Selective Pretenuring

And about allocation sites

Sebastian JORDAN MONTANO (1)
sebastian.jordan@inria.fr
jordanmontt

1. Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL

ESUG ’25, July 2025, Gdańsk, Poland

Hey, it’s me: Sebastian

* 100% real no fake

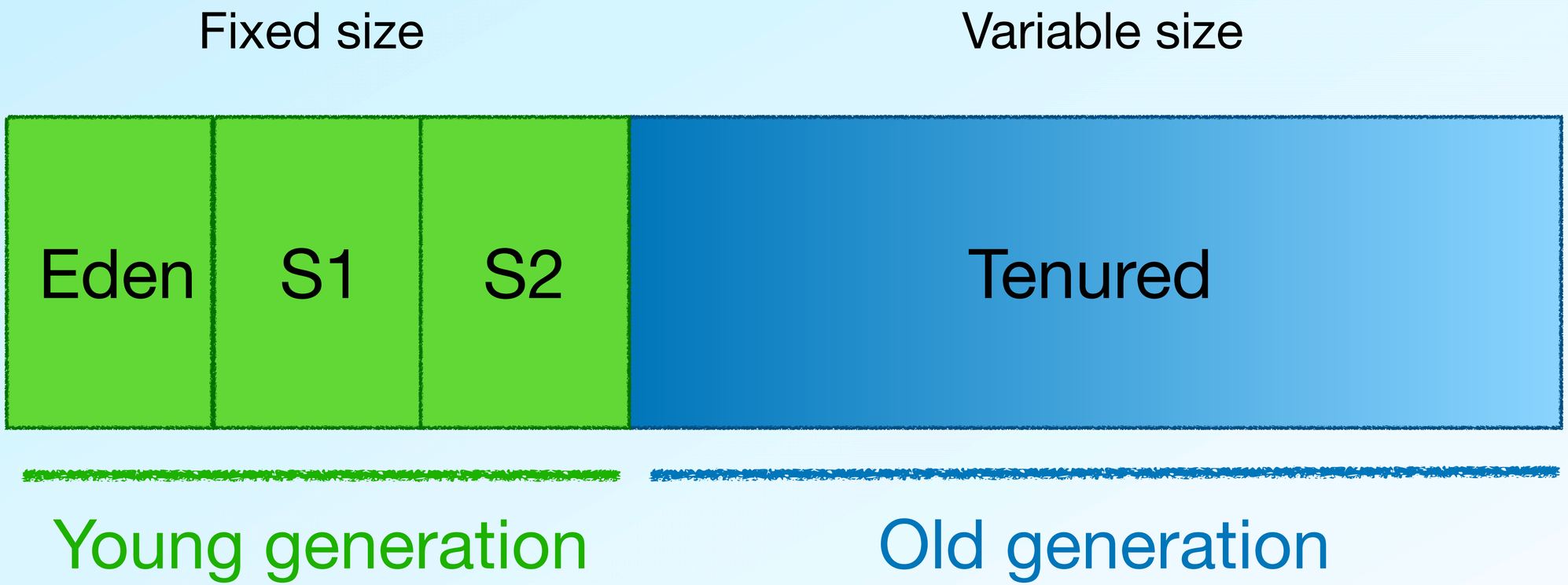
- Software Engineer & PhD student at INRIA Lille
- Open-source contributor since 2020
- Professional developper since 2017
- Passionate about languages – speak Spanish, French, English, and learning Italian :D
- Check me out on  at `jordanmontt`
- I’ll be looking for a job next year 👀

- King Crimson = best progressive rock group 🎸🎼

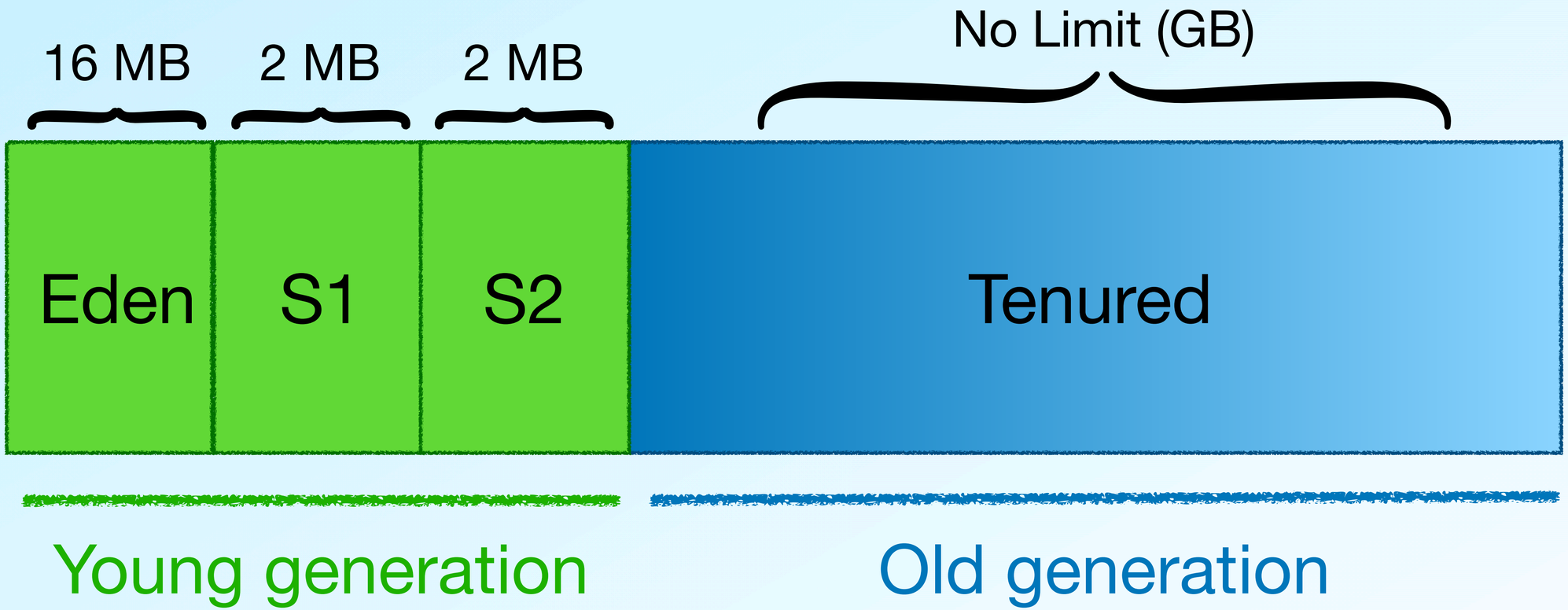
I’m doing a PhD @ INRIA - France

Memory profilers and safe code instrumentation for OOP

Pharo’s generational garbage collector



Default parameters (some)*



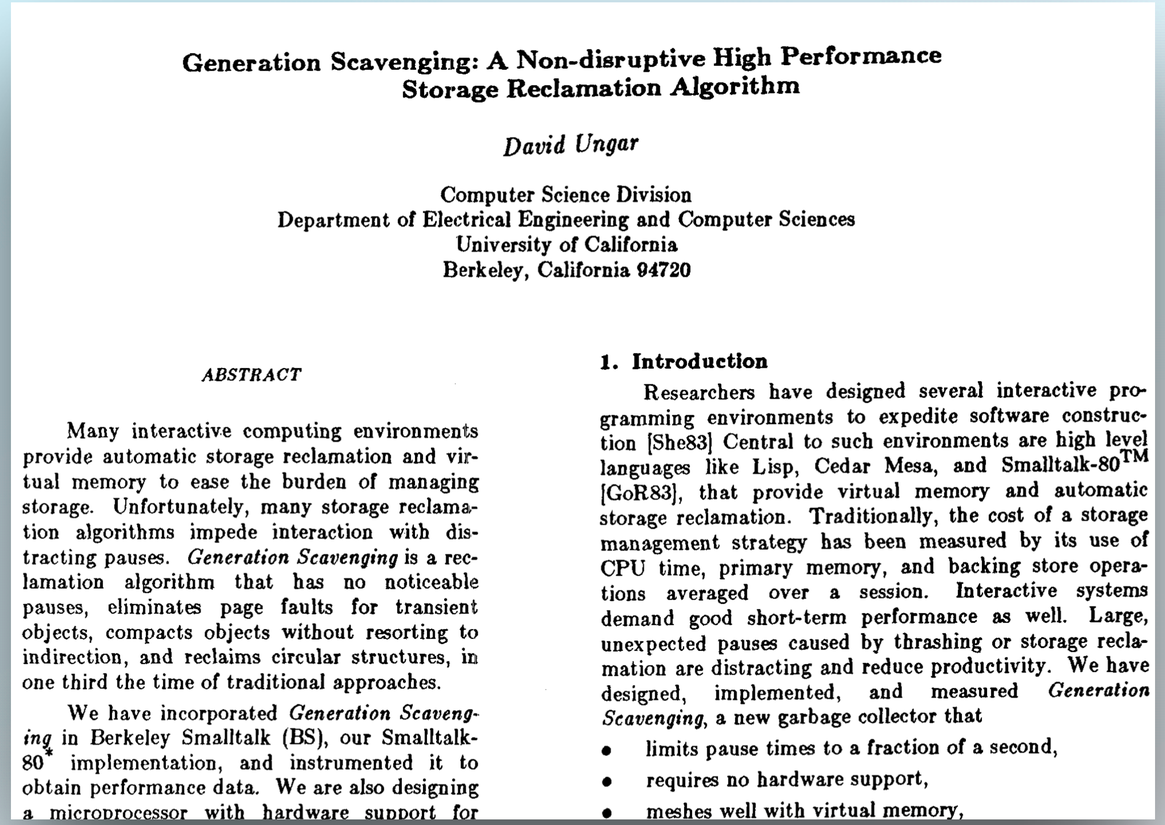
* for Pharo 13

Why are you talking about this?

Most objects die young ☠️🐣🔫

Weak generational hypothesis

Generation scavenging: A non-disruptive high performance storage reclamation algorithm, Ungar, 1984



Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm

David Ungar

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

ABSTRACT

Many interactive computing environments provide automatic storage reclamation and virtual memory to ease the burden of managing storage. Unfortunately, many storage reclamation algorithms impede interaction with distracting pauses. *Generation Scavenging* is a reclamation algorithm that has no noticeable pauses, eliminates page faults for transient objects, compacts objects without resorting to indirection, and reclaims circular structures, in one third the time of traditional approaches.

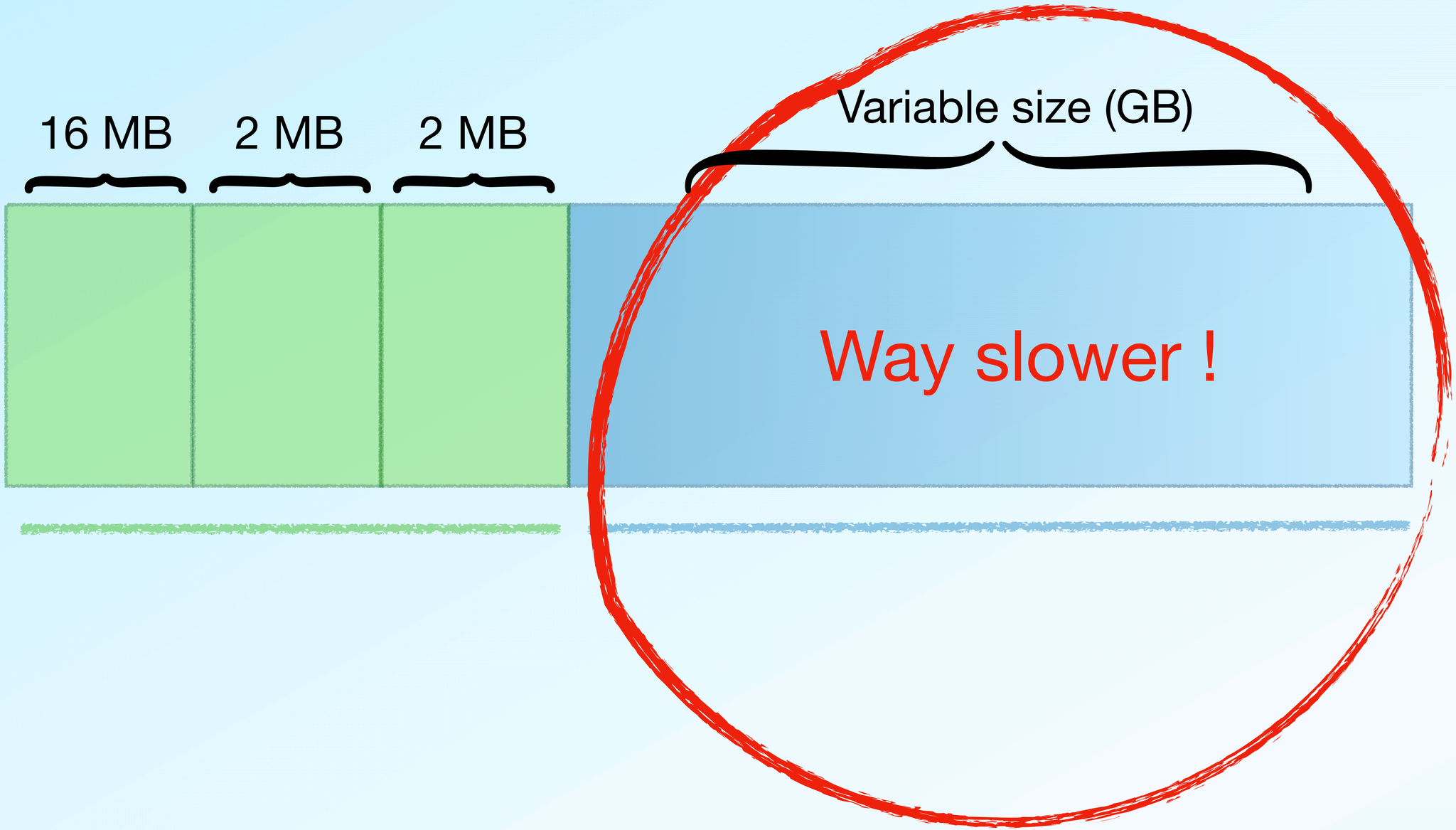
We have incorporated *Generation Scavenging* in Berkeley Smalltalk (BS), our Smalltalk-80* implementation, and instrumented it to obtain performance data. We are also designing a microprocessor with hardware support for

1. Introduction

Researchers have designed several interactive programming environments to expedite software construction [She83] Central to such environments are high level languages like Lisp, Cedar Mesa, and Smalltalk-80™ [GoR83], that provide virtual memory and automatic storage reclamation. Traditionally, the cost of a storage management strategy has been measured by its use of CPU time, primary memory, and backing store operations averaged over a session. Interactive systems demand good short-term performance as well. Large, unexpected pauses caused by thrashing or storage reclamation are distracting and reduce productivity. We have designed, implemented, and measured *Generation Scavenging*, a new garbage collector that

- limits pause times to a fraction of a second,
- requires no hardware support,
- meshes well with virtual memory,

Because



Some info

Open Pharo, open a Playground write some random code

- Hundreds++ of scavenges
- ZERO old generation traversals (normally)

Normally when we allocated objects

Young generation

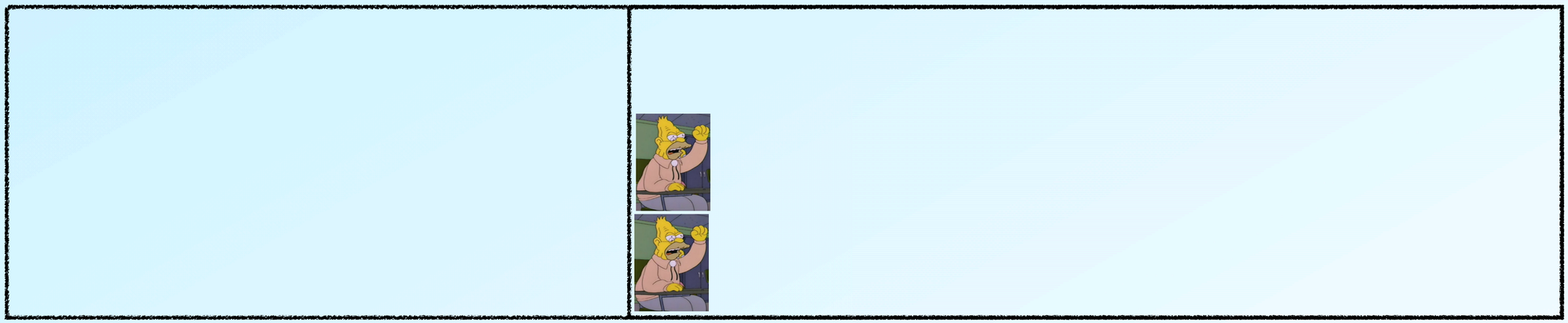
Old generation

Only few survive the scavenges

Young generation

Old generation

Few objects get tenured*



Young generation

Old generation

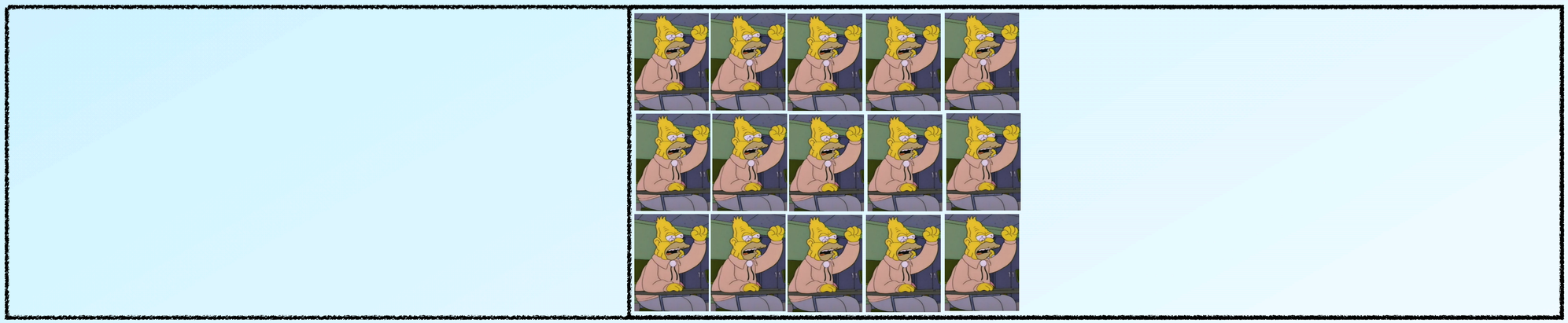
* Tenured = promoted to the old generation

But some applications have different memory behaviours

Young generation

Old generation

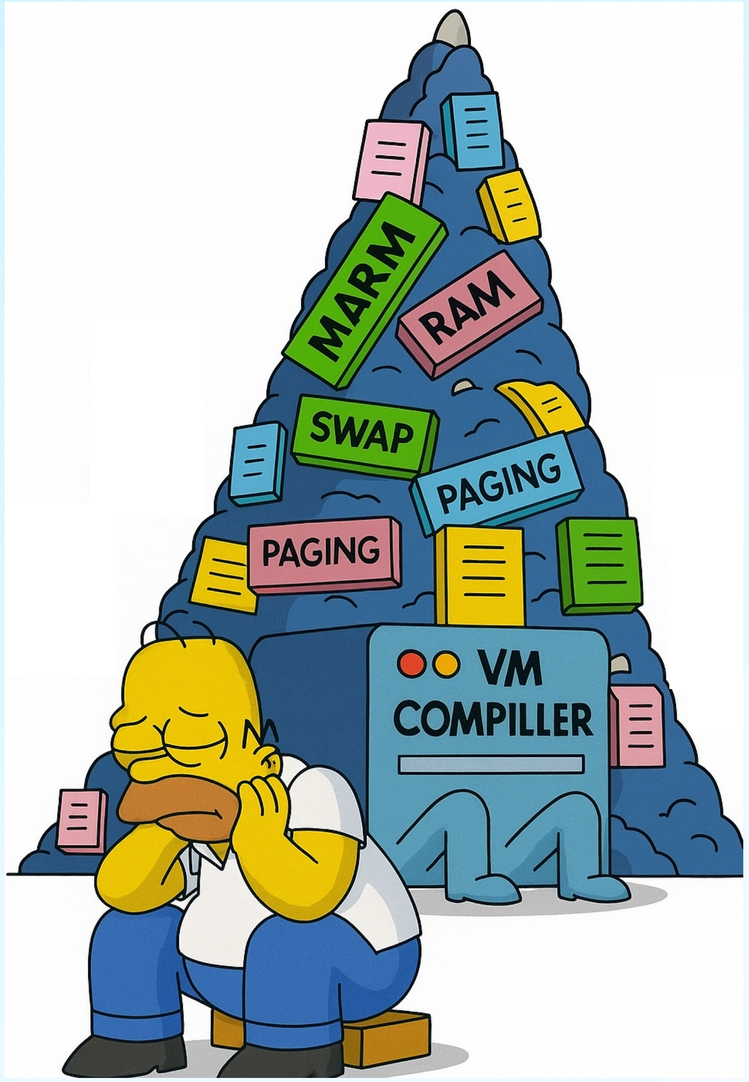
We start to put the old generation under heavy pressure



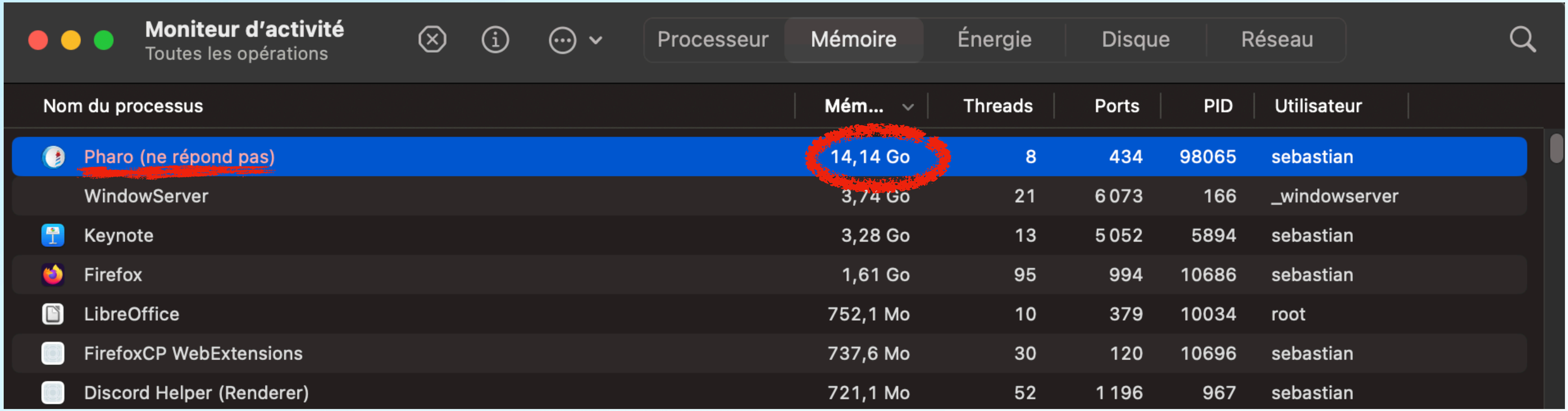
Young generation

Old generation

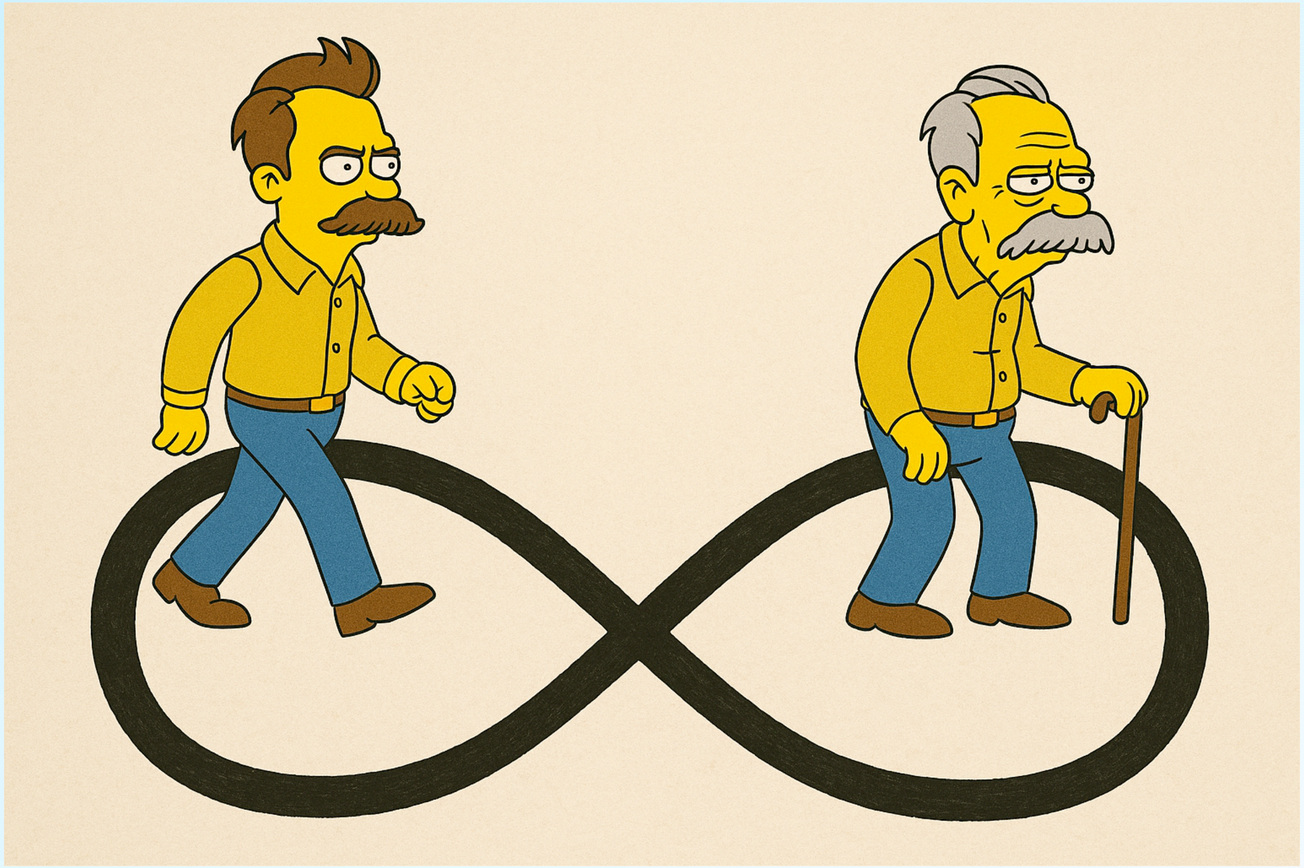
The VM spends more time managing memory than actually executing code



And we do not want that



We can use object lifetimes to improve this situation



The eternal lifetimes return

If we know the object will be long-lived

We can skip copying it from new to old space

Pretenuring

When we allocate an object directly in the old generation

Young generation

Old generation

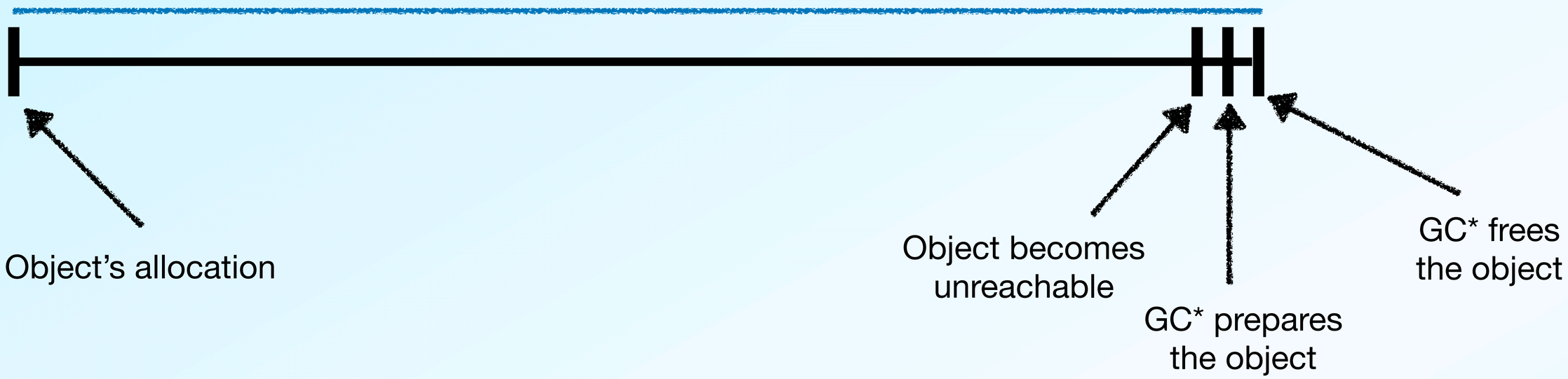
Profile-Based Pretenuring

STEPHEN M. BLACKBURN
Australian National University
MATTHEW HERTZ
Canisius College
KATHRYN S. McKINLEY
University of Texas at Austin
and
J. ELIOT B. MOSS and TING YANG

```
Behavior >> basicNewTenured:
    <p
```

```
Behavior >> basicNewTenured
    <primitive: 597>
```

Understanding object lifetimes



* GC = Garbage Collector

Estimating an object's lifetime

$$lifetime = t_{collection} - t_{allocation}$$

Motivating example

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionary
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self basicNew key: newKey value: newValue
```

```
Behavior >> basicNew
    <primitive: 70>
```

cormas

COmmon pool Resources and Multi-Agent Simulations

Where should we pretenure?

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionary
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self basicNew key: newKey value: newValue
```

```
Behavior >> basicNew

    <primitive: 70>
```

Maybe here?

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionary
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self basicNew key: newKey value: newValue
    self basicNewTenured
```

```
Behavior >> basicNew
    <primitive: 70>
```

cormas

COmmon pool Resources and Multi-Agent Simulations

We will pretenure all Associations ❌

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionary
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self ~~basicNew~~ key: newKey value: newValue
    self basicNewTenured
```

```
Behavior >> basicNew
    <primitive: 70>
```

cormas

COmmon pool Resources and Multi-Agent Simulations

Same, we will pretenure all Associations ❌

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionary
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self basicNew key: newKey value: newValue
```

```
Behavior >> basicNew
    <primitive: 70>
```

cormas

COmmon pool Resources and Multi-Agent Simulations

We want to pretenure *only* when it comes from Cormas ✅

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name -> probe value]) asDictionar
```

```
Object >> #->
  ^ Association key: self value: anObject
```

```
Association class >> #key:value:
  ^ self basicNew key: newKey value: newValue
```

```
Behavior >> basicNew

    <primitive: 70>
```

cormas

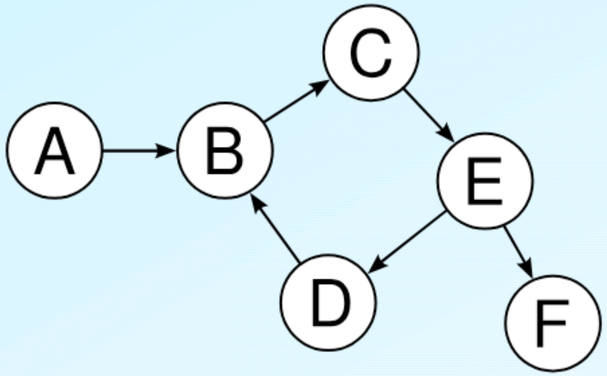
COmmon pool Resources
and Multi-Agent Simulations

Solution overview

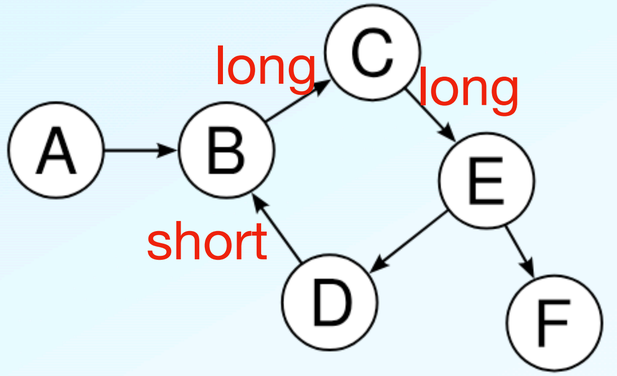
1

Object lifetime profiler

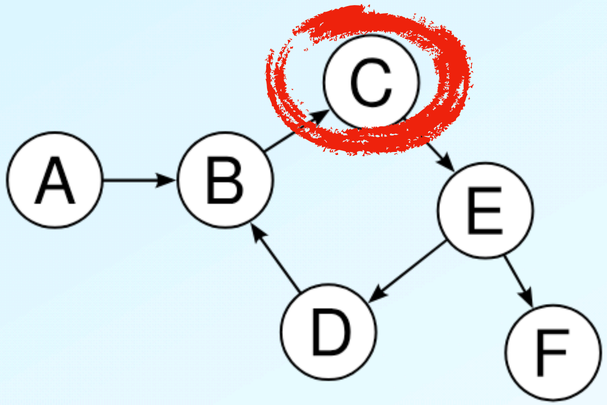
2



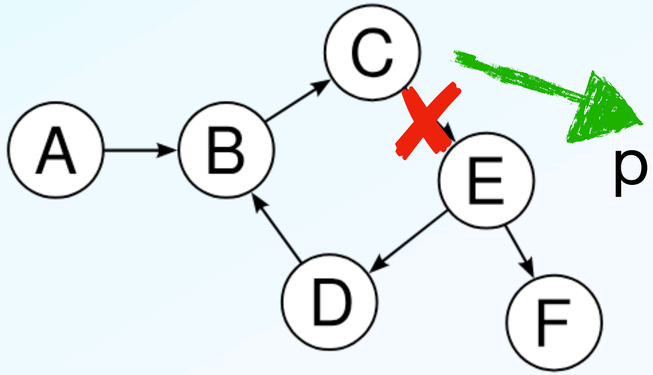
3



4



5



pretenuring

Step 1. Profiling

To estimate the object lifetimes

Object lifetime profiler

=

Array new: 7

Capture the allocation → MethodProxies [1]

\+

Register the finalization → Ephemerons [2]

jordanmontt/illimani-memory-profiler

[1] github.com/pharo-contributions/MethodProxies

[2] github.com/pharo-project/pheps/blob/main/phep-0003.md

Step 1. Profiling

The profiler give us

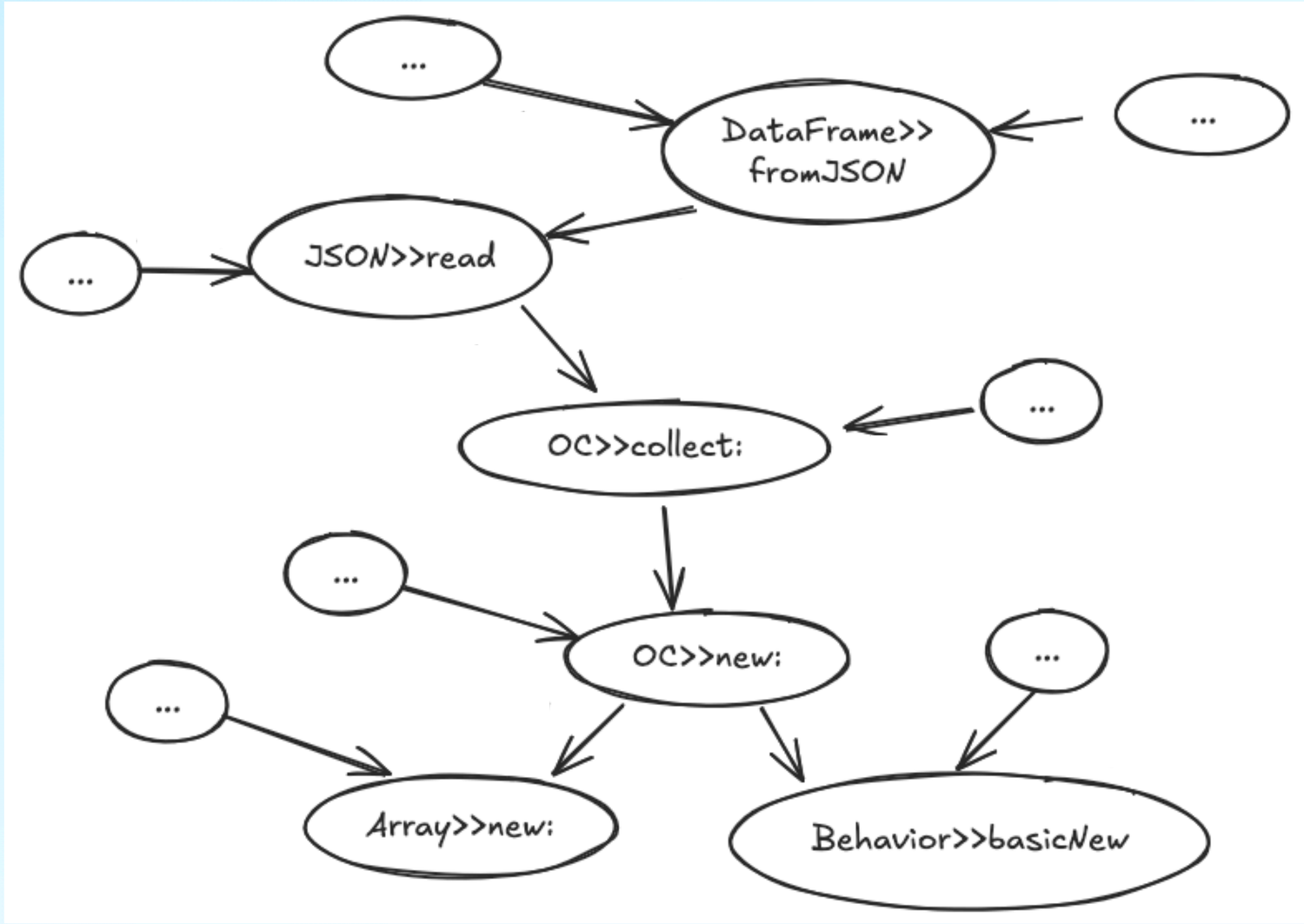
Object lifetime profiler

- Object lifetimes
- **Stack trace for each allocation**
- Size in memory
- Object types
- And other useful information

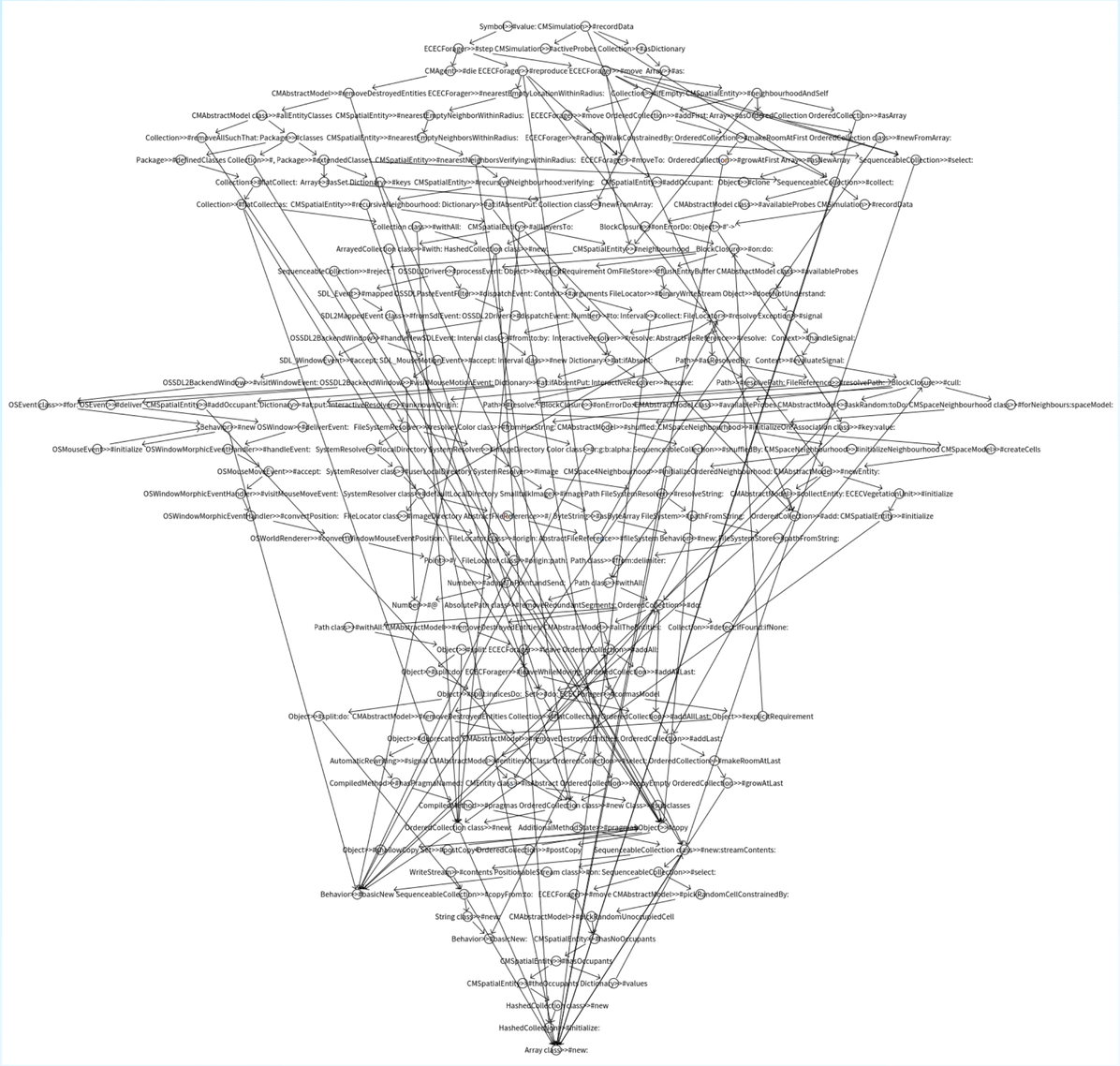
jordanmontt/illimani-memory-profiler

Step 2. Allocation call graph

We build the allocation call graph

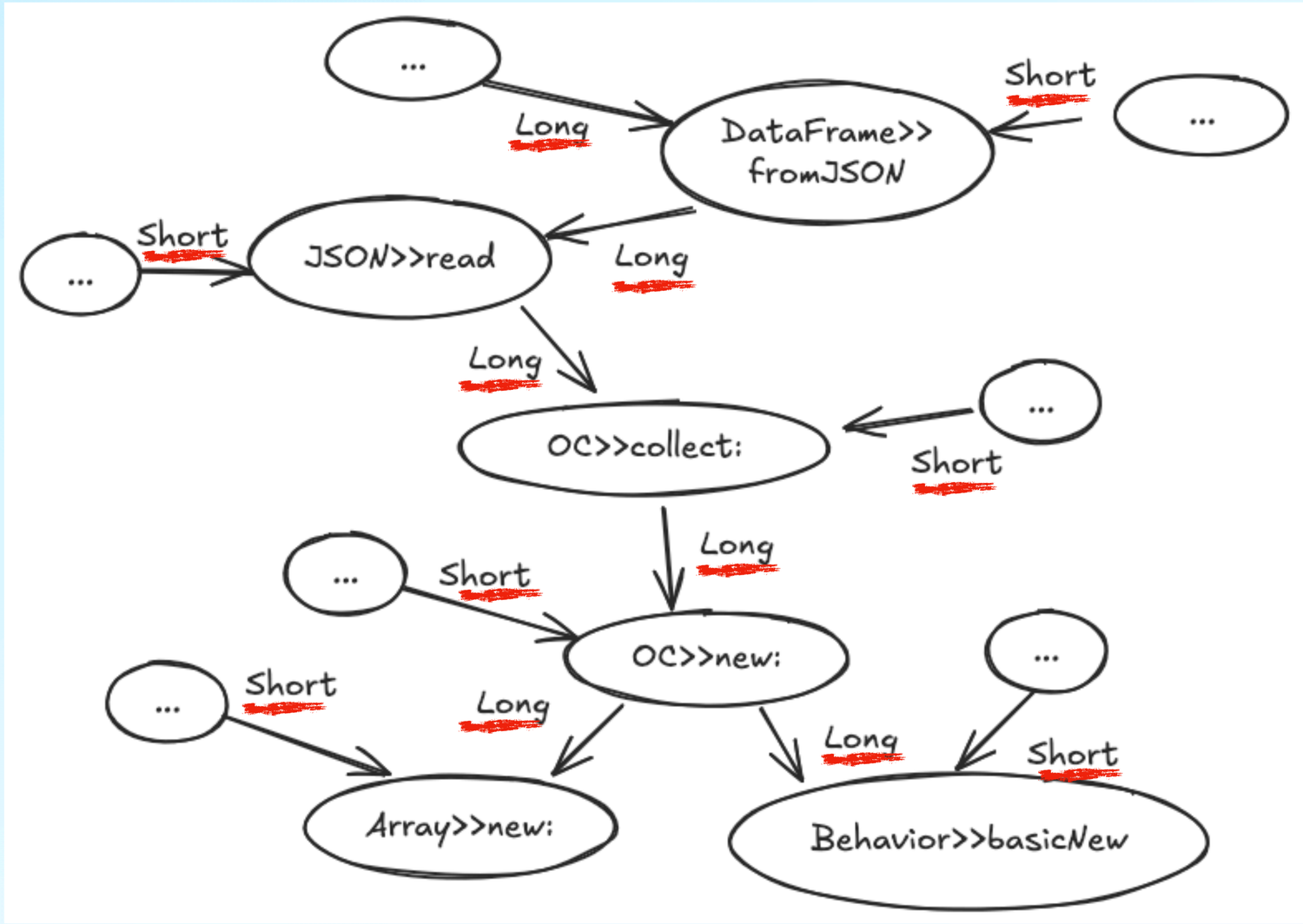


Cormas real pruned allocation call graph



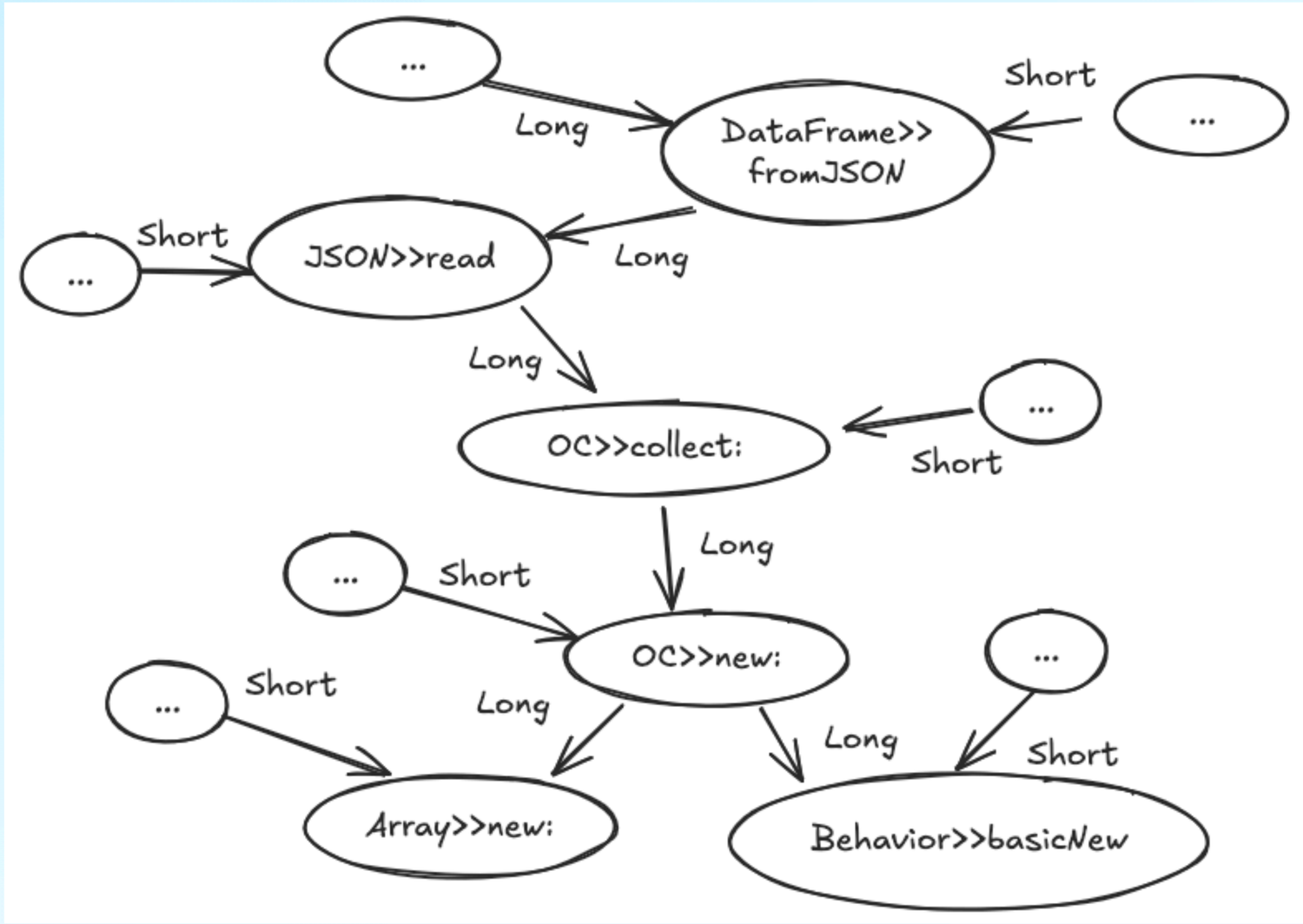
Step 3. Allocation sites classification

We classify each edge as long or short lived



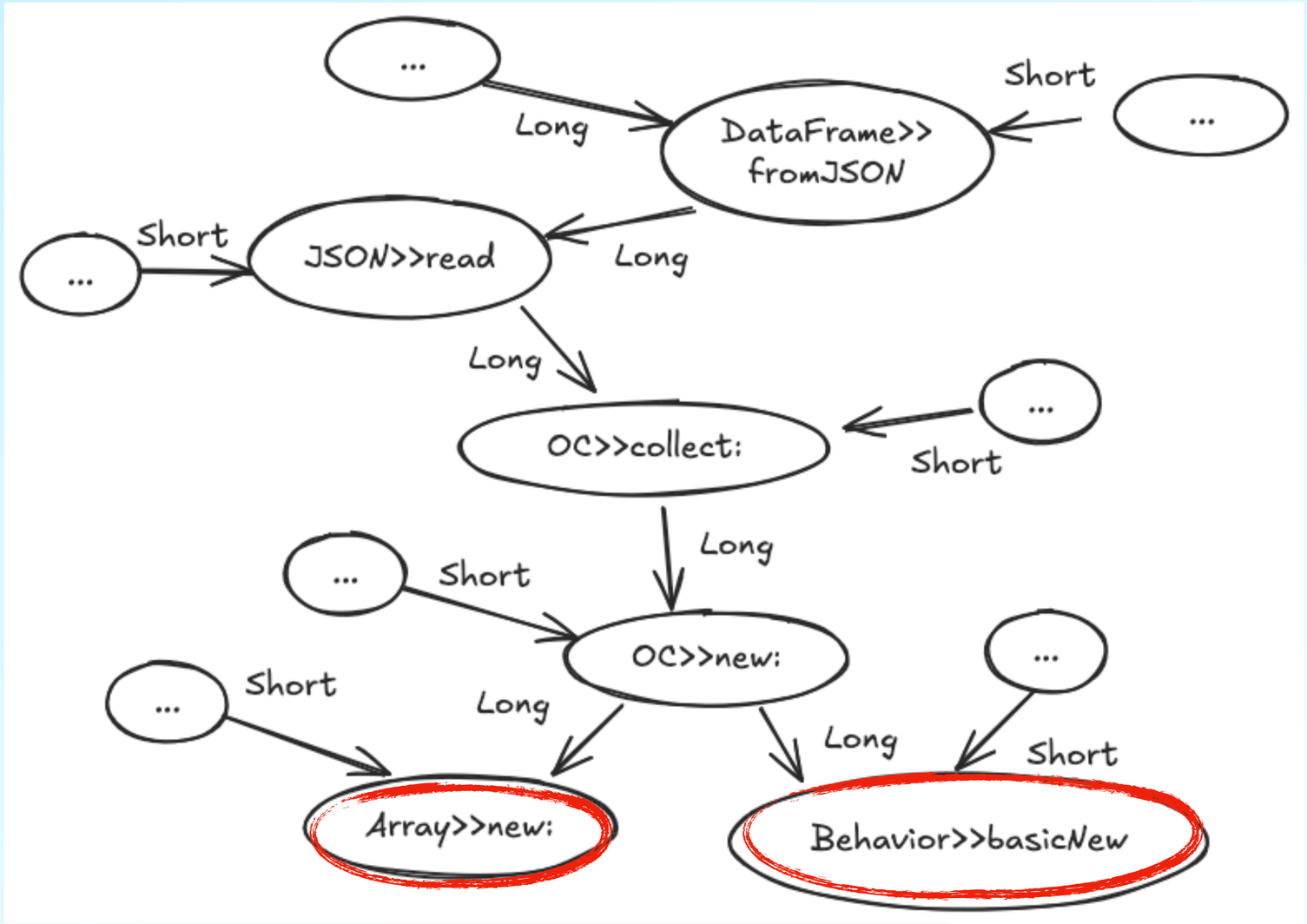
Step 4. Allocation sites identification

Where we should pretenure the allocations



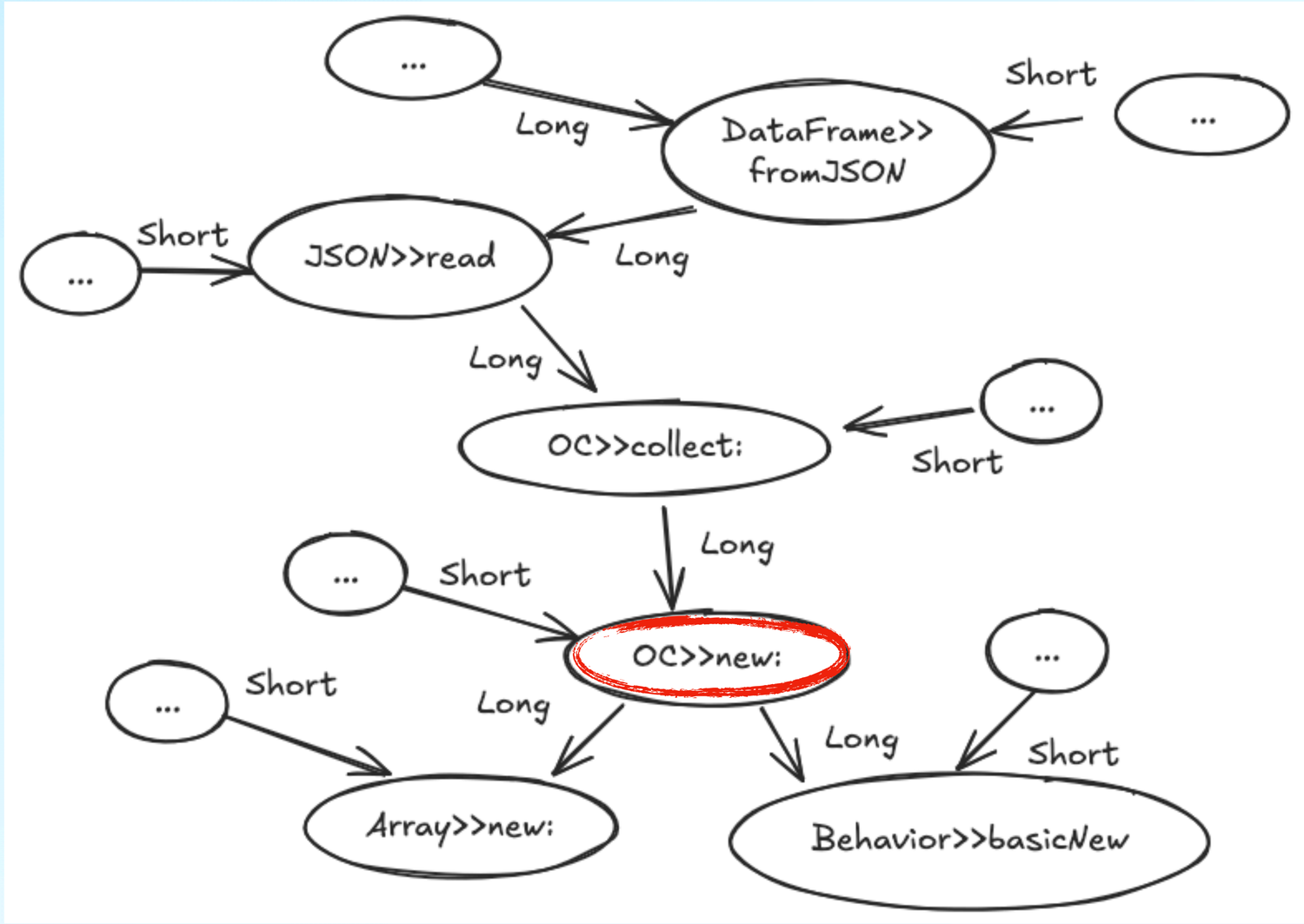
Step 4. Allocation sites identification

We start at the leaves



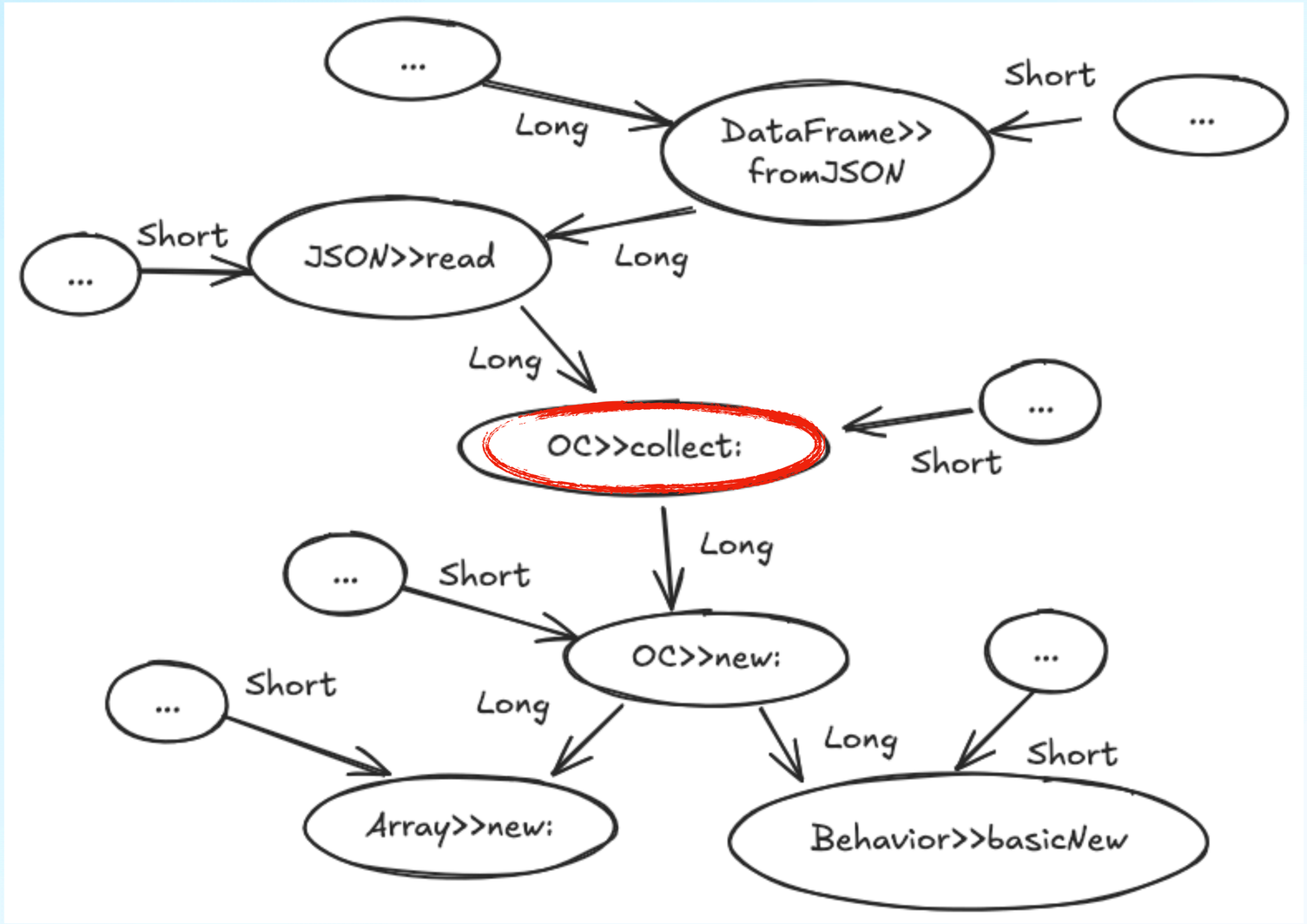
Step 4. Allocation sites identification

We go up in the hierarchy



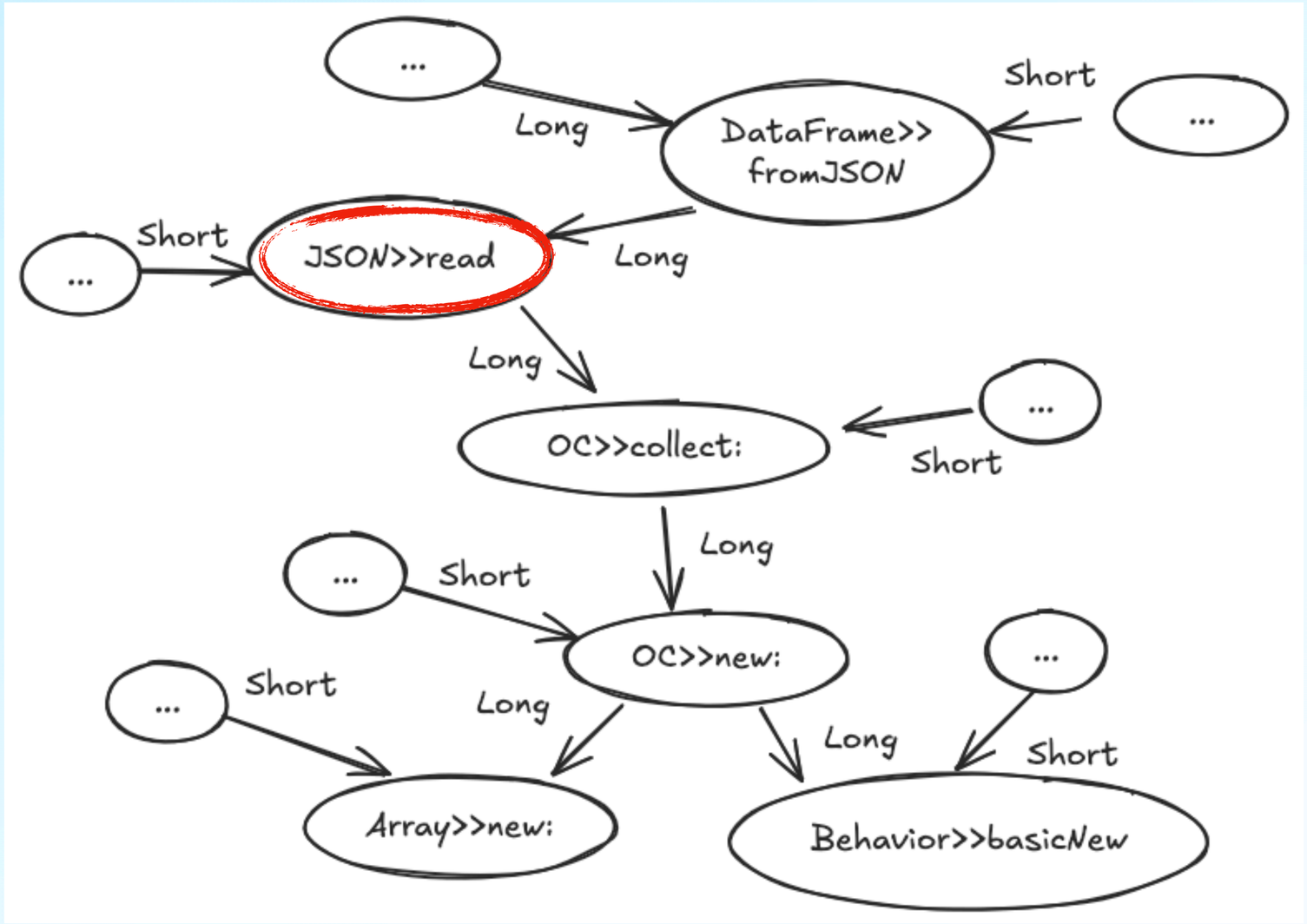
Step 4. Allocation sites identification

We go up



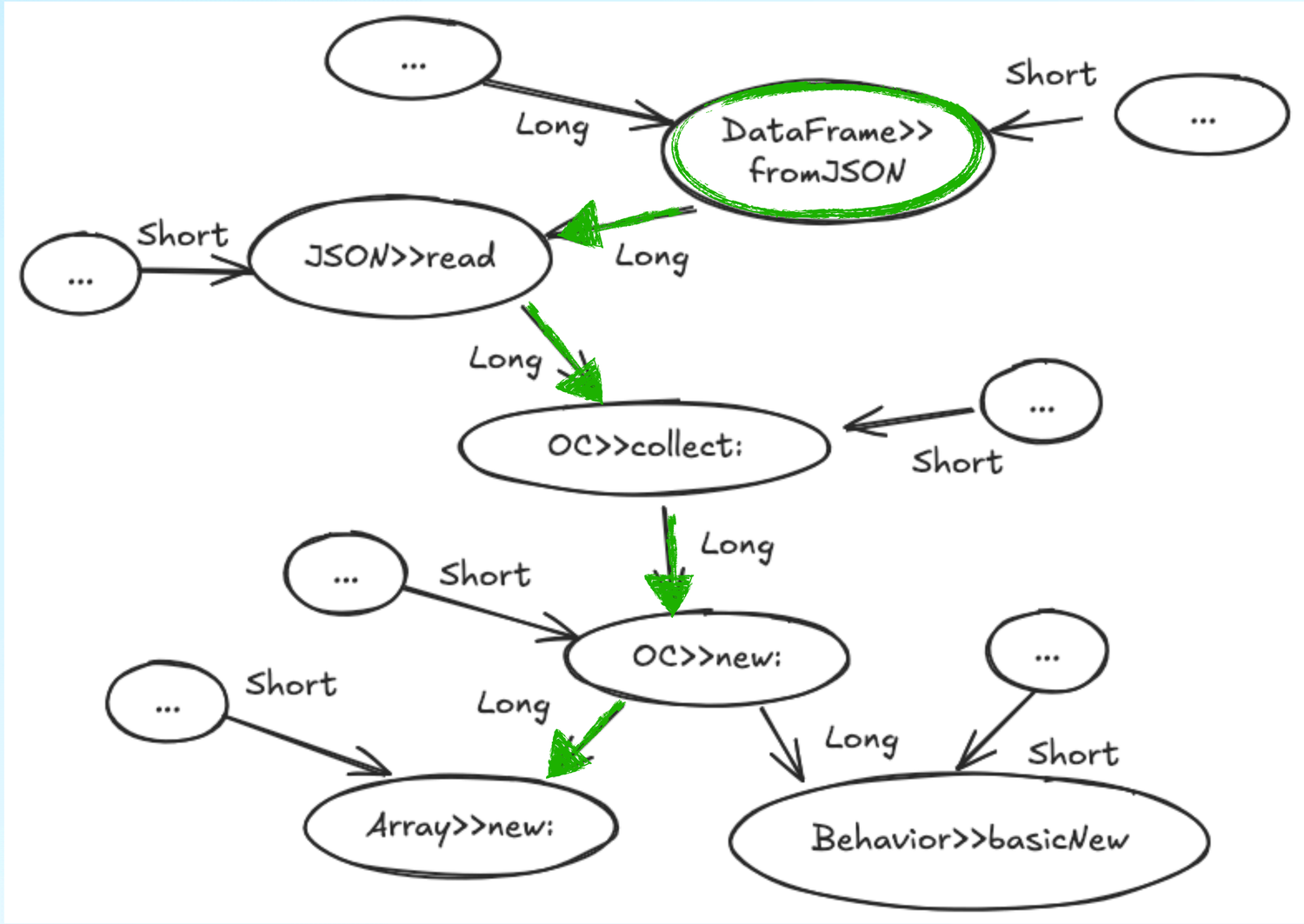
Step 4. Allocation sites identification

And up



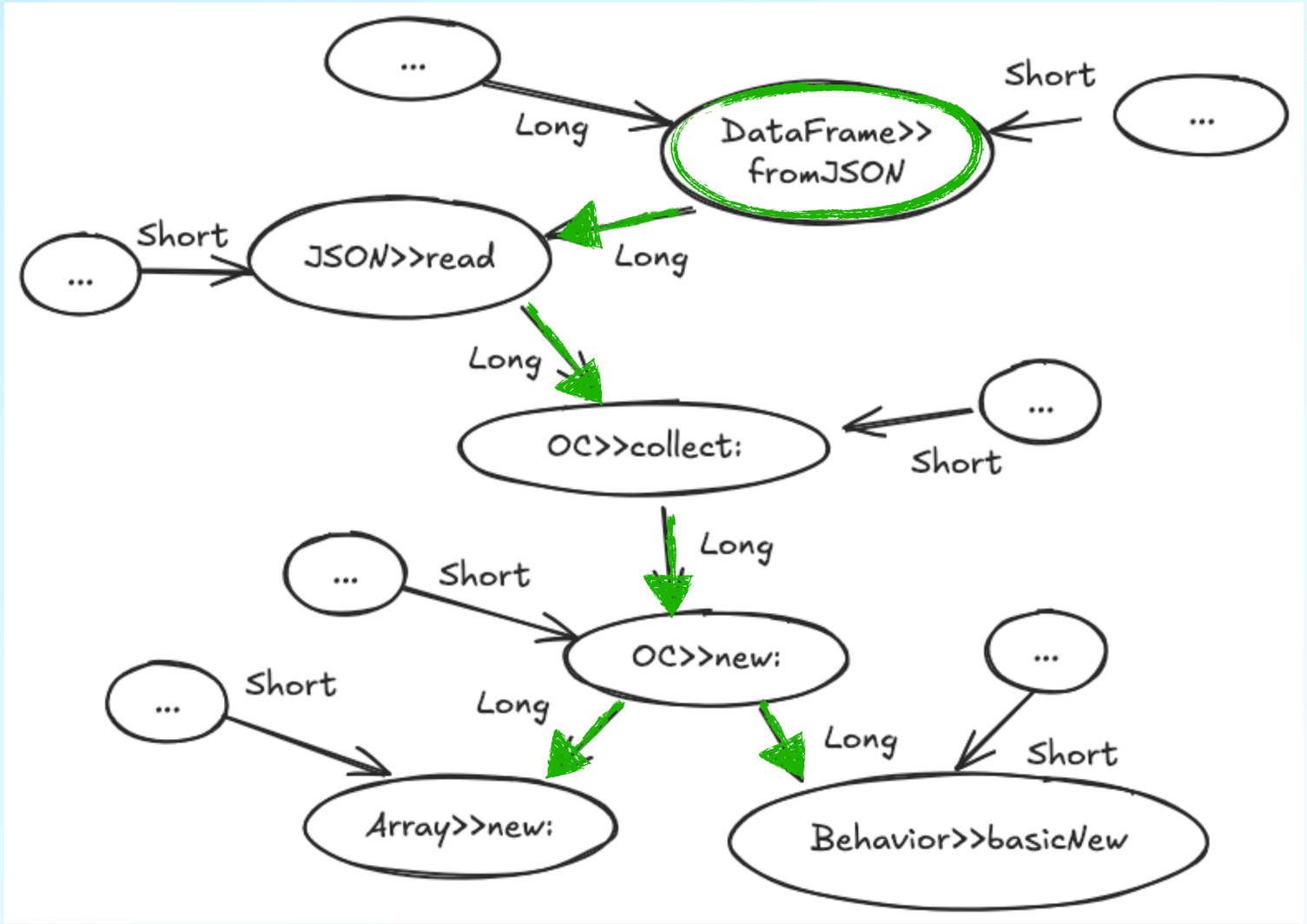
Step 4. Allocation sites identification

Until we found the desire allocation site



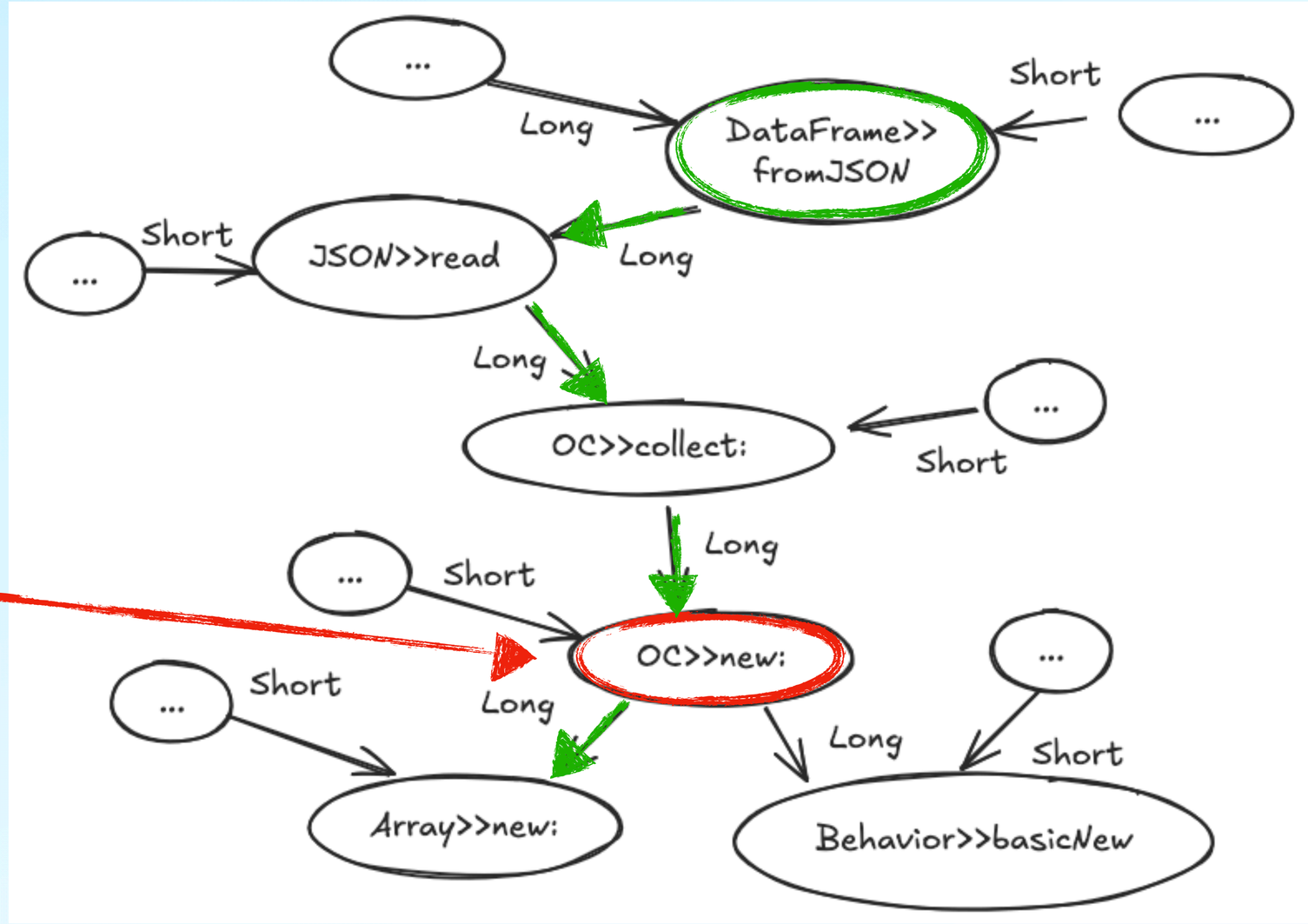
Step 4. Allocation sites identification

All of them



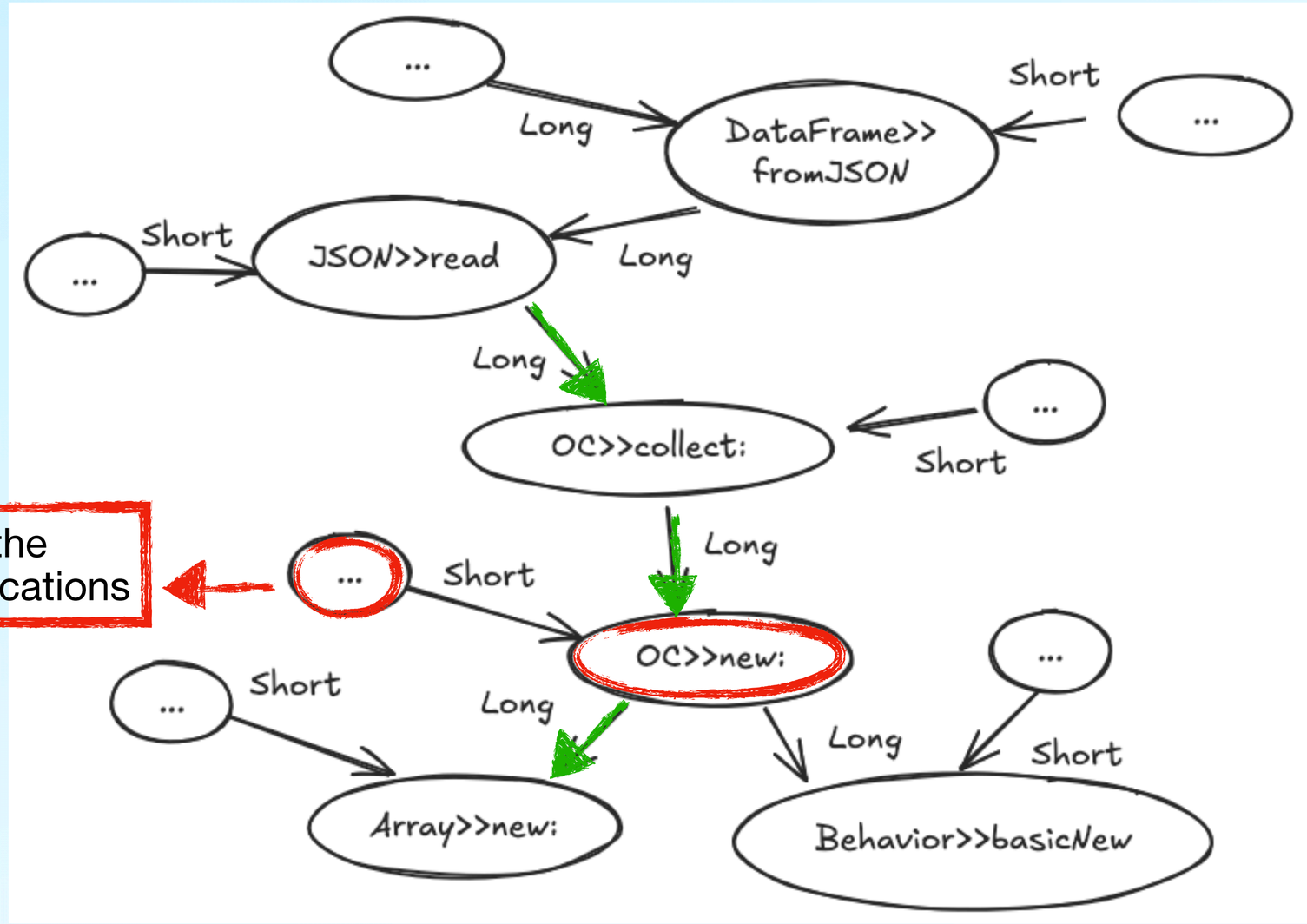
Step 4. Allocation sites identification

Why not taking the other nodes?



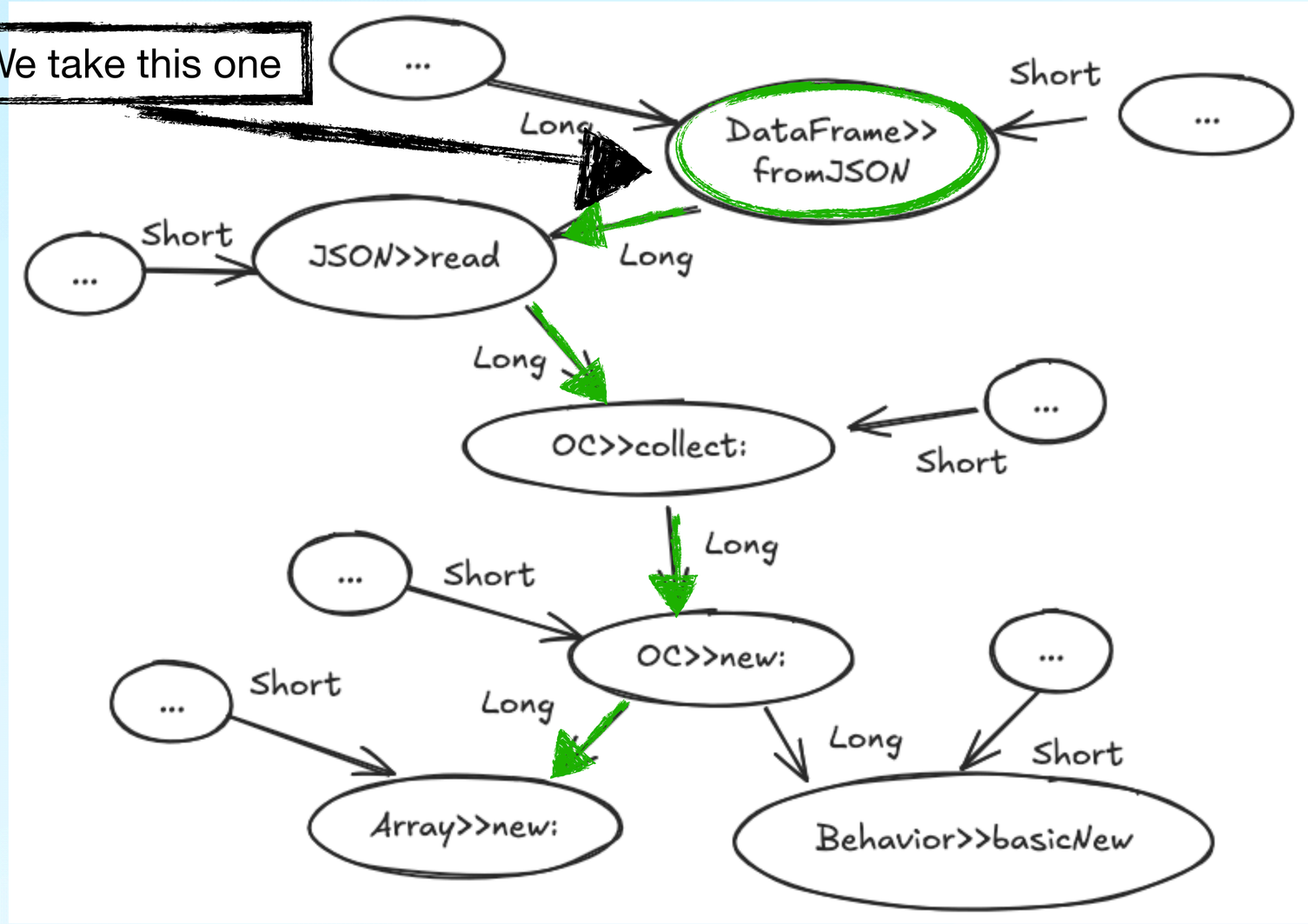
Step 4. Allocation sites identification

Because we'll also be tenuring the short-lived allocations



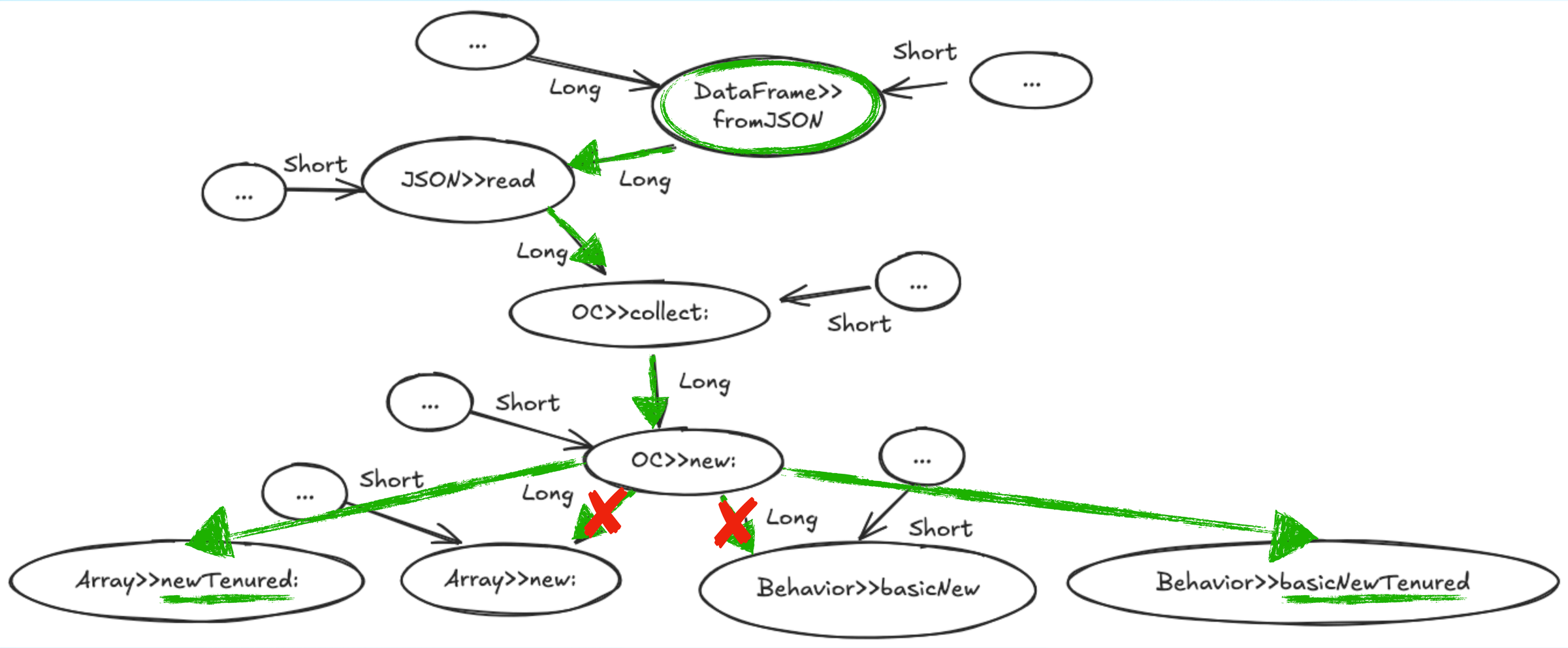
Step 4. Allocation sites identification

So we use different strategies to avoid the penalization



Step 5. Code rewriting

We replace the calls to the allocation primitives

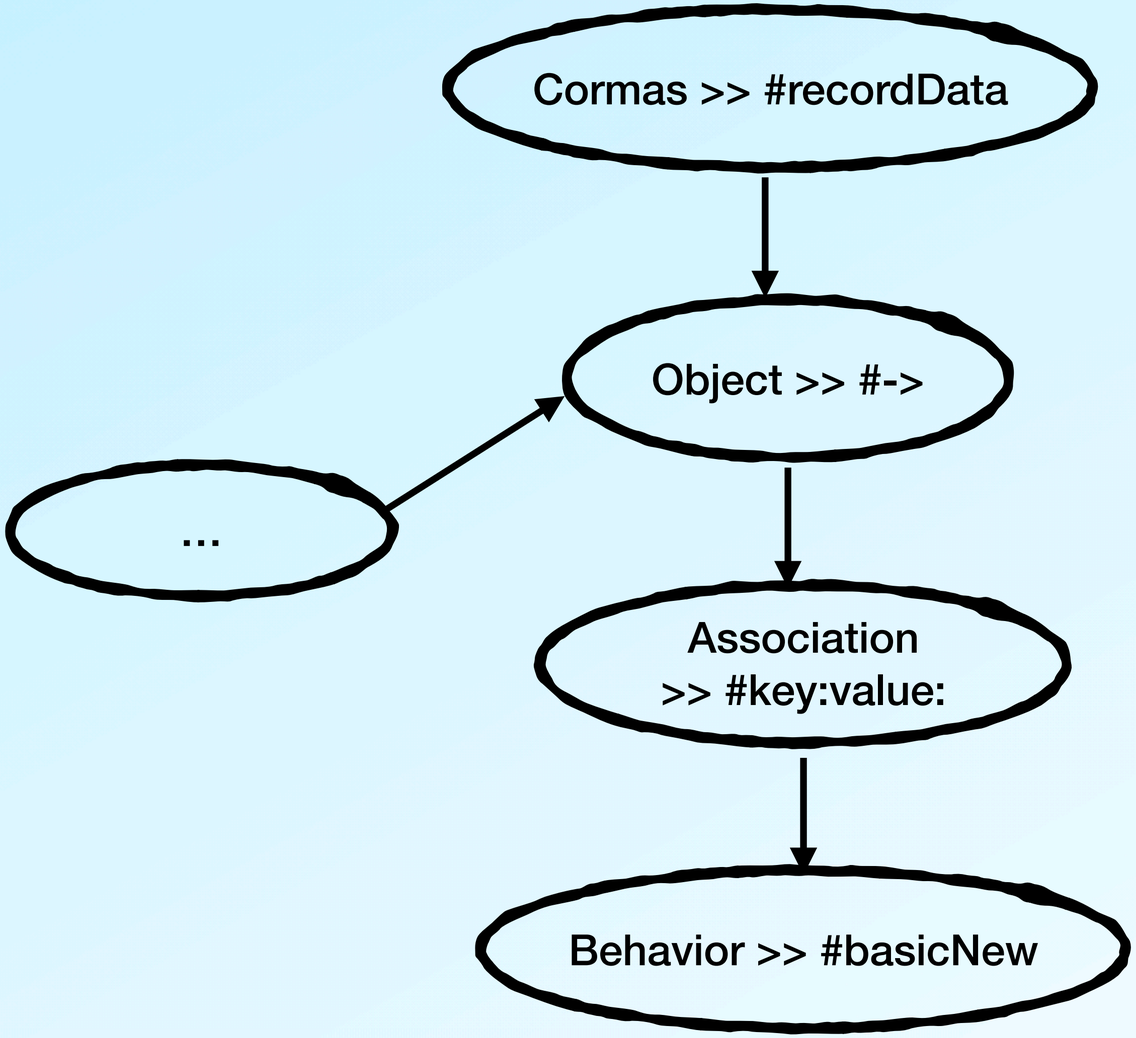


Coming back to a concrete example

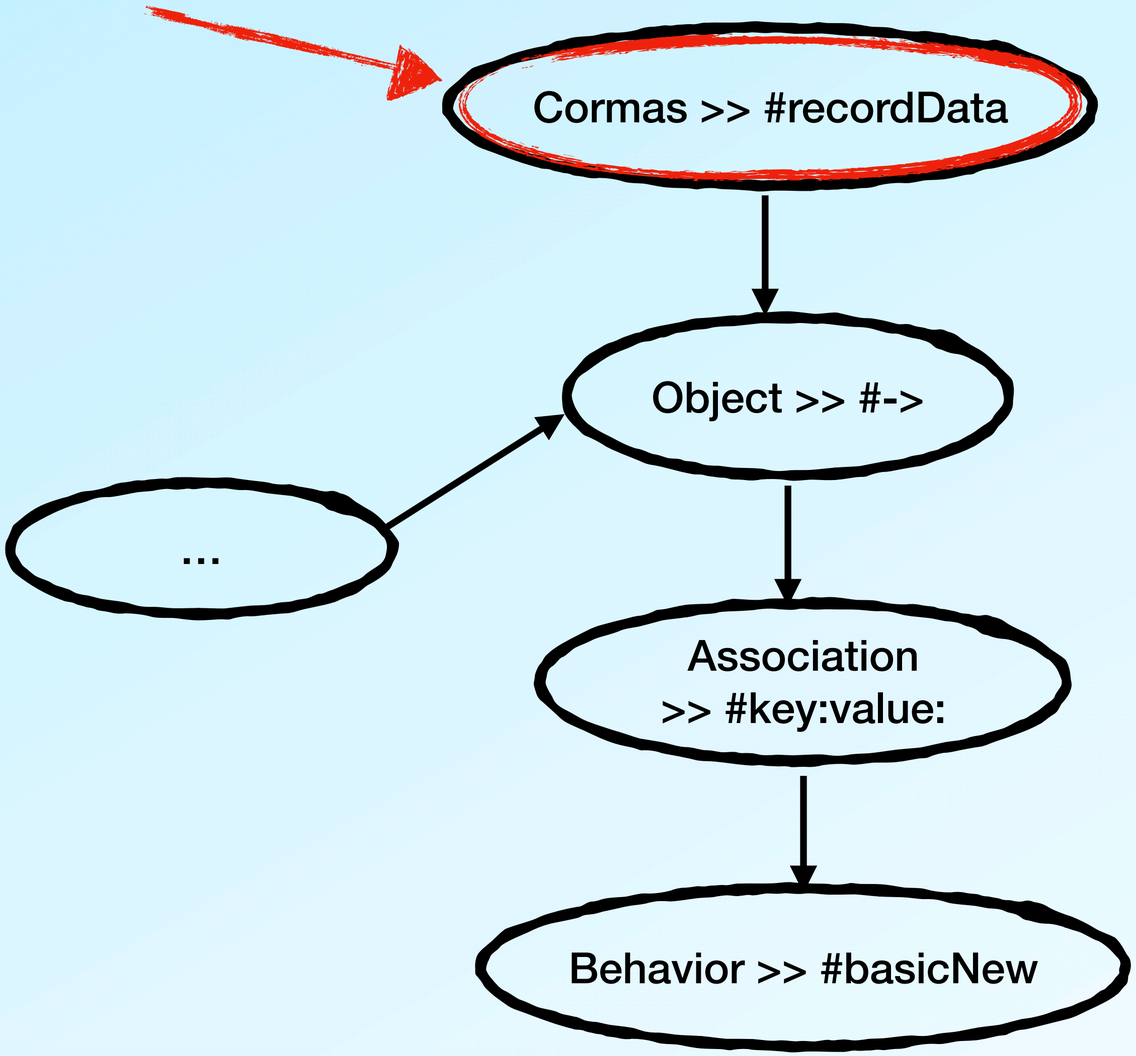
We profile Cormas

```
IllMemoryProfiler new
        profileOn: [ CormasExperiment new run ];
        yourself.
```

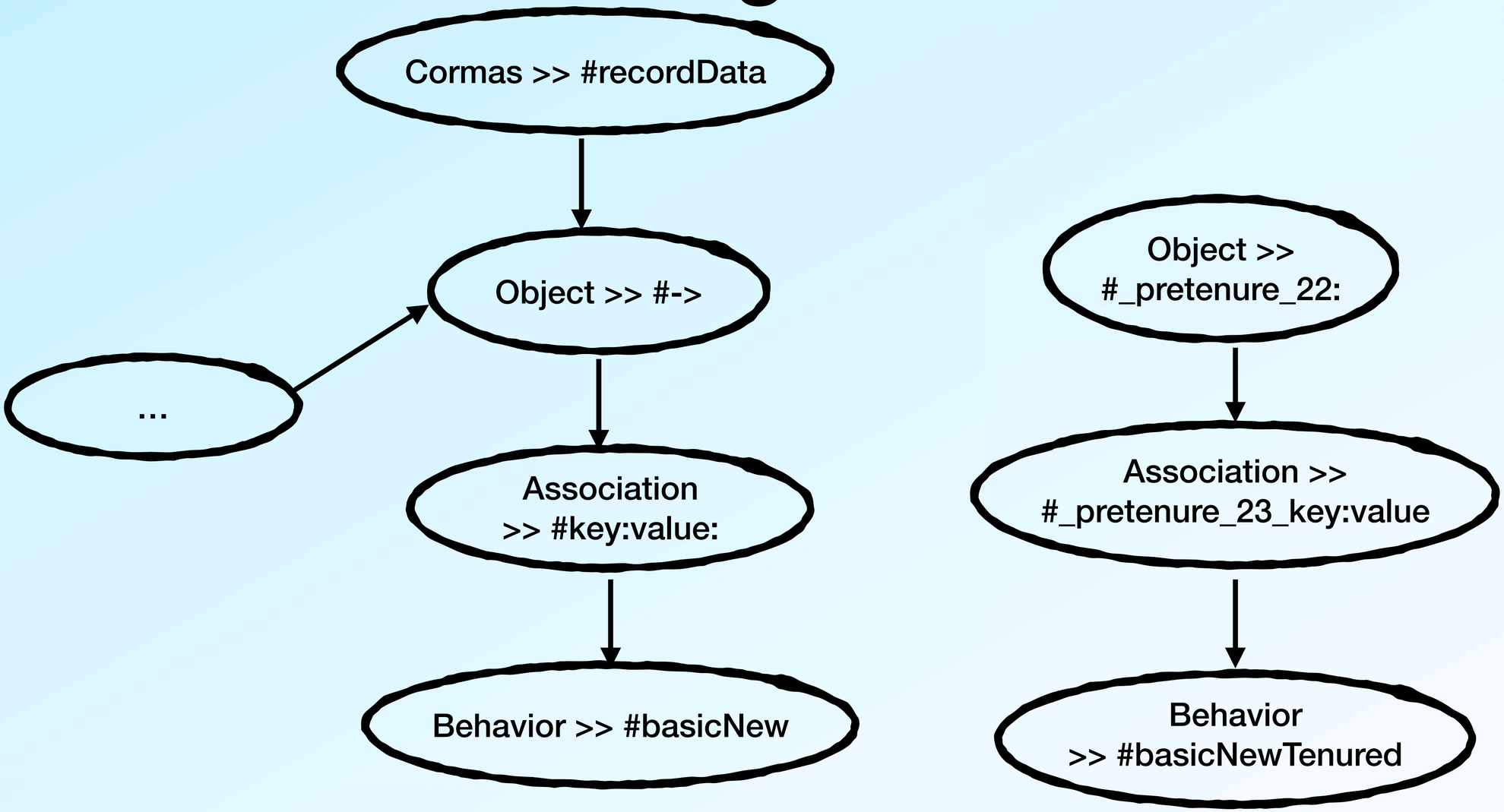
We build the allocation call graph



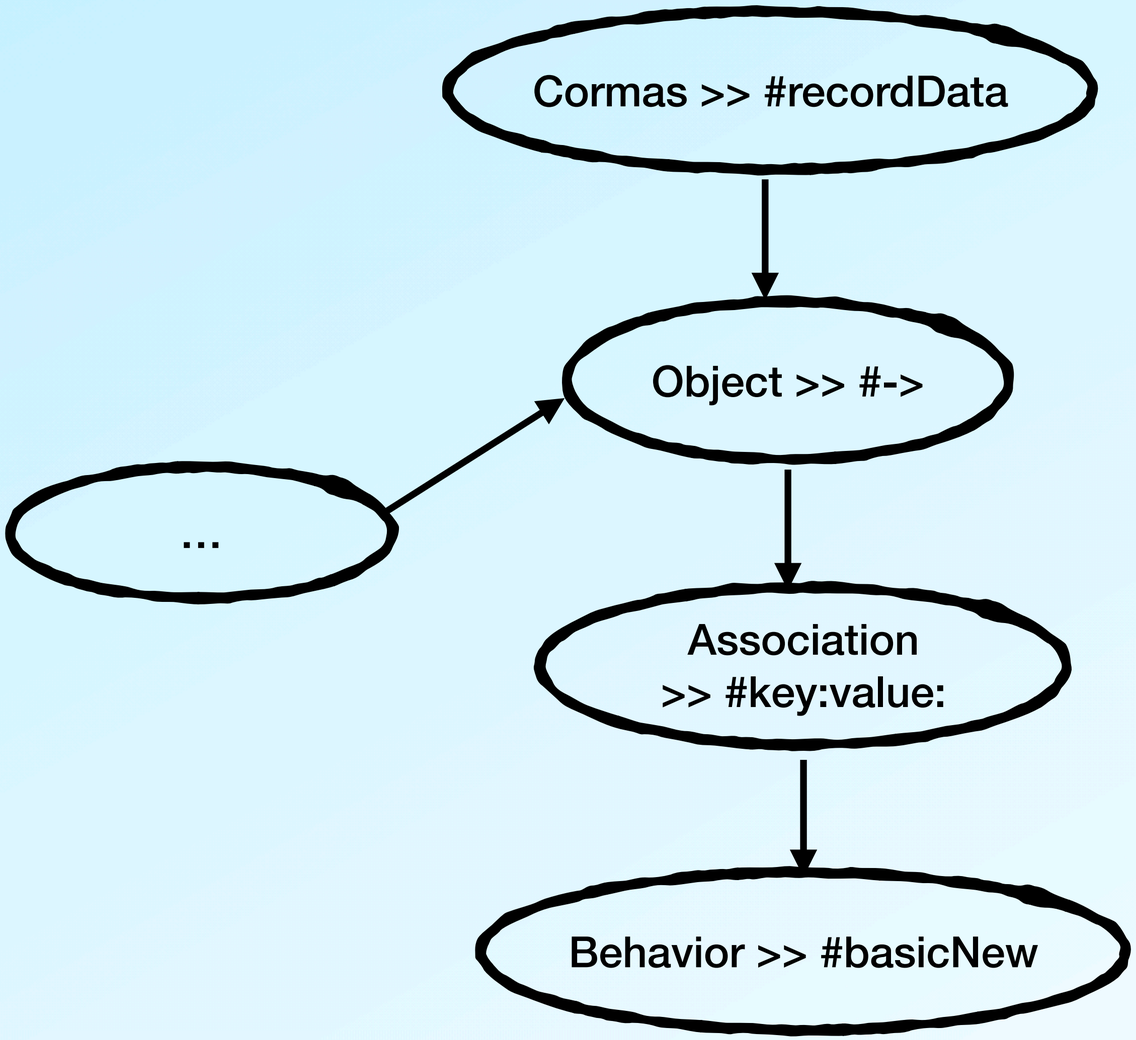
We identify from where we want to pretenure

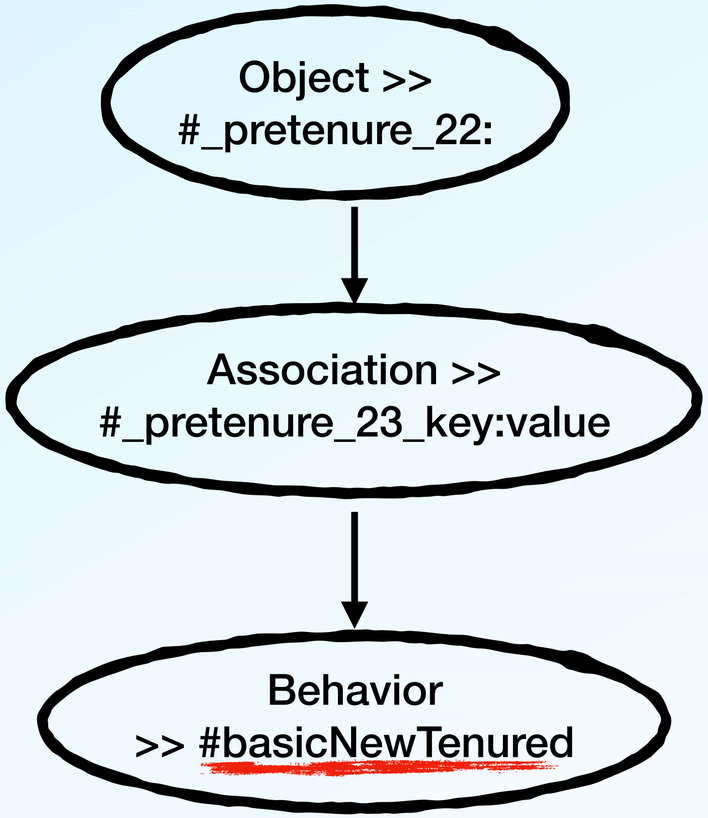


We create copies with unique selectors of the methods being called

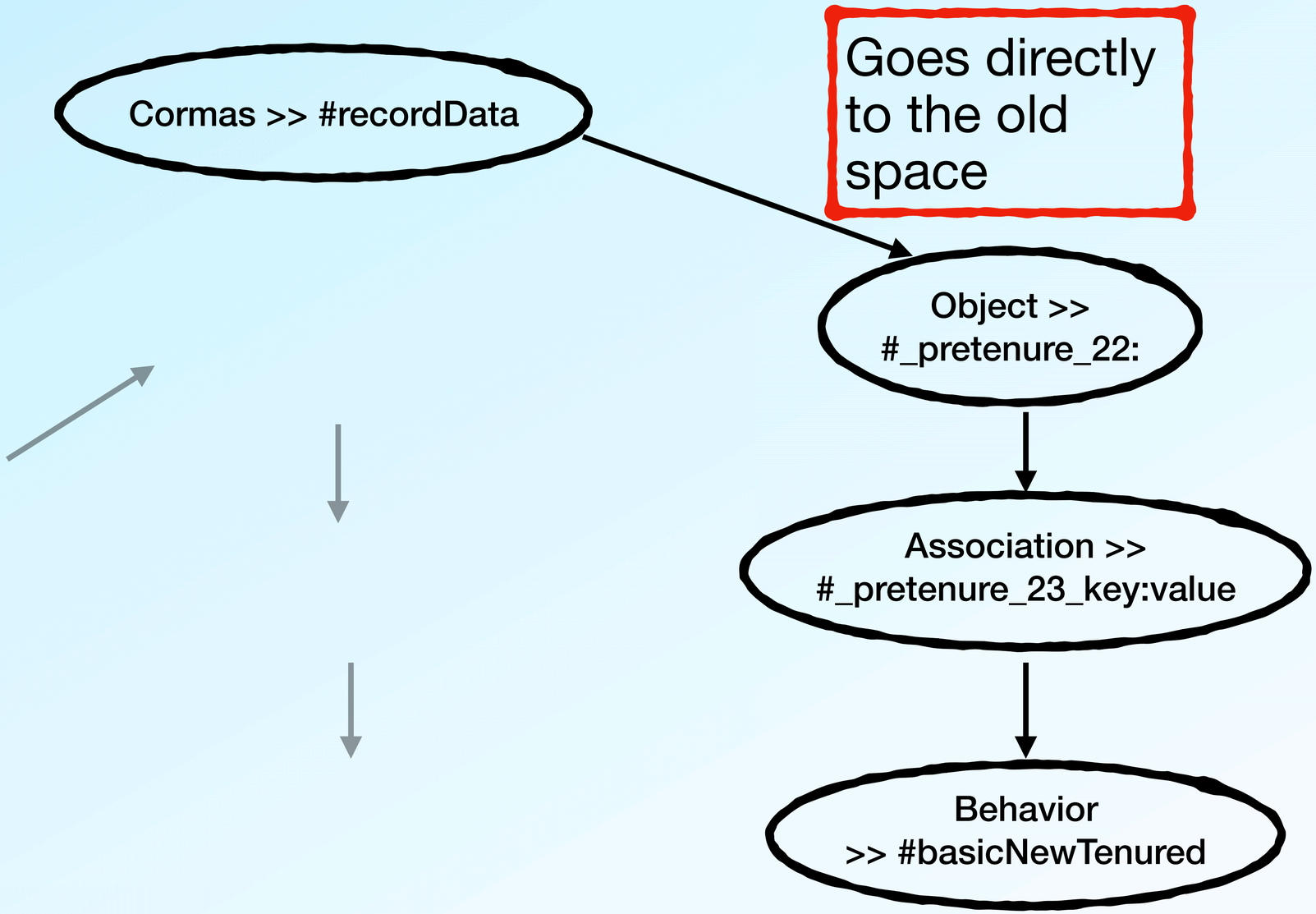


That end up calling the pretenuring primitive

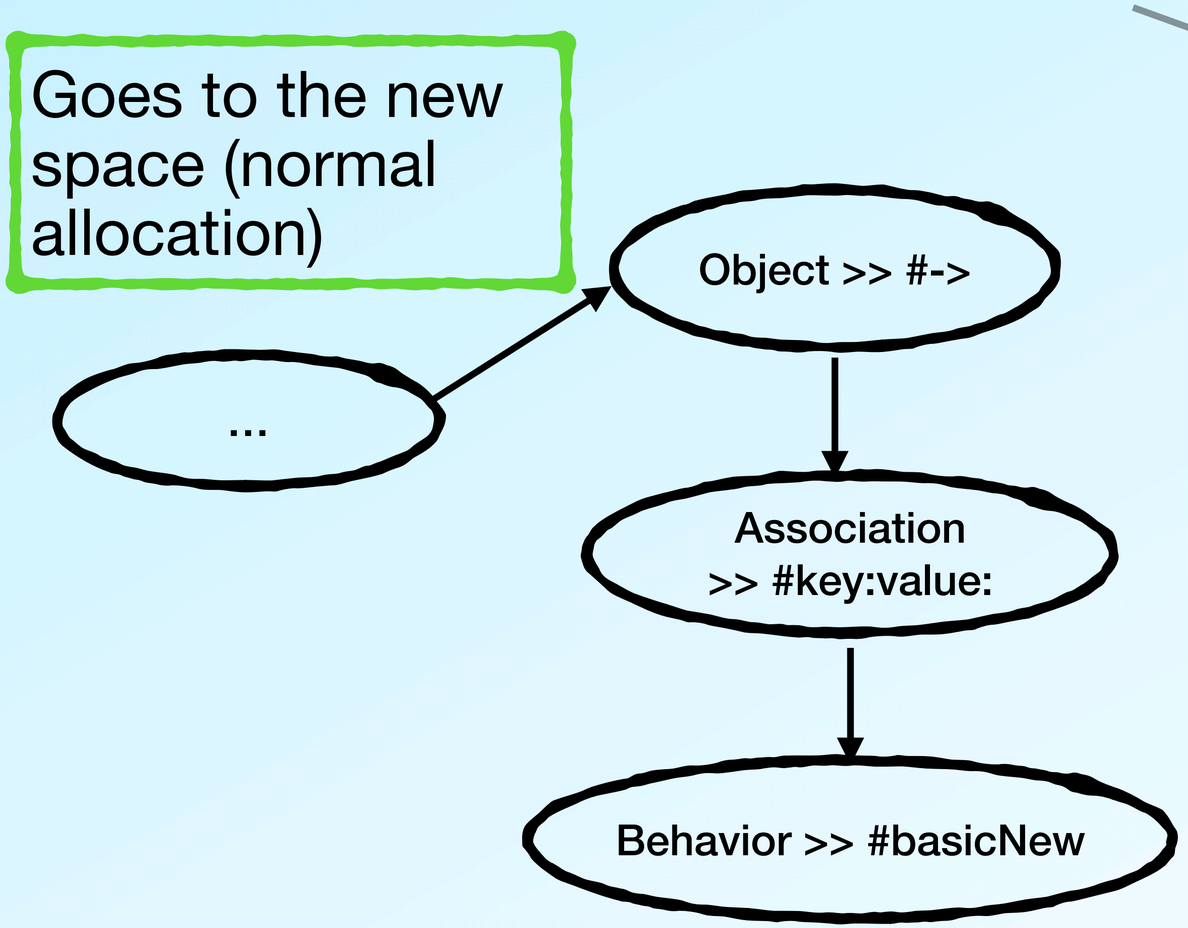




We replace the call



We *do not affect* the other users of #->



How the code rewriting looks

```
CMSimulation >> #recordData
  data add:
    (self activeProbs
        collect: [ :probe | probe name _pretenure_23412: probe value]) asDictionary
```

```
Object >> #_pretenure_23412: anObject
  ^ Association _pretenure_23412_key: self value: anObject
```

```
Association class >> #_pretenure_23412_key:value:
  ^ self basicNewTenured key: newKey value: newValue
```

```
Behavior >> basicNewTenured

    <primitive: 597>
```

Some results

- Around 15% faster 🔥✅

- More are about to come 😎

Selective pretenuring

- We can use an object lifetimes profiler to identify long-lived allocation sites
- Pretenuring can reduce the GC overhead
- We only pretenure the application-specific allocations

Sebastian JORDAN MONTAÑO
sebastian.jordan@inria.fr

jordanmontt/illimani-memory-profiler

jordanmontt/senders-chain-transformer