lead to a runtime of less than one second. Many more steps had to be performed by the algorithm due to inaccurate numerical approximations of the gradients and the Hessian vector products. Refer to Section 5 for details on the Truncated Newton method.

2 AD Compiler's View on Numerical Code

In this section we introduce parse tree manipulation algorithms for the compiler-based generation of tangent-linear, adjoint, and second-order adjoint numerical programs. We consider (Fortran) implementations

$$F(\overset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{y}})$$

(an overset downarrow marks an input, an underset downarrow marks an output) of multvariate vector functions

$$\mathbf{y} = F(\mathbf{x}), \quad \mathbf{x} \in I\!R^n, \mathbf{y} \in I\!R^m \quad .$$

2.1 Tangent-Linear Code $\dot{F}$

The signature of the tangent-linear code is the following:

$$\dot{F}(\overset{\downarrow}{\mathbf{x}}, \overset{\downarrow}{\dot{\mathbf{x}}}, \underset{\downarrow}{\dot{\mathbf{y}}})$$

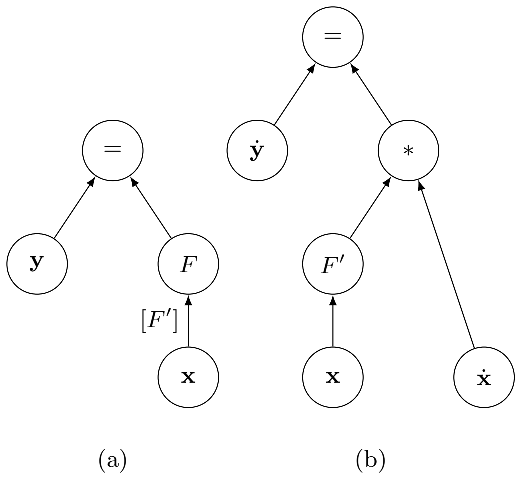

Fig. 2. From F (a) to $\dot{F}$ (b) by Parse Tree Manipulation

where

$$\dot{\mathbf{y}} = F'(\mathbf{x}) * \dot{\mathbf{x}}, \quad \dot{\mathbf{x}} \in I\!R^n, \; \dot{\mathbf{y}} \in I\!R^m \quad .$$

$F' \equiv F'(\mathbf{x})$ denotes the $(m \times n)$-matrix of the partial derivatives of all outputs with respect to all inputs (the Jacobian matrix). The gradient of an output is represented by the respective row in F'.

The tangent-linear parse tree manipulation algorithm for assignments of the form $\mathbf{y} = F(\mathbf{x})$ is illustrated in Figure 2. The right-hand side is linearized by labeling the edge $(\mathbf{x}, F)$ with the symbolic partial derivative $F' = F'(\mathbf{x})$ of its target with respect to its source. The tangent-linear assignment overwrites $\dot{\mathbf{y}}$. The right-hand side of the tangent-linear assignment is build top-down for each node by summation over all incoming edges of the products of the respective edge's label with the tangent-linear subtree of its source. All leaf nodes are replaced by their respective tangent-linear symbols, that is, $\dot{\mathbf{x}}$ and $\dot{\mathbf{y}}$. The partial derivatives are functions of the respective edge's source. This dependence is also represented in the tangent-linear parse tree.

For $z = x * y$ and with $F' = (y, x)$ we simply get the product rule of differentiation as

$$\dot{z} = F' * \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = y * \dot{x} + x * \dot{y} \quad .$$

The recursive nature of the algorithm will become clearer in the context of second-order adjoints illustrated by the transition from Figure 3 to Figure 4.

2.2 Adjoint Code $\bar{F}$

The signature of the adjoint code is

$$\bar{F}(\overset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\overset{\downarrow}{\bar{\mathbf{x}}}}, \overset{\downarrow}{\bar{\mathbf{y}}})$$

where

$$\bar{\mathbf{x}} = \bar{\mathbf{x}} + F'(\mathbf{x})^T * \bar{\mathbf{y}}, \quad \bar{\mathbf{x}} \in I\!R^n, \; \bar{\mathbf{y}} \in I\!R^m \quad .$$

The adjoint parse tree manipulation algorithm for assignments of the form $\mathbf{y} = F(\mathbf{x})$ starts with the linearization of the right-hand side as in Section 2.1. As

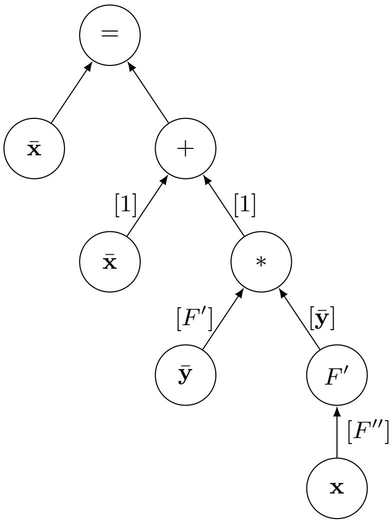

Fig. 3. Linearized Parse Tree of $\bar{F}$

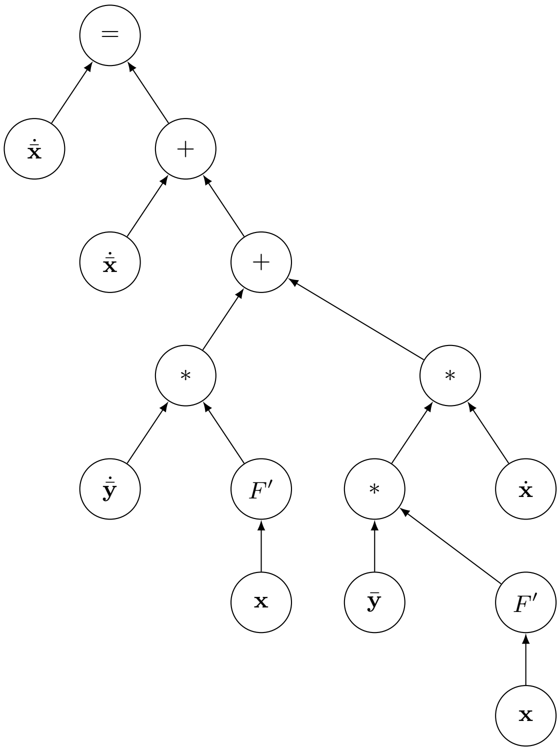

Fig. 4. Parse Tree of $\dot{\bar{F}}$

an immediate consequence of the chain rule the adjoint assignment increments $\bar{\mathbf{x}}$ with the product of the row vector $\bar{\mathbf{y}}$ with the local Jacobian matrix $F'(\mathbf{x})$. The incrementation is due to the data-flow reversal that accumulates adjoints of variables that are read by the original assignment. The corresponding instances can potentially be read by several assignments. Hence, the respective adjoints need to be composed incrementally out of contributions from all of those assignments. Moreover, all adjoints need to be initialized properly (e.g., to zero for local program variables). The algorithm is illustrated in Figure 3. The adjoint parse tree is linearized in order to derive the second-order adjoint code in the next section. Refer to Section 3.1 for a description of the globalization of the data-flow reversal.

For $z = x * y$ we get

$$\left(\bar{x}\;\bar{y}\right) = \left(\bar{x}\;\bar{y}\right) + \bar{z} * \left(y\;x\right)$$

resulting in the two scalar assignments

$$\begin{aligned} \bar{x} &= \bar{x} + \bar{z} * y \\ \bar{y} &= \bar{y} + \bar{z} * x \quad . \end{aligned} \tag{1}$$

2.3 Second-Order Adjoint Code $\dot{\bar{F}}$

Second-order adjoint code can be generated by building a tangent-linear version of the adjoint code. The signature of the second-order adjoint code is

$$\dot{\bar{F}}(\overset{\downarrow}{\mathbf{x}}, \overset{\downarrow}{\dot{\mathbf{x}}}, \underset{\downarrow}{\overset{\downarrow}{\dot{\bar{\mathbf{x}}}}}, \overset{\downarrow}{\bar{\mathbf{y}}}, \overset{\downarrow}{\dot{\bar{\mathbf{y}}}})$$

where

$$\dot{\bar{\mathbf{x}}} = \dot{\bar{\mathbf{x}}} + \dot{\bar{\mathbf{y}}} * F'(\mathbf{x}) + \bar{\mathbf{y}} * F''(\mathbf{x}) * \dot{\mathbf{x}} \quad \dot{\bar{\mathbf{x}}} \in I\!R^n, \;\; \dot{\bar{\mathbf{y}}} \in I\!R^m \quad .$$

Products of the Hessian matrix $F'' = F''(\mathbf{x}) \in R^{n \times n}$ with a vector $\dot{\mathbf{x}}$ can be computed by setting $\dot{\bar{\mathbf{x}}} = \dot{\bar{\mathbf{y}}} = 0$ and $\bar{\mathbf{y}} = 1$ at roughly twice the cost of running the adjoint code. The benefit for the Truncated Newton algorithm is illustrated in Figure 1. Mathematical details are outlined in Section 5.

In order to generate second-order adjoint code for the simple assignment $z = x * y$ we apply the rules for tangent-linear parse tree construction from Section 2.1 to Equation (1) to obtain

$$\begin{aligned} \dot{\bar{x}} &= \dot{\bar{x}} + \dot{\bar{z}} * y + \bar{z} * \dot{y} \\ \dot{\bar{y}} &= \dot{\bar{y}} + \dot{\bar{z}} * x + \bar{z} * \dot{x} \end{aligned}$$

Setting $\dot{\bar{x}} = \dot{\bar{y}} = \dot{\bar{z}} = 0$ and $\bar{z} = 1$ we get

$$F'' \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} \dot{y} \\ \dot{x} \end{pmatrix}$$

as expected. Figure 4 illustrates the general approach.

3 Implementation

The conceptual insight into derivative code generation gained so far needs to be commented on further in order to understand the work of an AD compiler. Various additional issues arise when implementing the theory.

3.1 Data-Flow Reversal

The easiest way to reverse the flow of control is to enumerate all assignments in the original program and to push their indexes onto a control stack during an appropriately instrumented program execution (the augmented forward sweep). The original order is reversed by poping the indexes from the control stack. The following code listing is built on the example from Section 1.

```
1   ...
2   integer i
3   call push_c(1)
4   f=x(1)
5   do i=2,n
6     call push_c(2)
7     call push_d(f)
8     call push_d(i)
9     f=f*x(i)
10  end do
11  ...
12  do while (.not.empty_c())
13    int i
14    call pop_c(i)
15    if (i==1)
16      ...
17    else if (i==2) then
18      call pop_d(i)
19      call pop_d(f)
20      b_x(i) = b_x(i) + f*b_f
21      b_f=x(i)*b_f
22    end if
23    ...
24  end do
```

Pushes to and pops from the control stack are performed by the subroutines `push_c` and `pop_c`. The Boolean function `empty_c` tests for emptiness of the control stack.

Instances of program variables may be required in reverse order if they occur as arguments of nonlinear intrinsics or arithmetic operators. However these instances may be lost during the augmented forward sweep due to overwrites. Such instances need to be made persistent by pushing them onto a data stack before the corresponding memory gets overwritten. The required values need to be restored prior to their use in the computation of a local partial derivative. For example, `f` is overwritten repeatedly in line 9. A push of the old instance of `f` onto the data stack needs to be inserted in front of line 9. The matching pop preceedes the corresponding adjoint statements in the backward sweep. Special care must be taken for address computations in order to reference the correct adjoint memory. See lines 8 and 18. Various domain-specific static code analyses and optimizations must be performed to be able to handle large-scale numerical simulation programs. They are part of our todo list. A complete description of the issues arising in adjoint code generation is beyond the scope of this paper. To our knowledge there is no single publication that covers it all. Refer to [5, 10, 11] for further reading.

3.2 AD by Overloading

A very robust and convenient way to implement tangent-linear code is by overloading of a programming language's arithmetic operators and intrinsic functions for a derived data type that contains in addition to the original function value v a component for the derivative $\dot{v}$. All active program variables (those that potentially carry nonzero derivatives) need to get their types changed to the new data type. This is done automatically by the compiler as outlined in Section 4.2. A static data-flow analysis can compute a conservative estimate for this set [7]. The overhead of the additional function call can be eliminated by the compiler through inlining. Below we show excerpts from the Fortran module that we use for propagating directional derivatives in the context of second-order adjoint codes as described in Section 4.1.

```
module compad_module
implicit none

private

public :: compad_type, tls_type
public :: assignment(=), operator(*) ...
public :: sin ...

type tls_type
  double precision :: v=0
  double precision :: d=0
end type tls_type

type compad_type
  type(tls_type) val
  type(tls_type) drv
end type compad_type

interface assignment(=)
  module procedure tls_assign_tls
end interface assignment(=)

interface operator(*)
  module procedure tls_times_tls
end interface operator(*)
```

```

interface sin
  module procedure sin_tls
end interface sin

...

contains

elemental subroutine tls_assign_tls(y,x)
  type(tls_type), intent(out) :: y
  type(tls_type), intent(in) :: x
  y%v=x%v; y%d=x%d
end subroutine tls_assign_tls

elemental function tls_times_tls(x1,x2) result(y)
  type(tls_type), intent(in) :: x1,x2
  type(tls_type) :: y
  y%v=x1%v*x2%v; y%d=x1%d*x2%v+x2%d*x1%v
end function tls_times_tls

elemental function sin_tls(x) result(y)
  type(tls_type), intent(in) :: x
  type(tls_type) :: y
  y%v=sin(x%v); y%d=cos(x%v)*x%d
end function sin_tls

...

end module compad_module
```

Conceptually, adjoint codes can be implemented by overloading too. However, the interpretive overhead that results from the mapping of the real address space onto a virtual one may be significant. Moreover, static code optimization is even more relevant in adjoint codes. Refer to [3] and [9] for further details on the use of overloading techniques in the differentiation-enabled NAGWare Fortran compiler.

4 AD Compiler Support

In this section we describe a hybrid approach to second-order adjoint code generation that uses extensive parse tree manipulation techniques in addition to overloading. Our focus is on the discussion of the modifications of the compiler's internal representation needed to make the generation of second-order adjoint code fully automatic for the set of test problems currently under consideration. We expect further work to become necessary to cover the full Fortran standard. However, a large step has been made already.

4.1 A Hybrid Approach

We use parse tree manipulation techniques to generate the adjoint code as described in Section 4.3. Second-order adjoints are computed by overloading the adjoint code as outlined in Section 3.2. The use of the second-order adjoint code is best illustrated with an example: Consider the Fortran routine

```
subroutine F(x,y)
  double precision :: x
  double precision :: y
  y=sin(x)
end subroutine
```

The parse tree is manipulated internally to produce adjoint code such that adjoints are associated with original variables *by address*; that is, v (`val`) and $\bar{v}$ (`drv`) are the two components of a derived type `compad_type`. Both of them are of type `tls_type` (with components `v` and `d` of type `double precision` as shown in Section 3.2). Thus, directional derivatives of the adjoints (second-order adjoints) can be computed by overloading. Conceptually, the following code is generated automatically by the compiler:

```
subroutine dbF(x,y)
  use compad_module
  type(compad_type) :: x
  type(compad_type) :: y
  y%val=sin(x%val)
  x%drv=x%drv+cos(x%val)*y%drv
end subroutine
```

The second-order adjoint code is never unparsed. Instead the routine `dbF` is used in a driver routine (for example, in the context of a truncated Newton algorithm) as illustrated below.

```
program main
  use compad_module
  external dbF

  type(compad_type) x,y

  x%val%v=1.;    x%val%d=.5
  x%drv%v=0.;    x%drv%d=0.
  y%drv%v=1.;    y%drv%d=0.

  call dbF(x,y)

  print*, "x= ", x%val
  print*, "bx= ", x%drv
  print*, "y= ", y%val
  print*, "by= ", y%drv

end program
```

Here we simply print the results producing the following (edited) output:

```
1 x=    1.000000   0.5000000
2 bx=   0.540302  -0.4207354
3 y=    0.841470   0.2701511
4 by=   1.000000   0.0000000
```

The interesting value is that of $\dot{\bar{x}}$ = `x%drv%d` (line 2, second value) holding the product of the second derivative of $\sin(x)$, that is, $-\sin(x)$, with $\dot{x} = 0.5$ at point $x = 1$. Take the value of $y = \sin(x)$ (line 3, first value) and divide it by -2 to verify numerical correctness.

4.2 Support for Overloading

The compiler provides support for overloading by automatic type changes and by importing a module named `compad_module`. As simple as it might sound, changing the types of all floating-point variables from `REAL` or `DOUBLE PRECISION` to `compad_type` while parsing the source code yields a number of technical challenges.

`use compad_module` statements must be inserted into all scopes that are to be differentiated.

Calls to structure constructors of type `compad_type` need to be generated for right-hand sides of initializations, for `DATA` entries, and whenever a named or literal constant is passed as an actual parameter to a subroutine that expects a dummy parameter of type `compad_type`.

The types of named floating-point constants is also changed to `compad_type`. Derivatives with respect to constants are known to be identically zero. Hence, their type should remain unchanged meaning that the changes that have already happened need to be reversed. Both `X` and `Y` should have their types changed in the following example.

```
REAL(KIND=2), DIMENSION(2)   :: X, P, Y
...
PARAMETER( P = 3.14 )
```

Initially the type of `P` is also changed to `compad_type` leading to the (edited) parse tree on the left of Figure 5. Note, that the original declaration (its parse tree) is kept in a `COMMENT` node for potential undoing of the type change. When the parser finds the `PARAMETER` statement it backtracks to the original declaration of `P` to generate a separate declaration based on the subtree of the `COMMENT` node. The result is shown in Figure 5 on the right.

Fortran offers a large variety of syntactical constructs making the development of robust parse tree manipulation algorithms a major technical challenge. We have shown a very simple example. Still we are able to handle more compilcated situations including higher-rank array declarations and additional attributes such as `SAVE` and `ALLOCATABLE` as well.

4.3 Support for Parse Tree Manipulation

The adjoint parse tree manipulation algorithm is an extension of our original approach. That used association *by name* for augmenting the original memory space with adjoint variables, that is, v and $\bar{v}$ were two separate variables of type

```
|-TYPE                    |-TYPE
  |-USERTYPE                |-USERTYPE
  | |-NAME=COMPAD_TYPE      | |-NAME=COMPAD_TYPE
  | |-COMMENT               | |-COMMENT
  |   |-REAL                |   |-REAL
  |   | |-PAIR              |   | |-PAIR
  |   |   |-ID=KIND         |   |   |-ID=KIND
  |   |   |-ICONST=2        |   |   |-ICONST=2
  |   |-ICONST=2            |   |-ICONST=2
  |   |-ICONST=2            |   |-ICONST=2
  |-DIMENSION               |-DIMENSION
  | |-E_BOUND               | |-E_BOUND
  |   |-ICONST=2            |   |-ICONST=2
  |-NAME=X                  |-NAME=X
  |-NAME=P                  |-NAME=Y
  |-NAME=Y                |-TYPE
                            |-REAL
                            | |-PAIR
                            |   |-ID=KIND
                            |   |-ICONST=2
                            |-DIMENSION
                            | |-E_BOUND
                            |   |-|ICONST=2
                            |-NAME=P
```

Fig. 5. Undoing Type Changes

`double precision` linked via some naming convention, for example `v` and `_v`. See [8] for details.

Our new method uses association *by address.* Moreover, the components of the corresponding derived type can be of derived type themselves, thus allowing for overloading to be applied to the adjoint code as described above.

5 A Little Mathematical Background

The following is a gentle introduction to the Truncated Newton algorithm aimed at those without an extensive background in numerical analysis.

Suppose that some smooth function f has a vector $\mathbf{x}$ of inputs and a scalar output y, and that we want to find the value of $\mathbf{x}$ for which $y = f(\mathbf{x})$ is a minimum.

Probably the most famous way of doing this is Newton's method, which consists of evaluating the gradient $\mathbf{g} = f'$ and Hessian $H = \mathbf{g}' = f''$ of f at a trial point $\mathbf{x}$ and solving the linear equations $H\mathbf{d} = -\mathbf{g}$ for $\mathbf{d}$. Provided that lots of assumptions are satisfied, this gives the value of $\mathbf{d}$ which minimizes the quadratic approximation $q(\mathbf{d}) = f(\mathbf{x}) + \mathbf{g}\cdot\mathbf{d} + \frac{1}{2}\mathbf{d}H\mathbf{d}$ to $f(\mathbf{x}+\mathbf{d})$, and $\mathbf{x}+\mathbf{d}$ will be a better approximation to the optimal value of f than was the starting point $\mathbf{x}$. This process is then repeated iteratively.

Newton's method has two drawbacks: it tends to be distracted by saddle points unless we start off close to the minimum, and the labour of evaluating the complete Hessian and then solving a huge system of equations at every iteration is so immense as to be impractical if $\mathbf{x}$ has a very large number of dimensions.

At the other extreme, a simple approach which is guaranteed to work eventually is to follow the gradient (like walking downhill) by setting $\mathbf{d}$ to be some multiple of $-\mathbf{g}$. This method, which is called "steepest descent", converges terribly slowly in practice, because the gradient (and hence the correct direction in which to walk) changes continuously as we go along. Really we should adjust the downhill direction $\mathbf{d} = -\mathbf{g}$ by a multiple of $-H\mathbf{d} = H\mathbf{g}$ to allow for the change in gradient caused by the curvature as we move along, and then by a multiple of $H^2\mathbf{d} = -H^2\mathbf{g}$ to allow for the effect on the gradient of the first adjustment, and so on.

Truncated Newton (TN) is a way of iteratively approximating the solution to Newton's equation $H\mathbf{d} = -\mathbf{g}$ for a given "outer" step, by a series $\mathbf{d}_i$ of "inner" steps. These inner steps exactly minimize $q(\mathbf{d}_i)$ in the search space spanned by $\{\mathbf{g}, H\mathbf{g}, H^2\mathbf{g}, \ldots, H^{i-1}\mathbf{g}\}$. The inner iterations are stopped when the approximation $\mathbf{d}_i$ to the Newton step is "close enough". As normal, a line search along $\mathbf{d}$ is then performed before the next outer step in order to ensure global convergence.

The TN approach has several advantages over the full Newton method. First, it can avoid saddlepoints by detecting directions of negative curvature (search directions in which f is concave instead of convex). Second, it does not require the full Hessian, but only directional Hessians (vectors of the form $H\mathbf{p}$), which can be efficiently and accurately calculated by the second-order adjoint codes introduced previously, even for very large dimensions of $\mathbf{x}$. Thirdly, TN can automatically exploit symmetries in the function f: the number of inner iterations required in theory for exact convergence is not the number of dimensions of $\mathbf{x}$, but the number of distinct eigenvalues of H, which is often much smaller.

The reason for this is that, if $\lambda_1, \lambda_2 \ldots \lambda_k$ are the distinct eigenvalues, then $(H - \lambda_1 I)(H - \lambda_2 I) \ldots (H - \lambda_k I) = 0$, so $H^k\mathbf{g}$ is already a linear combination of lower powers.

Even where the number of distinct eigenvalues of H is large, approximate equalities between them often lead in a smaller number of inner iterations to a good enough approximate Newton solution.

It turns out that an effective way to construct the TN sequence of inner approximations $\mathbf{d}_i$ is via a series of two-dimensional searches on q. At the i−th stage the two search directions (starting from $\mathbf{d}_i$) in which q is minimized are the previous direction of movement $\mathbf{d}_i - \mathbf{d}_{i-1}$, which we can think of as a momentum term, and the direction opposite to the updated gradient $\mathbf{g}_i = \mathbf{g} + H\mathbf{d}_i$ at $\mathbf{d}_i$, which is the steepest descent direction for q (not f) at the current point $\mathbf{d}_i$. (Indeed one way to think of the TN algorithm is as the Conjugate Gradient algorithm applied to q instead of f.) The TN algorithm starts with a one-dimensional search in the steepest descent direction $-\mathbf{g}$. Initialize i to zero.

$$\mathbf{d}_0 = 0; \quad \mathbf{g}_0 = \mathbf{g}; \quad \Delta_0 = 0; \quad \mathbf{p}_0 = -\mathbf{g}_0$$

Now for each i form $H\mathbf{p}_i$ and stop if $\mathbf{p}_i H \mathbf{p}_i < \epsilon \mathbf{p}_i^2$

$$\begin{aligned}
\beta_i &= \frac{\mathbf{g}_i^2}{\mathbf{p}_i H \mathbf{p}_i} \\
\mathbf{d}_{i+1} &= \mathbf{d}_i + \beta_i \mathbf{p}_i \\
\mathbf{g}_{i+1} &= \mathbf{g}_i + \beta_i H \mathbf{p}_i \\
\Delta_{i+1} &= \Delta_i + \beta_i^2 \mathbf{p}_i H \mathbf{p}_i / 2
\end{aligned}$$

stop if $\mathbf{g}_{i+1}^2/\mathbf{g}_0^2 < \min(\mathbf{g}_0^2, 1.0/K^2)$

$$\alpha_{i+1} = \mathbf{g}_{i+1}^2/\mathbf{g}_i^2; \quad \mathbf{p}_{i+1} = \alpha_{i+1}\mathbf{p}_i - \mathbf{g}_{i+1}$$

Increment i.

The coefficients α_i, β_i are chosen so that $\mathbf{g}_{i+1}$ is orthogonal to both $\mathbf{g}_i$ and $\mathbf{p}_{i-1}$. This ensures that the search directions $\mathbf{p}_i$ generated by the TN algorithm are conjugate, $\mathbf{p}_i H \mathbf{p}_j = 0$ for $i \neq j$, and that the gradients $\mathbf{g}_i$ are orthogonal, $\mathbf{g}_i \cdot \mathbf{g}_j = 0$ for $i \neq j$. With respect to the partial orthogonal basis $\mathbf{g}_i$, the Hessian H is tridiagonal, since $\mathbf{g}_i H \mathbf{g}_j = 0$ for $i > j+1$.

If TN stops because $\mathbf{p}_i H \mathbf{p}_i/\mathbf{p}_i^2$ is close to zero then we take $\mathbf{d}_i$ as the step direction for the outer iteration. If TN stops because $\mathbf{p}_i H \mathbf{p}_i/\mathbf{p}_i^2$ is significantly negative then we have found a direction of negative curvature, and take the descent direction $\mathbf{p}_i$ as the step direction for the outer iteration.

If TN stops because $\mathbf{g}_{i+1}$ is small (Newton itself is only quadratically convergent, so there is no point in trying to do better than this) then we know that H is positive definite in the space spanned by the $\mathbf{g}_i$. (Any linear combination of the $\mathbf{g}_i$ can be written as $\mathbf{d} = \sum_i \gamma_i \mathbf{p}_i$ whence by conjugacy $\mathbf{d}H\mathbf{d} = \sum_i \gamma_i^2 \mathbf{p}_i H \mathbf{p}_i$.) Note that $\mathbf{g}_i^2$ is not guaranteed to decrease at each step, although q does decrease at each step, by the amount $\beta_i^2 \mathbf{p}_i H \mathbf{p}_i/2$. (This is because $\mathbf{g}_0 \cdot \mathbf{p}_i = -\mathbf{g}_i^2$, whence $\mathbf{g}_0 \cdot \beta_i \mathbf{p}_i = -\beta_i^2 \mathbf{p}_i H \mathbf{p}_i$.) Hence $\mathbf{d}_i$ is a descent direction, with $\mathbf{g}_0 \cdot \mathbf{d}_i = -\mathbf{d}_i H \mathbf{d}_i$.

Another condition under which it is desirable to stop the inner iterations is when $\mathbf{d}_i$ moves into a region where the quadratic model q breaks down. One way to detect this is periodically to compare $f(\mathbf{x}) - f(\mathbf{x}+\mathbf{d}_i)$ with Δ_i and stop if the relative error is more than a specified proportion. Alternatively $\mathbf{g}(\mathbf{x}+\mathbf{d}_i)$ can be periodically compared with $\mathbf{g}_i$ in some norm. This breakdown could be due to $\mathbf{d}_i$ growing too large (trust regions are sometimes used in conjunction with TN to monitor this) but it could also be due to the accumulation of roundoff errors in the values for $\mathbf{g}_i, \mathbf{p}_i$. The TN algorithm responds well when provided with accurate second derivatives, which allow a much larger number of inner iterations to be taken before this second form of breakdown occurs.

For more information about the TN algorithm, the best place to begin is the original paper by Dembo and Steihaug [4].

6 Case Study

We consider the application of the Truncated Newton algorithm to a Fortran implementation of the well-known Rosenbrock function

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2 \quad .$$

The Hatfield NOC OPTIMA TN algorithm expects the user to provide the following three subroutines:

- CALFUN to compute the function value FF at point X;
- CALGRD to compute the gradient G at point X;
- CALHP to compute the product HP of the Hessian at point X with a vector P.

Both CALGRD and CALHP use the second-order adjoint routine dbRosenbrock that is generated automatically by the compiler. All three routines are shown in the following code listing.

```
SUBROUTINE CALFUN(X,N,FF,INF)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(N)
INF=0
t1 = (1-x(1))
t2 = t1*t1
t3 = x(2)-x(1)*x(1)
FF  = t2 + 100*t3*t3
RETURN
END

SUBROUTINE CALGRD(X,N,FF,G,INF)
use compad_module
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
external dbRosenbrock
DIMENSION X(N),G(N)
type(compad_type) ctx(N), ctf
INF=0

ctx(1)%val%v = X(1)
ctx(1)%val%d = 0.
ctx(1)%drv%v = 0.
ctx(1)%drv%d = 0.

ctx(2)%val%v = X(2)
ctx(2)%val%d = 0.
ctx(2)%drv%v = 0.
ctx(2)%drv%d = 0.

ctf%val%v = 0.
ctf%val%d = 0.
ctf%drv%v = 1.
ctf%drv%d = 0.

call dbRosenbrock(ctx, ctf)

G(1) = ctx(1)%drv%v
G(2) = ctx(2)%drv%v

RETURN
END

SUBROUTINE CALHP(X,G,N,P,HP,INF)
use compad_module
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
external dbRosenbrock
DIMENSION X(N),G(N),P(N),HP(N)
type(compad_type) ctx(N), ctf
```

```
49 INF=0
50
51 ctx(1)%val%v = X(1)
52 ctx(1)%val%d = P(1)
53 ctx(1)%drv%v = 0.
54 ctx(1)%drv%d = 0.
55
56 ctx(2)%val%v = X(2)
57 ctx(2)%val%d = P(2)
58 ctx(2)%drv%v = 0.
59 ctx(2)%drv%d = 0.
60
61 ctf%val%v = 0.
62 ctf%val%d = 0.
63 ctf%drv%v = 1.
64 ctf%drv%d = 0.
65
66 call dbRosenbrock(ctx, ctf)
67
68 HP(1) = ctx(1)%drv%d
69 HP(2) = ctx(2)%drv%d
70
71 RETURN
72 END
```

The gradient at the current point X (lines 20 and 25) is computed by setting the adjoint of FF to one (line 32). Its entries are accumulated in the adjoint components of X (lines 37 and 38).

The product of the Hessian with the vector P at point X (lines 51 and 56) is computed by initializing the directional derivatives of X(1) and X(2) to P(1) (line 52) and P(2) (line 57), respectively. Again, the adjoint of FF needs to be set to one (line 63). The entries of the corresponding directional Hessian occur as the directional derivatives of the adjoints of X (lines 68 and 69).

The semantic transformation of the original function evaluation routine is taken over entirely by the compiler. Writing corresponding code by hand and debugging it may require substantial manpower for larger problems. Undoubtedly, life is much easier with a second-order adjoint compiler.

7 Conclusion and Outlook

Writing a compiler for the automatic generation of robust and efficient adjoint and second-order adjoint codes is a highly challenging task. Substantial source code analysis and manipulation is required at the level of the compiler's internal representation. The availability of such a compiler facilitates a faster transition form pure numerical simulation to derivative-based optimization of nonlinear mathematical models. Thus, the compiler is likely to become a central element in the toolbox of every computational scientist or engineer.

We are in the process of developing a research prototype of the NAGWare Fortran compiler that provides the previously mentioned functionalities. Our current test set comprises problems from the Minpack test problem collection as well as several others. Feasibilty and usefulness of the adjoint compiler approach has been illustrated with the help of a Truncated Newton algorithm for unconstrained nonlinear optimization where gradients and projected Hessians are provided automatically by the compiler for a given implementation of the objective function. The efficient gradient evaluation via adjoints combined with the accuracy of the obtained numerical values resulted in a speedup of an order of maginitude.

Further work is necessary to achieve the level of robustness that is desirable in an industrial setting. Most likely this highliy ambitious software development task cannot be completed in a purely academic environment where incremental progress in language coverage is achieved by considering new target applications as they become relevant.

References

1. B. Averik, R. Carter, and J. More. The Minpack-2 test problem collection (preliminary version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1991.
2. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, 2005.
3. M. Cohen, U. Naumann, and J. Riehme. Towards differentiation-enabled Fortran 95 compiler technology. In *Proceedings of the 18th ACM Symposium on Applied Computing, Melbourne, Florida, USA, March 9–12, 2003*, pages 143–147, 2003.
4. . Dembo and T. Steihaug. Truncated-newton algorithms for large-scale optimization. *Math. Prog.*, 26:190–212, 1982.
5. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24:437–474, 1998.
6. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. SIAM, Apr. 2000.
7. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
8. M. Maier and U. Naumann. Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In G. Montero B.H.V. Topping and R. Montenegro, editors, *Proceedings of the Fifth International Conference on Engineering Computational Technology*, page paper 112. Civil-Comp Press, Kippen, Stirlingshire, United Kingdom, 2006.
9. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In *[2]*. 2005.
10. U. Naumann and J. Utke. Source templates for the automatic generation of adjoint code through static call graph reversal. In P. Sloot et al., editor, *Computational Science - ICCS 2005, Proceedings of the International Conference on Computational Science, Atlanta, GA, USA, May 22-25, 2005, Part I*, volume 3514 of *LNCS*, pages 338–346. Springer, 2005.
11. J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the intraprocedural flow of control in adjoint computations. *Journal of Systems and Software*, 79:1280–1294, 2006.