Data-Flow reversal and Garbage Collection

Laurent Hascoet

Data-Flow reversal and Garbage Collection

Laurent Hascoët

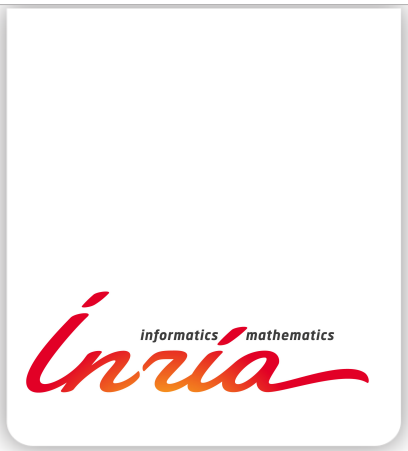
RESEARCH REPORT

N° 9416

Juillet 2021

Project-Team Ecuador

ISSN 0249-6399 ISRN INRIA/RR--9416--FR+ENG



Data-Flow reversal and Garbage Collection

Laurent Hascoët*

Project-Team Ecuador

Research Report n° 9416 — Juillet 2021 — 18 pages

Abstract: Data-Flow reversal is at the heart of Source-Transformation reverse Algorithmic Differentiation (Reverse ST-AD), arguably the most efficient way to obtain gradients of numerical models. However, when the model implementation language uses Garbage Collection (GC), for instance in Java or Python, the notion of address that is needed for Data-Flow reversal disappears. Moreover, GC is asynchronous and does not appear explicitly in the source. We present an extension to the model of Reverse ST-AD suitable for a language with GC. We validate this approach on a Java implementation of a simple Navier-Stokes solver. We compare performance with existing AD tools ADOL-C and Tapenade on an equivalent implementation in C and Fortran.

Key-words: Algorithmic Differentiation, Automatic Differentiation, Data-Flow Reversal, Dynamic Memory, Garbage Collection

* Projet Ecuador, INRIA Sophia-Antipolis

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Renversement du flot de données et Garbage Collection

Résumé : Le renversement du flot de données est au centre du mode inverse de la Différentiation Algorithmique, qui est la manière la plus efficace d'obtenir le gradient d'un code numérique. Pourtant, quand le langage d'implémentation de ce code utilise le GC (Garbage Collection), par exemple en Java ou en Python, la notion d'adresse disparaît alors que cette notion est nécessaire aux techniques classiques de renversement du flot de données. De plus, le GC est asynchrone et ne correspond pas à une instruction explicite dans le code. Nous présentons une extension du mode inverse de la Différentiation Algorithmique qui le rend compatible avec le GC. Nous validons ce modèle étendu sur un solveur Navier-Stokes simple implémenté en Java. Nous comparons les performances avec celles d'outils de DA existants tels que ADOL-C et Tapenade, appliqués à des implémentations équivalentes du même solveur réécrit en C et Fortran.

Mots-clés : Différentiation Algorithmique, Différentiation Automatique, Mémoire Dynamique, Garbage Collection

1 Intoduction

Algorithmic Differentiation (AD), also known as Automatic Differentiation or Differentiable Programming, and the related method of backpropagation, have received growing interest for their many uses in optimization, uncertainty quantification, and machine learning. Given a program that implements some mathematical function, AD creates a new program that computes derivatives of that function. While many approaches and tools have been developed to this end, we focus here on Reverse AD by Source-Transformation (ST-AD) which, compared to other approaches, often results in the higher efficiency to obtain gradients. ST-AD transforms the source code that computes the function into a new source code in the same language, that computes the derivatives.

While Scientific Computing applications have mostly applied AD to languages such as Fortran or C, newer usages such as Machine Learning apply AD to more recent languages, e.g. Python, that feature Garbage Collection (GC). However, Reverse AD in presence of GC raises issues that have not been studied yet. Consequently, AD tools popular in Machine Learning either use more costly Overloading-Based AD (OO-AD), or apply to smaller or restricted code fragments where GC does not occur, thus limiting applicability of AD.

In this work, we extend the model of Reverse ST-AD to deal with GC. More precisely, we discuss how we can perform the central task of Reverse AD, known as Data-Flow reversal, in the case where the data that needs to be restored has vanished due to GC. We prove that this extension does not alter the stack-like behavior on which Data-Flow reversal is based. We validate the extended model on a representative CFD application written in JAVA, therefore with GC. We compare performance with alternative models i.e. ST-AD without GC and OO-AD which is independent of GC.

To our knowledge, our paper makes the following novel contributions:

- We present the first Source-Transformation Reverse AD model for codes where Dynamic Memory management is based on GC.
- We propose a Data-Flow reversal scheme when storing and restoring the contents of a variable through their address is not permitted.
- We propose and motivate implementation principles for associating derivative Objects and variables with their primal counterpart, when the application language is Object-Oriented.
- We provide performance measurements and comparisons with ST-AD models as well as OO-AD.

Our work has the following limitations:

- There is at present no implementation of our model inside an automated AD tool. Implementation by hand is required, although it can be inspired by the output of ST-AD tools on C-like rewritten code pieces. True implementation would be based on an ST-AD platform for (a large subset of) e.g. Java or Python, which still requires substantial development and research.
- Further experimental measurements are necessary to refine assessment of our model's overhead.
- This work does not consider checkpointing, a storage/recomputation trade-off needed for Reverse AD of large codes. This is left for further study.

- Our model requires the application language to provide a `finalize` callback upon GC. Moreover, problematic interactions may occur if the application code already uses `finalize` in a sophisticated way.

Our paper is structured as follows. In Section 2 we summarize the AD concepts relevant to this work, as well as basic notions of Dynamic Memory management and Garbage Collection. We explain our approach in Section 3, and our implementation approach for testing in Section 4. Test results are presented in Section 5. We discuss related work in Section 6 and review our results in Section 7.

2 Background

We present the fundamental concepts needed for this research. First we present Source-Transformation Reverse AD in the framework of AD in general, and introduce the notion of Data-Flow reversal. Then we present the interaction of Data-Flow reversal with Dynamic Memory models, and summarize the methods that may cope with those. Finally we focus on the challenge posed by Garbage Collection.

2.1 Reverse Source-Transformation AD and Data-Flow reversal

Algorithmic Differentiation computes derivatives of a differentiable function, when this function is provided in the form of a computer program. AD applies the chain rule of calculus to the elementary operations that the given program executes. Readers interested in the models of AD may refer to the monography [5], and to [8] about the AD tools implementation aspects. The series of AD conferences AD1996, AD2000 ... AD2016 [3] is also a comprehensive source on the advancement of AD research and applications. The increased interest for Machine Learning (ML) has added a new application domain for Reverse AD [1], known in this case as backpropagation. Training a Deep Learning model requires to compute repeatedly the gradient of the model output score with respect to the model weight matrices. Reverse propagation of derivatives is the only efficient way to do so. This has given birth to new research on AD, in particular in the context of functional languages and lambda calculus, and to a blooming of new AD tools for the languages and libraries popular for ML, in particular Python.

We will here restrict to a concise description of the AD concepts that we use in the sequel. Consider a program with floating-point inputs and outputs, implementing a differentiable function $F : \mathbb{R}^n \to \mathbb{R}^m$. AD sees every particular execution of the program as a sequence of elementary operations, each of them of the form *read-execute-write*. In other words, one execution is a (possibly long) list of three-address code. This sequence is mathematically equivalent to an ordered composition of functions, one per operation. Therefore the chain rule of calculus gives the first-order derivative of the full F as the product of the derivatives of each elementary operation. Each elementary derivative, as well as their final product, are (Jacobian) matrices. The coefficients of each elementary Jacobian are easily computed from the values of the program variables at the time of execution of the corresponding elementary operation.

Associativity of matrix multiplication lets one compute these products in any order. However storage constraints do not allow one to store all elementary Jacobians at the same time. Although other combinations can be studied, this leads to two classical strategies to accumulate the elementary Jacobians into the final full Jacobian.

- **Tangent AD:** progressively accumulate from the first elementary operation on, i.e. in the same order as the original code execution. At any step, only one elementary Jacobian need

be prepared and accumulated. One can then dispose of this elementary Jacobian before preparing the next one. Accumulation is done by multiplying the elementary Jacobian to the left of the accumulated Jacobian. The accumulated Jacobian itself has size $p \times n$ where p is the *window size* i.e. the number of variables that convey the data from the program *before* this operation to the program *after* it. From the size $p \times n$ of the accumulated Jacobian one can see that the cost of the accumulation, in time and in memory, grows with the number of inputs n, and almost not with m as $p = m$ only at the end of propagation.

- **Reverse AD:** progressively accumulate from the last elementary operation on, i.e. in the reverse of the original code execution. Like in Tangent AD, only one elementary Jacobian must be stored, in addition to the accumulated Jacobian. Accumulation is done by multiplying the elementary Jacobian to the right of the accumulated Jacobian. The accumulated Jacobian this time has size $m \times p$. From the size $m \times p$ of the accumulated Jacobian one can see that the cost of the accumulation, in time and in memory, grows with the number of outputs m, and almost not with n as $p = n$ only at the end of propagation.

In a great many applications, what is needed is a gradient i.e. the derivative of one or a few cost scalars with respect to a large number of design parameters. This is classically the case in optimum design or in inverse problems where the cost scalar is some distance from the current state to a prescribed state, whereas the parameters are fields of geometric or geometry-based values. It is also the case in ML where the cost is the error in the final estimation, whereas the parameters are the coefficients used at each convolution step. Therefore m is of the order of 1, and n of the order of millions. Tangent AD is irrealistic and Reverse AD is the only way to go. This paper is exclusively concerned with Reverse AD.

There is an additional difficulty specific to Reverse AD, though. The coefficients of elementary Jacobians at a given instruction require in general the values of the program variables when this instruction is executed. On the other hand, these elementary Jacobians are required in the reverse of the program execution order. We therefore need to provide the values of the program variables in reverse order. As programs most often overwrite their variables or let them fall out of scope, we need to restore or retrieve the successive values of variables backwards in time. This leads us to a scheme with a **forward sweep** that computes and stores these values, followed by a **backward sweep** that pops these values and uses them to compute the derivatives backwards. This is called **Data-Flow reversal**, and this paper concentrates on this issue.

One radical way of achieving Data-Flow reversal is to store the execution trace of the original code, complete with the operations performed and the values they operated on. The trace is then read in reverse and the gradient is computed along. The Data-Flow reversal issue is trivially solved. Traditionally, this is called **Overloading-Based AD** (OO-AD). The drawback of this popular AD approach is that it completely unrolls the program for this particular execution. The patterns in the trace that correspond to the body of loops are replicated as many time as loops iterate, yielding a possibly long trace. An alternative approach, called **Source-Transformation AD** (ST-AD), builds a new differentiated code as a source program akin to the original code, using variables, data structures, and control-flow structures derived from those of the original code. Memory consumption can be dramatically reduced as most of the control remains in the compact form of a code and, essentially, only the data is stored in the Data-Flow reversal trace. Derivatives are stored in a way that mimics the original variables and data structures, using the same access patterns. From now on, we will focus on ST-AD, but will come back to OO-AD in the conclusion.

There are finer variations in the ST-AD approaches. For example, the forward recomputation approach tries to avoid storage, by restarting computation from the initial state each time values one step backwards in time are needed. In principle, no Data-Flow reversal is needed anymore.

Obviously the cost in runtime grows quadratically with the original run time. This cost must be mitigated with so-called **checkpointing** strategies, yielding good performance. However, checkpointing strategies amount to store intermediate states in the original computation and to restore them backwards in time, so that the Data-Flow reversal problem is back. Another approach is to store into the trace the coefficients of the Jacobian matrices rather than the original values that they use. The backward sweep need not refer to the primal values anymore. However, it still refers to the derivatives with the same access patterns as in the original code.

2.2 Data-Flow reversal and Dynamic Memory

Dynamic memory allows a program to *allocate* i.e. borrow a chunk of free memory space from the *heap* and declare that it is now accessible through a variable. Conversely the program may later *deallocate* this memory i.e. return it to the heap of free memory. In the meantime, the program may use this memory in the same manner as that of standard variables that are declared in the subroutines preambles and whose memory lies in a different area, the *execution stack*. As allocation and deallocation can be done at any time, dynamic memory gives programmers a welcome flexibility compared to the unorthodox manipulations that were commonplace before. Most imperative languages now allow for Dynamic Memory management.

Dynamic Memory poses a specific challenge to the Data-Flow reversal of Reverse AD. Several answers have been proposed, that we will review shortly as they provide a useful basis for the specific case of GC. Reports on these possible answers are few [6], in part because they might be considered ancilliary compared to issues of general efficiency of gradient computation, and their application to e.g. a gradient descent algorithm in an optimization problem. Also many applications with a long development history show a clear separation between initialization, computation, and post-processing phases, with memory allocation concentrated in the initialization, memory deallocation in or after the post-processing phase, whereas AD is required only on the computation phase. However newer codes show a closer interleaving of Dynamic Memory usage inside the computation phase, as this lets the code benefit better from Dynamic Memory. This is even more the case with Object-Oriented languages, where creating and deleting objects (implying allocating and deallocating memory) are made very easy.

Let us describe the challenge with the help of Figure 1. Consider a C code, on which we apply Reverse AD. The differentiated code consists of a forward sweep, shown on the left, followed by a backward sweep, shown on the right. We show each instruction of the forward sweep side-by-side with its corresponding derivative instruction(s) in the backward sweep. Therefore bear in mind that the right half of Figure 1 is executed from the bottom up. The possibly surprising form of the Reverse AD derivative instructions follows from the right-accumulation of elementary Jacobians. Readers can take it for granted or refer to [5]. We do not show the original C code, as it is exactly the forward sweep without the Data-Flow reversal `push` operations. Figure 1 focuses on the life time of a chunk of dynamic memory `M` and on a pointer `p` pointing inside `M` and using it in a typical way. In Source-Transformation AD, derivatives are stored in memory in a way that mimics the original variables: the reverse-derivative of variable `z` is stored in a new variable, named `zb`[1]. The derivative of the destination of pointer `p` is stored as the destination of a new pointer, named `pb`. It also follows that dynamic memory allocated and deallocated in the forward sweep must be symmetrically re-allocated and deallocated again in the backward sweep. Along with re-allocation of `M` in the backward sweep, there is also an allocation for its derivative counterpart `Mb`, not shown on figure.

Looking closer at the central derivative instruction, we see that the partial derivative `cos(*p)` needs the same value of `*p` as was used in the primal `sin(*p)`. As `*p` may be overwritten or

[1] The appended `b` traditionally stands for "backwards" or "bar"

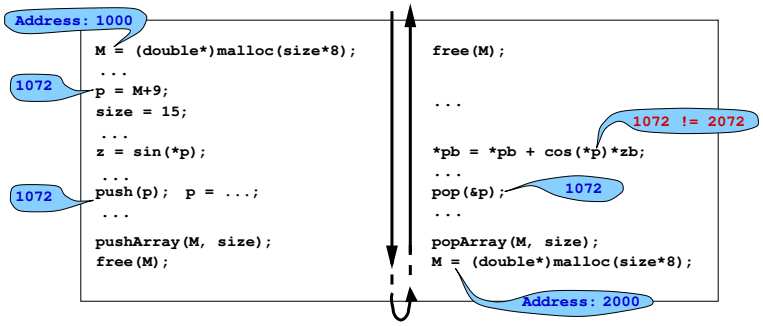


Figure 1: The challenge of Data-Flow reversal with dynamic memory. C syntax. The address stored by `push(p)` in the forward sweep (left) is based on the memory chunk allocated by the forward sweep. When it is retrieved by `pop(&p)` in the backward sweep (right), it makes no sense with respect to the memory chunk re-allocated by the backward sweep. To highlight correspondence between the two sweeps, the code on the right must be read upwards.

fall out of scope in the meantime, Data-Flow reversal must store it and restore it. This is the expected role of the `pushArray` placed immediately before memory chunk `M` is deallocated, and of the corresponding `popArray` when the backward sweep re-allocates a memory chunk. Likewise, as evaluating `*p` requires `p`, the address in `p` must also be stored and restored in case it is overwritten. This the expected role of the `push(p)` inserted before `p` is overwritten, and of the matching `pop(&p)`.

Unfortunately the code as sketched in Figure 1 will not work as expected, because the memory chunk returned by the backward sweep `malloc` has no reason to be the same as the forward sweep `malloc`. The retrieved address for `p` is therefore wrong. As a side note, The backward `malloc(size)` as well as the `pushArray` and `popArray` are also wrong because there is no guarantee that the value of `size` is preserved. In our example it was indeed modified, and preserving it with another `push`/`pop` pair would not work because the `push` would occur too early (before `size=15`) and the `pop` therefore too late.

This Data-Flow reversal issue in presence of dynamic memory is not limited to primal variables (`p, *p, z`...). The same applies to the derivative variables `pb, *pb, zb` and to the derivative memory chunk `Mb` that must be allocated in addition to `M` in the backward sweep. For example we must ensure that `pb` effectively addresses the location of the reverse derivative of `*p`. For conciseness, our simple example does not illustrate these problems but they resort to the same answers. Let us review the typical answers that have been proposed or used on real applications:

- **Inhibit deallocates:** the primal code may deallocate `M` as it becomes useless. This is no longer the case in Reverse AD code, as it uses `M` again in the backward sweep. Therefore, why not remove the `free(M)` from the forward sweep, as well as the corresponding `malloc(size)` from the backward sweep? The drawback of this simple answer is that we just loose all the benefit of Dynamic Memory. The peak memory use, at the turn point between forward and backward sweeps, may be too high especially on large codes. One may argue that our Reverse AD example code of Figure 1 already uses memory space to store the contents of `M`, and the present answer makes this storage unnecessary. However, Reverse AD has developed a number of sophisticated strategies to reduce or avoid many of these storage actions, or possibly to direct them to more distant memory.

- **Recompute addresses:** in some situations, the backward sweep can recompute the

needed value of the address in `p`. In our example, one can replace the `pop(&p)` with a copy of the original computation `p = M+9`. However, this requires that all ingredients of this computation are themselves restored to their original value, which may prove very complex in the framework of a *backward* sweep.

- **Rewrite allocation manager:** an elegant answer can be to take control over the memory allocation mechanism, so as to guarantee that the memory chunk returned by the `malloc(size)` in the backward sweep is exactly the same as by the forward sweep's `malloc(size)`. Taking advantage of the symmetry between the forward and backward sweeps of Reverse AD, one can show that this is possible. At the same time, this retains the main advantage of dynamic memory allocation which is that the sequel of the forward sweep can reallocate and reuse the same memory chunk as `M`. Implementation when possible is system-dependent.
- **Rebase addresses:** the forward sweep `free(M)` can record the address of `M`, so that the backward sweep `malloc(size)` can deduce the offset to apply to `p` to make it correct again. The difficulty is to detect that `p` is effectively referring to this memory chunk of `M`. Under a few restrictive conditions, this can be achieved by testing that the stored address for `p` belongs to the memory chunk returned by the forward allocation of `M`. However this answer is not general as a code may compute addresses that are outside of any allocated chunk, as long as it does not dereference them. Also, our experience shows that the cost of searching for an address in a list of memory chunks is expensive and significantly slows down the differentiated code.
- **Track chunk identifiers:** one can overcome the limitations of the **Rebase addresses** answer by assigning a unique identifier to each memory chunk allocation, and by attaching this identifier to each address (e.g. `M` or `p`) that is based on this allocated chunk. This requires an additional memory space to keep the reference identifier of each pointer, which is an overhead we consider acceptable. Every pointer assignment must assign the reference identifier of its right hand side into the reference identifier of the address in its left hand side. This lifts the limitations of the **Rebase addresses** answer, and avoids the expensive phase of searching for an address in a list of memory chunks.

2.3 Garbage Collection

One issue introduced by Dynamic Memory is the task of deallocating memory at the right time. Doing so too late (or not at all) leads to failures by exhaustion of the available memory. Doing so too early leads to forbidden access to unallocated memory and to bugs that are very hard to track. One elegant answer is Garbage Collection (GC): memory allocation remains, but the application is not anymore in charge of deallocating explicitly. Schematically, for each chunk of allocated memory, the runtime system instead counts the number of locations in memory that refer to this chunk. When this number falls to zero, implying that the chunk cannot possibly be accessed anymore, the system takes care of deallocating the "garbage" chunk. This is done asynchronously by the runtime system and the application source need not contain any mention of GC. The overhead execution cost of GC is significant, although this technology has made impressive progress. Consequently languages with GC have not been very popular for computation-intensive activities such as Scientific Computing. This is changing with the recent rise of ML which, although requiring intensive computation, elects as its standard programming language Python, a language with GC. As ML is also a major user of Reverse AD (backpropagation) the question of Reverse AD on a language with GC is gaining importance.

Garbage Collection restricts the operations allowed on Dynamic Memory:

- It is no longer possible to use memory addresses, either to store or to restore them. The very notion of addresses disappears. One reason for that is: if the application could take an address and store it for later use, the reference counter of the GC would not know that and might reclaim the associated memory before the stored address is used. Consequently, the well-known *pointer arithmetic* of C (e.g. `p = M+9`) is forbidden. Likewise, pushing and popping the address in pointer `p` must be replaced by a more elaborate marking and re-assignment of the object in variable `p`.
- The action of deallocating memory occurs asynchronously. The GC takes place at some unpredictable time, unrelated to any particular location in the source code. This is adverse to the usual architecture of a Reverse AD code, which exhibits a control-flow symmetry between its forward and backward sweeps, thus allowing an orderly reverse restoration of the Data structures. In other words, if the GC does not happen at a known location in the forward sweep, where should we write the code for "reverse GC" in the backward code?

Consequently, many of the answers of section 2.2 do not apply anymore. The `push` and `pop` operations could store their argument on a stack, under the hood actually storing a reference to the pushed object. This is indeed an adequate implementation of the **Inhibit deallocates** answer, as this extra reference will prevent GC to occur. Unfortunately the associated drawback remains, which is an unreasonably large memory consumption. **Recomputing addresses** is in a sense simpler, as pointer arithmetic is no longer permitted, but still bears the same difficulties as in the non-GC case: recomputing means inserting into a backward sweep a selected slice of the forward sweep that resets the needed objects. This incurs in general a quadratic increase of the code runtime. **Rewriting the allocation manager** is not possible anymore, as the deallocation part is managed by the GC and is therefore out of reach. **Rebasing addresses** is no longer possible, as the notion of address is not available.

It seems the only option compatible with GC is to assign and **Track chunk identifiers**, the unique identifiers acting as pointers. Moreover, languages with GC generally provide a callback method (typically called `finalize`) that is called at the time of GC, and this can be used to define a symmetric time for a "reverse GC" inside the backward sweep.

3 Data-Flow reversal for a language with GC

We propose a strategy for Data-Flow reversal in the presence of GC. For convenience, we will often use basic Object-Oriented vocabulary to anticipate on the implementation and experiments of the next sections, which use Java. However Objects are not necessary to apply this strategy.

Our approach uses two main ingredients. The first ingredient is to assign a unique identifier to each newly created memory chunk or object. This will cope with the absence of explicit addresses. This can be done by introducing a global integer counter, and by extending each class or allocatable data structure with an integer field to hold this identifier. In an Object-Oriented context, this can be enforced rigorously by having all differentiated classes inherit from a special-purpose class or interface.

The second ingredient copes with the absence of an explicit calling location in the source for GC. We rely on the existence of a language-specific callback that is called by GC immediately before a variable is deallocated. This feature is now commonplace in languages with GC. For instance in Java, the method `finalize` is called immediately before any object is garbage collected. Our second ingredient is therefore to have each Object involved in the differentiated code portion redefine the `finalize` method to:

1. memorize the contents of the Object, through storage onto the AD stack

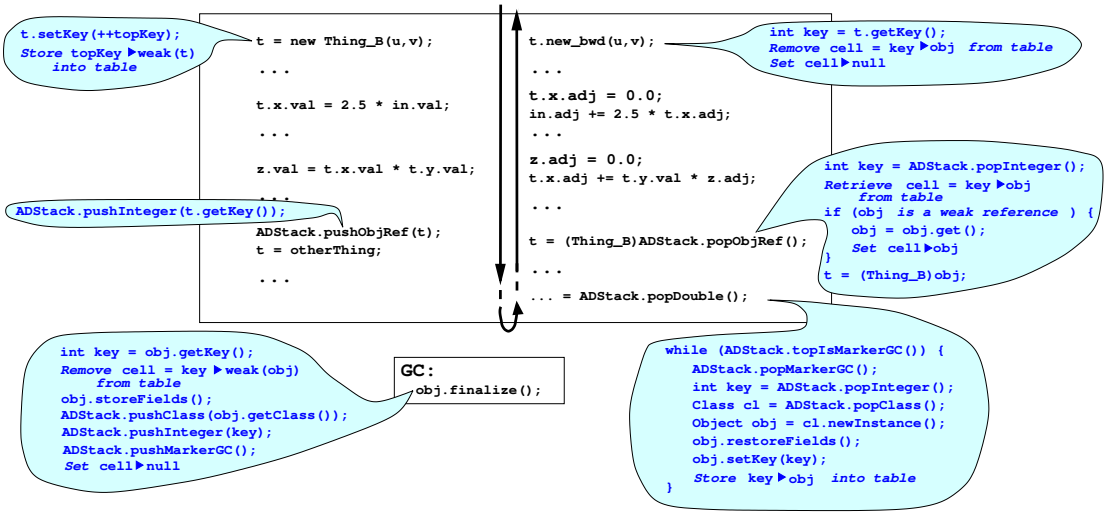


Figure 2: Data-Flow reversal when GC is involved. Java syntax. Forward sweep is on the left. Backward sweep on the right must be read upwards. When it occurs, GC is asynchronous and is not part of either sweep. Bubbles explicit the special behavior added to key operations of each sweep and of GC to guarantee correct reversal.

2. place a marker on this AD stack to mark that some Objects were deallocated, thus enforcing a correct interleaving of re-allocation with the other operations in the backward sweep.

We will use figure 2 to support the step-by-step description of our strategy. We will follow the order of the key moments in the lifetime of an object subject to GC, during the forward then backward sweeps of the Reverse AD code. Operations occurring in the forward sweep appear on the left half of figure 2, and read from top to bottom.

- First, the object (call it *Ot*) is allocated when it is created during the forward sweep. Conceptually, we distinguish *Ot* from e.g `t`, which holds a pointer to *Ot*. Also, our AD model extends the type of *Ot* (here to `Thing_B`) to make room for the derivatives of the fields of the primal Object (here of type `Thing`). Our strategy systematically adds bookkeeping operations: a new unique integer key is created, similar to the *pseudo-address* introduced by OO-AD approaches. This key is stored into the special-purpose `key` field of *Ot*. Moreover, the correspondence (`key`▶weak(`t`)) from `key` to *Ot* is added into a global special-purpose table. Most importantly, this correspondence must use a so-called *weak* reference i.e. a reference that is not counted in the GC reference count, so that its existence does not prevent GC from reclaiming *Ot* when no other reference to it exists. These operations are indeed encapsulated/hidden in the object creation method, but for clarity figure 2 explicits them in a "bubble".

- The new *Ot* is then used in the forward sweep in the same way it is used in the primal code, following the usual pattern initialize-read-overwrite-read-etc. As usual in a forward sweep, variables that are going to be overwritten or that fall out of scope, and whose present value will be needed in the backward sweep, must store their present value on the AD stack before being overwritten or going out of scope. Our strategy comes into play when the value to store is a reference to *Ot*, thus implying a call to the special-purpose primitive

`pushObjRef`. In figure 2, this is illustrated by the overwrite of `t`, which was previously holding a reference to *Ot*. As there are no addresses in the language, `pushObjRef` pushes instead the unique key attached to *Ot*.

- Later, if no other object in memory holds a reference to *Ot*, GC may wake up and reclaim the memory of *Ot*. As GC does not occur at a specific location in the code, figure 2 shows it in a separate box. Our strategy comes into play through redefinition of the `finalize` method on any `obj` of type `Thing_B`:
  - from the unique key in `obj`, retrieve and remove the corresponding entry `key`▶weak(`obj`) in the table.
  - push the contents of `obj` on the AD stack.
  - push on the AD stack, in order, the unique key, the class name, and a special "GC" marker.

  We describe this behavior on GC while discussing the forward sweep just for convenience, but notice that GC can also occur during the backward sweep, any time before restoration of `t`. As we will prove later, this does not prevent correct restoration of `t` before it is needed in the backward sweep. After GC, everything is in place for restoring `t` in the backward sweep. If GC does not occur, the bookkeeping table still holds the entry `key`▶weak(`obj`) so that the backward sweep can reuse it.

The forward sweep is followed by the backward sweep, shown on the right half of figure 2. Following our convention of figure 1, the backward sweep must be read from the bottom up, to highlight correspondence between operations in the forward sweep and their counterpart in the backward sweep. Still concentrating on object *Ot*, the relevant steps are:

- Each time the AD stack top changes, more precisely before each call to a `pop` operation, we check if the AD stack top is the special "GC" marker. This is a right time to undo the corresponding GC from the forward sweep. In a sense we mimic the asynchronous behavior of the GC by triggering this step through the marker on the AD stack, and not through an explicit procedure at some predefined location in the code of the backward sweep. If the marker is present, what happens is:
  - We successively pop the GC marker, then the class name (here `Thing_B`), then the unique key of the late *Ot*.
  - Informed by the class name, we build a new Object of this class and fill its contents by popping from the stack the contents we had stored before GC. We may just as well call this new object *Ot*, as the initial *Ot* has been destroyed by GC.
  - We set the unique key of the new *Ot* to the key we just retrieved
  - We add a new entry (`key`▶`obj`) into the bookkeeping table, associating the key to the new *Ot*.
- If *Ot* is needed in the backward sweep, we know there has been at least one call to `pushObjRef` in the forward sweep, and the corresponding `popObjRef` has been placed in regard in the backward sweep. By construction, there is a `popObjRef` before any use of *Ot* in the backward sweep. Our strategy comes into play at that stage, by actually popping the unique key that has been pushed, then retrieving the actual *Ot* itself by a lookup in the bookkeeping table, and actually returning (a pointer to) this *Ot*. It is worth noting that at this stage, regardless of GC having occurred or not, the correct *Ot* is associated with the

key. If GC has occurred, then the `popObjRef`, like any `pop`, will have seen the GC marker and placed correspondence to the new *Ot* in the table. If GC has not occurred, then the initial *Ot* is still in place in the table, and returning it through the `popObjRef` builds a real reference to it so that no GC will collect it anymore. In the sequel of the backward sweep, all references to *Ot* are restored, so that Data-Flow reversal works as expected. This is the case for the instructions involving `t` shown on the right of figure 2.

- Last, as the backward sweep reaches the location corresponding to the forward creation of *Ot*, our strategy places a call to a method `new_bwd` that removes the entry (`key`▶`obj`) from the bookkeeping table. The system is then free to reclaim the memory of *Ot* during a subsequent GC.

A few refinements are necessary, that are not shown on figure 2 for clarity. When an object A refers to an object B, it is in general not granted that GC collects A before B. One counter example is with circular references e.g. B also refers to A. Therefore in the backward sweep restoration of A may occur first. When restoration of A comes to restoring the field that should hold B, it has the key for B but B is not yet stored in the table. We implemented a waiting list for these unfinished fields such that when B is restored at last, all fields waiting for it are set.

Correct scheduling of this Data-Flow reversal requires some justification. In figure 2 operations that are attached to one specific location in the forward or backward sweeps are naturally ordered by the code in these sweeps, just like in standard Reverse ST-AD. The difficulty comes from the two sets of operations at the bottom of the figure, that may occur at arbitrary moments, if they occur at all. The main **property** that we must guarantee is that the table cell for the given `key`, which is sought by the `popObjRef` in the backward sweep, is certainly available at that moment. To this end, we will first show that:
lemma: The state of the AD stack just after a `push` in the forward sweep is identical to its state before the corresponding `pop` in the backward sweep.
This lemma is obvious in standard Reverse ST-AD, but not as much with the new asynchronous mechanism for GC. We will show it by induction on the structure of the Reverse AD code.

- as the base case, consider the last push of the forward sweep and its counterpart first pop of the backward sweep. If GC occurs between the two, then one or more Objects are stored on the stack, each of them topped by a GC marker, as shown on the bottom left of figure 2. No other operation touches the stack before we reach the first `pop`. Just before this `pop`, our strategy peeks for GC markers on top of the stack and repeatedly pops from the stack all data that was added by GC since the `push`. This process stops when no GC marker is left, which is when the stack is the same as after the last `push`.

- as the induction case, consider a `push`/`pop` pair, and the pair `npush`/`ppop` of the next push and the previous pop. If some GC occurs between `push` and `npush`, then the data for some Objects is added on the stack, topped by GC markers. Whatever happens between `npush` and `ppop`, we know by induction that the stack state is the same after `ppop`. Then again between `ppop` and `pop` some GC may occur, adding again some Objects data topped by GC markers. No other operations affect the stack. Again, before `pop` our strategy restores all the GC'ed objects until no GC marker is left, which is when the stack is the same as after `push`.

This demonstrates the **lemma**. Now to prove the **property**, consider any `pushObjRef(t)`/`popObjRef()` pair. If GC collects *Ot*, it must be after `pushObjRef(t)`, since before that there is at least one reference to *Ot*: `t` itself. It follows from the **lemma** that the stack before the `popObjRef()` is the same as after the `pushObjRef(t)`. Therefore the effect of this GC on the stack has been undone,

as the GC markers are gone. This undoing was done by the code shown on the bottom-right of figure 2. This code has stored a table cell associating `key` to the rebuilt *Ot*.

If on the other hand no GC has collected *Ot* then neither of the two sets of operations at the bottom of figure 2 was done, and the table still holds the association from `key` to a weak reference to the forward sweep *Ot*. The `popObjRef` can retrieve this reference, turn it into a strong reference to prevent GC, and return it. In other words when GC does not occur, the overhead of our strategy remains low as the same object is used in both sweeps. This proves the **property**.

4 Implementation

We have chosen to experiment our strategy for Data-Flow reversal with GC on Java. Another choice could be Python. However, Java is closer to C, for which we have an ST-AD tool ready for comparison, Tapenade. More importantly, it seems to us that the programming constructs offered by Java are more structured, more strict, less permissive than in Python. For the present study, permissiveness may raise extra issues that could divert us from our objective.

Our goal is to test our strategy and to measure performance on a non-trivial example. We also want to compare performance with more classical Reverse ST-AD model on C, i.e. without GC. Please note that we have not implemented an automatic ST-AD tool for Java. Before undertaking such a development, it is necessary to evaluate the performance of the proposed underlying AD model, and possibly adapt it (or even abandon it!). This is the purpose of this experiment. Therefore, our implementation is indeed a hand-written Reverse AD of a selected Java computational code, following our ST-AD model as systematically as an automated tool would do.

It may be of interest to discuss features that we must introduce into our ST-AD model in relation to Objects. They are only loosely related to GC, but might be interesting in general for ST-AD of Objects. In any case, they are useful to understand the source code that we provide for examination. The question is the mode of association of primal to derivative variables [4]. Classically **association-by-name** suggests to create a new variable e.g `vb` to hold the derivative corresponding to some original variable `v`. The alternative is **association-by-address**, where only the original variable name `v` is used, but its type is changed to a structure holding both primal and derivative values. These values would then reside at "nearby" memory addresses, which incidentaly improves locality. The two approaches extend to constructs of a larger granularity. For instance a structured data type in the original code may give birth in the Reverse AD code to just one data type that contains both primal and derivative objects, or to two separate data types, one for primals and one for derivatives. At an even coarser granularity, a package or a Fortran90 `MODULE` could give birth to one composite package containing e.g. all primal or derivative procedures, or to two separate packages.

As granularity becomes coarser, it becomes harder to split constructs. For example we recommend building just one composite `MODULE` containing both primal and derivative variables and code. Just one reason for that is the possible `private` variables of the `MODULE`, that must be visible to both primal and derivative code. The same applies to Objects in general, because they behave in part like packages. Therefore for each Object-type variable of the primal code, the ST-AD code will contain only one variable, with same name but with a composite, differentiated type.

On the other hand we will still use association-by-name for variables or fields of (differentiable) primitive type. A `double` variable named `v` gives birth to `v` and `vd` in Tangent AD and `v` and `vb` in Reverse AD, all of them of type `double`. This avoids introducing many new Objects in the

ST-AD code, which may have an overhead. For the special case of arrays of primitive type, which in Java are Objects, we choose to introduce a new specific type e.g. `DoubleArray_B` containing one array for primals and one array for derivatives. This introduces fewer new Objects than the association-by-address alternative i.e. just one array of pair-like Objects.

A last point, related to GC, is that since Java does not let us take addresses, it has no by-reference argument passing. This is unfortunate as the derivative of an "in" procedure argument generally gives birth in Reverse ST-AD to an "in-out" derivative argument. This is an issue only for arguments of primitive type, which we solve by introducing temporary containers, of a type we call `Todouble`.

5 Experiment

Our test application is an open-source 2D incompressible Navier-Stokes solver[2] written in Java by J. Scollay in 2018. The solver applies an advection-diffusion scheme on a square regular grid. We isolated the solver kernel, so that it takes as input an initial state consisting of velocities and density at each grid node, and computes a final state after a number of time steps. We observed that the solver chooses to manage the swaps between "previous" and "current" states by copy. A possibly more efficient alternative is to keep two variables that point at these states and to only swap the variables. As this second alternative implies more overwriting of Object-type variables, it would make an even stronger test of our approach to restore Object values. We modified the source to apply this alternative at places, thus enlarging the range of programming styles tested.

We use AD to obtain the gradient of a scalar cost function with respect to a number of control parameters. These cost scalar and control parameters are quite arbitrary, our only constraint being that they show a nontrivial dependency. As the cost, we choose the sum of squared velocities on a line of grid points (i ranging from 0 to 81, j=70), at the final simulation time. As the control parameters, we choose the two components of velocity at initial time on 82 grid points (i=30, j ranging from 0 to 81), plus velocity at one particular grid point (i=40, j=40) for each time step, thus pretending there is a actuator at that point. Again this is arbitrary, mainly to show that we compute the gradient of the cost with respect to any set of input or control parameters.

Since we want a gradient, i.e. the derivative of a scalar output with respect to a number of inputs (here 163), we apply Reverse AD to the solver. Usage of the resulting gradient, e.g. to solve an optimization problem or an inverse problem, is outside the scope of this study: provided the gradients are correct, which we show by classical methods, the way they are obtained does not influence the final optimization results. The only relevant question is the run-time and memory cost to obtain this gradient. For completeness, we also apply and test Tangent AD.

For comparison purpose, we wrote two equivalent solvers, one in Fortran and one in C, keeping the same implementation choices regarding algorithm, data structures, and use of addresses and pointers. We differentiate the Fortran and C versions using Tapenade `3.16`. We also differentiate the C version with `ADOL-C 2.7.2` [11], which is a leading OO-AD tool. All tests are run on the same platform, which is a workstation running Linux/Fedora FC33. No test involves parallel computing, only one of the underlying 4 cores is used. Compilers used are `OpenJDK 1.8.0_201` and `gcc`, `g++`, `gfortran`, all referring to `GCC 10.3.1-1` and called with optimization flag `-O3`. Although we also checked individual gradient components, we present here only the values of the *condensed Jacobian*

$$\texttt{output\_b}^T \times F' \times \texttt{input\_d}$$

[2] https://github.com/deltabrot/fluid-dynamics

	F90/Tapenade	**C/Tapenade**	**Java/hand-written**	**C(++)/ADOL-C**
Finite Diff.	1.8525807	1.8525807	1.852580*7021*	1.8525807
Tangent AD	1.85258050858638*13*	1.85258050858638*20*	1.85258050858638*13*	1.8525805085863*80*
Reverse AD	1.85258050858638*18*	1.85258050858638*22*	1.85258050858638*07*	1.852580508586*379*

Figure 3: Condensed Jacobian obtained for each AD model. Finite differences are approximate by definition. All values obtained by AD match, up to machine precision. Non-matching digits are shown in italics.

where `input_d` and `output_b` are two arbitrary vectors of the same size as the inputs and outputs of the given function F, here respectively 163 and 1. Tangent AD computes $F' \times$ `input_d`. Reverse AD computes `output_b`$^T \times F'$. From each, we compute the condensed Jacobian, see Figure 3. We also compute an approximate $F' \times$ `input_d` by finite differences.

Time performance is evaluated by comparing the gradient computation run-time with that of the primal simulation. Storage needed for Data-Flow reversal is evaluated by measuring the peak size of the stack for ST-AD models, or the trace size for ADOL-C. For reference, we also time the Tangent AD code. Figure 4 shows the results for each AD model and language of the corresponding implementation.

As expected, costs are higher with the OO-AD model provided by ADOL-C, which is the price to pay for its superior flexibility with respect to the sophisticated constructs of C++, and for the fact that the OO-AD model applies regardless of the presence or absence of GC. If we take strictly as a reference the runtime of the primal code (0.127 s. like in the C implementation), the slowdown ratios are 7.1 for Tangent AD and 18.1 for Reverse AD, and more if we count taping time. Interpretation of these slowdowns is delicate: ADOL-C provides several primitives that perform Reverse AD, one of them taking only 1.04 s. but which requires preparatory work by Tangent AD costing 1.28 s. The 18.1 slowdown factor thus remains valid. As is well known, storage needs are also higher than with ST-AD.

At the other end of the spectrum, costs are quite low and basically equivalent for the ST-AD model provided by Tapenade for Fortran and for C. The two implementations apply the same coding style and therefore result in the same differentiation approach taken by Tapenade. The slight cost improvement with Fortran90 may result from the potentially restricted use of pointers and from the use of array notation both in primal and differentiated codes. Being Source-Transformation AD, both tests benefit from static data-flow analysis, among which activity analysis has a major impact. Being OO-AD, ADOL-C has no data-flow analysis. To be fair we ran another experiment with the ST-AD models with activity analysis turned off, resulting into slowdown factors 1.41 for Tangent, 3.27 for Reverse, with a peak stack size of 147 MegaBytes.

Performance with the Java implementation and its hand-written ST-AD differentiated codes show a significant slowdown compared to the ST-AD reference. Admittedly, the Java runtime is slightly higher. Still, the slowdown factors Tangent/primal and Reverse/primal are also higher. This may be due to a combination of reasons:

- There may be Java internal reasons, which are visible from the degradation of the Tangent/primal slowdown factor, although Tangent AD does not implement Data-Flow reversal.
- The Data-Flow reversal stack management primitives that we defined for Java do not lend themselves to the optimizations of Tapenade's `adStack.c` primitives.
- The GC-specific management, i.e. the bookkeeping table and the `finalize` callback, probably induces a cost, however small. To evaluate it, we enlarged the Java heap size so that

	F90/Tapenade	**C/Tapenade**	**Java/hand-written**	**C(++)/ADOL-C**
Primal time (s.)	0.125	0.127	0.200	3.09 *(with taping)*
Tangent time (s.)	0.133 (×1.06)	0.140 (×1.10)	0.312 (×1.56)	0.90
Reverse time (s.)	0.243 (×1.93)	0.266 (×2.09)	0.507 (×2.53)	2.30
Storage (Mb.)	107	107	107	1354

Figure 4: Runtime (in seconds) and memory use for each AD model. Timings are averaged on 20 runs, with a standard deviation below 2% (except for ADOL-C). In parentheses is the slowdown factor compared to the primal time.

GC does not occur. In that case the bookkeeping table remains but no calls to `finalize` occur. The associated gain was less than the standard deviation on time measurements.

Even if it is still 107 MegaBytes, the peak stack size indeed increases slightly due to GC management, by an amount negligible compared to the number of intermediate floating-point values stored.

We also verified our proposed strategy regarding GC occurring or not. By enlarging the Java heap size, we can suppress the need for GC. In that case, we observe that the Objects from the forward sweep are duly recycled in the backward sweep i.e. there is no re-creation of a new identical Object.

The setup for these experiments is publicly available for inspection and execution at `https://gitlab.inria.fr/llh/dataflowreversalmodels`

6 Related Work

To our knowledge, there has been no prior work on Data-Flow reversal for languages with GC. For instance, existing Reverse AD tools for Python prefer to pay the memory cost of avoiding Data-Flow reversal completely. This can be achieved by adopting an OO-AD model [9] therefore storing a complete tape. Another way is to prevent variable overwrite in the primal code, for example by using *immutable* variables, thus restricting to SSA (Static Single Assignment) primal code. This is for instance the choice of JAX [2] or of Zygote [7] for the Julia language. These approaches have in common that Data-Flow reversal is achieved through additional references in the backward sweep to variables of the forward sweep. Therefore, variables whose reference count would fall to zero in the primal code will have a higher reference count in the differentiated code, and will not be collected by GC. Consequently, heap memory use becomes much higher.

Our approach that adds a unique pseudo-address to each Object of the primal code is also an answer to the limitations of the **Rebase addresses** strategy for Data-Flow reversal without GC. Although we showed in [6] that rebasing addresses works well even on large applications, it is still not a fully general approach and checking its applicability is delicate.

Adding a pseudo-address is an idea directly inspired by OO-AD tools such as ADOL-C [11], dco [8], or CoDiPack [10]. One difference is that our method attaches one pseudo-address to each newly allocated Object or array, whereas OO-AD attaches one to each new value assigned to a differentiable variable. Consequently, the pseudo-address (a `long int`) grows much more rapidly with OO-AD approaches.

The classical operation of *serializing* may also use similar pseudo-addresses. Serializing means turning a graph-structured data built with pointers, into a serial representation that can be stored out of core, e.g. into a file. The original graph-structured data can then be rebuilt later from the serialized storage. One can see our strategy as a form of serialization that is selective (some

objects remain in core, and they may reference objects that have been serialized out of core) and that is distributed over time as data is progressively restored backwards.

7 Conclusion and outlook

We proposed an approach to implement Data-Flow reversal, i.e. a selective restoration of past memory states of a given computation, when the implementation language features Garbage Collection (GC). The main application we have in mind is Reverse Algorithmic Differentiation (AD), which is the most efficient way to obtain gradients of a scalar-valued function. Whereas Reverse AD has been mainly applied to Scientific Computing models written in languages without GC, it is gaining importance for Machine Learning applications, where the choice implementation language, Python, has GC.

Our approach relies on the existence of a GC callback method, such as `finalize`, invoked immediately before an Object is GC'ed. It also attaches a unique integer key to each Object, similar to the pseudo-address mechanism used by OO-AD tools (ADOL-C, dco, CoDiPack...). We sketched a proof that the Reverse AD stack, that holds intermediate computation data for later use by derivative computation, remains consistent compared to Reverse AD without GC.

In contrast with Data-Flow reversal based on SSA form, only the data from the forward sweep that is needed by the backward sweep is stored. Moreover, this data is stored into the Reverse AD stack, in a form that is serialized by construction, and that can be sent at will to some more distant memory to recover heap space.

We implemented our approach for Java, and validated it on a representative Scientific Computing application in Java. Experiments provide an indication on the overhead of the approach, compared to Reverse AD on more classical languages such as Fortran and C, and also compared to OO-AD tools in Reverse AD mode. We feel that experimental results can lead towards two different conclusions.

- On the one hand, one may consider that a Reverse AD slowdown factor of 2.5 is acceptable, even if not as good as the factor of 2 obtained with Fortran or C. This ST-AD overhead is still significantly better than the overhead of OO-AD, also on the front of memory use. A downside is the intricacy of the Reverse ST-AD code, making it harder to read and follow. We also note that methods such as `finalize` can probably be used only once, thus forbidding higher-order AD by recursive differentiation.

- On the other hand, one may consider that we are progressively introducing into ST-AD some ingredients of OO-AD e.g. pseudo-addresses, so why not switch to OO-AD altogether and get rid of the Data-Flow reversal problem? It is true that OO-AD cannot exhibit a human-readable Reverse AD code, but the Reverse ST-AD code is getting obscure anyway. Still, the memory overhead of OO-AD is hard to accept, and a trade-off between ST-AD and OO-AD is desirable.

Exploring trade-offs between ST-AD and OO-AD could be an interesting outlook of the present work. Most of the memory cost of OO-AD as we understand it is due to storing the computation sequence of the primal code, with all loops unrolled. This leads to many parts of the trace differing only by the offset of the pseudo-addresses they operate on. The static code analysis of ST-AD may be able to determine statically this offset and the address it is based on. It may be possible to devise an OO-AD model in which parts of the trace that differ only by an offset could be shrunk as just one copy, taking advantage of some ST-AD preliminary analysis work. In other words this would turn a part of the OO-AD trace into a program. From another angle, this could be viewed as an ST-AD model in which the backward sweep is not anymore

written in the language of the primal code, and therefore need not obey the language-specific style of this primal code (polymorphism, GC ...) but concentrates mostly on derivative computation.

Acknowledgments

We thank Andrea Walther for her help in setting up and interpreting the ADOL-C experiment.

References

[1] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind. Automatic Differentiation in Machine Learning: a survey. *Journal of Machine Learning Research*, 2018.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] Bruce Christianson, Shaun A. Forth, and Andreas Griewank, editors. *Special issue of Optimization Methods & Software: Advances in Algorithmic Differentiation*, 2018.

[4] M. Fagan, L. Hascoët, and J. Utke. Data representation alternatives in semantically augmented numerical models. In *6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, Philadelphia, PA, USA*, 2006.

[5] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[6] L. Hascoët and M. Morlighem. Source-to-source adjoint Algorithmic Differentiation of an ice sheet model written in C. *Optimization Methods and Software*, pages 1–15, 2017.

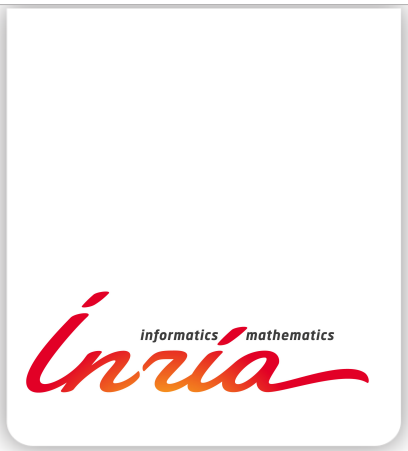
[7] M. Innes. Don't unroll adjoint: differentiating SSA-form programs. 2019. arXiv:1810.07951.

[8] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Demaison, L. Antiga, and A. Lerer. Automatic Differentiation in PyTorch. In *NIPS 2017 Workshop Autodiff*, 2017.

[10] M. Sagebaum, T. Albring, and N.R. Gauger. High-performance derivative computations using CoDiPack. *ACM Transactions on Mathematical Software*, 45(4), 2019.

[11] A. Walther and A. Griewank. Getting started with adol-c. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, chapter 7, pages 181–202. Chapman-Hall CRC Computational Science, 2012.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399