

# OpenAD update

Jean Utke<sup>1</sup>

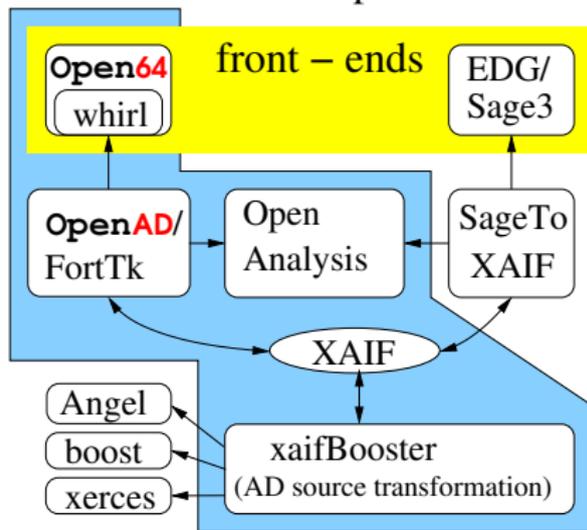
<sup>1</sup>University of Chicago and Argonne National Laboratory

CCSM Workshop  
June 30, 2010

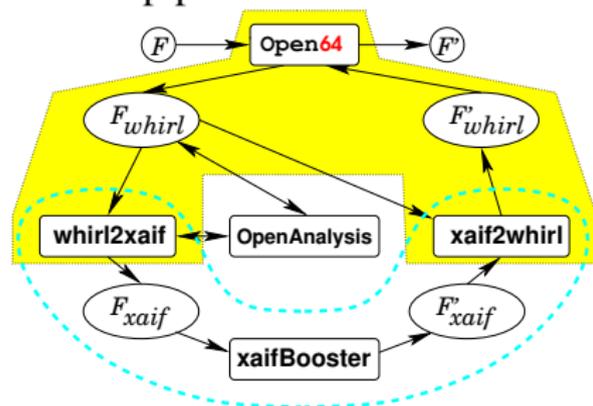


# update on OpenAD tool - new frontend

current frontend is Open64



Fortran pipeline:



## move to Rose frontend II

- Open64 front-end was split off from main trunk (U of Delaware and Pathscale) 9 years ago by Rice U
- is open source (but in parts inscrutable)
- was maintained until 3 years ago by Rice U but never updated from main trunk
- therefore not maintainable for newer Fortran features
- Rose is the only open source alternative suitable for our source transformation purpose
- uses the Fortran parser by Craig Rasmussen, LANL
- Rose compiler infrastructure is maintained by Dan Quinlan, LBNL + others
- also provides the C/C++ frontend (EDG, binary distribution) for ADIC
- move to Rose underway
  - reimplement analysis interfaces
  - reimplement interface to OpenAD transformation engine
  - debug Fortran parser / Rose AST functionality
- meanwhile maintain existing OpenAD to differentiate models from NE, NP, etc.

## move to Rose frontend II

- Open64 front-end was split off from main trunk (U of Delaware and Pathscale) 9 years ago by Rice U
- is open source (but in parts inscrutable)
- was maintained until 3 years ago by Rice U but never updated from main trunk
- therefore not maintainable for newer Fortran features
- Rose is the only open source alternative suitable for our source transformation purpose
- uses the Fortran parser by Craig Rasmussen, LANL
- Rose compiler infrastructure is maintained by Dan Quinlan, LBNL + others
- also provides the C/C++ frontend (EDG, binary distribution) for ADIC
- move to Rose underway
  - reimplement analysis interfaces
  - reimplement interface to OpenAD transformation engine
  - debug Fortran parser / Rose AST functionality
- meanwhile maintain existing OpenAD to differentiate models from NE, NP, etc.

... but since I got the chance to say something at this workshop

## part 2 - things to consider for a code to be “adjointed”

ingredients for adjoints:

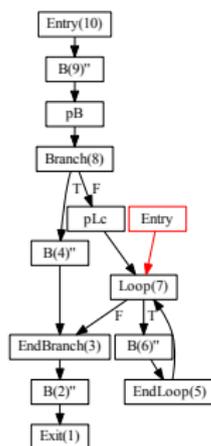
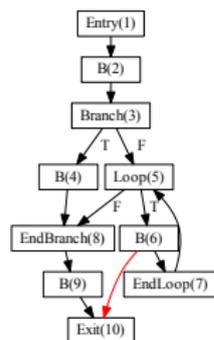
- reverse data dependencies, call structure, control flow
- done using a (partial) execution trace (needs lots of memory)
- memory requirements mitigated by recomputation from checkpoints

Some language features make the above harder / less efficient than other programming idioms that yield the same semantics!

(e.g. compare to compiler vectorization)

# Structured vs. Unstructured Control Flow

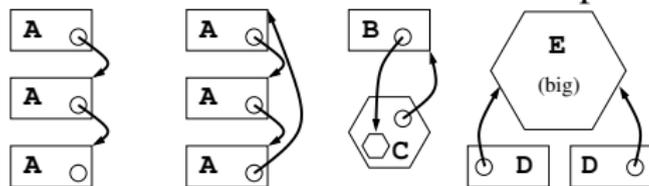
- think - GOTO, alternative ENTRY, early RETURN, ...
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, 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)

## Reversal/Checkpointing and non-contiguous data

- “contiguous” data: scalars, arrays (even with stride  $> 1$ ), strings, structures,...
- “non-contiguous” data: linked lists, rings, structures with pointers,...
- reversing  $p=p \rightarrow next$ ; needs backpointers or store the equivalent
- checkpointing is very similar to “serialization” (of data to disk)
- 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
- work-arrays (multiple, 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)
  - `void*` interfaces: exchanging data that is identified by “string” + cast

# Dynamic Memory

- dynamic memory is “easy” as long as nothing is deallocated before the adjoint sweep is complete OR no pointers point to allocated memory.
- prefer `ALLOCATABLE` instead of allocating space to a `POINTER` because the implied properties allow tighter analysis (and better code optimization by the compiler).

## summary

- most of the work goes to the Rose move
- initial analysis done for glimmer-cism (before the merge) w J. Campbell
- have started with small prototype models with Patrick Heimbach