TBR Analysis in Reverse-Mode Automatic Differentiation

Laurent Hascoët, Uwe Naumann, Valérie Pascual

To cite this version:

Laurent Hascoët, Uwe Naumann, Valérie Pascual. TBR Analysis in Reverse-Mode Automatic Differentiation. RR-4856, INRIA. 2003. inria-00071727

HAL Id: inria-00071727
https://hal.inria.fr/inria-00071727

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

TBR Analysis in Reverse-Mode Automatic Differentiation

Laurent Hascoët — Uwe Naumann — Valérie Pascual

N° 4856

Juin 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4856--FR+ENG

INRIA
SOPHIA ANTIPOLIS

TBR Analysis in Reverse-Mode Automatic Differentiation

Laurent Hascoët* , Uwe Naumann† , Valérie Pascual*

Thème 1 — Réseaux et systèmes
Projet Tropics

Rapport de recherche n° 4856 — Juin 2003 — 21 pages

Abstract: The automatic generation of adjoints of mathematical models that are implemented as computer programs is receiving a increased attention in the scientific and engineering communities. Reverse-mode Automatic Differentiation (AD) is of particular interest for large-scale optimization problems. It allows for the computation of gradients at a small constant multiple of the cost for evaluating the objective function itself, that is independent of the number of input parameters. Source-to-source transformation tools for AD are available to generate adjoint codes based on the adjoint version of every statement, built by applying simple differentiation rules. A reversal of the control flow of the original program becomes necessary. To guarantee correctness, certain values that are computed and overwritten in the original program must be made available in the adjoint program. They can be determined by performing a static data flow analysis, the so-called TBR analysis. Overestimation of this set must be kept minimal to get efficient adjoint codes.For many real-world applications, this efficiency is compulsory to apply a source-to-source transformation such as AD.

Key-words: automatic differentiation, adjoint, gradient, static analysis, data-flow analysis, compilation

* Projet Tropics, INRIA Sophia-Antipolis
† Mathematic and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Analyse TBR et autres analyses statiques pour le mode inverse de la Differentiation Automatique

Résumé : La génération automatique de codes adjoints commence à intéresser sérieusement le monde du calcul scientifique. En particulier le mode "inverse" de la Différentiation Automatique est intéressant pour les problèmes d'optimisation de grande taille, car il construit un programme adjoint qui calcule des gradients à un coût qui est un petit multiple du coût de la fonction initiale, et qui ne dépend pas du nombre de paramètres à ajuster. Il existe des outils logiciels de transformation de programme, qui génèrent ces programmes adjoints en utilisant la règle de dérivation des fonctions composées. Le mode inverse demande que certaines valeurs calculées dans le programme initial soient disponibles dans l'ordre inverse pour le programme adjoint. Ces valeurs peuvent être détectées par une analyse data-flow, statique, du programme initial, que nous appelons "TBR". Les sur-approximations de cet ensemble de valeurs sont inévitables, mais elles doivent être réduites au minimum pour que la différentiation automatique soit une approche viable pour l'optimisation de systèmes de taille réelle.

Mots-clés : différentiation automatique, adjoint, gradient, analyses statiques, analyses data-flow, compilation

1 Automatically Generated Adjoints

We consider a computer program P evaluating a vector function $\mathbf{y} = F(\mathbf{x})$, where $F : \mathbb{R}^n \to \mathbb{R}^m$. Usually, P implements the mathematical model of some underlying real-world application and it is referred to as the *original* code.

P is assumed to decompose into a sequence of scalar assignments of the form

$$v_j = \varphi_j(v_k)_{k \prec j}, \quad j = 1, \ldots, p+m, \tag{1}$$

with each variable v_j assigned only once. Each φ_j is some intrinsic (i.e., mathematical) function or elementary arithmetic operation provided by the programming language that P is written in. Among the v_j, the last m represent the output variables $\mathbf{y} \in \mathbb{R}^m$. Similarly the n inputs $\mathbf{x} \in \mathbb{R}^n$ are represented by variables v_k for $k = 0, \ldots, 1-n$. We say that a value v_j *depends in a differentiable way*, or *depends*, on v_k, and we write $v_k \prec v_j$ or $k \prec j$, iff the partial derivative of v_k with respect to v_j is defined. For example, in the assignment `x(i)=a*b(i+j)`, with `x`, `a`, and `b` of type real, the left-hand side `x(i)` depends on `a` and `b(i+j)`, but not on `i` nor `j`. Here, $\prec^*$ denotes the transitive closure of the relation $\prec$. Furthermore, we assume that the local partial derivatives

$$c_{ji} = \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \quad j \in 1, \ldots, p+m, \quad i \prec j, \tag{2}$$

are jointly continuous in some neighborhood of their arguments v_k, $k \prec j$. For example,

$$\frac{\partial \texttt{x(i)}}{\partial \texttt{a}} = \texttt{b(i+j)} \quad \text{for} \quad \texttt{x(i)=a*b(i+j)}.$$

With these assumptions, we can use automatic differentiation (AD) [4, 6, 7] to create an augmented version of P that computes and combines these local partial derivatives according to the chain rule, returning some derivatives of F. In particular, the reverse mode of AD (see, for example, [13, Section 3.3]) computes *adjoints* $\overline{v}_k$ for all intermediate and input variables v_k according to the recurrence

$$\bar{v}_k = \sum_{j: k \prec j} c_{jk} \cdot \bar{v}_j, \quad k = p, \ldots, 1-n. \tag{3}$$

Initializing $\overline{v}_{p+1}, \ldots, \overline{v}_{p+m}$ to $\overline{\mathbf{y}}$, we obtain in $\overline{v}_0, \ldots, \overline{v}_{1-n}$, the "transposed Jacobian matrix times vector" product $\overline{\mathbf{x}} = F'(\mathbf{x})^T \cdot \overline{\mathbf{y}}$ at complexity $O(m)$. Thus, gradients of a single output variable with respect to all inputs are obtained at a computational cost that is a small multiple of the cost of running the original code (see *cheap gradient principle* in [13]).

Considering equation (3), we observe that the adjoints are computed in reverse order from $k = p$ down to $1-n$. Thus, the local partial derivatives c_{jk} must be made available in reverse order as well. Their values depend, however, on the values of the input and intermediate variables v_k, where the latter are computed for $k = 1, \ldots, p$ in the original code. Since computer programs usually *overwrite* variables, some v_k may have been lost when they are

required by the derivatives. To recover these values, one can either (A) *store* the values required by the local partial derivatives on a so-called tape before they are overwritten and *retrieve* them whenever required in the adjoint code [8] or (B) *recompute* them "from scratch" when they become required in the adjoint code [11]. In both cases, we will say that these values have *to be recorded (TBR)*. Obviously, approach (A) may lead to enormous memory requirements for large-scale application programs, whereas (B) may result in a quadratic computational complexity. Sophisticated implementations of approach (B) can reduce this cost by reusing values of variables that become available during the computation of other required values [10]. Often a combination of the store and recompute strategies is employed to achieve reasonable trade-offs between memory use and execution time. See, for instance, the *checkpointing* schemes [12]. Whatever the approach, its efficiency would strongly benefit from the knowledge about whether some value is actually to be recorded. This is the purpose of the TBR analysis described in this paper.

Section 2 gives an intuitive description of the principles behind TBR analysis. Section 3 contains general introductory comments on data flow analysis. Sections 4 and 5 present the formal specifications of activity and TBR analyses, using the formalism of data flow equations. Section 6 concludes with a case study and some experimental measurements.

2 The Principle of TBR Analysis

The left column of figure 1 shows an example original code, which is the body of a subroutine that uses values from an array `x` to compute new values of `x`. The right column of figure 1 shows the corresponding adjoint code, which implements equation (3).

Notice that the control flow of the adjoint code is reversed from the control flow of the original code. This reversal can be achieved in various ways, which are outside the scope of this paper. Here, we count the number of iterations of loops into an integer `COUNT` and execute the adjoint loops the same number of times, but in reverse order. Similarly for conditionals, each time the original control flow merges, we remember where the control comes from, and this indicates where the adjoint control flow must go to. The `PUSH(`w`)` (resp. `POP(`w`)`) subroutine pushes (resp. pops) the value of variable w onto (*resp.* from) a stack that implements the tape. A similar tactic is implemented in our AD tool TAPENADE [1].

Since the adjoint code requires values from the execution of the original code, which may have been overwritten in the meantime, the adjoint code is preceded by a copy of the original code, augmented with instructions that store values on the tape before they get overwritten. We call this augmented original code the *forward sweep*, shown in the middle column of figure 1. Symmetrically, the adjoint code is augmented with instructions that retrieve these values when they are required (see approach (A) above). The forward sweep may precede immediately the adjoint code, as in the *joint program reversal mode* [13, Chapter 12], or at some previous time, as in the *split mode*. TBR analysis [8] observes that not all values need be stored during the forward sweep. One must store only the values that are effectively required in the adjoint code and will be overwritten during the rest of the forward sweep.

Original Code	**Forward Sweep**	**Adjoint Code**
i=0;j=10;a=3.14159	i=0;j=10;a=3.14159	POP(COUNT)
	COUNT=0	while (COUNT>0) {
while (check(j)) {	while (check(j)) {	COUNT=COUNT-1
if (max(i,j)>7) {	if (max(i,j)>7) {	POP(j)
	PUSH(x(i))	POP(i)
x(i)=j+sin(x(i))	x(i)=j+sin(x(i))	POP(test)
	PUSH(true)	if (test) {
} else {	} else {	POP(x(i))
	PUSH(x(j))	$\bar{x}$(i)=
x(j)=j*cos(x(j))+a	x(j)=j*cos(x(j))+a	cos(x(i))*$\bar{x}$(i)
	PUSH(false)	} else {
}	}	POP(x(j))
	PUSH(i)	$\bar{a}$=$\bar{a}$+$\bar{x}$(j)
i=i+1	i=i+1	$\bar{x}$(j)=
	PUSH(j)	-j*sin(x(j))*$\bar{x}$(j)
j=j-1; a=a/2	j=j-1; a=a/2	}
	COUNT=COUNT+1	}
}	}	
	PUSH(COUNT)	

Figure 1: Original code, forward sweep, adjoint code

Before describing TBR analysis, we must first introduce the notion of *active variables*. The user of AD often requests only the derivatives of some of the outputs $\mathbf{y}$ (the "*dependent*" variables $\mathbf{y_D}$), with respect to some of the inputs $\mathbf{x}$ (the "*independent*" variables $\mathbf{x_I}$). Let us use the term "variable" for a scalar component of an instance of some *program variable* present in the original code. A static analysis can detect, for any intermediate variable v in the original code, whether $\exists x \in \mathbf{x_I} : x \prec^* v$ and $\exists y \in \mathbf{y_D} : v \prec^* y$. In that case, v is called an *active variable*. Otherwise, the derivative of v will be neither computed nor used by the adjoint code, because it is either useless or trivially null. This will make the differentiated program simpler and more efficient. Section 4 gives the data-flow equations that define this *activity analysis*.

Consider now an original instruction $v = \phi(u)$, where $\mathbf{u} = \{u_1, u_2, \ldots, u_{n_\phi}\}$ denotes the set of scalar arguments of ϕ. This instruction generates a set of adjoint statements in the adjoint code that involve $\bar{v}$, $\bar{u}_i$, and $\frac{\partial \phi}{\partial u_i}(\mathbf{u})$. Activity matters: if v is not active, there will be no adjoint statement, and if some u_i is not active, the adjoint statements will involve neither $\bar{u}_i$ nor $\frac{\partial \phi}{\partial u_i}(\mathbf{u})$. The values required by the remaining adjoint statements are the arguments $\mathbf{u}$ of the local partial derivatives $\frac{\partial \phi}{\partial u_i}(\mathbf{u})$, plus the arguments of possible indices of the u_i and v, when these variables are array references. Precise rules are given in section 5.

Going back to our example in figure 1, we can check the following sets of required values:

Req(`x(i)=j+sin(x(i))`) $=$ {`x`} $\cup$ {`i`} $\cup$ {`i`} $=$ {`x,i`}
Req(`x(j)=j*cos(x(j))+a`) $=$ {`x,j`} $\cup$ {`j`} $\cup$ {`j`} $=$ {`x,j`}

Knowing these required sets, TBR analysis follows the flow of the original code, looking for overwritings of these values. When a required value is overwritten, TBR analysis inserts a `PUSH` instruction just before the overwriting and symmetrically restores the value with a `POP` instruction before it is required in the adjoint code. For both statements above, the original value of an element of `x` is required by the adjoint. Without array region analysis (see section 3) any component of `x` must be recorded when overwritten. The indices `i` or `j` are also required. But the overwriting occurs later, and so does the `PUSH`/`POP` pair. On the other hand, `a`, although repeatedly overwritten, is in no *Req* set and therefore need not be recorded.

3 Data Flow Analyses

Data flow analyses extract information from the text of programs about properties of run-time values. Data flow analyses are *static*, which means they are done at compilation time, without knowledge of data or behavior at run time. Hence, most static analyses are *undecidable*, that is, there always exists a particular program for which the result of the analysis is uncertain. Therefore, to obtain safe results, conservative *overapproximations* of the computed information are generated. For instance, such approximations are made when analyzing the activity or the TBR status of some individual element of an array. Static and dynamic *array region analyses* [15] provide very good approximations. Otherwise, we make a coarse approximation, in which the activity (resp. "requiredness") of one element implies the activity (resp. "requiredness") of the whole array.

Data flow analysis depends on the internal representation of programs, as discussed in classical literature on compiler theory (see, in particular, [2]). The most appropriate description appears to be in terms of *data flow equations*, defined on *call graphs* of *control flow graphs* (or simply *flow graphs*), which we have selected for TAPENADE.

- The call graph is a directed graph with one node for each subroutine or function of the program, and an arrow from node A to node B iff A possibly calls B. Recursion leads to cycles in the call graph.

- A subroutine or function is represented by a flow graph. There is one flow graph per node in the call graph. A flow graph is a directed graph, whose nodes are *basic blocks* [2]. Arrows in the flow graph represent the flow of control, that is, the possible destinations of the execution pointer after completion of a basic block. At run time, a test located at the end of the basic block decides on the direction of control flow.

At the lowest level, the individual *instructions* are represented simply as abstract syntax trees. To each basic block is associated a symbol table that gives access to properties of variables, constants, function names, type names, and so on. Symbol tables are nested to implement *scoping*.

Data flow analyses must be carefully designed to avoid or control combinatorial explosion. The classical solution is to choose a hierarchical model. In this model, information, or at least a computationally expensive part of it, is synthesized. Specifically it is computed bottom up, starting on the lowest (and smallest) levels of the program representation and then recursively combined at the upper (and larger) levels. Consequently, this synthesized information must be made independent of the context (i.e. the rest of the program). When the synthesized information is built, it is used in a final pass, essentially top down and context dependent, that propagates information from the "extremities" of the program (its beginning or end) to each particular subroutine, basic block, or instruction. This is the approach we use for both activity and TBR analyses.

Each data flow analysis is described concisely by *data flow equations*. In their most general form, these equations apply to *unstructured* flow graphs [2], because real programs have unstructured flow graphs in general. On the other hand, these general equations can be specialized to *structured* flow graphs, that is, cleanly nested loops and conditionals, yielding structured data flow equations. The latter may be applicable to a smaller class of program but are usually more efficient and also more illustrative.

4 Activity Analysis

As explained in section 2, activity analysis detects the variables for which a derivative must be computed, namely, all v such that $\exists x \in \mathbf{x_I} : x \prec^* v$ and $\exists y \in \mathbf{y_D} : v \prec^* y$. Therefore, given the set $\mathbf{x_I}$ of independent input variables, and the set $\mathbf{y_D}$ of dependent output variables, both sets provided by the end-user, activity analysis must do two tasks:

- Forward from the beginning of the program, it must propagate the set of all variables that possibly depend on some independent input.
- Backward from the end of the program, it must propagate the set of all variables on which some dependent output possibly depends.

Those are two static interprocedural data flow analyses. Therefore, we must control combinatorial explosion by selecting appropriate synthesized (i.e., *bottom-up*) information.

4.1 Differentiable Dependency Analysis

Classically, the bottom-up analysis that we need here is the *differentiable dependency* analysis, which computes, for each particular structure (i.e., instruction, basic block, or subroutine) all pairs of values $(v_b.v_a)$, v_b just *before* the structure and v_a just *after* the structure, such that $v_b \prec^* v_a$. We call this set *Dep*. It can be implemented very efficiently as an array of Booleans, but we willfocus on sets for the present description. We are going to give the data-flow equations that compute *Dep* at each level of the program representation.

For an individual instruction, there are two main cases: assignments and subroutine calls. We will examine subroutine calls when we deal with the interprocedural aspect. First let us focus on assignments. After an assignment, the assigned variable depends on all variables that occur in differentiable positions in the right-hand side. This set (call it *DP*) is given by the following constructive definition:

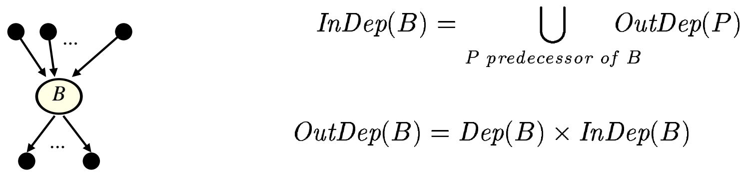
$expr$:	$e_1 \ op \ e_2$	$\varphi(e_1)$	$e_1[i]$	v	c
$DP(expr)$:	$DP(e_1) \cup DP(e_2)$	$DP(e_1)$	$DP(e_1)$	$\{v\}$	$\emptyset$

Above we have $\varphi \in \{\sin, \exp, \tan, \ldots\}$, $op \in \{+, -, *, \ldots\}$. Here, v denotes a single variable, and c stands for some constant value. All other variables not occurring on the left-hand side of an assignment remain unchanged. They depend just on themselves. When dealing with arrays, we must overestimate *Dep* as follows: if the left-hand side is an array reference and some reference to the same array occurs on the right-hand side, then we must recognize the fact that the array variable may depend on itself.

For a basic block, *Dep* is built by *composition* "$\times$" of the *Dep* set of each instruction. More generally, for any two structures S_1 and S_2 executed in sequence, $Dep(S_1; S_2) = Dep(S_1) \times Dep(S_2)$, where "$\times$" is defined by

$$(v_b \prec^* \ v_a) \in D_1 \times \ D_2 \iff \exists v, (v_b \prec^* \ v) \in D_1 \wedge (v \prec^* \ v_a) \in D_2.$$

For a subroutine, *Dep* is built on its flow graph. Strictly speaking, it represents the solution of the data flow equations that are solved iteratively on the whole (possibly unstructured) flow graph. For each basic block in the flow graph, we introduce *InDep* (resp. *OutDep*) as the dependencies from the entry of the subroutine to the *entry* (resp. *exit*) of the basic block. Clearly *InDep* on the subroutine entry is the identity (every variable depends on itself only), while *OutDep* on the subroutine exit is exactly the desired set *Dep* of the subroutine. In general, for any basic block, *InDep* and *OutDep* are related by the data flow



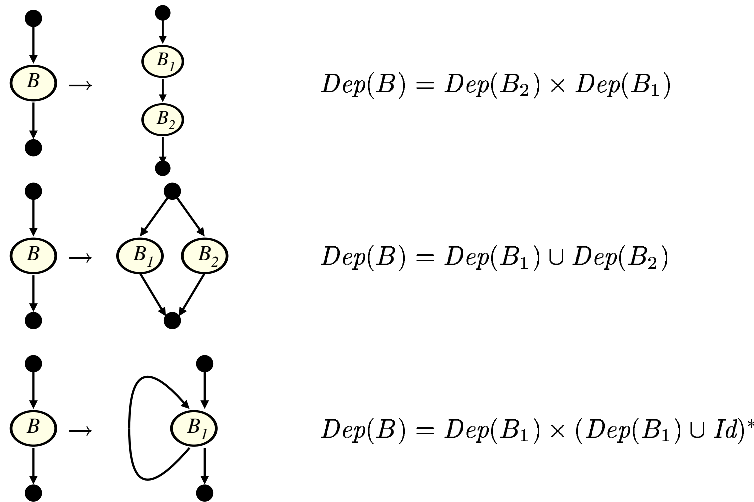
$$InDep(B) = \bigcup_{P\ predecessor\ of\ B} OutDep(P)$$

$$OutDep(B) = Dep(B) \times InDep(B)$$

Figure 2: *General data flow equations for differentiable dependency analysis*

equations shown in figure 2. Practically, these equations are solved iteratively on the flow graph. As for any iterative algorithm, a proof of termination is required, which we shall just sketch here: The idea of the proof is that the *InDep* and *OutDep* sets are initialized to $\emptyset$, and the successive, iterative values of these sets are strictly growing, inside a finite domain that has a maximum element (when each variable depends on every variable).

In the special case of structured flow graphs, the data flow equations can be specialized into structured data flow equations, shown in figure 3. They compute the *Dep* sets bottom up on the structured flow graph, which is certainly more efficient, but less general, than solving the unstructured equations. In particular, the fixpoint iteration is local to single loops.



$$Dep(B) = Dep(B_2) \times Dep(B_1)$$

$$Dep(B) = Dep(B_1) \cup Dep(B_2)$$

$$Dep(B) = Dep(B_1) \times (Dep(B_1) \cup Id)^*$$

Figure 3: *Structured data flow equations for the differentiable dependency analysis*

Subroutine calls are handled at the call graph level. For an individual instruction that is a call to subroutine S, the set of dependencies of the instruction is basically $Dep(S)$.

There is, however, a technical step of translating the variable names, as the name space of S differs from that of the calling subroutine. Now, the set *Dep* of each subroutine can be computed as soon as the *Dep* sets of all subroutines possibly called inside it have been computed. Consequently, when the call graph is acyclic, the *Dep* sets of each subroutine are computed by a bottom-up sweep. Otherwise they must be computed iteratively. This iterative computation does not pose any fundamental problems. For the sake of brevity, we shall not describe it here.

4.2 Varied and Useful Variables

To terminate activity analysis, one uses the result of the differentiable dependency analysis to propagate two data flow sets through the program:

- The *Varied* variables are variable instances v such that $\exists x \in \mathbf{x_I}, x \prec^* v$. *InVary* (resp. *OutVary*) denotes the set of varied varbiables just *before* (resp. *after*) a given program structure. By definition $\mathbf{x_I}$ are the varied variables at the program entry. The data flow equations describe the propagation of this information forward on the program flow.

- The *useful* variables are instances of variables v such that $\exists y \in \mathbf{y_D}, v \prec^* y$. *InUseful* (resp. *OutUseful*) denotes the set of useful variables just *before* (resp. *after*) a given program structure. Here, $\mathbf{y_D}$ are the useful variables at the program exit. Again, data flow equations describe the propagation of this information backward on the program flow.

Both analyses run top down on the call graph. Solutions must be obtained iteratively if the call graph contains cycles. For an acyclic call graph, subroutines are analyzed top down in a compatible order obtained by topological sorting. This approach ensures that all calls to subroutine S are analyzed before looking at S itself.

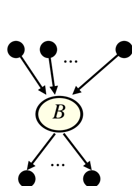
At the flow graph level, both analyses run similarly. The data flow equations are shown in figure 4 for unstructured flow graphs and in figure 5 for some sample structured flow graphs. The *InVary* set of the entry basic block is initialized to be the possibly varied variables with respect to some calling context. Similarly, the *OutUseful* set of the exit basic block is initialized as the possibly useful variables. For consistency, we overload the operator $\times$ as follows:

$$v_b \in S \times Dep(B) \quad \Leftrightarrow \quad \exists v_a \in S : \; (v_a \prec^* v_b) \in Dep(B);$$

$$v_a \in Dep(B) \times S \quad \Leftrightarrow \quad \exists v_b \in S : \; (v_a \prec^* v_b) \in Dep(B).$$

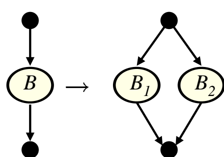
Finally, at the instruction level, the two important cases are as follows:

- For a simple assignment I, we have
  $$\begin{aligned} OutVary(I) &= InVary(I) \times Dep(I) \\ InUseful(I) &= Dep(I) \times OutUseful(I). \end{aligned}$$

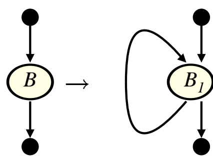


$$InVary(B) = \bigcup_{P \text{ predecessor of } B} OutVary(P)$$
$$OutVary(B) = InVary(B) \times Dep(B)$$
$$OutUseful(B) = \bigcup_{S \text{ successor of } B} InUseful(S)$$
$$InUseful(B) = Dep(B) \times OutUseful(B)$$

Figure 4: *General data flow equations for activity analysis*



$$InVary(B_1) = InVary(B_2) = InVary(B)$$
$$OutVary(B) = OutVary(B_1) \cup OutVary(B_2)$$
$$OutUseful(B_1) = OutUseful(B_2) = OutUseful(B)$$
$$InUseful(B) = InUseful(B_1) \cup InUseful(B_2)$$



$$InVary(B_1) = InVary(B) \times (Id \cup Dep(B))$$
$$OutVary(B) = OutVary(B_1)$$
$$OutUseful(B_1) = (Id \cup Dep(B)) \times OutUseful(B)$$
$$InUseful(B) = InUseful(B_1)$$

Figure 5: *Examples of structured data flow equations for activity analysis*

- For a subroutine call instruction I : `call` S`(...)`, we have

$$\begin{array}{rcl} \mathit{OutVary}(I) & = & \mathit{InVary}(I) \times \mathit{Dep}(S) \\ \mathit{InUseful}(I) & = & \mathit{Dep}(S) \times \mathit{OutUseful}(I). \end{array}$$

  In general, a subroutine S can be called at different points in the program or in different *contexts*. The information on varied and useful variables must be generated such that it represents the union of all these local results. Consequently, the current varied and useful variables are added to the respective summaries for all calling contexts of S:

$$\begin{array}{rcl} \mathit{InVary}(S) & = & \mathit{InVary}(S) \cup \mathit{InVary}(I) \\ \mathit{OutUseful}(S) & = & \mathit{OutUseful}(S) \cup \mathit{OutUseful}(I). \end{array}$$

5 TBR Analysis

As defined in section 2, TBR analysis determines the set of values to be recorded, namely, the values that are effectively required in the adjoint code and will be overwritten during the remainder of the forward calculation. Thus, TBR analysis follows the flow of the original code, propagating the set of variables whose current value is required in the adjoint code, and flags assignments that overwrite such a variable, so that its value will be recorded. As before, we identify a bottom-up analysis in order to control combinatorial explosion.

5.1 Bottom-Up TBR Analysis

For each structure S of the program, we synthesize a summary of the effect of this structure on TBR propagation. Concretely, this effect is composed of two parts:

- The *killed* variables, those that are certainly completely overwritten inside the given structure. We denote this set as $\mathit{Kill}(S)$.
- The *required* variables, those whose current value will be required in the adjoint of the given structure. We denote this set as $\mathit{Req}(S)$.

Data flow equations are used to compute these two pieces of information.

For an individual instruction, let us again focus on assignments. Subroutine calls will be treated later, when we look at interprocedural TBR analysis. As defined in section 2, the *Req* set of an assignment is empty if the assigned value is not active. Otherwise, a variable is required for the adjoint of an instruction $v = \phi(\mathbf{u})$ if it appears in $\frac{\partial \phi}{\partial u_i}(\mathbf{u})$, $i = 1, \ldots, n_\phi$, or, in other words, if it appears in an expression `e` of $\phi(u)$ that is an argument of a nonlinear operation whose result `a` is active, for example `sin(e)` or `e*a`. Furthermore, variables occurring in the indices of the u_i and v are required whenever the latter are array references. This requirement leads us to the operational rules below, expressed recursively on the structure

of the syntax tree. *VARS* is simply the set of all variables occurring in an expression. The rules for the killed variables are simpler, with just a special case for arrays: if array region analysis is not performed, an assignment to one array cell does not kill the array. Notice also that, since the right-hand side of an assignment is executed before the left-hand side is written, variables killed by the left-hand side are actually erased from the *Req* set.

Kill(`T[exp]=b`) $:= \emptyset$
Kill(`x=b`) $:= \{$`x`$\}$

Req(`a=exp`) := **if** `exp` active **then** (*Req*(`a`)∪*Req*(`exp`))*Kill*(`a=exp`) **else** ∅
(*and* `a` *must be recorded if it belongs to* *Req*(`exp`))
Req(`sin(a)`) := *VARS*(`a`)
Req(`a*b`) := (**if** `a` active **then** *VARS*(`b`)∪ *Req*(`a`) **else** ∅) ∪
(**if** `b` active **then** *VARS*(`a`)∪ *Req*(`b`) **else** ∅)
Req(`a+b`) := *Req*(`a`) ∪ *Req*(`b`)
Req(`T[index]`) := *Req*(`T`) ∪ *VARS*(`index`)
Req(*variable*) := ∅
Req(*constant*) := ∅

For a subroutine, *Req* and *Kill* are built jointly on the flow graph and solved iteratively. The corresponding data flow equations are shown in figure 6. For a basic block, these sets are jointly defined by composition of the inside instruction sets, just like the composition defined in figure 7 for the sequence of basic blocks. For each basic block, we introduce *InReq* (resp. *OutReq*) and *InKill* (resp. *OutKill*), the required and killed variables from the entry of the subroutine to the *entry* (resp. *exit*) of the basic block. Clearly *InReq* and *InKill* on the subroutine entry are ∅, and *OutReq* and *OutKill* on the subroutine exit are exactly the desired *Req* and *Kill* sets of the subroutine. Figure 6 essentially states that a variable is required after basic block B if it is required on at least one path leading to B and is not killed in B, or else if it is required inside B. Termination of the iterative solution process is granted by the fact that the computed sets are growing with respect to set inclusion and that they are bounded by the finite set of all variables in the program.

For structured flow graphs, the data flow equations can be specialized as shown in figure 7.

At the call graph level, the *Req* and *Kill* sets of a call to a subroutine S are equal to *Req*(S) and *Kill*(S). This implies one bottom-up sweep for acyclic call graphs.

5.2 Top-Down TBR Analysis

To terminate TBR analysis, we propagate the required variables along the program control flow. This is a top-down sweep on the program, which uses the results of the previous bottom-up phase. Two sets are computed and propagated for each program structure: *InReq*

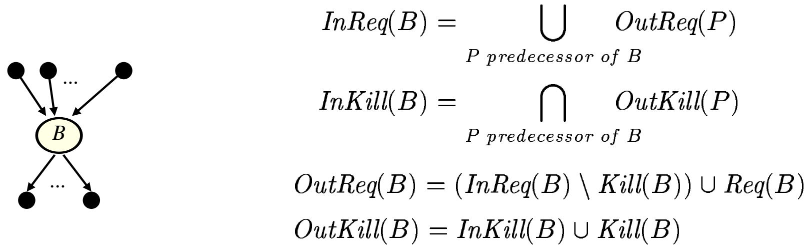


Figure 6: *General data flow equations for bottom-up TBR analysis*

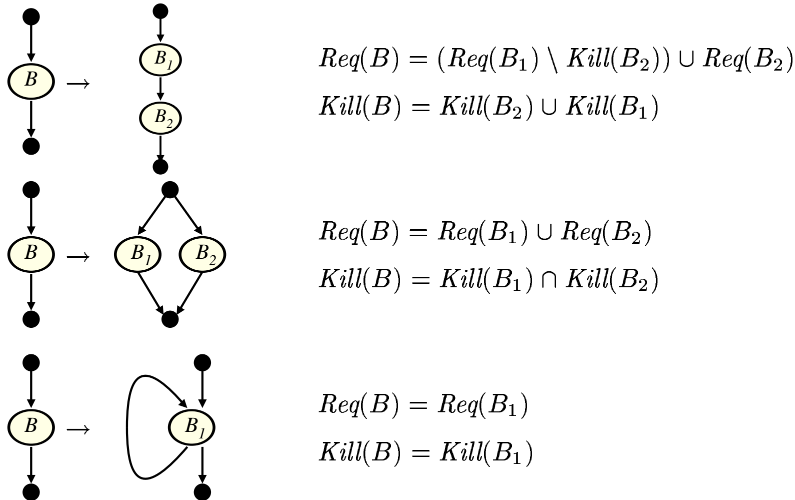
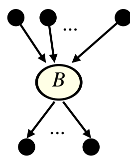


Figure 7: *Structured data flow equations for bottom-up TBR analysis*

(resp. *OutReq*) is the set of the variables whose value, *before* (resp. *after*) the current structure, is possibly required by the adjoint of previous instructions. Each time an individual instruction overwrites a required value (i.e. a variable present in the *InReq* set), we flag the overwritten value as "to be recorded".

At the call graph level, subroutines are analyzed top down, in an order obtained by topological sorting. This approach ensures that a called subroutine is analyzed after *all* of its calling sites have been analyzed. Thus the data flow equations for the subroutine call, given below, ensure that the *InReq* of all calling contexts are accumulated into the *InReq* of the called subroutine itself. The *InReq* of the top subroutine is initialized to $\emptyset$.

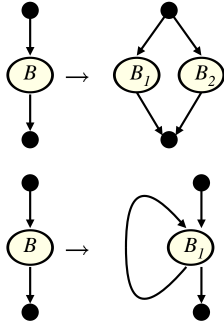
At the flow graph level, data flow equations for the TBR analysis are shown in figure 8. Figure 9 shows specialized data flow equations for some sample structured flow graphs. The *InReq* set of the entry basic block is initialized to the *InReq* of the subroutine itself.



$$InReq(B) = \bigcup_{P\ predecessor\ of\ B} OutReq(P)$$

$$OutReq(B) = (InReq(B) \setminus Kill(B)) \cup Req(B)$$

Figure 8: *General data flow equations for top-down TBR analysis*



$$InReq(B_1) = InReq(B_2) = InReq(B)$$

$$OutReq(B) = OutReq(B_1) \cup OutReq(B_2)$$

$$InReq(B_1) = InReq(B) \cup Req(B_1)$$

$$OutReq(B) = (InReq(B) \setminus Kill(B_1)) \cup Req(B_1)$$

Figure 9: *Examples of structured data flow equations for top-down TBR analysis*

Observe that, unlike for dependencies (figure 3), the structured data flow equations for the loop are not iterative and therefore can be solved at low cost with no fixpoint. This result was demonstrated in [14]. Here we reformulate the proof in the present formalism. The general data flow equations from figure 8, specialized to the structured loop of figure 9,

are

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{OutReq}(B_1);$$
$$\mathit{OutReq}(B_1) = (\mathit{InReq}(B_1) \setminus \mathit{Kill}(B_1)) \cup \mathit{Req}(B_1).$$

Substituting $\mathit{OutReq}(B_1)$ into the first equation gives the fixpoint definition

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup (\mathit{InReq}(B_1) \setminus \mathit{Kill}(B_1)) \cup \mathit{Req}(B_1),$$

whose solution is the first data flow equation for structured loops:

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{Req}(B_1). \ \blacksquare$$

Similarly, $\mathit{OutReq}(B)$ is equal to $\mathit{OutReq}(B_1)$ and

$$\mathit{OutReq}(B) = ((\mathit{InReq}(B) \cup \mathit{Req}(B_1)) \setminus \mathit{Kill}(B_1)) \cup \mathit{Req}(B_1),$$

which can be simplified to get the second data flow equation for structured loops:

$$\mathit{OutReq}(B) = (\mathit{InReq}(B) \setminus \mathit{Kill}(B_1)) \cup \mathit{Req}(B_1). \ \blacksquare$$

Finally, at the level of instructions, the two important cases are as follows:

- For a simple assignment I : `a=exp`, we have

  $$\mathit{OutReq}(I) = (\mathit{InReq}(I) \setminus \mathit{Kill}(I)) \cup \mathit{Req}(I)$$

  and `a` must be flagged as "to be recorded" if it belongs to $\mathit{InReq}(I)$.

- For a subroutine call I : `call` S`(...)`, we have

  $$\mathit{OutReq}(I) = (\mathit{InReq}(I) \setminus \mathit{Kill}(S)) \cup \mathit{Req}(S),$$

  and we add the current required variables $\mathit{InReq}(I)$ to the summary of required variables for all calling contexts of S:

  $$\mathit{InReq}(S) = \mathit{InReq}(S) \cup \mathit{InReq}(I).$$

6 Case Study and Experimental Results

We have applied TAPENADE with and without TBR analysis to a variant of the Bratu problem [3]. It models the thermal explosion of solid fuels, which can be described by the system of differential equations

$$x''(\tau) + s \cdot e^{\frac{x(\tau)}{1+t \cdot x(\tau)}} = 0,$$

where $\tau \in (-1, 1)$ and $x(-1) = x(1) = 0$. The problem has been discretized by using step size h as

$$F_i = x_{i-1} - 2x_i + x_{i+1} + h^2[f_{i-1} + 10f_i + f_{i+1}]/12$$

for $i = 1, \ldots, 10000$, with $x_0 = x_{10001} = 0$ and $f_i = s \cdot \exp(x_i/(1 + tx_i))$. Of interest are the derivatives of the component functions F_i with respect to the current state x_i as well as the parameters s and t. The original code implementing the discretized problem is shown in Appendix A. Appendix B lists the source of the main loop of the adjoint code generated by TAPENADE with TBR analysis. The values of the intermediate variables `exp5`, ..., `exp10` resulting from the canonicalization of the input code must be pushed onto the tape because their values are used nonlinearly in active terms inside the loop body. For example, `exp8` appears in `h*h*prm(1)/1.2*exp8`. Neither `f(i)` nor `f(i-1)` or `f(i+1)` is involved in the computation of any local partial derivative. This fact is recognized by the TBR analysis, and their values are not recorded.

With TBR analysis switched off, the value of all variables that appear on the left-hand side of some assignments must be recorded. In particular, additional push and pop statements have to be inserted for `f(i-1), f(i)`, and `f(i+1)`. This strategy is implemented, for example, in ADIFOR 3.0 [5] and Odyssée 1.7 [9]. While it took 377 sec. to run the code in Appendix B on a 233 MHz Pentium II (Linux) machine, the lack of TBR analysis increased the execution time to 466 sec.

Further experimental implementations of the ideas formalized in this paper showed even more promising reductions of the memory requirement when following a pure "store all" strategy. In [8] TBR analysis was applied to a large industrial thermal-hydraulic code developed at EDF-DER in France (70,000 lines, 500 subprograms, 1,000 parameters). The tape size could be decreased by a factor of 5. The size of the standard tape generated by Odyssée 1.7 is $213,920 \cdot 10^6$ scalar values (or 1,711,360 MBytes if every value is a double), whereas the optimized tape contains only $40486 \cdot 10^6$ scalar values (or 323,888 MBytes).

Acknowledgments

Naumann was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] `http://www-sop.inria.fr/tropics`. URL.

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

[3] B. Averik, R. Carter, and J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory, 1991.

[4] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, Philadelphia, 1996. SIAM.

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

[6] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.

[7] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, 1991. SIAM.

[8] C. Faure and U. Naumann. The taping problem in automatic differentiation. In *[6]*.

[9] C. Faure and Y. Papegay. *Odyssée* user's guide, version 1.7. Technical Report 0224, INRIA, September 1998.

[10] R. Giering and T. Kaminski. Towards an optimal trade-off between recalculation and taping in reverse mode ad. In *[6]*.

[11] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Software*, 24:437–474, 1998.

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

[13] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 2000.

[14] 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.

[15] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation.* ACM, 2000.

A Bratu Problem

```
      subroutine bratu(dim,parmax,x,prm,F)
      integer    dim    , parmax
C     independent variables
      double precision x(dim), prm(parmax)
```

```
C     dependent variables
      double precision F(dim)
C
      integer i
      double precision h

      h = 2.0/(dim+1)
      F(1) = -2*x(1)+h*h*prm(1)/12.0*(1+10*exp(x(1)/(1.0+prm(2)*x(1))))
      F(2) = x(1)+h*h*prm(1)/12.0*exp(x(1)/(1.0+prm(2)*x(1)))

      do 1 i=2,dim-1
        F(i-1) = F(i-1)+x(i)+h*h*prm(1)/12.0*exp(x(i)/(1.0+prm(2)*x(i)))
        F(i) = F(i)-2*x(i)+h*h*prm(1)/1.2*exp(x(i)/(1.0+prm(2)*x(i)))
        F(i+1) = x(i)+h*h*prm(1)/12.0*exp(x(i)/(1.0+prm(2)*x(i)))
    1 continue

      F(dim-1) = F(dim-1)+x(dim)+h*h*prm(1)/12.0*exp(x(dim)/(1.0
     *           +prm(2)*x(dim)))
      F(dim) = F(dim)-2*x(dim)
      F(dim) = F(dim)+h*h*prm(1)/12.0*(1+10*exp(x(dim)/(1.0
     *         +prm(2)*x(dim))))
      end
```

B Adjoint Bratu Problem (with TBR analysis)

```
      SUBROUTINE BRATUCL(dim, parmax, x, xccl, prm, prmccl, f, fccl)
        INTEGER dim, parmax
        DOUBLE PRECISION f(dim), fccl(dim)
        DOUBLE PRECISION prm(parmax), prmccl(parmax), x(dim), xccl(dim)
        REAL*8 exp1, exp11, exp11ccl, exp13, exp13ccl, exp1ccl, exp3,
     +         exp3ccl, exp5, exp5ccl, exp7, exp7ccl, exp9, exp9ccl
        REAL*8 exp10, exp10ccl, exp12, exp12ccl, exp14, exp14ccl, exp2,
     +         exp2ccl, exp4, exp4ccl, exp6, exp6ccl, exp8, exp8ccl
        DOUBLE PRECISION h
        INTEGER adTo, i
        ...

        DO i=2,dim-1
          CALL PUSHREAL8(exp5)
          exp5 = x(i) / (1.0+prm(2)*x(i))
          CALL PUSHREAL8(exp6)
          exp6 = EXP(exp5)
```

```
      f(i-1) = f(i-1) + x(i) + h * h * prm(1) / 12.0 * exp6
      CALL PUSHREAL8(exp7)
      exp7 = x(i) / (1.0+prm(2)*x(i))
      CALL PUSHREAL8(exp8)
      exp8 = EXP(exp7)
      f(i) = f(i) - 2 * x(i) + h * h * prm(1) / 1.2 * exp8
      CALL PUSHREAL8(exp9)
      exp9 = x(i) / (1.0+prm(2)*x(i))
      CALL PUSHREAL8(exp10)
      exp10 = EXP(exp9)
      f(i+1) = x(i) + h * h * prm(1) / 12.0 * exp10
    ENDDO
    CALL PUSHINTEGER4(i - 1)
    ...

    CALL POPINTEGER4(adTo)
    DO i=adTo,2,-1
      exp10ccl = exp10ccl + h * h * prm(1) * fccl(i+1) / 12.0
      exp9ccl = exp9ccl + EXP(exp9) * exp10ccl
      xccl(i) = xccl(i) + fccl(i+1) + (1 / (1.0+prm(2)*x(i)) - x(i)
+       * prm(2) / (1.0+prm(2)*x(i))**2) * exp9ccl
      prmccl(1) = prmccl(1) + exp10 * h * h * fccl(i+1) / 12.0
      fccl(i+1) = 0.D0
      CALL POPREAL8(exp10)
      exp10ccl = 0.0
      CALL POPREAL8(exp9)
      prmccl(2) = prmccl(2) - x(i) * x(i) * exp9ccl / (1.0+prm(2)*x(
+       i))**2
      exp9ccl = 0.0
      exp8ccl = exp8ccl + h * h * prm(1) * fccl(i) / 1.2
      exp7ccl = exp7ccl + EXP(exp7) * exp8ccl
      exp6ccl = exp6ccl + h * h * prm(1) * fccl(i-1) / 12.0
      exp5ccl = exp5ccl + EXP(exp5) * exp6ccl
      xccl(i) = xccl(i) + ((1 / (1.0+prm(2)*x(i)) - x(i) * prm(2) /
+       (1.0+prm(2)*x(i))**2) * exp7ccl - 2 * fccl(i) + fccl(i-1)) +
+       (1 / (1.0+prm(2)*x(i)) - x(i) * prm(2) / (1.0+prm(2)*x(i))**
+       2) * exp5ccl
      prmccl(1) = prmccl(1) + exp8 * h * h * fccl(i) / 1.2
      CALL POPREAL8(exp8)
      exp8ccl = 0.0
      CALL POPREAL8(exp7)
      prmccl(2) = prmccl(2) - x(i) * x(i) * exp7ccl / (1.0+prm(2)*x(
```

```
+       i))**2
      exp7ccl = 0.0
      prmccl(1) = prmccl(1) + exp6 * h * h * fccl(i-1) / 12.0
      CALL POPREAL8(exp6)
      exp6ccl = 0.0
      CALL POPREAL8(exp5)
      prmccl(2) = prmccl(2) - x(i) * x(i) * exp5ccl / (1.0+prm(2)*x(
+       i))**2
      exp5ccl = 0.0
    ENDDO
    ...

  END
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399