"To Be Recorded" Analysis in Reverse-Mode Automatic Differentiation

Laurent Hascoët [a] Uwe Naumann [b] Valérie Pascual [a]

[a] *Projet Tropics, INRIA, Sophia-Antipolis, F-06902, France*

[b] *Software and Tools for Computational Engineering, RWTH Aachen University, 52056 Aachen, Germany*

Abstract

The automatic generation of adjoints of mathematical models that are implemented as computer programs is receiving increased attention in the scientific and engineering communities. Reverse-mode automatic differentiation is of particular interest for large-scale optimization problems. It allows the computation of gradients at a small constant multiple of the cost for evaluating the objective function itself, independent of the number of input parameters. Source-to-source transformation tools apply simple differentiation rules to generate adjoint codes based on the adjoint version of every statement. In order to guarantee correctness, certain values that are computed and overwritten in the original program must be made available in the adjoint program. For their determination we introduce a static data-flow analysis called "to be recorded" analysis. Possible overestimation of this set must be kept minimal to get efficient adjoint codes. This efficiency is essential for the applicability of source-to-source transformation tools to real-world applications.

1 Automatically Generated Adjoints

We consider a computer program P evaluating a vector function $\mathbf{y} = F(\mathbf{x})$, where $F : I\!R^n \to I\!R^m$. Usually, P implements the mathematical model of some underlying real-world application and it is referred to as the *original* code. The goal of automatic differentiation (AD) [3,7,15] by source transformation is to build automatically a new source program P' evaluating some derivatives of F. This is arrow AD in Figure 1.

We consider a simplified mathematical model, symbolized by arrow R in Figure 1: Every particular run of P on a particular set of inputs is equivalent to a simple sequence of p scalar assignments

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

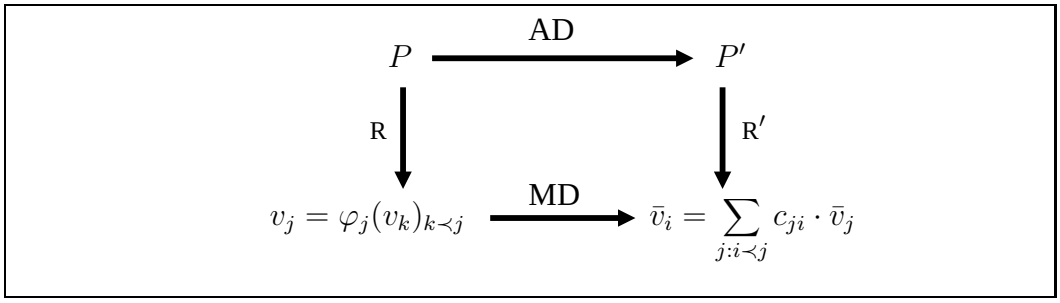


Fig. 1. *Automatic differentiation and mathematical differentiation*

where n variables $v_k, k = 1-n, \ldots, 0$ represent the input $\mathbf{x} \in I\!R^n$ and the last m variables [1] among the v_j form the output $\mathbf{y} \in I\!R^m$. Each φ_j is some mathematical function or arithmetic operation provided by the programming language that P is written in and its result is assigned to one of the q unique intermediate variables. We set $q = p + m$. This sequence may change when the input changes. Compared with P, this model is extremely simplified: array indices have been solved, so that variables v_j are scalars, and these variables are assigned only once. There are no control statements, only assignments. We say that a variable v_j *depends in a differentiable way*, or *depends*, on v_i, and we write $v_i \prec v_j$ or $i \prec j$, iff v_i is effectively an argument of φ_j and the partial derivative

$$c_{ji} \equiv \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j}$$

of φ_j with respect to v_i is defined. Furthermore, we assume that the c_{ji} are jointly continuous in some neighborhood of their arguments v_k, $k \prec j$. With these assumptions, one can write formulas that use the chain rule to get the desired derivatives. Let us call this *mathematical differentiation* (arrow MD in Figure 1). The following formulas are of particular importance (see, for example, [17], [24], [14, Section 3.3]). They compute *adjoints* $\overline{v}_i$ for all intermediate and input variables v_i according to the recurrence

$$\bar{v}_i = \sum_{j:i \prec j} c_{ji} \cdot \bar{v}_j, \quad i = p, \ldots, 1-n \quad . \tag{2}$$

Initializing $\overline{v}_{p+1}, \ldots, \overline{v}_q$ 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 time 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 [14, Section 3.4]).

However, for AD by source transformation [4,5,20,6,25,19,12,27] the path we follow in Figure 1 is arrow AD, creating a new program P', then arrow R′, which

[1] This restriction can be relaxed to account for more general situations where arbitrary intermediate variables can be outputs. The current approach allows us to use a slightly simpler notation.

corresponds to executing P'. The rules to create P' are motivated and justified by Equation (2), but AD is a *static* program analysis and transformation, which ignores run-time information, unlike MD. Therefore, AD is based on a static representation of programs such as call graphs of flow graphs, summarized in Section 3. Section 2 and in particular Table 2.1 describe AD transformation precisely, but we shall underline here some questions that the mathematical model does not need to take into account.

In the sequel, we shall keep the term *variable* for the mathematical variable represented by a program variable at a given location in the program. In particular, one program variable a, whether located *before* or *after* an operation that overwrites a, represents two different variables. In *static* analyses [18], finding the variable associated with a program variable at some location is a difficult task, undecidable in many cases such as pointers or arrays. Therefore, static data-flow analyses must make conservative overestimations for information based on variables. The relation $\prec$ is extended to this new definition of variables: For two program variables a and b each at a given location, a $\prec$ b means that a (resp. b) may represent variable v_a (resp. v_b) such that $v_a \prec v_b$.

In Equation (1), each variable v_j is written only once. However, real programs often use the same memory location to store different variables. This *overwriting* loses the value of variables. Considering Equation (2), we observe that the adjoints are computed in reverse order from $i = p$ down to $1 - n$. Thus, the local partial derivatives c_{ji} must be made available in reverse order as well. Their values depend, however, on the values of the input and intermediate variables v_k, $k = 1-n, \ldots, p$. Because of overwriting, some v_k may have been lost. To recover the values of these variables, one can either (A) *store* their values on a data structure that is often referred to as a *tape* before they are overwritten and *retrieve* the values when required in the adjoint code [8] or (B) *recompute* them "from scratch" when they become required in the adjoint code [10]. In both cases, we say that these variables 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 variables [11]. 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 in [13]. Whatever the approach, its efficiency would strongly benefit from the knowledge about whether some variable is actually to be recorded. This is the purpose of the TBR analysis described in this paper.

Section 2 discusses reverse-mode AD of programs and describes intuitively the principle of 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 Reverse AD and the Principle of TBR Analysis

The left column of Table 2.1 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 shows the corresponding adjoint code, which implements Equation (2).

Since we use the *store* strategy to retrieve overwritten values needed by the adjoint computation, the adjoint code is preceded by a copy of the original code, augmented with instructions that store values of certain variables before they get overwritten. We call this the *forward sweep*, shown in the middle column of Table 2.1. Symmetrically, the adjoint code is augmented with instructions that retrieve these values when they are required. TBR analysis [8] observes that not all values need be stored during the forward sweep. One needs to store only those values that are effectively used in the adjoint code and will be overwritten during the rest of the forward sweep.

Before describing TBR analysis, we must 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 $\mathbf{x_I}$). For any relation $\circ$, we define the closure $\circ^+ = \cup_{k=1}^{\infty} \circ^k$, where $\circ^k$ is the composition of k times $\circ$. 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 should neither be computed nor used by the adjoint code, because it is either useless or trivially null. This strategy makes the differentiated program simpler and more efficient. Section 4 gives the data-flow equations that define this *activity analysis*.

Consider now an original instruction $I : a = \phi(B)$, where $B = \{b_1, b_2, \ldots, b_k\}$ denotes the set of scalar arguments of ϕ. Instruction I generates a set of adjoint statements in the adjoint code that involve $\bar{a}$, $\bar{b}_i$, and $\frac{\partial \phi}{\partial b_i}(B)$ for $i = 1, \ldots, k$. Activity matters: If a on the left-hand side of I is not active, then there is no adjoint statement. [2] Similarly, if some b_i on the right-hand side of I is not active, then the adjoint statements involve neither $\bar{b}_i$ nor $\frac{\partial \phi}{\partial b_i}(B)$. The variables used by the remaining adjoint statements are the arguments B of the local partial derivatives $\frac{\partial \phi}{\partial b_i}(B)$, plus possible indices of the b_i and a, when these variables are array references. Precise rules are given in Section 5. Going back to our example in Table 2.1, we can check the following "*AdjU*" sets of variables required for the correct evaluation of the adjoint:

[2] Note that in this case none of the b_i on the right-hand side of I can be active either.

Table 2.1 A possible implementation of reverse-mode AD transforms the original code in the left column into a code that consists of a forward sweep (middle column) to compute the values required by the adjoint code that follows (right column). Adjoint statements are generated for all active variables occurring on right-hand sides of assignments, that is, only for components of the vector `x`. Because Equation (2) computes the adjoints in reverse order, 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 [27].

Original Code	**Forward Sweep**	**Adjoint Code**

`i=0;j=10;a=3.14159`
 	
`while (check(j)) {`
` if (max(i,j)>7) {`
 	
`  x(i)=j+sin(x(i))`
 	
` } else {`
 	
`  x(j)=`
`   j*cos(x(j))+a`
 	
` }`
 	
` i=i+1`
 	
` j=j-1; a=a/2`
 	
`}`
	`i=0;j=10;a=3.14159`
	`COUNT=0`
	`while (check(j)) {`
	` if (max(i,j)>7) {`
	`  PUSH(x(i))`
	`  x(i)=j+sin(x(i))`
	`  PUSH(true)`
	` } else {`
	`  PUSH(x(j))`
	`  x(j)=`
	`   j*cos(x(j))+a`
	`  PUSH(false)`
	` }`
	` PUSH(i)`
	` i=i+1`
	` PUSH(j)`
	` j=j-1; a=a/2`
	` COUNT=COUNT+1`
	`}`
	`PUSH(COUNT)`
		`POP(COUNT)`
		`while (COUNT>0) {`
		` COUNT=COUNT-1`
		` POP(j)`
		` POP(i)`
		` POP(test)`
		` if (test) {`
		`  POP(x(i))`
		`  x̄(i)=`
		`   cos(x(i))*x̄(i)`
		` } else {`
		`  POP(x(j))`
		`  x̄(j)=`
		`   -j*sin(x(j))`
		`   *x̄(j)`
		` }`
		`}`

$$AdjU(\texttt{x(i)=j+sin(x(i))}) = \{\texttt{x}\} \cup \{\texttt{i}\} \cup \{\texttt{i}\} = \{\texttt{x,i}\} \quad ,$$

$$AdjU(\texttt{x(j)=j*cos(x(j))+a}) = \{\texttt{x,j}\} \cup \{\texttt{j}\} \cup \{\texttt{j}\} = \{\texttt{x,j}\} \quad .$$

For example, `a` is not in the second *AdjU* set because it is not used in the adjoint instructions. Using these *AdjU* sets, TBR analysis follows the flow of the original code, looking for overwriting of these variables. When a program variable that is required for the evaluation of the adjoint (or, simply, a *required variable*) is overwritten, TBR analysis inserts a `PUSH` instruction just before the overwriting and

symmetrically restores the variable with a POP instruction before it is required in the adjoint code. In both statements above, a component of the array variable x is used 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 necessary to access the correct element of the adjoint vector $\overline{x}$. But the overwriting occurs later, and so does the PUSH/POP pair. On the other hand, a, although repeatedly overwritten, is not an element of any *AdjU* set and therefore does not need to be recorded.

3 Data Flow Analyzes

Our AD transformation is static, as are the data-flow analyses it relies on. Because static analyses have no knowledge of data or behavior at run time, most of them are *undecidable*; that is, there always exists a particular program for which the result of the analysis is uncertain. Therefore, in order to obtain safe results, conservative *over-approximations* 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* [26] 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, [1]). 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* [1]. 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. The *entry block* (resp. *exit block*) represents the beginning (resp. end) of the subroutine. No actual computation is associated with them. They may be considered as anchors for all edges entering (resp. leaving) a flow graph.

At the lowest level, the individual *instructions* are represented simply as abstract

syntax trees. [3] A symbol table is associated with each basic block giving access to properties of variables, constants, function names, type names, and so on. Symbol tables are nested to implement *lexical scoping* [1].

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. We follow this approach 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 [1], 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. Examples of this specialization process are given in Lemmas 4 and 5. The structured data-flow equations may be applicable to a smaller class of programs but are usually more efficient and also more illustrative. Solving data-flow equations defining a set S, for example, the set of all active variables, on general unstructured flow graphs with loops requires an iterative algorithm. Starting with an initial state $(\mathit{InS}_0, \mathit{OutS}_0)$, each iteration $i > 0$ computes a new state $(\mathit{InS}_i, \mathit{OutS}_i)$, which is made of two mappings from each block B to $\mathit{InS}_i(B)$ the value of S before B, and to $\mathit{OutS}_i(B)$ the value of S after B. Each iteration i applies the data-flow equations on each block B (excluding the entry and exit blocks) to build $\mathit{InS}_i(B)$ and $\mathit{OutS}_i(B)$ using the previous state $(\mathit{InS}_{i-1}, \mathit{OutS}_{i-1})$. We will use the following lemma to prove that the iterative process terminates.

Lemma 1 *Iterative resolution of data-flow equations for a set S reaches a fixed point in a finite number of iterations if the set of possible elements of S is finite and $\forall i > 0, \forall B, \mathit{InS}_{i-1}(B) \subseteq \mathit{InS}_i(B)$ and $\mathit{OutS}_{i-1}(B) \subseteq \mathit{OutS}_i(B)$.*

Proof. Since there is only a finite number of possible elements in S, there is a finite number of possible states of the iterative process. Set inclusion induces a partial order $\subseteq$ on states and the hypothesis says that for each iteration $i > 0$, $(\mathit{InS}_{i-1}, \mathit{OutS}_{i-1}) \subseteq (\mathit{InS}_i, \mathit{OutS}_i)$. If the iterative resolution does not reach a fixed

[3] Certain AD-related semantic transformations benefit from the representation of expressions as directed acyclic graphs, sharing common subexpressions. See, for example, the preaccumulation techniques for local gradients of scalar assignments in [22]. The tree representation is sufficient for the purpose of this paper.

point, all the inclusions are strict, and thus the successive states form an infinite set of states. This is impossible since there is only a finite number of different states.□

The logic behind static data-flow analysis is based on sets of (mathematical) variables. However, its result is given in terms of program variables. Due to the conservative nature of data-flow analysis the named program variables themselves represent sets of mathematical variables. A particular mathematical variable gets associated with a program variable at a given point only at runtime.

4 Activity Analysis

As explained in Section 2, activity analysis detects the variables for which a derivative must be computed, that is, 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 perform 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

The bottom-up analysis that we need here is the *differentiable dependency* analysis, which computes, for each particular structured code fragment (i.e., instruction, basic block, or subroutine) all pairs of variables $(v_b.v_a)$, v_b just *before* it and v_a just *after* it, such that $v_b \prec^+ v_a$. We call this set *Dep*. It can be implemented very efficiently as a matrix of Booleans similar to the approach taken in [21], but we shall stick to sets for the present description. We shall 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 shall 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 on the right-hand side. This set (call it *DP*) is given by the following constructive definition.

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

Here, $op \in \{+, -, *, \ldots\}$, $\varphi \in \{\sin, \exp, \tan, \ldots\}$, $e_1[e_2]$ is an array reference, [4] v is a single variable, c is a constant, and e_1, e_2 are subexpressions. All variables other than the assigned variable remain unchanged. They depend just on themselves. Thus, for an assignment I : $\mathtt{v} = e$

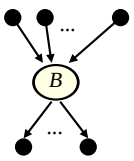
$$\mathit{Dep}(I) = \{(d.\mathtt{v}), \forall d \in DP(e)\} \cup \{(w.w), \forall w \neq \mathtt{v}\} \quad .$$

When dealing with arrays, we must overestimate *Dep* as follows: if the left-hand side v is an array reference, then some parts of the array may retain their old values, and therefore we must add dependency $(\mathtt{v}.\mathtt{v})$ to *Dep*(I).

For a basic block, and more generally for any sequence of structured program pieces p_i, we defi ne the sequential composition $\otimes$ of the *Dep* sets as follows:

$$\begin{aligned} \mathit{Dep}(p_1; p_2) &= \mathit{Dep}(p_1) \otimes \mathit{Dep}(p_2) \\ &= \{(v_b.v_a) : \exists v : (v_b.v) \in \mathit{Dep}(p_1) \wedge (v.v_a) \in \mathit{Dep}(p_2)\} \quad . \end{aligned}$$

For a subroutine S, *Dep*(S) is built from the *Dep* set of each basic block in its flow graph by using data-flow equations. For each basic block B, we introduce $InDep(B)$ (resp. $OutDep(B)$), the dependencies from the entry block of S to the *beginning* (resp. *end*) of B. The data-flow equations given in Figure 2 relate the *InDep* and the *OutDep* sets of adjacent basic blocks. These equations form a system



$$\mathit{InDep}(B) = \bigcup_{P \text{ pred. of } B} \mathit{OutDep}(P)$$

$$\mathit{OutDep}(B) = \mathit{InDep}(B) \otimes \mathit{Dep}(B)$$

Fig. 2. *General data-flow equations for differentiable dependency analysis;* As a forward data-flow analysis the set of dependencies at the entry of a block is the union of the sets of dependencies upon exit of all the immediate predecessors of the block. The block itself further modifies these dependencies. Note that when the flow graph contains loops, B can be a predecessor of itself and thus the data-flow equations must be solved iteratively.

that can be solved iteratively. *InDep*$_0(B) =$ *OutDep*$_0(B) = \emptyset$ for each block B, except the entry block for which *OutDep*$_0 =$ *Id*. Here, *Id* is the "identity", meaning that every variable depends on itself only.

[4] Note that e_1 can be another expression itself, such as in the C-expression `x[i][j]`. Program variables occurring in index expressions do not represent (mathematical) variables as in Equation (eqn:sac) and are therefore not in DP.

Lemma 2 *The solution of the data-flow equations in Figure 2 is obtained as a fixed point after a finite number of iterations.*

Proof. Since *InDep* and *OutDep* are both finite it is sufficient to show that

$$\mathit{OutDep}_i(B) \subseteq \mathit{OutDep}_{i+1}(B)$$

for $i = 0, 1, \ldots$.

By Lemma 1)

$$\mathit{InDep}_i(B) = \bigcup_{P \text{ pred. of } B} \mathit{OutDep}_{i-1}(P) \subseteq \bigcup_{P \text{ pred. of } B} \mathit{OutDep}_i(P) = \mathit{InDep}_{i+1}(B)$$

for $i = 0, 1, \ldots$. Considering in addition that $D_1 \subseteq D_2 \Rightarrow D_1 \otimes D \subseteq D_2 \otimes D$, for Dep sets D_1, D_2, and D, the above equation implies

$$\begin{aligned} \mathit{OutDep}_i(B) = \mathit{InDep}_i(B) \otimes \mathit{Dep}(B) \subseteq \\ \mathit{InDep}_{i+1}(B) \otimes \mathit{Dep}(B) = \mathit{OutDep}_{i+1}(B) \quad . \end{aligned}$$

□

The *InDep* set of the exit block of S is exactly the desired $\mathit{Dep}(S)$.

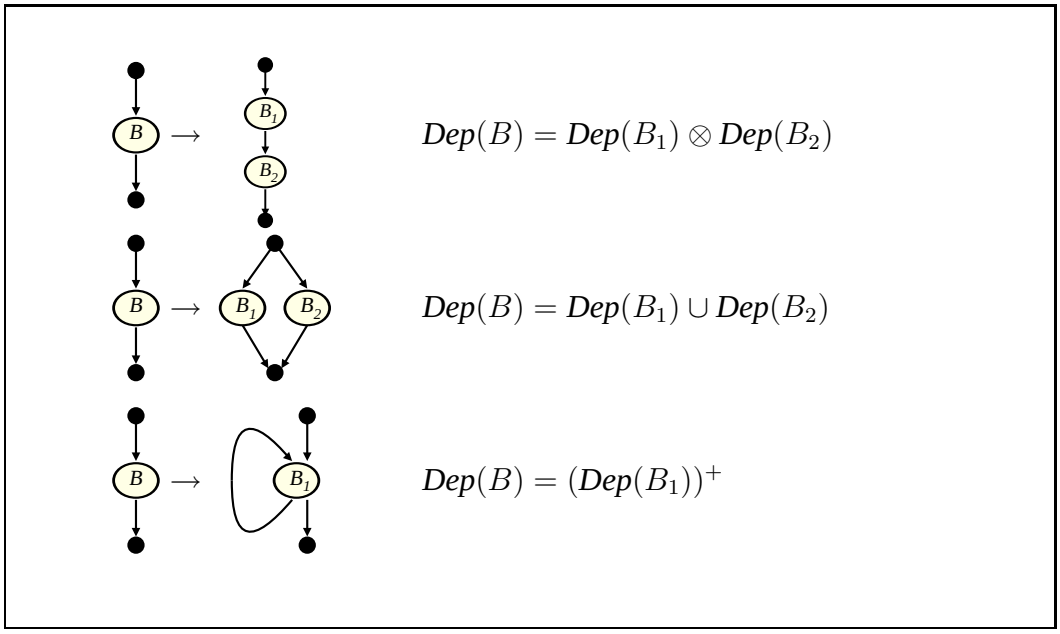


Fig. 3. *Structured data-flow equations for the differentiable dependency analysis;* For cascades of blocks the dependencies are built by sequential composition. Branches with common source and target points lead to the union of the dependencies defined in each of them. Loop dependencies are defined iteratively by computation of the closure of the loop body.

In the case of *structured* flow graphs, the data-flow equations can be specialized into structured data-flow equations as shown in Figure 3. They compute the *Dep* sets explicitly, recursively bottom-up on the structured flow graph, rather than iteratively

on the unstructured flow graph. Structured data-flow equations are less general but more efficient. In particular, there is no more iterative solving, except inside the equation for loops, to compute the closure of *Dep*(B_1), seen as a relation.

Subroutine calls are handled at the call graph level. For an individual instruction I : `call` $S(\ldots)$ the set *Dep*(I) is basically equal to *Dep*(S) (with a technical step of translating the variable names, as the name space of S differs from that of the calling subroutine). Therefore, the *Dep* set 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

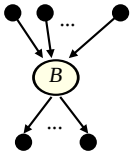
After the *Dep* sets are synthesized, activity analysis goes on propagating two data-flow sets through the program:

- The *varied* variables are variables v such that $\exists x \in \mathbf{x_I}, x \prec^+ v$. *InVary*$(p)$ (resp. *OutVary*(p)) denotes the set of varied variables just *before* (resp. *after*) a given program piece p. For the whole program P, by definition, *InVary*$(P) = \mathbf{x_I}$, which is then propagated forward on the program flow.
- The *useful* variables are variables v such that $\exists y \in \mathbf{y_D}, v \prec^+ y$. The notation *InUseful*(p) (resp. *OutUseful*(p)) is used for the set of useful variables just *before* (resp. *after*) a given program piece p. For the whole program P, by definition, *OutUseful*$(P) = \mathbf{y_D}$, which is then propagated backward on the program flow.

A variable is *active* when it is *varied* and *useful*. 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 in an order obtained by topological sorting. This approach ensures that all calls to subroutine S are analyzed before looking at S itself.

For a subroutine S, both analyses run similarly on the flow graph. The data-flow equations are shown in Figure 4 for unstructured flow graphs and in Figure 5 for some sample structured flow graphs. The *OutVary* set of the entry block of S is initialized to *InVary*(S), which is the union of the varied variables before S, on all calling contexts. Similarly, the *InUseful* set of the exit block of S is initialized to *OutUseful*(S), which is the union of the useful variables after S, on all calling contexts. Iterative resolution of the equations of Figure 4 terminates. The proof is the same as for Lemma 2. For these equations, we extended the operator $\otimes$ on sets V of variables as follows.

$$V \otimes Dep(B) = \{v_a : \exists v_b \in V : (v_b.v_a) \in Dep(B)\} \quad ,$$
$$Dep(B) \otimes V = \{v_b : \exists v_a \in V : (v_b.v_a) \in Dep(B)\} \quad .$$



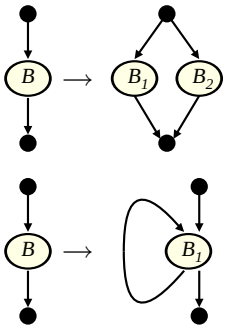
$$\mathit{InVary}(B) = \bigcup_{P \text{ pred. of } B} \mathit{OutVary}(P)$$

$$\mathit{OutVary}(B) = \mathit{InVary}(B) \otimes \mathit{Dep}(B)$$

$$\mathit{OutUseful}(B) = \bigcup_{S \text{ succ. of } B} \mathit{InUseful}(S)$$

$$\mathit{InUseful}(B) = \mathit{Dep}(B) \otimes \mathit{OutUseful}(B)$$

Fig. 4. *General data-flow equations for activity analysis;* Comments similar to those made in Figure 2 apply. The analyses of varied and useful variables are mutually symmetric if the dependencies induced by a given block are interpreted in a symmetric way.



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

$$\mathit{InVary}(B_1) = \mathit{InVary}(B) \otimes (\mathit{Id} \cup \mathit{Dep}(B))$$
$$\mathit{OutVary}(B) = \mathit{OutVary}(B_1)$$
$$\mathit{OutUseful}(B_1) = (\mathit{Id} \cup \mathit{Dep}(B)) \otimes \mathit{OutUseful}(B)$$
$$\mathit{InUseful}(B) = \mathit{InUseful}(B_1)$$

Fig. 5. *Examples of structured data-flow equations for activity analysis;* The equations for branches are straight-forward. For loops, iterative equations are avoided through direct use of $Dep(B)$, whose definition in Figure 3 is iterative.

For a subroutine call I : `call` $S(\ldots)$, the following rules accumulate the activity information for the present calling context into $\mathit{InVary}(S)$ and $\mathit{OutUseful}(S)$, their respective unions for all calling contexts. These sets are used upon initialization when analyzing S itself:

$$\mathit{InVary}(S) = \mathit{InVary}(S) \cup \mathit{InVary}(I) \quad ,$$
$$\mathit{OutUseful}(S) = \mathit{OutUseful}(S) \cup \mathit{OutUseful}(I) \quad .$$

5 TBR Analysis

As defined in Section 2, TBR analysis determines the set of variables to be recorded, namely, the variables that are effectively required in the adjoint code and whose values are lost during the remainder of the forward calculation as the result of overwriting. Thus, TBR analysis follows the flow of the original code, propagating forward the set of required variables, and flags assignments that overwrite such variables, so that their 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 structured piece p of the program, we synthesize a summary of the effect of p on TBR propagation. Concretely, this effect is composed of two parts:

- The *killed* variables *Kill*(p), those variables present at the beginning of p whose values are completely lost inside p, independent of the actual control flow inside p.
- The *adjoint-used* variables *AdjU*(p), those variables present at the end of p which are used in the adjoint code for p.

Both the $Kill$ and $AdjU$ sets are computed as sets of program variables each of which represents a set of (mathematical) variables. For all program variables in $AdjU$ we cannot guarantee that none of the (mathematical) variables they represent is used in the adjoint section of the code. Again, this is consequence of the conservative approach to static data-flow analysis.

For an individual instruction I, let us again focus on assignments. Subroutine calls are treated later, when we look at interprocedural TBR analysis. The *Kill* set of an assignment is the set of completely overwritten variables. If array region analysis is not performed, assignment to an array element does not kill the entire array and the *Kill* set is empty. As defined in Section 2, the *AdjU* set of an assignment is empty if the variable on the left-hand side is not active. Otherwise, a variable is used in the adjoint of an assignment $a = \phi(B), B = (b_1, \ldots, b_k)$, if it appears in $\frac{\partial \phi}{\partial b_i}(B)$, $i = 1, \ldots, k$, or, in other words, if it appears in a subexpression e of $\phi(B)$ that is an argument of a nonlinear operation φ, such as `sin(`e`)` or e`*v`, whose result is active. Furthermore, variables in the index expressions of the b_i and a are also used whenever the latter are array references. This leads us to the operational rules below, expressed recursively on the structure of the syntax tree. We use the same notation as for the *DP* sets in Section 4.1.

e:	$e_1{=}e_2$ (assignment)	$e_1{+}e_2 \quad e_1{-}e_2$	$\varphi(e_1)$
AdjU(e):	**if** e_2 active **then** (*AdjU*$(e_1) \cup$ *AdjU*$(e_2)) \setminus$ *Kill*$(e_1{=}e_2)$	*AdjU*$(e_1) \cup$ *AdjU*(e_2)	*Vars*(e_1)

e:	$e_1{*}e_2 \quad e_1/e_2$	$e_1[e_2]$	v	c
AdjU(e):	(**if** e_1 active **then** *AdjU*$(e_1) \cup$ *Vars*$(e_2)) \bigcup$ (**if** e_2 active **then** *AdjU*$(e_2) \cup$ *Vars*(e_1))	*AdjU*$(e_1) \cup$ *Vars*(e_2)	$\emptyset$	$\emptyset$

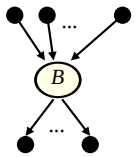
In the above, an expression is called active when its value, considered as a temporary variable, is active. *Vars*(e) is the set of all variables occurring in expression e. Concerning the *AdjU* rule for an assignment, remember that the overwritten variable represents different *mathematical* variables before and after this assignment. Therefore, the variable after the assignment is new, not yet used by any adjoint instruction, and therefore erased from the *AdjU* set. The same reasoning applies inside a basic block or to any sequence of structured program pieces (cf. top of Figure 7): The *Kill* and *AdjU* sets are jointly defined by composition of the *Kill* and *AdjU* sets of these pieces. Variables in the *Kill* set are removed from the *AdjU* set after the operation that overwrites them.

For a subroutine S, *AdjU*(S) and *Kill*(S) are built jointly and iteratively on the flow graph. For each basic block B, we introduce *InAdjU*(B) (resp. *OutAdjU*(B)) and *InKill*(B) (resp. *OutKill*(B)), the required and killed variables from the entry block of S to the *beginning* (resp. *end*) of B. The data-flow equations are given in Figure 6. They essentially state 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. These equations are solved iteratively. The initial state is *InAdjU*$_0(B) =$ *OutAdjU*$_0(B) =$ *InKill*$_0(B) =$ *OutKill*$_0(B) = \emptyset$ for each block B.

Lemma 3 *The solution of the data-flow equations in Figure 6 is obtained as a fixed point after a finite number of iterations.*

Proof. By induction on the iteration rank i, for all $i > 1$ and for all B, the inductive assumption that *OutAdjU*$_{i-1}(P) \subseteq$ *OutAdjU*$_i(P)$ applied to the first equation of Figure 6 implies that *InAdjU*$_i(B) \subseteq$ *InAdjU*$_{i+1}(B)$. Observing that *Kill*(B) and *AdjU*(B) are constant throughout the whole iterative process implies, with the third equation, that *OutAdjU*$_i(B) \subseteq$ *OutAdjU*$_{i+1}(B)$. Similarly, the second equation implies that *InKill*$_i(B) \subseteq$ *InKill*$_{i+1}(B)$, and therefore the fourth equation gives *OutKill*$_i(B) \subseteq$ *OutKill*$_{i+1}(B)$. Since on the other hand all these sets are subsets of the set of all variables they can have at most a finite number of elements and Lemma 1 ensures that the iteration terminates. □

After resolution, *InAdjU* and *InKill* of the exit block of S are exactly the desired *AdjU*(S) and *Kill*(S). In the case of structured flow graphs, the data-flow equations can be specialized, as shown in Figure 7 and proved by Lemma 4. The specialized



$$\mathit{InAdjU}(B) = \bigcup_{P \text{ pred. of } B} \mathit{OutAdjU}(P)$$

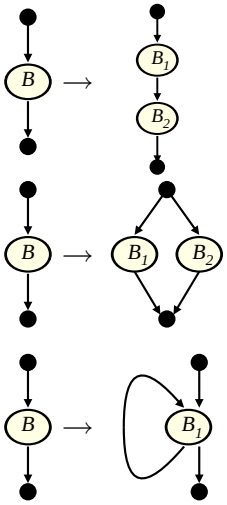
$$\mathit{InKill}(B) = \bigcap_{P \text{ pred. of } B} \mathit{OutKill}(P)$$

$$\mathit{OutAdjU}(B) = (\mathit{InAdjU}(B) \setminus \mathit{Kill}(B)) \cup \mathit{AdjU}(B)$$

$$\mathit{OutKill}(B) = \mathit{InKill}(B) \cup \mathit{Kill}(B)$$

Fig. 6. *General data-flow equations for bottom-up TBR analysis;* Variables that are used in the adjoint of the program up to the entry point of B are those that are in the corresponding sets for all predecessors of B in the flow graph. Variables that are guaranteed to be killed prior to the execution of B are guaranteed to be killed by all predecessors of B. Values of variables that are used in an adjoint statement at the exit point of B are used in the adjoint of B or in the adjoint of some predecessor of B while not being overwritten inside of B. If a value is required for the adjoint of some subblock of B and it is overwritten later in B, then this is represented in the definition of $\mathit{AdjU}(B)$ (see example below).

rules are explicit: no iterative resolution is needed.



$$\mathit{AdjU}(B) = (\mathit{AdjU}(B_1) \setminus \mathit{Kill}(B_2)) \cup \mathit{AdjU}(B_2)$$
$$\mathit{Kill}(B) = \mathit{Kill}(B_2) \cup \mathit{Kill}(B_1)$$

$$\mathit{AdjU}(B) = \mathit{AdjU}(B_1) \cup \mathit{AdjU}(B_2)$$
$$\mathit{Kill}(B) = \mathit{Kill}(B_1) \cap \mathit{Kill}(B_2)$$

$$\mathit{AdjU}(B) = \mathit{AdjU}(B_1)$$
$$\mathit{Kill}(B) = \mathit{Kill}(B_1)$$

Fig. 7. *Structured data-flow equations for bottom-up TBR analysis;* Notice that these equations are explicit: no iterative solving is needed to obtain the sets $\mathit{Kill}(B)$ and $\mathit{AdjU}(B)$, even in the case of loops. This is to be compared with the implicit equations in Figure 6, that require iterative solving for loops.

Lemma 4 *Specialization of the data-flow equations of Figure 6 to structured flow graphs gives the equations of Figure 7.*

Proof. Consider each structured flow-graph as a small subroutine B to which we apply the unstructured data-flow equations for $\mathit{AdjU}(B)$ and $\mathit{Kill}(B)$, which are, by definition, equivalent to *InAdjU* and *InKill* of the exit block. The only nontrivial

part of the proof is for the loop. We find

$$\mathit{AdjU}(B) = \mathit{OutAdjU}(B_1) = (\mathit{InAdjU}(B_1) \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1) \quad .$$

Two arrows reach B_1, one from the entry block. According to the unstructured equation we get

$$\mathit{InAdjU}(B_1) = \mathit{OutAdjU}(B_1) \cup \emptyset = \mathit{OutAdjU}(B_1) \quad ,$$

and $\mathit{AdjU}(B)$ appears as the solution of the fixed point equation

$$X = (X \setminus \mathit{Kill}(B_1)) \cup \mathit{AdjU}(B_1) \quad .$$

Since $\mathit{OutAdjU}_0(B_1)$ is initialized to $\emptyset$, the above equation obviously reaches its fixed point for $X = \mathit{AdjU}(B_1)$. The same proof applies to the *Kill* set. □

5.1.0.1 Example Consider

$$\begin{array}{ll} I1: & \mathtt{y} = \mathtt{sin}(\mathtt{x1} + \mathtt{x2}) \\ I2: & \mathtt{z} = \mathtt{y} \\ I3: & \mathtt{x1} = \mathtt{x2} \\ I4: & \mathtt{v} = \mathtt{z} * \mathtt{x2} \quad . \end{array}$$

An adjoint code is

$$\begin{array}{ll} \bar{I4}: & \bar{\mathtt{z}} += \bar{\mathtt{v}} * \mathtt{x2} \\ & \bar{\mathtt{x2}} += \bar{\mathtt{v}} * \mathtt{z} \\ \bar{I3}: & \bar{\mathtt{x2}} += \bar{\mathtt{x1}} \\ \bar{I2}: & \bar{\mathtt{y}} += \bar{\mathtt{z}} \\ \bar{I1}: & \bar{\mathtt{x1}} = \mathtt{cos}(\mathtt{x1} + \mathtt{x2}) * \bar{\mathtt{z}} \\ & \bar{\mathtt{x2}} += \bar{\mathtt{x1}} \quad . \end{array}$$

Application of the first two equations from Figure 7 results in

$$\begin{array}{ll} \mathit{AdjU}(I1) = \{\mathtt{x1}, \mathtt{x2}\}; & \mathit{Kill}(I1) = \{\mathtt{y}\} \\ \mathit{AdjU}(I2) = \varnothing; & \mathit{Kill}(I2) = \{\mathtt{z}\} \\ \mathit{AdjU}(I3) = \varnothing; & \mathit{Kill}(I3) = \{\mathtt{x1}\} \\ \mathit{AdjU}(I4) = \{\mathtt{x1}, \mathtt{z}\}; & \mathit{Kill}(I4) = \{\mathtt{v}\} \quad . \end{array}$$

Let $B1 = (I1, I2)$ and $B2 = (I3, I4)$. Then

$$\begin{array}{ll} \mathit{AdjU}(B1) = \{\mathtt{x1}, \mathtt{x2}\}; & \mathit{Kill}(B1) = \{\mathtt{y}, \mathtt{z}\} \\ \mathit{AdjU}(B2) = \{\mathtt{x1}, \mathtt{z}\}; & \mathit{Kill}(B2) = \{\mathtt{x1}, \mathtt{v}\} \quad . \end{array}$$

Finally, if $B3 = (B1, I3)$, then

$$AdjU(B3) = \{\texttt{x2}\}; \quad Kill(B1) = \{\texttt{y}, \texttt{z}, \texttt{x1}\} \quad .$$

The bottom-up character of TBR analysis ensures that the value of x1 is saved inside $B3$ before being overwritten by $I3$. The new value is not required for any adjoint statement in $B3$. Only for $B4 = (B3, I4)$ we get $AdjU(B4) = \{\texttt{x1}, \texttt{x2}\}$ since any statement succeeding $B4$ that overwrites x1 or x2 violates the correctness of the adjoint for $B4$. Again, the values of x1 or x2 have to be recorded.

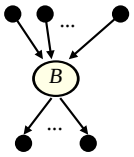
5.2 Top-Down TBR Analysis

After the *AdjU* and *Kill* sets are synthesized, the second step of TBR analysis computes and propagates the *required* variables, whose present value is possibly used by the adjoint of some previous instruction. *InReq*(p) (resp. *OutReq*(p)) denotes the set of the required variables just *before* (resp. *after*) a given program piece p. Each time an individual instruction *overwrites* a required variable (i.e., a variable present in the *InReq* set), we flag the overwritten variable as "to be recorded," and a PUSH/POP pair is inserted in the automatically generated derivative code. For the whole program P, *InReq*(P) is initialized to $\emptyset$ and is then propagated forward on the program flow. Subroutines are swept top-down on the call graph in an order obtained by topological sorting. This procedure ensures that a called subroutine is analyzed after *all* of its calling sites have been analyzed.

For a subroutine S, the data-flow equations are shown in Figure 8. The *InReq* set of the entry block of S is initialized to *InReq*(S), which is the union of the required variables before the call to S, on all calling contexts. Iterative resolution of the equations of Figure 8 terminates. The proof is the same as for Lemma 3. The second equation is obtained by the following reasoning: Suppose that the value of a program variable v is required at the entry of B. If v is overwritten inside B then a PUSH/POP pair is inserted in the automatically generated code and the status of v is set to "not required." The program variable keeps this status until the end of B unless it is in $AdjU(B)$ and therefore required to compute some adjoint statement that is not succeeded by another assignment to v.

Figure 9 shows specialized data-flow equations for some sample structured flow graphs, with proofs given in Lemma 5. The specialized rules are explicit; that is, no iterative resolution is needed. Consequently, *TBR analysis of structured subroutines requires no iteration to correctly establish the status of any variable that is overwritten by some assignment within the subroutine*. This result was demonstrated in [23]. Here, Lemmas 4 and 5 are a reformulation in the present formalism.

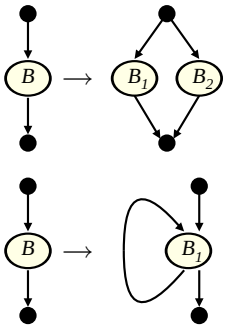
Lemma 5 *Specialization of the data-flow equations of Figure 8 to structured flow graphs gives the equations of Figure 9.*



$$\mathit{InReq}(B) = \bigcup_{P \text{ pred. of } B} \mathit{OutReq}(P)$$

$$\mathit{OutReq}(B) = (\mathit{InReq}(B) \setminus \mathit{Kill}(B)) \cup \mathit{AdjU}(B)$$

Fig. 8. *General data-flow equations for top-down TBR analysis*



$$\mathit{InReq}(B_1) = \mathit{InReq}(B_2) = \mathit{InReq}(B)$$

$$\mathit{OutReq}(B) = \mathit{OutReq}(B_1) \cup \mathit{OutReq}(B_2)$$

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{AdjU}(B_1)$$

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

Fig. 9. *Structured data-flow equations for top-down TBR analysis*

Proof. The only nontrivial part is about the loop. The general data-flow equations from Figure 8, specialized for the structured loop, yield

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

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

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

which is a fixed point equation for $\mathit{InReq}(B_1)$. Since $\mathit{InReq}_0(B_1)$ is initialized to $\emptyset$, the fixed point is reached in two steps, giving

$$\mathit{InReq}(B_1) = \mathit{InReq}(B) \cup \mathit{AdjU}(B_1) \quad ,$$

which is the first data-flow equation for structured loops. Similarly, $\mathit{OutReq}(B)$ is equal to $\mathit{OutReq}(B_1)$ and

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

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

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

□

Finally, at the instruction level, the following rule propagates the "required" information forward across any instruction I:

$$\mathit{OutReq}(I) = (\mathit{InReq}(I) \setminus \mathit{Kill}(I)) \cup \mathit{AdjU}(I) \quad .$$

Variables overwritten by I must be flagged as "to be recorded" if they belong to *InReq*(I). In addition, for a subroutine call I : `call` $S(\ldots)$, the following rule accumulates the required information for the present calling context into *InReq*(S). A top-down sweep on the call graph ensures that *InReq*(S) eventually contains the union for all calling contexts when S itself is analyzed; i.e.,

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

6 Case Study and Experimental Results

We have applied TAPENADE with and without TBR analysis to a variant of the Bratu problem [2]. 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 they 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. More recently, we made measurements on a Navier-Stokes solver differentiated in the reverse mode of AD by TAPENADE, more specifically on the part of the solver that assembles the second-order state equation residual [16]. The following table gives the results both in execution time and tape memory space. In addition to the two extreme approaches, namely with and without TBR analysis, we experimented with a simpler analysis that records variables only if they have been initialized. The execution time of the original nondifferentiated subprograms is 0.15 *s*.

	AD (no TBR)	**AD (only initialized)**	**AD (TBR)**
memory *(Mb)***:**	2.09	0.38	0.12
run time *(s)***:**	1.01	0.91	0.77

7 Conclusion

Data-flow analysis is a very powerful tool for generating efficient tangent-linear or adjoint code by forward or reverse mode automatic differentiation, respectively. Knowing the activity status of a (program) variable may result in significant gains regarding the computational complexity by saving a potentially large number of trivial (multiplications by zero) operations. Building on this knowledge we have proposed TBR analysis with the objective to exploit linearity leading to a significant decrease in the memory requirement of reverse mode. The potential benefits were demonstrated on a large industrial code.

We believe that data-flow analyses that combine classical compiler concepts with doman-specific (e.g., mathematical) knowledge will play an increasingly important role in the future. In particular, semantic transformations of numerical program may benefit greatly from such techniques.

Acknowledgments

While working at the Mathematics and Computer Science Division of Argonne National Laboratory until June 2004 Naumann was partly supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38. Further funding was provided by NSF under ITR contract OCE-0205590.

A Bratu Problem

```
      subroutine bratu(dim,parmax,x,prm,F)
      integer dim, parmax, i
      double precision x(dim), prm(parmax), F(dim)
      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 BRATU_B(dim, parmax, x, xb, prm, prmb, f, fb)
        INTEGER dim, parmax
        DOUBLE PRECISION f(dim), fb(dim)
        DOUBLE PRECISION prm(parmax), prmb(parmax), x(dim), xb(dim)
        REAL*8 exp1, exp11, exp11b, exp13, exp13b, exp1b, exp3,
     +         exp3b, exp5, exp5b, exp7, exp7b, exp9, exp9b
        REAL*8 exp10, exp10b, exp12, exp12b, exp14, exp14b, exp2,
     +         exp2b, exp4, exp4b, exp6, exp6b, exp8, exp8b
        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
      exp10b = exp10b + h * h * prm(1) * fb(i+1) / 12.0
      exp9b = exp9b + EXP(exp9) * exp10b
      xb(i) = xb(i) + fb(i+1) + (1 / (1.0+prm(2)*x(i)) - x(i)
+       * prm(2) / (1.0+prm(2)*x(i))**2) * exp9b
      prmb(1) = prmb(1) + exp10 * h * h * fb(i+1) / 12.0
      fb(i+1) = 0.D0
      CALL POPREAL8(exp10)
      exp10b = 0.0
      CALL POPREAL8(exp9)
      prmb(2) = prmb(2) - x(i) * x(i) * exp9b / (1.0+prm(2)*x(
+       i))**2
      exp9b = 0.0
      exp8b = exp8b + h * h * prm(1) * fb(i) / 1.2
      exp7b = exp7b + EXP(exp7) * exp8b
      exp6b = exp6b + h * h * prm(1) * fb(i-1) / 12.0
      exp5b = exp5b + EXP(exp5) * exp6b
      xb(i) = xb(i) + ((1 / (1.0+prm(2)*x(i)) - x(i) * prm(2) /
+       (1.0+prm(2)*x(i))**2) * exp7b - 2 * fb(i) + fb(i-1)) +
+       (1 / (1.0+prm(2)*x(i)) - x(i) * prm(2) /
+       (1.0+prm(2)*x(i))**2) * exp5b
      prmb(1) = prmb(1) + exp8 * h * h * fb(i) / 1.2
      CALL POPREAL8(exp8)
      exp8b = 0.0
      CALL POPREAL8(exp7)
      prmb(2) = prmb(2) - x(i) * x(i) * exp7b / (1.0+prm(2)*x(
+       i))**2
      exp7b = 0.0
      prmb(1) = prmb(1) + exp6 * h * h * fb(i-1) / 12.0
      CALL POPREAL8(exp6)
      exp6b = 0.0
      CALL POPREAL8(exp5)
```

```
      prmb(2) = prmb(2) - x(i) * x(i) * exp5b / (1.0+prm(2)*x(
     +      i))**2
      exp5b = 0.0
    ENDDO
    ...
  END
```

References

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

[2] 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.

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

[4] C. Bischof, A. Carle, P. Khademi, and A. Maurer. The ADIFOR 2.0 system for automatic differentiation of Fortran 77 programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.

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

[6] M. Cohen, U. Naumann, and J. Riehme. Toward differentiation-enabled Fortran 95 compiler technology. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 143–147, 2003.

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

[8] C. Faure and U. Naumann. The taping problem in automatic differentiation. In *[7]*, 2001.

[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. Recipes for adjoint code construction. *ACM Trans. Math. Software*, 24:437–474, 1998.

[11] R. Giering and T. Kaminski. Towards an optimal trade-off between recalculation and taping in reverse mode AD. In *[7]*, 2001.

[12] R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of Applied Mathematics in Mechanics*, volume 2, pages 54–57, 2003.

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

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

[15] A. Griewank and G. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.

[16] L. Hascoët, M. Vazquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V. Kumar, M. Gavrilova, C. Tan, and P. Ecuyer, editors, *Computational Science and its Applications – ICCSA 2003*, number 2668 in LNCS, pages 85–94. Springer, 2003.

[17] M. Iri. History of automatic differentiation and rounding estimation. In *[15]*, pages 1–16. SIAM, 1991.

[18] J. Knoop. *Optimal Interprocedural Program Optimization*. Number 1428 in LNCS Tutorial. Springer, New York, 1998.

[19] K. Kubota. PADRE2 - Fortran precompiler for automatic differentiation and estimates of rounding errors. In *[3]*, pages 367–374. SIAM, 1996.

[20] S. Lee and P. Hovland. Sensitivity analysis using parallel ODE solvers and automatic differentiation in C: SensPVODE and ADIC. In *[7]*, chapter 26, pages 223–229. Springer, New York, NY, 2001.

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

[22] U. Naumann. Optimal pivoting in tangent-linear and adjoint systems of nonlinear equations. Technical Report ANL/MCS-P944-0402, Mathematical and Computer Science Division, Argonne National Laboratory, April 2002.

[23] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Computational Science – ICCS 2002, Part II*, volume 2330 of *LNCS*, pages 1039–1048, Berlin, 2002. Springer.

[24] L. Rall and G. Corliss. An introduction to automatic differentiation. In *[3]*, pages 1–17. SIAM, 1996.

[25] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A:558–568, 1993.

[26] 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.

[27] The TROPICS Team. Tapenade 2.0. `http://www-sop.inria.fr/tropics`, 2003.