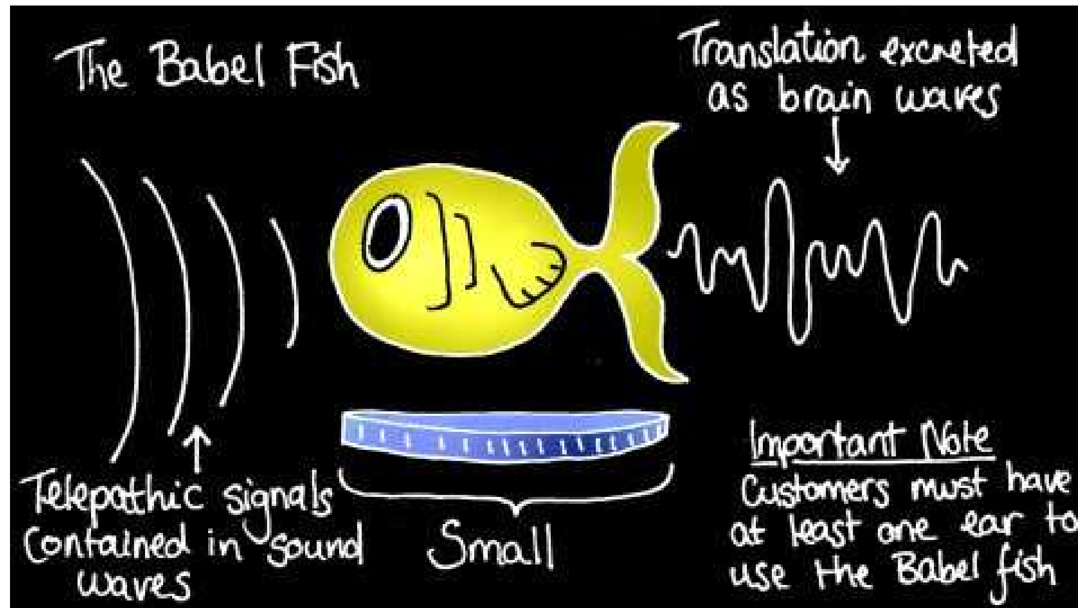
Note for the website version: This is the babel fish!

©somebody on the web

Look for its insightful blue translations!

Jean, despite his name, quite lamentably does not speak French! He won't even attempt to pronounce things. He has been trying to learn but it's nothing to speak of (yet).



©somebody else on the web or may be the same person

Babel Fish:

"Jean, en dbépit de son nom, tout à fait lamentably ne parle pas français! Il n'essayera pas même de prononcer des choses. Il avait essayé d'apprendre mais il n'est rien à parler de (pourtant)."

Automatic checkpoints and adaptive reversal schemes

(Points de contrôle automatiques et arrangements adaptatifs d'inversion)

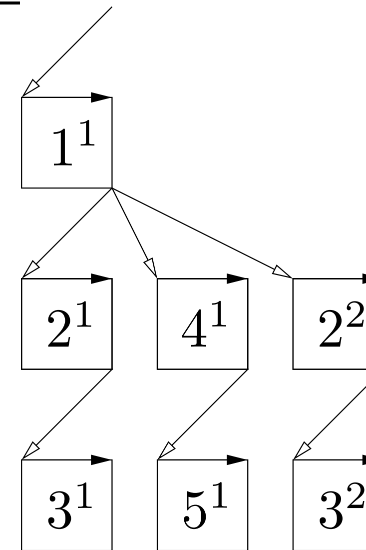
J. Utke

- Merci des poissons de Babel!
- thanks to Uwe and Michelle
- keep options for OpenAD extensions
- *automatic* checkpointing
- subroutine argument and result checkpointing
- semi-automatic checkpointing with hints
- use of OpenAnalysis
- consider Fortran and C++

the "easy" part

PSfrag replacements

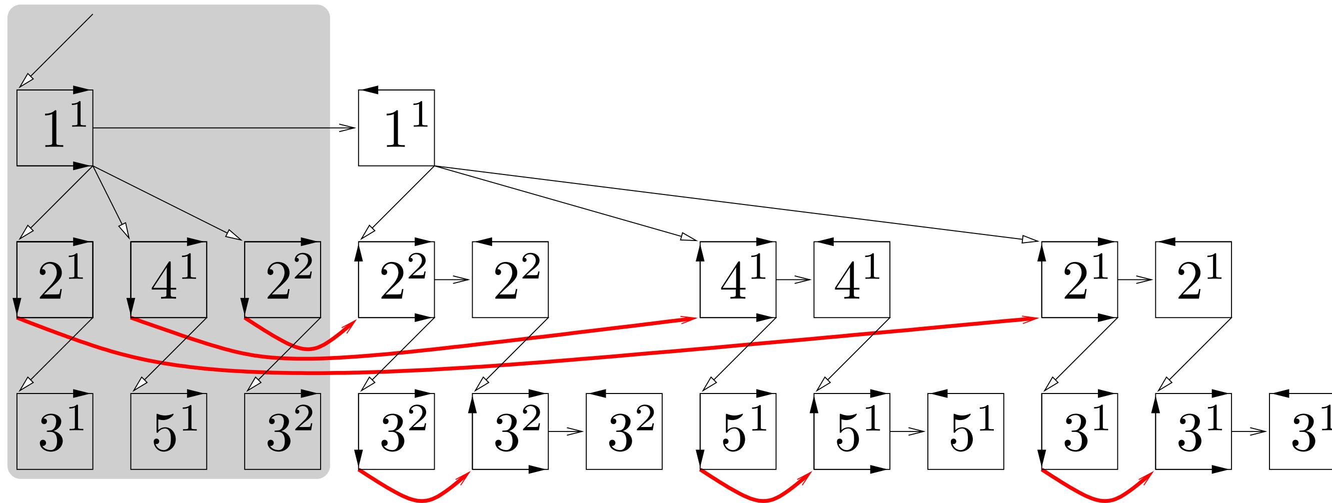
```
subroutine 1
  call 2; ...  call 4; ...  call 2;
end subroutine 1
subroutine 2
   call 3
end subroutine 2
subroutine 4
  call 5
end subroutine 4
```



- What do argument checkpointing for subroutines consist of?
- arguments, references to global variables
- OpenAnalysis provides *side-effect analysis*
- we ask for four sets: `ModLocal` $\subseteq$ `Mod`, `ReadLocal` $\subseteq$ `Read`
- What do these sets consist of? *Variable references!*

ag replacements

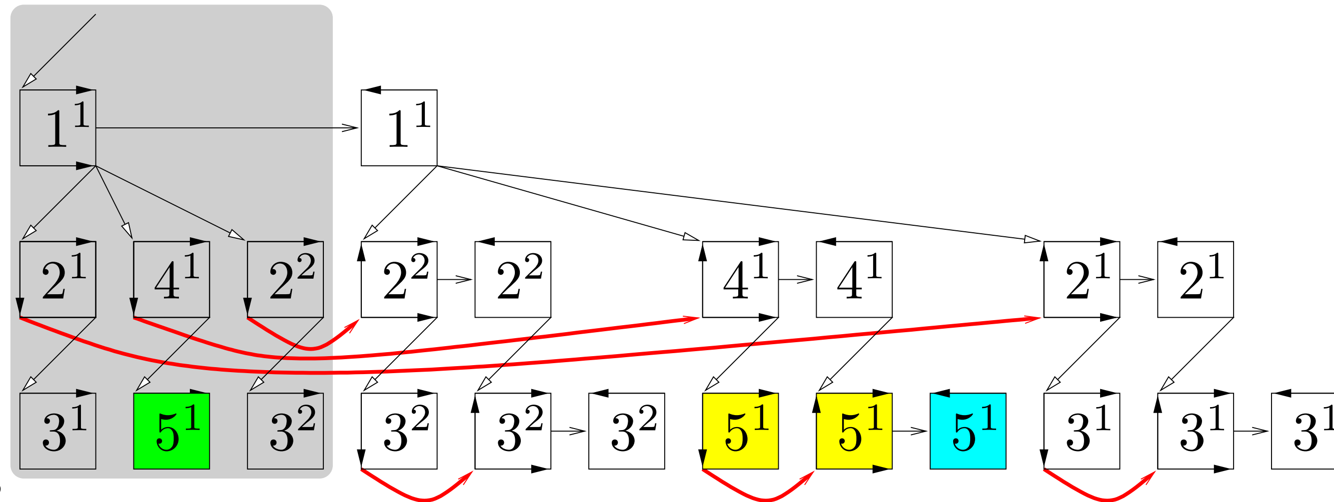
all set for joint mode



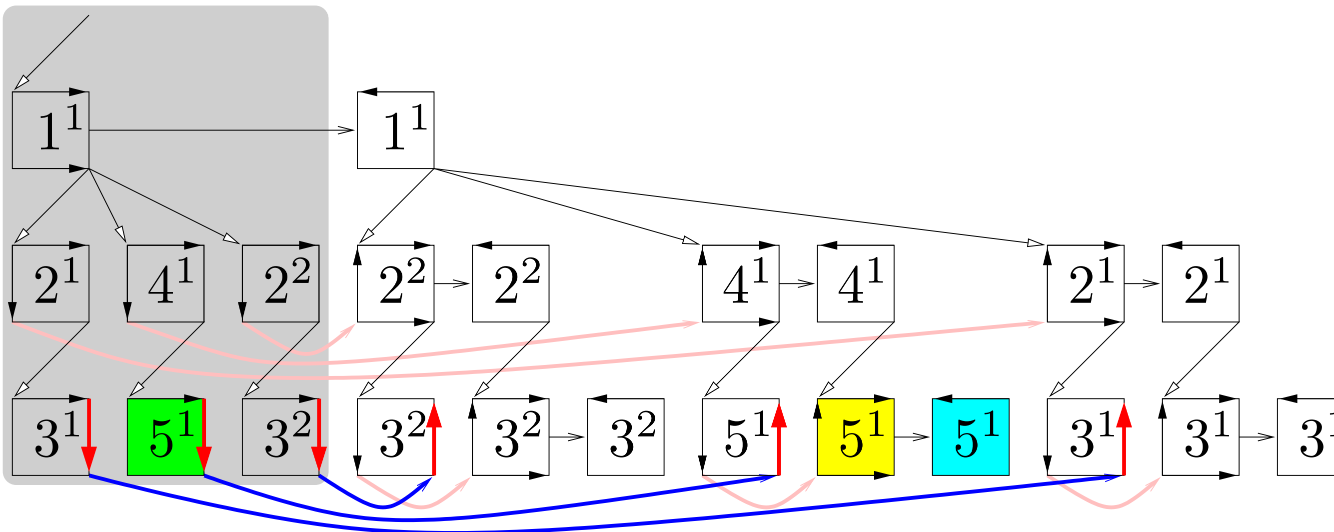
- we get away with a stack to store checkpoints
  (nous partons avec une pile pour stocker des points de contrôle)
- What about result checkpointing?

repeated evaluations for deep call stacks

ag replacements



ag replacements

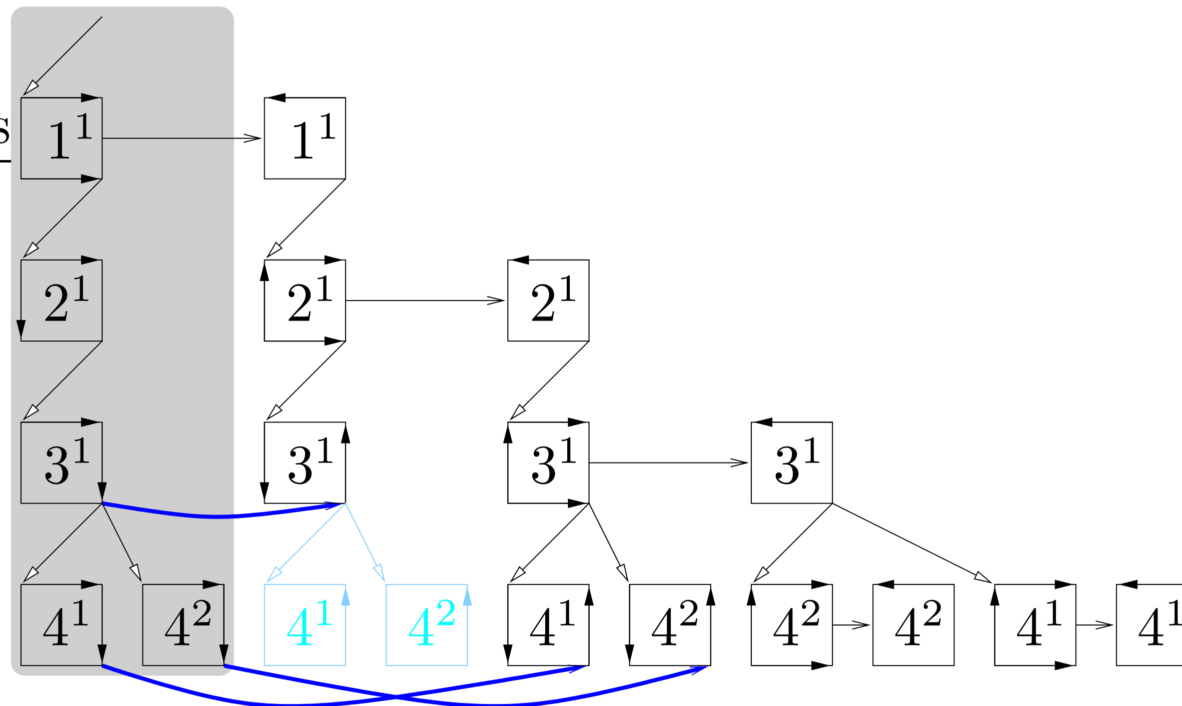


The reevaluation count is reduced but we lose stack storage.

(Le compte de réévaluation est réduit mais nous perdons le stockage de pile.)

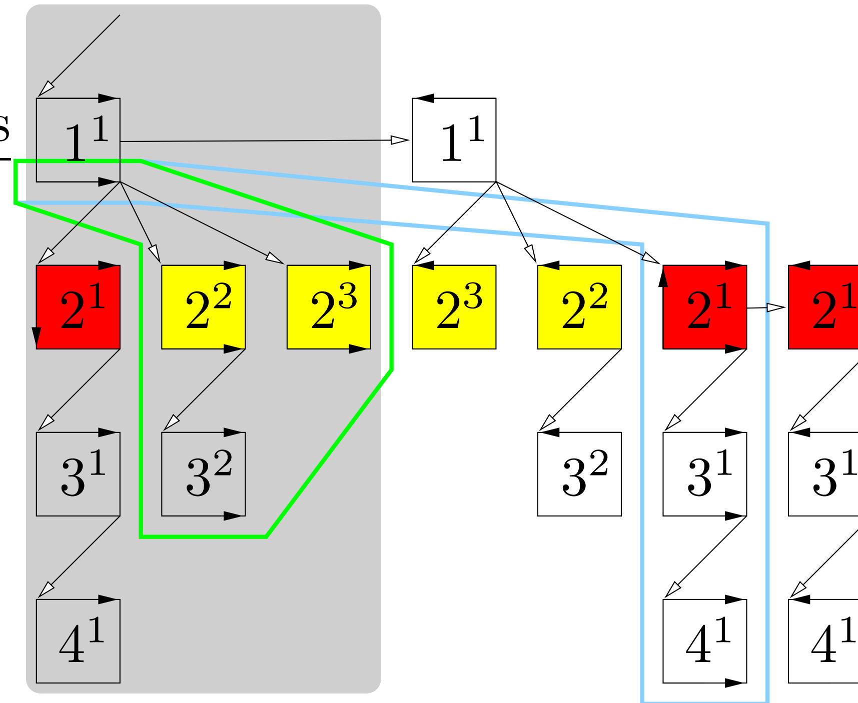
one more layer

Sfrag replacements



- a more suitable storage format is the *dynamic call tree*
- it is required by general reversal schemes, where there is no fixed reversal mode per subroutine
- for instance, "shallow" parts of the call tree need less tape than joint mode requires for the "deep" parts (in subroutine units)

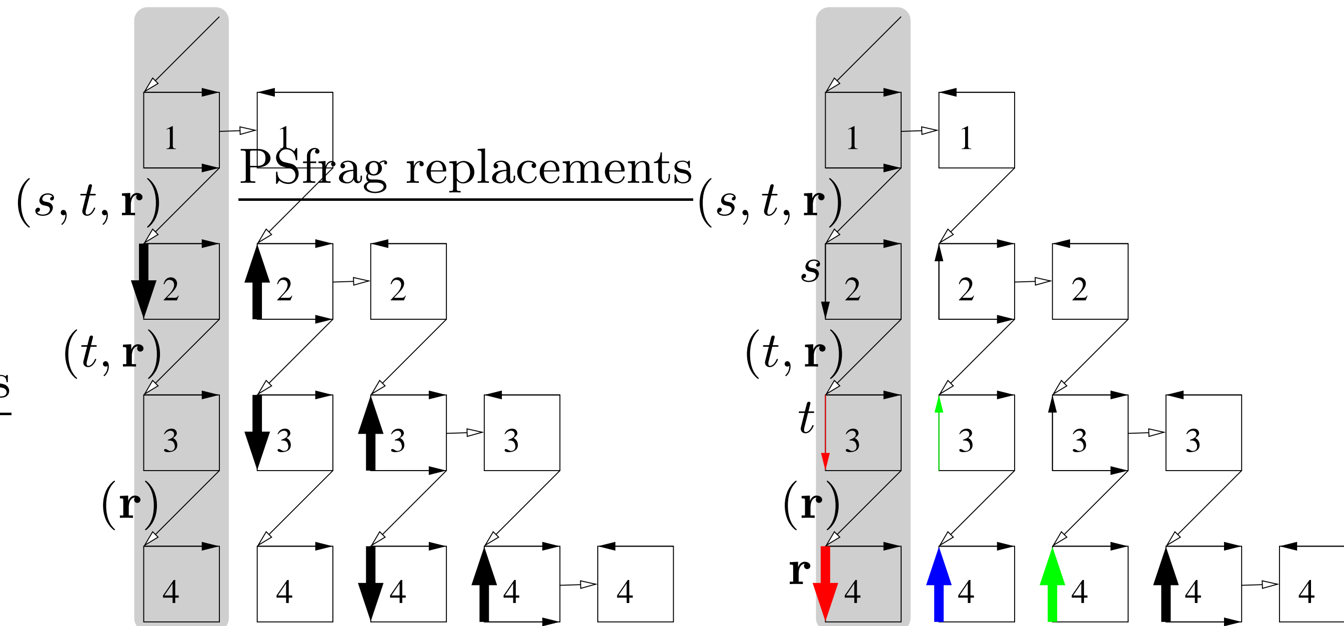
general reversal example



- we have 4 tape units
- 2^2 and2^3 behave like split, 2^1 behaves like joint
- How do we control the behavior?
- runtime estimates for checkpoint/tape size and recomputation effort → derive reversal scheme according to memory/runtime limits as dynamic call tree

reducing the checkpoints?

- always $\texttt{Read}_{callee} \subseteq \texttt{Read}_{caller}$
- multiple writes of $x \notin \texttt{ReadLocal}$
- can store only $x \in \texttt{ReadLocal}$ (except in callers whose callees don't store anything)

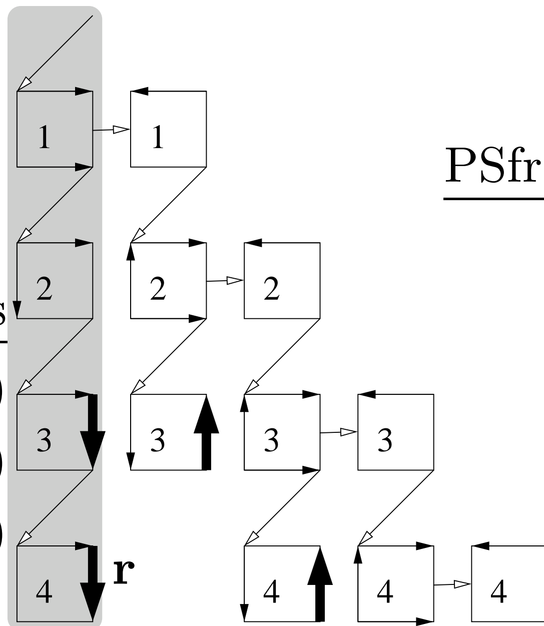


(**r** is 'big')

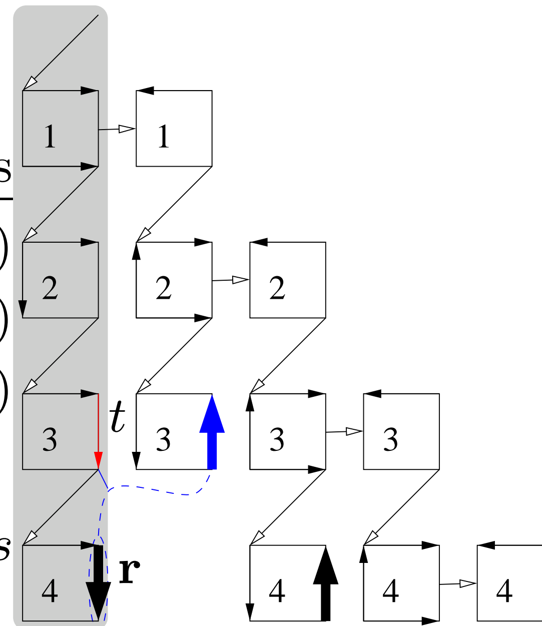
- loose stack format; same storage requirements;
- same number of ('big') reads; fewer 'big' writes.
- How about result checkpoints?

result checkpoints

- always $\texttt{Mod}_{callee} \subseteq \texttt{Mod}_{caller}$
- multiple writes and simultaneous representations of all $y \notin \texttt{ModLocal}$
- can store only $y \in \texttt{ModLocal}$ (except in callers whose callees don't store anything)



PSfrag replacements
$(s, t, \mathbf{r})$
$(t, \mathbf{r})$
$(\mathbf{r})$



($\mathbf{r}$ is 'big')

- now 3's result restore has to traverse the hierarchy to be complete
- but this isn't so bad since we have the dynamic call tree anyway

(mais ce n'est pas aussi mauvais puisque nous avons l'arbre dynamique d'appel de toute façon)

What did you say you store?

I said *variable references* !

- `v, *v_p, V[i], V` etc. works ok for cases with "fixed" addresses
- doesn't work if `i` in `V[i]` is computed in the code
- store `V` instead
- subroutine arguments with user defined types require *serialization*

```
struct S{ double d; int i;};
foo (S s){ ...checkpoint(s);...};
```

- should serialization follow pointers/references? think linked list vs. const reference

```
struct S{ double d; S* n;};
foo (S& s){ ...while (s.n) { x=bar(s.d); s=*(s.n);}...};
```

- "checkpoint on read"

```
foo(S& s){ ...while (s.n) {
  checkpoint(s.d); x=bar(s.d); s=*(s.n);
}...};
```

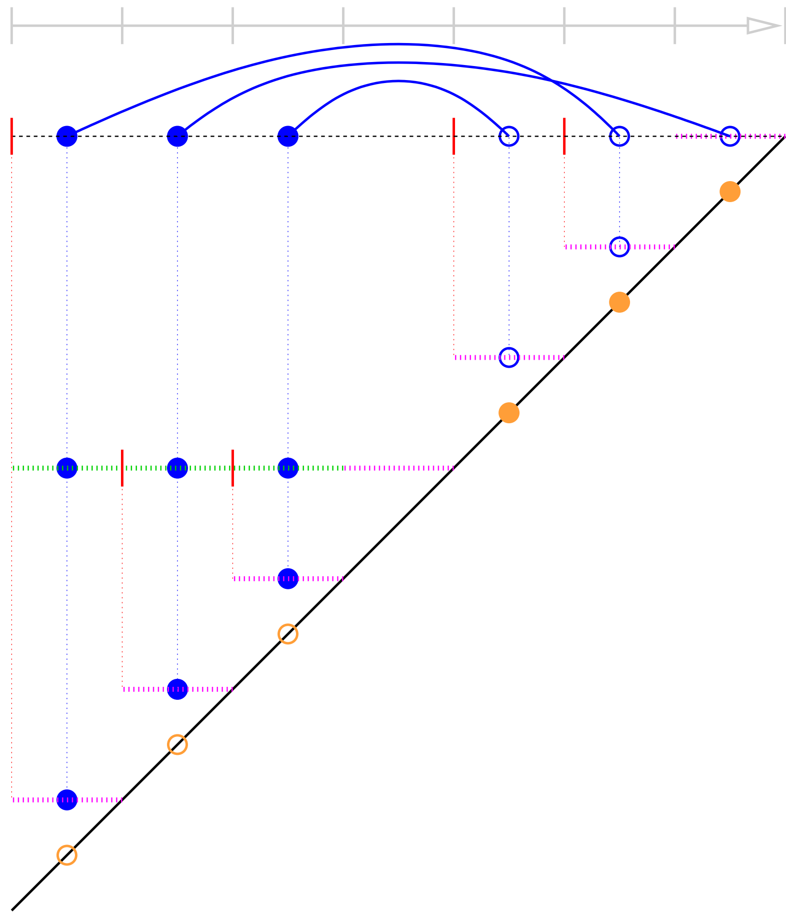
...but

- multiple uses of `s.d` → checkpoint on *first* read
- similar to deciding if `V[i]` loop reads the same data as a `V[j]` loop

```
for (i=0;i<n;i+=2) {
  ...V[i] ...
}
for (j=1;j<n;j+=2) {
  ...V[j] ...
}
```

- → array section analysis (or remember addresses along with values but this is expensive)
- result checkpoints don't have the "restore mixed with subroutine code" option
- they could be stored with (stack) addresses
- heap addresses?

dynamic memory 2nd



- dynamic memory 1st was at Hatfield in the context of taping
- similar issues for checkpointing
- for taping:
  - possible option: don't do anything for allocations in the reverse sweep
  - or reverse allocations/deallocations and map
- for checkpointing:
  - no obvious de/allocation pairs
  - ignore allocations
  - instead keep addresses in the checkpoint and restore
  - address assignments are part of `ModLocal`
  - scope does not fit checkpoints
  - consider not just memory but any *resource*

Oh boy, that's a whole new can of worms!

Le garçon d'Oh,

celui est un nouveau bidon entier de vers !