

Data-Flow Algorithms for Reverse Automatic Differentiation

Laurent Hascoët, Mauricio Araya-Polo

TROPICS Project, INRIA Sophia-Antipolis

ECCOMAS 2004, Jyväskylä, July 25-28, 2004

Principles of Reverse AD

AD rewrites **source programs** to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots; I_p; \}$ implementing $f : \mathbf{R}^m \rightarrow \mathbf{R}^n$
identify with: $f = f_p \circ f_{p-1} \circ \dots \circ f_1$
name: $x_0 = x$ and $x_k = f_k(x_{k-1})$

Principles of Reverse AD

AD rewrites **source programs** to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots; I_p; \}$ implementing $f : \mathbf{R}^m \rightarrow \mathbf{R}^n$
 identify with: $f = f_p \circ f_{p-1} \circ \dots \circ f_1$
 name: $x_0 = x$ and $x_k = f_k(x_{k-1})$
chain rule: $f'(x) = f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdot \dots \cdot f'_1(x_0)$

Principles of Reverse AD

AD rewrites **source programs** to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots I_p; \}$ implementing $f : \mathbf{R}^m \rightarrow \mathbf{R}^n$
 identify with: $f = f_p \circ f_{p-1} \circ \dots \circ f_1$
 name: $x_0 = x$ and $x_k = f_k(x_{k-1})$
chain rule: $f'(x) = f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdot \dots \cdot f'_1(x_0)$

$f'(x)$ generally too large and expensive \Rightarrow take useful views!

$$\begin{aligned} \dot{y} = f'(x) \cdot \dot{x} &= f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdot \dots \cdot f'_1(x_0) \cdot \dot{x} && \text{tangent AD} \\ \bar{x} = f'^*(x) \cdot \bar{y} &= f_1'^*(x_0) \cdot \dots \cdot f_{p-1}'^*(x_{p-2}) \cdot f_p'^*(x_{p-1}) \cdot \bar{y} && \text{reverse AD} \end{aligned}$$

Evaluate both **from right to left** !

Reverse AD

► There are plenty of advantages to reverse AD. Essentially, it returns gradients in just one run!

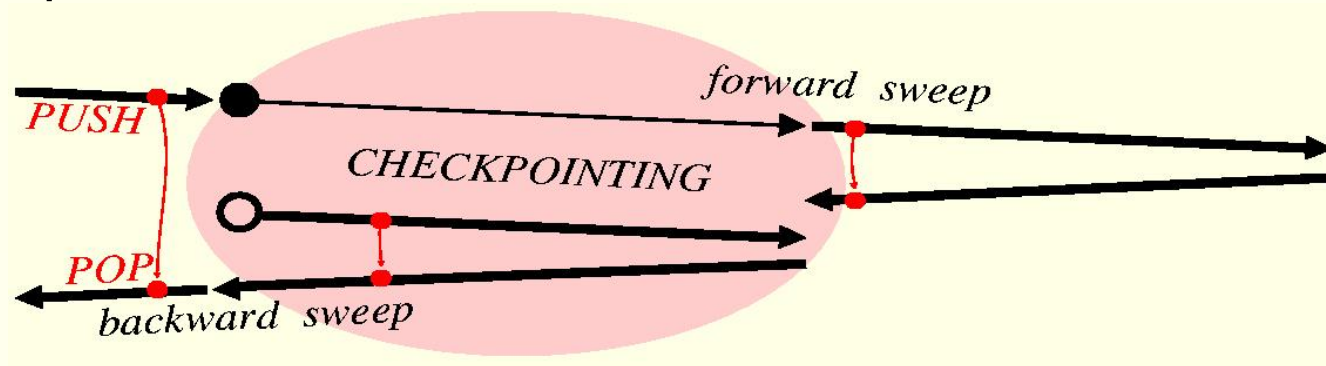
$$\bar{x} = f'^*(x) \cdot \bar{y} = f_1'^*(x_0) \cdot \dots \cdot f_{p-1}'^*(x_{p-2}) \cdot f_p'^*(x_{p-1}) \cdot \bar{y}$$

Reverse AD

- ▶ There are plenty of advantages to reverse AD. Essentially, it returns gradients in just one run!

$$\bar{x} = f'^*(x) \cdot \bar{y} = f_1'^*(x_0) \cdot \dots \cdot f_{p-1}'^*(x_{p-2}) \cdot f_p'^*(x_{p-1}) \cdot \bar{y}$$

- ▶ but it implies a structure that has drawbacks too...

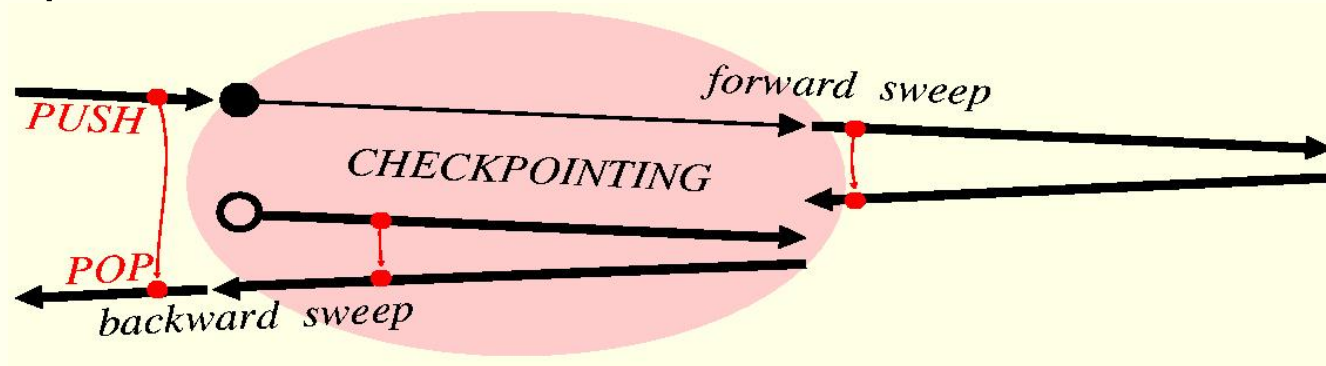


Reverse AD

- ▶ There are plenty of advantages to reverse AD. Essentially, it returns gradients in just one run!

$$\bar{x} = f'^*(x) \cdot \bar{y} = f_1'^*(x_0) \cdot \dots \cdot f_{p-1}'^*(x_{p-2}) \cdot f_p'^*(x_{p-1}) \cdot \bar{y}$$

- ▶ but it implies a structure that has drawbacks too...



- ▶ so AD tools use many Data-Flow improvements, such as:
 - (*General:*) dependency, activity (in both directions)
 - (*Memory:*) TBR, Snapshot analysis
 - (*Time:*) ERA, Adjoint dead code, Reverse snapshots

Specifying Improvements

► Unfortunately, improvements are generally specified or justified **informally** (at best graphically). They sometimes conflict. Many problems are found after implementation...

⇒ **We want to** define an “algebraic” specification of reverse programs, that captures these improvements, so to get:

- derived data-flow analyses, specialized for reverse programs, and taking profit of their particular structure,
- formal justifications,
- modelization of tradeoffs and conflicts,
- a firm ground for implementation.

Focus on TBR and Adjoint Liveness

The (too) simple reverse AD model . . .

$$\overline{I}; \overline{D} = \overrightarrow{I}; \overline{D}; \overleftarrow{I} = \text{PUSH}(\mathbf{W}(I)); I; \overline{D}; \text{POP}(\mathbf{W}(I)); I'$$

. . . should also include . . .

- **TBR analysis:** Only restore variables necessary in the sequel, i.e. $\mathbf{W}(I) \cap \mathbf{R}(\overleftarrow{U})$.
- **Adjoint Liveness:** Execute I only if its output is needed in \overline{D} , i.e. $\mathbf{W}(I) \cap \mathbf{N}(\overline{D}) \neq \emptyset$

. . . but be careful, the two are apparently coupled!

Complete model for reverse AD

Algebraic model for reverse AD, with TBR and Adjoint liveness:

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

where $\text{adj-live}(I, D)$ is defined as $\mathbf{W}(I) \cap \mathbf{N}(\overline{D}) \neq \emptyset$
 and \mathbf{W} , \mathbf{R} , \mathbf{N} are the written, read, and needed sets.

From this complete model, we are able to derive/prove formally the following 4 properties.

(1) Deriving rules for TBR

The general rule for the \mathbf{R} analysis is classical:

$$\mathbf{R}(A; B) = \mathbf{R}(A) \cup (\mathbf{R}(B) \setminus \mathbf{K}(A))$$

We can specialize it on our complete model:

$$\mathbf{R}(\overleftarrow{U}; I) = \begin{cases} \mathbf{R}(\text{POP}(\mathbf{W}(I) \cap \mathbf{R}(I'; \overleftarrow{U})); I'; \overleftarrow{U}) \\ \quad = (\mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U})) \setminus \mathbf{K}(I) & \text{if } \textit{adj-live}(I, D) \\ \\ \mathbf{R}(I'; \overleftarrow{U}) = \mathbf{R}(I') \cup \mathbf{R}(\overleftarrow{U}) & \textit{otherwise} \end{cases}$$

We thus re-discover formally the intuitive rules for TBR,
but they depend on *adj-live*!

(2) Adequacy of PUSH/POP lemma

The PUSH/POP mechanism in the complete model is adequate: it ensures that all pairs of instructions I and I' are executed in an equivalent context.

Formally, for any split $U; X$ of P , we can prove that

$$\mathbf{W}(U \vdash \overline{X}) \cap \mathbf{R}(\overleftarrow{U}) = \emptyset$$

by induction on the length of X , and exploring all possible cases.

(3) Deriving rules for Adjoint Liveness

We specialize the general rule for liveness analysis:

$$\mathbf{N}(A; B) = \mathbf{N}(B) \otimes Dep(A)$$

for the complete model of reverse AD, computing

$$\mathbf{N}(U \vdash \overline{I}; \overline{D})$$

This gives (using adequacy lemma):

$$\begin{aligned} \mathbf{N}(\overline{[]}) &= \emptyset \\ \mathbf{N}(\overline{I}; \overline{D}) &= \mathbf{N}(I') \cup (\mathbf{N}(\overline{D}) \otimes Dep(I)) \end{aligned}$$

which turns out to be **independent** from U and *adj-live*!

So there is **no circularity** after all: Adjoint Liveness \longrightarrow TBR.

(4) Deriving rules for Adjoint Write

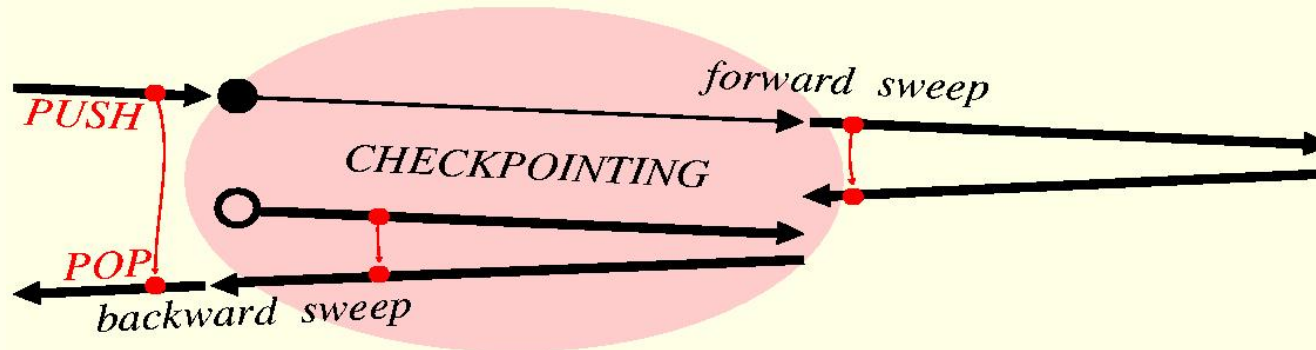
Definition of a very concise snapshot for checkpointing piece C in code $U; C; D$:

$$\mathbf{snapshot} = \mathbf{N}(\overline{C}) \cap (\mathbf{W}(C) \cup \mathbf{W}([U; C] \vdash \overline{D}))$$

Therefore we need specialized rules for $\mathbf{W}([U; C] \vdash \overline{D})$. Again we specialize the general rule for \mathbf{W} on the complete model of reverse AD. We obtain:

$$\mathbf{W}(U \vdash \overline{I}; \overline{D}) = \begin{cases} (\mathbf{W}(I) \cup \mathbf{W}([U; I] \vdash \overline{D})) \setminus (\mathbf{K}(I) \cap \mathbf{R}(I'; \overleftarrow{U})) & \text{if } \text{adj-live}(I, D) \\ \mathbf{W}([U; I] \vdash \overline{D}) & \text{otherwise} \end{cases}$$

A Tradeoff to explore



We chose to build \bar{D} in the context $[U; C]$, therefore:

$$\text{snapshot} = \mathbf{N}(\bar{C}) \cap (\mathbf{W}(C) \cup \mathbf{W}([U; C] \vdash \bar{D}))$$

Alternatively, we could add an extra requirement to \bar{D} 's context, asking TBR to also preserve $\mathbf{N}(\bar{C}) \setminus \mathbf{W}(C)$ during \bar{D} . Then we would build:

$$(\mathbf{R}(\bar{U}) \cup \mathbf{N}(\bar{C})) \setminus \mathbf{W}(C) \vdash \bar{D}$$

that may PUSH/POP more, but the snapshot

$$\text{snapshot} = \mathbf{N}(\overline{C}) \cap \mathbf{W}(C)$$

gets smaller. \Rightarrow needs further study!...

Applications

- Formalization makes us confident in the data-flow analyses.
 - Implementation follows the data-flow equations closely.
- ⇒ illustration on a piece of code.
- ⇒ speedup measurements.

```

subroutine  $\overline{\text{FLW2D}}$ (...,  $\overline{g3}$ ,  $\overline{g3}$ ,  $\overline{g4}$ ,  $\overline{g4}$ ,  $\overline{rh3}$ ,  $\overline{rh3}$ ,  $\overline{rh4}$ ,  $\overline{rh4}$ , ...)
  ...
do iseg=nsg1,nsg2
  is1 = nub0(1, iseg)
  ...
  qs = t3(is2)*vnocl(2, iseg)
  dplim = qsor*g4(is1) + qs*g4(is2)
  rh4(is2) = rh4(is2) - dplim
  pm = pres(is1) + pres(is2)
  dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2, iseg)
  rh3(is1) = rh3(is1) + dplim
  call PUSH(pm, sq)
  call LSTCHK(pm, sq)
  call POP(pm, sq)
  call  $\overline{\text{LSTCHK}}$ (pm,  $\overline{pm}$ , sq,  $\overline{sq}$ )
   $\overline{dplim}$  =  $\overline{rh3}$ (is1) -  $\overline{rh3}$ (is2)
  ...
   $\overline{vnocl}$ (2, iseg) =  $\overline{vnocl}$ (2, iseg)+t3(is2)* $\overline{qs}$ +t3(is1)* $\overline{qsor}$ 
   $\overline{t3}$ (is1) =  $\overline{t3}$ (is1) + vnocl(2, iseg)* $\overline{qsor}$ 
enddo
end

```

Experimental Results

Adjoint Liveness and Adjoint Write implemented in TAPENADE.

application:	ALYA (<i>CFD</i>)	UNS2D (<i>CFD</i>)	THYC (<i>Thermo</i>)	LIDAR (<i>Optics</i>)
t(P):	0.85	2.39	2.67	11.22
t(\bar{P}):	5.65	29.70	11.91	23.17
new t:	4.62	24.78	10.99	22.99
gain:	18%	16%	8%	7%
M(\bar{P}):	10.9	260	3614	16.5
new M:	9.4	259	3334	16.5
gain:	14%	0%	8%	0%

Adjoint Data Dependency Analysis

► Classical Data-Dependency analysis can also be specialized for adjoint programs.

⇒ Application 1: gather derivative instructions:

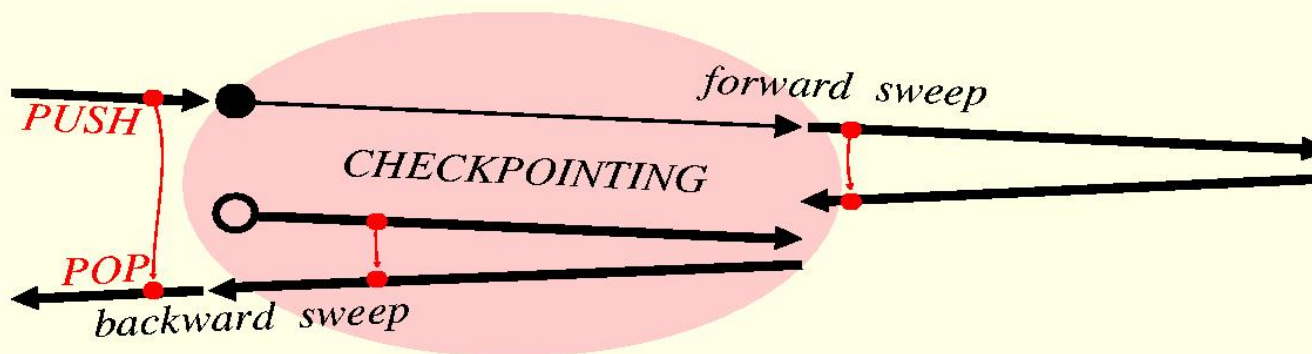
$$\begin{aligned} \overline{dplim} &= 0.0 \\ \overline{dplim} &= \overline{dplim} - \overline{rh3(is2)} & \Rightarrow & \overline{dplim} = \overline{rh3(is1)} - \overline{rh3(is2)} \\ \overline{dplim} &= \overline{dplim} + \overline{rh3(is1)} \end{aligned}$$

⇒ Application 2: gather “vector” derivative loops (tangent mode):

<pre>Do i =1,ndir a = x + 2*a EndDo a = x + 2*a Do i =1,ndir x = a*cos(x)*x + sin(x)*a EndDo x = a*sin(x)</pre>	⇒	<pre>a = x + 2*a Do i =1,ndir a = x + 2*a x = a*cos(x)*x + sin(x)*a EndDo x = a*sin(x)</pre>
---------------------------------------------------------------------------------------------------------------------	---	--------------------------------------------------------------------------------------------------

Conclusion

- ▶ Defined AD-specific Data-Flow analyses.
 - ▶ Formulated adjoint AD programs algebraically.
 - ▶ Derived formally the rules of Data-Flow analyses.
 - ▶ Compilers' general Data-Flow analyses can't perform as well, because they can't use the adjoint structure!
 - ▶ Still more to gain applying compiler technology to AD!
- ⇒ More tradeoffs to explore
(*e.g. sequences of checkpoints*)
- ⇒ New analyses to incorporate
(*e.g. reverse checkpoints*).



i.e. if $\text{adj-live}(C, D)$:

$$\begin{aligned}
 U \vdash \overline{C}; \overline{D} &= \text{PUSH}(\mathbf{W}(C) \cap \mathbf{R}(\overleftarrow{U})); \\
 &\text{PUSH}(\mathbf{SNP}(U, C, D)); \\
 &C; \\
 &[U; C] \vdash \overline{D}; \\
 &\text{POP}(\mathbf{SNP}(U, C, D)); \\
 &\square \vdash \overline{C}; \\
 &\text{POP}(\mathbf{W}(C) \cap \mathbf{R}(\overleftarrow{U}));
 \end{aligned}$$

$$\mathbf{SNP}(U, C, D) = \mathbf{N}(\overline{\text{overline}C}) \cap (\mathbf{W}(C) \cup \mathbf{W}([U; C] \vdash \overline{D}))$$

and otherwise:

$$\begin{aligned}
 U \vdash \overline{C}; \overline{D} &= \text{PUSH}(\mathbf{SNP}(U, C, D)); \\
 & [U] \vdash \overline{D}; \\
 & \text{POP}(\mathbf{SNP}(U, C, D)); \\
 & [U] \vdash \overline{C}; \\
 \mathbf{SNP}(U, C, D) &= \mathbf{N}(\overline{C}) \cap \mathbf{W}(U \vdash \overline{D})
 \end{aligned}$$