Data-Flow Algorithms for Reverse Automatic Differentiation

Laurent Hascoët, Mauricio Araya-Polo

TROPICS Project, INRIA Sophia-Antipolis

Principles of Reverse AD

AD rewrites source programs to make them compute derivatives.

consider: \( P : \{I_1; I_2; \ldots I_p; \} \) implementing \( f : \mathbb{R}^m \rightarrow \mathbb{R}^n \)

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: \[ P : \{I_1; I_2; \ldots I_p; \} \text{ implementing } f : \mathbb{R}^m \to \mathbb{R}^n \]

identify with: \[ f = f_p \circ f_{p-1} \circ \cdots \circ f_1 \]

name: \[ x_0 = x \text{ and } x_k = f_k(x_{k-1}) \]

chain rule: \[ f'(x) = f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdots \cdot f'_1(x_0) \]
Principles of Reverse AD

AD rewrites source programs to make them compute derivatives.

consider: $P : \{I_1; I_2; \ldots I_p; \}$ implementing $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$

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: $f'(x) = f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdots \cdot f'_1(x_0)$

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

$tangent AD$

\[ \dot{y} = f'(x) \cdot \dot{x} = f'_p(x_{p-1}) \cdot f'_{p-1}(x_{p-2}) \cdots \cdot f'_1(x_0) \cdot \dot{x} \]

\[ \overline{x} = f'^*(x) \cdot \overline{y} = f'^*_1(x_0) \cdots f'^*_{p-1}(x_{p-2}) \cdot f'^*_p(x_{p-1}) \cdot \overline{y} \]

reverse AD

Evaluate both from right to left!
Reverse AD

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

\[ \bar{x} = f'(x) \cdot \bar{y} = f'_1(x_0) \cdot \ldots \cdot f'_{p-1}(x_{p-2}) \cdot f'_p(x_{p-1}) \cdot \bar{y} \]
Reverse AD

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

\[
\overline{x} = f'(x) \cdot \overline{y} = f'_1(x_0) \cdot \ldots \cdot f'_{p-1}(x_{p-2}) \cdot f'_p(x_{p-1}) \cdot \overline{y}
\]

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!

\[
\bar{x} = f'^*(x) \cdot \bar{y} = f'^*(x_0) \cdot \ldots \cdot f'^*_{p-1}(x_{p-2}) \cdot f'^*_p(x_{p-1}) \cdot \bar{y}
\]

▶ 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 . . .

\[ \overrightarrow{I; D} = \overrightarrow{I}; \overrightarrow{D}; \overleftarrow{I} = \text{PUSH}(\text{W}(I)); I; \overrightarrow{D}; \text{POP}(\text{W}(I)); I' \]

. . . should also include . . .

- **TBR analysis:** Only restore variables necessary in the sequel, i.e. \( \text{W}(I) \cap \text{R}(U) \).

- **Adjoint Liveness:** Execute \( I \) only if its output is needed in \( \overrightarrow{D} \), i.e. \( \text{W}(I) \cap \text{N}(D) \neq \emptyset \)

. . . but be careful, the two are apparently coupled!
Complete model for reverse AD

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

\[ U \vdash I; D = [\text{PUSH}(W(I) \cap R(I'; \overleftarrow{U})); I]; \text{if adj-live}(I, D) \]

\[ [U; I] \vdash \overline{D}; \]

\[ [\text{POP}(W(I) \cap R(I'; \overleftarrow{U})); ]; \text{if adj-live}(I, D) \]

\[ I' \]

where \( \text{adj-live}(I, D) \) is defined as \( W(I) \cap N(D) \neq \emptyset \)

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:

\[ R(A; B) = R(A) \cup (R(B) \setminus K(A)) \]

We can specialize it on our complete model:

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

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

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

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:

\[ N(A; B) = N(B) \otimes Dep(A) \]

for the complete model of reverse AD, computing

\[ N(U \vdash \overline{I}; \overline{D}) \]

This gives (using adequacy lemma):

\[
\begin{align*}
N([[]]) &= \emptyset \\
N(I; D) &= N(I') \cup (N(D) \otimes Dep(I))
\end{align*}
\]

which turns out to be independent from \( U \) and \( adj-live! \)

So there is no circularity after all: Adjoint Liveness \( \rightarrow \) TBR.
(4) Deriving rules for Adjoint Write

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

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

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

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

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

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

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

$$(R(\overline{U}) \cup N(\overline{C})) \setminus W(C) \vdash \overline{D}$$
that may PUSH/POP more, but the snapshot

\[
\text{snapshot} = N(\overline{C}) \cap W(C)
\]

gets smaller. ⇒ 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 FLW2D(...,g3,g3,g4,g4,rh3,rh3,rh4,rh4,...) 
...
do iseg=nsg1,nsg2
    is1 = nubo(1,iseg)
...
    qs = t3(is2)*vnoc1(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*vnoc1(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)
    ...
    vnoc1(2,iseg) = vnoc1(2,iseg)+t3(is2)*qs+t3(is1)*qsor
    t3(is1) = t3(is1) + vnoc1(2,iseg)*qsor
enddo
end
# Experimental Results

Adjoint Liveness and Adjoint Write implemented in TAPENADE.

| application: | ALYA  
(CFD) | UNS2D  
(CFD) | THYC  
(Thermo) | LIDAR  
(Optics) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(t(P)):</td>
<td>0.85</td>
<td>2.39</td>
<td>2.67</td>
<td>11.22</td>
</tr>
<tr>
<td>(t(\overline{P})):</td>
<td>5.65</td>
<td>29.70</td>
<td>11.91</td>
<td>23.17</td>
</tr>
<tr>
<td>new (t):</td>
<td>4.62</td>
<td>24.78</td>
<td>10.99</td>
<td>22.99</td>
</tr>
<tr>
<td>gain:</td>
<td>18%</td>
<td>16%</td>
<td>8%</td>
<td>7%</td>
</tr>
<tr>
<td>(M(P)):</td>
<td>10.9</td>
<td>260</td>
<td>3614</td>
<td>16.5</td>
</tr>
<tr>
<td>new (M):</td>
<td>9.4</td>
<td>259</td>
<td>3334</td>
<td>16.5</td>
</tr>
<tr>
<td>gain:</td>
<td>14%</td>
<td>0%</td>
<td>8%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Adjoint Data Dependency Analysis

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

⇒ Application 1: gather derivative instructions:

\[
\frac{dp_{lim}}{dt} = 0.0 \\
\frac{dp_{lim}}{dt} = \frac{dp_{lim}}{dt} - \frac{rh3(is2)}{dt} \\
\frac{dp_{lim}}{dt} = \frac{dp_{lim}}{dt} + \frac{rh3(is1)}{dt}
\]

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

Do i = 1, ndir
\[
\dot{a} = \dot{x} + 2*\dot{a}
\]
EndDo
\[
a = x + 2*a
\]
Do i = 1, ndir
\[
\dot{x} = a*cos(x) * \dot{x} + \sin(x) * \dot{a}
\]
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!

⇒ More tradeoffs to explore
(\textit{e.g. sequences of checkpoints})
⇒ New analyses to incorporate
(\textit{e.g. reverse checkpoints}).
i.e. if \( \text{adj-live}(C, D) \):

\[
U \vdash \overline{C}; \overline{D} = \ \text{PUSH}(\mathbf{W}(C) \cap \mathbf{R}(\overline{U}));
\text{PUSH}(\mathbf{SNP}(U, C, D));
C;
[U; C] \vdash \overline{D};
\text{POP}(\mathbf{SNP}(U, C, D));
\emptyset \vdash \overline{C};
\text{POP}(\mathbf{W}(C) \cap \mathbf{R}(\overline{U}));
\mathbf{SNP}(U, C, D) = \mathbf{N}(\overline{\text{overline}C}) \cap (\mathbf{W}(C) \cup \mathbf{W}([U; C] \vdash \overline{D}))
\]
and otherwise:

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