1

The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications

Laurent Hascoët and Mauricio Araya-Polo

INRIA Sophia-Antipolis, TROPICS team
2004 Route des lucioles, BP 93, 06902 Sophia-Antipolis, France
Laurent.Hascoet@sophia.inria.fr, Mauricio.Araya@sophia.inria.fr

Summary. Automatic Differentiation (AD) is a program transformation that yields derivatives. Building efficient derivative programs requires complex and specific static analysis algorithms to reduce run time and memory usage. Focusing on the *reverse mode* of AD, which computes *adjoint programs*, we specify jointly the central static analyses that are required to generate an efficient adjoint code. We use a set-based formalization from classical data-flow analysis to specify *Adjoint Liveness*, *Adjoint Write*, and *To Be Recorded* analyses, and their mutual influences, taking into account the specific structure of adjoint programs. We give illustrations on examples taken from real numerical programs, that we differentiate with our AD tool TAPENADE, which implements these analyses.

Key words: Automatic Differentiation, Adjoint code, Adjoint algorithm, Data-Flow analysis, Reverse Mode

1.1 Introduction

Classically, tools that perform code optimization require information on which variables are used or overwritten by a given piece of code. This is particularly true when trying to optimize *adjoint* code produced by the reverse mode of Automatic Differentiation (AD), which poses serious problems of run time and memory consumption. To this end, in addition to the classical program analyses, e.g. Read-Write, several research groups have experimented specific analyses such as *Adjoint Liveness* analysis, *To Be Recorded* analysis, and *Adjoint Write* analysis. These three analyses, defined in sect. 1.4, appear tightly related.

Adjoint code has a particular structure, defined by the model of reverse AD. For example, an adjoint code consists of two sweeps with symmetric control flow [9, 8]. It also features matching pairs of instructions that *store* and *restore* values. In our particular model this is done by pushing and popping these values to and from a stack. We are going to use these features of adjoint

programs to define the rules of the AD-specific data-flow analyses, by formal specialization of the rules of classical data-flow analyses.

Notice that generic data-flow analyzers, like the ones found in optimizing compilers, are unable to detect nor take advantage of this structure. They can't find out the pairs of matching push and pop's far apart, nor can they understand the mechanism used to reproduce the symmetric flow of control. Running them a posteriori on the adjoint program will return weak results. Therefore, we shall define AD-specific data-flow analyses that will run on the original code before generation of the adjoint code, incorporating knowledge on how the adjoint code will be built.

The goal of this paper is to give a formal uniform specification of the three adjoint data-flow analyses above, defined on the structure of the original code. This specification relies on a preliminary description of the model of reverse AD that goes from the original code to its adjoint. This specification will be used to demonstrate data-flow properties of adjoint codes, to highlight the relationship between these three analyses, and to derive the data-flow equations implemented in our AD tool TAPENADE [8].

Outline: sect. 1.2 summarizes the knowledge about reverse AD that is necessary for this paper, and sect. 1.3 gives basic notation and formulae about classical data-flow analyses. We refer to [6] for a full discussion about AD, and to [1] about data-flow analyses. Section 1.4 presents the reverse AD model and uses it to define and specify Adjoint Liveness, "To Be Recorded", and Adjoint Write analyses. Section 1.5 gives an illustrative example adapted from an industrial numerical code and shows experimental measurements.

1.2 Adjoints by Automatic Differentiation

Automatic Differentiation differentiates *programs*. An AD tool takes as input a source computer program P that, given a vector argument $X \in I\!R^n$, computes some vector function $Y = F(X) \in I\!R^m$. We call F' the Jacobian of F. A star * denotes transposition, and the dot $\cdot$ denotes product. In reverse mode, the AD tool generates an adjoint source program $\overline{\mathtt{P}}$ that, given the argument X and a weight vector $\overline{Y}$, computes the gradient $F'^*(X) \cdot \overline{Y}$ of the scalar output $Y^* \cdot \overline{Y}$. To keep things simple, consider that P is a sequence of instructions I_k : AD identifies it with a composition of vector functions so that

$$\mathtt{P} : [I_1; I_2; \dots I_p;] \quad \text{represents} \quad F = f_p \circ f_{p-1} \circ \cdots \circ f_1,$$

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

$$F'^*(X) \cdot \overline{Y} = f_1'^*(X_0) \cdot f_2'^*(X_1) \cdot \ \dots \ \cdot f_{p-1}'^*(X_{p-2}) \cdot f_p'^*(X_{p-1}) \cdot \overline{Y}, \qquad (1.1)$$

which can be mechanically translated back into the adjoint program $\overline{\mathtt{P}}$.

We observe that (1.1) is most efficiently computed from right to left, because matrix×vector products are cheaper than matrix×matrix products. This yields the gradient in a time which is only a small multiple of the time of P. However, there is a difficulty because the f' instructions require the intermediate values X_k in *reverse* of their computation order. If the original program *overwrites* a part of X_k, the differentiated program must restore X_k before it is used by $f'^*_{k+1}(X_k)$. There are two main strategies for that:

- **Recompute-All (RA):** the X_k are recomputed when needed, restarting P on input X_0 until instruction I_k. The brute-force RA strategy has a quadratic time cost with respect to the number of run-time instructions p. The TAF [5] tool uses this strategy. A so-called "Efficient Recomputation Algorithm" limits recomputations to those actually needed to obtain X_k.
- **Store-All (SA):** the X_k are stored on a stack. This stack is filled during the *forward sweep* $\overrightarrow{\mathtt{P}}$, a preliminary run of P that stores variables on the stack just before they are overwritten. The differentiated instructions strictly speaking form the *backward sweep* $\overleftarrow{\mathtt{P}}$, which pops values from the stack when needed. The brute-force SA strategy has a linear memory cost with respect to p. The ADIFOR [2] and TAPENADE [8] tools use this strategy.

Practically, both RA and SA strategies need a special storage/recomputation trade-off in order to be really efficient. This trade-off is called *checkpointing*. Since TAPENADE uses checkpointing on subroutine calls, we describe checkpointing in this context. Let us define some vocabulary and graphical no-

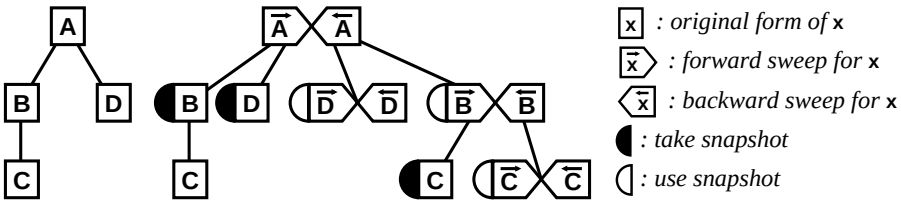

Fig. 1.1. Checkpointing on calls in TAPENADE reverse AD

tations. Execution of a subroutine A in its original form is shown as A. The *forward sweep* is shown as $\overrightarrow{\mathtt{A}}$. The *backward sweep* is shown as $\overleftarrow{\mathtt{A}}$. The adjoint program is just $\overline{\mathtt{A}} = \overrightarrow{\mathtt{A}}; \overleftarrow{\mathtt{A}}$. Checkpointing consists of choosing a part B of A, which will be run *without* storage during $\overrightarrow{\mathtt{A}}$. When the backward sweep $\overleftarrow{\mathtt{A}}$ reaches B, it runs $\overrightarrow{\mathtt{B}}$, i.e. B again but this time with storage and then immediately runs $\overleftarrow{\mathtt{B}}$ and the rest of $\overleftarrow{\mathtt{A}}$. Duplicate execution of B requires that some variables used by B (a *snapshot*) be stored. TAPENADE applies checkpointing at each procedure call, leading to the pattern shown on fig. 1.1. If the run-time call tree is well balanced, the memory size as well as the computation time required for the reverse differentiated program grow only like the depth of the call tree, i.e. like the logarithm of the run time of P.

1.3 Classical Data-Flow Analyses

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

- The set of variables whose value at the beginning of Z is overwritten inside Z (at least partly overwritten during some possible execution of Z) is denoted by $\mathbf{out}(Z)$. For two successive pieces of program A and B:

$$\mathbf{out}(A;B) = \mathbf{out}(A) \cup \mathbf{out}(B). \tag{1.2}$$

  However we will use a refined rule when a stack is used: if a variable is PUSH'ed and later POP'ed from a stack, it is unmodified globally, so that

$$\mathbf{out}(\mathrm{PUSH}(v);A;\mathrm{POP}(v)) = \mathbf{out}(A) \setminus \{v\}. \tag{1.3}$$

- The set of variables whose value at the beginning of Z is always completely overwritten inside Z is denoted $\mathbf{kill}(Z)$. This subset of $\mathbf{out}(Z)$ is often a strict subset, e.g. a single assignment to an array cell does not kill this array, so that $\mathbf{kill}(\texttt{T(i)=0.0}) = \emptyset$. Array region analysis [3] copes with this in some cases. In general for two successive pieces of program A and B we take the conservative under-approximation:

$$\mathbf{kill}(A;B) = \mathbf{kill}(A) \cup \mathbf{kill}(B).$$

- The set of variables whose value at the beginning of Z is read inside Z is denoted $\mathbf{use}(Z)$. For two successive pieces of program A and B, the variables killed by A hide the variables read by B, so that:

$$\mathbf{use}(A;B) = \mathbf{use}(A) \cup (\mathbf{use}(B) \setminus \mathbf{kill}(A)). \tag{1.4}$$

- When Z is a tail of `P` (i.e. the end of Z is the end of `P`), we define the set $\mathbf{live}(Z)$ of live variables at the beginning of Z, i.e. whose value is involved in computations that eventually influence the final result of `P`. All final outputs of `P` are live by definition, so we initialize $\mathbf{live}([\,])$ to this set. Recursively, for any two successive pieces of program A and B, B being a tail of `P`, the variables live just before B lead to the variables live just before A through $\mathbf{Dep}(A)$, the "dependence across A" information, defined as

$$\mathbf{Dep}(A) = \{(v_o, v_i) \in \mathrm{Outputs}(A) \times \mathrm{Inputs}(A) \mid v_o \text{ depends on } v_i\}$$

  and through the combinator $\otimes$, defined as

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

  so that:

$$\mathbf{live}(A;B) = \mathbf{live}(B) \otimes \mathbf{Dep}(A). \tag{1.5}$$

1.4 Adjoint Data-Flow Analyses

We consider a piece of program P, to be differentiated by the reverse mode of AD into its adjoint program $\overline{\mathtt{P}}$. P can be the complete function that will be differentiated, or it can be a checkpointed sub-part. In both cases, using notation from sect. 1.2, $\overline{\mathtt{P}} = \overrightarrow{\mathtt{P}}; \overleftarrow{\mathtt{P}}$. *Adjoint Liveness* analysis observes that the only required results of $\overline{\mathtt{P}}$ are the differentiated variables, and *not* the original results of P, which in most implementations will be overwritten and lost during $\overleftarrow{\mathtt{P}}$. Therefore, several instructions at the end of $\overrightarrow{\mathtt{P}}$ may be dead. This is true in particular for the last instruction of P. *Adjoint Liveness* analysis computes, for any location in P and thus in $\overrightarrow{\mathtt{P}}$, the set of original variables *live* for $\overline{\mathtt{P}}$. Adjoint Liveness is computed on original variables only, but it originates from differentiated variables, which are all considered live, whereas all the original variables are assumed dead at the end of $\overline{\mathtt{P}}$. *Adjoint Liveness* analysis is strongly related to the *To Be Recorded* analysis and the *Adjoint Write* analysis, so that we will define and study the three of them jointly.

Here is an outline of the general structure of this technical section. In sect. 1.4.1, we first give a precise specification (or model) of adjoint programs, that defines and uses the *Adjoint Liveness*, *To Be Recorded*, and *Adjoint Write* analyses, in order to produce an efficient code. Then we formalize these analyses using this model of adjoint programs, starting with *To Be Recorded* analysis in sect. 1.4.2. Notice that this might introduce a circularity into the definition. After proving in sect. 1.4.3 an important lemma about the variables left unmodified by an adjoint program, we will be able in sect. 1.4.4 to formally derive specific rules that define the *Adjoint Liveness* analysis. We can then show that the definitions circularity mentioned above disappears, and consequently the *Adjoint Liveness* analysis must be run first, followed by the *To Be Recorded* analysis and finally by the *Adjoint Write* analysis. Section 1.4.5 formally derives the specific rules that define the *Adjoint Write* analysis, and highlights its usage for the checkpointing strategy. All three analyses are defined directly on the original program, with a low computational cost.

1.4.1 Structure of Adjoint Programs

Strictly speaking, the fact that a variable is necessary for (a part of) $\overline{\mathtt{P}}$ depends on the architecture of $\overline{\mathtt{P}}$, i.e. on the reverse AD model and on the strategy used to make intermediate values available to the backward sweep. Here, we shall rely on the SA strategy used in TAPENADE, but the following specifications and demonstrations can be adapted to the RA strategy. Let us make our reverse AD model explicit. We define $\overline{\mathtt{P}}$ recursively on the structure of P. To keep things simple, suppose P is a straight-line program of simple statements. For an empty program $\mathtt{P} = []$, $\overline{\mathtt{P}}$ is simply $\overline{[\,]} = []$. Recursively, for a assignment I followed by any "downstream" sequel D, the basic model is:

$$\overline{I;D} = \overrightarrow{I};\overline{D};\overleftarrow{I} = \mathrm{PUSH}(\mathbf{out}(I));I;\overline{D};\mathrm{POP}(\mathbf{out}(I));I'. \qquad (1.6)$$

It states that values overwritten by I are PUSH'ed onto a stack just before I, and restored by a POP before they may be used by I', the derivative instructions for I. However, at least three refinements can be applied to the model to obtain a more efficient, yet equivalent, adjoint code.

An immediate refinement is to use *activity*, specified for example in [7]: at analysis time, some variables can be proved to have always a zero derivative with respect to the independent inputs or dependent outputs. When the variable written by assignment I is inactive, then I' can be removed. When some variable used by assignment I is inactive, I' is simplified, therefore using fewer intermediate variables. Adjoint data-flow analyses yields even better results when run after activity analysis.

Another refinement is to use Adjoint Liveness analysis, that we define as computing $\mathbf{live}(\overline{D})$ with the initial condition on the tail of D that $\mathbf{live}(\overline{[\,]}) = \emptyset$. In model (1.6), $\overline{I;D}$ contains instruction I. We observe that if the results of I are used later in $\overline{\mathtt{P}}$, this can be only in $\overline{D}$ because the backward sweep $\overleftarrow{\overline{U;I}}$ of I and instructions before I in `P` ("upstream") only uses intermediate values that existed *before* execution of I, and certainly not the results of I. Therefore, I and its associated PUSH and POP can be removed if its results are out of $\mathbf{live}(\overline{D})$, i.e. if the predicate *adj-live*(I, D), defined as $(\mathbf{out}(I) \cap \mathbf{live}(\overline{D}) \neq \emptyset)$, is *false*. One can check this is the case for the last I of `P`, i.e. when $D = [\,]$.

The third refinement is to PUSH and POP a variable in $\mathbf{out}(I)$ *only* if it is really required by the following differentiated instructions. This is the goal of *To Be Recorded* analysis, abbreviated as TBR [4, 10]. For example, the derivative of the "linear" instruction `x = y+2*z` does not require the values of `y` nor `z`. Since this refinement depends on the *following* differentiated instructions, which include the backward sweep of U, the part of `P` *upstream* I, we must introduce this U as a context into definition (1.6). We use the notation $\vdash$ to separate U from the part of the program currently differentiated. We introduce the set of variables used by instructions I' and after, which is $\mathbf{use}(I'; \overleftarrow{U})$. The only variables actually PUSH'ed and POP'ed for instruction I are now $\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})$.

Consequently, model (1.6) turns into the following refined model:

$$\begin{aligned} U \vdash \overline{I;D} = {} & [\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I;] \ \textit{if adj-live}(I, D) \\ & [U; I] \vdash \overline{D}; \\ & [\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U}));] \ \textit{if adj-live}(I, D) \\ & I' \end{aligned} \tag{1.7}$$

1.4.2 Derived Rules for TBR Analysis

From the classical equation (1.4) of the **use** analysis, we can write the rules that compute $\mathbf{use}(I'; \overleftarrow{U})$ and $\mathbf{use}(\overleftarrow{U})$, yielding a formal specification of the TBR analysis. Since I' only overwrites differentiated variables, and we study here data-flow properties of the original variables only, $\mathbf{kill}(I') = \emptyset$. Therefore

$$\mathbf{use}(I'; \overleftarrow{U}) = \mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U}), \tag{1.8}$$

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

$$\mathbf{use}(\overleftarrow{[\,]}) = \mathbf{use}([\,]) = \emptyset$$

$$\mathbf{use}(\overleftarrow{U;I}) = \begin{cases} \mathbf{use}(\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I'; \overleftarrow{U}) \\ \quad = (\mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U})) \setminus \mathbf{kill}(I) \quad \textit{if adj-live}(I, D) \\ \\ \mathbf{use}(I'; \overleftarrow{U}) = \mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U}) \qquad \textit{otherwise} \end{cases} \tag{1.9}$$

These equations translate easily into forward data-flow equations. They can be implemented efficiently as described in [7].

1.4.3 Adequacy Lemma for the PUSH/POP Mechanism

Equations (1.8) (1.9) allow us to verify an important property of model (1.7): the PUSH/POP mechanism inside $\overline{D}$ does leave unchanged all variables used in $\overleftarrow{U;I}$, the backward sweep of the upstream instructions:

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

$$\mathbf{out}(U \vdash \overline{Z}) \cap \mathbf{use}(\overleftarrow{U}) = \emptyset.$$

Proof. By induction on the length of Z. **Terminal case:** if $Z = [\,]$, $U \vdash \overline{Z} = [\,]$ too, so its **out** set is empty, and the property is true. **Induction case:** if $Z = I; D$, then $U \vdash \overline{Z}$ is defined by (1.7):

- If *adj-live*$(I; D)$ is *false*, then $\mathbf{out}(U \vdash \overline{I;D}) = \mathbf{out}([U;I] \vdash \overline{D}; I') = \mathbf{out}([U;I] \vdash \overline{D}) \cup \mathbf{out}(I')$, from definition (1.2). We know that $\mathbf{out}(I') = \emptyset$ because I' overwrites only differentiated variables. By the induction hypothesis, $\mathbf{out}([U;I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U;I}) = \emptyset$. From (1.9), we find that $\mathbf{use}(\overleftarrow{U;I})$ contains $\mathbf{use}(\overleftarrow{U})$ and therefore $\mathbf{out}([U;I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U}) = \emptyset$, and the property is true.
- On the other hand if *adj-live*$(I; D)$ is *true*, consider any variable $v \in \mathbf{use}(\overleftarrow{U})$. This implies through (1.8) that $v \in \mathbf{use}(I'; \overleftarrow{U})$.
  - Either $v \in \mathbf{out}(I)$. Since *adj-live*$(I; D)$ is *true*, and since v is also in $\mathbf{use}(I'; \overleftarrow{U})$, v will be PUSH'ed and then POP'ed. Thus from (1.3), v is unchanged just after the POP. Since I' overwrites only differentiated variables, v is unchanged through execution of $U \vdash \overline{I;D}$.
  - Or $v \notin \mathbf{out}(I)$. In that case, the only part of $U \vdash \overline{I;D}$ that might overwrite v is $[U;I] \vdash \overline{D}$. Equation (1.9) says that $\mathbf{use}(\overleftarrow{U;I}) = (\mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U})) \setminus \mathbf{kill}(I)$. Since $v \notin \mathbf{out}(I)$, $v \notin \mathbf{kill}(I)$ because the **kill** set is always included in the **out** set. So from $v \in \mathbf{use}(\overleftarrow{U})$ we

get $v \in \mathbf{use}(\overleftarrow{U;I})$. The induction hypothesis ensures that $\mathbf{out}([U;I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U;I}) = \emptyset$, and therefore $v \notin \mathbf{out}([U;I] \vdash \overline{D})$, so v is unchanged through execution of the whole $U \vdash \overline{I;D}$.

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

1.4.4 Derived Rules for Adjoint Liveness Analysis

We can now specify the rules of Adjoint Liveness analysis. We want to find equations which, for any tail Z of P, build the set $\mathbf{live}(U \vdash \overline{Z})$. By definition $\mathbf{live}(\overline{[\,]}) = \emptyset$. Recursively for $Z = I;D$, recall that Adjoint Liveness originates from differentiated variables. In model (1.7), only $\overline{D}$ and I' write differentiated variables. Therefore $\mathbf{live}(U \vdash \overline{I;D})$ is the union of the necessary variables of the two slices of $U \vdash \overline{I;D}$ required for $\overline{D}$ and I'.

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

$$[\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I';\overleftarrow{U}));I;] \text{ if } \textit{adj-live}(I,D)$$
$$[U;I] \vdash \overline{D};$$

  From (1.5), the necessary variables are $\mathbf{live}([U;I] \vdash \overline{D}) \otimes \mathbf{Dep}(I)$. This formula applies even when $\textit{adj-live}(I,D)$ is *false* because in this case $\mathbf{out}(I) \cap \mathbf{live}([U;I] \vdash \overline{D}) = \emptyset$, i.e. I doesn't write any variable in $\mathbf{live}([U;I] \vdash \overline{D})$, and thus $\mathbf{live}([U;I] \vdash \overline{D}) \otimes \mathbf{Dep}(I) = \mathbf{live}([U;I] \vdash \overline{D})$.
- Another slice for variables that are necessary due to I'. These variables can be found by a simple analysis of I, not requiring I'. From Lemma 1, the variables necessary for I', which belong to $\mathbf{use}(\overleftarrow{I})$, are left unmodified by $[U;I] \vdash \overline{D}$. The slice is thus:

$$[\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I';\overleftarrow{U}));I;] \text{ if } \textit{adj-live}(I,D)$$
$$[\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I';\overleftarrow{U}));] \text{ if } \textit{adj-live}(I,D)$$
$$I'$$

  which is equivalent to I', whatever U, D, and $\textit{adj-live}(I,D)$. Its necessary variables are $\mathbf{live}(I')$.

We end up with the following formula, which does not use the context U:

$$\mathbf{live}(\overline{I;D}) = \mathbf{live}(I') \cup (\mathbf{live}(\overline{D}) \otimes \mathbf{Dep}(I)). \tag{1.10}$$

A priori, there was a risk of circularity in this specification, since it used the *adj-live* property in many places. However (1.10) turns out to be independent from U and $\textit{adj-live}(I,D)$, so there is no circularity. In practice it suffices to run Adjoint Liveness analysis that computes $\mathbf{live}(\overline{Z})$ before TBR analysis that computes $\mathbf{use}(\overleftarrow{U})$. Equation (1.10) extends to basic blocks instead of instructions: for any block B followed by a downstream code D

$$\mathbf{live}(\overline{B;D}) = \mathbf{live}(\overline{B}) \cup (\mathbf{live}(\overline{D}) \otimes \mathbf{Dep}(B)).$$

This backward data-flow equation is particularly efficient since $\mathbf{live}(\overline{B})$ and $\mathbf{Dep}(B)$ can be precomputed.

1.4.5 Derived Rules for Adjoint Write Analysis

Suppose P contains a checkpointed piece C, and thus is made of three parts $[U, C, D]$. Checkpointing modifies model (1.7), because the forward sweep runs C, and the backward sweep runs $\overline{C} = \overrightarrow{C}; \overleftarrow{C}$. This requires storing a *snapshot*, i.e. enough variables to restore the calling context of C. Storing $\mathbf{use}(C)$ is sufficient, but we can do better. We need to run $\overline{C}$ again, and not C, and we saw that $\mathbf{live}(\overline{C}) \subset \mathbf{use}(C)$. Moreover we need to restore a variable only if it was modified "in between," i.e. is in the **out** set of code sequence $C; \overline{D}$, and we shall use the fact that $\mathbf{out}(\overline{D}) \subset \mathbf{out}(D)$. Therefore, in the non-trivial case where predicate *adj-live*(C, D) is true, we define the snapshot $\mathbf{snp}(U, C, D)$ as $\mathbf{live}(\overline{C}) \cap (\mathbf{out}(C) \cup \mathbf{out}([U; C] \vdash \overline{D}))$, and the reverse AD model becomes:

$$\begin{aligned} U \vdash \overline{C; D} = \; & \mathrm{PUSH}(\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U})); \\ & \mathrm{PUSH}(\mathbf{snp}(U, C, D)); \\ & C; \\ & [U; C] \vdash \overline{D}; \\ & \mathrm{POP}(\mathbf{snp}(U, C, D)); \\ & [\,] \vdash \overline{C}; \\ & \mathrm{POP}(\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U})); \end{aligned} \tag{1.11}$$

Model (1.11) is not necessarily optimal. Other choices could perform better for some programs. For example, putting U instead of $[\,]$ as the context for the generation of $\overline{C}$ costs more PUSH/POP inside $\overline{C}$, and on the other hand makes it unnecessary to store $\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U})$ in (1.11). Exploration of these tradeoffs is an open problem. In any case, we must specify the Adjoint Write analysis, to compute $\mathbf{out}(U \vdash \overline{Z})$. If $Z = [\,]$, obviously $\mathbf{out}(U \vdash [\,]) = \emptyset$. If $Z = I; D$, we use model (1.7) and distinguish two cases according to *adj-live*(I, D). Using also definition (1.3) on PUSH/POP pairs, we get:

$$\mathbf{out}(U \vdash \overline{I; D}) = \begin{cases} (\mathbf{out}(I) \cup \mathbf{out}([U; I] \vdash \overline{D})) \setminus (\mathbf{kill}(I) \cap \mathbf{use}(I'; \overleftarrow{U})) & \\ \qquad\qquad\qquad\qquad\qquad\qquad \textit{if adj-live}(I, D) & \\ \mathbf{out}([U; I] \vdash \overline{D}) \qquad\qquad\qquad\qquad \textit{otherwise.} & \end{cases} \tag{1.12}$$

As anticipated, we see that $\mathbf{out}(U \vdash \overline{I; D})$ is always included in $\mathbf{out}(I; D)$, and often strictly thanks to the PUSH/POP pairs. Again, (1.12) easily runs backward on a flow graph.

1.5 Application

Consider the example procedure `FLW2D1COL` (fig. 1.2) from a Navier-Stokes flow solver, shortened for readability preserving its structure. On a large mesh, this typical gather-scatter loop accounts for many computations, and thus

```
      subroutine FLW2D1COL(nsg1,nsg2,nubo,t3,pres,vnocl,
     +        g3,g4,rh3,rh4,ns,nseg,sq)
      < omitted declarations >
      do 30 iseg=nsg1,nsg2
         is1 = nubo(1,iseg)
         is2 = nubo(2,iseg)
         qsor = t3(is1)*vnocl(2,iseg)
         qs = t3(is2)*vnocl(2,iseg)
         dplim = qsor*g4(is1)+qs*g4(is2)
         rh4(is1) = rh4(is1) + dplim
         rh4(is2) = rh4(is2) - dplim
         pm = pres(is1)+pres(is2)
         dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
         rh3(is1) = rh3(is1) + dplim
         rh3(is2) = rh3(is2) - dplim
         call CK(pm,sq)
30    continue
      end
```

Fig. 1.2. *An example gather-scatter loop from a real code*

many derivatives. We differentiate `FLW2D1COL` in the reverse mode with TAPENADE, using the SA strategy. The call to `CK` is checkpointed. Figure 1.3 shows the resulting subroutine $\overline{\texttt{FLW2D1COL}}$. Differentiated variables are shown with a $\overline{\text{bar}}$ above. Since the loop's iterations are independent, the adjoining operation and the `do` loop operator commute (*cf* [9]), and therefore the resulting subroutine is a single loop, containing a forward sweep followed by a backward sweep. The benefits from the adjoint data-flow analyses are:

- From Adjoint Liveness analysis, variables `dplim`, `rh3`, and `rh4` are not necessary in the adjoint. Furthermore, the call to `CK` is the last instruction in its own checkpointed sub-part (i.e. its downstream sequel is []). Therefore this call can be removed as well as its associated PUSH/POP. Removed statements are here left as comments in boxes.
- From TBR Analysis, variable `dplim` is not used in the backward sweep, and therefore is not saved before it is overwritten. Variable `qsor` is used in the backward sweep $\overleftarrow{\texttt{P}}$, but is not overwritten before this use occurs. This explains there are no PUSH/POP for these variables.
- From Adjoint Liveness and Adjoint Write analyses, $\textbf{live}(\overline{\texttt{FLW2D1COL}})$ is smaller than $\textbf{use}(\texttt{FLW2D1COL})$, and that $\textbf{out}(\overline{\texttt{FLW2D1COL}})$ is smaller than $\textbf{out}(\texttt{FLW2D1COL})$. Specifically, arrays `rh3` and `rh4` are excluded from the snapshot in the procedure that calls `FLW2D1COL`.

We measured the benefits of Adjoint Liveness and Adjoint Write analyses on five large applications that we use as validation tests. Activity and TBR analyses are already applied systematically in TAPENADE, so this experiment strictly shows the additional benefit coming from Adjoint Liveness and Adjoint Write. The results strongly depend on the actual application, but can be quite interesting as shown in table 1.1. The speedup ranges between 7%

```
      subroutine FLW2D1COL(nsg1,nsg2 nubo,t3,t3,pres,pres,vnocl,
     +        vnocl,g3, g3,g4,g4,rh3, rh3,rh4, rh4,ns,nseg,sq,sq)
      < omitted declarations >
      do iseg=nsg1,nsg2
         is1 = nubo(1,iseg)
         is2 = nubo(2,iseg)
         qsor = t3(is1)*vnocl(2,iseg)
         qs = t3(is2)*vnocl(2,iseg)
C        dplim = qsor*g4(is1) + qs*g4(is2)
C        rh4(is1) = rh4(is1) + dplim
C        rh4(is2) = rh4(is2) - dplim
         pm = pres(is1) + pres(is2)
C        dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
C        rh3(is1) = rh3(is1) + dplim
C        rh3(is2) = rh3(is2) - dplim
C        call PUSH(sq)
C        call PUSH(pm)
C        call CK(pm, sq)
      < forward sweep ends, backward sweep begins >
C        call POP(pm)
C        call POP(sq)
         call CK(pm, pm, sq, sq)
         dplim = rh3(is1) - rh3(is2)
         qsor = g3(is1)*dplim
         g3(is1) = g3(is1) + qsor*dplim
         qs = g3(is2)*dplim
         g3(is2) = g3(is2) + qs*dplim
         pm = pm + vnocl(2,iseg)*dplim
         vnocl(2,iseg) = vnocl(2,iseg) + pm*dplim
         pres(is1) = pres(is1) + pm
         pres(is2) = pres(is2) + pm
         dplim = rh4(is1) - rh4(is2)
         qsor = qsor + g4(is1)*dplim
         g4(is1) = g4(is1) + qsor*dplim
         qs = qs + g4(is2)*dplim
         g4(is2) = g4(is2) + qs*dplim
         t3(is2) = t3(is2) + vnocl(2,iseg)*qs
         vnocl(2,iseg) = vnocl(2,iseg)+t3(is2)*qs+t3(is1)*qsor
         t3(is1) = t3(is1) + vnocl(2,iseg)*qsor
      enddo
      end
```

Fig. 1.3. *The adjoint of subroutine* FLW2D1COL *from fig. 1.2*

and 18%, and the improvement in memory between 0% and 49%. The STICS code is so large that its adjoint makes a heavy use of the swap. This explains the huge slowdown of the reverse mode, and makes it even more important to spare 49% in memory thanks to the Adjoint Write analysis.

Table 1.1. Time and memory improvements on five large validation codes. We compare run times and maximum stack size for original program, AD adjoint (using activity and TBR), and AD adjoint using adjoint liveness and adjoint write analyses.

Code name:	ALYA	UNS2D	THYC	LIDAR	STICS
Application domain:	*CFD*	*CFD*	*Thermo*	*Optics*	*Agronomy*
Original program runtime (s):	0.85	2.39	2.67	11.22	1.80
Adjoint program runtime (s):	5.65	29.70	11.91	23.17	42.60
... after adjoint data-flow analysis:	4.62	24.78	10.99	22.99	35.70
Improvement:	18%	16%	8%	7%	16%
Adjoint program memory use (Mb):	10.9	260	3614	16.5	456
... after adjoint data-flow analysis:	9.4	259	3334	16.5	230
Improvement:	14%	0%	8%	0%	49%

1.6 Conclusion

We have described the adjoint data-flow analyses that help Automatic Differentiation tools improve the performances of adjoints produced by the reverse mode.

These analyses rely on the special structure of adjoint programs. However, these are data-flow analyses and therefore can be described with the classical set-based notations used in compiler theory. To take full advantage of the knowledge of the reverse AD model, we view these analyses as specific data-flow analyses on the original source, rather than as generic data-flow analyses on the adjoint source. We obtain the data-flow equations of adjoint analyses on the original source, by formal specialization of the standard data-flow equations with respect to the reverse AD model. In addition, we obtain a global view that clarifies the relationship between adjoint data-flow analyses, and formal proofs of fundamental properties of our reverse AD model. We advocate this sort of transposition of techniques that originate from compilation into AD technology.

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

Our AD tool TAPENADE progressively implements the analyses we described here, and our first experiments show that this is definitely worthwhile.

References

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

2. Alan Carle and Mike Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
3. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.
4. C. Faure and U. Naumann. Minimizing the tape size. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 34, pages 293–298. Springer, New York, NY, 2001.
5. R. Giering and T. Kaminski. Generating recomputations in reverse mode AD. In George Corliss, Andreas Griewank, Christèle Faure, Laurent Hascoet, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, chapter 33, pages 283–291. Springer Verlag, Heidelberg, 2002.
6. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
7. L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint ANL-MCS/P936-0202, Argonne National Laboratory, 2002. also *Rapport de recherche number 4856*, INRIA.
8. L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.
9. Laurent Hascoët, Stefka Fidanova, and Christophe Held. Adjoining independent computations. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 35, pages 299–304. Springer, New York, NY, 2001.
10. U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1039–1048, Berlin, 2002. Springer.