Argonne NATIONAL LABORATORY

Identifying Active Variables to Improve the Performance of Operator Overloading Automatic Differentiation

Drew Wicke, Sri Hari Krishna Narayanan, and Paul Hovland

Introduction

Automatic differentiation (AD) is a technique of computing the derivative of given source code. AD can be implemented by the operator overloading or source-to-source transformation approach.

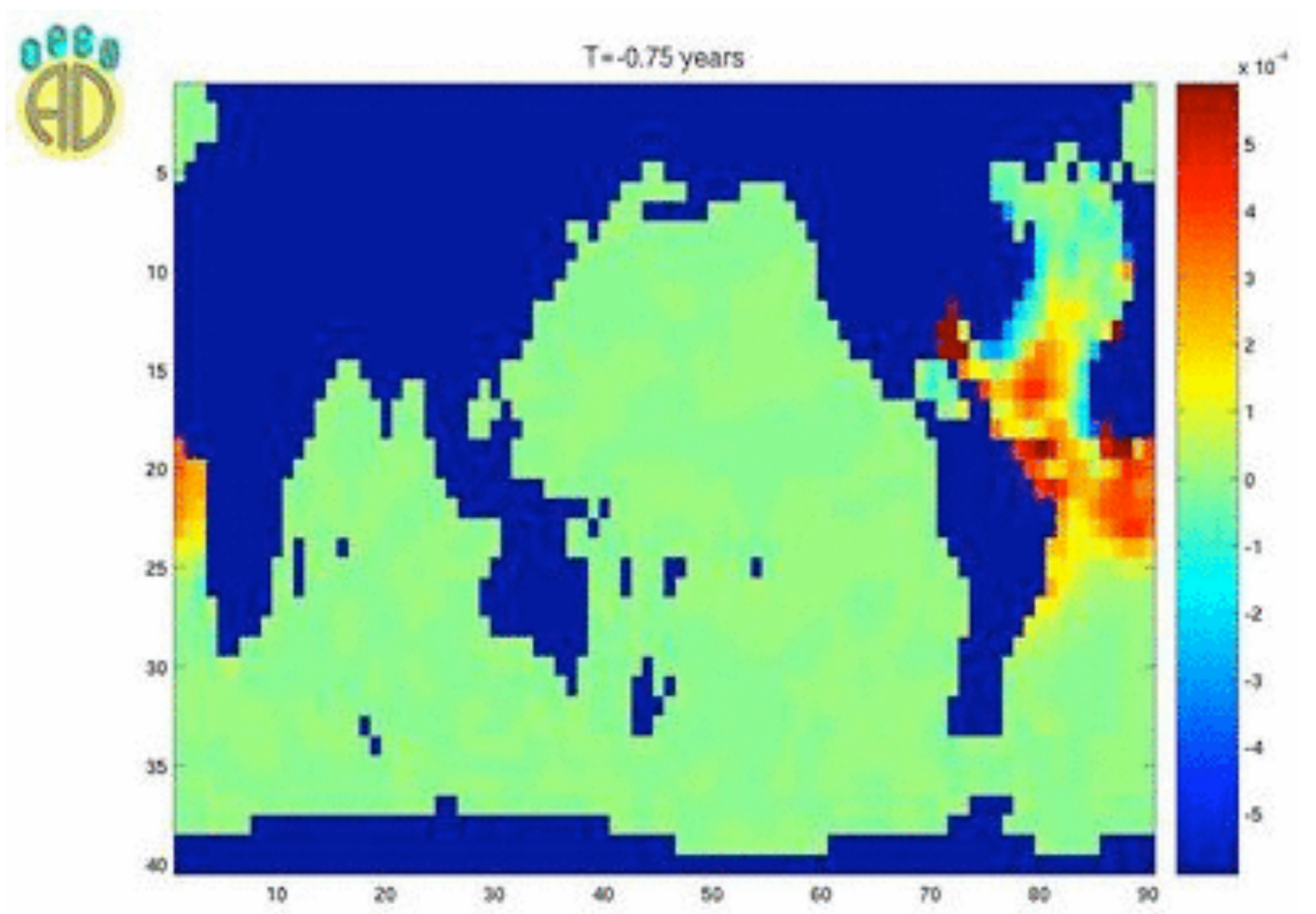


Figure 1: A snapshot of the sensitivity map of the heat transport in the north atlantic to temperature in a depth of 1590 meters over a period of 10 years going backwards in time.
Source: http://www.mcs.anl.gov/OpenAD/

Operator Overloading AD

Uses features of the programming language to alter the semantics of mathematical operators in order to compute the derivative.

Operator overloading allows for maintainable code; however, speed of computation is sacrificed. The goal of this work is to use activity analysis to improve the performance of the calculation of derivatives using Sacado.

```
DERIV_TYPE operator* (const DERIV_TYPE
                                    &other) {
   this->val = this->val() * other.val();
   this->dx = this->val() * other.dx() +
        other->val() * this->dx();
   return *this;
}
```

Figure 2: Example of an overloaded multiplication operator. Not only must the product be computed, but also the value of the derivative after applying the product rule.

Sacado is a package in the Trilinos framework and is an implementation in C++ of the operator overloading method of AD.

Use of Sacado Derivative Type

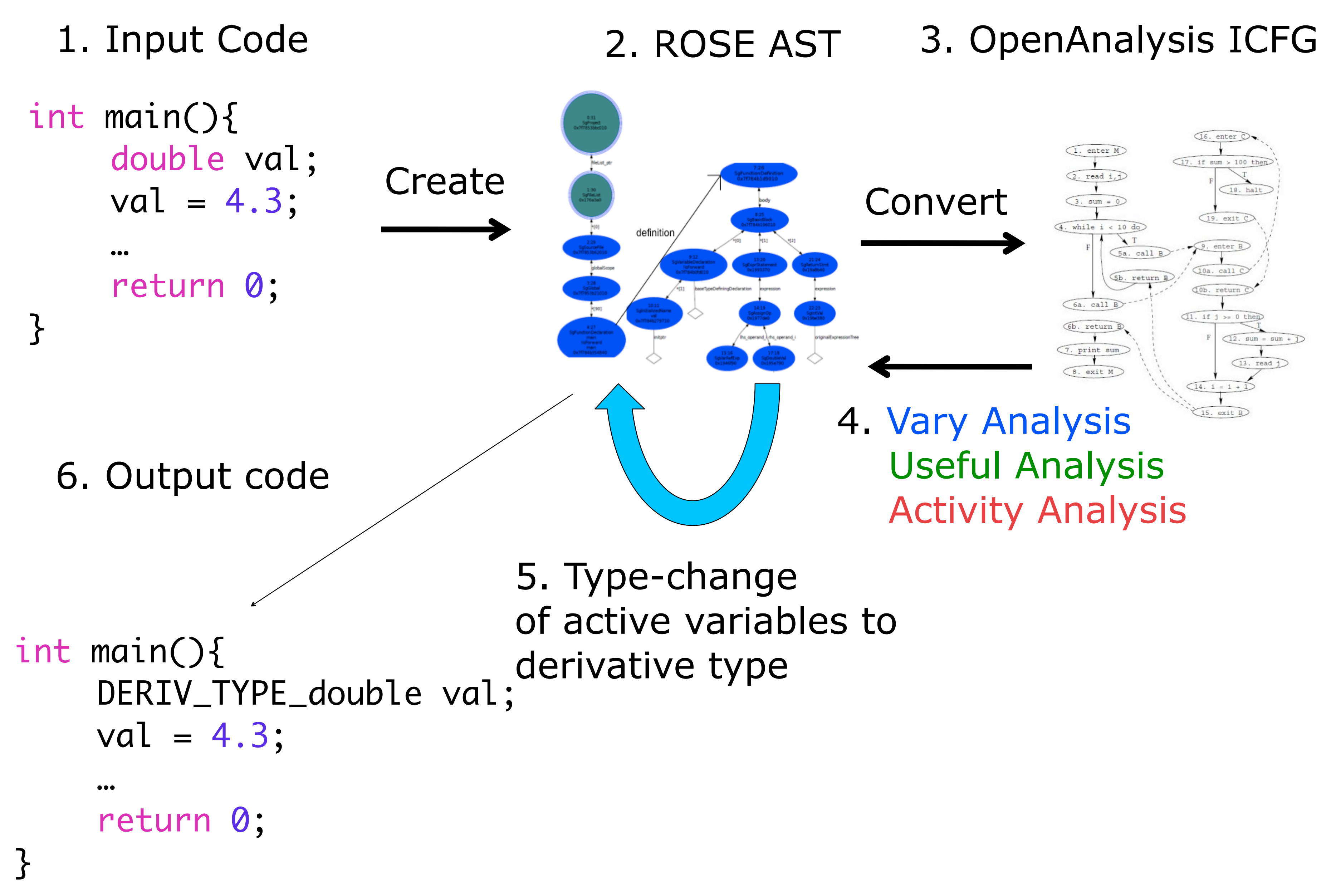
- •Vary variables are those whose value is computed using an independent variable.
- •Useful variables are used to compute the value of the dependent variable
- •Active variables are both vary and useful.

```
typedef Sacado::Fad::DFad<double> DERIV_TYPE_double;
//x-independent, f-dependent
void foo(DERIV_TYPE_double *x, DERIV_TYPE_double *f){
    DERIV_TYPE_double squarePoly = (*x) * (*x);
    DERIV_TYPE_double theConst = 3.14 * 2;
    DERIV_TYPE_double div = ((*x) / 2);

    (*f) = div * theConst;
}
```

Figure 3: The code illustrates how a function can be written to be differentiated by Sacado. All the variables are of the Sacado derivative type DERIV_TYPE_double. Both the function and the derivatives are computed using overloaded operators when the variables have the Sacado derivative type. Since only active variables need be of derivative type, the code is inefficient due to unnecessary memory allocation and overloaded function calls.

Activity Analysis Tool Flow



```
int main(){
    double val;
    val = 4.3;
    …
    return 0;
}
```

```
int main(){
    DERIV_TYPE_double val;
    val = 4.3;
    …
    return 0;
}
```

Figure 4: Process through which our tool changes the types of active variables.
1. Specify input source.
2. Create a ROSE abstract syntax tree (AST).
3. Convert AST to a interprocedural control flow graph (ICFG) within OpenAnalysis.
4. Using OpenAnalysis, perform vary, useful, and activity analysis to make a list of active variables.
5. Change the active variable's type to the Sacado derivative type within the AST.
6. Generate output code.

Sample Output

```
typedef Sacado::Fad::DFad<double> DERIV_TYPE_double;
void foo(DERIV_TYPE_double *x, DERIV_TYPE_double *f)
{
# pragma $indep,x // specify the independent
# pragma $dep,f   // and dependent variables
    double squarePoly = ADValue((*x) * (*x));
    double theConst = 3.14 * 2;
    DERIV_TYPE_double div = ((*x) / 2);

    (*f) = div * theConst;
}
```

Figure 5: This code is efficient because only the active variable is of the Sacado derivative type, DERIV_TYPE_double.

Challenges

We encountered challenges in type changing. For example, to change the type of typedef-ed active variables we must extract base type to set to the derivative type.

```
// before:                   // after:
typedef double* fir;         typedef double* fir;
typedef fir* sec;            typedef fir* sec;
sec test; // active          DERIV_TYPE_double **test;
```

Figure 6: Code shows how double and pointer data types are gathered to change the type of test to DERIV_TYPE_double**.

Conclusion/Future Steps

Our tool:

- ●can be used with any operator overloading AD package
- ●replaces the manual process, which is slow and likely to overestimate the number active variables
- ●successfully identifies active variables
- ●can change the data type of active variables of most C data types

In the future we would like to:

- ●gather performance data
- ●fully support C++

References:
Automatic differentiation: http://www.autodiff.org/
OpenAnalysis: http://openanalysis.berlios.de/
ROSE: http://www.rosecompiler.org/
Sacado: http://trilinos.sandia.gov/