The Tapenade Automatic Differentiation tool: principles, model, and specification

Laurent HASCOET, Valérie PASCUAL

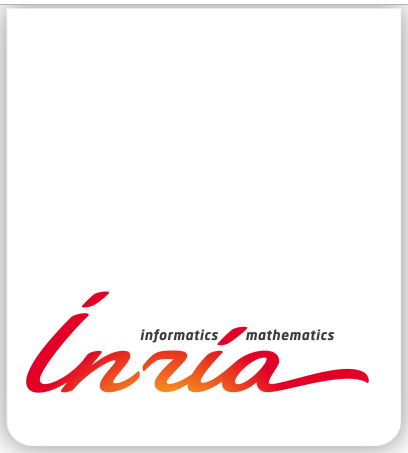
RESEARCH REPORT

N° 7957

May 2012

Project-Team Tropics

ISSN 0249-6399 ISRN INRIA/RR--7957--FR+ENG



The Tapenade Automatic Differentiation tool: principles, model, and specification

Laurent HASCOET, Valérie PASCUAL

Project-Team Tropics

Research Report n° 7957 — May 2012 — 50 pages

Abstract: Tapenade is an Automatic Differentiation tool which, given a Fortran or C code that computes a function, creates a new code that computes its tangent or adjoint derivatives. Tapenade puts particular emphasis on adjoint differentiation, which computes gradients at a remarkably low cost. This paper describes the principles of Tapenade, a subset of the general principles of AD. We motivate and illustrate on examples the AD model of Tapenade, i.e. the structure of differentiated codes and the strategies used to make them more efficient. Along with this informal description, we formally specify this model by means of Data-Flow Equations and rules of Operational Semantics, making this the reference specification of the tangent and adjoint modes of Tapenade. One benefit we expect from this formal specification is the capacity to study formally the AD model itself, especially for the adjoint mode and its sophisticated strategies. This paper also describes the architectural choices of the implementation of Tapenade. We describe the current performances of Tapenade on a set of codes that include industrial-size applications. We present the extensions of the tool that are planned in a foreseeable future, deriving from our ongoing research on AD.

Key-words: Automatic differentiation, Program transformation, Compilers, Preprocessors, Operational semantics, Program analysis, Adjoint, Gradient

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

L'outil de Différentiation Automatique Tapenade: principes, modèle et spécification

Résumé : Tapenade est un outil de Différentiation Automatique (DA) qui, étant donné un code Fortran ou C calculant une fonction, crée un nouveau code qui calcule ses dérivées tangente ou adjointe. Tapenade porte une attention particulière à la différentiation adjointe, qui calcule des gradients très efficacement. Nous décrivons les principes de DA utiles pour comprendre Tapenade. Nous motivons les choix qui nous guident dans son développement. Nous illustrons sur des exemples courts le modèle de différentiation choisi et les stratégies pour produire un code différentié efficace. Après cette description intuitive, nous donnons une spécification formelle de Tapenade au moyen d'équations data-flow et de règles de Sémantique Opérationnelle. Cette formalisation peut servir de base à des preuves de correction, principalement pour le mode adjoint. Nous décrivons enfin l'architecture de Tapenade, ses structures de données principales, et nous présentons ses performances sur des applications de taille industrielle. En conclusion, nous présentons les recherches en cours ou prévues.

Mots-clés : Différentiation automatique, Transformation de programmes, Compilateurs, Préprocesseurs, Sémantique opérationnelle, Analyse de programmes, Adjoint, Gradient

Contents

The Tapenade Automatic Differentiation tool: principles, model, and specification

May 10, 2012

1 Introduction

Computing accurate derivatives of a numerical model $f : I\!R^n \to I\!R^m$ is a crucial task in many domains of Scientific Computing. This is a computation-intensive task for which research and development of software tools are most wanted.

Automatic Differentiation (AD) is a collection of techniques to obtain analytical derivatives of differentiable functions, in the case where these functions are provided in the form of a computer program. Instead of using a mathematical representation, explicit or implicit, of these functions, AD uses the computer program itself as the basis to compute the derivatives. In general, AD techniques are made available to the end-users through specific software libraries or through software tools. For most applications, e.g. for optimization, AD competes with hand-coded derivatives, with methods that approximate derivatives such as *Divided Differences*, or even with methods that don't use derivatives such as Evolutionnary Algorithms or Stochastic Methods. Therefore, research on AD focuses on:

- designing techniques and models to compute efficient analytical derivatives of functions given as programs,
- developing libraries and tools that make these techniques directly available to end-users, and
- applying AD to large-size applications to demonstrate its potential and compare its benefits with alternative methods.

This paper is intended as a reference for Tapenade, the AD tool that we have been developing at INRIA since 1999. Therefore we will restrict our description of AD techniques to those actually relevant to this tool. Several research articles describe the existing AD techniques, many in the AD conference proceedings [4, 3, 2]. Other invaluable sources are the `www.autodiff.org` website or the most complete book [9]. Specifically, since Tapenade relies on automatic transformation of the source program, the alternative AD techniques that use operator overloading are outside the scope of this paper. Similarly, we will not discuss second nor higher derivatives, although we did some experiments with those, nor Taylor expansions. Tapenade focuses on first-order derivatives.

Rather than a plain presentation of Tapenade describing its Java implementation, the available tool options and other ways to interact with it, which may change over time and is more the role of a user manual [12], this paper is meant to specify and formalize the AD model that was chosen for Tapenade, and give the motivation for this choice. By model, we mean the approach

that we have selected (source transformation on imperative programs, association by name, adjoint mode by control- and data-flow reversal, Store-All trajectory recovery mitigated with some recomputation, Checkpointing...) among all approches explored by AD research. We also mean a number of enhancements that we contributed to this model, often based on global data-flow analysis. These enhancements, many of them to the so-called *adjoint mode*, were inspired over the years by application of Tapenade to large scientific software.

We feel that this model, including our own contributions to it, has been mostly described so far in an informal or intuitive way. Although necessary, this description is insufficient to make a sound link with actual implementation, nor to study the model itself in order to prove important correctness properties. This becomes even more annoying as the refinements to the model accumulate, making it increasingly complex.

This paper intends to fill this gap by providing a formal specification of AD by source transformation. This specification is given as a set of inference rules in the Natural Semantics fashion [13], a branch of Operational Semantics [19]. These rules can be on one hand implemented directly as an AD tool, or on the other hand used by formal proof systems to establish properties of the AD transformation itself. As will be shown, this transformation relies on global data-flow analysis of the source, and this paper will specify these analyses uniformly with data-flow equations. Like the inference rules, these data-flow equations are the basis of the implementation and can be used in formal proofs. We will provide in this manner an almost complete specification of Tapenade, except for a few global code simplifications that take place after differentiation. We believe this formalization has a larger reach and applies to many AD models in general.

This paper is organized as follows: Section 2 presents the principles of AD, or at least those principles that are useful to understand Tapenade. Section 3 lists the choices that directed development of Tapenade. This section also contrasts these choices with other tools and related works. Section 4 describes the model of differentiation that Tapenade follows. It is an informal description based on examples, intended to help an end-user understand the code produced, including most optimizations. Section 5 is the formal specification of Tapenade. It obeys the model and is the guideline for the actual implementation. This section describes the actual implementation in detail regarding global architecture and internal representation of differentiated programs. It then gives the equations of the data-flow analyses and the Operational Semantics of the tangent and adjoint differentiation. Little detail needs to be added on the actual Java implementation. Section 6 gives short practical notes on Tapenade and installation directions, followed by a synthetic view of the performances of derivative code produced on a selection of applications of all sizes. Section 7 concludes with developments planned.

2 Elements of Automatic Differentiation

Given a computer algorithm `P` (identified with a piece of program) that implements a function $F : X \in I\!R^n \mapsto Y \in I\!R^m$, AD builds a new algorithm (a program piece) `P'` that computes derivatives of F by computing the analytical derivative of each elementary mathematical operation in `P`. The fundamental observation is that any run-time trace of the algorithm `P`

$$\{I_1; I_2; \dots I_p;\}$$

computes the composition of elementary mathematical functions, one per instruction I_k,

$$f_p \circ f_{p-1} \circ \cdots \circ f_1 \ ,$$

which we can identify to F. This is of course assuming that `P` is a correct implementation of F, i.e. the discretization and approximation employed in `P` are sufficiently accurate and do not introduce non-differentiability.

Let us clarify the correspondence between the mathematical variables $(X, Y \dots)$ and the program variables found in P. As imperative programs classically overwrite their variables to save memory space, let us call V the collection of all the program variables of P and consider that each instruction I_k (partly) overwrites V. With these conventions (this run-time trace of) P is indeed the program:

original program P	
	Initialize V *with* X
(I_1)	V := f_1(V)
	...
(I_k)	V := f_k(V)
	...
(I_p)	V := f_p(V)
	Retrieve Y *from* V

At any given location in P, the program variables V correspond to one particular set, or vector, of mathematical variables. We will call this vector X_k for the location between instructions I_k and I_{k+1}. The set V is actually large enough to accomodate X, Y, or each successive X_k. At each location, V may thus "contain" more than the X_k but only the X_k play a role in the semantics of the program. The program instruction V := f_k(V) actually means taking from V the mathematical variables X_{k-1} before the instruction and applying $f_k(X_{k-1})$ to obtain X_k. After I_k, V corresponds to X_k. The *Initialize with* and *Retrieve from* instructions in the program sketch define $X_0 = X$ and identify Y to X_p.

Since we identify F with a composition of functions, the chain rule of calculus gives the first-order full derivative, i.e. the Jacobian:

$$F'(X) = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \dots \times f'_1(X_0) \ .$$

It is thus possible in theory to adapt algorithm P so that it computes $F'(X)$ in addition to $F(X)$. This can be done simply by extending instruction I_1 that computes $X_1 = f_1(X_0)$ with a piece of code that computes $J_1 = f'_1(X_0) \times Id$, and by extending likewise every instruction I_k by a piece of code that computes $J_k = f'_k(X_{k-1}) \times J_{k-1}$. This transformation is local to each instruction I_k. It is not limited to straight-line code and can be applied to any program P with control. The extended algorithm P′ just reproduces the control decisions taken by P. Of course, derivatives are valid only if the control does not change in an open neighborhood around X. Otherwise, the risk is that AD may return a derivative in cases where F is actually non-differentiable. Keeping this caveat in mind, the adapted algorithm can return J_p, the complete Jacobian $F'(X)$. However, the J_k are matrices whose height and width are both of the order of the number of variables in the original P, and may require too much memory space.

To work around this difficulty, one observes that the derivative object that is needed for the target application is seldom the full Jacobian matrix, but rather one of the two projections

$$F'(X) \times \dot{X} \quad \text{or} \quad \overline{Y} \times F'(X)$$

where $\dot{X}$ is some vector in $\mathbb{R}^n$ whereas $\overline{Y}$ is some row-vector in $\mathbb{R}^m$. Moreover when $F'(X)$ is needed explicitly, it is very often sparse and can therefore be retrieved from a relatively small number of the above projections. This motivates the so-called tangent and adjoint modes of AD:

- **Tangent mode:** evaluate $\dot{Y} = F'(X) \times \dot{X}$, the directional derivative of F along direction $\dot{X}$. It expands as

$$\dot{Y} = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \dots \times f'_1(X_0) \times \dot{X} \ . \tag{1}$$

Since $\dot{X}$ is a vector, this formula is most efficiently evaluated from right to left i.e., using mathematical variables:

$$\begin{array}{rcl} X_0 & = & X \\ \dot{X}_0 & = & \dot{X} \\ X_1 & = & f_1(X_0) \\ \dot{X}_1 & = & f_1'(X_0) \times \dot{X}_0 \\ & \ldots & \\ X_k & = & f_k(X_{k-1}) \\ \dot{X}_k & = & f_k'(X_{k-1}) \times \dot{X}_{k-1} \\ & \ldots & \\ X_p & = & f_p(X_{p-1}) \\ \dot{X}_p & = & f_p'(X_{p-1}) \times \dot{X}_{p-1} \\ Y & = & X_p \\ \dot{Y} & = & \dot{X}_p \end{array}$$

An algorithm $\dot{\texttt{P}}$ for this evaluation is relatively easy to construct, as the derivative instructions follow the order of the original instructions. Keeping the original program variables `V` to hold the successive X_k, and introducing a set of new program variables $\dot{\texttt{V}}$ of the same size as `V` to hold the successive $\dot{X}_k$, $\dot{\texttt{P}}$ writes:

tangent differentiated program $\dot{\texttt{P}}$	
	Initialize `V` *with* X *and* $\dot{\texttt{V}}$ *with* $\dot{X}$
$(\dot{I}_1)$	$\dot{\texttt{V}}$ `:=` $f_1'(\texttt{V}) \times \dot{\texttt{V}}$
(I_1)	`V :=` $f_1(\texttt{V})$
	$\ldots$
$(\dot{I}_k)$	$\dot{\texttt{V}}$ `:=` $f_k'(\texttt{V}) \times \dot{\texttt{V}}$
(I_k)	`V :=` $f_k(\texttt{V})$
	$\ldots$
$(\dot{I}_p)$	$\dot{\texttt{V}}$ `:=` $f_p'(\texttt{V}) \times \dot{\texttt{V}}$
(I_p)	`V :=` $f_p(\texttt{V})$
	Retrieve Y *from* `V` *and* $\dot{Y}$ *from* $\dot{\texttt{V}}$

Notice that each derivative statement $\dot{I}_k$ now precedes I_k, because I_k overwrites `V`.

- **Adjoint mode:** evaluate $\overline{X} = \overline{Y} \times F'(X)$, the gradient of the scalar function $\overline{Y} \times F(X)$ derived from F and weights $\overline{Y}$. It expands as

$$\overline{X} = \overline{Y} \times f_p'(X_{p-1}) \times f_{p-1}'(X_{p-2}) \times \ldots \times f_1'(X_0) \ . \tag{2}$$

Since $\overline{Y}$ is a (row) vector, this formula is most efficiently evaluated from left to right i.e.,

with mathematical variables:

$$\begin{aligned}
X_0 &= X \\
X_1 &= f_1(X_0) \\
&\dots \\
X_k &= f_k(X_{k-1}) \\
&\dots \\
X_p &= f_p(X_{p-1}) \\
Y &= X_p \\
\overline{X}_p &= \overline{Y} \\
\overline{X}_{p-1} &= \overline{X}_p \times f'_p(X_{p-1}) \\
&\dots \\
\overline{X}_{k-1} &= \overline{X}_k \times f'_k(X_{k-1}) \\
&\dots \\
\overline{X}_0 &= \overline{X}_1 \times f'_1(X_0) \\
\overline{X} &= \overline{X}_0
\end{aligned}$$

However, an algorithm that evaluates these formula is not immediate to construct, as the derivative instructions will follow the *inverse* order of the original instructions. Similarly to the tangent mode, we want the adjoint program to use only the original program's variables V plus a corresponding set of new program variables $\overline{\texttt{V}}$, of the same size as V, to hold the successive $\overline{X}_k$. In that case, we see that e.g. X_{k-1} contained in V will be overwritten by X_k and thus lost, before it is needed to evaluate $\overline{X}_k \times f'_k(X_{k-1})$. We will see later how this problem is solved, but let us keep in mind that there is a *fundamental penalty attached to the adjoint mode* that comes from the need of a data-flow (and control-flow) reversal.

Let us compare the run-time costs of the tangent and adjoint modes. Each run of the tangent differentiated algorithm $\dot{\texttt{P}}$ costs only a small multiple of the run-time of the original P. The ratio, that we will call R_t, varies slightly depending on the given P. Typical R_t ranges between 1 and 3. Using a simplified cost model that only counts the number of costly arithmetical operations (only *,/, and transcendentals), R_t is always less than 4. Similarly, for the adjoint differentiated algorithm $\overline{\texttt{P}}$, the run-time is only a small multiple of the run-time of P. The ratio, that we will call R_a, varies slightly depending on the given P. In the simplified cost model that only counts costly arithmetical computations, R_t and R_a are identical, but in practice $\overline{\texttt{P}}$ suffers from the extra penalty coming from the data-flow reversal. Typical R_a range between 5 and 10. Let us compare, with the help of figure 1, the costs of computing the complete Jacobian $F'(X)$, using no sparsity property, by employing either the tangent mode or the adjoint mode.

- With the tangent mode, we obtain $F'(X)$ column by column by setting $\dot{X}$ successively to each element of the Cartesian basis of the input space $\mathbb{R}^n$. The run time for the full Jacobian is thus $n \times R_t \times \text{runtime}(\texttt{P})$.
- With the adjoint mode, we obtain $F'(X)$ row by row by setting $\overline{Y}$ successively to each element of the Cartesian basis of the output space $\mathbb{R}^m$. The run time for the full Jacobian is thus $m \times R_a \times \text{runtime}(\texttt{P})$.

When n is much larger than m, the adjoint mode is recommended. In particular, this is the case when gradients are needed, e.g. in optimization or in inverse problems. There are typically

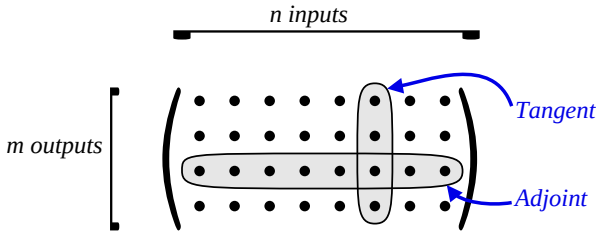


Figure 1: Elements of the Jacobian computable by tangent AD and adjoint AD

very few optimization criteria (often $m = 1$), and on the other hand n is often large as the optimization parameters may be functions, shapes, or other complex objects. In that case, no matter R_a being higher than R_t, the adjoint mode of AD is the only reasonable option. This is the most flagrant situation where adjoint AD can outperform all other strategies, in particular *divided differences* (i.e. evaluating $(F(X + h\dot{X}) - F(X))/h$) or even tangent AD.

Considering the design and implementation of AD tools, there are two principal ways to code the algorithms $\dot{\mathtt{P}}$ and $\overline{\mathtt{P}}$ namely, *operator overloading* and *program transformation*. The implementation choice for Tapenade is program transformation, and to motivate this requires an elementary description of these two ways.

- **Operator Overloading:** if the language of `P` permits, one can replace the types of the floating-point variables with a new type that contains additional derivative information, and overload the arithmetic operations for this new type so as to propagate this derivative information along. Schematically, the AD tool boils down to a library that defines the overloaded type and arithmetic operations. This approach is both elegant and powerful. The overloaded library can be quickly redefined to compute higher-order derivatives, Taylor expansions, intervals... By nature, evaluation of the overloaded operations will follow the original order of `P`. This is fine for the tangent mode, but requires some acrobacy for the adjoint mode, bearing severe consequences on performance and/or loosing a part of the elegance of the approach.

- **Program Transformation:** One can instead decide to explicitly build a new source code that computes the derivatives. This means parse the original `P`, build an internal representation, and from it build the differentiated $\dot{\mathtt{P}}$ or $\overline{\mathtt{P}}$. This approach allows the tool to apply some global analysis on `P`, for instance data-flow, to produce more efficient differentiated code. This is very similar to a compiler, except that it produces source code. This approach is more development-intensive than Operator Overloading, which is one reason why Operator Overloading AD tools appeared earlier and are more numerous. It also explains why Program Transformation AD tools are perhaps slightly more fragile and need more effort to follow the continuous evolution of programming constructs and styles. On the other hand, the possibility of global analysis makes Program Transformation the choice approach for the adjoint mode of AD, which requires control-flow and data-flow reversal and where global analysis is essential to produce efficient code.

These elements of AD are about all the background we need to describe a tangent mode AD tool. For the adjoint mode however, we need to address the question of *data-flow reversal*. We saw that the equations of the adjoint mode do not transpose immediately into a program because the values X_k are overwritten before they are needed by the derivatives. Basically, there are two ways to solve this problem, and a variety of combinations between them.

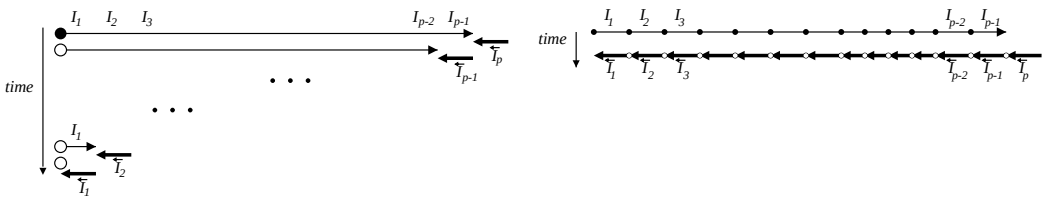


Figure 2: Data-flow reversal with Recompute-All approach (*left*) and Store-All approach (*right*). Big black dot represents storage of X_0, big white dots are retrievals. Small black dots are values being recorded before overwriting. Small white dots are corresponding restorations.

- *Recompute-All:* For each derivative instruction $\overleftarrow{I_k}$, we recompute the X_{k-1} that it requires by a repeated execution of the original code, from the stored initial state X_0 to instruction I_{k-1} that computes X_{k-1}. The memory cost is only the storage of X_0. On the other hand, the run time cost is quadratic in p.

- *Store-All:* Each time instruction I_k overwrites a part of `V`, we record this part of `V` into a stack just before overwriting. Later, we restore these values just before executing $\overleftarrow{I_k}$ The memory cost is proportional to p, whereas the run time cost comes from stack manipulation, usually minimal and proportional to p. For our example program `P`, this strategy results in the following $\overline{\texttt{P}}$

adjoint differentiated program $\overline{\texttt{P}}$	**(Store-All)**
	Initialize `V` *with* X *and* $\overline{\texttt{V}}$ *with* $\overline{Y}$
	`push(out`(I_1)`)`
(I_1)	`V :=` f_1`(V)`
	...
	`push(out`(I_k)`)`
(I_k)	`V :=` f_k`(V)`
	...
	`push(out`(I_{p-1})`)`
(I_{p-1})	`V :=` f_{p-1}`(V)`
$(\overleftarrow{I_p})$	$\overline{\texttt{V}} := \overline{\texttt{V}} \times f'_p(\texttt{V})$
	`pop(out`(I_{p-1})`)`
	...
	`pop(out`(I_k)`)`
$(\overleftarrow{I_k})$	$\overline{\texttt{V}} := \overline{\texttt{V}} \times f'_k(\texttt{V})$
	...
	`pop(out`(I_1)`)`
$(\overleftarrow{I_1})$	$\overline{\texttt{V}} := \overline{\texttt{V}} \times f'_1(\texttt{V})$
	Retrieve $\overline{X}$ *from* $\dot{\texttt{V}}$

This program $\overline{\texttt{P}}$ uses the Store-All approach, using `push` and `pop` primitives for stack manipulations, and definining **out**(I_k) to be the subset of the variables `V` that are effectively overwritten by I_k. We see two successive sweeps in $\overline{\texttt{P}}$. The *forward sweep* $\overrightarrow{\texttt{P}}$ is essentially a copy of `P` augmented with storage of overwritten values. The *backward sweep* $\overleftarrow{\texttt{P}}$ is the computation of the derivatives, in reverse order, augmented with retrieval of recorded values. Due to retrievals, the exit `V` does not contain the original result Y.

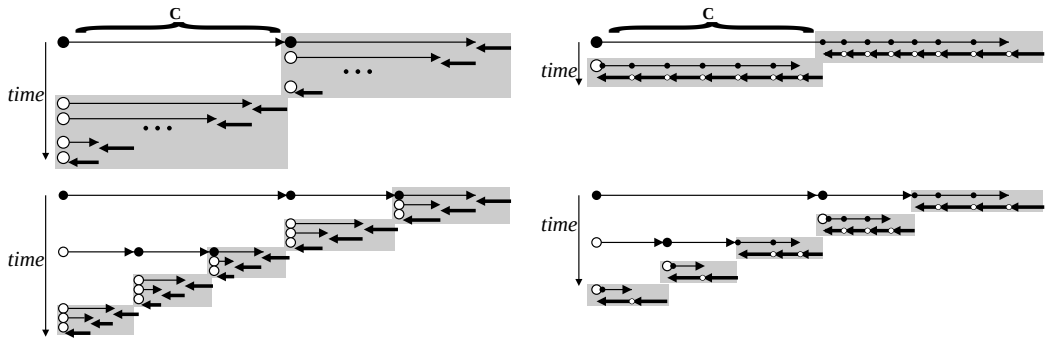


Figure 3: Checkpointing with the Recompute-All (*left*) and Store-All (*right*) approaches. The shaded areas reproduce the basic pattern of the chosen approach. *top:* single checkpointing, *bottom* nested checkpointing.

These data-flow reversal strategies can be adapted to programs `P` that are no longer straight-line, but that use control. This implies that the control decisions of the original code (branch taken, number of iterations, ...) or equivalently of the forward sweep, must be made available in reverse for the backward sweep. In a Store-All context, this can be done just like for data: control-flow decisions must be recorded at the exit of the control structure (conditional, loop,...) and then retrieved in the backward sweep to control the backward execution. This can use the same stack as data-flow reversal. The conjunction of the recorded control and data values is called the *trajectory*.

In practice, neither pure Recompute-All nor pure Store-All can be applied to large programs, because of their respective cost in run-time or memory space. Trade-offs are needed, and the classical trade-off is called *checkpointing*.

- In the Recompute-All approach, checkpointing means choosing a part `C` of `P` and storing the state upon exit from this part. Recomputing can then start from this state instead of the initial state. This is sketched on the top-left part of figure 3. At the cost of storing one extra state, the run-time penalty has been divided roughly by two. Checkpoints can be nested to further reduce the run-time penatly, as shown on the bottom-left of figure 3.

- In the Store-All approach, checkpointing means choosing a part `C` of `P` and *not* recording the overwritten values during `C`. Before the backward sweep reaches $\overleftarrow{\texttt{C}}$, `C` is run again from a stored state this time with recording. This is sketched on the top-right part of figure 3. At the cost of storing one extra state and of running `C` twice, the peak memory used to record overwritten data is divided roughly by two. Checkpoints can be nested to further reduce the peak memory consumption, as shown on the bottom-right of figure 3.

Notice on figure 3 that the execution scheme at the bottom, for nested checkpoints, become very similar. Recompute-All and Store-All are the two ends of a spectrum, with optimal checkpointing scheme(s) lying somewhere in between. A good placement of (nested) checkpoints is crucial for efficient adjoint differentiation of large codes. Natural candidates to become a checkpointed part are procedure calls and loop bodies, but any piece of code with a single entry point and a single exit point can be chosen. There is no formula nor efficient algorithm to find this optimal placement of checkpoints, except in the case [8] of a loop with a known number of iterations all of the same cost. A good enough default strategy is to apply checkpointing at the

level of each procedure call. In practice, it is important to give the end-user the freedom to place checkpoints by hand. Good placements of checkpoints perform similarly: their memory and run-time costs grow with the logarithm of the run-time of P or more precisely

- the peak memory size during execution, to store states and record data, grows like the logarithm of the run-time of P.
- the maximum number of times a checkpointed piece of the program is re-executed, which approximates the slowdown factor of $\overline{\mathtt{P}}$ compared to P, also grows like the logarithm of the run-time of P. This explains why the slowdown ratio R_a of $\overline{\mathtt{P}}$ compared to P can be larger by a few units than the ratio R_t of $\dot{\mathtt{P}}$ compared to P.

Actually, one need not store the entire memory state to checkpoint a program piece C. What must be stored is called the *snapshot*. In a Store-All context, we can see that a variable need not be in the snapshot if it is not used by C. Likewise, a variable need not be in the snapshot if it is not overwritten between the initial execution of C and the execution of its adjoint $\overline{\mathtt{C}}$. This can be refined further, *cf* section 5.4.

3 Design Choices Rationale

This section presents the design choices made when developing Tapenade. When appropriate, we give references to the underlying techniques and contrast with the choices made by other AD tools.

Source Transformation: as the main focus or our research is on the adjoint mode and how to make it efficient on very large applications, we decided to stick with the program transformation approach. This approach was the choice made by the ancestor of Tapenade, Odyssée, developed at INRIA in the years 1990 [20]. An efficient adjoint mode should completely remodel the control-flow structure of the code to achieve control-flow reversal. While Operator Overloading AD tools such as ADOL-C [24] need to store a trace of all operations made by the code to run it backwards, Program Transformation AD tools can store as little as a history of branches taken, of size in practice small compared to the tape of intermediate values computed [18]. Of equal importance is the fact that global data-flow analysis such as TBR [11] allow the AD tool to reduce significantly the quantity of intermediate values that need to be stored on that tape. Let us mention two other interesting AD approaches that we have left aside. One way to build an AD tool that is truly independent from the source language is to apply AD to the assembly code. The problem is that there are also many different assembly languages. Moreover, compilation has wiped out structural information that can be used to optimize the AD code. Another way is to embed the AD process directly inside a compiler [17], in its "middle end" which is very often independent both from the source language and from the target assembly language. This can take advantage from the data-flow analyses already performed by the compiler, but these are not global due to the principle of separate compilation. Also implementation of the adjoint mode appears hard, due to a complex internal representation specialized for compilation.

Complete support of imperative languages: the tool's primary goal is to experiment and validate refinements of the adjoint AD model. This implies to run it on real, large application codes such as the ones found in industry. Also, a widespread use of the tool in an industrial context would be definitely a mark of the quality of its algorithms. For both reasons, we wanted the tool to be used on real-size applications, not asking for heavy preprocessing nor hand preparation. In other words, we didn't want to indulge restrictions of the tool with respect to the size of the application, nor to the language effectively accepted. Accepting large applications essentially means choosing a compact internal representation and a careful design of data-flow

analysis, such that the complexity of data-flow analyses remain close to linear with respect to code size and, depending on the analysis, linear or quadratic with respect to the number of variables. Accepting the full application language means treating correctly memory constructs such as global variables, Fortran `COMMON`'s, `EQUIVALENCE`'s, and control constructs such as `GOTO`'s, that are sometimes neglected when only studying principles. This means a precise model of the internal memory, and the use of control-flow graphs (flow graphs for short) rather than mere syntax trees.

Readability: the output of the tool is a new source code, in which the end-user should be able to recognize the structure of the original code. This is useful not only for debugging purposes. This requires that the internal representation of codes keeps information that a compiler normally discards, and which is used during the source reconstruction phase. As a consequence, we don't want the tool to begin with a set of "canonicalization" steps, unless they can be reversed in the end.

Source language independence: obviously, the application language should not be only Fortran, like it was for Odyssée. The differentiation algorithms must not be expressed in terms of one given application language, but rather on an abstract language, called IL for *Imperative Language*, that is the sole input and output of the tool. IL must be rich enough to represent all imperative programming constructs, including Object Oriented constructs for future extension of the tool to these languages. Whenever possible, a unique IL operator must be used to represent similar constructs of different languages. This design choice, also made by the AD tool OpenAD [23, 22], allows a refinement in the AD model to benefit at the same time to applications written in Fortran or C.

Scoping: the internal representation must be rich enough to represent variable scoping. We already motivated our choice of flow graphs. Each basic block of these graphs points to a symbol table, and these symbol tables are nested to implement scoping. The hierarchy of nested symbol tables must provide separation between public/formal parameters symbol table and private/local variables symbol table, and a mechanism to import the public symbol table of an imported module.

Arrays as atoms: every static data-flow analysis must face the problem of arrays and array indices. To distinguish data-flow information of different cells of arrays, one must be able to evaluate and compare each array indices in the code. This is out of reach of a static analysis in general, one reason being nondecidability. This can be achieved only in special cases such as regular array indices, but at a high development cost. While array index analysis is certainly vital for some static analysis tools such as loop-parallelizing or loop-vectorizing compilers, it is questionable whether it is profitable for an AD tool. Even if we discover that only a fraction of an array needs a derivative counterpart, leading to a smaller derivative array, the index computations needed to implement the correspondence between an array cell and its derivative cell may be very costly and will certainly hamper readability of the differentiated code. As a result, the design choice is to consider arrays as atoms and to accept the incurred information "blurring". If an array access `A(i)` modifies data-flow information, this will conservatively affect information on the complete array `A`. As only one cell of `A` is affected in reality, a "must" information on `A(i)` will result in a weaker "may" information on `A`. There are two cases where we will afford a little more accuracy on array data-flow analysis:

- when an array is split in sequential fragments by means of a Fortran `EQUIVALENCE` or `COMMON`, we will consider each fragment as a separate memory zone for data-flow analysis. The data-flow information may then be more accurate on each fragment.

- when every access to an array inside a loop has the same index expression, and this index is a simple function of the loop indices, e.g. with a constant offset, data-flow analysis will

gain accuracy by temporarily individualizing array elements.

Distinguished structure components: In contrast to array elements, components of structures must be separated. Our belief is that organizing data as an array conceptually reflects that these data have a similar behavior. Organizing them in a structure does not reflect a similar behavior. Therefore, data-flow information on one structure component must not affect information on the other components of the same structure.

Association by name: A memory reference, e.g. `a[i].x`, must be associated in some way to its differentiated counterpart. We choose *association by name*, i.e. to create new variables with new names. On our example, this means creating a new `ad` and referring to `ad[i].x`. In contrast, we don't choose *association by address*, i.e. to attach the derivative to the address of each reference by extending the type of atomic components. On our example, this would mean extending the type of the `x` component and referring to `a[i].x.diff`. AD tools that rely on overloading naturally use association by address. Our choice strongly impacts the differentiated code [6]. In some cases such as pointers, it may appear awkward. We believe a future version of Tapenade should offer both options.

Context-sensitive and Flow-sensitive Data-Flow analysis: Accurate data-flow analysis must be flow-sensitive and context-sensitive [21], which is easier to achieve on a graph-based representation i.e. a call graph of flow graphs. The analyses perform a number of fixed-point sweeps on these graphs. However, different calling contexts may lead to different differentiated procedures, and there is a risk of combinatorial explosion. To avoid that, our choice is **generalization** instead of specialization: at the time of differentiation, only one differentiated subroutine is built for each original subroutine. In other words, the analyses implement context-sensitivity, but its effects at differentiation time are restricted.

Store-All in adjoint mode, Save-on-kill: For the adjoint mode, we choose the Store-All approach to recover intermediate values. Intermediate values are saved just before a statement may overwrite them (*Save-on-kill*). We don't consider the alternative choice of storing the partial derivatives themselves (*preaccumulation*). OpenAD [23] is also basically Store-All, unlike TAF [7] which is basically Recompute-All. This contrast is reduced by the use of checkpointing as shown in figure 3. Also, Tapenade applies when possible a limited form of local recomputing instead of storing.

4 The Tapenade Automatic Differentiation model

This section intends to give the reader familiarity with the tangent and adjoint codes produced by Tapenade, going from simple creation of differentiated variable names to the more sophisticated improvements of adjoint AD. Unless otherwise stated, the illustration codes use the Fortran syntax but they easily transpose to C.

4.1 Symbol names

If a variable `v` has a non-trivial derivative, this derivative must be stored. As Tapenade uses association by name, we need a new variable name which is built by adding a user-defined prefix and/or suffix to `v`. By default in tangent mode the new variable will be `vd` ("`v` *dot*"), and in adjoint mode `vb` ("`v` *bar*"). Similarly, Tapenade AD model introduces derivatives of `COMMONS`, procedures, interfaces, user-defined types, and modules, and must build new names for them. By default this will be done by appending "`_d`" in tangent mode and "`_b`" in adjoint mode. Figure 4 shows the transformation of the declaration part of a procedure. If a conflict is found with some existing symbol, then a "0" is appended, then "1" until finding a free symbol.

original code	Tapenade **tangent**	Tapenade **adjoint**

`SUBROUTINE T1(a)`
`  REAL a(10),w`
`  INTEGER jj`
`  TYPE(OBJ1) b(5)`
 	
`  COMMON /param/ jj,w`
	`SUBROUTINE T1_D(a,ad)`
	`  REAL a(10),ad(10),w,wd`
	`  INTEGER jj`
	`  TYPE(OBJ1) b(5)`
	`  TYPE(OBJ1_D) bd(5)`
	`  COMMON /param/ jj,w`
	`  COMMON /param_d/ wd`
		`SUBROUTINE T1_B(a,ab)`
		`  REAL a(10),ab(10),w,wb`
		`  INTEGER jj`
		`  TYPE(OBJ1) b(5)`
		`  TYPE(OBJ1_B) bb(5)`
		`  COMMON /param/ jj,w`
		`  COMMON /param_b/ wb`

Figure 4: Names of differentiated symbols

4.2 Types of differentiated variables

When a variable has a non-trivial derivative, Tapenade builds and declares a new differentiated variable whose type comes from the type of the original variable. When the variable is real or complex, more or less modified according to the language as in `REAL*8` or `double`, the differentiated variable has exactly the same type. Variables of non-differentiable types such as booleans, integers, characters, and strings simply have no differentiated variables, and therefore no type for those. When the original variable is a pointer to a destination with non-trivial derivative, the differentiated variable is also a pointer. When the original variable is an array of elements with non-trivial derivatives, the differentiated variable is also an array with the same dimensions. Recalling section 3 about array analysis, tapenade makes the choice of creating a derivative array with the same dimension, even if only a few of the array elements actually have a non-trivial derivative.

Things are slightly more complex for user-defined structured types[1]. From the design choice to keep most of the structure of the original code, Tapenade declares the derivative variable also with a structured type. However this structured type may differ since components that never have a non-trivial derivative are just wiped out. In this case, the original structured type gives birth to a differentiated structured type, and Tapenade will declare it. For each component `x` of the original structured type `T`, there will be a component `x` of the same name and type in the differentiated structured type `T_D` or `T_B` if somewhere in the code there is a variable of type `T` with a non-trivial derivative for its component `x`. Notice that this clearly makes the choice of generalization *vs* specialization: as different variables of type `T` may have different components holding non-trivial derivatives, specialization may create many differentiated types for `T`, for a benefit that we judge negligible. This choice of generalization will be made again for subroutines and functions (*cf* section 4.5).

Figure 5 illustrates this on a small tangent mode example, where structured type `VECTOR` is differentiated into `VECTOR_D`, and type components `y` and `name` do not appear in `VECTOR_D` because there are never any derivatives to store there.

Please notice the way references to structure components are differentiated. With our choice of association by name, the derivative of component `b%x` is stored in `bd%x`, i.e. the `x` component of the derivative name associated to `b`. If we chose association by address, the `VECTOR_D` type would contain components `x`, `y`, `z`, and `name`, where `x` and `z` would be of type, say, `ACTIVE_REAL` instead of `REAL`. Access to the primitive value would become `b%x%v` and to the derivative `b%x%d`.

[1] Fortran 95 standard uses the term *derived type*. This terminology leads to confusion with "derivative", of course frequent in AD. We'll rather speak of a *structured type*

original code	Tapenade **tangent**
...	...
	TYPE VECTOR_D
	REAL :: x,z
	END TYPE VECTOR_D
TYPE VECTOR	TYPE VECTOR
CHARACTER(64) :: name	CHARACTER(64) :: name
REAL :: x,y,z	REAL :: x,y,z
END TYPE VECTOR	END TYPE VECTOR
TYPE (VECTOR) u,v,w	TYPE (VECTOR) u,v,w
	TYPE (VECTOR_D) ud
...	...
FUNCTION TEST(a,b)	FUNCTION TEST_D(a,ad,b,bd,test)
REAL TEST	REAL TEST, TEST_D
TYPE (VECTOR) a,b	TYPE (VECTOR) a,b
	TYPE (VECTOR_D) ad,bd
	TEST_D = a%x*bd%x + ad%x*b%x + ud%z
TEST = a%x*b%x + u%z	TEST = a%x*b%x + u%z
...	...

Figure 5: Differentiated Data Types

4.3 Simple assignments

Now consider an assignment I_k. In *tangent* mode (*cf* equation (1)), derivative instruction $\dot{I}_k$ implements $\dot{X}_k = f'_k(X_{k-1}) \times \dot{X}_{k-1}$, with initial $\dot{X}_0 = \dot{X}$. In *adjoint* mode (*cf* equation (2)), derivative instruction(s) $\overleftarrow{I_k}$ implements $\overline{X}_{k-1} = \overline{X}_k \times f'_k(X_{k-1})$, with initial $\overline{X}_p = \overline{Y}$.

In practice, the original code may overwrite variables, i.e. implement $X_k = f_k(X_{k-1})$ through an assignment I_k that puts the resulting X_k into the memory locations that used to contain X_{k-1}. Tapenade will do the same for the derivative assignments: the old $\dot{X}_{k-1}$ will be overwritten by the new $\dot{X}_k$, and the old $\overline{X}_k$ will be overwritten by the new $\overline{X}_{k-1}$. Suppose for instance I_k is `a(i)=x*b(j) + COS(a(i))`, with `a`, `b`, and `x` having nontrivial derivatives (*cf* section 4.6). It is straightforward to write the Jacobian of the corresponding function

$$f_k : \begin{array}{ccc} \mathbb{R}^3 & \to & \mathbb{R}^3 \\ \begin{pmatrix} \texttt{a(i)} \\ \texttt{b(j)} \\ \texttt{x} \end{pmatrix} & \mapsto & \begin{pmatrix} \texttt{x} * \texttt{b(j)} + \texttt{COS(a(i))} \\ \texttt{b(j)} \\ \texttt{x} \end{pmatrix} \end{array}$$

which is the matrix

$$\begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

from which we write the operations that $\dot{I}_k$ and $\overleftarrow{I_k}$ must implement:

$$\dot{I}_k \text{ implements } \begin{pmatrix} \dot{\texttt{a(i)}} \\ \dot{\texttt{b(j)}} \\ \dot{\texttt{x}} \end{pmatrix} = \begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \dot{\texttt{a(i)}} \\ \dot{\texttt{b(j)}} \\ \dot{\texttt{x}} \end{pmatrix},$$

$$\overleftarrow{I_k} \text{ implements } \begin{pmatrix} \overline{\texttt{a(i)}} & \overline{\texttt{b(j)}} & \overline{\texttt{x}} \end{pmatrix} = \begin{pmatrix} \overline{\texttt{a(i)}} & \overline{\texttt{b(j)}} & \overline{\texttt{x}} \end{pmatrix} \times \begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

Tapenade thus produces the derivative instructions shown on figure 6.

Tapenade **tangent**	Tapenade **adjoint**

`ad(i) = xd*b(j) + x*bd(j)`
`- ad(i)*SIN(a(i))`
	`xb = xb + b(j)*ab(i)`
	`bb(j) = bb(j) + x*ab(i)`
	`ab(i) = -SIN(a(i))*ab(i)`

Figure 6: Differentiation of a simple assignment

4.4 Sequences of instructions and control

The expressions from the Jacobian, e.g. `SIN(a(i))` in section 4.3 refer to values found in X_{k-1}. The differentiation model must ensure that these values are available when the derivative instructions are executed.

In tangent mode, this is achieved by putting the derivative instructions of I_k just *before* a copy of I_k. Thus the tangent differentiated code for a procedure `P` is basically a copy of `P` with derivative instructions inserted just before each assignment statement. Procedure calls are treated differently, *cf* section 4.5.

In Adjoint mode, our design choice is fundamentally the *store-all* strategy (*cf* section 3). Therefore control-flow reversal uses a two-sweeps architecture. The first sweep (the *forward sweep*) is basically a copy of `P`, augmented with recording of the control. This recorded control is used by the second sweep (the *backward sweep*) to orchestrate control-flow reversal. The forward sweep also records intermediate data values (the X_k), that will be used by the backward sweep to evaluate elements of the Jacobian.

As both control- and data-flow are reversed, the natural way to record is to use a stack that grows during the forward sweep and shrinks during the backward sweep. Because of this stack architecture, data values must be recorded *after* their last use forward, so as to be restored *before* their first use in the backward sweep. The simplest choice to achieve this (Save-on-kill), saves a data value immediately before an instruction overwrites it. Concerning control, the analogue of Save-on-kill consists in saving control values just before control-flow merges, rather than when it splits just after the forward control tests. One advantage is that this choice applies naturally to any flow graph, even not cleanly structured. At the same time, the reversed control-flow retains most of the structure of the original code. The storage strategies for control-flow and for data-flow are compatible, which means one stack is enough to store both. Figure 7 shows the Tapenade tangent and adjoint AD models on a representative example.

original code	Tapenade **adjoint, forward sweep**

`DO i=3*n*m,1000`
 	
`  a = a - 2*B(i)`
`  IF (a.gt.0) THEN`
 	
`    c = c*a`
 	
`  ENDIF`
 	
`ENDDO`
	`tmpF = 3*n*m`
	`DO i=tmpF,1000`
	`  CALL PUSHREAL4(a)`
	`  a = a - 2*B(i)`
	`  IF (a.gt.0) THEN`
	`    CALL PUSHREAL4(c)`
	`    c = c*a`
	`    CALL PUSHCONTROL(0)`
	`  ELSE`
	`    CALL PUSHCONTROL(1)`
	`  ENDIF`
	`ENDDO`
	`CALL PUSHINTEGER4(tmpF)`

Tapenade **tangent**	Tapenade **adjoint, backward sweep**

`DO i=3*n*m,1000`
`  ad = ad - 2*Bd(i)`
`  a = a - 2*B(i)`
`  IF (a.gt.0) THEN`
`    cd = cd*a + c*ad`
`    c = c*a`
`  ENDIF`
`ENDDO`
	`CALL POPINTEGER4(tmpF)`
	`DO i=1000,tmpF,-1`
	`  CALL POPCONTROL(branch)`
	`  IF (branch.eq.0) THEN`
	`    CALL POPREAL4(c)`
	`    ab = ab + c*cb`
	`    cb = a*cb`
	`  ENDIF`
	`  CALL POPREAL4(a)`
	`  Bb(i) = Bb(i) - 2*ab`
	`ENDDO`

Figure 7: Tapenade tangent and adjoint on some code with control

4.5 Procedure calls

Procedure calls are treated differently from atomic instructions such as assignments. In tangent mode, atomic instructions and their derivatives are finely interleaved as described in section 4.4. Therefore a procedure call cannot be preceded by some differentiated counterpart, but rather replaced by another, differentiated call. The differentiated procedure will contain at the same time the original computation and the derivative instructions. In adjoint mode, the original computation and the derivative computation are effectively separated. Therefore in the basic adjoint model of Tapenade, each call to a procedure `FOO` in the original code gives birth to two procedure calls in the adjoint:

- one call is located in the forward sweep of the calling context and calls a new procedure `FOO_FWD` that executes the forward sweep of `FOO`.
- the other call is located in the backward sweep of the calling context and calls a new procedure `FOO_BWD` that executes the backward sweep of `FOO`.

Notice however that this basic adjoint model is not the default behavior of Tapenade. The default behavior applies checkpointing (*cf* section 4.10) to each procedure call, yielding a different structure. But the basic model is effectively what one gets when checkpointing is turned off.

The arguments of the differentiated calls are deduced naturally from the arguments of the original call. Arguments that have a nontrivial derivative give birth to an additional argument for this derivative. Specifically, consider the more general case of a function call:

```
y = FOO(a, x, 2.5)
```

assuming that `x` and `y` have nontrivial derivatives, whereas `a` and of course the constant `2.5` have not. In the tangent differentiated code, the corresponding call is:

```
yd = FOO_D(a, x, xd, 2.5, y)
```

Notice the last argument `y` which holds the original result of `FOO`, as a function cannot return two results. In the forward sweep of the adjoint differentiated code, the corresponding call is simply:

```
y = FOO_FWD(a, x, 2.5)
```

that requires no argument change as the forward sweep does the same computations as `FOO` augmented with trajectory storage. No derivatives are used. On the other hand in the backward sweep, the corresponding call is:

```
call FOO_BWD(a, x, xb, 2.5, yb)
```

which is not a function any more because the original result `y` is not needed in the backward sweep. Because of reversal of the data-flow, `yb` is an input of `FOO_BWD` and if `x` is an input of `FOO`, `xb` is an output of `FOO_BWD`.

Data-flow reversal for the adjoint interacts with the parameter-passing discipline of the application language. This is transparent on a language that uses call by reference, or call by value-return like Fortran, but becomes apparent with call by value as used in C. An input argument of the original code can very well be passed by value. However the corresponding differentiated argument becomes an output of the adjoint differentiated procedure. Therefore the adjoint procedure will expect the differentiated argument to be a reference.

At the level of procedure calls, we again encouter the option of generalization versus specialization. Suppose function `FOO` is actually called at two locations:

```
y = FOO(a, x, 2.5)
b = FOO(c, z, t)
```

and suppose only `x`, `y`, `z`, and `t` have a nontrivial derivative. As `FOO` is used with two different contexts, there could be two different, specialized derivatives e.g., `FOO_D1` and `FOO_D2` in tangent mode. Tapenade makes the choice of generalization instead, with only one derivative of `FOO` in a context that is the conservative union of all actual call contexts. This prevents a potential combinatorial explosion of the differentiated code, but may cause unnecessary computation in `FOO_D` and requires passing explicit zero or dummy arguments for some derivatives. As `FOO_D` expects derivatives now for its result and arguments 2 and 3, the derivative code is:

```
yd = FOO_D(a, x, xd, 2.5, 0.0, y)
dumm = FOO_D(c, z, zd, t, td, b)
```

4.6 Activity

The past sections compose a naïve AD model that effectively computes the expected derivatives. It is the simplest AD model that Tapenade can actually obey, using a combination of command-line options that deactivate all further refinements. However, there are a number of efficiency improvements that can and should be done. They are the subject of the next sections.

In the previous sections we mentioned the notion of a nontrivial derivative. At a given location in the source, we may know statically that the derivative of a variable is either always null or useless for the desired final derivatives. This variable there is said *passive*. Otherwise, the variable is said *active*, which means it has a nontrivial derivative. One could assume that all variables of a "differentiable" type (e.g. float, complex...) are active everywhere in the code. One can obviously do much better with the help of a static data-flow analysis called *activity analysis*. In case of doubt, the safe conservative answer is to assume a variable is active. Therefore active means "possibly active" and passive means "certainly passive".

Differentiation operates on an algorithm coded as a part (named `P` in section 2) of a bigger program. Among the inputs and outputs of `P`, the end user must designate the candidate variables for differentiation. This in turn specifies the context in which the differentiated code will be called and its results used:

- The *independent* variables are the subset of the inputs that will be differentiated. In *tangent* mode this means that the context must provide the input values of derivatives of the independent variables before calling the tangent differentiate code `P_D`, whereas input values of derivatives of other variables are meaningless and may very well be overwritten by `P_D`. In *adjoint* mode this means that the context has the right to use the values returned in the derivatives of the independent variables upon return from `P_B`, whereas return values of derivatives of other variables are meaningless and must not be used by the context.

- The *dependent* variables are the subset of the outputs that will be differentiated. In *tangent* mode this means that the context has the right to use the values returned in the derivatives of the dependent variables upon return from `P_D`, whereas return values of derivatives of other variables are meaningless and must not be used by the context. In *adjoint* mode this means that the context must provide the input values of the adjoint derivatives of the dependent variables before calling the adjoint differentiate code `P_B`, whereas input values of derivatives of other variables are meaningless and may very well be overwritten by `P_B`.

At any given location in the source code, a variable `v` is said *varied* (resp. *useful*) if there exists a run of `P` that reaches this location such that at this location `v` depends on (resp. influences)

one of the independents (resp. dependents) in a differentiable way. At any location, `v` is active if it is both varied and useful. Varied and useful variables will be detected by a static data-flow analysis on the source code. Tapenade model does not distinguish array elements: if one array element becomes active, so does the complete array.

The effect of activity analysis on the Tapenade model is to simplify the differentiated code, through slicing and partial evaluation.

- In tangent mode when `v` is not varied, or in adjoint mode when `v` is not useful, we know the current derivative of `v` is certainly null. The differentiated code is therefore simplified by forward partial evaluation.
- In tangent mode when `v` is not useful, or in adjoint mode when `v` is not varied, we know the current derivative of `v` is not needed for the rest of the derivative computation. The differentiated code is therefore simplified by backward slicing.

Figure 8 illustrates the result of activity on the tangent AD model.

original code	**naïve tangent**	**with activity analysis**
	`xd = a*bd + ad*b`	
`x = a*b`	`x = a*b`	`x = a*b`
	`ad = 0.0`	
`a = 5.0`	`a = 5.0`	`a = 5.0`
	`dd = a*cd + ad*c`	`dd = a*cd`
`d = a*c`	`d = a*c`	`d = a*c`
	`ed=ad/x-a*xd/x**2`	
`e = a/x`	`e = a/x`	`e = a/x`
	`ed = 0.0`	
`e=floor(e)`	`e = floor(e)`	`e = floor(e)`

Figure 8: Effect of activity analysis on Tangent AD model. Independents are `a`, `b`, `c`. Dependents are `d`, `e`. Partial evaluation follows from `a` becoming not varied after receiving constant 5.0. Slicing follows from `e` being not useful before being truncated by the call to `floor`.

To further reduce the cost of the differentiated instructions, we decided that derivatives that become null need not be immediately reset to 0.0. We say that they are *implicit-zero*. This happened twice in figure 8. This spares the reset statement and can be compensated by the next instruction that assigns to the derivative variable. However, the model must pay special attention when the control flow merges a branch in which the variable is implicit-zero with another branch in which the variable is active. The implicit-zero differentiated variable must be explicitly reset to 0.0 *before* the flow merges. All in all, this implicit-zero tactique yields a limited gain, at the cost of extra complexity in the code produced. Therefore, the end-user can turn off the implicit-zero mechanism with a command-line option.

4.7 Multi-directional extension

It happens that derivatives are required for the same values of the input parameters, i.e. at the same *point* X in the input space, but for several sets of values for the derivatives $\dot{X}$ or $\overline{Y}$. For example in section 2, we saw that the full Jacobian Matrix at a given point X can be obtained by repeatedly running the tangent code for each $\dot{X}$ in the Cartesian basis of the input space, or by repeatedly running the adjoint code for each $\overline{Y}$ in the Cartesian basis of the output space. At the same time, we observe that there are two sorts of instructions in a differentiated code, regardless tangent or adjoint:

- some instructions are in fact copies of the original code, and do not operate on derivative variables.
- some instructions actually compute the derivatives, using values both from the original code and from other derivatives computation

The copies of the original code operate on the same values and return the same results regardless of the derivatives. Therefore they can be factored out. To this end, the *multi-directional* extension of Tapenade produces a code where all instructions that operate on derivatives are encapsulated into an extra loop on all different values of $\dot{X}$.

Figure 9 illustrates multi-directional tangent mode on a small procedure. All derivative variables receive an additional dimension, that represents the different directions $\dot{X}$ in the input space. In the case of a variable of a structured type, or of array type, the additional dimension is placed at the deepest location in the data structure, so that taking a part of the variable can easily translate into taking the corresponding part of the derivative.

original code	**tangent**	**multi-directional**
`SUBROUTINE G(x,y)`	`SUBROUTINE G_D(x,xd,y,yd)`	`SUBROUTINE G_DV(x,xd,y,yd,` `  nbdirs)`
		`  USE DIFFSIZES`
`  REAL :: x, y(*)`	`  REAL :: x, y(*)`	`  REAL :: x, y(*)`
	`  REAL :: xd, yd(*)`	`  REAL :: xd(nbdirsmax)`
`  EXTERNAL F`	`  ...`	`  REAL :: yd(nbdirsmax,*)`
`  REAL :: F`		`  ...`
		`  REAL :: result1d(nbdirsmax)`
	`  xd = xd + yd(2)`	`  INTEGER :: nbdirs, nd`
`  x = x + y(2)`	`  x = x + y(2)`	`  x = x + y(2)`
		`  DO nd=1,nbdirs`
		`  xd(nd) = xd(nd) + yd(nd, 2)`
	`  xd = xd*x + x*xd`	`  xd(nd) = xd(nd)*x + x*xd(nd)`
		`  END DO`
`  x = x*x`	`  x = x*x`	`  x = x*x`
	`  result1d =`	`  CALL F_DV(y(9),yd(:,9),`
	`  F_D(y(9),yd(9),result1)`	`  result1,result1d,nbdirs)`
		`  DO nd=1,nbdirs`
	`  yd(10) = xd*result1`	`  yd(nd, 10) = xd(nd)*result1`
	`  + x*result1d`	`  + x*result1d(nd)`
		`  END DO`
`  y(10) = x*F(y(9))`	`  y(10) = x*result1`	`  y(10) = x*result1`
`END SUBROUTINE G`	`END SUBROUTINE G_D`	`END SUBROUTINE G_DV`

Figure 9: Tangent `G_D` and multi-directional tangent `G_DV` derivatives of a procedure `G`. `nbdirsmax` is a static constant to be defined in module `DIFFSIZES`, that defines the size of the extra dimension. Argument `nbdirs` ($\leq$`nbdirsmax`) is the actual number of differentiation directions for one particular run.

Tapenade makes a data-dependency analysis on the differentiated code that allows differentiated instructions to safely move about the code. In particular, this allows the tool to group loops on differentiation directions when possible, and then merge them into a single loop. This reduces loop overhead, see figure 9. More importantly since these loops are essentially parallel, this builds larger parallel sections e.g. for a multi-threaded execution context like OpenMP.

At present, there is no equivalent multi-directional extension for the adjoint mode. However this extension could be made provided some sizeable application has a need for it.

4.8 Splitting and merging the differentiated instructions

From this section on, the features described concern only the adjoint mode of AD. These features are probably more complex and demanding regarding implementation. On the other hand, Tapenade clearly focuses on an efficient adjoint mode, and the benefits of this mode are certainly worth the effort.

The adjoint differentiation model applies elementary transformations to the differentiated code, such as splitting and merging instructions, to eliminate redundant operations and improve performance. Instructions merging relies on Tapenade running data-dependency analysis on the basic blocks of the differentiated code. These transformations remain localized to each basic block of the control-flow graph. Although they are local to each basic block, these transformations are not redundant with compiler optimizations because basic blocks of numerical programs are often larger than the peep-hole optimization window of the compiler. Also, the compiler doesn't know the special structure of an adjoint code. Let us consider as an example the following basic block:

```
res = (tau-w(i,j))*g(i,j)*(z(j)-2.0)/v(j)
g(i,j) = res*v(j)
```

Differentiation of complex expressions often introduces common subexpressions. Tapenade does not look for common sub-expressions in the original code, but at least it should not introduce new ones. In the context of AD, the problem of common subexpression detection is

naïve adjoint	**split and merge**

`resb = v(j)*gb(i, j)`
`vb(j) = vb(j) + res*gb(i, j)`
`gb(i, j) = 0.0`
`taub = taub`
`  +(z(j)-2.0)*g(i, j)*resb/v(j)`
`wb(i, j) = wb(i, j)`
`  -g(i, j)*(z(j)-2.0)*resb/v(j)`
`gb(i, j) = gb(i, j)`
`  +(z(j)-2.0)*(tau-w(i, j))*resb/v(j)`
`zb(j) = zb(j)`
`  +(tau-w(i, j))*g(i, j)*resb/v(j)`
`vb(j) = vb(j)`
`  -(tau-w(i, j))*g(i, j)*(z(j)-2.0)*resb/v(j)**2`
	`resb = v(j)*gb(i, j)`
	`temp = (z(j)-2.0)/v(j)`
	`tempb0 = temp*g(i, j)*resb`
	`tempb = (tau-w(i, j))`
	`  *g(i, j)*resb/v(j)`
	`vb(j) = vb(j)`
	`  +res*gb(i, j) -temp*tempb`
	`gb(i, j) = temp`
	`  *(tau-w(i, j))*resb`
	`taub = taub + tempb0`
	`wb(i, j) = wb(i, j) - tempb0`
	`zb(j) = zb(j) + tempb`

Figure 10: Splitting and merging adjoint differentiated instructions

simpler than in the general case because we know how derivative statements are structured and built, and therefore when new common subexpressions are introduced. On our current example, assuming that `res`, and only `res`, is passive outside the basic block, the adjoint statements of the naïve backward sweep could be as shown on the left of figure 10. However, from the structure of the derivative statements, Tapenade can find that three subexpressions are duplicated and are worth being precomputed namely, `(z(j)-2.0)/v(j)`, `g(i, j)*(z(j)-2.0)*resb/v(j)`, and `(tau-w(i, j))*g(i, j)*resb/v(j)`. It is also apparent that the two statements incrementing `vb(j)` can be merged, and that one statement resets `gb(i, j)` to zero before it is later incremented. The right of figure 10 shows the differentiated code after all this splitting and merging.

4.9 Improving trajectory computation

The adjoint differentiation model of Tapenade applies several improvements to eliminate useless instructions from the trajectory computation done by the forward sweep. This section describes three such improvements, that must be applied in order. We will illustrate this on the following piece of code, considered as the complete body of the differentiated procedure, so that its adjoint forward sweep is immediately followed by its backward sweep:

```
n = IND1(i)
B(n) = (A(n)+B(n))*0.5
A(n) = A(n)+x
c = A(n)*B(n)
A(n) = A(n)*A(n+1)
n = IND2(i)
Z(n) = Z(n) + c
n = IND2(i+1)
c = A(n)/B(n)
T(n) = T(n) - c
```

Figure 11 shows the successive adjoint codes after each improvement is performed. Diff-liveness is about original instructions present in the forward sweep, that are actually not needed because their result is used for the original result but not for any derivative. Tapenade considers that the adjoint code must not bother to compute the result of the original, non-differentiated code. The only important result are the derivatives. Therefore, we perform a slicing of the forward sweep to take away original computation that is not needed for the derivatives. This slicing goes backwards from the end of the forward sweep.

At first sight, this slicing is of limited impact as it applies only "near" the end of the forward sweep. Moreover, some end-users require that the adjoint code also returns the original result, and therefore this slicing may need to be deactivated by a command-line option. However, this slicing is also done for every part of the code where *checkpointing* is applied (*cf* section 4.10), i.e. for every place where a forward sweep and its backward sweep meet. In this case the original result is definitely useless.

In the example of figure 11, we can see that the values of `t(n)` and `z(n)`, as well as the two successive values of `c` are not used in the derivatives computation. Therefore the corresponding assignments are not *diff-live*, i.e. they are dead code for the adjoint. This also spares the `PUSH/POP` statements that save into the trajectory the values that are overwritten by these assignments.

To-Be-Recorded analysis (**TBR**) deals with original values which, although they are diff-live, do not appear in the derivative. For instance if a variable is always used in *linear* expressions, it never appears in the derivative computations. In the example, this is the case for the initial `a(n)`, and also for the more trivial cases of the initial `n` and `b(n)` that are simply not used at all. This takes away three pairs of `PUSH/POP` calls.

Finally, recomputation analysis looks for variables that are recorded in the trajectory, and that could instead be recomputed because their value is the result of a relatively cheap assignment and all inputs to this assignment are still available at the time of the `POP`. In this case, it is preferable to recompute the value by duplicating the assignment, rather than using precious memory space to record it. In the example, this is the case for the first two values of variable `n`, but note that this is not limited to index variables and can occur for float variables in the computation.

naïve	**Diff-liveness**	**TBR**	**recomputation**
CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)		
n = ind1(i)	n = ind1(i)	n = ind1(i)	n = ind1(i)
CALL PUSHREAL4(b(n))	CALL PUSHREAL4(b(n))		
b(n)=(a(n)+b(n))*0.5	b(n)=(a(n)+b(n))*0.5	b(n)=(a(n)+b(n))*0.5	b(n)=(a(n)+b(n))*0.5
CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))		
a(n) = a(n) + x	a(n) = a(n) + x	a(n) = a(n) + x	a(n) = a(n) + x
CALL PUSHREAL4(c)			
c = a(n)*b(n)			
CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))
a(n) = a(n)*a(n+1)	a(n) = a(n)*a(n+1)	a(n) = a(n)*a(n+1)	a(n) = a(n)*a(n+1)
CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	
n = ind2(i)	n = ind2(i)	n = ind2(i)	
CALL PUSHREAL4(z(n))			
z(n) = z(n) + c			
CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	
n = ind2(i+1)	n = ind2(i+1)	n = ind2(i+1)	n = ind2(i+1)
CALL PUSHREAL4(c)			
c = a(n)/b(n)			
CALL PUSHREAL4(t(n))			
t(n) = t(n) - c			
CALL POPREAL4(t(n))			
cb = -tb(n)	cb = -tb(n)	cb = -tb(n)	cb = -tb(n)
CALL POPREAL4(c)			
tempb = cb/b(n)	tempb = cb/b(n)	tempb = cb/b(n)	tempb = cb/b(n)
ab(n) = ab(n)+tempb	ab(n) = ab(n)+tempb	ab(n) = ab(n)+tempb	ab(n) = ab(n)+tempb
bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)
-a(n)*tempb/b(n)	-a(n)*tempb/b(n)	-a(n)*tempb/b(n)	-a(n)*tempb/b(n)
CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	n = ind2(i)
CALL POPREAL4(z(n))			
cb = zb(n)	cb = zb(n)	cb = zb(n)	cb = zb(n)
CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	n = ind1(i)
CALL POPREAL4(a(n))	CALL POPREAL4(a(n))	CALL POPREAL4(a(n))	CALL POPREAL4(a(n))
ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)
+a(n)*ab(n)	+a(n)*ab(n)	+a(n)*ab(n)	+a(n)*ab(n)
ab(n) = b(n)*cb	ab(n) = b(n)*cb	ab(n) = b(n)*cb	ab(n) = b(n)*cb
+a(n+1)*ab(n)	+a(n+1)*ab(n)	+a(n+1)*ab(n)	+a(n+1)*ab(n)
CALL POPREAL4(c)			
bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)
+a(n)*cb	+a(n)*cb	+a(n)*cb	+a(n)*cb
CALL POPREAL4(a(n))	CALL POPREAL4(a(n))		
xb = xb + ab(n)	xb = xb + ab(n)	xb = xb + ab(n)	xb = xb + ab(n)
CALL POPREAL4(b(n))	CALL POPREAL4(b(n))		
ab(n) = ab(n)	ab(n) = ab(n)	ab(n) = ab(n)	ab(n) = ab(n)
+0.5*bb(n)	+0.5*bb(n)	+0.5*bb(n)	+0.5*bb(n)
bb(n) = 0.5*bb(n)	bb(n) = 0.5*bb(n)	bb(n) = 0.5*bb(n)	bb(n) = 0.5*bb(n)
CALL POPINTEGER4(n)	CALL POPINTEGER4(n)		

Figure 11: Diff-liveness, TBR, and recomputation improvements applied successively to a small example

4.10 Checkpointing

Checkpointing (*cf* section 2) is the standard way to organize a tradeoff between run-time and memory consumption in adjoint AD. In our context that is based on the *store-all* strategy, checkpointing applied to a fragment `C` of an original code `P=U;C;D` amounts to replacing the normal two-sweep adjoint of `P`

$$\overline{\mathtt{P}} = \overrightarrow{\mathtt{P}}; \overleftarrow{\mathtt{P}} = \overrightarrow{\mathtt{U}}; \overrightarrow{\mathtt{C}}; \overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}; \overleftarrow{\mathtt{C}}; \overleftarrow{\mathtt{U}};$$

by

$$\overline{\mathtt{P}} = \overrightarrow{\mathtt{U}}; \bullet; \mathtt{C}; \overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}; \circ; \overrightarrow{\mathtt{C}}; \overleftarrow{\mathtt{C}}; \overleftarrow{\mathtt{U}},$$

where $\bullet$ and $\circ$ respectively store and restore a *snapshot* i.e. "enough" values to guarantee that the execution context is the same for runs of `C` and of $\overrightarrow{\mathtt{C}}; \overleftarrow{\mathtt{C}}$ The gain is that the trajectory for `C` and for `D` are never stored at the same time in memory. The cost is the relatively small memory size of the snapshot, and the time for the extra run of `C`.

By default, the Tapenade adjoint AD model applies checkpointing systematically at every procedure call. Therefore checkpoints are nested just like calls are nested. Checkpointing changes radically the differentiated code for a procedure call, as shown in figure 12. When the call to

Without checkpointing	**With checkpointing**
	`CALL PUSHREAL4(a)`
	`CALL PUSHREAL4(x)`
`y = FOO_FWD(a, x, 2.5)`	`y = FOO(a, x, 2.5)`
`...`	`...`
	`CALL POPREAL4(x)`
	`CALL POPREAL4(a)`
`CALL FOO_BWD(a, x, xb, 2.5, dummyb, yb)`	`CALL FOO_B(a, x, xb, 2.5, dummyb, yb)`

Figure 12: The effect of checkpointing on the adjoint of a procedure call

`FOO` is checkpointed, the forward sweep stores a snapshot which is specific for each call site, then calls plain `FOO`. The backward sweep restores the snapshot, then calls `FOO_B` which contains both forward and backward sweeps for `FOO`.

When checkpointing is applied to `C`, the adjoint code contains the sequence $\overrightarrow{\mathtt{C}}; \overleftarrow{\mathtt{C}}$, in addition to the sequence $\overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}$ that was already present before checkpointing was applied. Recalling that the code improvements such as diff-liveness discussed in section 4.9 bear more fruit when the forward sweep is immediately followed by the backward sweep, we see that checkpointing opens more opportunity to simplify the adjoint code. In any case, when checkpointing is applied, an AD tool should restart the static data-flow analyses for the checkpointed part, in order to discover these additional simplifications.

The default checkpointing strategy of Tapenade can be amended in several ways. The directive `$AD NOCHECKPOINT` attached to a procedure call deactivates checkpointing on this particular call. To deactivate checkpointing on all calls, attach the directive to the procedure header itself. Conversely, the pair of directives `$AD CHECKPOINT-START` and `$AD CHECKPOINT-END` define an arbitrary portion of code (well-structured) on which checkpointing must be applied.

Concerning loops, directive `$AD BINOMIAL-CKP` attached to the loop header tells Tapenade to apply binomial checkpointing as defined in [8], which is the optimial strategy to apply to iterative loops such as time-stepping loops. Finally, directive `$AD II-LOOP` attached to the loop header tells Tapenade that the loop has Independent Iterations. Checkpointing can be therefore applied to each iteration of the loop, thus reducing the peak memory used for trajectory storage [10].

4.11 Adjoint of array notation

The adjoint model applies nicely to Fortran array notation. It exhibits a natural duality between the sum reductions and the spread operations that turn a scalar into an array. Masks and `where` guards are preserved by the transformation. Figure 13 presents a few examples, which can be checked manually by writing the instruction's Jacobian like in section 4.3.

original	**adjoint**
`a(:) = 2.9*a(:) + x`	`xb = xb + SUM(ab(:))` `ab(:) = 2.9*ab(:)`
`s = s` `  + SUM(a(:)*x, mask=b(:)>0)`	`xb = xb + sb*SUM(a(:), mask=b(:)>0)` `where(b(:)>0) ab(:) = ab(:) + x*sb`
`where(t(:)<0) a(:) = r*r`	`rb = rb + 2.0*r*SUM(ab(:), mask=t(:)<0)` `where(t(:)<0) ab(:) = 0.0`

`a(1:n) = c(0:n-1)`
`  *SUM(d(:,:) dim=2)`
	`cb(0:n-1) = cb(0:n-1) + SUM(d(:,:),dim=2)*ab(1:n)`
	`db(:,:) = db(:,:) + SPREAD(c(0:n-1)*ab(1:n),n,2)`
	`ab(1:n) = 0.0`

Figure 13: Adjoints of a few example Fortran array operations

5 Specification and notes on actual implementation

We will give the specification of Tapenade, from its general architecture down to the individual analyses and differentiation transformation. The architecture part is the basis on which analyses and differentiation run. It is therefore described in every necessary detail. It corresponds exactly to the actual implementation. The analyses and differentiation are specified in a more abstract way, namely with data-flow equations and inference rules of Operational Semantics. They can be used for formal reasoning on the AD model. Their link with the actual implementation is discussed in the end of the section.

5.1 Internal Representation

Figure 14 summarizes the architecture of Tapenade, motivated by our design choices. In its left part, this architecture resembles that of a compiler, building an internal representation and running data-flow analysis on it. A big difference, though, is that efficient AD analyses must be global and therefore there is no such thing as separate compilation: all the source code must be parsed, then analyzed jointly. The right part of figure 14 differs from a compiler in that Tapenade produces its result back in the original programming language instead of machine code. This imposes additional constraints to keep the result readable, and to maintain some degree of resemblance between source and differentiated codes. For instance unlike a compiler, the internal representation will store the order and arrangement of declarations inside a procedure. Although this order is useless for the data-flow analysis and differentiation phases, as declarations are compiled into symbol tables, it is used to regenerate declarations for the differentiated code in a matching order.

Language independence is enforced by delegating parsing and unparsing to separate language-specific modules, each of which communicate with the central part through a generic Imperative Language (IL). IL is an abstract language, with no concrete syntax, i.e. no textual representation. Parsing any source code produces an IL tree, whose nodes are IL operators. IL provides enough operators for all constructs of the accepted source languages, such as procedure definition,

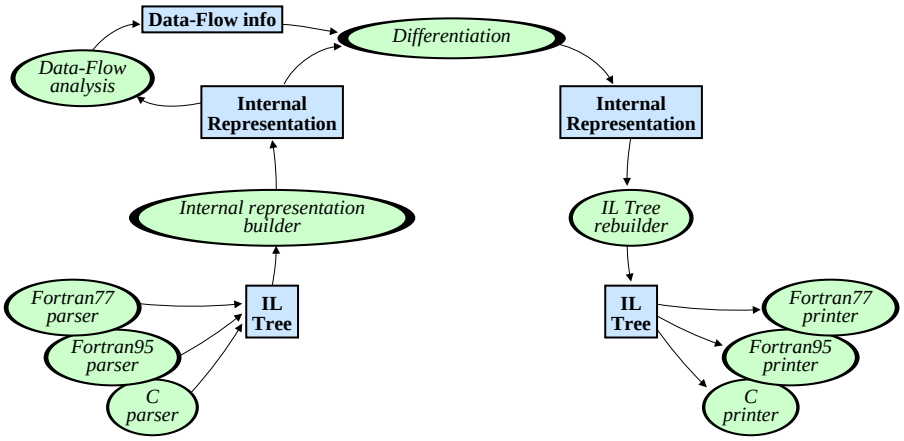


Figure 14: General architecture of Tapenade

variable declaration, assignment, procedure call, variable reference... Overlapping is promoted: at present, 71 IL operators are common to Fortran and C, 55 are Fortran-specific and 26 are C-specific. In addition, 19 operators exist in provision for Object languages. Promotion of overlapping sometimes implies (reversible) tree transformation, such as using the more explicit C-style pointer access methods or using a generalized form of memory allocation.

However, the central part of Tapenade i.e. all data-flow analysis and actual differentiation, operate on an internal representation of programs which is more elaborate than plain IL trees. This internal representation is sketched in figure 15, and its motivation follows. As we chose to make data-flow analysis context-sensitive, the internal representation must provide easy access to the call graph. The call graph is on the left part of figure 15. The example illustrates the 3 kinds of arrows: `Top` *contains* `M3` which *contains* `P5`, `M5` *imports* (or *uses*) `M3`, `P7` *calls* `P5`. In fact, the top level of each analysis consists of a sweep on the call graph, visiting each procedure possibly repeatedly until reaching a fixed-point. This also proves useful for differentiation itself, as the differentiated code is built procedure per procedure.

At the level of an individual procedure, the internal representation must essentially capture the flow of control. The natural representation is therefore a classical flow graph, whose nodes are *Basic Blocks* (sequential lists of instructions) and whose arrows are jumps from a Basic Block to another. In figure 15, all the flow graphs are gathered in the middle part. Notice that the flow graph is reduced to a single Basic Block for declarative units such as modules. When several arrows flow from one Basic Block, the instruction at the end of this Basic Block is the control instruction that decides the arrow taken at run-time. This uniform representation of control flow makes it easier for all data-flow analysis to sweep through the procedure code in a flow-sensitive way. It also makes it easier for the differentiation tool to generate a differentiated flow graph, leaving the details of how to create a code that implements this differentiated flow graph to the IL Tree rebuilder. In theory the declaration statements could be taken out of the flow graph, since they are not "executed". However we will keep them because their order is important. Moreover, keeping the original form and order of declarations enhances readability of the differentiated code.

Efficient access from a symbol name (variable, procedure, type ...) to its relevant information is ensured through Symbol Tables, sketched on the right part of figure 15. In particular data-flow analysis requires that each procedure call provides efficient access to the Call Graph node(s)

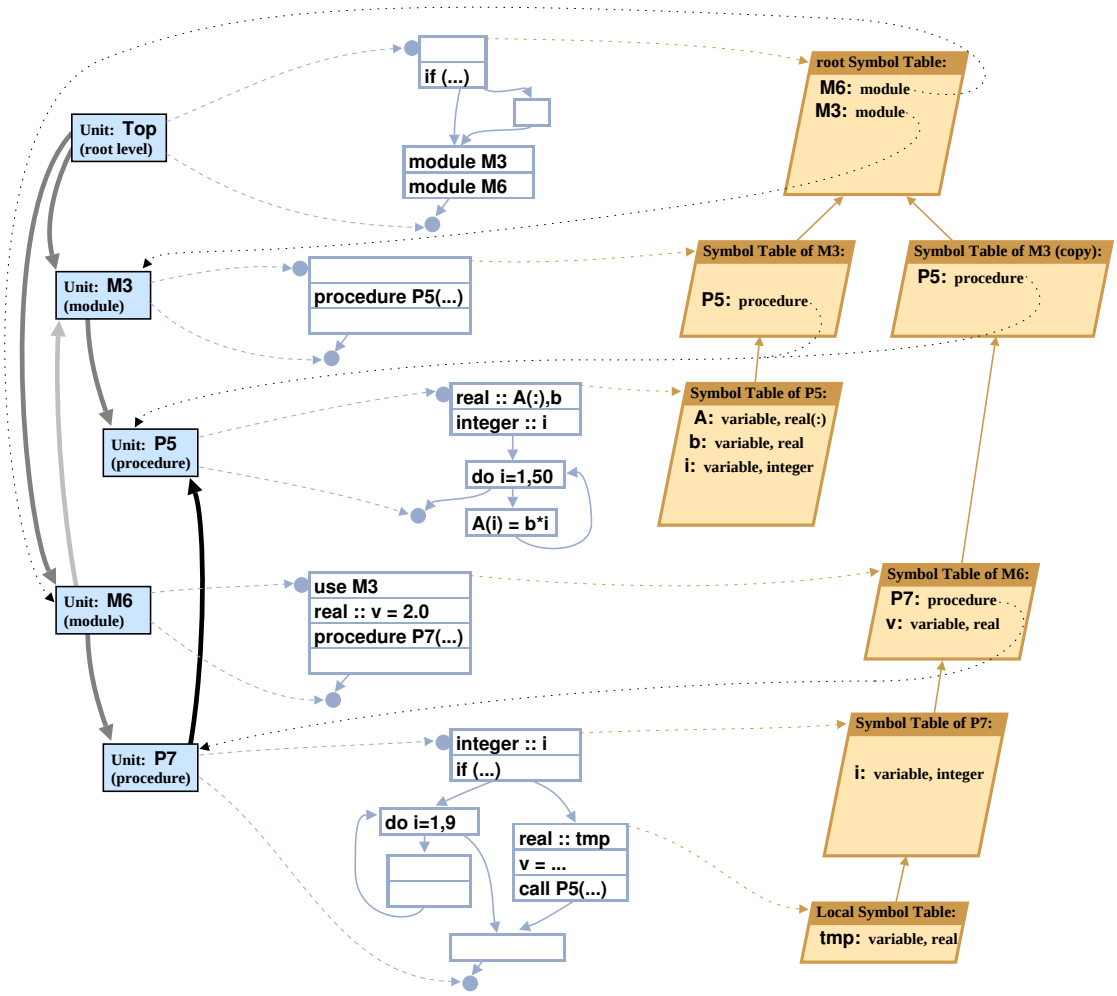


Figure 15: Internal representation of a program inside Tapenade

of the called procedure, which can be several when overloading comes into play. This is done through the Symbol Table of the calling location. Symbol Tables use a hash code algorithm. Symbol Tables are organized as a tree to implement scoping. A Symbol Table can be copied elsewhere in the tree, with possible renaming, when another program element "imports" or "uses" it. Each Symbol Table sees the symbols that are defined in its parent Symbol Table. Each Basic Block has a link to the deepest Symbol Table it can see. Searching for the definition of a symbol from a Basic Block starts from this Symbol Table, then if not found continues to the parent Symbol Table possibly until the "root" Symbol Table.

Most symbols in the Symbol Table are in fact associated with a kind (variable, procedure...) a type, and miscellaneous information. When the symbol is a procedure name, its entry in the Symbol Table contains in addition the corresponding node of the Call Graph, itself related to each of its callers and callees. This node in turn has a link to its flow graph.

The complete network of all these links as illustrated by figure 15 is not so intricate after all. This is about all we need to implement the data-flow analysis and program transformations of Tapenade.

At the level of an individual statement, the internal representation can remain the IL Tree itself, as the very simple control in this statement is adequately represented by this tree. However, when an instruction of the original code incorporates some degree of control such as an if-expression or a function call nested in an expression, then the internal representation must split this instruction into simpler instructions. In practice, this is done selectively and creates far less splitting than a systematic normalization step.

When a symbol is associated with storage (a variable, but also in the future an object or a class), it is given a (bunch of) unique integer number that represent its memory location(s). This allows all data-flow analysis to manipulate bitsets, based on these integer numbers, called *zones*. A scalar or an array of scalars receive only one zone. On the other hand, a structured object receives one zone per "component" or "field". We believe this greatly contributes to efficiency of the data-flow analysis.

5.2 Data-Flow Equations of program static analysis

Static data-flow analysis [1] is an essential step to achieve efficient differentiation. In other words, almost every improvement to our Automatic Differentiation model required a new specific data-flow analysis in Tapenade. Figure 16 shows them all.

Regarding notation, when a data-flow information **Info** is defined for a given location in the code, i.e. in our case before or after an arbitrary instruction I, we will write $\mathbf{Info}^-(I)$ (resp. $\mathbf{Info}^+(I)$) the value of **Info** immediately before (resp. after) I.

To begin with, we need to run two classical, AD-independent data-flow analyses that are needed almost everywhere:

- **Pointer analysis** finds *pointer destinations*, which tell for each instruction and for each available pointer, the smallest possible set of possible targets of this pointer. This complex pointer analysis is not specific to AD and is outside the scope of this paper.
- **In-out analysis** finds *in-out sets*, which contain for each procedure the smallest possible sets of variables that may or must be read or overwritten by this procedure. This composite piece of information is best represented by four booleans per variable, true when during some execution of the procedure, a part of the input value of the variable may be
  - read and then overwritten,
  - read and not overwritten,

- not used but overwritten,
- neither used nor overwritten.

The in-out sets also include, for each instruction I in each procedure, the sets $Cst(I)$ (resp. $Dead(I)$) of variables that must-remain constant (resp. unused) from this location till the exit from the procedure. As a side effect, in-out analysis makes it possible to detect potential aliasing and potential use of uninitialized variables, yielding warning messages especially valuable in the context of AD.

There is a minor issue of relative dependence between the two analyses. In some cases, pointer destinations analysis could benefit from the in-out sets, but the influence of pointer destinations on in-out analysis is far more important and dictates the order in which the analyses are run. Pointer destinations and in-out sets are necessary to almost all subsequent analyses in Tapenade.

Then come the AD-specific analyses. They are needed to implement the differentiation model described in sec. 4. Their relative dependences are shown in figure 16. We start with activity analysis, that allows for the improvements discussed in sec. 4.6. Then come diff-liveness and TBR analyses, allowing for the improvements discussed in sec. 4.9, and finally comes adjoint-out analysis, which is one ingredient of the snapshots used when checkpointing, *cf* section 4.10. For efficiency reasons we introduce two preliminary analyses which, for each procedure, compute useful summarized information used in activity, diff-liveness, and adjoint-out analyses: dependency analysis and diff-dependency analysis. TBR and adjoint-out analyses are needed only by the adjoint mode of AD. We will now give the data-flow equations of these AD-specific analyses.

- **Activity analysis** is the combination of a forward and a backward analysis. It propagates forward the **Varied** set of the variables that depend in a differentiable way on some independent input. Similarly, it propagates backwards the **Useful** set of the variables that influence some dependent output in a differentiable way. Since the relation "depends in a differentiable way of" is transitive on code sequences, the essential equations of the propagation are:

$$\begin{aligned} \mathbf{Varied}^+(I) &= \mathbf{Varied}^-(I) \otimes \mathbf{Diff\text{-}dep}(I) \\ \mathbf{Useful}^-(I) &= \mathbf{Diff\text{-}dep}(I) \otimes \mathbf{Useful}^+(I) \end{aligned}$$

in which the diff-dependency relation $\mathbf{Diff\text{-}dep}(I)$ between variables before I and after I is such that $(v1.v2) \in \mathbf{Diff\text{-}dep}(I)$ iff the variable $v2$ after I depends in a differentiable way on the variable $v1$ before I. Composition $\otimes$ is defined naturally as:

$$v2 \in S \otimes \mathbf{Diff\text{-}dep}(I) \iff \exists v1 \in S | (v1.v2) \in \mathbf{Diff\text{-}dep}(I)$$

and likewise in the reverse direction $\mathbf{Diff\text{-}dep}(I) \otimes S$. When I is a simple statement such as an assignment, $\mathbf{Diff\text{-}dep}(I)$ can be computed on the fly: the left-hand-side variable depends differentiably on all right-hand-side variables except those that occur in a non-differentiable way such as indices or other address computation. All other variables are unchanged and therefore depend on themself only. When the left-hand-side is only a part of its variable v, e.g. an array element, we take the other elements of v into account, conservatively adding $(v.v)$ into $\mathbf{Diff\text{-}dep}(I)$. When on the other hand I is a call to a procedure P, then we rely on the preliminary diff-dependency analysis to provide us with $\mathbf{Diff\text{-}dep}(P)$. The data-flow equations are closed with the equations at control-flow branches:

$$\begin{aligned} \mathbf{Varied}^-(I) &= \bigcup_{J \text{ predecessor of } I} \mathbf{Varied}^+(J) \\ \mathbf{Useful}^+(I) &= \bigcup_{J \text{ successor of } I} \mathbf{Useful}^-(J) \end{aligned}$$

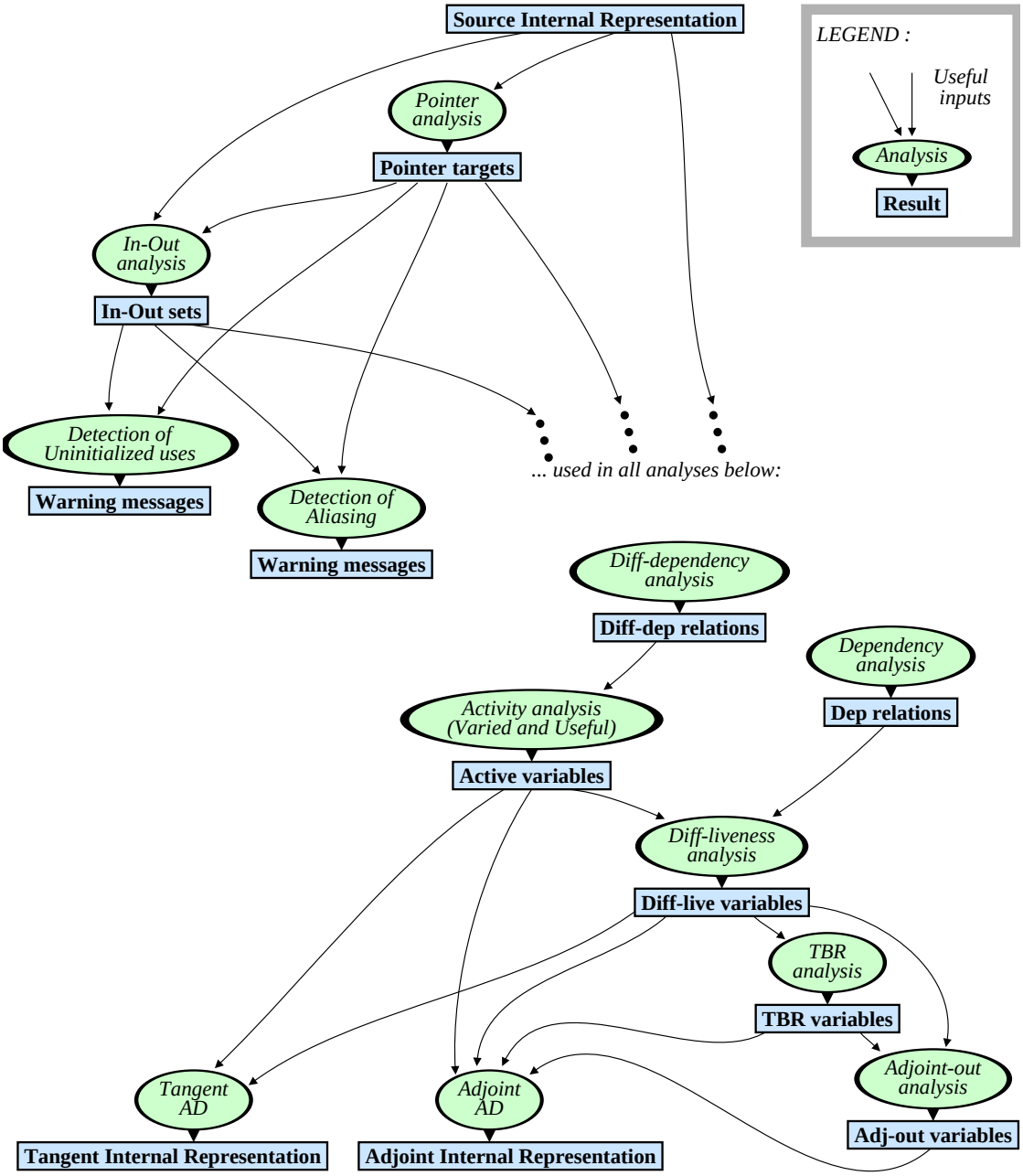


Figure 16: The chain of data-flow analyses implemented in Tapenade

The final activity is just the intersection:

$$\begin{aligned}\mathbf{Active}^-(I) &= \mathbf{Varied}^-(I) \cap \mathbf{Useful}^-(I)\\ \mathbf{Active}^+(I) &= \mathbf{Varied}^+(I) \cap \mathbf{Useful}^+(I)\end{aligned}$$

- **Diff-dependency analysis** is the preliminary computation, for activity analysis, of the diff-dependency **Diff-dep**(P) for each procedure P. If we define the composition of **Diff-dep** objects as:

$$(v1.v3) \in \mathbf{Diff\text{-}dep}(A) \otimes \mathbf{Diff\text{-}dep}(B) \iff \exists v2 | (v1.v2) \in \mathbf{Diff\text{-}dep}(A) \& (v2.v3) \in \mathbf{Diff\text{-}dep}(B)$$

  then we can compute **Diff-dep**(P) by a forward sweep on P, exploiting again the natural transitivity of dependency. Calling I_0 the entry instruction of P, I_∞ its exit instruction, and **Id** the identity dependence relation, the data-flow equations are:

$$\begin{aligned}\mathbf{Diff\text{-}dep}^+(I_0) &= \mathbf{Id}\\ \mathbf{Diff\text{-}dep}^+(I) &= \mathbf{Diff\text{-}dep}^-(I) \otimes \mathbf{Diff\text{-}dep}(I)\\ \mathbf{Diff\text{-}dep}^-(I) &= \bigcup_{J \text{ predecessor of } I} \mathbf{Diff\text{-}dep}^+(J)\end{aligned}$$

  and the result **Diff-dep**(P) is found in **Diff-dep**$^-(I_\infty)$.

- **Diff-liveness analysis** finds instructions of the original program that, although perfectly useful to compute its results, are not needed to compute its derivatives. If only the derivatives are of interest in the end, then these instructions can be removed from the differentiated code. This is therefore a special case of dead code detection. Diff-liveness analysis propagates, backwards on the flow graph, the **Diff-live** set of variables that are needed to ultimately compute a derivative. At the tail of the differentiated code, it is initialized to $\emptyset$. A variable v is diff-live before an instruction I either if v is used in the derivative instruction I', or if some variable that depends on v is diff-live after I. Hence the data-flow equations:

$$\begin{aligned}\mathbf{Diff\text{-}live}^-(I_\infty) &= \emptyset\\ \mathbf{Diff\text{-}live}^-(I) &= \mathbf{use}(I') \cup (\mathbf{Dep}(I) \otimes \mathbf{Diff\text{-}live}^+(I))\end{aligned}$$

  that use **Dep**(I), the dependency relation between variables before I and after I. Like **Diff-dep**(I), **Dep**(I) is precomputed for efficiency when I is a call to a procedure P. We just need to add the classical closure data-flow equation for a backward propagation:

$$\mathbf{Diff\text{-}live}^+(I) = \bigcup_{J \text{ successor of } I} \mathbf{Diff\text{-}live}^-(J)$$

  Diff-liveness must be extended to control: a statement that has a derivative or that computes a diff-live variable makes its surrounding control diff-live. A diff-live control makes all the variables it uses diff-live.

- **Dependency analysis** is the preliminary computation of the dependency between inputs and outputs of each procedure P. We need the same composition operation $\otimes$ as was needed for Diff-dependency analysis. Only the elementary dependencies through a simple instruction, e.g. an assignment, is different: the left-hand side of the assignment depends on all variables occurring in the right-hand-side, including indices, and also on the indices

occurring in the left-hand side itself. We can thus compute $\mathbf{Dep}(P)$ by a forward sweep on P, on the model of Diff-dependency analysis:

$$\begin{array}{lcl} \mathbf{Dep}^+(I_0) & = & \mathbf{Id} \\ \mathbf{Dep}^+(I) & = & \mathbf{Dep}^-(I) \otimes \mathbf{Dep}(I) \\ \mathbf{Dep}^-(I) & = & \bigcup_{J \text{ predecessor of } I} \mathbf{Dep}^+(J) \end{array}$$

and the result $\mathbf{Dep}(P)$ is found in $\mathbf{Dep}^-(I_\infty)$.

- **TBR analysis** is specific to the adjoint mode of AD when using the Store-All strategy. It finds variables whose present value is used in a derivative instruction. This property is dynamic: it is reset to false when the variable is completely overwritten with a new value. The goal of TBR analysis is to find locations where a variable is overwritten, and we know that the overwritten value is needed for a derivative instruction. At such a location, our adjoint AD model (*store-all*) will insert storage of the value before it gets overwritten in the forward sweep, and will restore it at the corresponding location in the backward sweep. TBR analysis propagates, forward on the flow graph, the **TBR** set of variables whose present value is (actually may be) used in a derivative instruction. Initially, this set is empty at the beginning of the code to differentiate. A variable v enters the **TBR** set after instruction I if v is used in the derivative instruction I', and is removed from this set if I is present in the differentiated program and v is certainly and completely overwritten by I ($v \in \mathbf{kill}(I)$). This refinement about I being present in the differentiated code comes directly from diff-liveness analysis:

$$\mathbf{Diff\text{-}present}(I) \Longleftrightarrow \mathbf{out}(I) \cap \mathbf{Diff\text{-}live}^+(I) \neq \emptyset$$

Hence the data-flow equations:

$$\begin{array}{lcl} \mathbf{TBR}^+(I) & = & \left\{ \begin{array}{ll} (\mathbf{TBR}^-(I) \cup \mathbf{use}(I')) \setminus \mathbf{kill}(I) & \text{if } \mathbf{Diff\text{-}present}(I), \\ \mathbf{TBR}^-(I) \cup \mathbf{use}(I') & \text{otherwise} \end{array} \right. \\ \mathbf{TBR}^-(I) & = & \bigcup_{J \text{ predecessor of } I} \mathbf{TBR}^+(J) \end{array}$$

- **Adjoint-out analysis** is specific to the adjoint mode of AD. It finds the variables whose value may be modified by the fragment of the adjoint code that consists of the sequel of the code to differentiate (i.e. the sequel of the forward sweep) followed by its adjoint (i.e. its backward sweep). In other words, seeing the code to differentiate `P` as the sequence `P=U;`I`;D` around instruction I, and recalling that its standard two-sweep adjoint is:

$$\overline{\mathtt{P}} = \overrightarrow{\mathtt{U}}; \overrightarrow{I}; \overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}; \overleftarrow{I}; \overleftarrow{\mathtt{U}} \quad ,$$

then $\mathbf{Adj\text{-}out}^+(I)$ contains the variables maybe modified by $\overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}$, and $\mathbf{Adj\text{-}out}^-(I)$ those modified by $\overrightarrow{I}; \overrightarrow{\mathtt{D}}; \overleftarrow{\mathtt{D}}; \overleftarrow{I}$. Adjoint-out analysis propagates the **Adj-out** sets backwards, with an initial value $\emptyset$ at the tail of `P`. A variable v enters the **Adj-out** before instruction I if it is overwritten by the I that is contained in $\overrightarrow{I}$ (when $\mathbf{Diff\text{-}present}(I)$). $\overleftarrow{I}$ only overwrites derivative variables, and modifies no original variables. However, the possible store/restore mechanism controlled by the TBR analysis plays a role: if the value of a variable v is in $\mathbf{TBR}^-(I)$, or will enter the **TBR** set because I' uses it, then the Store-All strategy will make sure that whatever overwriting is done to v in $\overrightarrow{I}; \overrightarrow{\mathtt{D}}$ it will be undone in $\overleftarrow{\mathtt{D}}; \overleftarrow{I}$. Therefore v can be removed from $\mathbf{Adj\text{-}out}^-(I)$. Hence the data-flow equations:

$$
\begin{array}{lcl}
\mathbf{Adj\text{-}out}^-(I_\infty) & = & \emptyset \\
\mathbf{Adj\text{-}out}^-(I) & = & \left\{ \begin{array}{ll} (\mathbf{Adj\text{-}out}^+(I) \cup \mathbf{out} & (I)) \setminus (\mathbf{TBR}^-(I) \cup \mathbf{use}(I')) \\ & \text{if } \mathbf{Diff\text{-}present}(I), \\ \mathbf{Adj\text{-}out}^+(I) & \text{otherwise} \end{array} \right. \\
\mathbf{Adj\text{-}out}^+(I) & = & \bigcup_{J \text{ successor of } I} \mathbf{Adj\text{-}out}^-(J)
\end{array}
$$

We encounter a difficulty here: it turns out that the recomputation mechanism of section 4.9 uses the **Adj-out** sets, and modifies the **TBR** sets, thus causing a dependence cycle between the analyses. We will not detail the way we break this cycle, but we hope the present formalization can help us find a more accurate way to do so.

5.3 Inference Rules for Tangent differentiation

To specify actual differentiation of a procedure, we will focus on the case of well-structured code, i.e. code where the control structure is apparent and embodied in the abstract syntax tree. This case is general enough, especially for industry codes that respect strict coding rules. For well-structured code, the differentiation algorithms can be specified in the formalism of Natural Semantics [13], a branch of Operational Semantics [19] that uses inference rules to describe program transformation. The Natural Semantics rules that follow capture almost all the refinements presently implemented in Tapenade, except for a few optimizations that require a global vision of the code, discussed in section 5.5. In particular they describe exactly for both tangent and adjoint mode,

- how activity analysis benefits to actual differentiation,
- how the slicing resulting from diff-liveness analysis is done.

The algorithm described by these rules can be readily implemented in any programming language. Implementation is almost immediate in logical languages such as PROLOG. This yields a complete and faithful view of the actual implementation of tangent and adjoint AD in Tapenade, missing only a few global code optimization.

We will use a classical syntax for the inference rules: a possibly empty list of hypothesis predicates, on top of a fraction bar, on top of a conclusion predicate. The first hypothesis predicates can be boolean conditions that restrict applicability of the rule. The other hypotheses and the conclusion are rewrite predicates that turn a source tree pattern into a rewritten one. The source pattern is connected to the rewritten pattern by a $\rightarrow$ sign that can bear the name of the predicate. The predicate with no name can be read as "is differentiated as". Tree patterns may have tree variables at their leaves, displayed in upper-case *ITALICS*, that get instantiated by the corresponding sub-tree by matching, also called unification. Apart from tree variables, tree patterns are IL trees. We display them here with a concrete syntax, akin to a classical imperative language, and with a `typewriter font`. The inference rules we give are complete and mutually exclusive: one and only one rule is applicable to any given abstract syntax tree, so that no backtracking is needed during the tree transformation process.

The boolean conditions that appear in the rules only depend on the result of the static data-flow analysis. By default, they are checked on the source tree to be matched, and otherwise the boolean condition specifies as its argument the tree to be tested.

- *isActiveExpr*(*EXPR*) is true if *EXPR* is (or uses in a differentiable way) a reference to a variable that belongs to $\mathbf{Active}^-$.

- *isActiveCall*(*CALL*) is true if one argument of this *CALL* is (or uses in a differentiable way) a reference to a variable that belongs to **Active**$^-$ or to **Active**$^+$.
- *isLiveForDiff*(*I*) is true if the instruction *I* produces a value that is diff-live, i.e. useful for the derivatives, i.e. $\mathbf{out}(I) \cap \mathbf{Diff\text{-}live}^+(I) \neq \emptyset$.
- *isRef*(*EXPR*) is true if *EXPR* is a memory reference, e.g. `x` or `A[i]` but not `x*2.0`.
- *isDiffProc*(*P*) is true if *isActiveCall* is true for at least one call to *P*.
- *isDiffFormalArg*(*ARG*) is true if at least one call to the current procedure feeds into this formal argument an actual argument that belongs to **Active**$^-$ or to **Active**$^+$.
- *isDiff*(*REF*) is true if memory reference *REF* refers to a variable that is **Active** at some location in the program.

The rules also use a few primitives to create new symbol names, with the guarantee that the returned name is not used in the original program.

- *varName* builds a name for the differentiated counterpart of a variable, generally appending "`d`" (tangent/dot) or "`b`" (adjoint/bar) to the original variable name.
- *procName* builds a name for the differentiated counterpart of a procedure, generally appending "`_D`" or "`_B`" to the original procedure name.
- *newFreeVar* builds a new name for a new variable of given type.

The differentiated program must contain declarations for these new symbols. The inference rules deal with that through the rewrite predicates

$$DECLS \rightarrow \overset{\bullet}{DECLS} \qquad \text{and} \qquad DECLS \rightarrow \overline{DECLS}$$

but we will not detail the rules that define them.

In tangent mode, a procedure is differentiated if at least one call to it is active. The envelope of all active calls determine which formal arguments are differentiated. This results in a single differentiated procedure. The corresponding inference rules are:

$$\frac{\mathit{isDiffProc}(P) \qquad P \xrightarrow{procName} \overset{\bullet}{P} \qquad ARGS \rightarrow \overset{\bullet}{ARGS} \qquad INSTRS \rightarrow \overset{\bullet}{INSTRS} \qquad DECLS \rightarrow \overset{\bullet}{DECLS}}{\texttt{procedure}\ P\ \texttt{(}\ ARGS\ \texttt{)}\ \texttt{\{}DECLS\texttt{;}\ INSTRS\texttt{\}} \rightarrow \texttt{procedure}\ \overset{\bullet}{P}\texttt{(}\overset{\bullet}{ARGS}\texttt{)}\ \texttt{\{}\overset{\bullet}{DECLS}\texttt{;}\ \overset{\bullet}{INSTRS}\texttt{\}}}$$

$$\frac{\mathit{isDiffFormalArg}(ARG) \qquad ARG \xrightarrow{varName} \overset{\bullet}{ARG} \qquad ARGS \rightarrow \overset{\bullet}{ARGS}}{\texttt{(}ARG\ \texttt{.}\ ARGS\texttt{)} \rightarrow \texttt{(}ARG\ \texttt{,}\ \overset{\bullet}{ARG}\ \texttt{.}\ \overset{\bullet}{ARGS}\texttt{)}}$$

$$\frac{!\mathit{isDiffFormalArg}(ARG) \qquad ARGS \rightarrow \overset{\bullet}{ARGS}}{\texttt{(}ARG\ \texttt{.}\ ARGS\texttt{)} \rightarrow \texttt{(}ARG\ \texttt{.}\ \overset{\bullet}{ARGS}\texttt{)}}$$

$$\frac{}{\texttt{()} \rightarrow \texttt{()}}$$

The rules for the sequence reflect that tangent instructions follow the order of original instructions:

$$\frac{INSTR \rightarrow \overset{\bullet}{INSTR} \qquad INSTRS \rightarrow \overset{\bullet}{INSTRS}}{\{INSTR\ ;\ INSTRS\} \rightarrow \{\overset{\bullet}{INSTR}\ ;\ \overset{\bullet}{INSTRS}\}}$$

$$\frac{}{\{\} \rightarrow \{\}}$$

Tangent differentiation is pushed deep down the control structures. There are special optimized cases when the control structure is not diff-live. Note that when the control structure is active, i.e. contains at least one active instruction, then it is labelled diff-live too. For instance for conditionals and loops:

$$\frac{isLiveForDiff() \qquad PART1 \rightarrow \overset{\bullet}{PART1} \qquad PART2 \rightarrow \overset{\bullet}{PART2}}{\texttt{if}\ (TEST)\ PART1\ \texttt{else}\ PART2 \rightarrow \texttt{if}\ (TEST)\ \overset{\bullet}{PART1}\ \texttt{else}\ \overset{\bullet}{PART2}}$$

$$\frac{!isLiveForDiff()}{\texttt{if}\ (TEST)\ PART1\ \texttt{else}\ PART2 \rightarrow \{\}}$$

$$\frac{isLiveForDiff() \qquad INSTRS \rightarrow \overset{\bullet}{INSTRS}}{\texttt{loop(}CONTROL\texttt{)}\ INSTRS \rightarrow \texttt{loop(}CONTROL\texttt{)}\ \overset{\bullet}{INSTRS}}$$

$$\frac{!isLiveForDiff()}{\texttt{loop(}CONTROL\texttt{)}\ INSTRS \rightarrow \{\}}$$

The exemplary atomic instruction is the assignment. Four cases must be considered, depending on the assignment being diff-live or not, and on the left-hand side reference having a derivative or not:

$$\frac{isLiveForDiff() \qquad isDiff(REF) \qquad REF \xrightarrow{ref} \overset{\bullet}{REF} \qquad EXPR \xrightarrow{expr} \overset{\bullet}{EXPR}}{REF\texttt{:=}EXPR \rightarrow \{\overset{\bullet}{REF}\texttt{:=}\overset{\bullet}{EXPR}\ ;\ REF\texttt{:=}EXPR\}}$$

$$\frac{isLiveForDiff() \qquad !isDiff(REF)}{REF\texttt{:=}EXPR \rightarrow REF\texttt{:=}EXPR}$$

$$\frac{!isLiveForDiff() \qquad isDiff(REF) \qquad REF \xrightarrow{ref} \overset{\bullet}{REF} \qquad EXPR \xrightarrow{expr} \overset{\bullet}{EXPR}}{REF\texttt{:=}EXPR \rightarrow \overset{\bullet}{REF}\texttt{:=}\overset{\bullet}{EXPR}}$$

$$\frac{!isLiveForDiff() \qquad !isDiff(REF)}{REF\texttt{:=}EXPR \rightarrow \{\}}$$

A procedure call is in general just replaced by a call to the differentiated procedure, which contains both the original statements of the procedure and their differentiated counterparts. There are specialized rules whether the particular call is active or not, or diff-live or not. As there is only one differentiated procedure for possibly many call activity patterns, the activity of the actual argument may differ from that of the formal argument and this requires dummy differentiated arguments to be initialized and passed:

$$\frac{isActiveCall() \qquad P \xrightarrow{procName} \overset{\bullet}{P} \qquad ARGS \xrightarrow{actualArgs} INITS, \overset{\bullet}{ARGS}, POSTS}{\texttt{call}\ P\texttt{(}ARGS\texttt{)} \rightarrow \{INITS;\ \texttt{call}\ \overset{\bullet}{P}\texttt{(}\overset{\bullet}{ARGS}\texttt{)}\ ;\ POSTS\}}$$

$$\frac{!isActiveCall() \qquad isLiveForDiff()}{\texttt{call}\ P\texttt{(}ARGS\texttt{)} \rightarrow \texttt{call}\ P\texttt{(}ARGS\texttt{)}}$$

$$\frac{!isActiveCall() \qquad !isLiveForDiff()}{\texttt{call } P(ARGS) \rightarrow \{\}}$$

$$\frac{isDiffFormalArg(ARG) \qquad ARG \xrightarrow{actualArg} INIT, \dot{ARG}, POST \qquad ARGS \xrightarrow{actualArgs} INITS, \dot{ARGS}, POSTS}{(ARG.ARGS) \xrightarrow{actualArgs} \{INIT; INITS\}, (ARG, \dot{ARG}.\dot{ARGS}), \{POST; POSTS\}}$$

$$\frac{!isDiffFormalArg(ARG) \qquad ARGS \xrightarrow{actualArgs} INITS, \dot{ARGS}, POSTS}{(ARG.ARGS) \xrightarrow{actualArgs} INITS, (ARG.\dot{ARGS}), POSTS}$$

$$\frac{}{() \xrightarrow{actualArgs} \{\}, (), \{\}}$$

$$\frac{isRef() \qquad isDiff() \qquad EXPR \xrightarrow{ref} \dot{EXPR}}{EXPR \xrightarrow{actualArg} \{\}, \dot{EXPR}, \{\}}$$

$$\frac{!(isRef() \quad isDiff()) \qquad typeOf(EXPR) \xrightarrow{newFreeVar} V \qquad V \xrightarrow{varName} \dot{V} \qquad EXPR \xrightarrow{expr} \dot{EXPR}}{EXPR \xrightarrow{actualArg} \{\dot{V}:=\dot{EXPR}\}, \dot{V}, \{\}}$$

Finally, tangent differentiation of a single expression applies the usual rules of calculus. For simplicity, only the rule for the product is presented here. Activity analysis allows us to define an improved rule that replaces the derivative of a passive expression with zero. This opens the way for further simplifications of the differentiated expressions. We will not detail this (optional) simplification done by predicate $\xrightarrow{simplify}$.

$$\frac{isActiveExpr() \qquad EXPR \xrightarrow{activeExpr} \dot{EXPR} \qquad \dot{EXPR} \xrightarrow{simplify} \dot{SIMPLEXPR}}{EXPR \xrightarrow{expr} \dot{SIMPLEXPR}}$$

$$\frac{!isActiveExpr()}{EXPR \xrightarrow{expr} 0.0}$$

$$\frac{EXPR1 \xrightarrow{expr} \dot{EXPR1} \qquad EXPR2 \xrightarrow{expr} \dot{EXPR2}}{EXPR1*EXPR2 \xrightarrow{activeExpr} EXPR1*\dot{EXPR2} + \dot{EXPR1}*EXPR2}$$

$$\frac{isRef() \qquad REF \xrightarrow{ref} \dot{REF}}{REF \xrightarrow{activeExpr} \dot{REF}}$$

$$\frac{REF \xrightarrow{ref} \dot{REF}}{REF(INDEX) \xrightarrow{ref} \dot{REF}(INDEX)}$$

$$\frac{NAME \xrightarrow{varName} \dot{NAME}}{NAME \xrightarrow{ref} \dot{NAME}}$$

This completes the inference rules for the tangent mode of AD.

5.4 Inference Rules for Adjoint differentiation

The rules that we are going to list express all the adjoint differentiation model, except for a few global manipulations that require a global vision of the code (*cf* section 5.5). They express in particular

- how TBR analysis is used to reduce the memory storage,
- the exact implementation of user control on coarse-grain checkpointing, and of the other specific improvements described in section 4.10.

The rules can also capture a few other technical points that we decided to skip for clarity. These are the treatment of pointers and memory allocation, call-by-value arguments, and the recomputation of cheap intermediate variables described in section 4.9. Also, the general mechanism used by the inference rules on assignements to cope with possible aliasing is too systematic and is actually applied only when needed.

In addition to the boolean conditions and utility primitives already defined for the tangent mode, the inference rules for the adjoint mode use new conditions related to checkpointing:

- *isJoint*(*CALL*) is true if the choice for this *CALL* is *joint* differentiation i.e., to apply checkpointing to this *CALL*
- *isCalledJoint*(*P*) is true if at least one call to this procedure *P* was selected for *joint* differentiation.
- *isCalledSplit*(*P*) is true if at least one call to this procedure *P* was not selected for *joint* differentiation.

We also use two new primitives to create procedure names for the split mode:

- *procNameFwd* builds a name for the split forward sweep of the adjoint of a procedure, generally appending "`_FWD`" to the original procedure name.
- *procNameBwd* builds a name for the split backward sweep of the adjoint of a procedure, generally appending "`_BWD`" to the original procedure name.

Checkpointing a piece of code *INSTRS* opens the way for simplification (*cf* section 4.9) because the forward sweep is immediately followed by the backward sweep. This requires that the data-flow analyses influenced by checkpointing (diff-liveness and TBR) be run again on the checkpointed program piece before it is differentiated. After differentiation of the checkpointed piece, the analyses results are reset to their previous values. This mechanism is expressed in the rules by "**With** *CKPdataFlow(INSTRS)* **:**"

In adjoint mode, a procedure can be differentiated in one of two ways (or both) depending on the checkpointing choice made by the end-user on each call. By default checkpointing is applied to the call, which implies building one differentiated procedure which calls the forward sweep followed by the backward sweep of adjoint AD. This is also known as the *joint mode*. The end-user can also choose not to apply checkpointing on some calls, which implies building two differentiated procedures, one for the forward sweep and one for the backward sweep. This is also known as the *split mode*. It may happen that a procedure must be differentiated both in joint and split mode, which amounts to a total of three differentiated procedures. The corresponding inference rules are:

$$\frac{\begin{array}{c}\mathit{isDiffProc}(P) \qquad \mathit{isCalledSplit}(P) \qquad P \xrightarrow{procNameFwd} \overrightarrow{P} \qquad P \xrightarrow{procNameBwd} \overleftarrow{P} \\ ARGS \to \overline{ARGS} \qquad INSTRS \to \begin{bmatrix} \overrightarrow{INSTRS} \\ \overleftarrow{INSTRS} \end{bmatrix} \qquad DECLS \to \overline{DECLS}\end{array}}{\begin{array}{l}\texttt{procedure}\ P\texttt{(}ARGS\texttt{)}\ \texttt{\{}DECLS\texttt{;}\ INSTRS\texttt{\}} \\ \to \texttt{procedure}\ \overrightarrow{P}\texttt{(}ARGS\texttt{)}\ \texttt{\{}DECLS\texttt{;}\ \overrightarrow{INSTRS}\texttt{\}} \quad \texttt{procedure}\ \overleftarrow{P}\texttt{(}\overline{ARGS}\texttt{)}\ \texttt{\{}\overline{DECLS}\texttt{;}\ \overleftarrow{INSTRS}\texttt{\}}\end{array}}$$

$$\frac{\begin{array}{c}ARGS \to \overline{ARGS} \qquad \left\{ \begin{array}{r} \mathit{isDiffProc}(P) \qquad \mathit{isCalledJoint}(P) \qquad P \xrightarrow{procName} \overline{P} \\ \textbf{With}\ \mathit{CKPdataFlow}(INSTRS)\ \textbf{:} \\ INSTRS \to \begin{bmatrix} \overrightarrow{INSTRS} \\ \overleftarrow{INSTRS} \end{bmatrix} \end{array} \right\} \qquad DECLS \to \overline{DECLS}\end{array}}{\begin{array}{l}\texttt{procedure}\ P\texttt{(}ARGS\texttt{)\{}DECLS\texttt{;}\ INSTRS\texttt{\}} \\ \to \texttt{procedure}\ \overline{P}\texttt{(}\overline{ARGS}\texttt{)}\ \texttt{\{}\overline{DECLS}\texttt{;}\ \overrightarrow{INSTRS}\texttt{;}\ \overleftarrow{INSTRS}\texttt{\}}\end{array}}$$

$$\frac{\mathit{isDiffFormalArg}(ARG) \qquad ARG \xrightarrow{varName} \overline{ARG} \qquad ARGS \to \overline{ARGS}}{\texttt{(}ARG\ \texttt{.}\ ARGS\texttt{)} \to \texttt{(}ARG\ \texttt{,}\ \overline{ARG}\ \texttt{.}\ \overline{ARGS}\texttt{)}}$$

$$\frac{!\mathit{isDiffFormalArg}(ARG) \qquad ARGS \to \overline{ARGS}}{\texttt{(}ARG\ \texttt{.}\ ARGS\texttt{)} \to \texttt{(}ARG\ \texttt{.}\ \overline{ARGS}\texttt{)}}$$

$$\frac{}{\texttt{()} \to \texttt{()}}$$

One specificity of the adjoint AD model is the two-sweeps structure, forward then backward, of the adjoint code. The rules for the sequence and for the control structures reflect that by producing the pair of the forward and backward code:

$$\frac{INSTR \to \begin{bmatrix} \overrightarrow{INSTR} \\ \overleftarrow{INSTR} \end{bmatrix} \qquad INSTRS \to \begin{bmatrix} \overrightarrow{INSTRS} \\ \overleftarrow{INSTRS} \end{bmatrix}}{\texttt{\{}INSTR\ \texttt{;}\ INSTRS\texttt{\}} \to \begin{bmatrix} \texttt{\{}\overrightarrow{INSTR}\ \texttt{;}\ \overrightarrow{INSTRS}\texttt{\}} \\ \texttt{\{}\overleftarrow{INSTRS}\ \texttt{;}\ \overleftarrow{INSTR}\texttt{\}} \end{bmatrix}}$$

$$\frac{}{\texttt{\{\}} \to \begin{bmatrix} \texttt{\{\}} \\ \texttt{\{\}} \end{bmatrix}}$$

$$\frac{\mathit{isLiveForDiff}() \qquad boolean \xrightarrow{newFreeVar} BRANCH \qquad PART1 \to \begin{bmatrix} \overrightarrow{PART1} \\ \overleftarrow{PART1} \end{bmatrix} \qquad PART2 \to \begin{bmatrix} \overrightarrow{PART2} \\ \overleftarrow{PART2} \end{bmatrix}}{\begin{array}{l}\texttt{if (}TEST\texttt{)}\ PART1\ \texttt{else}\ PART2 \\ \to \begin{bmatrix} \texttt{if (}TEST\texttt{)}\ \texttt{\{}\overrightarrow{PART1}\texttt{;}\ \texttt{push(true)\}}\ \texttt{else}\ \texttt{\{}\ \overrightarrow{PART2}\texttt{;}\ \texttt{push(false)\}} \\ \texttt{pop(}BRANCH\texttt{);}\ \texttt{if (}BRANCH\texttt{)}\ \overleftarrow{PART1}\ \texttt{else}\ \overleftarrow{PART2} \end{bmatrix}\end{array}}$$

$$\frac{!\mathit{isLiveForDiff}()}{\texttt{if (}TEST\texttt{)}\ PART1\ \texttt{else}\ PART2 \to \begin{bmatrix} \texttt{\{\}} \\ \texttt{\{\}} \end{bmatrix}}$$

$$\frac{\mathit{isLiveForDiff}()\qquad \mathit{integer} \xrightarrow{\mathit{newFreeVars}} VF,\ VT,\ VS\qquad INSTRS \rightarrow \left[\begin{array}{l}\overrightarrow{INSTRS}\\ \overleftarrow{INSTRS}\end{array}\right.}{\begin{array}{l}\texttt{loop(}I\texttt{=}F,T,S\texttt{)}\ INSTRS\\ \rightarrow \left[\begin{array}{l}\texttt{push(}I\texttt{);}\ \ VF\texttt{:=}F\texttt{;}\ \ VS\texttt{:=}S\texttt{;}\ \ \texttt{loop(}I\texttt{=}VF,T,VS\texttt{)}\,\overrightarrow{INSTRS}\texttt{;}\ \ \texttt{push(}I\texttt{-}VS,VF,\texttt{-}VS\texttt{)}\\ \texttt{pop(}VF,VT,VS\texttt{);}\ \ \texttt{loop(}I\texttt{=}VF,VT,VS\texttt{)}\,\overleftarrow{INSTRS}\texttt{;}\ \ \texttt{pop(}I\texttt{)}\end{array}\right.\end{array}}$$

$$\frac{!\mathit{isLiveForDiff}()}{\texttt{loop(}CONTROL\texttt{)}\ INSTRS \rightarrow \left[\begin{array}{l}\texttt{\{\}}\\ \texttt{\{\}}\end{array}\right.}$$

The rules for an assignment distinguish the same cases as for the tangent mode. Actual computation of derivatives is placed into the backward sweep. Saving and restoring of the intermediate values is done here, through a `push` and `pop` of the variables $RVARS$ overwritten by this assignment I that are needed in derivative expressions. Predicate $\xrightarrow{restore}$ computes $RVARS = \mathbf{out}(I) \cap (\mathbf{TBR}^{-}(I) \cup \mathbf{use}(I'))$. Notice also the systematic introduction of the temporary variable V, which is necessary to cope with aliasing in the adjoint mode. Consider the assignment `a(i) := 2*a(j)` and assume there is no static way to decide whether `i` equals `j` or not. Introducing V is the only way to create correct adjoint instructions. However we can spare introduction of V when the aliasing question between left- and right-hand sides of the assignment is decidable statically. The inference rules here do not show this improvement.

$$\frac{\begin{array}{c}\mathit{isLiveForDiff}()\qquad \mathit{isDiff}(REF)\qquad REF\texttt{:=}EXPR \xrightarrow{restore} RVARS\qquad REF \xrightarrow{ref} \overline{REF}\\ \mathit{typeOf}(REF) \xrightarrow{\mathit{newFreeVar}} V\qquad V \xrightarrow{\mathit{varName}} \overline{V}\qquad EXPR, \overline{V} \xrightarrow{expr} \overline{INSTRS}\end{array}}{REF\texttt{:=}EXPR \rightarrow \left[\begin{array}{l}\texttt{push(}RVARS\texttt{);}\ \ REF\texttt{:=}EXPR\\ \texttt{pop(}RVARS\texttt{);}\ \ \overline{V}\texttt{:=}\overline{REF}\texttt{;}\ \ \overline{REF}\texttt{:=0.0;}\ \ \overline{INSTRS}\end{array}\right.}$$

$$\frac{\mathit{isLiveForDiff}()\qquad !\mathit{isDiff}(REF)\qquad REF\texttt{:=}EXPR \xrightarrow{restore} RVARS}{REF\texttt{:=}EXPR \rightarrow \left[\begin{array}{l}\texttt{push(}RVARS\texttt{);}\ \ REF\texttt{:=}EXPR\\ \texttt{pop(}RVARS\texttt{);}\end{array}\right.}$$

$$\frac{\begin{array}{c}!\mathit{isLiveForDiff}()\qquad \mathit{isDiff}(REF)\qquad REF \xrightarrow{ref} \overline{REF}\\ \mathit{typeOf}(REF) \xrightarrow{\mathit{newFreeVar}} V\qquad V \xrightarrow{\mathit{varName}} \overline{V}\qquad EXPR, \overline{V} \xrightarrow{expr} \overline{INSTRS}\end{array}}{REF\texttt{:=}EXPR \rightarrow \left[\begin{array}{l}\texttt{\{\}}\\ \overline{V}\texttt{:=}\overline{REF}\texttt{;}\ \ \overline{REF}\texttt{:=0.0;}\ \ \overline{INSTRS}\end{array}\right.}$$

$$\frac{!\mathit{isLiveForDiff}()\qquad !\mathit{isDiff}(REF)}{REF\texttt{:=}EXPR \rightarrow \left[\begin{array}{l}\texttt{\{\}}\\ \texttt{\{\}}\end{array}\right.}$$

The rules for differentiating a procedure call are different whether a joint or split differentiation is chosen. As joint mode actually means checkpointing the procedure call, we must compute the snapshots, i.e. the set SNP of variables that must be restored in order to repeat execution of the call, and possibly another set SBK of variables that must be restored after execution of the adjoint call $\overline{P}$. There are many possible choices for SNP and SBK and no systematically better

choice. The choice made by Tapenade and returned by predicate $\xrightarrow{snapshot}$ is for any checkpointed piece of code C:

$$\begin{aligned} SNP &= \mathbf{out}(C) \cap \left(\mathbf{TBR}^-(C) \cup \mathbf{Diff\text{-}live}^-(C)\right) \\ SBK &= \emptyset \end{aligned}$$

The generated code uses procedure `look`, that restores values from the stack without actually popping them. In Joint mode, we can also make an extra simplification when the call is not diff-live:

$$\frac{\begin{array}{c}\mathit{isActiveCall}() \quad \mathit{isJoint}() \quad \mathit{isLiveForDiff}() \quad \texttt{call } P(ARGS) \xrightarrow{snapshot} SNP,SBK \\ P \xrightarrow{procName} \overline{P} \qquad ARGS \xrightarrow{actualArgs} INITS, \overline{ARGS}, POSTS\end{array}}{\begin{array}{l}\texttt{call } P(ARGS) \\ \rightarrow \left[\begin{array}{l}\texttt{push}(SBK)\texttt{; push}(SNP \backslash SBK)\texttt{; call } P(ARGS) \\ \texttt{pop}(SNP \backslash SBK)\texttt{; look}(SBK \cap SNP)\texttt{; } INITS\texttt{; call } \overline{P}(\overline{ARGS})\texttt{; } POSTS\texttt{; pop}(SBK)\end{array}\right.\end{array}}$$

$$\frac{\begin{array}{c}\mathit{isActiveCall}() \quad \mathit{isJoint}() \quad !\mathit{isLiveForDiff}() \quad \texttt{call } P(ARGS) \xrightarrow{snapshot} SNP,SBK \\ P \xrightarrow{procName} \overline{P} \qquad ARGS \xrightarrow{actualArgs} INITS, \overline{ARGS}, POSTS\end{array}}{\begin{array}{l}\texttt{call } P(ARGS) \\ \rightarrow \left[\begin{array}{l}\texttt{push}(SBK)\texttt{; push}(SNP \backslash SBK) \\ \texttt{pop}(SNP \backslash SBK)\texttt{; look}(SBK \cap SNP)\texttt{; } INITS\texttt{; call } \overline{P}(\overline{ARGS})\texttt{; } POSTS\texttt{; pop}(SBK)\end{array}\right.\end{array}}$$

$$\frac{\begin{array}{c} \qquad\qquad\qquad\qquad \mathit{isActiveCall}() \quad !\mathit{isJoint}() \\ P \xrightarrow{procNameFwd} \overrightarrow{P} \qquad P \xrightarrow{procNameBwd} \overleftarrow{P} \qquad ARGS \xrightarrow{actualArgs} INITS, \overline{ARGS}, POSTS\end{array}}{\texttt{call } P(ARGS) \rightarrow \left[\begin{array}{l}\texttt{call } \overrightarrow{P}(ARGS) \\ INITS\texttt{; call } \overleftarrow{P}(\overline{ARGS})\texttt{; } POSTS\end{array}\right.}$$

$$\frac{!\mathit{isActiveCall}() \quad \mathit{isLiveForDiff}() \quad \texttt{call } P(ARGS) \xrightarrow{restore} RVARS}{\texttt{call } P(ARGS) \rightarrow \left[\begin{array}{l}\texttt{push}(RVARS)\texttt{; call } P(ARGS) \\ \texttt{pop}(RVARS)\end{array}\right.}$$

$$\frac{!\mathit{isActiveCall}() \quad !\mathit{isLiveForDiff}()}{\texttt{call } P(ARGS) \rightarrow \left[\begin{array}{l}\{\} \\ \{\}\end{array}\right.}$$

The next rule implements the possibility offered to the end-user to require application of checkpointing to any well-structured piece of source code:

$$\frac{INSTRS \xrightarrow{snapshot} SNP,SBK \qquad \left\{\begin{array}{r}\textbf{With } \mathit{CKPdataFlow}(INSTRS)\textbf{ :} \\ INSTRS \rightarrow \left[\begin{array}{l}\overrightarrow{INSTRS} \\ \overleftarrow{INSTRS}\end{array}\right.\end{array}\right\}}{\begin{array}{l}\texttt{checkpoint}(INSTRS) \\ \rightarrow \left[\begin{array}{l}\texttt{push}(SBK)\texttt{; push}(SNP \backslash SBK)\texttt{; } INSTRS \\ \texttt{pop}(SNP \backslash SBK)\texttt{; look}(SBK \cap SNP)\texttt{; } \overrightarrow{INSTRS}\texttt{; } \overleftarrow{INSTRS}\texttt{; pop}(SBK)\end{array}\right.\end{array}}$$

The rules for adjoint differentiation of the actual arguments of a procedure call are very similar to the tangent case. However, the initializations *INITS* before the call and the trailing `POSTS` instructions are different:

$$\frac{isDiffFormalArg(ARG) \qquad ARG \xrightarrow{actualArg} INIT, \overline{ARG}, POST \qquad ARGS \xrightarrow{actualArgs} INITS, \overline{ARGS}, POSTS}{(ARG.ARGS) \xrightarrow{actualArgs} \{INIT; INITS\}, (ARG, \overline{ARG}.\overline{ARGS}), \{POST; POSTS\}}$$

$$\frac{!isDiffFormalArg(ARG) \qquad ARGS \xrightarrow{actualArgs} INITS, \overline{ARGS}, POSTS}{(ARG.ARGS) \xrightarrow{actualArgs} INITS, (ARG.\overline{ARGS}), POSTS}$$

$$\frac{}{() \xrightarrow{actualArgs} \{\}, (), \{\}}$$

$$\frac{isRef() \qquad isDiff() \qquad EXPR \xrightarrow{ref} \overline{EXPR}}{EXPR \xrightarrow{actualArg} \{\}, \overline{EXPR}, \{\}}$$

$$\frac{!(isRef() \quad isDiff()) \qquad typeOf(EXPR) \xrightarrow{newFreeVar} V \qquad V \xrightarrow{varName} \overline{V} \qquad EXPR, \overline{V} \xrightarrow{expr} POSTS}{EXPR \xrightarrow{actualArg} \{\overline{V} := 0.0\}, \overline{V}, POSTS}$$

Finally, adjoint differentiation of a single expression catches the specificity of adjoint differentiation: an adjoint expression $\overline{VAL}$ is propagated through the expression *EXPR* down to its leaves (i.e. references to an active variable), updating the adjoint expression each time it goes down into an arithmetic operator (only the rule for the product is presented here). For each active leaf, a new assignment statement is produced that increments the adjoint variable of this leaf with the adjoint expression.

$$\frac{isActiveExpr() \qquad EXPR, \overline{VAL} \xrightarrow{activeExpr} INSTRS}{EXPR, \overline{VAL} \xrightarrow{expr} INSTRS}$$

$$\frac{!isActiveExpr()}{EXPR, \overline{VAL} \xrightarrow{expr} \{\}}$$

$$\frac{EXPR1, EXPR2 * \overline{VAL} \xrightarrow{expr} INSTRS1 \qquad EXPR2, EXPR1 * \overline{VAL} \xrightarrow{expr} INSTRS2}{EXPR1 * EXPR2, \overline{VAL} \xrightarrow{activeExpr} \{INSTRS1; INSTRS2\}}$$

$$\frac{isRef(REF) \qquad REF \xrightarrow{ref} \overline{REF}}{REF, \overline{VAL} \xrightarrow{activeExpr} \{\overline{REF} := \overline{REF} + \overline{VAL}\}}$$

$$\frac{REF \xrightarrow{ref} \overline{REF}}{REF(INDEX) \xrightarrow{ref} \overline{REF}(INDEX)}$$

$$\frac{NAME \xrightarrow{varName} \overline{NAME}}{NAME \xrightarrow{ref} \overline{NAME}}$$

This completes the inference rules for the adjoint mode of AD.

5.5 Implementation notes for analyses and transformations

Tapenade runs data-flow analysis to the code's internal representation, then applies differentiation to it. This results into the internal representation of a new code.

As each data-flow analysis runs on a Call Graph whose nodes are actually Flow Graphs, the data-flow equations that we provided are implemented with a number of sweeps on these graphs. Depending on the equations, these sweeps can be top-down or bottom-up (on the Call Graph), and forward or backward (on the Flow Graphs). For instance activity analysis requires a top-down sweep on the Call Graph, which performs for each Flow Graph a forward sweep for the **Varied** information and a backward sweep for the **Useful** information. Because of cycles, these sweeps must iterate until reaching a fixed point. For efficiency, these iterations visit the graph nodes in a well-chosen order (e.g. *depth first spanning tree* for a forward sweep) and maintain a *wait list* of nodes that require a new iteration. Implementation-wise, this very uniform structure of data-flow analyses allows each of them to inherit from a parent Java class that defines these sweeps.

Differentiation itself is implemented at the upper level (Call Graph) as a quite simple graph transformation: activity analysis has labelled the procedures that must be differentiated, and differentiation amounts just to creating the new, differentiated, procedure(s). When the application language supports a notion of modules, then differentiated modules are created accordingly to contain the differentiated procedures and declarations, together with the original ones to cope with visibility restrictions. There are other minor subtleties, irrelevant here.

At the intermediate level, each procedure is indeed stored as a graph (Flow Graph) and differentiation is therefore a graph transformation. In particular, this enables Tapenade to equally differentiate structured and unstructured code. This implementation is thus slightly different from its inference rules specification of sections 5.3 and 5.4, but for well-structured code the result of the transformation is identical to the code built with the inference rules. The implemented algorithm essentially consists of two phases:

1. Build new, differentiated Basic Block(s) for each Basic Block of the original flow graph. At this deepest level of our program representation, which uses abstract syntax trees, the inference rules apply directly.

2. Connect the new Basic Blocks with Control Flow arrows according to the Control Flow arrows of the original flow graph. This is easy in tangent mode and in the forward sweep of the adjoint mode. In the backward sweep, the basic idea is to remember the control at each place where the flow merges in the forward sweep, and to use this memory to take the corresponding direction backwards in the backward sweep. This graph approach yields a better dead code elimination at the junction of forward and backward sweeps than can be expressed by the inference rules.

What Tapenade does that the inference rules do *not* express is the following non-local manipulations on the differentiated code:

- the reordering and the common subexpression elimination that take place in each differentiated expression.

- the additional slicing and simplifications that occur in adjoint mode at the junction of forward and backward sweeps.

- the implicit-zero mechanism that spares initialization of some derivatives, or at least postpones it till it is absolutely necessary

- in the adjoint mode, the reordering and fusion of increments to the same differentiated variable, sometimes known as incremental *vs* non-incremental adjoint mode
- in the backward sweep of the adjoint mode, the choice to recompute an intermediate variable instead of restoring it from the stack, as shown on the rightmost column of figure 11.

6 Development status and Performances

Development of Tapenade started in 1999 as a major redesign of the AD tool Odyssee [20]. In its present distributed version (v3.6, revision 4344), Tapenade source is mainly 138,000 lines of Java. The development environment also features 1,120 automated non-regression tests and 15 complete validation codes.

From the end-user point of view, Tapenade can be used as a web server at url `http://www-tapenade.inria.fr:8080`. Tapenade can also be installed locally on most architectures, which gives access to the full set of command-line options. All Tapenade documentation is available online with a user's guide, a tutorial, a FAQ, and downloading instructions. Users may subscribe to a dedicated mailing list. Downloading Tapenade is free for nonprofit academic research or education. Although still incomplete, the documentation describes the parameters of a differentiation request, the command-line options, and the AD directives that can be inserted in the source program. In addition to the differentiated code, Tapenade may return error and warning messages, discussed in the documentation.

Tapenade provides a special mode to validate its own output. This validation mode also provides support to locate possible bugs in the differentiated code. Validation of the tangent code is done by comparing the tangent derivatives with an approximation computed by divided differences. Validation of the adjoint code is done by computing a signature of the function's Jacobian matrix, using the tangent mode on one hand, and the adjoint mode on the other hand. Recall the notations of a program `P` that computes a function $F : X \in I\!R^n \mapsto Y \in I\!R^m$, call $F'(X)$ its Jacobian matrix at point X, $\dot{X}$ a column vector of $I\!R^n$ and $\overline{Y}$ a row vector of $I\!R^m$. Examples of possible signatures for comparison are:

- The "checksum", that computes $\overline{Y} \times F'(X) \times \dot{X}$ for any arbitrary $\dot{X}$ and $\overline{Y}$. By associativity, this can be computed as $\overline{Y} \times (F'(X) \times \dot{X})$ using the tangent code, or as $(\overline{Y} \times F'(X)) \times \dot{X}$ using the adjoint code.
- The "dot-product", that computes $\dot{X}^* \times F'(X)^* \times F'(X) \times \dot{X}$, where the * denotes transposition, for an arbitrary $\dot{X}$. This can be computed as the square norm of $\dot{Y} = F'(X) \times \dot{X}$, which is obtained using the tangent code, or by setting $\overline{Y} = \dot{Y}^*$, then computing $\overline{X} = \overline{Y} \times F'(X)$ using the adjoint mode, and finally computing the dot product $\dot{X}.\overline{X}$.

Derivatives are useful in most fields of Scientific Computing, and AD tools only provide a way to obtain them efficiently. In particular, gradients computed through the adjoint mode are one proeminent achievement of AD. For some classes of applications, there have been fruitful joint research between the Scientific Computing and AD specialists, to devise better computing algorithms that rely on specially tailored kinds of derivatives [5]. This meant joint progress for both Scientific Computing and AD. Still, derivatives are standalone mathematical objects, and the way they are used by scientific applications is disconnected in principle from the way an AD tool provides them. Therefore, from the AD tool point of view, it is not relevant here to go into the detail of particular Scientific Computing applications. To illustrate the performances of Tapenade, we have selected some of its applications that we are aware of. For each, after a brief description of the application, we will concentrate on the strictly AD-related performance

figures i.e., the accuracy of the derivatives and the time and memory costs of the differentiated program.

- **uns2D** is a 2D steady-state Navier-Stokes CFD simulation, on an unstructured mesh with 3000 triangles, and which on this application converges after 566 iterations then outputs a handful of aerodynamic characteristics (lift, drag...).
- **nsc2ke** is a small unsteady Euler CFD simulation, adapted for the needs of the car industry. On this particular simulation, the size of the unstructured mesh is 1516 triangles and the number of time steps is fixed to 80.
- **lidar** simulates the propagation of light across the atmosphere. As an inverse problem, it is used to estimate parameters of the atmosphere from measurements of the light received by a lidar.
- **nemo** is a large "configuration" of the OPA [16] ocean circulation model, here solving an inverse problem around the Antarctic. Mesh elements are 2^o wide, and this simulation runs for 400 time steps, i.e. about one month.
- **gyre** is a reduced configuration of the OPA model on a rectangular basin meshed with 21824 cells. The test computes the derivatives of a short simulation of 10 time steps.
- **winnie** is a simplified 2D version of the **grisli** model, that simulates the movement of polar ice caps. This particular unsteady simulation runs on a mesh with 250 elements, for 1000 time steps.
- **stics** [15] simulates the growth of a crop during one year (350 time steps) depending on climate and agricultural practices. The goal of this particular study was to evaluate the sensitivity of the biomass produced on the agricultural parameters.
- **smac-sail** [14] are two coupled atmosphere-surface radiative transfer models used here to estimate biophysical variables, from canopy reflectance data observed from a satellite. When used to minimize a discrepancy function, the code can be seen as having a scalar output. For other uses, it can have many outputs, like 7801 here.
- **traces** simulates the transport of pollutants through layers of soil and rock, carried by water. It was used here in an inverse problem fashion to estimate the permeability parameters of rock. Unfortunately the quantity of available measurements was very small in this application, so only 8 parameters could be estimated.

For each application, table 17 gives the application language, the size in **lines** of the source part that was submitted for differentiation, and its average run-time T. It gives also the dimensions n and m of the independent input space and dependent output space and the differentiation time T_{AD} to build the adjoint. The time to build the tangent code is a little shorter. For the tangent mode, the table gives the accuracy A_t of tangent derivatives tt with respect to divided differences dd , which basically measures the inaccuracy of divided differences. Actually $A_t = -\log_{10}(\frac{tt-dd}{dd})$ so that better accuracy means higher A_t. The table then shows the ratio R_t of the time to obtain one tangent derivative $\dot{Y}$ divided by the time T of the original code. For the adjoint mode, the table gives (the opposite $\log_{10}$ of) the accuracy A_a with respect to the tangent derivatives. This accuracy A_a should only reflect the different computation order of the derivatives and should therefore be high. The table then shows the ratio R_a of the time to obtain one gradient $\overline{X}$ divided by T. These time ratios along with n and m allow one to check the superiority of the adjoint mode for computing gradients. Specifically for the adjoint mode, we finally give the **peak** size of

	original code				tangent		adjoint			
	lines	T (s)	$n \rightarrow m$	T_{AD} (s)	A_t	R_t	A_a	R_a	**peak** (Mb)	**traffic** (Mb)
uns2d *(F77)*	2000	1.3	14000 → 3	6	3.4	**2.4**	15.1	**5.9**	**241**	1243
nsc2ke *(F77)*	3500	0.4	1602 → 5607	11	1.9	**2.4**	4.5	**16.2**	**168**	2806
lidar *(F90)*	330	3.3	37 → 37	2	6.7	**1.1**	14.4	**2.0**	**11**	11
nemo *(F90)*	55000	208	9100 → 1	95	3.0	**2.0**	8.1	**6.5**	**1591**	85203
gyre *(F90)*	21000	30	21824 → 1	26	4.5	**1.9**	13.3	**7.9**	**481**	48602
winnie *(F90)*	3700	0.6	3 → 1	8	1.4	**1.7**	13.7	**5.9**	**421**	614
stics *(F77)*	17000	0.2	739 → 1467	206	8.6	**2.4**	15.3	**3.9**	**155**	186
smac-sail*(F77)*	1700	2.2	1321 → 7801	6	5.9	**1.0**	10.5	**3.1**	**2**	21
traces *(F90)*	19800	13.2	8 → 1	53	4.0	**1.3**	12.9	**3.8**	**159**	4390

Figure 17: Performances of Tapenade derivative code on representative applications

the trajectory stack, and the total **traffic** of data pushed to and popped from this stack. Most of these applications make use of Tapenade directives on checkpointing to reduce these memory costs.

7 Outlook

We presented the specification of the Automatic Differentiation tool Tapenade. This specification is naturally split in two parts, one about the necessary static analysis of the provided source program, and one about its actual transformation into the derivative source program. Tapenade follows a model of AD that is only one in a panorama of possible models. We motivated why we chose this model, and illustrate it on examples.

Tapenade has been applied with success to several large codes, both academic and industrial, and its adjoint mode has often been considered valuable by end users, especially by those having written an adjoint code by hand in the past. Much work remains to make it even more widely applicable. In our long list of future developments, let us mention specific differentiation of well-known patterns such as solution of linear systems or fixed-point iterations. Also, we are considering a tool interface that could give freedom to differentiate an intermediate variable wrt another. We also want to give the option of association by address instead of by name, and evaluate the performance benefit brought by a better memory locality. Support for repeated differentiation must be improved, in order to compute second derivatives by Tangent-over-Adjoint differentiation. This last point is gaining importance with the growing concern for Uncertainty Quantification.

We identify three more ambitious developments, each of them meaning actual research on some aspects of the model that are still not fixed. Most of the difficulty lies with the adjoint mode. These are (1) Message-passing communication, (2) Dynamic memory management, and (3) Object-Oriented languages.

Message-passing communication can be theoretically transposed in the backward sweep of the adjoint. This exhibits very elegant duality between "sends" and "receives", which are adjoint of one another. Non-blocking (or asynchronous) communication such as an `mpi_isend`/`mpi_wait` pair, causes the backward sweep to contain an `mpi_irecv`/`mpi_wait` pair, in reverse order. There are a number of open issues on the best way to detect matching "sends" and "receives", as well as matching asynchronous communication pairs. These questions are not safely implemented in Tapenade yet.

Dynamic memory management causes the adjoint backward sweep to use memory that was deallocated by the forward sweep. One answer is to allocate some memory again, but this can return a different chunk of memory. Therefore, saved addresses are inconsistent with the new chunk of memory. We investigate two complementary answers. First, saved addresses can also save their offset with respect to the allocated base address, and one can recompute the new addresses from the saved offsets. This amounts to dynamically monitoring allocation and deallocation, which is quite general but costly. In favorable cases, we plan to mitigate with a special efficient strategy that recomputes memory addresses on the fly when possible.

Object-Oriented languages imply a deep change in the internal representation of programs. Because of the Object point of view, association by address seems more adapted for these languages. It also seems reasonable to first address Java rather than C++, although implicit deallocation by Garbage Collection makes it harder to locate the adjoint allocation during the backward sweep.

References

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

[2] Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and Jean Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008.

[3] H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, 2005.

[4] George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation: from Simulation to Optimization*. Computer and Information Science. Springer, New York, NY, 2001.

[5] A. Dervieux, L. Hascoët, M. Vazquez, and B. Koobus. Optimization loops for shape and error control. In B. Uthup, S.-P. Koruthu, R.-K. Sharma, and P. Priyadarshi, editors, *Recent Trends in Aerospace Design and Optimization*, pages 363–373. Tata-McGraw Hill, New Delhi, 2005. Post-SAROD-2005, Bangalore, India.

[6] 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.

[7] R. Giering. Tangent linear and Adjoint Model Compiler, Users manual. Technical report, 1997. http://www.autodiff.com/tamc.

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

[9] 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.

[10] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. pages 299–304 in [4]. 2002.

[11] L. Hascoët, U. Naumann, and V. Pascual. "To Be Recorded" analysis in reverse-mode Automatic Differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

[12] L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, 2004.

[13] G. Kahn. Natural Semantics. *Lecture Notes in Computer Science*, 247:22–39, 1987. Proceedings of STACS 1987, Passau, Germany.

[14] C. Lauvernet, F. Baret, L. Hascoët, S. Buis, and F.-X. LeDimet. Multitemporal-patch ensemble inversion of coupled surface-atmosphere radiative transfer models for land surface characterization. *Remote Sensing of Environment*, 112(3):851–861, 2008.

[15] C. Lauvernet, L. Hascoët, F.-X. Le Dimet, and F. Baret. Using Automatic Differentiation to study the sensitivity of a crop model. 2012. To appear, proceedings of AD2012, Fort Collins, Co.

[16] G. Madec, P. Delecluse, M. Imbard, and C. Levy. OPA8.1 ocean general circulation model reference manual. Technical report, Pole de Modelisation, IPSL, 1998.

[17] U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. pages 159–169 in [3]. 2005.

[18] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[19] G. Plotkin. A structural approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

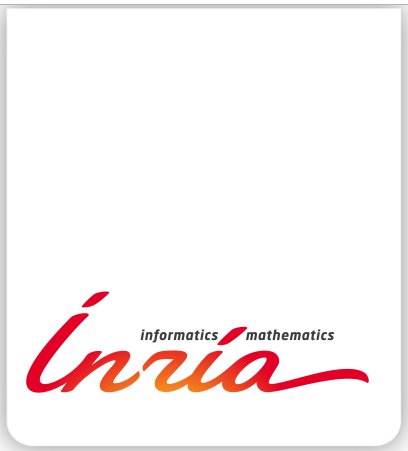
[20] N. Rostaing, S. Dalmas, and A. Galligo. Automatic Differentiation in Odyssée. *Tellus A*, 45(5):558–568, 1993.

[21] J. Shin, P. Malusare, and P. Hovland. Design and implementation of a Context-Sensitive, Flow-sensitive activity analysis algorithm for Automatic Differentiation. pages 115–125 in [2]. 2008.

[22] Jean Utke and Uwe Naumann. Separating language dependent and independent tasks for the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications (SEA 2004)*, pages 552–558, Anaheim, Calgary, Zurich, 2004. ACTA Press.

[23] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for Automatic Differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.

[24] 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