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

Principles of Reverse AD

AD rewrites source programs to make them compute derivatives.

$$
\begin{array}{lrl}
\text{consider:} & P: & \{I_1; I_2; \ldots I_p;\} \;\; \text{implementing } f: \mathbb{R}^m \to \mathbb{R}^n \\
\text{identify with:} & f = & f_p \circ f_{p-1} \circ \cdots \circ f_1 \\
\text{name:} & x_0 = & x \;\; \text{and} \;\; x_k = f_k(x_{k-1}) \\
\text{chain rule:} & f'(x) = & f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdot \cdots \cdot f'_1(x_0)
\end{array}
$$

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

$$
\begin{array}{rll}
\dot{y} = f'(x) \cdot \dot{x} = & f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdot \cdots \cdot f'_1(x_0) \cdot \dot{x} & \text{tangent AD} \\
\overline{x} = f'^*(x) \cdot \overline{y} = & f'^*_1(x_0) \cdot \ldots f'^*_{p-1}(x_{p-2}) \cdot f'^*_p(x_{p-1}) \cdot \overline{y} & \text{reverse AD}
\end{array}
$$

Evaluate both from right to left !

Reverse AD

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

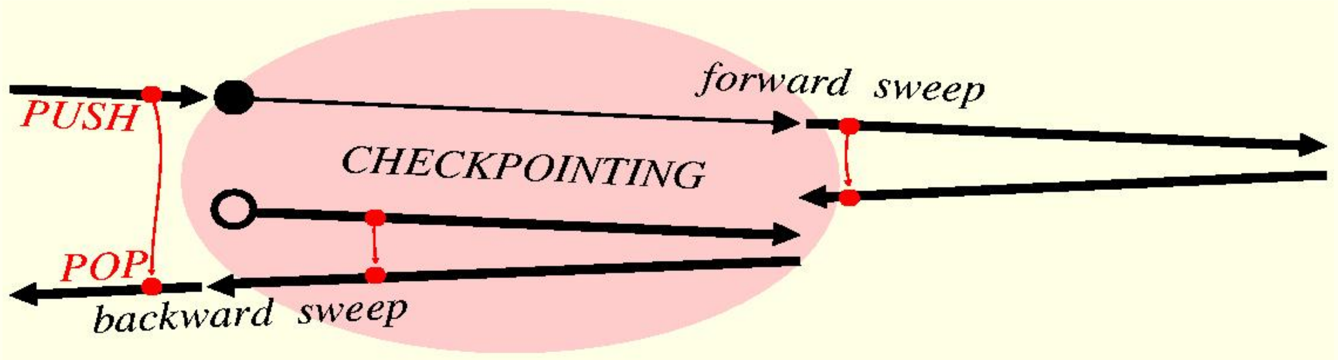
$\overline{x} = f'^*(x) \cdot \overline{y} = f_1'^*(x_0) \cdot \ldots f_{p-1}'^*(x_{p-2}) \cdot f_p'^*(x_{p-1}) \cdot \overline{y}$

Reverse AD

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

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

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

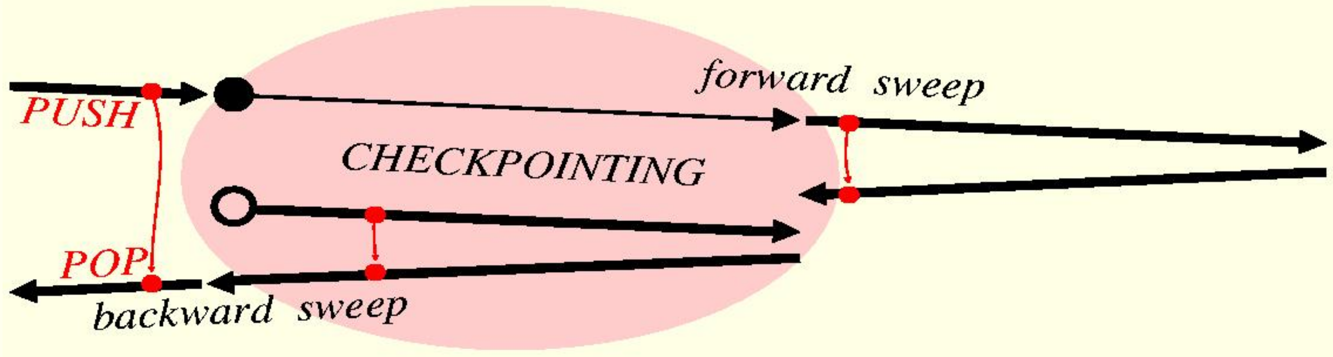


Reverse AD

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

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

▶ 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...

$\Rightarrow$ 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 . . .

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

. . . 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:

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

where $\textit{adj-live}(I,D)$ is defined as $\mathbf{W}(I) \cap \mathbf{N}(\overline{D}) \neq \emptyset$
and **W**, **R**, **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 **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:

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

We thus re-discover formally the intuitive rules for TBR,
but they depend on $\textit{adj-live}$!

(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

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

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

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

This gives (using adequacy lemma):

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

which turns out to be independent from U and $adj\text{-}live$!
So there is no circularity after all: Adjoint Liveness $\longrightarrow$ TBR.

(4) Deriving rules for Adjoint Write

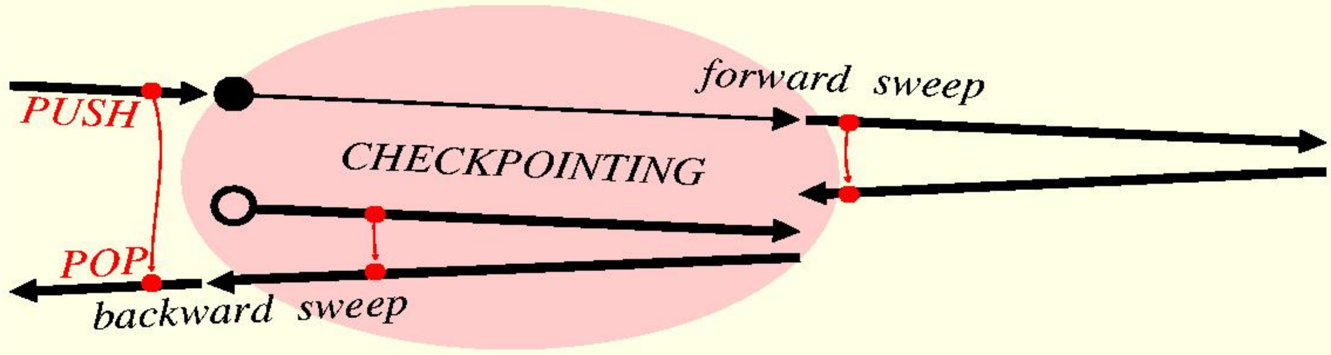
Definition of a very concise snapshot for checkpointing piece C in code $U; C; D$:

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

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

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

A Tradeoff to explore



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

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

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

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

that may PUSH/POP more, but the snapshot

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

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

Applications

- Formalization makes us confident in the data-flow analyses.

- Implementation follows the data-flow equations closely.

$\Rightarrow$ illustration on a piece of code.

$\Rightarrow$ speedup measurements.

```
subroutine FLW2D(...,g3,g3,g4,g4,rh3,rh3,rh4,rh4,...)
  ...
do iseg=nsg1,nsg2
  is1 = nubo(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 LSTCHK(pm, pm, sq, sq)
  dplim = rh3(is1) - rh3(is2)
  ...
  vnocl(2,iseg) = vnocl(2,iseg)+t3(is2)*qs+t3(is1)*qsor
  t3(is1) = t3(is1) + vnocl(2,iseg)*qsor
enddo
end
```

Experimental Results

Adjoint Liveness and Adjoint Write implemented in TAPENADE.

application:	ALYA (CFD)	UNS2D (CFD)	THYC ($Thermo$)	LIDAR ($Optics$)
t(P):	0.85	2.39	2.67	11.22
t($\overline{\text{P}}$):	5.65	29.70	11.91	23.17
new t:	4.62	24.78	10.99	22.99
gain:	18%	16%	8%	7%
M($\overline{\text{P}}$):	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:

$$\overline{\texttt{dplim}} = 0.0$$
$$\overline{\texttt{dplim}} = \overline{\texttt{dplim}} - \overline{\texttt{rh3}}\texttt{(is2)}$$
$$\overline{\texttt{dplim}} = \overline{\texttt{dplim}} + \overline{\texttt{rh3}}\texttt{(is1)}$$

⇒

$$\overline{\texttt{dplim}} = \overline{\texttt{rh3}}\texttt{(is1)} - \overline{\texttt{rh3}}\texttt{(is2)}$$

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

```
Do i =1,ndir
   ȧ = ẋ + 2*ȧ
EndDo
a = x + 2*a
Do i =1,ndir
   ẋ = a*cos(x)*ẋ + sin(x)*ȧ
EndDo
x = a*sin(x)
```

⇒

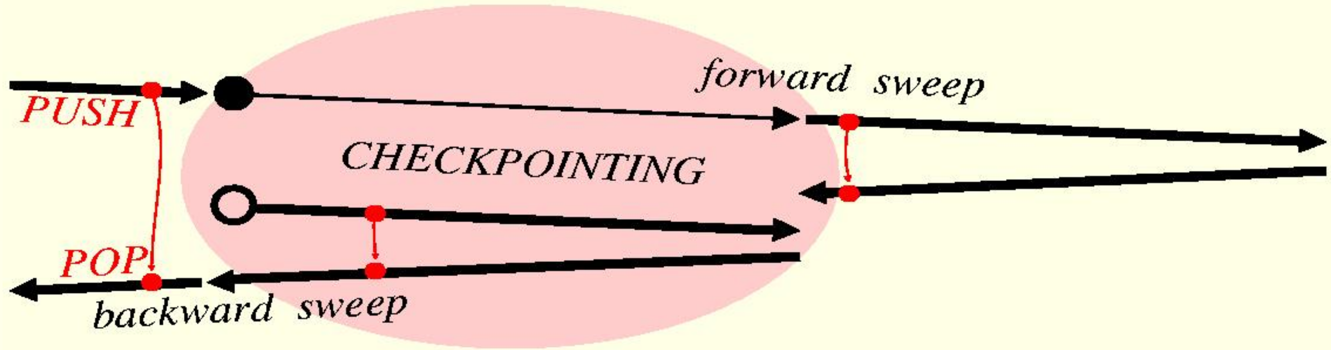
```
a = x + 2*a
Do i =1,ndir
   ȧ = ẋ + 2*ȧ
   ẋ = a*cos(x)*ẋ + sin(x)*ȧ
EndDo
x = a*sin(x)
```

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!

$\Rightarrow$ More tradeoffs to explore
(*e.g. sequences of checkpoints*)

$\Rightarrow$ New analyses to incorporate
(*e.g. reverse checkpoints*).



i.e. if adj-live(C, D):

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

and otherwise:

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