European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS 2004
P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer (eds.)
Jyväskylä, 24–28 July 2004

DATA FLOW ALGORITHMS IN THE TAPENADE TOOL FOR AUTOMATIC DIFFERENTIATION

Mauricio Araya-Polo* and Laurent Hascoët*

*INRIA Sophia-Antipolis, TROPICS team
2004 Route des lucioles, BP 93, 06902 VALBONNE, FRANCE
e-mails: Mauricio.Araya@sophia.inria.fr, Laurent.Hascoet@sophia.inria.fr

Key words: Automatic Differentiation, Adjoint code, Adjoint algorithm, Data-flow analysis, Dependence Graph

Abstract. *Automatic Differentiation (AD) is a program transformation that yields derivatives. Building efficient derivative programs requires complex and specific static analysis algorithms to reduce run time and memory usage. Focusing on the* reverse mode *of AD, which computes* adjoint programs*, we specify jointly the central static analyses that are required to generate an efficient adjoint code. We use a set-based formalization from classical data-flow analysis to specify* Adjoint Liveness, Adjoint Write*, and* To Be Recorded *analyses, and their mutual influences. These specifications are derived formally from the classical rules of Read-Write analysis. We also show how the notion of Dependence Graph, used in the parallelization domain, can be helpful to build better differentiated programs. We give illustrations on examples taken from real numerical programs, that we differentiate with our AD tool* TAPENADE*, which implements these analyses.*

1 INTRODUCTION

Classically, tools that perform code optimization require information on which variables are used by a given piece of code, which are overwritten, and which are killed i.e. completely overwritten. This is true for compilers. This is also the case when trying to optimize *adjoint* code produced by the reverse mode of Automatic Differentiation (AD).

Adjoint code is particularly complex. It poses serious problems, especially in terms of memory consumption. Run time is obviously crucial too. A number of data-flow analyses have been defined by several research groups to reduce this memory usage. In addition to the classical program analyses, such as Read-Write, some entirely new and specific analyses had to be defined, such as *Adjoint Liveness* Analysis, *To Be Recorded (TBR)* Analysis, and *Adjoint Write* Analysis.

Adjoint code has a particular structure, defined by the model of reverse AD. We must use this structure of adjoint programs to define the AD-specific data-flow analyses. For example, an adjoint code consists of two sweeps whose control flow are the exact symmetric of each other. Another point is the presence of instructions that *restore* previous values of variables. In our particular model this is done with a stack. It is clear that a general purpose data-flow analyzer will not be able to detect nor take advantage of these complex behaviors. We therefore believe that AD-specific data-flow analyses must be defined, and they must run on the original code, incorporating knowledge on how the adjoint code will derive from the original code.

The goal of this paper is to give a uniform specification of these analyses, defined on the structure of the original code. This specification relies on a model of the differentiation process that goes from the original code to its adjoint. Therefore this model must be described first. This specification will be used to demonstrate data-flow properties of adjoint codes and to highlight the relationship between these three analyses. It is also a foundation for implementation inside our AD tool TAPENADE [9].

We view data dependence analysis as one of the most precise kinds of data flow analysis. It is fundamental in code optimization and parallelization. This paper describes data dependence analysis on adjoint codes, and how this technique is applied by TAPENADE to improve speed and data locality in adjoint codes and other differentiated programs.

The next section 2 summarizes the necessary knowledge about reverse AD that will be used in the sequel, and section 3 gives basic notation and formulae about classical data-flow analyses. We refer to [6] for a full discussion about AD, and to [1] about data-flow analyses. Section 4 presents the model of reverse AD which is used in TAPENADE, and uses this model to define and specify Adjoint Liveness Analysis, TBR Analysis, and Adjoint Write Analysis. The following section 5 gives an illustration example adapted from an industrial numerical code. We underline the interest of the three analyses. Section 6 shows how data dependence analysis can be applied to differentiated programs, and even how it can be refined for adjoint programs. Section 7 shows how TAPENADE uses data dependence analysis to optimize differentiated programs in two particular situations.

2 ADJOINTS BY AUTOMATIC DIFFERENTIATION

Automatic Differentiation differentiates *programs*. An AD tool takes as input a source computer program P that, given a vector argument $X \in \mathbb{R}^n$, computes some vector function $Y = F(X) \in \mathbb{R}^m$. Let's write F' the Jacobian of F. A * superscript will denote transposition, and the dot $\cdot$ will denote product. In reverse mode, the AD tool generates a new source program $\overline{\mathtt{P}}$ that, given the argument X and a weight vector $\overline{Y}$, computes the gradient $F'^*(X) \cdot \overline{Y}$ of the scalar output $Y^* \cdot \overline{Y}$. To this end, AD first assumes that P represents all its possible run-time sequences of instructions, and it will in fact differentiate these sequences. Therefore, the *control* of P is put aside temporarily, and is put back into the differentiated program at the end. Any sequence of instructions is identified with a composition of vector functions. Thus, for a given control:

$$\begin{array}{rcl} \mathtt{P} & \text{is} & [I_1; I_2; \ldots I_p;], \\ F & = & f_p \circ f_{p-1} \circ \cdots \circ f_1, \end{array} \tag{1}$$

where each f_k is the elementary function implemented by instruction I_k. Finally, AD simply applies the chain rule to obtain derivatives of F. If we write for short X_k the values of all variables after each instruction I_k, i.e. $X_0 = X$ and $X_k = f_k(X_{k-1})$, the chain rule gives the required gradient

$$F'^*(X) \cdot \overline{Y} = f_1'^*(X_0) \cdot f_2'^*(X_1) \cdot \ \ldots \ \cdot f_{p-1}'^*(X_{p-2}) \cdot f_p'^*(X_{p-1}) \cdot \overline{Y}, \tag{2}$$

which can be mechanically translated back into a sequence of instructions. Then the control of P is reused to fuse all these sequences into a single program $\overline{\mathtt{P}}$.

We observe that equation (2) is most efficiently computed from right to left, because matrix×vector products are so much cheaper than matrix×matrix products. This method yields the gradient in a time which is only a small multiple of the time of the original function. However, there is a difficulty because the f' instructions require the intermediate values X_k in the *inverse* of their computation order. If the original program *overwrites* a part of X_k, the differentiated program must restore X_k before it is used by $f_{k+1}'^*(X_k)$. There are two main strategies for that:

- **Recompute-All (RA):** the X_k are recomputed when needed, restarting P on input X_0 until instruction I_k. Brute-force RA strategy has a quadratic time cost with respect to the total number of run-time instructions p. The TAF [5] tool uses this strategy, together with *checkpointing* to reduce its time complexity.

- **Store-All (SA):** the X_k are restored from a stack when needed. This stack is filled during a preliminary run of P that additionally stores variables on the stack just before they are overwritten. This preliminary run is called the *forward sweep* $\overrightarrow{\mathtt{P}}$. The differentiated instructions strictly speaking form the *backward sweep* $\overleftarrow{\mathtt{P}}$. Brute-force SA strategy has a linear memory cost with respect to p. The ADIFOR [2] and TAPENADE tools use this strategy.

Practically, both RA and SA strategies need a special storage/recomputation trade-off in order to be really efficient. This trade-off is called *checkpointing*. Since TAPENADE uses checkpointing on subroutine calls, we will describe checkpointing in this context. Let

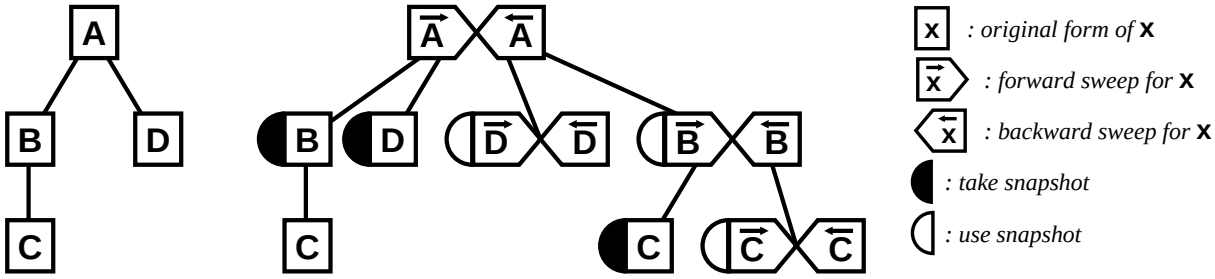

Figure 1: Checkpointing on calls in TAPENADE reverse AD

us define some vocabulary and graphical notations. Execution of a subroutine `A` in its original form is shown as `A`. The *forward sweep*, i.e. execution of `A` augmented with storage of variables on the stack just before they are overwritten, is shown as $\overrightarrow{\texttt{A}}$. The *backward sweep*, i.e. actual computation of the gradient of `A`, which pops values from the stack when they are needed to restore the X_k's, is shown as $\overleftarrow{\texttt{A}}$. The adjoint program is just $\overline{\texttt{A}} = \overrightarrow{\texttt{A}}; \overleftarrow{\texttt{A}}$. Checkpointing consists in choosing a part `B` of `A`, which will be run *without* storage during $\overrightarrow{\texttt{A}}$. When the backward sweep $\overleftarrow{\texttt{A}}$ reaches `B`, it runs $\overrightarrow{\texttt{B}}$, i.e. `B` again but this time with storage and then immediately runs the backward sweep $\overleftarrow{\texttt{B}}$ and the rest of $\overleftarrow{\texttt{A}}$. Duplicate execution of `B` requires that some variables used by `B` (a *snapshot*) be stored. In TAPENADE, checkpointing is applied at each procedure call. Figure 1 shows the resulting differentiated call tree for an example initial program call tree. If the program's call tree is well balanced, the memory size as well as the computation time required for the reverse differentiated program grow only like the depth of the original call tree, i.e. like the logarithm of the size of P.

3 CLASSICAL DATA-FLOW ANALYSES

We introduce some notation and basic formulae on classical data-flow analyses. These rules will be used in the next sections to derive formally specialized rules for adjoint data-flow analyses. Consider any fragment X of a program `P`.

- The set of all variables whose value given at the beginning of X is overwritten inside X (at least for some part of the variable and for some possible execution of X) is denoted $\mathbf{W}(X)$. For two successive pieces of program A and B, we have obviously:

$$\mathbf{W}(A;B) = \mathbf{W}(A) \cup \mathbf{W}(B) \tag{3}$$

  We will however use a refined rule when reverse AD uses a stack: if a variable is

PUSH'ed and later POP'ed, it is unmodified globally, so that

$$\mathbf{W}(\text{PUSH}(v); A; \text{POP}(v)) = \mathbf{W}(A) \setminus \{v\} \tag{4}$$

- The set of all variables whose value given at the beginning of X is always completely overwritten ("killed") inside X is denoted $\mathbf{K}(X)$. This set is always included in $\mathbf{W}(X)$. It is often different from $\mathbf{W}(X)$ in the case of arrays. Array region analysis is a standard technique that partially copes with this, but in general a single assignment to an array cell does not kill this array, so that $\mathbf{K}(\texttt{T(i)=0.0}) = \emptyset$. In general for two successive pieces of program A and B we take the standard conservative under-approximation:
$$\mathbf{K}(A; B) = \mathbf{K}(A) \cup \mathbf{K}(B) \tag{5}$$

- The set of all variables whose value given at the beginning of X is read inside X is denoted $\mathbf{R}(X)$. For two successive pieces of program A and B, the variables killed by A hide the variables read by B, so that:
$$\mathbf{R}(A; B) = \mathbf{R}(A) \cup (\mathbf{R}(B) \setminus \mathbf{K}(A)) \tag{6}$$

- When X is a tail of `P` (i.e. the end of X is the end of `P`), we can define the set of all variables whose value given at the beginning of X is necessary to obtain the final result of `P`. This set is denoted $\mathbf{N}(X)$. For us "necessary" means that the variable is involved in computations that eventually influence the final result. In other words if the variable is not necessary, then it is *dead* and might be modified without affecting the result. The set of all final outputs of `P` are necessary by definition, and therefore we initialize $\mathbf{N}([])$ to this set. Recursively, for any two successive pieces of program A and B, B being a tail of `P`, the variables necessary just before B lead to the variables necessary just before A through $Dep(A)$, the "dependence across A" information, defined as

$$Dep(A) = \{(v_o, v_i) \in \text{Out}(A) \times \text{In}(A) \mid v_o \text{ depends on } v_i\}$$

and through the combinator $\otimes$, defined as

$$V \otimes Dep = \{x \mid \exists y \in V \mid (y, x) \in Dep\}$$

so that:
$$\mathbf{N}(A; B) = \mathbf{N}(B) \otimes Dep(A) \tag{7}$$

4 ADJOINT DATA-FLOW ANALYSES

We consider a piece of program P, which is going to be differentiated by AD in the reverse mode, yielding its adjoint program $\overline{\mathtt{P}}$. P can be the complete function that will be differentiated, or it can be a checkpointed sub-part. In both cases, $\overline{\mathtt{P}}$ is made of a forward sweep $\overrightarrow{\mathtt{P}}$, *immediately* followed by a backward sweep $\overleftarrow{\mathtt{P}}$. This implies that the original results of P, which are also computed by $\overrightarrow{\mathtt{P}}$, are *not* a result of $\overline{\mathtt{P}}$. The only required results of $\overline{\mathtt{P}}$ are the differentiated (*adjoint*) variables, and not the original results which in most implementations will be overwritten and lost during $\overleftarrow{\mathtt{P}}$. It is well known that the last instruction of P is therefore not necessary in $\overline{\mathtt{P}}$, and by transitivity that several other instructions can also be dead in $\overline{\mathtt{P}}$.

We are going to specify a static analysis on P that will find out for any location in P (and therefore in $\overrightarrow{\mathtt{P}}$) the set of all *live* variables, i.e. variables necessary for the sequel of $\overline{\mathtt{P}}$ after this location. Necessary variables are defined recursively: we assume all differentiated variables in $\overleftarrow{\mathtt{P}}$ are necessary, which is equivalent to assuming that there is no dead code in the original program P. Recursively, all variables that are used in an instruction whose result is necessary are also necessary. This dead code analysis is strongly related to two others, the *To Be Recorded* analysis and the *Adjoint Write* analysis, so that we will have to define and study the three of them jointly.

We insist that these three analyses only deal with the original variables of P, and not their differentiated counterparts. In the sequel, an instruction that only writes into differentiated variables will have an empty **W** set.

An outline of the general structure of this technical section may be helpful. In section 4.1, we first give a precise specification of adjoint programs, that takes into account the *Adjoint Liveness*, *To Be Recorded*, and *Adjoint Write* analyses, in order to produce an improved adjoint code. Then we shall formalize these analyses using this specification of adjoint programs, starting with TBR analysis in section 4.2. Notice that this a priori introduces a circularity into the definition. After proving in section 4.3 an important lemma about the variables left unmodified by an adjoint program, we will be able in section 4.4 to formally derive specific rules that define the *Adjoint Liveness* analysis. We can then show that the definitions circularity mentioned above disappears, and consequently the *Adjoint Liveness* analysis must be run first, followed by the *To Be Recorded* analysis and finally by the *Adjoint Write* analysis. Section 4.5 formally derives the specific rules that define the *Adjoint Write* analysis, and highlights its usage for the checkpointing strategy. For all three analyses, the obtained definitions run directly on the original program. This gives a firm foundation for an implementation with a low computational cost.

4.1 Structure of adjoint programs

Strictly speaking, the fact that a variable is necessary for (a part of) $\overline{\mathtt{P}}$ depends on the architecture of $\overline{\mathtt{P}}$, i.e. on the reverse AD model and in particular on the strategy used to make intermediate values available to the backwards sweep. Here, we shall rely on the

Store-All strategy used in TAPENADE We believe however that the following specifications and demonstrations can be adapted to the Recompute-All strategy with minor adaptations. Let us explicit our reverse AD model. We define $\overline{\mathtt{P}}$ recursively on the structure of P. To keep things simple, suppose P is a straight-line program of atomic instructions. For an empty program $\mathtt{P} = [\,]$, $\overline{\mathtt{P}}$ is simply:

$$\overline{[\,]} = [\,] \tag{8}$$

Recursively, for an atomic assignment I followed by any downstream sequel D, the basic reverse AD model states that:

$$\overline{I;D} = \overrightarrow{I};\overline{D};\overleftarrow{I} = \mathrm{PUSH}(\mathbf{W}(I));I;\overline{D};\mathrm{POP}(\mathbf{W}(I));I' \tag{9}$$

where $\mathbf{W}(I)$ is the set of all variables that might be overwritten by I, I' are the derivative instructions for I, PUSH and POP are the usual stack routines, and ";" indicates concatenation of programs. In other words, values are saved just before they are overwritten, and restored before they may be used by a derivative instruction. However, at least three refinements are usually applied to the model to obtain a more efficient, yet equivalent, adjoint code.

An immediate refinement is to use *activity*, specified for example in [8]: at analysis time, some variables can be proved to have always a zero derivative with respect to the independent inputs or dependent outputs. When the variable written by assignment I is inactive, then I' can be freely replaced by $[\,]$. When some variable used by assignment I is inactive, I' can be simplified, therefore using fewer intermediate variables.

Another refinement is to use the adjoint liveness analysis that we are going to define below. In model (9), instruction I is a priori copied into $\overline{I;D}$. We observe that if the results of I are used later in $\overline{\mathtt{P}}$, this can be only in $\overline{D}$ because the backwards sweep $\overleftarrow{U;I}$ of instructions I and upstream in P can only use intermediate values that existed *before* execution of I, and certainly not use the results of I. Therefore, if no result of I is necessary for $\overline{D}$, then I is not necessary at all and can be removed from $\overline{\mathtt{P}}$, as well as the associated PUSH and POP instructions. Formally, we write $\mathbf{N}(\overline{D})$ the set of variables necessary for $\overline{D}$, and we define predicate *adj-live*$(I,D) = (\mathbf{W}(I) \cap \mathbf{N}(\overline{D}) \neq \emptyset)$ that tests whether some output of instruction I is necessary for $\overline{D}$. When this predicate is *false*, it will prevent generation of I, the PUSH, and the POP. For example, this is the case for the last instruction of P, i.e. when $D = [\,]$. As we said above, the output of the last I is not a direct result of $\overline{\mathtt{P}}$, and it is followed in $\overline{\mathtt{P}}$ only by the derivative instructions of I and upstream, that use only intermediate values that existed upstream I. Therefore the last I and the associated PUSH and POP can be removed. In other words for an isolated instruction I, we have $\overline{I} = I'$.

The third refinement is to PUSH and POP a variable from $\mathbf{W}(I)$ *only* if this variable is really used by the following differentiated instructions. This is the goal of TBR analysis [4, 10]. For example, the derivative of the "linear" instruction `x = y+2*z` does not use the

values of y nor z. Since the following differentiated instructions are $I'; \overleftarrow{U}$, where U is the instructions of P upstream I, we introduce U as a context into definition (9). Notation-wise, we use symbol $\vdash$ to separate this context from the part of the program currently differentiated. We introduce the set of all variables really used by instructions I' and after, which is $\mathbf{R}(I'; \overleftarrow{U})$. The only variables actually PUSH'ed and POP'ed for instruction I are now $\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U})$.

Consequently, model (9) turns into the following, more complex one:

$$\begin{array}{rcl} U \vdash \overline{I; D} & = & [\mathrm{PUSH}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U})); I;] \ \textit{if adj-live}(I, D) \\ & & [U; I] \vdash \overline{D}; \\ & & [\mathrm{POP}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U}));] \ \textit{if adj-live}(I, D) \\ & & I' \end{array} \tag{10}$$

4.2 Derived rules for TBR analysis

From the classical equation (6) of the $\mathbf{R}$ analysis, we can explicit the rules that recursively compute $\mathbf{R}(I'; \overleftarrow{U})$ and $\mathbf{R}(\overleftarrow{U})$, to obtain a formal specification of the TBR analysis. Since I' only overwrites differentiated variables, and we are here analyzing data-flow properties of the original variables only, $\mathbf{K}(I') = \emptyset$. Therefore

$$\mathbf{R}(I'; \overleftarrow{U}) = \mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U}), \tag{11}$$

where $\mathbf{R}(\overleftarrow{U})$ is defined recursively by:

$$\begin{array}{rcl} \mathbf{R}(\overleftarrow{[\,]}) & = & \mathbf{R}([\,]) = \emptyset \\ \\ \mathbf{R}(\overleftarrow{U; I}) & = & \begin{cases} \mathbf{R}(\mathrm{POP}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U})); I'; \overleftarrow{U}) & \\ \quad = (\mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U})) \setminus \mathbf{K}(I) & \textit{if adj-live}(I, D) \\ \\ \mathbf{R}(I'; \overleftarrow{U}) = \mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U}) & \textit{otherwise} \end{cases} \end{array} \tag{12}$$

These equations translate easily into data-flow equations that can be implemented efficiently, forward on a Flow Graph.

4.3 Adequacy lemma for the PUSH/POP mechanism

Equations (11) and (12) allow us to verify an important property of model (10): the PUSH/POP mechanism inside $\overline{D}$ does leave unchanged all variables used in $\overleftarrow{U; I}$, the backward sweep of the upstream instructions:

Lemma 1 *For any tail X of program* P*, preceded by upstream instructions U:*

$$\mathbf{W}(U \vdash \overline{X}) \cap \mathbf{R}(\overleftarrow{U}) = \emptyset$$

Proof. By induction on the length of X. **Terminal case:** if $X = [\,]$, $U \vdash \overline{X} = [\,]$ too, so its $\mathbf{W}$ set is empty and the property is true. **Induction case:** if $X = I; D$, then $U \vdash \overline{X}$ is defined by equation (10).

- If *adj-live*$(I; D)$ is *false*, then $\mathbf{W}(U \vdash \overline{I; D}) = \mathbf{W}([U; I] \vdash \overline{D}; I') = \mathbf{W}([U; I] \vdash \overline{D}) \cup \mathbf{W}(I')$, from definition (3). We know that $\mathbf{W}(I') = \emptyset$ because I' overwrites only differentiated variables. By induction hypothesis, $\mathbf{W}([U; I] \vdash \overline{D}) \cap \mathbf{R}(\overleftarrow{U; I}) = \emptyset$. From equation (12), we find $\mathbf{R}(\overleftarrow{U; I}) = \mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U})$ and therefore $\mathbf{W}([U; I] \vdash \overline{D}) \cap \mathbf{R}(\overleftarrow{U}) = \emptyset$ and the property is true.

- On the other hand if *adj-live*$(I; D)$ is *true*, Consider any variable $v \in \mathbf{R}(\overleftarrow{U})$. Notice that this implies through equation (11) that $v \in \mathbf{R}(I'; \overleftarrow{U})$.

  - Either $v \in \mathbf{W}(I)$. Since *adj-live*$(I; D)$ is *true*, and since v is also in $\mathbf{R}(I'; \overleftarrow{U})$, v will be PUSH'ed and then POP'ed. Thus from equation (4), v is unchanged just after the POP. Since I' overwrites only differentiated variables, v is unchanged through execution of $U \vdash \overline{I; D}$.
  - Or $v \notin \mathbf{W}(I)$. In that case, the only part of $U \vdash \overline{I; D}$ that might overwrite v is $[U; I] \vdash \overline{D}$. Equation (12) says that $\mathbf{R}(\overleftarrow{U; I}) = (\mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U})) \setminus \mathbf{K}(I)$. Since $v \notin \mathbf{W}(I)$, $v \notin \mathbf{K}(I)$ because the "killed" set is always included in the "written" set. So from $v \in \mathbf{R}(\overleftarrow{U})$ we get $v \in \mathbf{R}(\overleftarrow{U; I})$. The induction hypothesis ensures that $\mathbf{W}([U; I] \vdash \overline{D}) \cap \mathbf{R}(\overleftarrow{U; I}) = \emptyset$ and therefore $v \notin \mathbf{W}([U; I] \vdash \overline{D})$ and v is unchanged through execution of the whole $U \vdash \overline{I; D}$.

  Therefore $v \notin \mathbf{W}(U \vdash \overline{X})$ and the property is true. □

4.4 Derived rules for Adjoint Liveness analysis

We now can specify the Adjoint Liveness Analysis. We want to find recursive equations which, for any tail X of P, build the set $\mathbf{N}(U \vdash \overline{X})$ of necessary variables for $U \vdash \overline{X}$. For $X = [\,]$, we have of course $\mathbf{N}(\overline{[\,]}) = \emptyset$, independently from U. For $X = I; D$ we use definition (10). By definition the adjoint liveness property originates from differentiated variables, which are all assumed necessary. In definition (10), only $\overline{D}$ and I' write differentiated variables. Therefore $\mathbf{N}(U \vdash \overline{I; D})$ is the union of the necessary variables of two slices of $U \vdash \overline{I; D}$.

- One slice for variables that are necessary due to $\overline{D}$:

$$\begin{array}{l} [\mathrm{PUSH}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U})); I;] \;\; \textit{if adj-live}(I, D) \\ [U; I] \vdash \overline{D}; \end{array} \tag{13}$$

  From equation 7, the necessary variables are $\mathbf{N}([U; I] \vdash \overline{D}) \otimes \mathit{Dep}(I)$. This formula applies even when *adj-live*(I, D) is *false* because in this case $\mathbf{W}(I) \cap \mathbf{N}([U; I] \vdash$

$\overline{D}) = \emptyset$, i.e. I doesn't write any variable in $\mathbf{N}([U;I] \vdash \overline{D}$, and therefore $\mathbf{N}([U;I] \vdash \overline{D}) \otimes Dep(I) = \mathbf{N}([U;I] \vdash \overline{D}$.

- Another slice for variables that are necessary due to I'. From lemma 1, the variables necessary for I', which belong to $\mathbf{R}(\overleftarrow{I})$, are left unmodified by $[U;I] \vdash \overline{D}$. The slice is thus:

$$\begin{array}{l} [\text{PUSH}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U}));I;] \ \textit{ if adj-live}(I,D) \\ [\text{POP}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U}));] \ \textit{ if adj-live}(I,D) \\ I' \end{array} \tag{14}$$

  which is obviously independent from U, D, and *adj-live*(I,D). It is equivalent to I', and also to $\overline{I}$, whatever the context U. Its necessary variables are $\mathbf{N}(\overline{I})$.

In total, we obtain the following formula, which turns out to be independent from the context U:

$$\mathbf{N}(\overline{I;D}) = \mathbf{N}(\overline{I}) \cup (\mathbf{N}(\overline{D}) \otimes Dep(I)) \tag{15}$$

A priori, there was a risk of circularity in this specification, since it used the *adj-live* property in many places. But equation (15) turns out to be independent from the context, so there is no circularity after all. In practice, this implies that Adjoint Liveness analysis, that computes $\mathbf{N}(\overline{X})$, must be run before TBR analysis, that computes $\mathbf{R}(\overleftarrow{U})$. Equation (15) easily extends to Basic Blocks instead of instructions so that for any Basic Block B followed by a downstream code D:

$$\mathbf{N}(\overline{B;D}) = \mathbf{N}(\overline{B}) \cup (\mathbf{N}(\overline{D}) \otimes Dep(B)) \tag{16}$$

This equation translates easily into a data-flow equation that can be implemented efficiently, backwards on a Flow Graph. It is particularly efficient since $\mathbf{N}(\overline{B})$ and $Dep(B)$ can be precomputed once and for all.

4.5 Derived rules for Adjoint Write analysis

Let us now consider the case where the program piece P contains itself a piece which is checkpointed. For example in TAPENADE, this happens at the level of procedure calls. When I is replaced by a checkpointed piece of program C, the reverse AD model is different from (10), because checkpointing means to run C instead of $\overrightarrow{I}$ during the forward sweep $\overrightarrow{\mathtt{P}}$, then run $\overline{C} = \overrightarrow{C};\overleftarrow{C}$ instead of $\overleftarrow{I}$ during the backward sweep. To run C twice requires storing a *snapshot*, i.e. enough variables to restore the calling context of C. Clearly, storing $\mathbf{R}(C)$ is sufficient, but one can do better: First, what we need to run again is not C, but $\overline{C}$, and we just saw that the set of variables $\mathbf{N}(\overline{C})$, required to run $\overline{C}$, is smaller than $\mathbf{R}(C)$. Second, we need to restore a variable only if it was modified "in between", which can be detected by the $\mathbf{W}$ analysis. And we shall take advantage of the fact that $\mathbf{W}(\overline{D})$ is smaller than $\mathbf{W}(D)$. Therefore we define the snapshot

as $\mathbf{SNP}(U, C, D) = \mathbf{N}(\overline{C}) \cap (\mathbf{W}(C) \cup \mathbf{W}([U; C] \vdash \overline{D}))$, and the reverse AD model for a checkpointed program piece C is:

$$\begin{array}{rcl} U \vdash \overline{C; D} & = & [\text{PUSH}(\mathbf{W}(C) \cap \mathbf{R}(\overleftarrow{U}));] \ \textit{if adj-live}(C, D) \\ & & \text{PUSH}(\mathbf{SNP}(U, C, D)); \\ & & [C;] \ \textit{if adj-live}(C, D) \\ & & [U; C] \vdash \overline{D}; \\ & & \text{POP}(\mathbf{SNP}(U, C, D)); \\ & & [\,] \vdash \overline{C}; \\ & & [\text{POP}(\mathbf{W}(C) \cap \mathbf{R}(\overleftarrow{U}));] \ \textit{if adj-live}(C, D) \end{array} \tag{17}$$

Notice that model (17) is not necessarily optimal. Other choices could perform better for some programs. For example, putting U instead of $[\,]$ as the context for the generation of $\overline{C}$ will cost more PUSH/POP inside $\overline{C}$, and on the other hand storing $\mathbf{W}(C) \cap \mathbf{R}(\overleftarrow{U})$ becomes unnecessary in (17). Exploration of these tradeoffs is an open problem. In any case, we see that we need to specify another adjoint analysis, the Adjoint Write Analysis $\mathbf{W}(U \vdash \overline{X})$. If $X = [\,]$, obviously $\mathbf{W}(U \vdash [\,]) = \emptyset$. If $X = I; D$, we use model (10) and distinguish two cases according to *adj-live*(I, D). We also use definition (4), i.e. a PUSH/POP pair on a variable leaves it unmodified by definition:

$$\mathbf{W}(U \vdash \overline{I; D}) = \begin{cases} (\mathbf{W}(I) \cup \mathbf{W}([U; I] \vdash \overline{D})) \backslash (\mathbf{K}(I) \cap \mathbf{R}(I'; \overleftarrow{U})) & \\ \qquad\qquad \textit{if adj-live}(I, D) & \\ \mathbf{W}([U; I] \vdash \overline{D}) & \textit{otherwise} \end{cases} \tag{18}$$

As anticipated, we see that $\mathbf{W}(U \vdash \overline{I; D})$ is always included in $\mathbf{W}(I; D)$, and often strictly thanks to the PUSH/POP pairs. Again, equation (18) is easily implemented, backwards on the Flow Graph of `P`.

5 APPLICATION

Consider the example procedure `FLW2D1COL` (figure 2) taken from a Navier-Stokes flow solver, and shortened for readability without altering its structure. It contains a typical gather-scatter loop, which can account for many computations at run-time, and therefore many derivatives, because the number of mesh segments `nsg2-nsg1` may be large. We differentiate `FLW2D1COL` in the reverse mode with TAPENADE, which uses the Store-All strategy. The checkpointing strategy is applied to each subroutine call, and in particular to the call to `LSTCHK` at the end of the loop body. Figure 3 shows the resulting subroutine `FLW2D1COL`. Differentiated variables are shown here with a bar above. Since the loop's iterations are independent, the adjoining operation and the `do` loop operator commute (*cf* [7]). Let us examine the benefits of the data-flow analyses discussed above: the Adjoint Liveness Analysis, the TBR Analysis, and the Adjoint Write Analysis.

```
      subroutine FLW2D1COL(nsg1,nsg2,nubo,t3,pres,vnocl,
     +   g3,g4,rh3,rh4,ns,nseg,sq)
      integer nsg1,nsg2,ns,nseg,nubo(2,nseg),is1,is2
      real*8 t3(ns),g3(ns),g4(ns),rh3(ns),rh4(ns),pres(ns)
      real*8 vnocl(2,nseg),qsor,qs,pm,dplim,sq

      do 30 iseg=nsg1,nsg2
        is1 = nubo(1,iseg)
        is2 = nubo(2,iseg)
        qsor = t3(is1)*vnocl(2,iseg)
        qs = t3(is2)*vnocl(2,iseg)
        dplim = qsor*g4(is1)+qs*g4(is2)
        rh4(is1) = rh4(is1) + dplim
        rh4(is2) = rh4(is2) - dplim
        pm = pres(is1)+pres(is2)
        dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
        rh3(is1) = rh3(is1) + dplim
        rh3(is2) = rh3(is2) - dplim
        call LSTCHK(pm,sq)
30    continue
      end
```

Figure 2: *An example gather-scatter loop from a real code*

```
  subroutine FLW2D1COL(nsg1,nsg2 nubo,t3,t3_b,pres,pres_b,vnocl,
+   vnocl_b,g3,g3_b,g4,g4_b,rh3,rh3_b,rh4,rh4_b,ns,nseg,sq,sq_b)
  < omitted declarations >
  do iseg=nsg1,nsg2
     is1 = nubo(1,iseg)
     is2 = nubo(2,iseg)
     qsor = t3(is1)*vnocl(2,iseg)
     qs = t3(is2)*vnocl(2,iseg)
     dplim = qsor*g4(is1) + qs*g4(is2)
     rh4(is1) = rh4(is1) + dplim
     rh4(is2) = rh4(is2) - dplim
     pm = pres(is1) + pres(is2)
     dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
     rh3(is1) = rh3(is1) + dplim
     rh3(is2) = rh3(is2) - dplim
     call PUSH(sq)
     call PUSH(pm)
     call LSTCHK(pm, sq)
  < forward sweep ends, backward sweep begins >
     call POP(pm)
     call POP(sq)
     call LSTCHK(pm, pm_b, sq, sq_b)
     dplim_b = rh3_b(is1) - rh3_b(is2)
     qsor_b = g3(is1)*dplim_b
     g3_b(is1) = g3_b(is1) + qsor*dplim_b
     qs_b = g3(is2)*dplim_b
     g3_b(is2) = g3_b(is2) + qs*dplim_b
     pm_b = pm_b + vnocl(2,iseg)*dplim_b
     vnocl_b(2,iseg) = vnocl_b(2,iseg) + pm*dplim_b
     pres_b(is1) = pres_b(is1) + pm_b
     pres_b(is2) = pres_b(is2) + pm_b
     dplim_b = rh4_b(is1) - rh4_b(is2)
     qsor_b = qsor_b + g4(is1)*dplim_b
     g4_b(is1) = g4_b(is1) + qsor*dplim_b
     qs_b = qs_b + g4(is2)*dplim_b
     g4_b(is2) = g4_b(is2) + qs*dplim_b
     t3_b(is2) = t3_b(is2) + vnocl(2,iseg)*qs_b
     vnocl_b(2,iseg) = vnocl_b(2,iseg)+t3(is2)*qs_b+t3(is1)*qsor_b
     t3_b(is1) = t3_b(is1) + vnocl(2,iseg)*qsor_b
  enddo
  end
```

Figure 3: *The adjoint of subroutine* FLW2D1COL *from figure 2*

- Adjoint Liveness analysis shows that variables `dplim`, `rh3`, and `rh4` are not necessary in the adjoint. Furthermore, the call to `LSTCHK` is the last instruction in its own checkpointed sub-part (i.e. its downstream sequel is [ ]). Therefore this call can be removed too, as well as its associated PUSH and POP. Instructions that Adjoint Liveness Analysis would remove are shown on a grey background. Adjoint Liveness Analysis is under development in TAPENADE, therefore these instructions are not actually removed in the present version.

- TBR analysis shows for example that variable `dplim` is not used in the backward sweep, and therefore is not saved before it is overwritten. Variable `qsor` is used in the backward sweep, but it is not overwritten in its enclosing $\overline{\mathtt{P}}$. This explains the absence of PUSH and POP instructions for these variables.

- Adjoint Liveness Analysis and Adjoint Write Analysis help build smaller snapshots. On this example, one can check that $\mathbf{N}(\overline{\mathtt{FLW2D1COL}}) \subset \mathbf{R}(\mathtt{FLW2D1COL})$:

  $$\mathbf{R}(\mathtt{FLW2D1COL}) = \{\mathtt{nsg1}, \mathtt{nsg2}, \mathtt{nubo}, \mathtt{t3}, \mathtt{pres}, \mathtt{vnocl}, \mathtt{g3}, \mathtt{g4}, \mathtt{rh3}, \mathtt{rh4}, \mathtt{sq}\}$$
  $$\mathbf{N}(\overline{\mathtt{FLW2D1COL}}) = \{\mathtt{nsg1}, \mathtt{nsg2}, \mathtt{nubo}, \mathtt{t3}, \mathtt{pres}, \mathtt{vnocl}, \mathtt{g3}, \mathtt{g4}\} \cup (\{\mathtt{sq}\} \cap \mathbf{N}(\overline{\mathtt{LSTCHK}}))$$

  and that $\mathbf{W}(\overline{\mathtt{FLW2D1COL}}) \subset \mathbf{W}(\mathtt{FLW2D1COL})$:

  $$\mathbf{W}(\mathtt{FLW2D1COL}) = \{\mathtt{rh3}, \mathtt{rh4}, \mathtt{sq}\}$$
  $$\mathbf{W}(\overline{\mathtt{FLW2D1COL}}) = \{\mathtt{sq}\} \cap \mathbf{W}(\overline{\mathtt{LSTCHK}})$$

  These smaller sets allow for smaller snapshots, for example for the adjoint of the program that calls `FLW2D1COL`.

6 ADJOINT DATA DEPENDENCE ANALYSIS

We focus here on the use of the Dependence Graph, a tool from the domain of parallelization, applied to AD in reverse mode. The Dependence Graph represents a sub-order of a program's execution order. Any reordering or rescheduling of instructions inside the program, that preserves this sub-order is guaranteed to preserve the results of the program. The nodes of the Dependence Graph are the "atomic" operations of the program, and no reordering is considered inside one node. Here, we define one node per atomic instruction. The arrows of the Dependence Graph, called *data dependences*, link each two instructions that access the same variable, when at least one of the two accesses is a write. Intuitively, this means one cannot interchange an instruction that reads `x` and an instruction that writes `x`. In contrast, two successive reads of `x` may be exchanged, at least for what concerns `x`.

We shall here define a specific data dependence analysis for adjoint programs. We therefore pay particular attention to *incrementation* instructions `v = v +` *exp*, where `v` does not appear in *exp*. Incrementation instructions are commonplace in adjoint programs

(*cf* figure 3), and we observe that one can always exchange two successive incrementations of the same variable `v`. This independence property is slightly weaker than between two reads of `v`: it actually requires that the incrementation operations be atomic. However, this is verified in any sequential execution, and can be enforced in other cases. Therefore, we take the following refined definition of data dependences: Variable `v` causes a dependence from instruction I_1 to instruction I_2 if the following two conditions hold jointly:

- There is no instruction I between I_1 and I_2 where `v` is completely overwritten.
- Naming a_1 (*resp.* a_2) the action of I_1 (*resp.* I_2) on variable `v`, which can be one of {ⓦ *(write)*, ⓡ *(read)*, ⓘ *(increment)*, ⓝ *(don't use)*}, the relation $\mathcal{D}(a_1, a_2)$ is true, where $\mathcal{D}$ is defined by the table on the right.

$\mathcal{D}$:	ⓦ	ⓡ	ⓘ	ⓝ
ⓦ	*true*	*true*	*true*	
ⓡ	*true*		*true*	
ⓘ	*true*	*true*		
ⓝ				

The Dependence Graph is a very powerful, yet expensive, tool. Its size is such that it is usually computed only for parts of programs such as loop nests or basic blocks. Here we shall consider a basic block in the backward sweep $\overleftarrow{\texttt{P}}$ of the adjoint of a program `P`. During differentiation in reverse mode, this basic block accumulates differentiated instructions, and for each differentiated instruction we add a node into the Dependence Graph, and the corresponding new data dependences. Data dependences can be caused by original variables, for example between POP instructions that restore them and differentiated instructions that use them. Data dependences can also be caused by differentiated variables, for example between instructions that initialize them to instructions that use or modify them. If the number of instructions in the block is n_i and the number of variables is n_v, the complexity of building the Dependence Graph is at worst ${n_i}^2 \times n_v$ and its final memory size at most ${n_i}^2$.

Actual construction of the differentiated basic block is simply a topological sorting of the Dependence Graph. Among all possible solutions, some are more efficient because particular instructions are put closer, and in particular can then be merged into a more efficient code.

In the next section, we show two applications of the Dependence Graph to produce better differentiated code. One application occurs in the reverse mode of AD, like all we saw before. The other application deals with another mode of AD that we didn't consider so far, traditionally known as the *vector tangent* mode.

7 APPLICATIONS

As explained in section 2, each differentiated instruction I'_k in the backward sweep $\overleftarrow{\texttt{P}}$ of the adjoint of `P` implements the vector assignment

$$\overline{Y_{k-1}} = f'^{*}_{k}(X_{k-1}) \cdot \overline{Y_k}.$$

As usual in computer programs, program variables are overwritten, so that vectors $\overline{Y_k}$ and $\overline{Y_{k-1}}$ are stored in the same program variables $\overline{v_1}$ to $\overline{v_q}$. It is easy to see that these $\overline{I}_k$ contain a large number of *incrementation* instructions. Suppose that I_k, like an assignment, uses several variables and overwrites only one. If the assigned variable is put first, then f'_k has the following shape:

$$f'_k = \begin{pmatrix} \bullet & \bullet & \bullet & \cdots & \bullet \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \tag{19}$$

where dots "•" represent entries that may be non null, "1" represent entries that are exactly one, and the other entries are always null. Calling v_n the variables involved in I (v_1 assigned), and $\overline{v_n}$ their adjoint variables, the multi-instruction I'_k must implement:

$$\begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} := f'^*_k \times \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} = \begin{pmatrix} \bullet & & & & \\ \bullet & 1 & & & \\ \bullet & & 1 & & \\ \vdots & & & \ddots & \\ \bullet & & & & 1 \end{pmatrix} \times \begin{pmatrix} \overline{v_1} \\ \overline{v_2} \\ \overline{v_3} \\ \vdots \\ \overline{v_q} \end{pmatrix} \tag{20}$$

In a computer program, this Matrix×Vector product must be evaluated from the last row up, because $\overline{v_1}$ must not be overwritten before it is used by the other Row×Vector products. Each row, except for the topmost row, generates an instruction of the shape:

$$\overline{v_n} := \overline{v_n} + exp * \overline{v_1}$$

which is an incrementation (*exp* does not contain $\overline{v_n}$).

The Dependence Graph tells us when we can reorder differentiated instructions to gather instructions that initialize a differentiated variable and instructions that later increment it. Obviously, an initialization followed by successive increments can be fused in a single instruction, therefore saving memory traffic and cache misses. It is also closer to what people used to write when they programmed adjoint codes by hand. Illustration of this can be found on figure 3. Many adjoint instructions in the differentiated program `FLW2D1COL` have been merged. For example, instruction:

```
dplim = rh3(is1) - rh3(is2)
```

results from the gathering and fusion of the three adjoint instructions:

```
dplim = 0.0
dplim = dplim - rh3(is2)
dplim = dplim + rh3(is1)
```

Our second application of the Dependence Graph deals with the so-called *tangent vector* mode. Tangent mode computes directional derivatives $F'(X)\cdot(\dot{X})$ whereas reverse mode computes gradients $F'^*(X)\cdot\overline{Y}$. In tangent mode, differentiated instructions are

interleaved with the original instructions. Therefore there is no need for two separate sweeps, nor for PUSH/POP mechanism.

However, it is often useful to compute directional derivatives for *one* particular set of inputs but for *many* different directions. The simplest way to do that is to run the tangent differentiated program once for each direction. But then the original instructions keep running the same operations on the same data at every run. Only the differentiated instructions compute different values. Instead of that, a *tangent vector* program wraps each differentiated instruction into a loop on the differentiation directions, and the original instructions are run only once.

This results in a lot of small loops, which incurs a high loop overhead. This is a problem especially when noticing that the loops on differentiation directions are by essence parallel. To take full profit of this parallelism requires that the loop overhead is minimized.

The dependence graph is used to gather differentiated loops, so as to merge them and therefore reduce the loop overhead. For example on `FLW2D1COL` (figure 2), this results in all differentiated loops being merged into one, as shown on figure 4. Notice that in tangent mode differentiated variables are shown with a dot above, following a classical convention.

8 CONCLUSION

We have described the special-purpose data-flow analyses that are used in some Automatic Differentiation tools to improve the performances of the produced codes. We put a strong focus on those analyses that are useful for the reverse mode of AD, which produces adjoints.

These analyses make sense only for adjoint codes, and rely on the special structure of these programs. However, these are data-flow analyses and therefore can be described with the classical set-based notations used in literature on compiler theory. In addition to a formalized description of these analyses, we obtain a global view that clarifies the relationship between these analyses, their relative dependences and order. We also obtain formal proofs of important properties of our reverse AD model.

The goal of producing optimal adjoint programs is still not completely reached, and several other program optimizations will be necessary. We believe a formal description of analyses for adjoint programs can be useful to define and compare these analyses yet to come. In particular, we pointed out the link between TBR analysis and snapshots: finding the optimal tradeoff that minimizes the total memory usage would be a useful contribution.

In particular the AD tool TAPENADE progressively implements the analyses we described here. We want to promote this transposition of techniques that originate from compilation or parallelization into AD technology.

```
  subroutine FLW2D1COL(nsg1,nsg2 nubo,t3,t3,pres,pres,vnocl,
+   vnocl,g3,g3,g4,g4,rh3,rh3,rh4,rh4,ns,nseg,sq,sq)
  < omitted declarations >
  do iseg=nsg1,nsg2
    is1 = nubo(1, iseg)
    is2 = nubo(2, iseg)
    qsor = t3(is1)*vnocl(2, iseg)
    qs = t3(is2)*vnocl(2, iseg)
    pm = pres(is1) + pres(is2)
    do nd=1,nbdirs
      qs(nd) = t3(nd, is2)*vnocl(2, iseg) +
+       t3(is2)*vnocl(nd,2,iseg)
      qsor(nd) = t3(nd, is1)*vnocl(2, iseg) +
+       t3(is1)*vnocl(nd,2,iseg)
      pm(nd) = pres(nd, is1) + pres(nd, is2)
      dplim(nd) = qsor(nd)*g4(is1) + qsor*g4(nd, is1) +
+       qs(nd)*g4(is2) + qs*g4(nd, is2)
      rh4(nd, is1) = rh4(nd, is1) + dplim(nd)
      rh4(nd, is2) = rh4(nd, is2) - dplim(nd)
      dplim(nd) = qsor(nd)*g3(is1) + qsor*g3(nd, is1) + qs(nd)
+       *g3(is2) + qs*g3(nd, is2) + pm(nd)*vnocl(2, iseg)
+       + pm*vnocl(nd, 2, iseg)
      rh3(nd, is1) = rh3(nd, is1) + dplim(nd)
      rh3(nd, is2) = rh3(nd, is2) - dplim(nd)
    enddo
    dplim = qsor*g4(is1) + qs*g4(is2)
    rh4(is1) = rh4(is1) + dplim
    rh4(is2) = rh4(is2) - dplim
    dplim = qsor*g3(is1) + qs*g3(is2) + pm*vnocl(2, iseg)
    rh3(is1) = rh3(is1) + dplim
    rh3(is2) = rh3(is2) - dplim
    call LSTCHK(pm, pm, sq, sq)
  enddo
  end
```

Figure 4: *Tangent vector differentiation of subroutine* FLW2D1COL *from figure 2*

REFERENCES

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

[2] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann (Editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization*. LNCSE. Springer, 2002. selected papers from the AD2000 conference, Nice, France.

[4] C. Faure and U. Naumann. The taping problem in automatic differentiation. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 293–298, 2002.

[5] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. [www http://www.autodiff.com/tamc].

[6] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.

[7] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 299–304, 2002.

[8] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint ANL-MCS/P936-0202, Argonne National Laboratory, 2002. also *Rapport de recherche number 4856*, INRIA.

[9] INRIA Tropics team. On-line documentation of the Tapenade AD tool. Technical report. [www http://www.inria.fr/tropics].

[10] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In *Proceedings of the ICCS 2000 Conference on Computational Science, Part II*, LNCS. Springer, 2002.