Tutorial: Automatic differentiation with OpenAD & combinatorial problems

Jean Utke

- things to consider when choosing AD tools & methods
- OpenAD basic use
- reversal schemes
- computational graphs
- nonsmooth behavior
- writing models with AD in mind...
contributions to the OpenAD source code

Naumann
Norris
Tallent
Fagan
Strout
Gottschling
Lyons
summer students,...

numerous non-code contributions by Hovland, Hascoët, ...
what to pick...

i.e. matching application requirements with AD tools and techniques

the major advantages of AD are ... no need to repeat again

- knowing AD tool “internal” algorithms is of interest to the user
  (compare to compiler vector optimization)

- except for simple models and low computational complexity
  → can get away with “something”

- complicated models → worry about tool applicability

- high computational complexity → worry about efficiency of derivative computations

- tool availability (e.g. source transformation for C++ ?)
Source Transformation vs. Operator Overloading

- complicated implementation of tools
- especially for reverse mode
- full front end, back end, analysis
- efficiency gains from
  - compile time optimizations
  - activity analysis
  - explicit control flow reversal for reverse mode
- source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- benefits from external information
- efficiency depends on analysis accuracy
- simple tool implementation
- reverse mode (generating and reinterpreting an execution trace → inefficient)
- implemented as some library
- impact on efficiency:
  - library implementation (narrow scope)
  - compiler inlining capabilities (for low order)
  - use external information (sparsity etc.)
  - can do only runtime optimizations
- manual type change for operator overloading
  - complicated for formatted I/O, allocation
  - need matching signatures in Fortran
  - helped by use of templates

For higher-order derivatives combining source transformation based type change with overloaded operators is appropriate.
Forward vs. Reverse

- simplest rule: given $y = f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ use reverse if $n \gg m$ (gradient)

- what if $n \approx m$ and large
  - want only projections, e.g. $J \dot{x}$
  - sparsity (e.g. of the Jacobian)
  - partial separability (e.g. $f(x) = \sum(f_i(x_i)), x_i \in \mathcal{D}_i \subseteq \mathcal{D} \ni x$)
  - intermediate interfaces of different size

- the above may make forward mode feasible (projection $\bar{y}^T J$ requires reverse)

- higher order tensors (practically feasible for small $n$) → forward mode (reverse mode saves factor $n$ in effort only once)

- this determines overall propagation direction, not necessarily the local preaccumulation (combinatorial problem)
OpenAD overview

- www.mcs.anl.gov/OpenAD
- forward and reverse
- source transformation
- modular design
- large problems
- language independent transformation
- researching combinatorial problems
- current Fortran front-end Open64 (Open64/SL branch at Rice U)
- migration to Rose (already used for C/C++ with EDG)
- Rapsodia for higher-order derivatives via type change transformation
- uses association by address as opposed to association by name
toy example

```
subroutine head(x,y)
  double precision,intent(in) :: x
double precision,intent(out) :: y
!$openad INDEPENDENT(x)
y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

result of pushing it through the pipeline →

```
program driver
  use OAD_active
  implicit none
  external head
type(active):: x, y
  x%v=.5D0
  x%d=1.0
call head(x,y)
  print *, "F(1,1)=",y%d
end program driver
```

```c
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) OpenAD_Symbol_0...
...
REAL(w2f__8) OpenAD_Symbol_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
OpenAD_Symbol_0 = (X%v*X%v)
Y%v = SIN(OpenAD_Symbol_0)
OpenAD_Symbol_2 = X%v
OpenAD_Symbol_3 = X%v
OpenAD_Symbol_1 = COS(OpenAD_Symbol_0)
OpenAD_Symbol_5 = ((OpenAD_Symbol_3 +
  OpenAD_Symbol_2) * OpenAD_Symbol_1)
CALL sax(OpenAD_Symbol_5,X,Y)
RETURN
END SUBROUTINE
```
openad is Python script to invoke pipeline components for simple(!) settings

Usage:

```
~> openad -h
Usage: openad [options] <fortran-file>

Options:
-h, --help show this help message and exit
-m MODE, --mode=MODE basic transformation mode with MODE being one of: rs = reverse split; t = tracing; rj = reverse joint; f = forward;
-d DEBUG, --debug=DEBUG the debugging level
-i, --interactive requires to confirm each command
-k, --keepGoing keep going despite errors
-c, --copy copy run time support files instead of linking them
-n, --noAction display the pipeline commands, do not run them
```

progress messages:

```
~> openad -c -m f head.prepped.f90
parsing head.prepped.f90
analyzing source code and translating to xaif
  tangent linear transformation
  getting runtime support file OAD_active.f90
  getting runtime support file w2f__types.f90
  getting runtime support file iaddr.c
  translating transformed xaif to whirl
  unparsing transformed whirl to fortran
  postprocessing transformed fortran
⇒ run the example on the laptop and look at the transformation stages...
```
```bash
# parsing head.prepped.f90
${OPENADROOT}/Open64/osprey1.0/targ_ia32_ia64_linux/crayf90/sgi/mfef90 -z -F -N132 head.prepped.f90

# analyzing source code and translating to xaif
${OPENADROOT}/OpenADFortTk/OpenADFortTk-x86-Linux/bin/whirl2xaif -n -o head.prepped.xaif head.prepped.B

# tangent linear transformation
${OPENADROOT}/xaifBooster/../xaifBooster/algorithms/BasicBlockPreaccumulation/driver/oadDriver \n  -c ${OPENADROOT}/xaif/schema/examples/inlinable_intrinsics.xaif \n  -s ${OPENADROOT}/xaif/schema -i head.prepped.xaif -o head.prepped.xb.xaif

# getting runtime support file OAD_active.f90
cp -f ${OPENADROOT}/runTimeSupport/scalar/OAD_active.f90 ./

# getting runtime support file w2f__types.f90
cp -f ${OPENADROOT}/runTimeSupport/all/w2f__types.f90 ./

# getting runtime support file iaddr.c
cp -f ${OPENADROOT}/runTimeSupport/all/iaddr.c ./

# translating transformed xaif to whirl
${OPENADROOT}/OpenADFortTk/OpenADFortTk-x86-Linux/bin/xaif2whirl --structured head.prepped.B head.prepped.xb.xaif

# unparsing transformed whirl to fortran
${OPENADROOT}/Open64/osprey1.0/targ_ia32_ia64_linux/whirl2f/whirl2f -openad head.prepped.xb.x2w.B

# postprocessing transformed fortran
perl ${OPENADROOT}/OpenADFortTk/OpenADFortTk-x86-Linux/bin/multi-pp.pl -f head.prepped.xb.x2w.w2f.f
```
the same example in mode example

script invoked with a different flag

~> openad -c -m rj head.prepped.f90

```
program driver
  use OAD_active
  use OAD_rev
  implicit none
  external head
  type(active) :: x, y
  x%v=.5D0
  y%d=1.0D0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, 'driver running for x =',x%v
  print *, '  yields y =',y%v, ' dy/dx =',x%d
  print *, '  1+tan(x)^2-dy/dx =',1.0D0+tan(x%v)**2-x%d
end program driver
```

"> openad -c -m rj head.prepped.f90

parsing head.prepped.f90
analyzing source code and translating to xaif adjoint transformation
getting runtime support file OAD_active.f90
getting runtime support file w2f__types.f90
getting runtime support file iaddr.c
getting runtime support file ad_inline.f
getting runtime support file OAD_cp.f90
getting runtime support file OAD_rev.f90
getting runtime support file OAD_tape.f90
getting template file
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran

note: -m rj means reverse joint mode; needs extra run time support files OAD_cp/rev/tape; modified driver makes reference to the reversal scheme (checkpointing)
... need to talk about taping and checkpointing.
Reversal / Checkpointing Schemes

- why it is needed
- major modes
- OpenAD implementation
- alternatives
recap - why we need a tape...

\[ f : y = \sin(a \ast b) \ast c \]
yields a graph representing the order of computation:

- intrinsics \( \phi(\ldots, w, \ldots) \) have local partial derivatives \( \frac{\partial \phi}{\partial w} \)
- e.g. \( \sin(t_1) \) yields \( \cos(t_1) \)
- \textit{code list} \rightarrow intermediate values \( t_1 \) and \( t_2 \)
- all others already stored in variables

\[
\begin{align*}
t_1 &= a \ast b \\
p_1 &= \cos(t_1) \\
t_2 &= \sin(t_1) \\
y &= t_2 \ast c
\end{align*}
\]

What can we do with this?
reverse with adjoints

Assume variable and adjoints associated in pairs \((v, g_v)\):

append computations of adjoints

\[
\begin{align*}
t1 &= a * b \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
y &= t2 * c \\
g_c &= g_y * t2 \\
g_t2 &= g_y * c \\
g_t1 &= g_t2 * p1 \\
g_b &= g_t1 * a \\
g_a &= g_t1 * b
\end{align*}
\]

require \(p1\) in the adjoint sweep ⇒ recompute (time) or store (taping space)
may also need control flow trace and addresses...

original CFG $\Rightarrow$ record a path through the CFG $\Rightarrow$ adjoint CFG

often cheap with structured control flow and simple address computations (e.g. index from loop variables)

unstructured control flow and pointers are expensive
trace all at once = global split mode

- have memory limits - need to create tapes for short sections in reverse order
- subroutine is “natural” checkpoint granularity, different mode...

\[ S^n \] n-th invocation of subroutine S

- subroutine call
- order of execution
- run forward

- run forward and tape
- run adjoint
- store checkpoint
- restore checkpoint

- subroutine 1
  - call 2; ...
  - call 4; ...
  - call 2;
  - end subroutine 1
- subroutine 2
  - call 3
  - end subroutine 2
- subroutine 4
  - call 5
  - end subroutine 4
trace one SR at a time = global joint mode

taping-adjoint pairs
checkpoint-recompute pairs
the deeper the call stack - the more recomputations (unimplemented solution - result checkpointing)
familiar tradeoff between storing and recomputation at a higher level but in theory can be all unified.
• mix joint and split mode
• nested loop checkpointing in outer and inner loop body wrapper
• inner loop body in split mode
• \texttt{calc\_zonal\_transport} is used in both contexts
OpenAD reversal modes with checkpointing

subroutine level granularity

plain mode

split mode

Utke

Argonne
in OpenAD orchestrated with templates

- OpenAnalysis provides *side-effect analysis*
- provides checkpoint sets as (formal) arguments & references to global variables
- we ask for four sets: $\text{ModLocal} \subseteq \text{Mod}$, $\text{ReadLocal} \subseteq \text{Read}$

```
subroutine template()
  use OAD_tape ! tape storage
  use OAD_rev ! state structure

$\text{TEMPLATE\_PRAGMA\_DECLARATIONS}$
  if (rev_modetape) then
    ! the state component
    ! 'taping' is true

$\text{PLACEHOLDER\_PRAGMA}\; \text{id}=2$
  end if

    if (rev_modeadjoint) then
      ! the state component
      ! 'adjoint' run is true

$\text{PLACEHOLDER\_PRAGMA}\; \text{id}=3$
      end if

end subroutine template
```

$\Rightarrow$ run the simple example on the laptop with `-m rs` and `-m rj` and look at the output;
$\Rightarrow$ look at the ShallowWater example.
replacing hard wired logic with revolve

- loop extracted into subroutine...
- use revolve to control the behavior
- mercurial repository of a F9X implementation at
  http://mercurial.mcs.anl.gov/ad/RevolveF9X
User view on checkpointing

have model with high computational complexity and need adjoints

- have model with high computational complexity and need adjoints
- spatial requirements (NP complete DAG/call tree reversal)
- in theory: no distinction between checkpoints and trace
- limited automatic support
- in practice: well defined location for argument checkpoints
  - fix checkpoint location and spacing (trace fits into memory)
  - tool determines checkpoint elements
  - use hierarchical checkpointing (to limit number of checkpoints)
- optimize scheme e.g. with revolve (uniform steps)

but I want to try something else with this..., for instance
• we have 4 tape units
• $2^2$ and $2^3$ behave like split, $2^1$ behaves like joint
• How do we control the behavior?
• runtime estimates for checkpoint/tape size and recomputation effort → derive reversal scheme according to memory/runtime limits as dynamic call tree
runtime profiles...

- data “visibility” and upon forced inclusion in the scope name clashes...?
- experimenting with different checkpoint sets
  (OpenAnalysis supplies: REF vs. LREF vs. MOD vs. LMOD etc. and these \ {local vars})
- experimenting with result checkpoints...
checkpoint sets...

- always $\text{Read}_{\text{callee}} \subseteq \text{Read}_{\text{caller}}$
- multiple writes of $x \notin \text{Read}_{\text{Local}}$
- can store only $x \in \text{Read}_{\text{Local}}$ (except in callers whose callees don’t store anything)

- loose stack format; same storage requirements;
- same number of (‘big’) reads; fewer ’big’ writes.
- to experiment with this use different versions of store/restores in the template...
experimenting with result checkpoints

reevaluation count is reduced 😊
no stack storage 😊
... one more call layer

- a more suitable storage format is the *dynamic call tree*

- sample DCT generator can be found in the OpenAD run time support

and now for something completely different...
computational graphs in OpenAD
sidebar: preaccumulation & propagation I

- propagation = overall mode forward or reverse
- preaccumulation = local application of chain rule (view as graph operation)
- example: source code $\Rightarrow$ ssa form $\Rightarrow$ computational graph (DAG)

\[
\begin{align*}
t_1 &= x(1) + x(2) \\
t_2 &= t_1 + \sin(x(2)) \\
y(1) &= \cos(t_1 \times t_2) \\
y(2) &= -\sqrt{t_2}
\end{align*}
\Rightarrow
\begin{align*}
v_1 &= v_1 - v_0 + v_2 \\
v_2 &= \sin(v_0) \\
v_3 &= v_1 + v_2 \\
v_4 &= v_1 \times v_3 \\
v_5 &= \sqrt{v_3} \\
v_6 &= \cos(v_4) \\
v_7 &= -v_5
\end{align*}
\Rightarrow
\]

- chain rule application: multiplication of edge labels along paths & absorption of parallel edges by addition
- in the graph: elimination of (intermediate) vertices, edges, faces
• efficiency measure is operations count (at runtime)
• combinatorial problem (heuristics for optimization)
• problem: granularity ⇒ face elimination
• granularity is single \textit{fused multiply add}
• also requires heuristics
• elimination sequence terminates with tripartite dual graph, i.e. Jacobian
have preaccumulated local Jacobians;

given the $J_i, i = 1, \ldots, k$ we want to do:

- forward: $(J_k \circ \ldots \circ (J_1 \circ \dot{x}) \ldots)$, or
- reverse: $(\ldots (\bar{y}^T \circ J_k) \circ \ldots \circ J_1)$

the total cost:

- function evaluation + local partials (fixed)
- preaccumulation (NP-hard, varying with heuristic)
- propagation (fixed for a given preaccumulation)
  - for simplicity: one saxpy per non-unit $J_i$ element
  - potential for n-ary saxpys (generated)

What – other than the preaccumulation heuristic - can vary?
observation: Jacobian accumulation can obscure sparse / low rank dependencies

example: consider $f(x) = (D + ax^T)x$ with an intermediate variable $z = x^T x$
that has $\partial z/\partial x_i = 2x_i$

\[\begin{array}{c}
\Large{\textbf{scarcity}} \\
\end{array}\]

now we have $n^2$ variable edge labels vs. $n$ variable and $2n$ constant ones

- want: “minimal” representation
- scarcity: discrepancy of $nm$ vs dimension of the manifold of all $J(x), x \in \mathcal{D}$
- required ops: edge eliminations, reroutings, normalization
- avoid refill, backtrack, randomized heuristics, propagate through remainder graph
- reachability of a minimal representation. e.g. w/o algebraic dependencies?
- cheap propagation through remainder dual graph?
DAG with unit/constant edges
scarcity heuristics - example behavior

non-unit edge count over edge elimination step; variation via avoiding refill:

at minimum 26 reroutings performed; further post-elimination reduction via 8 normalizations
Note: relies heavily on precise data dependency analysis ⇐ coding style (!)
similar concerns as with sparsity: (local) automatic improvement observed up to factor 2 but application-level exploitation is desired.
experimenting with computational graphs...

... in *angel* (Automatic differentiation Nested Graph Elimination Library)

- build graphs within *xaifBooster*
- communicate via *CrossCountryInterface* to *angel*
- graph structure + extras for nodes/edges
- elimination etc happens within *angel*
- code generation within *xaifBooster*
- graph visualized with *graphviz*

⇒ look at an example
lion example

in Examples/Lion
do
make; make show; make showScarce
to get output like this:

⇒ look at some code in angel/src/heuristics.cpp:1125 and the interface Elimination.hpp.
is the model \( f \) smooth?

examples:

- \( y = \text{abs}(x) \); gives a kink
- \( y = (x > 0) ? 3 \times x : 2 \times x + 2 \); gives a discontinuity
- \( y = \text{floor}(x) \); same
- \( Y = \text{REAL}(Z) \); what about \( \text{IMAG}(Z) \)
- if (a == 1.0)
  \[ \begin{aligned}
  &\quad y = a; \\
  &\quad \text{else if (a == 0.0) then}
  \end{aligned} \]
  \( y = 0; \)
  else
  \( y = a \times b; \)
  intended: \( \dot{y} = a \dot{b} + b \dot{a} \)
- \( y = \sqrt{a^4 + b^4}; \)

**AD does not perform algebraic simplification**, i.e. for \( a, b \to 0 \) it does \( \left( \frac{d \sqrt{t}}{dt} \right) \xrightarrow{t \to +0} +\infty \).

AD computes derivatives of programs(!)

---

know your application e.g. fix point iteration, self adjoint, step size computation, convergence criteria
non-smooth models

observed:

- INF, NaN
- oscillating derivatives (may be glossed over by FD) or derivatives growing out of bounds
non-smooth models II

- blame AD tool - verification problem
  - forward vs reverse (dot produce check)
  - compare to FD
  - compare to other AD tool

- blame code, model’s built-in numerical approximations, external optimization scheme or inherent in the physics?

- higher order models in mech. engineering, beam physics, AtomFT explicit g-stop facility for ODEs, DAEs

- what to do about first order
  - Adifor: optionally catches intrinsic problems via exception handling
  - Adol-C: tape verification and intrinsic handling
  - OpenAD (comparative tracing)
Differentiability

piecewise differentiable function:
\[ |x^2 - \sin(|y|)| \]
is (locally) Lipschitz continuous; almost everywhere differentiable (except on the 6 critical paths)

- Gâteaux: if \( \exists \quad df(x, \dot{x}) = \lim_{\tau \to 0} \frac{f(x+\tau \dot{x})-f(x)}{\tau} \) for all directions \( \dot{x} \)

- Bouligand: Lipschitz continuous and Gâteaux

- Fréchet: \( df(., \dot{x}) \) continuous for every fixed \( \dot{x} \) ... not generally

- in practice: often benign behavior, directional derivative exists and is an element of the generalized gradient.
case distinction

3 locally analytic

2 locally analytic but crossed a (potential) kink (\texttt{min,max,abs,...}) or discontinuity (\texttt{ceil,...}) [for source transformation: also different control flow]

1 we are exactly at a (potential) kink, discontinuity

0 tie on arithmetic comparison (e.g. a branch condition) → potentially discontinuous (can only be determined for some special cases)

[-1 (operator overloading specific) arithmetic comparison yields a different value than before (tape invalid → sparsity pattern may be changed,...)]
Should AD make educated guesses?

consider \( y = \max(a(x), b(x)) \)

at the tie

\[ y = \sqrt{x} \text{ and } \dot{y}|_{x=+0} = \begin{cases} 
0 & \text{if } \dot{x} = 0 \\
+\infty & \text{if } \dot{x} > 0 \\
\text{NaN} & \text{if } \dot{x} < 0 
\end{cases} \]

consider \texttt{maxloc}: tie-breaking argument \texttt{maxval} may differ from argument identified by \texttt{maxloc}
classifying non-smooth events

```cpp
adouble foo(adouble x) {
    adouble y;
    if (x<=2.5)
        y=2*fmax(x,2.0);
    else
        y=3*floor(x);
    return y;
}
```

- tape at 2.2 and rerun at
  - 2.3 → 3
  - 2.0 → 1
  - 2.5 → 0
  - 2.6 → -1
- tape at 3.5 and rerun at
  - 3.6 → 3
  - 4.5 → 2
  - 2.5 → -1
- validates tape but is unspecific 😞

```cpp
#include "adolc.h"
adouble foo(adouble x);

int main() {
    adouble x,y;
    double xp,yp;
    std::cout << " tape at: " ;
    std::cin >> xp;
    trace_on(1);
    x <<= xp;
y=foo(x);
y >>= yp;
    trace_off();
    while (true) {
        std::cout << "rerun at: " ;
        std::cin >> xp;
        int rc=function(1,1,1,&xp,&yp);
        std::cout<<"return code: "<<rc<<std::endl;
    }
}
```
### tracing facility - example

```fortran
subroutine head(x,y)
  double precision :: x
  double precision :: y
  !$openad INDEPENDENT(x)
  y=tan(x)
  !$openad DEPENDENT(y)
end subroutine
```

```fortran
program driver
  use OAD_active
  use OAD_rev
  use OAD_trace

  type(active) :: x, y
  x%=.5D0
  ! first trace
  call oad_trace_init()
  call oad_trace_open()
  call head(x,y)
  call oad_trace_close()
  x%=x%+3.0D0
  ! second trace
  call oad_trace_open()
  call head(x,y)
  call oad_trace_close()
end program driver
```

```xml
<Trace number="1">
  <Call name="tan_scal" line="5">
    </Call>
  <Tan sd="0"/>
</Trace>

<Trace number="2">
  <Call name="tan_scal" line="5">
    </Call>
  <Tan sd="1"/>
</Trace>
```

indicates subdomain of \( \tan(x) \) is \( \text{sd}=k \) with integer \( k = \left\lfloor \frac{x+\pi/2}{\pi} \right\rfloor \)
tracing facility - control flow

check active control flow decisions:

test routine:

```fortran
subroutine head(x1,x2,y)
    real,intent(in) :: x1,x2
    real,intent(out) :: y
    integer i
    !$openad INDEPENDENT(x1)
    !$openad INDEPENDENT(x2)
    y=x1
    do i=int(x1),int(x2)+2
        y=y*x2
        if (y>1.0) then
            y=y*2.0
        end if
    end do
    !$openad DEPENDENT(y)
end subroutine head
```

trace at x=[0.5, 0.75]

```xml
<Trace number="1">
    <Loop line="8">
        <Branch line="10">
            <Cfval val="0"/>
        </Branch>
        <Branch line="10">
            <Cfval val="0"/>
        </Branch>
        <Branch line="10">
            <Cfval val="0"/>
        </Branch>
        <Cfval val="3"/>
    </Loop>
</Trace>
```

trace at x=[0.5, 1.75]

```xml
<Trace number="2">
    <Loop line="8">
        <Branch line="10">
            <Cfval val="0"/>
        </Branch>
        <Branch line="10">
            <Cfval val="1"/>
        </Branch>
        <Branch line="10">
            <Cfval val="1"/>
        </Branch>
        <Branch line="10">
            <Cfval val="1"/>
        </Branch>
        <Cfval val="4"/>
    </Loop>
</Trace>
```

note: difference between active and varied program variables.
tracing facility - data

associating events with program data:

test routine:

```fortran
subroutine head(x,y)
  real :: x(2),y
  !$openad INDEPENDENT(x)
  y=0.0
  do i=1,2
    y=y+sin(x(i))+tan(x(i))
  end do
  !$openad DEPENDENT(y)
end subroutine
```

trace at $x=[0.5, 0.75]$:

```xml
<Trace number="1">
  <Call name="tan_scal" line="6">
    <Arg name="X">
      <Index val="1"/>
    </Arg>
  </Call>
  <Tan sd="0"/>
  <Call name="tan_scal" line="6">
    <Arg name="X">
      <Index val="2"/>
    </Arg>
  </Call>
</Trace>
```

trace at $x=[0.5, 3.75]$:

```xml
<Trace number="2">
  <Call name="tan_scal" line="6">
    <Arg name="X">
      <Index val="1"/>
    </Arg>
  </Call>
  <Tan sd="0"/>
  <Call name="tan_scal" line="6">
    <Arg name="X">
      <Index val="2"/>
    </Arg>
  </Call>
</Trace>
```

note: no arguments recorded w/o address computation...
tracing facility - call stack

need call stack context (shown by nesting):

```
!$openad INDEPENDENT(x)
!
!
$openad DEPENDENT(y)
```

```
subroutine foo(t)
   real :: t
   call bar(t)
end subroutine

subroutine bar(t)
   real :: t
   t = tan(t)
end subroutine

subroutine head(x,y)
   real :: x
   real :: y
   y = x
   !$openad DEPENDENT(y)
end subroutine
```

```
trace at x=0.5
<Trace number="1">
  <Call name="foo" line="13">
    <Call name="bar" line="3">
      <Call name="tan_scal" line="7"></Call>
      <Tan sd="0"/>
    </Call>
  </Call>
</Trace>
```

```
trace at x=1.0
<Trace number="2">
  <Call name="foo" line="13">
    <Call name="bar" line="3">
      <Call name="tan_scal" line="7"></Call>
      <Tan sd="0"/>
    </Call>
  </Call>
</Trace>
```

note: tracing difference only for the direct call from `head`, not from `foo`
obvious (by now) recommendations regarding smoothness:

- avoid introducing numerical special cases
- pathological cases at domain boundaries, initial conditions
- filter out computations outside of the actual domain (e.g. $\sqrt{0}$)
- consider explicit logic to smooth (e.g. $C^1$) kinks and discontinuities

alternative (unimplemented) approaches:

- slopes (interval based)
- Laurent series (w different rules regarding $\pm \text{INF}$ and NaN)

more details later
model coding standard & AD tool capabilities II

want: precise compile-time data flow analysis (activity, side effect, etc...)
have: conservative overestimate of aliasing, MOD sets, ...
reducing the overestimate:

- extract the numerical core (!)
  - encapsulate ancillary logic (monitoring, debugging, timing, I/O,...)
  - small classes, routines, source files (good coding practice anyway)
  - extraction via source file selection
  - filtered-out routines treated as “black box”, with optimistic(!) assumptions
  - provide stubs when optimistic assumptions are inappropriate
  - transformation shielded from dealing with non-numeric language features
  - note: the top level model driver needs to be manually adjusted
- avoid semantic ambiguities (void*, union, equivalence)
- avoid unstructured control flow (analysis, control flow reversal)
- beware of non-contiguous data, e.g. linked lists (checkpointing, reverse access)
- beware of indirection, e.g. a[h[i]] vs. a[i] (data dependence)
- implicit F77 style reshaping (overwrite detection)
model coding standard & AD tool capabilities III

want: to use *nice* feature $\mathcal{N}$

have: a tool that has no clue how to deal with $\mathcal{N}$

- dynamic resource handling in reverse mode, some examples:
  - dynamic memory (when locally released)
  - file handles (same)
  - MPI communicators (same)
  - garbage collectors ...

  no generic tool support, requires extensive bookkeeping

- concerns when dealing with third party libraries
  - availability of the source code
  - numerical core extraction
  - smoothness
  - analysis overhead (e.g. MPI ?)

  research underway for blas, lapack, MPI, openMP

- beware of out-of-core data dependencies (data transfer via files)
further info

• http://www.mcs.anl.gov/openad
  – instructions to download & build
  – documentation
  – revision history
  – bibliography
  – wiki
  – bug tracker


• community website http://www.autodiff.org