

(open)ad-joint issues

Jean Utke¹

¹University of Chicago and Argonne National Laboratory

July 17, 2009



Why adjoints by source transformation?

operator overloading

- simple tool implementation as a library
- adjoints: generate & reinterpret an execution trace → **inefficient**
- efficiency gains come from:
 - runtime AD optimization
 - optimized library
 - inlining
- requires manual type change

source transformation

- complicated implementation of tools
- especially for adjoints
- full front end, back end, analysis
- efficiency gains from
 - **compile time AD optimizations**
 - source code (esp. activity) analysis
 - explicit control flow reversal

Adjoints for computationally complex applications require source transformation!

Why is this a user concern?

Adjoint efficiency depends on AD transformation algorithms and exploiting higher level model properties (sparsity, iterative solvers, self adjointness,...)

BUT source transformation efficiency depends also on

- **capability for structured control flow reversal**
- **code analysis accuracy**
- **partitioning the execution for checkpointing**

the above are affected by

- use of programming language features
- using such features in certain inherently difficult to handle patterns
- programming style

therefore

- knowing some AD tool “internal” algorithms is of interest to the user (e.g. compare to compiler vectorization or interval arithmetic)
- only very simple models with low computational complexity
→ can get away with “something”
- fully automatic solutions exist for narrowly defined setups (e.g. NEOS)

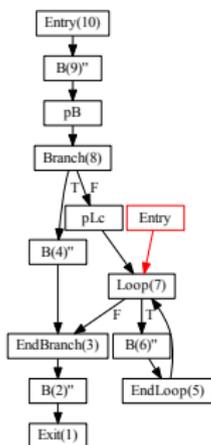
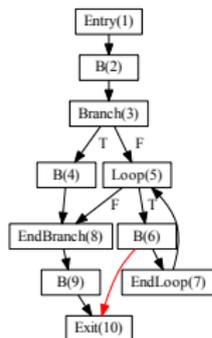
When dealing with any unsupported language feature / programming pattern :

- Does it have a supported alternative and is the alternative more efficient (and better maintainable in the model source)?
- Is the adjoint of such an alternative more efficient than the adjoint of the unsupported construct?
- What is the effort of changing the model vs. the effort of implementing a potentially complicated or rarely used or inherently inefficient adjoint transformation?

OpenAD mode of operation: implement language features on demand so that we can maximize the time available to improve the generally applicable AD algorithms!

Structured vs. Unstructured Control Flow

- think - GOTO, alternative ENTRY, early RETURN, ...
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, e.g. turn STOPS into some error routine call ,...)
- example: early return from within a loop (CFG left, adjoint CFG right)



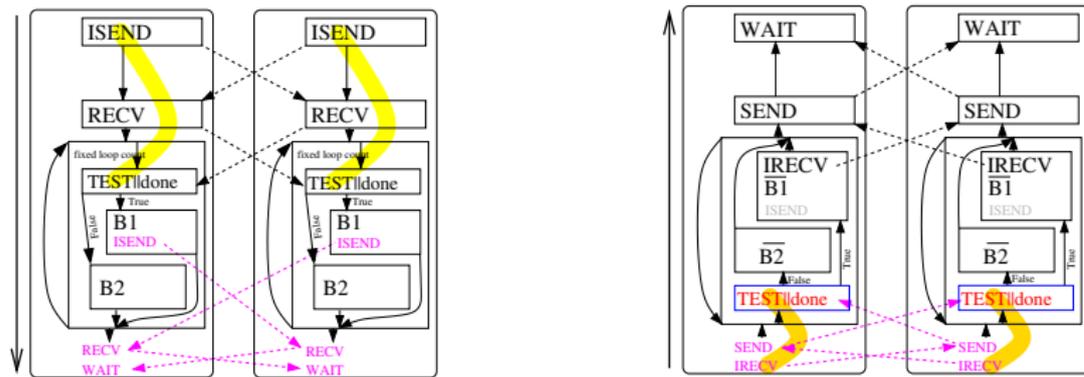
- all is fine without the red arrow
- by inspection: adjoint needs alternative ENTRY (or GOTO); but difficult to automate in general
- need to trace more control flow path details
- unstructured control flow is bad for compiler optimization, already for the original model!
- possible generic but inefficient fallback: trace enumerated basic blocks, replay inverse trace with GOTO <blockId> (no branches/loops left, more memory needed for trace)

Non-deterministic control flow

= control flow may change between two model executions on identical model inputs because of a multiuser system environment

examples:

- branching based on availability of system resources (that may be used by others), disk space, memory, system load
- communication in parallel execution for instance with mutexes, semaphores, (justified) use of `MPI_TEST` (test for completion of one exchg. 1 to early start exhg. 2, adjoint needs to switch test to exchg.2)



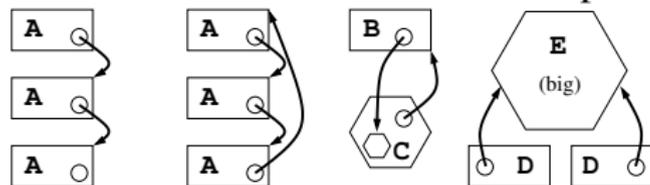
Non-deterministic control flow II

- hard to **automatically** detect the context to which a tested condition applies but the transformation requires the context information to correctly generate & place the adjoint test condition
- non-deterministic communication with MPI wildcards can be made deterministic (at the expense of lower efficiency) by recording the actual wild card values and using them in the adjoint sweep.
- google “adjoinable MPI”

Checkpointing and non-contiguous data

checkpointing = saving program data (to disk)

- “contiguous” data: scalars, arrays (even with stride > 1), strings, structures,...
- “non-contiguous” data: linked lists, rings, structures with pointers,...
- checkpointing is very similar to “serialization”
- Problem: decide when to follow a pointer and save what we point to



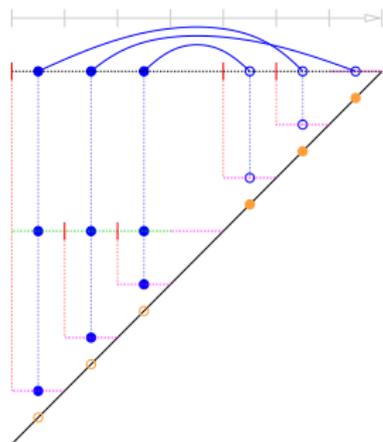
- unless we have extra info this is not decidable at source transformation time
- possible fallback: runtime bookkeeping of things that have been saved (is computationally expensive)

Semantically Ambiguous Data

- e.g. EQUIVALENCE (or its C counterpart `union`)
 - data dependence analysis: dependencies propagate from one variable to **all** equivalenced variables
 - “activity” (i.e. the need to generate adjoint code for a variable) leaks to all equivalenced variables whether appropriate or not
 - certain technical problems with the use of an active type (as in OpenAD)
- work-arrays (multiple, 0 semantically different fields are put into a (large) work-array); access via index offsets
 - data dependence analysis: there is *array section analysis* but in practice it is often not good enough to reflect the implied semantics
 - the entire work-array may become active / checkpointed
- programming patterns where the analysis has no good way to track the data dependencies:
 - data transfer via files (don't really want to assume all read data depends on all written data)
 - non-structured interfaces: exchanging data that is identified by a “string” as done for instance in the ESMF interfaces (if you feel bad about Fortran think of `void*` in C.)

Recomputation from Checkpoints and Program Resources

think of memory, file handles, sockets, MPI communicators,...



- problem when resource allocation and deallocation happen in different partitions (see hierarchical checkpointing scheme in the figure on the left)
- current AD checkpointing **does not track resources**
- dynamic memory is “easy” as long as nothing is deallocated before the adjoint sweep is complete.

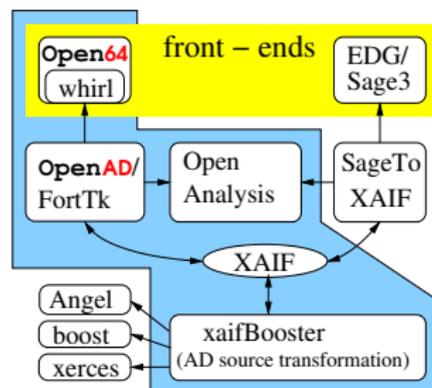
options to handle local deallocations

```
1  subroutine foo(p,t)
2  integer, intent(inout), pointer, dimension(:) :: p
3  integer, target :: t(:)
4  t=2*p ! need adjoint pointer to point to (invisible) t1
5  p=>t ! pointer is overwritten
6  end subroutine
7
8  subroutine bar
9  interface
10 subroutine foo(p,t)
11 integer, intent(inout), pointer, dimension(:) :: p
12 integer, target :: t(:)
13 end subroutine
14 end interface
15 integer, target, allocatable :: t1(:), t2(:)
16 integer, pointer, dimension(:) :: p
17 allocate(t1(1)); allocate(t2(1))
18 t1(1)=1
19 p=>t1
20 call foo(p,t2)
21 print*, p(1) ! p points now to t2
22 end subroutine ! t1 and t2 are deallocated
23
24 program p
25 call bar()
26 end program
```

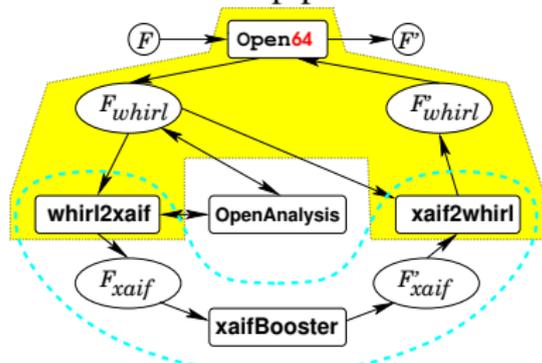
- modify model to reuse/grow allocated memory (rather than repeatedly allocate/deallocate), e.g. turn t_1 t_2 into global vars,...
- potential solution for allocate/deallocate **within a checkpointing partition without pointers**: track allocated memory to turn deallocates (here implicit on exit line 22) into allocates (of the appropriate size)
- potential (complicated) solution when pointers are involved: associate dynamic allocations in forward sweep to dynamic allocations in the adjoint sweep (adjoint needs to restore pointer overwritten on line 5, but stored pointer *value* references deallocated memory; need abstract association between forward allocate on line 17 and adjoint allocate corresponding to implicit deallocate on line 22)

quick OpenAD overview

- www.mcs.anl.gov/OpenAD
- forward and **reverse**
- source transformation
- modular design
- aims at 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)
- uses *association by address* (i.e. has an active type)
- Rapsodia for higher-order derivatives via type change transformation



Fortran pipeline:



summary

for OpenAD (and other AD tools) there is no “simple” characterization of what works and what doesn't

currently being extended in the OpenAD implementation are:

- complex/array arithmetic
- various Fortran syntax elements
- improved taping algorithm

If something doesn't work as expected - talk to us to find out if there is a quick fix or what it takes to make it work.