

The adjoint Data-Flow analyses: formalization, properties, and applications

Laurent Hascoët, Mauricio Araya-Polo
INRIA Sophia-Antipolis, TROPICS team

1 Introduction

Classically, tools that perform code optimization require information on which variables are used by a given piece of code, which are overwritten, and which are killed i.e. completely overwritten. This is 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, some entirely new and specific analyses have been defined by several research groups, in particular *Adjoint Liveness* analysis, *To Be Recorded (TBR)* analysis, and *Adjoint Write* analysis. These three analyses appear tightly related.

Adjoint code has a particular structure, defined by the model of reverse AD. We are going to use this structure of adjoint programs to define the AD-specific data-flow analyses, then we will derive formally their rules from those of classical data-flow analyses. In this process, we will take advantage of characteristics of adjoint codes such as the control flow symmetry between forward and backward sweeps or the strategy that restores intermediate variables. Any general-purpose data-flow analyzer will not be able to take advantage of these complex behaviors a posteriori on the adjoint program. Therefore, we shall define AD-specific data-flow analyses that will run on the original code, incorporating knowledge on how the adjoint code will derive from it. We will obtain a formal, uniform specification of the three analyses above. We will demonstrate data-flow properties of adjoint codes and highlight the relationship between the three analyses. This specification also gives a firm foundation for implementation inside AD tools.

2 Context

We focus on *adjoint programs*, produced by the so-called *reverse mode* of Automatic Differentiation (AD). We just need to recall that adjoint programs compute derivatives in an order which is the inverse of the original program's execution order. Since the derivatives use the values of intermediate variables from the original program, some strategy must be applied to restore these values when they are needed. The classical alternatives are the *Recompute-All* (RA) and the *Store-All* (SA) strategies. In this paper, we restrict to the SA strategy, although we believe that the following specifications and demonstrations can be adapted to the RA strategy with minor adaptations. For a given original

program P , all the derivative instructions form the so-called *backward sweep* \overleftarrow{P} . The intermediate variables of P are computed during a so-called *forward sweep* \overrightarrow{P} that also stores them on a stack. These variables will be retrieved from the stack during the *backward sweep*. The adjoint program \overline{P} is made of \overrightarrow{P} immediately followed by \overleftarrow{P} . We also consider the *checkpointing* strategy, that trades duplicate execution of some part of P in \overline{P} for a smaller maximum size of the stack.

From the general theory of data-flow analysis, we single out the classical rules of the following analyses, that we will formally specialize for adjoint programs to get the rules for the three adjoint data-flow analyses. Consider any piece of program X :

- $\mathbf{W}(X)$ is the set of all variables at least partly overwritten during execution of X . For two successive pieces of program A and B , we know we have $\mathbf{W}(A;B) = \mathbf{W}(A) \cup \mathbf{W}(B)$. However, for any matching PUSH/POP on a stack, we have: $\mathbf{W}(\text{PUSH}(v); A; \text{POP}(v)) = \mathbf{W}(A) \setminus \{v\}$.
- $\mathbf{K}(X)$ is the set of all killed variables, i.e. completely overwritten by X . For two successive pieces of program A and B , we take the standard conservative under-approximation $\mathbf{K}(A;B) = \mathbf{K}(A) \cup \mathbf{K}(B)$.
- $\mathbf{R}(X)$ is the set of all variables read by X . For two successive pieces of program A and B , the variables killed by A hide the variables read by B , so that $\mathbf{R}(A;B) = \mathbf{R}(A) \cup (\mathbf{R}(B) \setminus \mathbf{K}(A))$
- $\mathbf{N}(X)$ is the set of all “live” variables, i.e. variables necessary to produce the output of X . For two successive pieces of program A and B , it is easy to show that there exists dependence relation $\text{Dep}(A)$ between the inputs and the outputs of A , and a combination operator \otimes such that $\mathbf{N}(A;B) = \mathbf{N}(B) \otimes \text{Dep}(A)$.

3 Adjoint Data-Flow analyses

In this central section, we define the three adjoint data-flow analyses and we formally derive their operational rules by specialization on adjoint codes of the standard data-flow analyses. The plan is as follows. Initially, we give a complete model of adjoint programs, that takes into account the *Adjoint Liveness*, *To Be Recorded*, and *Adjoint Write* analyses, in order to produce an improved adjoint code. Then we formalize these analyses using this specification of adjoint programs, starting with TBR analysis. Notice that this a priori introduces a circularity into the definition. After proving an important lemma about the variables left unmodified by an adjoint program, we are able to derive specific rules for *Adjoint Liveness* analysis. We can then show that the definitions circularity mentioned above disappears if *Adjoint Liveness* is run first, followed by *To Be Recorded* and then by *Adjoint Write*. We then derive the rule for the *Adjoint Write* analysis, in the context of checkpointing.

The complete model for adjoint programs is:

$$\begin{aligned}
U \vdash \overline{I; \overline{D}} &= [\text{PUSH}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overline{U})); I;] \text{ if } \text{adj-live}(I, D) \\
&[U; I] \vdash \overline{D}; \\
&[\text{POP}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overline{U}));] \text{ if } \text{adj-live}(I, D) \\
&I'
\end{aligned} \tag{1}$$

This model defines the adjoint code of a piece of \mathbf{P} that starts with an instruction I , followed by a downstream sequel D that goes to the end of \mathbf{P} , in the context of the previous upstream instructions U . This definition recursively uses the adjoint of D in the context $[U; I]$. On this model, we see that the PUSH/POP is required before instruction I only for variables that are both overwritten by I and necessary for the reverse sweep $\mathbf{R}(I'; \overline{U})$ of instructions I and upstream. The model thus requires an analysis that returns $\mathbf{R}(\overline{U})$. This analysis is the center of the TBR analysis. We also see that instruction I , along with its PUSH and POP, may be dead code in the case where the output of I is not necessary for \overline{D} . We define predicate $\text{adj-live}(I, D) = (\mathbf{W}(I) \cap \mathbf{N}(\overline{D}) \neq \emptyset)$, which embodies the effect of Adjoint Liveness analysis on the model. It is worth noticing that the output of the adjoint $\overline{\mathbf{P}}$ does not include the output of \mathbf{P} . Only the derivatives are of interest, and besides the original output of \mathbf{P} is often destroyed by the restoring mechanism. Therefore the necessary variables of an empty downstream sequel $D = []$ is $\mathbf{N}(U \vdash []) = \emptyset$. So the last I in \mathbf{P} is dead code in $\overline{\mathbf{P}}$, and sometimes several instructions before the last I are recursively dead too.

After specializing the rules for the \mathbf{R} analysis to obtain the rules for the TBR analysis $\mathbf{R}(\overline{X})$, we prove an important lemma that states that the PUSH/POP mechanism inside \overline{D} indeed works well, i.e. it leaves unchanged all variables used in $\overline{U}; \overline{I}$. This is proved by induction on the length of D , examining separately both possibilities for adj-live and the cases where a variable is a member of $\mathbf{W}(I)$ or not. Using this lemma, we are able to derive specialized rules that compute $\mathbf{N}(\overline{D})$, recursively on the tail of D . We observe that the result is independent from the context U , and we obtain the simple recursive rules:

$$\begin{cases} \mathbf{N}(\overline{[]}) &= \emptyset \\ \mathbf{N}(\overline{I; \overline{D}}) &= \mathbf{N}(\overline{I}) \cup (\mathbf{N}(\overline{D}) \otimes \text{Dep}(I)) \end{cases} \tag{2}$$

Intuitively, this states that a variable is necessary either because it is used in the differentiated instruction of I , or because it is not modified in I and necessary for \overline{D} , or because it is used to compute an output of I which is necessary for \overline{D} .

Finally, in the context of checkpointing a part C of \mathbf{P} , e.g. a procedure call, followed by a downstream sequel D , we show that the set $\mathbf{W}(\overline{D})$ is also required, therefore introducing the Adjoint Write analysis. We derive the specialized rules for this analysis, and underline that $\mathbf{W}(\overline{D})$ is often a smaller set than $\mathbf{W}(D)$, thanks to the PUSH/POP mechanism. Since the set of variables that must be stored to run C twice (the “*snapshot*”) is defined as $\mathbf{N}(\overline{C}) \cap (\mathbf{W}(C) \cup \mathbf{W}(\overline{D}))$, this allows checkpointing to take smaller snapshots before the first execution of C , therefore saving memory space in the adjoint program.

4 Application and Performance measurements

We show the use of adjoint Data-Flow analyses on a procedure taken from an industrial Navier-Stokes flow solver. Three effects are particularly visible:

- Adjoint Liveness analysis allows reverse AD to remove several instructions at the end of the forward sweep, as well as their PUSH and POP calls.
- TBR analysis allows reverse AD to remove yet other PUSH and POP calls.
- Since the procedure is checkpointed, Adjoint Write and Adjoint Liveness analyses yield smaller snapshots for this procedure, saving crucial memory space.

We give measurements of improvements in terms of execution time and memory space on some example applications. The execution time can be reduced by up to 10%, but the most notable gain is on memory space, where snapshots are sometimes reduced by 50%.

5 Conclusion

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, like the present one, of analyses for adjoint programs is necessary to define safely and to compare these analyses yet to come. Adjoint Data-Flow analyses are tightly linked, which makes their study difficult. In particular, we point out a tradeoff between TBR analysis and snapshots which should be studied, and could bring additional improvements.

This formal study is also a firm foundation for implementation. In particular the AD tool TAPENADE progressively implements the analyses we described here. We believe more can be gained with this adaptation of techniques that originate from compilation or parallelization into AD technology.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
- [3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann (Editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization*. LNCSE. Springer, 2001. selected papers from the AD2000 conference, Nice, France.

- [4] C. Faure and U. Naumann. The taping problem in automatic differentiation. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 293–298, 2001.
- [5] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. [www <http://www.autodiff.com/tamc>].
- [6] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.
- [7] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 299–304, 2001.
- [8] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint ANL-MCS/P936-0202, Argonne National Laboratory, 2002. also *Rapport de recherche number 4856*, INRIA.
- [9] INRIA Tropics team. On-line documentation of the Tapenade AD tool. Technical report. [www <http://www.inria.fr/tropics>].
- [10] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In *Proceedings of the ICCS 2000 Conference on Computational Science, Part II*, LNCS. Springer, 2002.