Adifor90 Under Construction

Mike Fagan
Rice University

Outline

- Overview of Adifor90
- Pointer Problem

What is Adifor90?

- Adifor 3 is showing it’s age
- Using Adifor 3 on ‘mostly’ F77 with a ‘little’ F90 is about to collapse under the weight of the constant perl hacks I have been developing for various users.
- So, Adifor90 will be the next generation in the Adifor family.
  It will support Fortran 90 + k

New Features in Adifor90

- It will support both **by-name** and **by-address**
- There will be a native routine level **joint reversal mode**
- Improved derivative algorithms
  - Distributive law preaccumulation
  - OpenAD booster optimal preaccumulation
  - Other ?

Internals in Adifor90

- Internal representation is sexprs.
  (enhanced from the Adifor3 sexpr representation)
- Refactored Adifor 3 architecture:
  all differentiation, canonicalization done by the AD engine
- Added python to post processing (replace perl?)

A Word About Distribution Mechanisms

- Adifor uses a lot of components, so distributing working systems is problematic.
- The Adifor delivery system needs work.
  - Autoconf, libtool, etc
- Also, requiring that users set all of those environment variables is tedious. (Almost fixed now)

One more word about Distribution

- I plan on requiring that users have:
  - Perl >= 5.6
  - Python >= 2.4
  - Libc++ = lib.so.5
- Platforms I can support:
  - Intel 32- and 64-bit Linux
  - WinXP (likely using cygwin)
  - Suns
  - Macs (soon)
  - No more IBM AIX, SGI IRIX

Part 2: A vexing technical problem

- Last year, at the AD 2004 conference, many of us discussed the pointer problem.
- Of course, pointers per se are no problem, it is pointers in combination with dynamic memory.
- In essence, the problem is that in reverse mode, logging and then restoring a dynamic pointer value is unclean.

Do we have to save pointer values?

- Log partials instead of LHS *reduces* the need for pointer logging, but does not eliminate it.
- Like integer indices, need to log pointer values for storage of adjoints:

```
Pointer :: x        ➔        Pointer :: x
Z = x * y                    A_x += y * A_z
```

Saving pointers, cont.

- Always allocate adjoint quantities from a heap?
  - Storage management of help quantities
- some checkpointed quantities may be pointers

So, like to be able to log pointers
1. Adjoint storage
2. Checkpoints

In addition, pointer logging would be useful for enabling LHS value logging

Approaches I’ve Tried

- Ignore the free, log/restore the pointer value
- Surprisingly, this works on many Unix systems
  - Due to the way sbrk is implemented, and no multithreads
- Rejected, though because it depends on properties of code we can’t control

Approaches I’ve Tried, cont.

- Roll your own memory allocator
  - Replace users call to memory allocator with calls to special one.
  - Manage your own free list so that allocation requests that are repeated during the reverse sweep will yield the same pointer.
- Now log/restore actual pointer values
- This eliminates objections to ‘ignore’ approach, as we control the behavior of the memory allocator
- This is not so easy to get right / efficient.

Approaches I’ve Tried, cont.

- Limit user programs
  - No ‘unstacklike’ deallocation during active region

```
Allocate(A)
Allocate(B)
Allocate(C)
 ….
Deallocate(C)
Deallocate(B)
Deallocate(A)
```

This is ok

This covers a fair percentage of programs I’ve seen.

Correctness Theorem: Observational Equivalence

- How do we determine if a pointer reversal scheme is correct?
  ANSWER: Observational equivalence
- A pointer is 'correctly' restored if:
  - Any sequence of derefs that results in a primitive value yields the same primitive value from the restored pointer
  - Any 2 pointers that were the equal are still equal
    - Make sure that sharing still works
- In essence, pointer values do not matter, their effects do.

Dynamic Recompute: A Germ of an Idea

- Instead of logging a pointer (= address) value, consider logging code as well.
- Then, when restoring a pointer, execute the code instead of reassigning.

Motivating Example

```
Allocate(x,10)
Log(des(x),alloc,10)
…
x(3) = sin(z)
y => x(2)
Log(des(y),asgn,x(2))
Y(4) = 2 * z
…
Allocate(y,100)
Y(6) = z ** 2
…
Dealloc(x)
Log(des(x),dealloc)
…
Dealloc(y)
Log(des(y),dealloc)
```

```
Restore(y) ! Execs last code block = alloc ..

Restore(x) ! Execs last code block = alloc …

…

az += 2 * z *ay(6)

Dealloc(y)

Restore(y) ! Exec ptr assgn now (x restored)

…

az = 2 * ay

…

Az = cos(z) * ax(3)
```

What I think the rules are (so far)

- Associate with each pointer var a descriptor for keeping up with dynamic recompute
- When logging (or ckp) each pointer value changing operation gets a code block
- When restoring, first use of a pointer (including dealloc) triggers recompute
- When restoring, (fwd) alloc triggers dealloc, plus moves restore-instruction to next code block and restore
- When restoring, mark when code block has already been executed once (call by need)

Well, How is it working?

- Just started working with a user example [Actually uses F77 with malloc & cray pointers]
- Pointer stuff by hand
- Limited example works (pointer descriptors are global variables)
- I need to ‘do the math’ …

(not really a) Conclusion

- Adifor90 alpha probably middle/late may
- Adifor90 beta probably mid sept
- Adifor90 prelease end of year

pointer stuff is ongoing, and will be automated when ideas have crystallized