AD variants

Jean Utke(Argonne, MCS)

- recap from last year
- some of the transformation options we have ($\mapsto$ various tools)
- run-time/memory minimization and transformation options
- the path for OpenAD
- behind the scenes

recap from last year

the "motivational" bit:
we have a some model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ given as a (large) program

1. pretend to know nothing about the program and take finite differences of an oracle? - perhaps not.
2. get machine precision derivatives (avoid approximation vs. rounding problem)
3. the reverse mode (program reversal) yields "cheap gradients"
   locally for each elemental $y = \phi(x)$ we do $\boxed{\bar{x} = \bar{x} + \bar{y} \cdot \frac{\partial y}{\partial x}}$ which yields
   globally $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ (for $m = 1$, $\bar{\boldsymbol{x}} = \nabla f$)
4. if the program is large, so is the adjoint, so is the effort to do it manually
   ... and it is easy to get wrong but hard to debug

get a tool to do it "automatically"

a little reminder

model $\boldsymbol{f}$ contains:	reverse part (augmented for correctness):
$\boxed{1}$ $a = \alpha(x)$	$\boxed{k}$ $\bar{b} = \bar{b} + \bar{y} \cdot \frac{\partial \gamma}{\partial b}$ $\quad \bar{y} = 0$
$\boxed{2}$ $b = \beta(a)$	$\vdots$ `// lots of reversed code`
$\vdots$ `// lots of code`	$\boxed{2}$ $\bar{a} = \bar{a} + \bar{b} \cdot \frac{\partial \beta}{\partial a}$ $\quad \bar{b} = 0$
$\boxed{k}$ $y = \gamma(b)$	$\boxed{1}$ $\bar{x} = \bar{x} + \bar{a} \cdot \frac{\partial \alpha}{\partial x}$ $\quad \bar{a} = 0$

Question 1: *How do we get the* $\frac{\partial y}{\partial x}$ *in correct order?*

store them

compute the partials, use a stack:	
1	$a = \alpha(x)$ push$\left(\frac{\partial \alpha}{\partial x}\right)$
2	$b = \beta(a)$ push$\left(\frac{\partial \beta}{\partial a}\right)$
	$\vdots$ // lots of code
k	$y = \gamma(b)$ push$\left(\frac{\partial \gamma}{\partial b}\right)$
directly followed by the reverse part:	
k	$\bar{b} = \bar{b} + \bar{y} \cdot$ pop() $\frac{\partial \gamma}{\partial b}$ $\bar{y} = 0$
	$\vdots$ // lots of reversed code
2	$\bar{a} = \bar{a} + \bar{b} \cdot$ pop() $\frac{\partial \beta}{\partial a}$ $\bar{b} = 0$
1	$\bar{x} = \bar{x} + \bar{a} \cdot$ pop() $\frac{\partial \alpha}{\partial x}$ $\bar{a} = 0$

and preaccumulate (when useful)	
1	$a = \alpha(x)$
2	$b = \beta(a)$ push$\left(\frac{\partial \beta}{\partial a} \cdot \frac{\partial \alpha}{\partial x}\right)$
	$\vdots$ // lots of code
k	$y = \gamma(b)$ push$\left(\frac{\partial \gamma}{\partial b}\right)$
directly followed by the reverse part:	
k	$\bar{b} = \bar{b} + \bar{y} \cdot$ pop() $\frac{\partial \gamma}{\partial b}$ $\bar{y} = 0$
	$\vdots$ // lots of reversed code
2/1	$\bar{x} = \bar{x} + \bar{b} \cdot$ pop() $\frac{\partial \beta}{\partial a} \cdot \frac{\partial \alpha}{\partial x}$ $\bar{b} = 0$

Question 2: *When should we preaccumulate?*

local preaccumulation

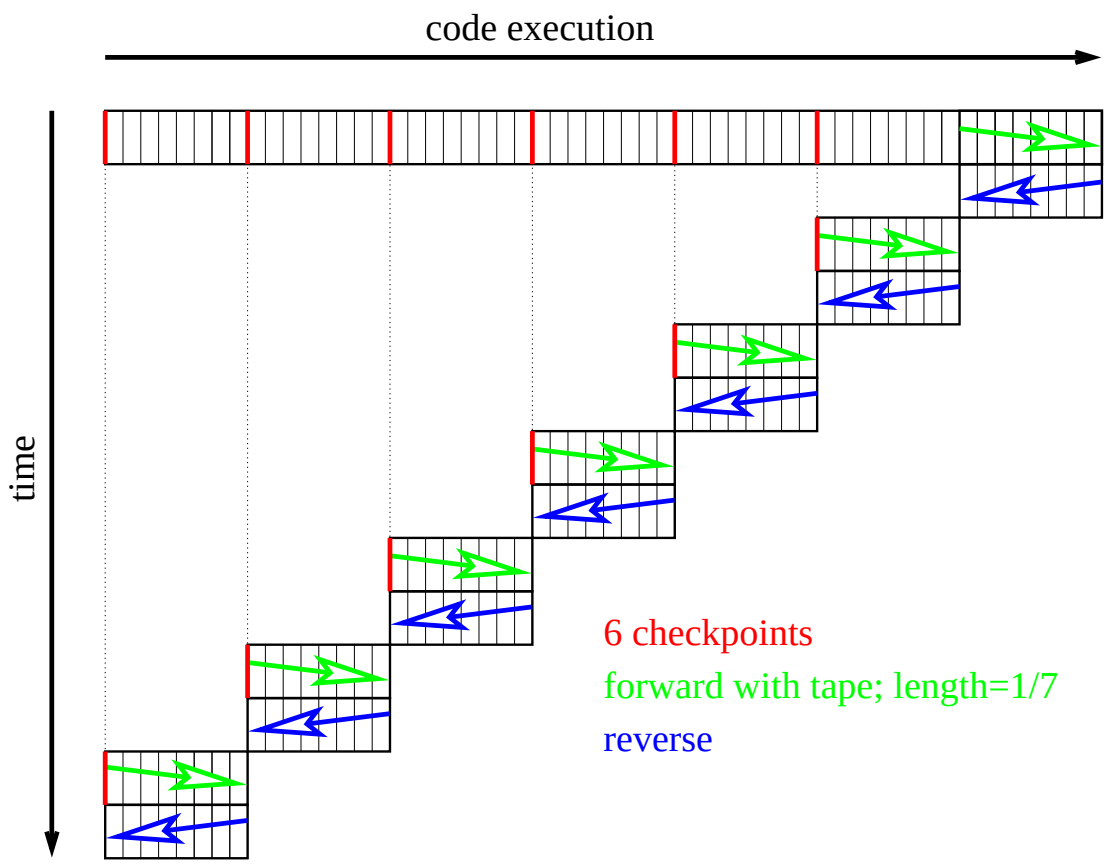
simplest interpretation:

- consider a subsection of code with n inputs, m outputs, and p computed intermediate values
- each $v_k = \phi(\ldots, v_(k'), \ldots)$ has partials $\{\ldots, \frac{\partial v_k}{\partial v_{k'}}, \ldots\}$
- eliminate the p intermediates yields the local Jacobian
  - computational cost of the elimination (vs. straight reverse)?
  - memory cost of the (sparse) local Jacobian (vs. the individual partials)?
  - in some cases "incomplete" elimination is beneficial (scarsity)
- NP hard problems
- to be decided locally (i.e. not uniformly for an entire model)
- tool needs capability to evaluate/generate these variants

the common framework is storing of partials, however

Question 3: *(what) should we store?*

can't store all at once anyway



- checkpoints & recomputation trade memory consumption for recomputation
- tool determines checkpoint contents using side effect analysis
- tool support to estimate reduced memory requirements
- checkpoint size (on file) is a lesser concern
- hierarchy of checkpoints

apply this locally ... and allow recomputation

store all vs. recompute all

store	recompute
pros	
• with preaccumulation • expect sparse Jacobians • linear complexity	• very few non-linear dependencies • partially covered by checkpoints • linear complexity with parallelizable loops
cons	
• Jacobians not sparse enough • no direct tap into checkpoints	• (local) quadratic complexity • difficult automation

local quadratic complexity: e.g. non-linear dependency computed in a loop

```
do i=1,k
  a=ζ(a); y(i)=φ(x(i),a,...)
end do
```

in practice...

- local decision on straight line code (basic block) to avoid quadratic complexity
- again non-uniform, i.e. requires capabilities for evaluating/generating variants
- in practice there is no such thing (yet)
- address computations are actually worse.

e.g. a statement like:

```
A(i)=2*B(j)
```

has the adjoint:

$$\bar{B}(j)=2*\bar{A}(i);\ \bar{B}(j)=0;$$

How do we get `i,j`?

...an 'incidental' example for the difference between a bandaid and a cure in OpenAD

the bandaid

assume *canonical* loops

- constant stride
- loop-constant boundaries
- no index updates in loop body
- all other index or condition variables loop-constant
- explicitly computable loop bounds
- tool needs to be able to flag violations!

for instance:

```
do i=1,n
  do j=1,i
    A(i)=2*B(j)
  end do
end do
```

reversed as:

```
do i=n,1,-1
  do j=i,1,-1
    B̄(j)=2*Ā(i); B̄(j)=0;
  end do
end do
```

spent a lot of time on implementing checks workarounds for “slight” deviations from canonical form

a larger bandaid to cover deviations

-various checks for canonical form;
-extended analysis results;
-extensions to control flow reversal;
-extra logic for external interfaces;
-coexisting transformation versions;

```
do i=1,n
  k=φ(i)
  do j=1,i
      A(i)=2*B(k)
  end do
end do
```

push after inner loop

nested indices

```
do i=1,n
  do j=1,i
    ...  A(I(i,j)) ...
  end do
end do
```

recursive

accept pseudo-constants wrt. top-level routine

```
do i=1,n
  do j=1,i
    if (Mask(i,j)!=0.0) ...
  end do
end do
```

constant/non-constant version

observing some symptoms

- reversing canonical loops suggests looking at inverse operations
- consider a simple example with non-constant stride

code	#1	#2	reverse	#2	#1
	j,k	j,k		j,k	j,k
j = 3	3,.		j=pop();k=pop()	9,7	
do while(j<10)	3,.	6,4	do while(j>=3)	9,7	6,7
k = j+1	3,4	6,7	$\overline{S}$(j,k)	9,7	6,7
j = k+2	6,4	9,7	k = j-2	9,7	6,4
S(j,k)	6,4	9,7	j = k-1	6,7	3,4
end do			end do		

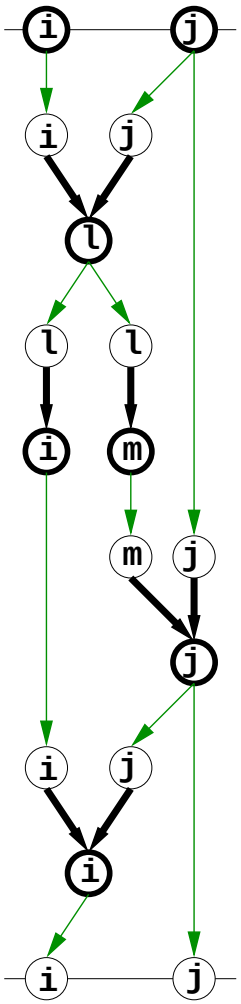
- a statement level inversion recipe doesn't help
- there is an ordering problem
- not all operations are invertible

an example for the cure

loop with address computation (intermediate active computations are omitted)

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```

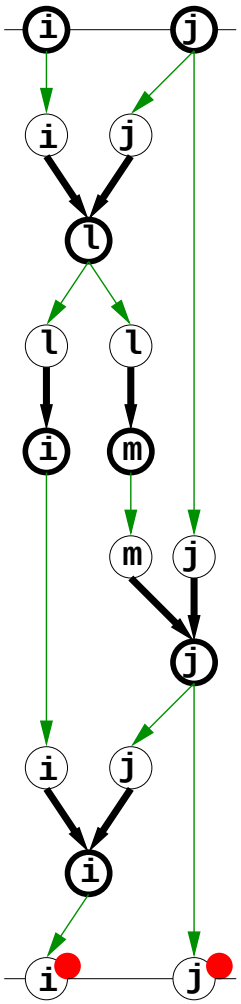


example cont. 1

the *token game*

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

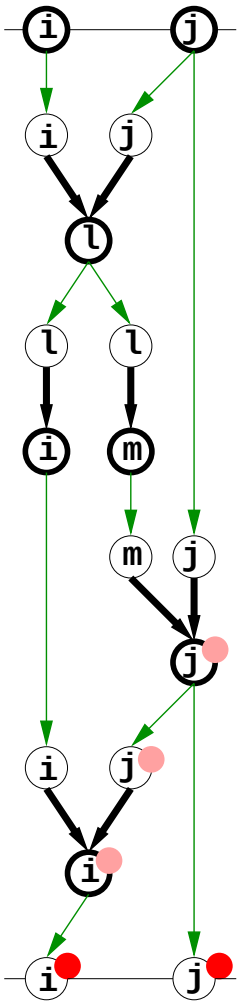
```
i=pop(); j=pop();
do while(...)

end do
```

example cont. 2

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

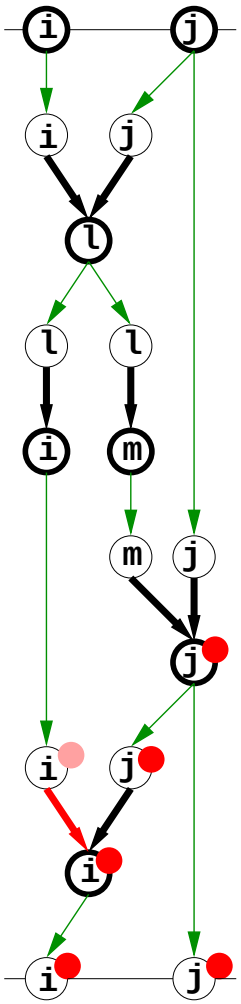
```
i=pop(); j=pop();
do while(...)

end do
```

example cont. 3

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

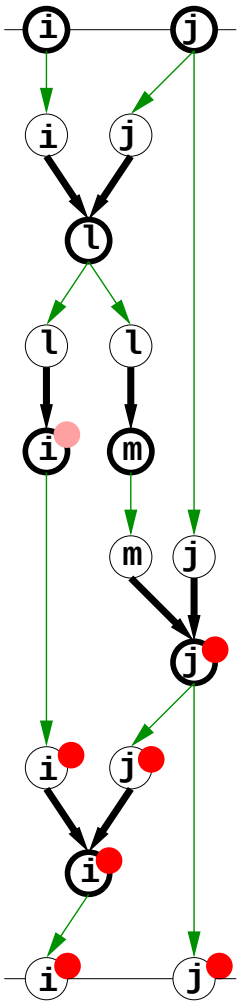
```
i=pop(); j=pop();
do while(...)
   i = i-j-1

end do
```

example cont. 4

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

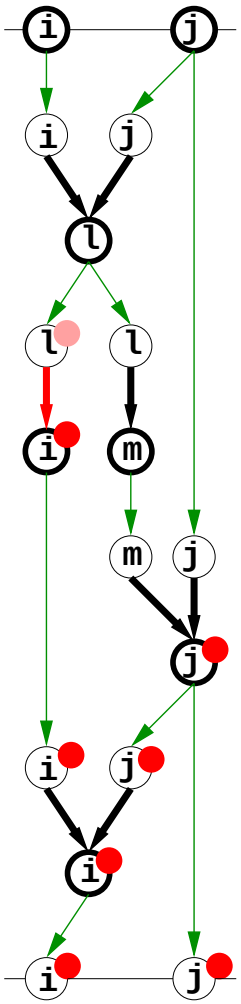
```
i=pop(); j=pop();
do while(...)
   i = i-j-1

end do
```

example cont. 5

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

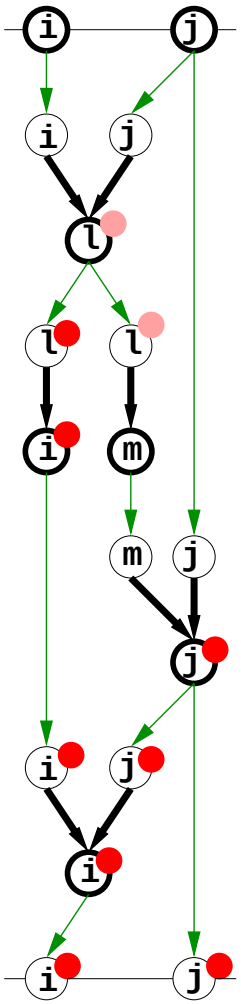
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2

end do
```

example cont. 6

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

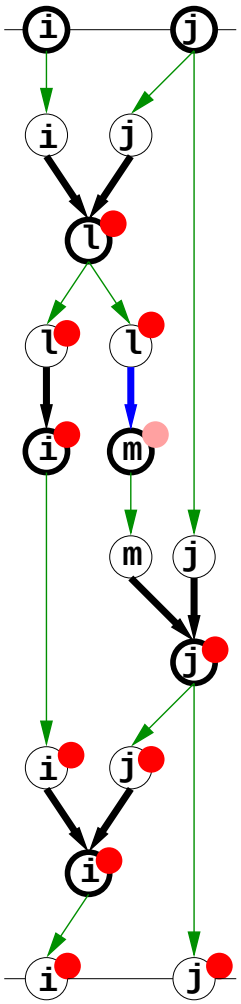
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2

end do
```

example cont. 7

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

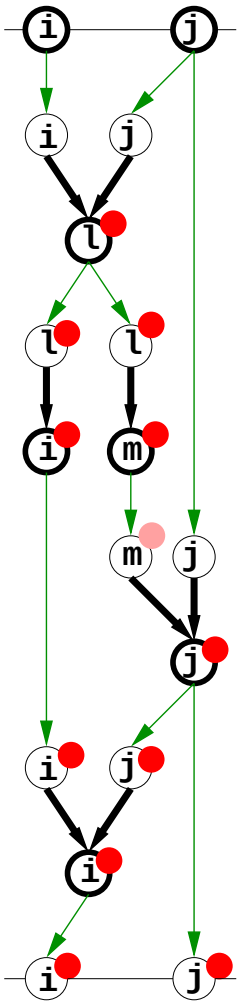
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3

end do
```

example cont. 8

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

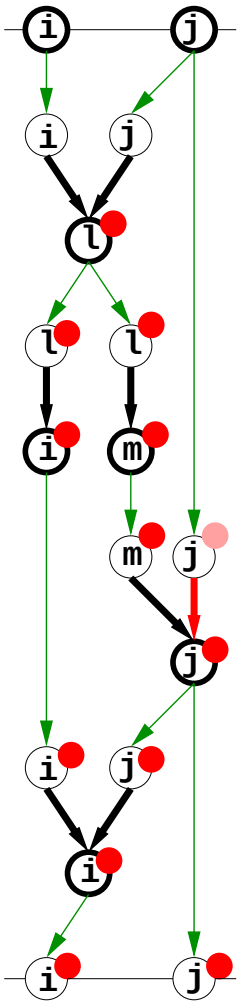
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3

end do
```

example cont. 9

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

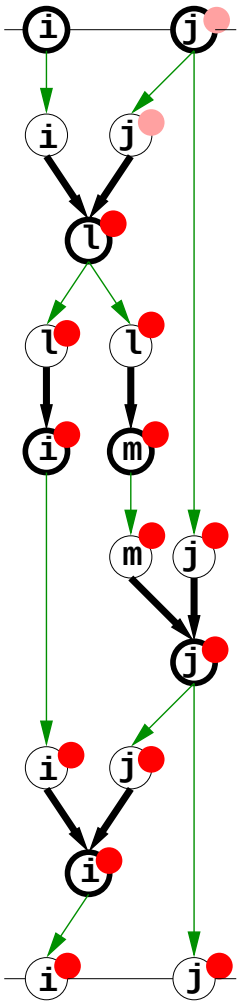
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3
   j = j-m-4

end do
```

example cont. 10

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

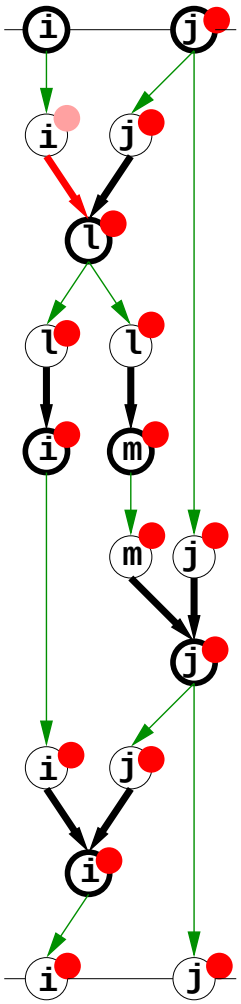
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3
   j = j-m-4

end do
```

example cont. 11

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



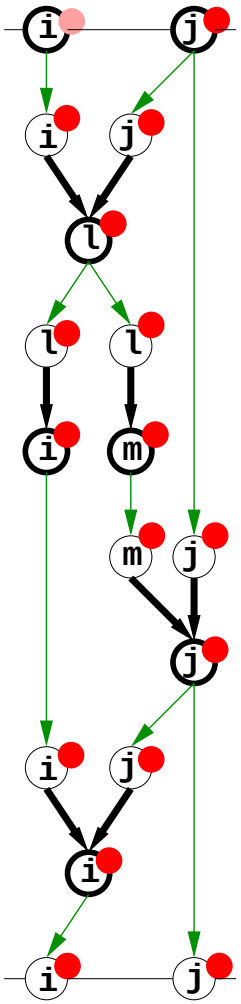
reverse:

```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3
   j = j-m-4
   i = l+j+3
end do
```

example cont. 12

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

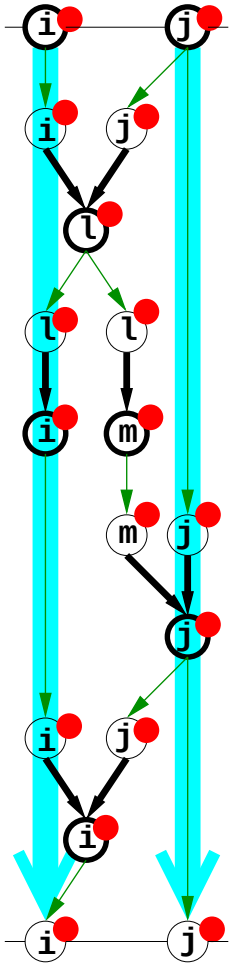
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3
   j = j-m-4
   i = l+j+3
end do
```

example cont. 13

reversal using a mix of inverse and forward computations

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l+2
   j = j+m+4
   i = i+j+1
end do
```



reverse:

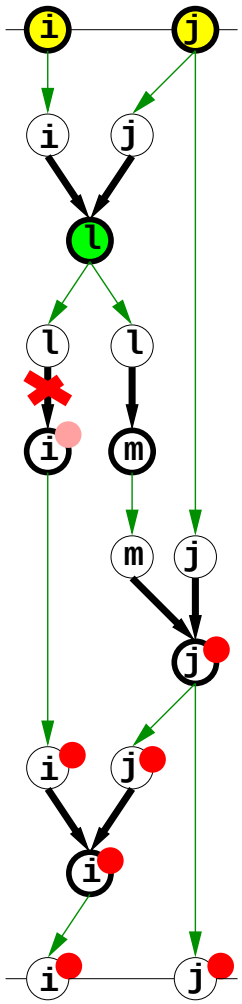
```
i=pop(); j=pop();
do while(...)
   i = i-j-1
   l = i-2
   m = l+3
   j = j-m-4
   i = l+j+3
end do
```

example cont. 14

same example with a non-invertible operation

forward:

```
do while(...)
   l = i-j-3
   m = l+3
   i = l%2
   j = j+m+4
   i = i+j+1
end do
```



- need to 'restore' a value
- minimize restore count to cover the comp. graph
- ongoing research with Hascoët/Naumann
- applicable to active computations (fewer benefits because of fewer cyclic dependencies)

```
do i=1,k
  a=ζ(a); y(i)=φ(x(i),a,...)
end do
```

reversed as

```
do i=k,1,-1
  a=ζ⁻¹(a); ....
end do
```

summary

- optimal adjoint variants is a local decision
- tool needs to be able to generate all variants/evaluate/select
- OpenAD started from store all corner, moves towards adding recompute variants
- other variations from checkpointing

some behind-the-scenes changes

- new code analysis version (including pointer analysis)
- canonicalization/postprocessor with common Python parser
- on the near term list:
  - missing intrinsics, 6/7 tensors
  - memory-hungry transformation step
  - array operations
  - application to FLASH code

upcoming events

end of May 2007: 5th European AD Workshop in Hatfield (UK)

AD 2008 : 5th International AD Conference
August 11 - 15 in Bonn (Germany)

- last one was in Chicago in 2004
- venue: Bonn/Aachen Center for Information Technology
- Monday noon - Friday noon
- 7 invited lectures
- single track presentations
- poster session

Wednesday afternoon & evening excursion by boat on the Rhine to the Loreley