# Data Representation Alternatives in Semantically Augmented Numerical Models

Michael Fagan
Rice University, 6100 Main Street, Houston, TX 77005, USA
mfagan@cs.rice.edu
Laurent Hascoet
INRIA, 2004 Route des lucioles, 06901 Sophia-Antipolis, France
laurent.hascoet@sophia.inria.fr
Jean Utke
Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA
utke@mcs.anl.gov

## Abstract

*Transformations of numerical source code may require the augmentation of the original variables with new data to represent additional data the transformed program operates on. Automatic differentiation makes extensive use of this concept. We describe the two principal approaches to implement the variable augmentation, complete encapsulation and complete separation. The paper concentrates on two major aspects. First, we characterize the advantages of each approach and illustrate the effort needed to realize these advantages in Fortran, C, and C++ as the languages we are most interested in. Second, we discuss the practical solutions that in effect represent hybrids of the two approaches.* [1]

## 1. Introduction

Many problems in engineering and science require the use of numerical models to simulate and optimize complex behavior. For practical computations these numerical models are implemented in a programming language such as Fortran, C, or C++ or in a system such as Matlab. Taking advantage of the numerical model representation as a program, we can extend it to gain insight into the model behavior that is beyond the plain computation of the model itself. Examples are the computation of uncertainty information, guaranteed bounds, envelope representations, and last but not least derivatives. For this paper the important commonality of these program transformations is that we *extend* the semantics; in other words the original model computation is preserved. To differentiate from program transformations that simplify the model or do not preserve the original semantics, we call them *augmenting transformations*. Because they extend the semantics, we have to find a way to relate the data in the augmenting code to the data in the original code. This paper looks at this data augmentation issue as a problem faced by all augmenting transformations of numerical programs. We investigate the problem in the presence of programming language features[2] that impact the augmentation, and we describe a range of solutions. The computation of derivatives with automatic differentiation (AD), in particular the generation of an adjoint code [7], is by far the most complicated augmentation of numerical programs. The solution of the data augmentation problem is essential for any implementation of AD, and therefore its practical relevance is at least at the same level as that of AD. The examples used here stem from an AD context, but we emphasize that the issues they illustrate are not specific to AD.

Throughout the paper we refer to a program $P$ that represents the numerical model $(\boldsymbol{y},\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{p})$ of interest. We consider as input $n$ independent variables $x_i$ and as output $m$ dependent variables $y_j$. In many practical cases there can also be $s$ parameters $p_k$ that affect the computation of $\boldsymbol{f}$, and symmetrically $\boldsymbol{f}$ may modify $t$ variables $q_l$. We observe that many practical applications perform simula-

[1]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38 and by NSF under ITR contract OCE-0205590.

[2]Because of the lack of practical relevance of functional languages for numerical models, we will consider only imperative programming languages.

tions and optimizations where the $x_i$ and $y_i$ vary but the $p_i$ are fixed and the $q_i$ hold no practical relevant information. An example for this split is a fluid dynamics model where the material constants are part of $\boldsymbol{p}$ and the velocities of interest are observed only at certain points in the discretized space. This split is exploitable for data augmentation; see Sec. 2. The augmenting transformation turns $P$, that is, $(\boldsymbol{y},\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{p})$, into $P^*$, that is, $(\boldsymbol{y}^*,\boldsymbol{q}) = \boldsymbol{f}^*(\boldsymbol{x}^*,\boldsymbol{p})$, where the $x_i^* = (x_i, x_i')$ and $y_j^* = (y_j, y_j')$ contain the original $x_i$ and $y_j$ plus the augmenting data $x_i'$ and $y_j'$ (e.g., uncertainties, bounds, or derivatives). The modified $\boldsymbol{f}^*$ is implemented by the original $P$ extended by code $P'$ that propagates the augmenting data. We point out that language features that obfuscate the semantics of a given piece of memory are inherently detrimental to automatic analysis and transformation and are therefore excluded from our consideration. Most prominent among these are Fortran's `equivalence`, C/C++'s `union`, and recasting of raw memory to pointers to unrelated types.

## 2. Limiting the Data Augmentation

A naive approach to data augmentation would be to pair each variable $v$ that occurs in the variable space of the program $P$ with an augmenting $v'$. The ensuing increase of the memory footprint may be unbearable, however, even for modest practical applications. Therefore only the variables that need augmentation, which we call *active*, should be augmented. For instance, in AD a variable $v$ is active iff at some time during the computation of $f$ its value depends on the value of any of the $x_i$ and the value of any of the $y_j$ depend on $v$. Activity is determined by activity analysis [5] and in practice can be implemented in a compiler-style data flow analysis. In most practical applications, the active set cannot be determined exactly because of aliasing of variables [9]; what is computed is a conservative overestimate. Exploiting activity analysis is comparable to program slicing known from compiler optimization [17]: only active variables are augmented, and the augmenting transformation needs to add code corresponding only to numerical operations involving active variables. The implied runtime savings for computing $\boldsymbol{f}^*$ are crucial.

## 3. Principal Approaches

Section 1 introduced our target, the augmentation of $P$ with additional data and operations (e.g., derivatives and their computation). This augmentation is clear enough from a semantics point of view. From a practical implementation point of view, however, many options and strategies can be employed. For example, consider the varying strategies used by many AD tools. Comparisons are difficult because of a lack of common vocabulary and concepts. We want to classify these strategies, study their strengths and weaknesses, and compare existing tools according to this classification.

Our given original program $P$ has its own data, operational, and call structure. Our augmenting transformations will introduce new data structures to hold the augmenting data $v'$ and new operational and call structures to compute these augmenting values. Here operational structures consist of arithmetic operations and intrinsics but also comparison operations used in the control flow. Call structure denotes for the most part the call graph, but here we will also look at the consequences of overloading if this language feature is present. The central question is how to associate the new augmenting structures with the original ones. In the following sections we will characterize two extreme approaches, for brevity called

1. complete encapsulation and
2. complete separation,

and we will explore the range between these two extremes.

### 3.1 Complete Encapsulation

Data augmentation needs to create a $v'$ for each active $v$ occurring in $P$. A natural choice is to associate $v$ and $v'$ by encapsulating them as a pair into a (new) type. For instance in C++ we might define

```
struct aFloat{ float o; float a;}
```

and redeclare active `float` variables `v` to be of type `aFloat`, where `v.o` is the original value and `v.a` the augmented value. Figure 1 shows complete encapsulation

```
#include <math.h>
class Ċ {
private:
struct Ṡ {
 aFloat u;
 float p;} s;
public:
void f̂(aFloat x, aFloat y){
  aFloat t;
  t=̂poŵ(x,2);
  s.u=̂s.u+̂t;
  y=̂2*̂x*̂s.u+̂s.p;}
};
```

**Figure 1. Example of complete encapsulation. The transformations occur explicitly (boxed) at the active leaves of the data structures, that have been changed from `float` to `aFloat`, and implicitly (capped) for the elementary operations via overloading.**

on a short code example. Not all programming languages provide user-defined structures: an obvious counterexample is Fortran77 which despite of its age is still in heavy use because of the existence of compilers with superior optimization capabilities. While of limited applicability there are several practical tricks to emulate a simple pairing, for example using the built-in `complex` type [15] or using arrays as we propose in Sec. 5.

Concerning the operational structure, we need to change all operations involving active variables `v` to perform the original operation on the `v.o` component, as well as the augmenting operations on the `v.a` component, possibly involving `v.o`, too. The obvious choice is to encapsulate the augmenting and original operations in new (overloaded) operations, shown capped in Fig. 1. The application of the chain rule in AD is a simple example, so the `pow` operation could be overloaded to compute derivatives as in

```
#include <math.h>
aFloat pow(const aFloat& m, int e){
 aFloat r;
 r.a=e*pow(m.o,e-1)*m.a; //augmenting
 r.o=pow(m.o,e); return r;}
```

Because of the signature change to `f`, overloading applies to $\widehat{\mathtt{f}}$, too, and therefore is shown capped. When overloading is not supported by the language, the augmented $\widehat{\mathtt{f}}$ may need to be renamed if one wants to retain the original code for `f` as well. Often, however, there is only a single augmented version $\widehat{\mathtt{f}}$, and neither the original `f` nor for that matter the original `C` and `S` are needed, so the augmented $\dot{\mathtt{C}}$ and $\dot{\mathtt{S}}$ that might otherwise be subject to renaming could in fact reuse the old names.

Overloading-based augmentation requires explicit modifications only on the atomic leaves of the data types. Several AD tools (e.g. Adol-C [3]) employ this strategy. However, we will see in Sec. 4.2 that we may want several specialized augmented versions of procedures and types in order to gain efficiency. In that respect the original `f`, `C`, and `S` may still be useful in the augmented code. Then renaming of the augmented $\dot{\mathtt{C}}$ and $\dot{\mathtt{S}}$ is mandatory, and the augmentation requires a more intrusive rewriting of the program.

### 3.2 Separating Operations

The strategy described in the previous section has the advantage of elegantly reusing nearly all of the program's structure. However, elegance doesn't necessarily imply efficiency. First, although the compiler implementation of overloading is improving, overloaded operators that encapsulate sufficiently complex code will not be inlined and therefore incur overhead through the call. Second, and much more important, the nature of the transformations concentrating on *numerical* operations almost always allows code optimizations based on the computational graph [4] that is implied by a given execution sequence of elemental operations [18]. Examples are constraint propagation [14] and from AD the cross-country elimination approach [8, 10]. Restricting the transformation to individual elementals as done with overloading precludes higher-level optimized transformations.[3] Finally, again in the AD context, some popular code transformations (e.g., the "adjoint code") need to handle control flow and call graph augmentations [11] and therefore are clearly beyond the scope of elemental operations.

To address the obvious drawback of the overloading approach, instead of complete encapsulation we start by separating augmenting from original operations. If done in a straightforward fashion without going into the optimizations mentioned in the previous paragraph, this process amounts to inlining of the overloaded operations of Sec. 3.1 turning `t=pow(x,2)` into an augmenting statement `t.a=2*x.o*x.a`[4] and a statement for the original semantics `t.o=pow(x.o,2)`. Now the transformation has to explicitly generate operations in terms of the components of the active type. Figure 2 shows the example from Fig. 1 with operation separation. Unlike the operational structure, the data structure remains the same as in the previous section. Since the augmenting data are stored only a fixed offset away from the original data, it is appropriate to call this strategy *association by address*. Because

```
#include <math.h>
class Ċ {
private:
struct Ṡ {
 aFloat u;
 float p;} s;
public:
void f̂(aFloat x, aFloat y){
  aFloat t;
  t.a=2*x.o*x.a;
  t.o=pow(x.o,2);
  s.u.a=s.u.a+t.a;
  s.u.o=s.u.o+t.o;
  y.a=2*(x.a*s.u.o+x.o*s.u.a);
  y.o=2*x.o*s.u.o+s.p;}
};
```

**Figure 2. Separating operations. Compared to the code in Fig. 1, augmenting data are kept encapsulated, but augmenting operations are separated from original operations.**

[3] Operators can be overloaded to create computational graphs on the fly, but that moves the transformation optimization away from compile time to run time, a move that is rarely beneficial.

[4] Recognizing the constant exponent 2 and avoiding the second call to `pow` is a little side benefit of this more context aware transformation.

it still uses the active type, it does not add any new variables. This is a significant advantage it retains from the all-encapsulating approach introduced in Sec. 3.1 because, as a consequence, there is *no extra cost for address computations* relating to the augmenting data. In other words, any array and structure dereferences or pointer arithmetic occurring for the original data implies that the corresponding augmenting data can be accessed via a fixed offset that is computed at compile time. The difference to additional address computations necessary at run time will become clear in the following sections.

A second and no less significant advantage is that while signatures of procedures may change individual parameter types to the active type, any *overloading resolution mechanism* known, for instance, in C++ and Fortran9x *remains invariant* with respect to the type changes in the signature, provided the active types properly emulate the built-in conversions for the basic numerical types in the programming language.

## 3.3 Separating Plain Data

Section 1 points out that user-defined types may not necessarily be supported, as for instance in Fortran77. At the same time, Fortran77 for instance does not have the concept of explicit pointer arithmetic[5] or overloading. Therefore one need not be concerned about losing either of the advantages pointed out at the end of Sec. 3.2. In such a context, thinking about the augmenting data as separate from the original data is quite natural, too. Rather than encapsulating data in an active type, one creates *new* augmenting variables, for any active original variable. The new variable is given a name derived from the name of the corresponding original variable following some convention. When the original procedure is still in use in the augmented program, the augmented procedure also must be given a new name. We call this strategy *association by name*. To make our point, we turn the example code from the previous sections into a somewhat simplified Fortran77 code shown in Fig. 3. Our naming convention just prepends an a. The complete separation of the data is apparent. From a general source transformation point of view a complete separation of data entails a number of important advantages.

- It separates memory allocation of original data from memory allocation of augmenting data.
- I/O operations remain unaffected.
- Interfacing with external (black box) routines and system calls remain unaffected.
- The eventual, manual changes necessary in the environment calling the transformed code (i.e., the "driver") are simple and explicit.

[5] However, index computations for array dereferences come close.

```
subroutine af(x,y,ax,ay)
real au
common/aS/ au
real u,p
common/S/ u,p
real ax,ay,at
real x,y,t
at=2*x*ax
t=x**2
au=au+at
u=u+t
ay=2*(ax*u+x*au)
y=2*x*u+p
end subroutine
```

**Figure 3. Separation of plain data.**

From a compiler point of view, however, the temporal locality of original and augmented data shown in the transformed example codes is no longer accompanied by spatial locality, which can result in an efficiency loss .

## 3.4 Partial Separation in Structured Data

To get closer to the other extreme strategy of the spectrum, we need to consider the complete separation of the data introduced in Sec. 3.3 for more complicated programming language features. Returning to our example in Fig. 2, one could reasonably attempt the separation shown in Fig. 4. This comes close to the code in Fig. 3. Clearly,

```
#include <math.h>
class Ċ {
private:
struct Ṡ {
 float au;
 float u;
 float p;} s;
public:
void af(float x, float y,
        float ax, float ay){
  float t, at ;
  at=2*x*ax;
  t=pow(x,2);
  s.au=s.au+at;
  s.u=s.u+t;
  ay=2*(ax*s.u+x*s.au);
  y=2*x*s.u+s.p;}
};
```

**Figure 4. Separation of data structures leaves**

however, the data separation applied only to active data structure leaves, as done here, is not complete. Consider, for instance, an array `S* sArr=new S[20]`. The memory of the original and augmenting data is not separate. A better memory separation requires separating the data structures themselves, as shown in Fig. 5, where the augmenting `struct aS` is distinct from the original `struct S`. The

```
#include <math.h>
class Ċ {
private:
struct aS{ float au;} as;
struct S {
 float u;
 float p;} s;
public:
void af(float x, float y,
        float ax, float ay){
  float t, at ;
  at=2*x*ax;
  t=pow(x,2);
  as.au=as.au+at;
  s.u=s.u+t;
  ay=2*(ax*s.u+x*as.au);
  y=2*x*s.u+s.p;}
};
```

**Figure 5. Plain data structures separation**

benefit is that an original allocation of `sArr` can be kept unmodified, separated from the augmenting allocation `aS* asArr=new aS[20]`. This method has to be concerned with overloading resolution. For example, `void f(float p, float q)`, where we assume neither `p` nor `q` is active, and `void f(float x)`, assuming `x` active, are unambiguously resolved. The augmented versions, however, `void f(float p, float q)` vs. `void f(float x, float ax)`, are not. In general for this particular approach with incomplete data separation, the transformation engine has to contain some knowledge about the overloading resolution to avoid these conflicts.

### 3.5 Complete Separation

To realize the advantages mentioned at the end of Sec. 3.3, one has to separate all structures that hold data, and not only simple C-like user-defined types as the `struct S` in our example. However, our C++ example reveals the dilemma right away. Obviously the separation applies to data structure hierarchies, and consequently we want to create a separate `class aC` that contains `struct aS`. At the same time the operations to be augmented are coded in a member function of `class C`, and they, and possibly also the augmenting operations, are using private data members (see the respective last augmenting statement). For our example code a possible

```
#include <math.h>
class aC{
private:
struct aS{ float au;} as;
public:
void af(float x, float y,
        float ax, float ay,
        C& c){
  float t, at ;
  at=2*x*ax;
  t=pow(x,2);
  as.au=as.au+at;
  c.s.u=c.s.u+t;
  ay=2*(ax*c.s.u+x*as.au);
  y=2*x*c.s.u+c.s.p;} };
```

**Figure 6. Complete separation in data structures**

solution is shown in Fig. 6. This requires `class aC` to be declared a `friend` within the definition of `class C` to enable access to the private members of `class C`. Any invocation such as `c.f(x,y)` would be augmented to `ac.af(x,y,ax,ay,c)`. If the `friend` notion is not available, augmentation can still try to release the access control, but in some languages (e.g., Fortran9x) access control and scoping are coupled, making name clashes possible. In general, complete separation becomes harder as the data structures contain more private or static components. We already pointed out that the augmenting code is expected to use numerical values from the original code and therefore a complete separation of the arithmetic is generally impossible. We also point out that the arithmetic operations may reuse not only values from the original data but also control flow conditions. The overloading resolution problem described in Sec. 3.4 persists. We mentioned in Sec. 3.2 the need to replicate address computations. For example, an address computation done for the original data `((&c)+(++i))->f(x,y)` needs to be replicated in the preceding[6] augmenting statement, `((&ac)+i+1)->af(x,y,ax,ay,*((&c)+i+1))`. The conclusion to be drawn from this section is that while complete data separation remains a possibility, the ensuing code transformation becomes vastly more complicated than the transformations introduced in Sec. 3.1 and Sec. 3.2, respectively.

## 4. Impact of the Separation Level

The paper intends to cover all important aspects a given choice of separation levels has on the augmenting transfor-

[6] For consistency reasons we let the augmentation always precede.

mations in the presence of various programming language features. To streamline the introduction of the separation levels in Sec. 3, we tacitly ignored a number of issues that originate from the activity analysis described in Sec. 2 as well as practical concerns and language features.

### 4.1. Transformation of Code Subsets

Section 2 explains the need to approximate the set of active variables as a subset of all variables in the code base. In order to ensure semantic correctness, the static activity analysis has to make conservative assumptions, in particular in the presence of aliasing, which lead to an overestimate of active variable subset. Alias analysis is a prerequisite of activity analysis, and we point out that aliasing also limits the ability to statically construct computational graphs needed for optimized transformations (see Sec. 3.2). Some improvements can be achieved with sophisticated analyses, and statically undecidable problems may be helped with runtime information. Either one can be prohibitively expensive for programs with a large code base, but one still would want to reduce the overestimate. In practice, such large applications often are not transformed in their entirety; instead, only a subset $P^+$ of the program's procedures containing all relevant code for the computation of $(\boldsymbol{y},\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{p})$ (see Sec. 1) is subject to analysis and transformation. The smaller code base reduces the number of conservative assumptions and therefore leads to better approximations of the active variable set, truly aliased variables, and so forth. Aside from the formally inactive subcomputations $(\boldsymbol{q}) = \boldsymbol{f}_a(\boldsymbol{x},\boldsymbol{p})$ and $(\boldsymbol{y},\boldsymbol{q}) = \boldsymbol{f}_b(\boldsymbol{p})$, one should think of the inactive remainder of procedures $P^-$ as diagnostic code, initializations, and data pre- and postprocessing. The split implies consequences for the call structure and the data structure. The data structure issues are addressed in Sec. 4.3. For the call structure we distinguish for a given procedure `s` the following two cases.

First, `s` is called in $P^+$ but defined in $P^-$; in other words `s` is not subject of the transformation. The procedures that are defined in $P^+$ may have their signature changed and may also be renamed. Consequently their call sites will be transformed accordingly. Neither one is the case for the `s` we are concerned with. However, there can be call sites in $P^+$ where the actual parameter is active; a typical example would be a diagnostic or debugging routine that periodically writes data to a log. Complete data separation will always have the original data available to pass as parameters. Incomplete data separation will pose problems, however. Suppose, for instance, that in the fashion of Fig. 4, active leaves are separated, but not active structures. Passing the encapsulated augmented structure to `s` will not pose problems as long as `s` follows the structure accessors. But operations that access raw memory, for instance for binary I/O, will be affected. Any approach using encapsulation within an active type will encounter a type mismatch unless the transformation algorithm inserts pre- or postcall conversions that use a passive temporary. For example, the original code may contain `float p; s(p);` where `p` is active and therefore needs to turn into

```
aFloat p;
float t;
t=p.o; // if p is in the IN set of s
s(t);
p.o=t; // if p is in the OUT set of s
```

where the `IN` and `OUT` sets have the usual data flow meaning. Because of the data copying there is an efficiency concern. Such calls may be infrequent, for instance during initial setup, or diagnostic, which might be disabled. For cases in which they are not active but are integral to the implementation, for instance the writing of checkpoints for restarts, one might have to consider moving `s` into $P^+$ or manual adjustments.

Second, `s` is defined in $P^+$ and called in $P^-$, and presumably also in $P^+$. A typical example may be a cost function computed on the model state at each step in a timestepping scheme but also during the initialization phase on the initial state. For the encapsulation approach with an active type we have the reverse type mismatch, but now there is no transformation applied to $P^-$ to solve the problem. Often one can solve the problem by replicating `s`, renaming the replicant to `as`, applying all augmenting transformation to the definition of `as`, and call `as` in $P^+$ in place of `s`. The caveat associated with this method is explained in Sec. 4.3. For the calls to `s` within $P^+$ there can obviously be cases where the conversion shown in the previous paragraph needs to be reversed, that is, when the formal parameter of `s` is active and the actual is not; see also Sec. 4.2. For data separation approaches we can employ the same replication method; however, the caveat applies.

A related problem is that of *external interfaces*. Unlike the situations considered above, external interfaces denote calls to procedures for which no source code definition is available. At the same time the computations performed are semantically active; that is, they depend on the independent variables and impact the dependents. This situation occurs whenever (third-party) numerical libraries are used and therefore is quite common. With rare exceptions for AD, there is no expectation to have libraries that already contain code implementing the semantics of the augmentation of interest and much less an implementation that matches any given choice of augmenting data representation. Consequently, one will have to use an interface layer that wraps the external calls. Regardless of the choice of data representation in the transformation itself, the wrapper will always have to rely on complete data separation in order to be able to make the original external call. In most cases the augmenting code will have to be manually created and be made part of the wrapper. An activity analysis can yield very large overestimates as the consequence of external calls. One of the conservatively correct assumptions has to be that all formal parameters depend on each other, that is, all actual parameters become active if one is active, when in reality there may only be a few dependencies. Incidentally this can be mitigated by stubbing out the external calls with (simplified) code emulating the actual dependencies or by feeding dependencies signatures into the augmentation tool. In this sense the stubs can be viewed as a natural extension to the wrapper layer.

### 4.2. Activity and Generalization

Activity analysis can be made more or less sensitive to context and control flow, yielding more or less precise activity information. The notion of activity that we have been using in Sec. 3 is context and flow insensitive. In its flow-insensitive form, activity analysis globally marks one variable active if it is active at some point in the flow, for some execution context. The activity of a variable does not change throughout its scope. In its flow-sensitive form, activity analysis computes for each variable an activity status that evolves as the flow advances; for example, in AD a variable becomes inactive after it is reset to a constant. Overapproximation is nevertheless necessary when the control flow merges two flow branches with different activities. Also, when only one element of an array becomes active, then the whole array becomes active, unless the analyzer implements array region analysis [6]. Flow-sensitive activity analysis closely follows the structure of the control flow graph, and its primary benefit is to further limit the number of operations that have to be augmented. Still, a flow-sensitive activity analysis may indicate a considerably smaller scope for the augmenting data than that of the original data. Consider the following example,

```
void f(float x, float y, float p) {
// lots of code with inactive uses of p
if (isFullMoon)  p=x; p=sin(p); y=p;
// lots of code with inactive uses of p
}
```

where no actual parameter for `p` is ever active. A naive active type approach forces `p` to become of type `aFloat` and thereby also forces all actual parameters to be converted. A sophisticated approach would retain the type of `p` but would have to introduce a local `aFloat lp` in the branch body and still have to do the conversion. The data separation approach would quite naturally introduce the corresponding `ap` within the loop body (as the nearest enclosing scope) without the need for conversion.

Activity naturally extends to objects more complex than variables, in which case it is no longer a single Boolean. Activity of a variable of a structured type is the collection of the activities of its components. Activity of a procedure call site is the collection of activities of the actual parameters, both before and after the call. Context-sensitive activity indicates activity for the formal parameters of a procedure related to specific call sites and as such is related to flow-sensitive analysis. For example, within the analyzed code a procedure `s(float u)` may at one location in the code be called with an inactive actual parameter, while at another location the call involves an active actual parameter. This notion of context sensitivity can also be applied to user-defined types. Here the equivalent of the individual call site of the procedure is the individual instantiation of the type in question. The usage of context-sensitive analysis is the specialization of procedures and user-defined types according to the varying activity patterns of parameters and members respectively. As in program slicing, specialization of data types saves memory, and specialization of procedures saves run time and memory. However, specialization can lead to a combinatorial growth in the number of variations of a procedure or type.

Generalization is the opposite of specialization. Conservatively correct generalization activates any formal procedure parameter that has at least one call site with an active actual parameter. This approach may necessitate the conversions mentioned in Sec. 4.1. Activity within user-defined types can be generalized similarly across all instantiations. This approach obviously forfeits the memory and runtime savings of specialization at the advantage of a simpler transformation. In many cases, however, generalization is preferred by tool developers, assuming that the intended meaning of a procedure implies consistent activity patterns. Similarly the assumed uniformity between array elements, unlike user-defined type members, justifies that we merge their individual activity information into a single Boolean.

Even if generalization is often preferred to specialization, the original procedure or user-defined type is often kept in the augmented program, as a particular case of specialization, when no parameter or member is active. As mentioned in Sec. 3.4, this may create augmented procedure name conflicts, which can be solved by renaming or incorporating overloading resolution mechanisms in the transformation engine.

### 4.3. Data Scopes

In Sec. 4.1 and Sec. 4.2 we conveniently avoided the possibility of data with a scope outside that of a single procedure invocation or type instance. Such data may have global, procedure static, or type static scope.[7] For global data the only problem arises with a split of the code base as in Sec. 4.1. If global data is activated because of its use

[7] C/C++ like file static scope is a rather technical problem.

in $P^+$ but also used in $P^-$, the activation is transparent as long as data separation is employed. For data encapsulation with an active type the problem cannot be solved by simple replication as done with procedures. However, if the original data is already properly encapsulated within a singleton class with access methods, one could manually insert code that synchronizes the original and the augmenting replicant. The original purpose of the code split was to not subject $P^-$ to analysis (and transformation). This indicates at least a need for some simple second stage of transformations applied to $P^-$.

Section 4.1 hinted at a problem in replicating procedures, which (obviously by now) is the existence of procedure static data. For all practical purposes, this is global data with access limited to the procedure body. In order to safely replicate procedures, this data needs to be promoted to truly global data regardless of the chosen data augmentation method. Once the data is globalized, all remarks made for global data apply. As with the global data already, the downside is that the original has to be subjected to minor transformations such as removal of the procedure static data declaration and potential access changes to disambiguate names and synchronize global data replications.

The type static data can be handled in a similar fashion. However, one should observe that the data encapsulation approach does not imply type replication. Only data separation does, but there the synchronization is not needed and hence should be a less complex transformation.

### 4.4. Nonscalar Data

Aside from purely scalar operations involving single floating-point and integer variables, many numeric models require computations involving vectors, matrices, and higher-dimensional arrays. Such operations are often executed by using highly optimized linear algebra libraries. Some languages (e.g., Fortran9x) even provide an array notation and built-in intrinsics that operate on array elements just like scalar intrinsics. Consequently a mix between operations with arrays as operands and scalar operations with array elements as operands is to be expected.

Consider, for instance, a typical *saxpy* operation[8] executed by calling `saxpy(a,x,y)` implemented in an external library, where `a` is a scalar and `float x[n],y[n]` are vectors of matching length, and also assume `x` and `y` are active, while `a` is not. With data separation we have `float ax[n],ay[n]` and can easily generate an augmenting statement, for example, for a derivative computation using the same library and calling `saxpy(a,ax,ay)`. With simple data encapsulation we have `aFloat x[n],y[n]`, and now there is an obvious need to extract the `.o` and `.a` components into `float` arrays if one still wants to reap the benefits of the library calls. On the other hand, when array elements are used in scalar operations, they maintain the proximity of `x[i].o` to `x[i].a`. Alternatively one might decide to introduce array classes (e.g., here simplistically `struct aVector{ float o[n]; float a[n];}`) and avoid the extraction at the expense of losing spatial locality for the scalar components `x.o[i]` to `x.a[i]` In other words, we can recast this as a problem of data layout that is dependent on the frequency of use of array vs scalar data in the model code.

[8]This means $y_i = a * x_i + y_i, i = 1, \ldots, n$.

We note that the distinction `x[i].o` vs. `x.o[i]` that a source transformation has to accomplish for C and C++ is not fully reflected in Fortran9x array notation. In particular, one can define an active type `aReal` equivalent to the C `aFloat` and declare `type(aReal)::x(n)`, a vector of `n` instances of `aReal` accessed componentwise by `x(i)%o` and `x(i)%a`, respectively. At the same time the vectors of all original and augmenting `reals` are accessible via `x%o` and `x%a`, respectively. For saxpy vector operations in our example above, one can therefore simply write `y%o=a*x%o+y%o` and for the augmenting `y%a=a*x%a+y%a` without having to consider the data layout. Obviously, spatial data locality still is a concern, and therefore array dimension specific types may still be desirable.

The implicit extraction also works for array slicing, for example, `x(1:n:3)%o`, which extracts every third original data component and even extends to arrays of nested structured types. Array notation may become ambiguous, for example if the augmenting component `a` is itself an array. Then an expression such as `x(1:n:3)%a` would be rejected by the compiler, and the extraction into an intermediate temporary array must still be done explicitly. This kind of array notation makes it particularly easy to generate the pre- and postconversion codes that switch dynamically between encapsulation and separation, for example by writing `ax = x%a`.

Code transformations employing encapsulation for languages that do not support array notation still may face the issue of the dual use of data as arrays and single components. Robust code generation for conversions into and out of the encapsulated data representation for library calls requires dimension information that should ideally be encapsulated with the array itself. A code transformation for low-level languages such as C could solve this problem by promoting the plain memory pointers that serve as array handles to proper C++ array class instances. However, this matter is clearly beyond the scope of this paper.

## 5. The Index Method

The complete encapsulation (association by address) technique elaborated thus far requires that the programming language support structured types. Some languages, such as Fortran77, do not support structured types. In Sec. 4.4 we pointed out that the Fortran9x array notation supports distributivity of structured type selection over array section operations. In other words, the meaning of the construct `select(a,section(3:7,x))` should be the array section defined by `foreach e in section(3:7,x), select(a,e)` Some programming languages, even though they offer some level of array notation, do not enjoy this property. This is the case, for example, for Python and the Matlab language. The Python "expression" `x[3:7].a` is not even syntactically correct. C and C++ don't have explicit sectioning operations; but for higher-dimensional arrays (e.g., `aFloat x[n][m]`) there is – similar to Python – no syntax `x[i].a` supporting this distributivity.

Both of these difficulties can be overcome, however, with a simple technique we call the *index method*. Instead of augmenting a variable by changing the variable type to a structured type, we augment a variable by adding an index. Using the index solution, a scalar becomes a 1-dimensional array, a 1-dimensional array becomes a 2-dimensional array, and so on. The first position of the augmented array holds the original function value. Positions $2, \ldots, d$ hold the augmenting values. For example, consider these sample Fortran77 declarations and their associated augmentations.

`real x` $\Longrightarrow$ `real x(2)`

`real y(m,n)` $\Longrightarrow$ `real y(2,m,n)`

Here we have $d = 2$, but obviously $d$ can be set to the desired number of augmenting values. In this scheme, all references to the value of `x` use index 1, and all augmenting references use index 2. Thus, a sample program statement

```
y = x * z
```

would be augmented, in the special case of AD, to

```
y(2) = x(1)*z(2) + x(2)*z(1) //augmenting
y(1) = x(1)*z(1) //original stmt
```

Note that array section operations are now completely benign. There is no interfering structured-type selection operation. For example, consider the simple Matlab line

```
v = sin(x(3:9))
```

The index-method AD-augmented code would be

```
v(2,:) = cos(x(1,3:9)) * x(2,3:9)
v(1,:) = sin(x(1,3:9))
```

In short, the index method emulates a complete encapsulation solution for languages (such as Fortran77) that have no structured types, and for languages (such as Python or Matlab) that have array section operations with weak distributivity properties. The main drawback of the index method compared to the structured type method is that the augmenting data type must match the original data type. One also has to be concerned with the proper initialization of the augmenting data that has to be made explicit, instead of simply using default initializers defined within an active type. Moreover, languages may impose limitations on the number of dimensions, preventing further dimension extension when the limit has already been reached in the original code. In the case of Fortran this limit is 7. Nevertheless, the index method is an elegant solution to the problem of scalar vs array use of array data explained in Sec. 4.4.

## 6. Tools and Summary

Section 1 points to AD as a heavy user of numeric augmentations. It seems worthwhile to look at a few AD tools in the context of the four data augmentation approaches characterized in Sec. 3.

**Adifor** [2] was originally conceived as a tool for Fortran77 and not surprisingly follows the complete separation approach as motivated in Sec. 3.3. The absence of language features in Fortran77 that complicate the complicate separation as shown in Sec. 3.5 has made it a reliable tool capable of handling most Fortran77 codes. Furthermore, an experimental postprocessor semiautomates the index method for Fortran 77 programs.

Adifor90, the follow-on to Adifor, is under development and supports association-by-name, association-by-address using structured types, and association-by-address using the index method.

**Adol-C** [3] comes from the other end of the spectrum and uses complete encapsulation described in Sec. 3.1 via the operator overloading capabilities of C++ for the automatic differentiation of C and C++ codes. Within the AD community the operator overloading-based tools are characterized as the opposite of "source-transformation" tools such as Adifor. The assumption is that one can get away with global type change of all floating-point variables to an active type perhaps via a `#define`. In practical codes, the presence of `unions`, the use of `malloc`, and built-in I/O operations require more involvement. Consequently the truly automatic use of Adol-C as in the web-based optimization server NEOS [12] employs a code transformation step that not only changes the floating-point type explicitly in the code but also turns `unions` into `structs` and `mallocs` into `news`.

**Tapenade** [16] follows the separation approach of Fig. 5. It was not originally devised for object languages, and therefore this is the maximal separation level it uses. However, Tapenade has been extended to Fortran9x and therefore deals with `modules`. At the `module` level, the choice was made not to apply separation: when a module is differentiated, it encapsulates both the original module variables, types, and procedures and their differentiated counterparts. Since activity analysis is flow and context dependent, generalization is an issue. Generalization is systematic for differentiated structured types and procedures, except for one particular specialization for the completely nonactive case, namely, the original type or procedure is kept.

**Adic and OpenAD/F** [1, 13] both originate from the OpenAD framework; in other words, they both use the same underlying transformation engine. The current implementation most closely resembles the encapsulated data with separate operations approach described in Sec. 3.2. By design, the framework is language independent. Currently all mature AD tools for C++ rely on complete encapsulation but the lack of awareness of structure beyond elemental operations is a major problem for efficiency. Adic is being developed to handle C++, and OpenAD/F is geared to Fortran9x. Avoiding the apparent complexities of data separation for these languages is the main rationale for this design.

We have categorized a spectrum of transformation approaches for data augmentation into

- complete encapsulation, Sec. 3.1;
- encapsulated data with separate operations, Sec. 3.2;
- partially separate data, Sec. 3.4; and
- complete separation, Sec. 3.5 and Sec. 3.3.

Each of the approaches yields different levels of complexity for the augmenting transformations. We have considered in particular the following language features.

- User-defined types and access control
- Overloading and overloading resolution
- Pointers and pointer arithmetic
- Vector operations
- Global data and static data

While we can characterize the consequences of the four different augmentation approaches on the transformation, the areas of increased complexity are fairly separate and are a tradeoff rather than enable a direct comparison. Because tool development is ongoing, in particular with regard to more problematic language features, we still lack enough practical experience to draw an informed conclusion about which of the four approaches is best.

## References


[1] Adic v2.0.
`http://www.mcs.anl.gov/adicserver`.

[2] Adifor.
`http://www.cs.rice.edu/~adifor`.

[3] Adol-C.
`http://www.math.tu-dresden.de/~adol-c`.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[5] C. Bischof, P. Khademi, A. Mauer, and A. Carle. Adifor 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Comput. Sci. Eng.*, 3(3):18–32, 1996.

[6] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1239 of *LNCS*, pages 86–100. Springer, 1997.

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

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

[9] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[10] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004.

[11] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, 2004. IEEE Computer Society.

[12] NEOS.
`http://neos.mcs.anl.gov`.

[13] OpenAD.
`http://www.mcs.anl.gov/OpenAD`.

[14] H. Schichl and A. Neumaier. Interval analysis on directed acyclic graphs for global optimization. *J. Global Optimization*, 33(4):541–562, 2005.

[15] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):110–112, March 1998.

[16] Tapenade.
`http://www-sop.inria.fr/tropics`.

[17] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[18] J. Utke. Flattening basic blocks. In M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2006.