Data-flow reversal and Garbage Collection

Laurent Hascoët

INRIA, France

MIAPbP Probabilistic Programming, June 2023, Munich, Germany

This is (Source-Transformation) AD

```
SUBROUTINE FOO(v1,    v2,    v4,    p1)

  REAL v1,v2,v3,v4,p1


  v3 = 2.0*v1 + 5.0

  v4 = v3 + p1*v2/v3
END
```

This is (Source-Transformation) AD

```
SUBROUTINE FOO(v1,v1d,v2,v2d,v4,v4d,p1)
  REAL v1d,v2d,v3d,v4d
  REAL v1,v2,v3,v4,p1

  v3d = 2.0*v1d
  v3 = 2.0*v1 + 5.0
  v4d = v3d + p1*(v2d*v3-v2*v3d)/(v3*v3)
  v4 = v3 + p1*v2/v3
END
```

Inserts differentiated instructions into FOO, automatically

Computes derivatives with machine accuracy

Outline

Programs are functions

See any (straight-line piece of) program $\mathtt{P}: \{I_1; I_2; \ldots I_p;\}$ as:

$$f : \mathbf{in} \in \mathbb{R}^m \rightarrow \mathbf{out} \in \mathbb{R}^n \qquad f = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

Define for short:

$$V_0 = \mathbf{in} \quad \text{and} \quad V_k = f_k(V_{k-1})$$

The chain rule yields:

$$f'(\mathbf{in}) = f'_p(V_{p-1}).f'_{p-1}(V_{p-2}).\ldots.f'_1(V_0)$$

In which order shall we multiply all these matrices?

Evaluate from the right or from the left?

We may start from the right (i.e. the inputs **in**) $\Rightarrow$ Tangent
$\Rightarrow$ start with a direction vector $\dot{\mathbf{in}}$, then progress leftwards:

$$\dot{\mathbf{out}} = f'(\mathbf{in}).\dot{\mathbf{in}} = f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0).\dot{\mathbf{in}}$$

We may start from the left (i.e. the outputs **out**) $\Rightarrow$ Adjoint
$\Rightarrow$ start with an weighting vector $\overline{\mathbf{out}}$, then progress rightwards:

$$\overline{\mathbf{in}} = \overline{\mathbf{out}}.f'(\mathbf{in}) = \overline{\mathbf{out}}.f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0)$$

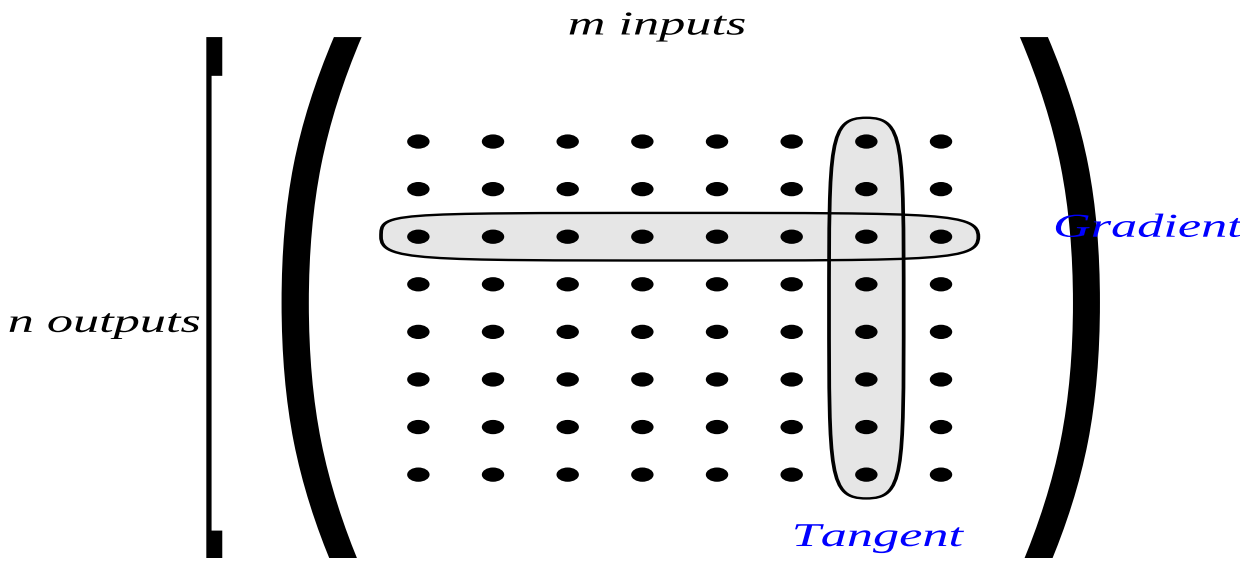
(for the full Jacobian, replace the start vectors by identity matrices)

Take the time to figure out the sizes and costs wrt sizes m and n

Full Jacobian with Tangent or Adjoint AD

$f \quad : \quad \mathbf{in} \in R^m \quad \rightarrow \quad \mathbf{out} \in R^n$

$\frac{\text{d out}}{\text{d in}} =$



- $\frac{\text{d out}}{\text{d in}}$ costs $m * 4? * \mathrm{P}$ using the tangent mode
  Good if $m <= n$
- $\frac{\text{d out}}{\text{d in}}$ costs $n * 4? * \mathrm{P}$ using the adjoint mode
  Good if $m >> n$ (e.g $n = 1$ for a gradient)

By the way: beware of control

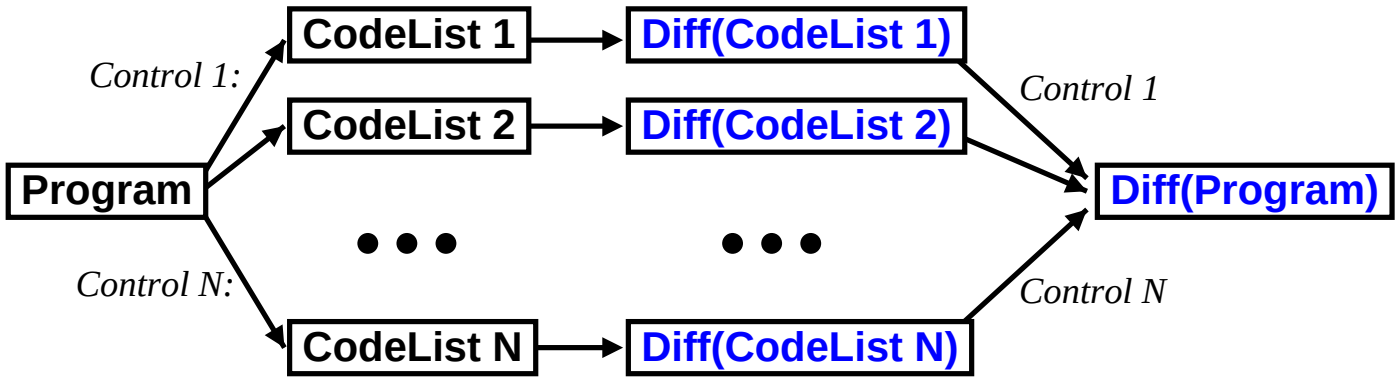
Function f must be differentiable,

but implementation may require control $\Rightarrow$ creates non-differentiability !

Freeze the current control:

$\Rightarrow$ the program becomes a simple sequence of instructions

$\Rightarrow$ AD differentiates these sequences:



$\Rightarrow$ and replaces them into the control.

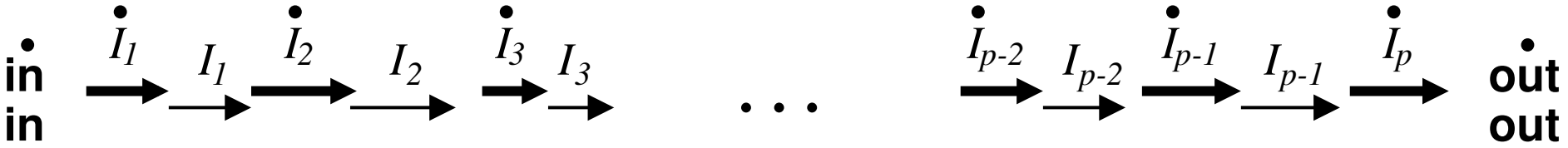
Caution: the diff program is only a piecewise diff !

$\Rightarrow$ see [Griewank] about the *Abs-Normal-Form*

Implementing Tangent AD

$$\dot{\mathbf{out}} = f'(\mathbf{in}).\dot{\mathbf{in}} = f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0).\dot{\mathbf{in}}$$

Implementation:



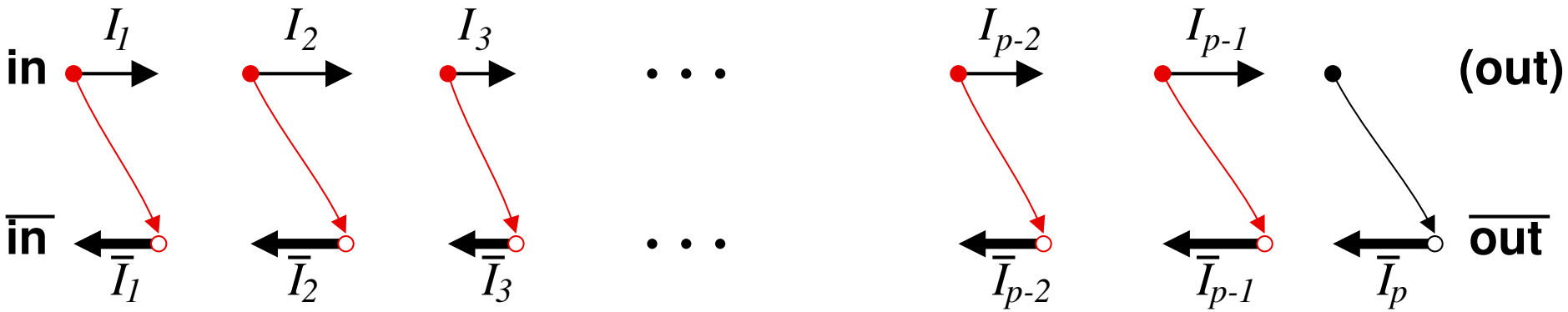
Tangent-diff instructions interleaved with the original instructions.

almost no problem...

Implementing Adjoint AD

$$\overline{\mathbf{in}} = \overline{\mathbf{out}}.f'(\mathbf{in}) = \overline{\mathbf{out}}.f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0)$$

Implementation:



Adjoint-diff instructions form the backward sweep.
There is a forward sweep and then the backward sweep.
Mechanism required to make the V_k available in reverse order.

This is hard, but worth the effort

Adjoint AD in a nutshell

Adjoint derivatives by Algorithmic Differentiation (AD):

- compute gradients of numerical models,
- from the models source program,
- more or less automatically,
- at a cost independant of #inputs,

...but there are serious challenges

By the way: Adjoint code is weird

Consider instruction I_k: `c := a*b` i.e. function:

$$f_k : \begin{array}{ccc} R^3 & \to & R^3 \\ \begin{pmatrix} a \\ b \\ c \end{pmatrix} & \mapsto & \begin{pmatrix} a \\ b \\ a*b \end{pmatrix} \end{array}$$

Its adjoint code must compute:

$$\begin{pmatrix} \overline{a} & \overline{b} & \overline{c} \end{pmatrix} := \begin{pmatrix} \overline{a} & \overline{b} & \overline{c} \end{pmatrix} \times f_k' == \begin{pmatrix} \overline{a} & \overline{b} & \overline{c} \end{pmatrix} \times \begin{pmatrix} 1 & & \\ & 1 & \\ b & a & 0 \end{pmatrix}$$

And therefore its adjoint "code" is:

```
ā := ā + b*c̄
b̄ := b̄ + a*c̄
c̄ := 0.0
```

This is not a problem: all you need is a tool

By the way: why the name "Adjoint AD"?

Code instructions can be seen as equality constraints *[Giles, Pironneau]*.

```
a := i1
b := i2
c := a*b
d := a*c
r := c + d
```

Adjoint AD →

```
d̄ := r̄
c̄ := r̄ + a*d̄
b̄ := a*c̄
ā := c*d̄ + b*c̄
```

↓ ?Lagrangian?

$$\mathcal{L} = \bar{r}(c+d-r)+\bar{d}(ac-d)+\bar{c}(ab-c)+\bar{b}(i_2-b)+\bar{a}(i_1-a)$$

↓

$$\frac{d\mathcal{L}}{dd} = 0 = \bar{r}-\bar{d}$$
$$\frac{d\mathcal{L}}{dc} = 0 = \bar{r}+a\bar{d}-\bar{c}$$
$$\frac{d\mathcal{L}}{db} = 0 = a\bar{c}-\bar{b}$$
$$\frac{d\mathcal{L}}{da} = 0 = c\bar{d}+b\bar{c}-\bar{a}$$

→

```
d̄ := r̄
c̄ := r̄ + a*d̄
b̄ := a*c̄
ā := c*d̄ + b*c̄
```

Outline

AD by Overloading, AD by Source-Transformation

Roughly, AD tools are based either on Source-Transformation, or on Operator-Overloading.

Overloading (available in F90, Object languages, ...) lets one redefine arithmetic operations to compute derivatives on the fly:
Change active `float, real` to `aDouble`, and link with a library that

- for Tangent: computes derivatives on `aDouble`'s
- for Adjoint: stores instructions on a "tape", for later backward derivative computation

Outline

Challenges of adjoint AD

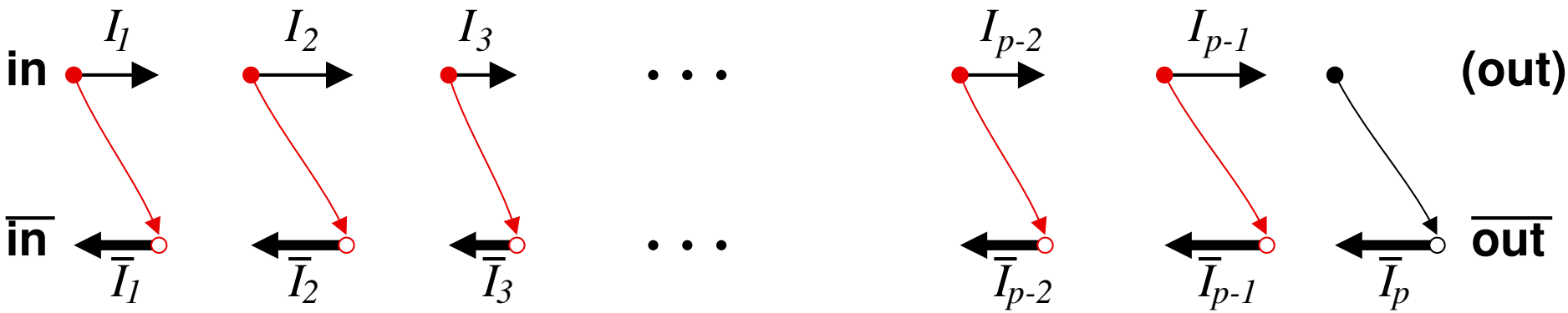
Gradients are propagated backwards,
using info from the (forward) primal code
$\Rightarrow$ Instruction flow reversal
$\Rightarrow$ Data flow reversal

There are many other challenges around AD:

- non-smoothness *[Griewank et al.]*
- stochastic or chaotic parts *[Wang]*
- higher derivatives (cost, size...) *[Walther, Wang, Pothen]*
- ...

Adjoint main difficulty: data flow reversal

$$\overline{\mathbf{in}} = f'^t(\mathbf{in}).\overline{\mathbf{out}} = f_1'^t(V_0) \ldots f_{p-1}'^t(V_{p-2}) \,.\, f_p'^t(V_{p-1}) \,.\, \overline{\mathbf{out}}$$



In most codes, V_0, V_1,... V_{p-1} successively overwrite one another. Most likely V_{p-2} is lost, overwritten by I_{p-1}, etc.

One can either store (our basic choice), or recompute
In practice, one always ends up using both!

In the sequel, data-flow reversal is based on storage
Recomputation only comes as an extra

See tool TAF/TAC++ for
data-flow reversal by recomputation

The memory challenge

The memory cost of storing intermediate values grows linearly with runtime.

Can we master memory consumption ?

- use every possible Data-Flow analysis
  $\rightarrow$ can gain 40 to 70%... still linear memory cost
- trade recomputation/storage ("Checkpointing")
  $\rightarrow$ achieves logarithmic growth
- exploit profitable situations, (math or algorithm) e.g.
  - Linear solvers
  - Parallel loops
  - Fixed-Point iterations

Outline

4 classic AD Data-Flow analyses

- **varied**:*[Fagan, Carle]*

  if current v depends on no "independent input", then $\overline{v}$ is useless

  $\Rightarrow$ slice out computation of $\overline{v}$
- **useful**:

  if current v influences no "dependent output", then $\overline{v}$ is zero

  $\Rightarrow$ propagate constant $\overline{v}$ and remove its initialization
- **diff-live**:

  if current v influences no useful derivative (may influence orig. result)

  $\Rightarrow$ slice out computation of v
- **TBR**:*[Naumann]*

  if current v not used in any derivative (e.g. only linear uses of v)

  $\Rightarrow$ slice out storage of v before it is overwritten

Diff-Live, TBR, Recompute

naïve

```
CALL PUSHINTEGER4(n)
n = ind1(i)
CALL PUSHREAL4(b(n))
b(n)=SIN(a(n))-b(n)
CALL PUSHREAL4(a(n))
a(n) = a(n) + x
CALL PUSHREAL4(c)
c = a(n)*b(n)
CALL PUSHREAL4(a(n))
a(n) = a(n)*a(n+1)
CALL PUSHINTEGER4(n)
n = ind2(i+2)
CALL PUSHREAL4(z(n))
z(n) = z(n) + c
CALL POPREAL4(z(n))
cb = zb(n)
CALL POPINTEGER4(n)
CALL POPREAL4(a(n))
ab(n+1) = ab(n+1)
    +a(n)*ab(n)
ab(n) = b(n)*cb
    +a(n+1)*ab(n)
CALL POPREAL4(c)
bb(n) = bb(n)
    +a(n)*cb
CALL POPREAL4(a(n))
xb = xb + ab(n)
CALL POPREAL4(b(n))
ab(n) = ab(n)
   +COS(a(n))*bb(n)
```

Diff-live

```
CALL PUSHINTEGER4(n)
n = ind1(i)
CALL PUSHREAL4(b(n))
b(n)=SIN(a(n))-b(n)
CALL PUSHREAL4(a(n))
a(n) = a(n) + x

CALL PUSHINTEGER4(n)
n = ind2(i+2)

cb = zb(n)
CALL POPINTEGER4(n)

ab(n+1) = ab(n+1)
    +a(n)*ab(n)
ab(n) = b(n)*cb
    +a(n+1)*ab(n)

bb(n) = bb(n)
    +a(n)*cb
CALL POPREAL4(a(n))
xb = xb + ab(n)
CALL POPREAL4(b(n))
ab(n) = ab(n)
   +COS(a(n))*bb(n)
```

TBR

```
n = ind1(i)

b(n)=SIN(a(n))-b(n)
CALL PUSHREAL4(a(n))
a(n) = a(n) + x

CALL PUSHINTEGER4(n)
n = ind2(i+2)

cb = zb(n)
CALL POPINTEGER4(n)

ab(n+1) = ab(n+1)
    +a(n)*ab(n)
ab(n) = b(n)*cb
    +a(n+1)*ab(n)

bb(n) = bb(n)
    +a(n)*cb
CALL POPREAL4(a(n))
xb = xb + ab(n)

ab(n) = ab(n)
   +COS(a(n))*bb(n)
```

Recompute

```
n = ind1(i)

b(n)=SIN(a(n))-b(n)
CALL PUSHREAL4(a(n))
a(n) = a(n) + x

n = ind2(i+2)

cb = zb(n)
n = ind1(i)

ab(n+1) = ab(n+1)
    +a(n)*ab(n)
ab(n) = b(n)*cb
    +a(n+1)*ab(n)

bb(n) = bb(n)
    +a(n)*cb
CALL POPREAL4(a(n))
xb = xb + ab(n)

ab(n) = ab(n)
   +COS(a(n))*bb(n)
```

Summary: good, but not sufficient

Adjoint data-flow analyses

- are classical compiler analyses/optims specialized for adjoint codes.
- bring substantial benefit
  - 20% to 50% in runtime
  - 40% to 70% in memory space

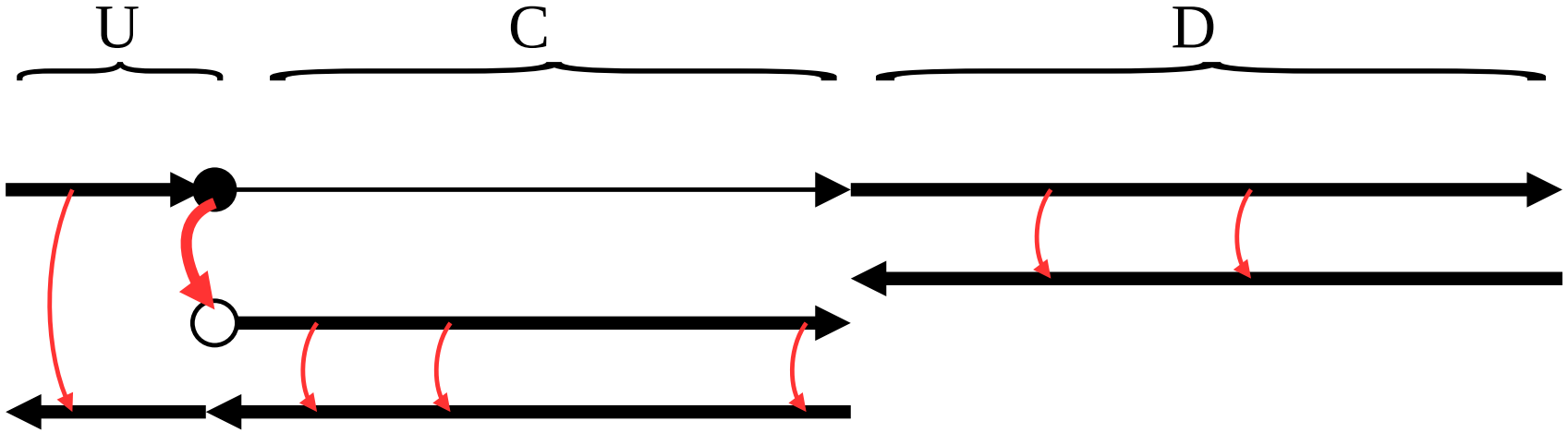
But memory still grows linearly with runtime

$\Rightarrow$ we need something else...

Outline

Trading recomputation (CPU) for storage (memory)

Checkpointing: elementary pattern

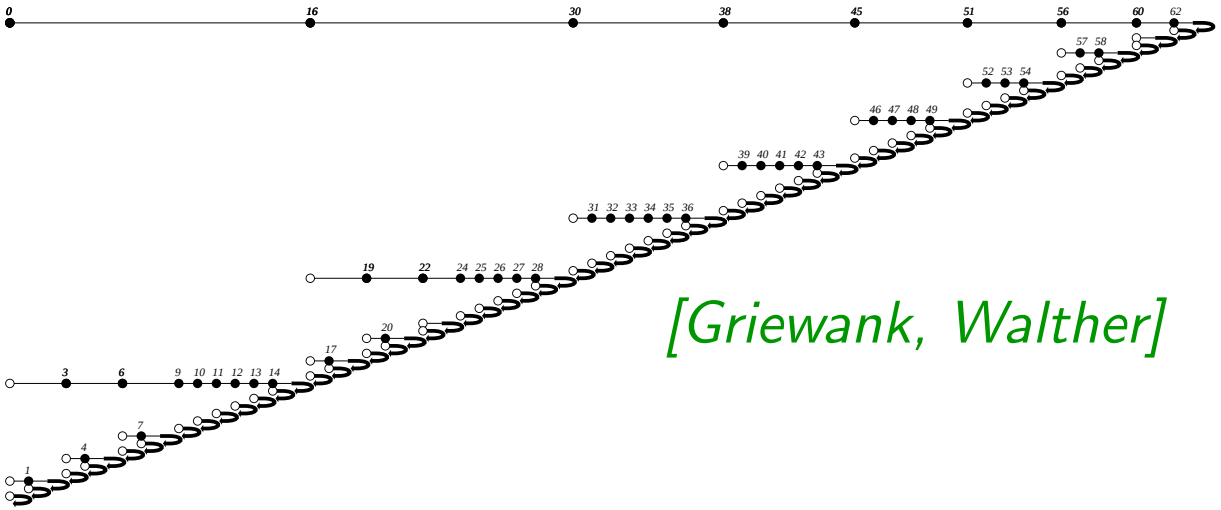


- reduces peak storage
- at the cost of duplicate execution
- also costs a memory "Snapshot", small enough:

$$\text{Snapshot} \subset \mathbf{use}(\overline{\mathtt{C}}) \cap \left(\mathbf{out}(\mathtt{C}) \cup \mathbf{out}(\overline{\mathtt{D}})\right)$$

Nesting checkpoints

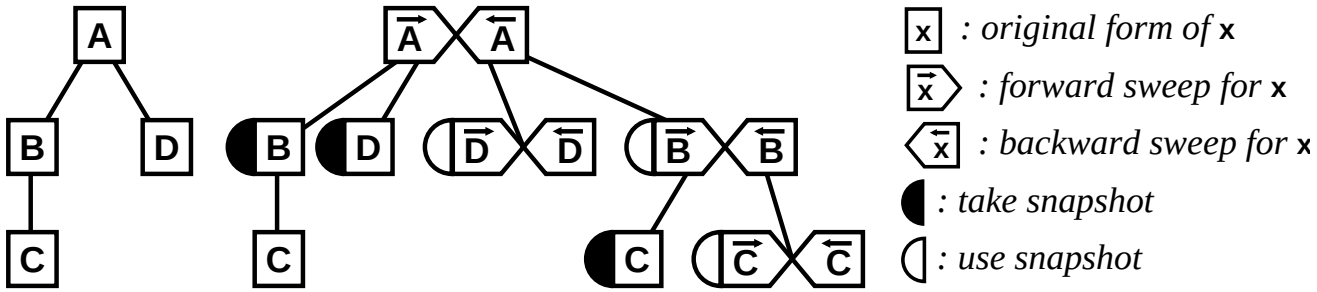
Checkpoints must be (carefully) nested.
Optimal nesting (binomial) exists for time-stepping loops:



- peak memory storage grows like log(runtime)
  execution duplication grows like log(runtime)
- in real life, storage is fixed to q snapshots,
  execution duplication grows like qth-root(runtime)

Checkpointing on calls

Nested checkpointing can be applied on procedure calls:



Not optimal(?), but still logarithmic if call tree is balanced.

Applies also to code sections that *could* be procedures.

A few limitations

- Checkpoints must respect code structure:
  - no checkpoint across procedures
  - no checkpoint across structured statements
  - *...well you could, but you need a flattened instruction tape*
- Checkpoints must contain both ends of system resources lifespan:

  `read`/`write`, `alloc`/`free`, `send`/`recv`, `isend`/`wait`...
- Checkpointed code must be reentrant

All in all, nested checkpointing is the answer

Outline

Profitable Situations

Take advantage of algorithmic or mathematic knowledge on parts of the code.

A selection:

- Adjoint of Linear Solvers
- Adjoint of Parallel Loops
- Adjoint of Fixed-Point iterations

Outline

Reading and writing variables

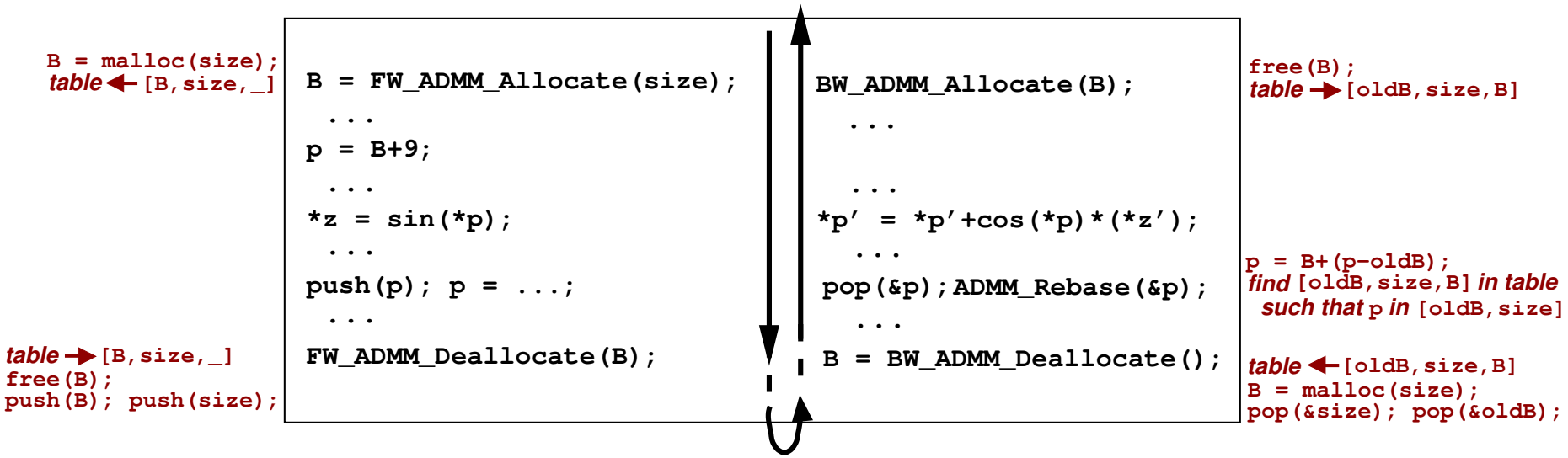
The adjoint of a use is an increment
The adjoint of an increment is a use

primal	**adjoint**
`... = ... x ...`	`xb = xb + ...`
`s = s + 2.1*x`	`xb = xb + 2.1*sb`

Assuming increments are atomic, they are independent
$\Rightarrow$ The adjoint of a parallel loop is (almost) a parallel loop

Dynamic memory

The adjoint of a `malloc` is a `free`
The adjoint of a `free` is a `malloc`



Parallel collective operations

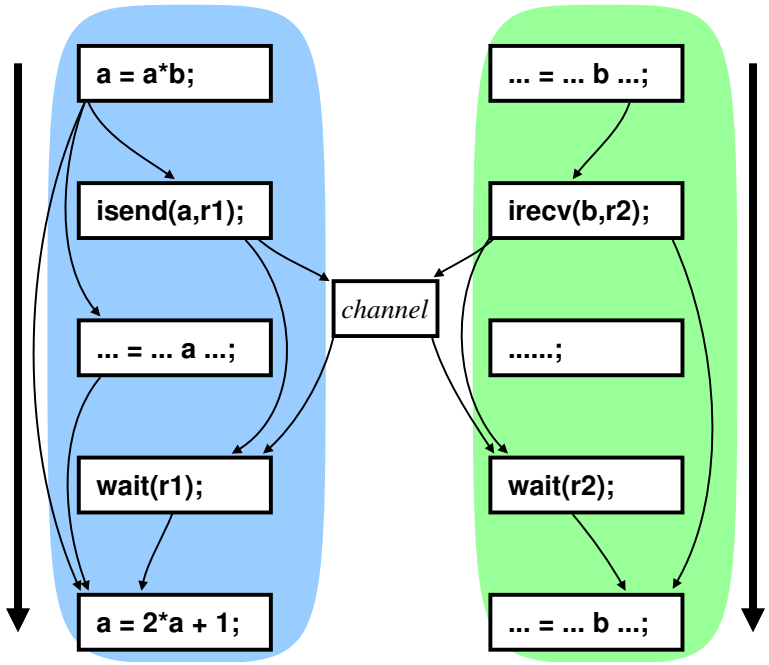
The adjoint of a sum is a spread
The adjoint of a spread is a sum

The adjoint of a `MPI_Bcast` is a (SUM)`MPI_Reduce`
The adjoint of a (SUM)`MPI_Reduce` is a `MPI_Bcast`
The adjoint of a `MPI_Gather` is a `MPI_Scatter`
The adjoint of a `MPI_Scatter` is a `MPI_Gather`

Message Passing

The adjoint of a SEND is a RECEIVE
The adjoint of a RECEIVE is a SEND

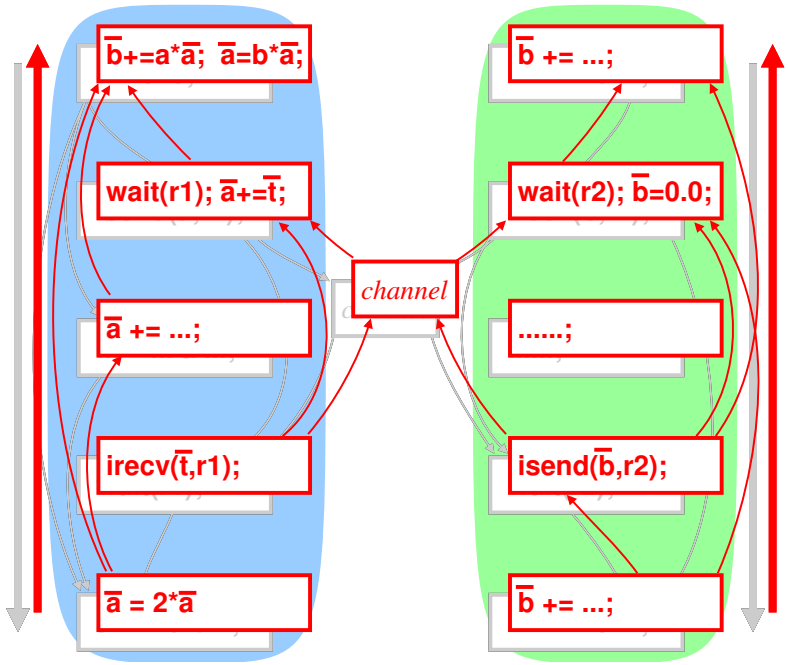
The adjoint of a `MPI_Isend/MPI_Wait` is a `MPI_Irecv/MPI_Wait`
The adjoint of a `MPI_Irecv/MPI_Wait` is a `MPI_Isend/MPI_Wait`



Message Passing

The adjoint of a SEND is a RECEIVE
The adjoint of a RECEIVE is a SEND

The adjoint of a `MPI_Isend/MPI_Wait` is a `MPI_Irecv/MPI_Wait`
The adjoint of a `MPI_Irecv/MPI_Wait` is a `MPI_Isend/MPI_Wait`



⇒ Good news: adjoint AD introduces no deadlock

Outline

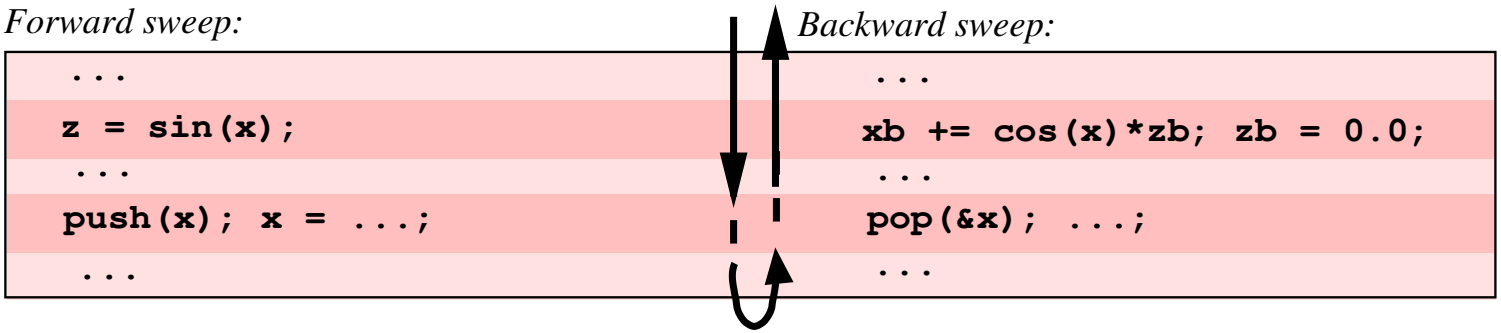
A few performance examples

	$n \rightarrow m$	**tangent** R_t	**adjoint** R_a	**peak** (Mb)	**traffic** (Mb)
uns2d *(2,000*F77)*	14000 → 3	**2.4**	**5.9**	**241**	1243
nsc2ke *(3,500*F77)*	1602 → 5607	**2.4**	**16.2**	**168**	2806
lidar *(330*F90)*	37 → 37	**1.1**	**2.0**	**11**	11
nemo *(55,000*F90)*	9100 → 1	**2.0**	**6.5**	**1591**	85203
gyre *(21,000*F90)*	21824 → 1	**1.9**	**7.9**	**481**	48602
winnie *(3,700*F90)*	3 → 1	**1.7**	**5.9**	**421**	614
stics *(17,000*F77)*	739 → 1467	**2.4**	**3.9**	**155**	186
smac-sail *(3,500*F77)*	1321 → 7801	**1.0**	**3.1**	**2**	21
traces *(19,800*F90)*	8 → 1	**1.3**	**3.8**	**159**	4390
mit-gcm *(258,225*F77)*	4704 → 1	**2.0**	**6.6**	**260**	5709
alif *(6,755*C)*	1413 → 1	**1.6**	**4.3**	**729**	

Outline

Data-flow reversal

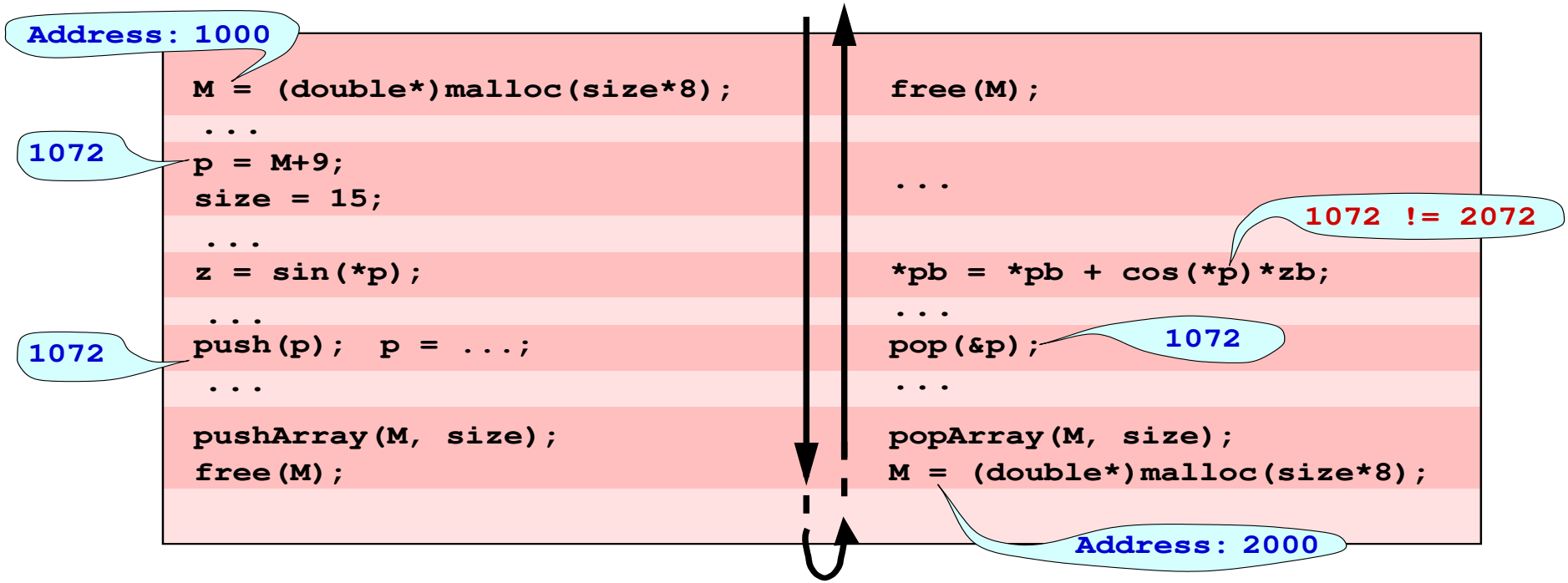
Source-transformation adjoint AD reverses control-flow and data-flow
$\Rightarrow$ requires undoing and restoring things



Must in particular undo deallocations and restore addresses

A well-known cause for trouble...

Trouble when memory is dynamic



- undoing deallocation generally returns a different chunk
- a (re)stored address offsets from the chunk base address
- the restored address has no meaning wrt the reallocated chunk

What can we do?

Possible fixes, possibly with limitations:

- Inhibit deallocations
- Rewrite allocation manager
- Rebase addresses
- Assign and track chunk identifiers

Also use profitable but non-general answers:

- Detect adjoint-dead addresses
- Recompute addresses

GC brings extra constraints

Garbage Collection (GC) in short:

- source program does not care to deallocate explicitly
- runtime maintains a reference count to allocated chunks
- after a memory chunk's reference count goes down to zero, deallocation may happen at any time, automatically, silently.

Annoying consequences:

- no addresses/pointers (would ruin reference counts)
- deallocation (and re-allocation) have no visible location in code

What options are we left with?

- Inhibit deallocations? costly. Looses benfits of dynamic memory.
- Rewrite allocation manager? you don't want to rewrite GC
- Rebase addresses? there are no addresses any more!
- Assign and track chunk identifiers Let's try that...

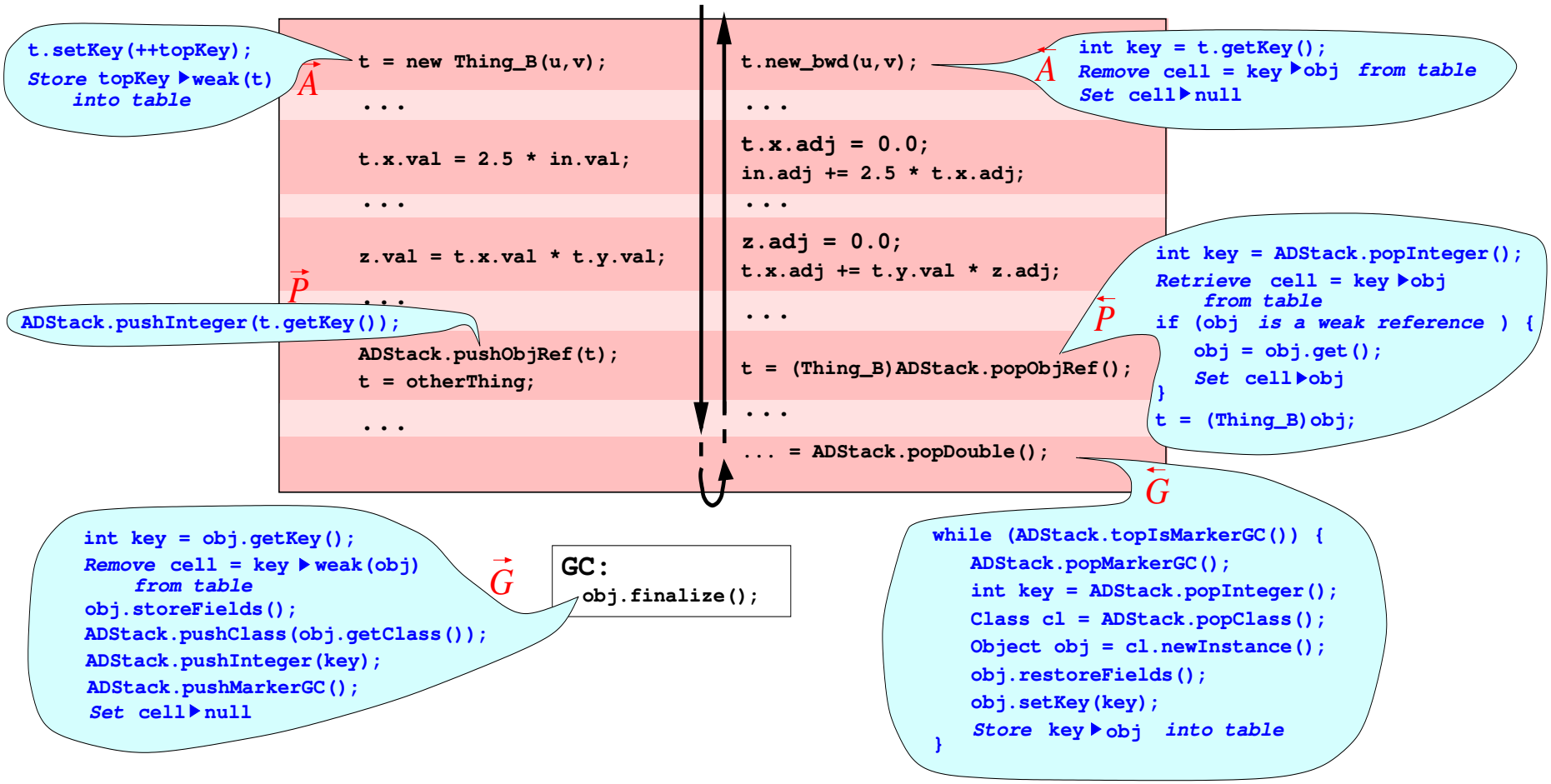
The proposed strategy

- Assign a unique integer pseudo-address to each (run-time) allocation
- Attach this pseudo-address to the created Object
- Push this pseudo-address every time necessary.
- If GC deallocates Object, use the finalizer.

... and in the backward sweep:

- Check the stack for reverse-GC marker.
- Re-allocate Object together with pseudo-address
- Every time a pseudo-address is popped, retrieve its Object instead

Proposed strategy sketch



What are the overheads?

All this definitely has a cost:

- an extra integer field on each Object
- a table lookup of the key, on each pop
- a finalizer call during GC
- a check for reverse-GC marker before each pop

Implementation

- Selected application language: Java
- Perform adjoint ST-AD by hand, but acting like an AD tool. Apply the proposed strategy as mechanically as possible.
- All Objects must implement `interface ADRestorable`.

Experiment

- Java application code: 2D incompressible Navier-Stokes solver (J. Scollay, 2018) `https://github.com/deltabrot/fluid-dynamics`.
- Adapt to allow variable grid size and time steps. Add a scalar cost function. Differentiate by hand.
- Rewrite the exact same code in Fortran90, with `ALLOCATE`s, `DEALLOCATE`s, pointers, but no GC. Differentiate it with Tapenade.
- Likewise rewrite the exact same code in C, with `malloc`s, `free`s, and pointers. Differentiate with Tapenade.
- Likewise rewrite the exact same code in C++. Differentiate with Adol-C (thanks Andrea!)

Validation

Fortran and C equivalent implementations differentiated with Tapenade:

	F90/Tapenade	**C/Tapenade**
Finite Diff.	1.852580*7*	1.852580*7*
Tangent AD	1.8525805085863*813*	1.8525805085863*820*
Reverse AD	1.8525805085863*818*	1.8525805085863*822*

Java implementation differentiated by hand.

C++ implementation differentiated with Adol-C:

	Java/hand-written	**C(++)/Adol-C**
Finite Diff.	1.8525807*021*	1.852580*7*
Tangent AD	1.8525805085863*813*	1.852580508586380
Reverse AD	1.8525805085863*807*	1.852580508586379

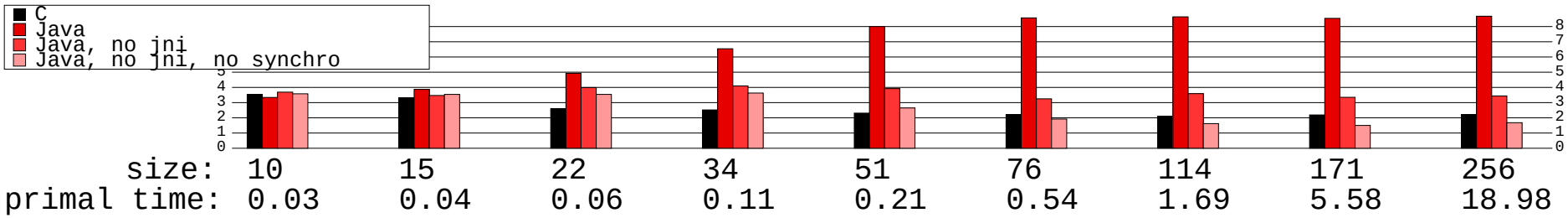
Time and memory performance

	F90/tapenade	**C/tapenade**	**Java/hand**	**C(++)/Adol-C**
Primal	0.125	0.127	0.203	3.09 *(with taping)*
Tgt	0.133 ($\times$1.06)	0.140 ($\times$1.10)	0.301 ($\times$1.48)	0.90
Adj	0.243 ($\times$1.93)	0.266 ($\times$2.09)	1.575 ($\times$7.76)	2.30
Peak	107 Mb.	107 Mb.	107 Mb.	1354 Mb.

Time overhead analysis

Two main causes identified for the time overhead:

- jni calls for the `push` and `pop` calls.
- Synchronization, against the danger of asynchronous GC.



- Most of the overhead comes from jni.
- Synchronization has a cost, but tolerable.
- Remaining overhead barely noticeable.

Summary

- Adjoint ST-AD can be extended to languages with GC
- ... at a tolerable cost wrt efficiency
- ... at a significant cost in sophistication
- limitations: finalize, complexity

Outlook:

- not implemented in an AD tool yet.
- looks like a step towards blending OO-AD into ST-AD.

Outlook, if one had infinite time

- Continue promoting applications of AD. Sci. Comp. is still a big user.
- Overcome competition between OO-AD and ST-AD, and ST-AD at different levels (source, IR, LLVM, assembly...). Target more new languages.
- Continue improving adjoint AD model.
- Continue address problematic constructs (Dyn. Mem., Objects, parallel...)
- Higher-order derivatives, easier, more efficient
- Similar targets: sub-gradients, ABS-normal form, intervals, McCormick relaxations...
- Reformulate AD, validate, prove, track AD pitfalls, connect with other domains.

Thank you for your attention!