Memory management Part II

Michel Schinz (based on Erik Stenman's slides)

Advanced Compiler Construction / 2006-04-07

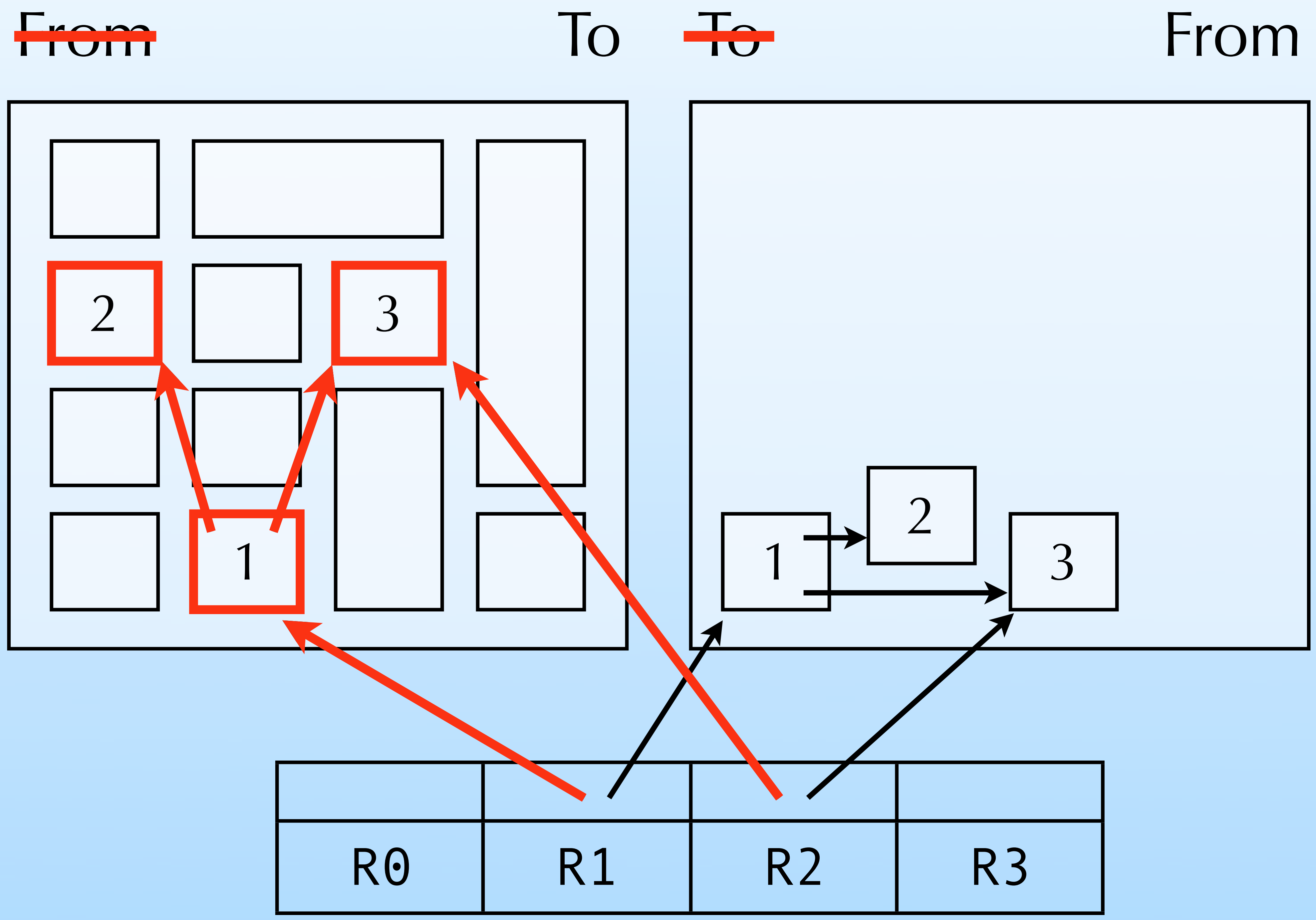
Copying garbage collection

Copying GC

The idea of **copying garbage collection** is to split the heap in two **semi-spaces** of equal size: the **from-space** and the **to-space**.

Memory is allocated in from-space, while to-space is left empty. When from-space is full, all reachable objects in from-space are copied to to-space, and pointers to them are updated accordingly. Finally, the role of the two spaces is exchanged, and the program resumed.

Copying GC



Allocation in a copying GC

In a copying GC, memory is allocated linearly in from-space.

There is no free list to maintain, and no search to perform in order to find a free block. All that is required is a pointer to the border between the allocated and free area of from-space.

Allocation in a copying GC is therefore as fast as stack allocation.

Forwarding pointers

Before copying an object, a check must be made to see whether it has already been copied. If this is the case, it must not be copied again. Rather, the already-copied version must be used.

How can this check be performed? By storing a **forwarding pointer** in the object in from-space, after it has been copied.

Cheney’s copying GC

The copying GC algorithm presented before does a depth-first traversal of the reachable graph. When it is implemented using recursion, it can lead to stack overflow.

Cheney’s copying GC is an elegant GC technique which does a breadth-first traversal of the reachable graph, requiring only one pointer as additional state.

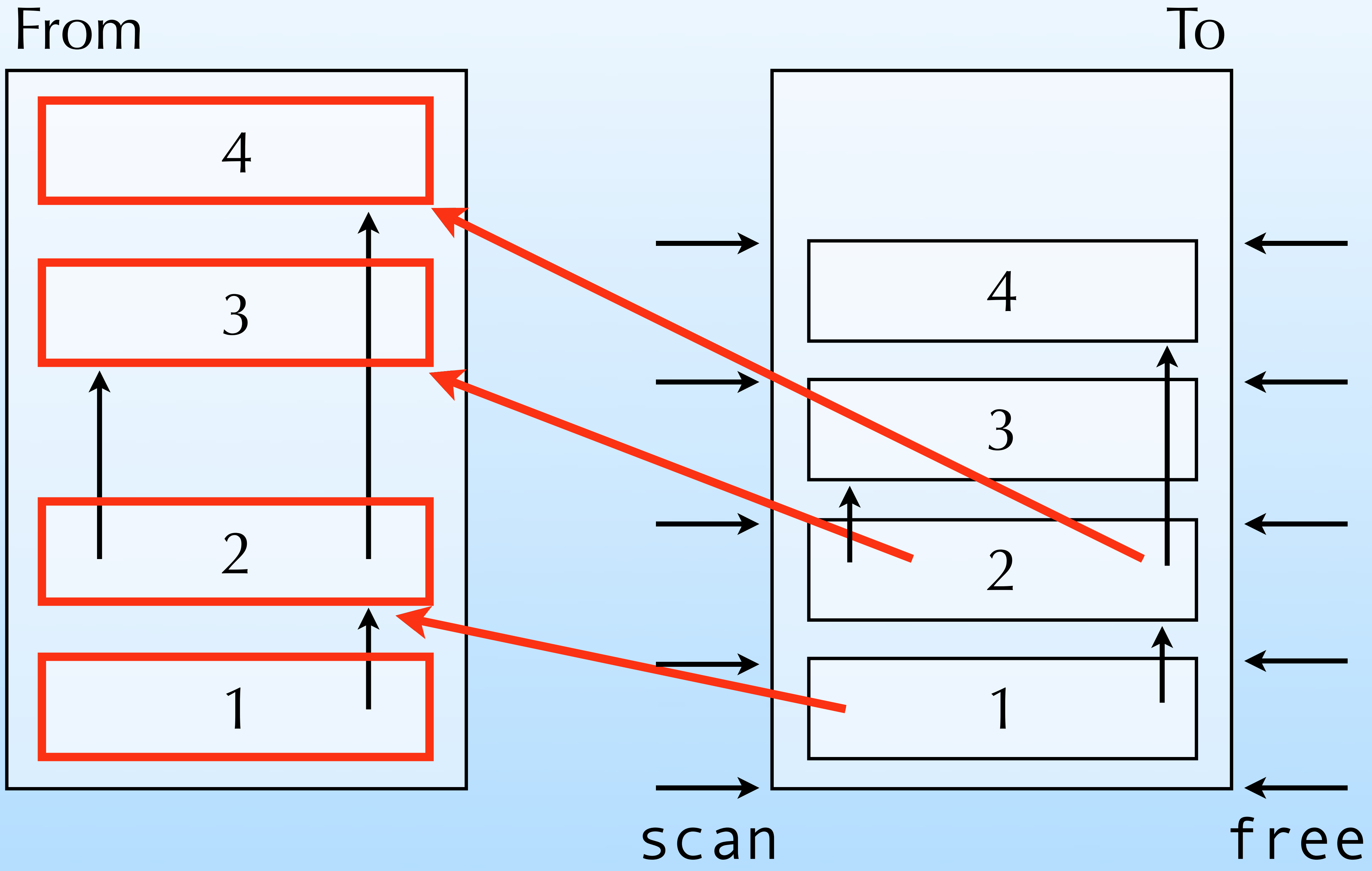
Cheney's copying GC

In any breadth-first traversal, one has to remember the set of nodes which have been visited, but whose children have not been.

The basic idea of Cheney's algorithm is to use to-space to store this set of nodes, which can be represented using a single pointer, called `scan`.

This pointer partitions to-space in two parts: the nodes whose children have been visited, and those whose children have not been visited.

Cheney's copying GC



Cost of copying GC

The collection takes time proportional to the amount of reachable data R. This is done to recover $H/2 - R$ words of memory.

Therefore, the amortised cost of copying GC is: $c_1 R / (H/2 - R)$.

That cost is high if $R \approx H/2$, that is if few objects are unreachable. But it can be very low if most objects are collected, which is often the case with some kinds of languages (*e.g.* functional).

Copying GC pros and cons

Copying GC completely avoids fragmentation by compacting memory at each collection. It also provides very fast allocation. Finally, its does not visit dead objects, unlike mark & sweep.

Its main disadvantages is that it needs twice the amount of memory compared to a marking GC, and that copying can become expensive with large objects. Since it moves objects around, it requires *precise* knowledge of the object graph.

Generational garbage collection

Generational GC

Empirical observation suggests that most objects die young.

The idea of **generational garbage collection** is to partition objects in generations – based on their age – and to collect the young generation more often than the old one(s).

This should improve the amount of memory collected per objects visited, and avoid repeated copying of long-lived objects.

Generational GC

In a generational GC, the heap is separated in at least two **generations**.

All objects are initially allocated in the youngest – and smallest – generation. When this generation is full, it is collected, and some surviving objects are promoted to the next generation based on a **promotion policy**.

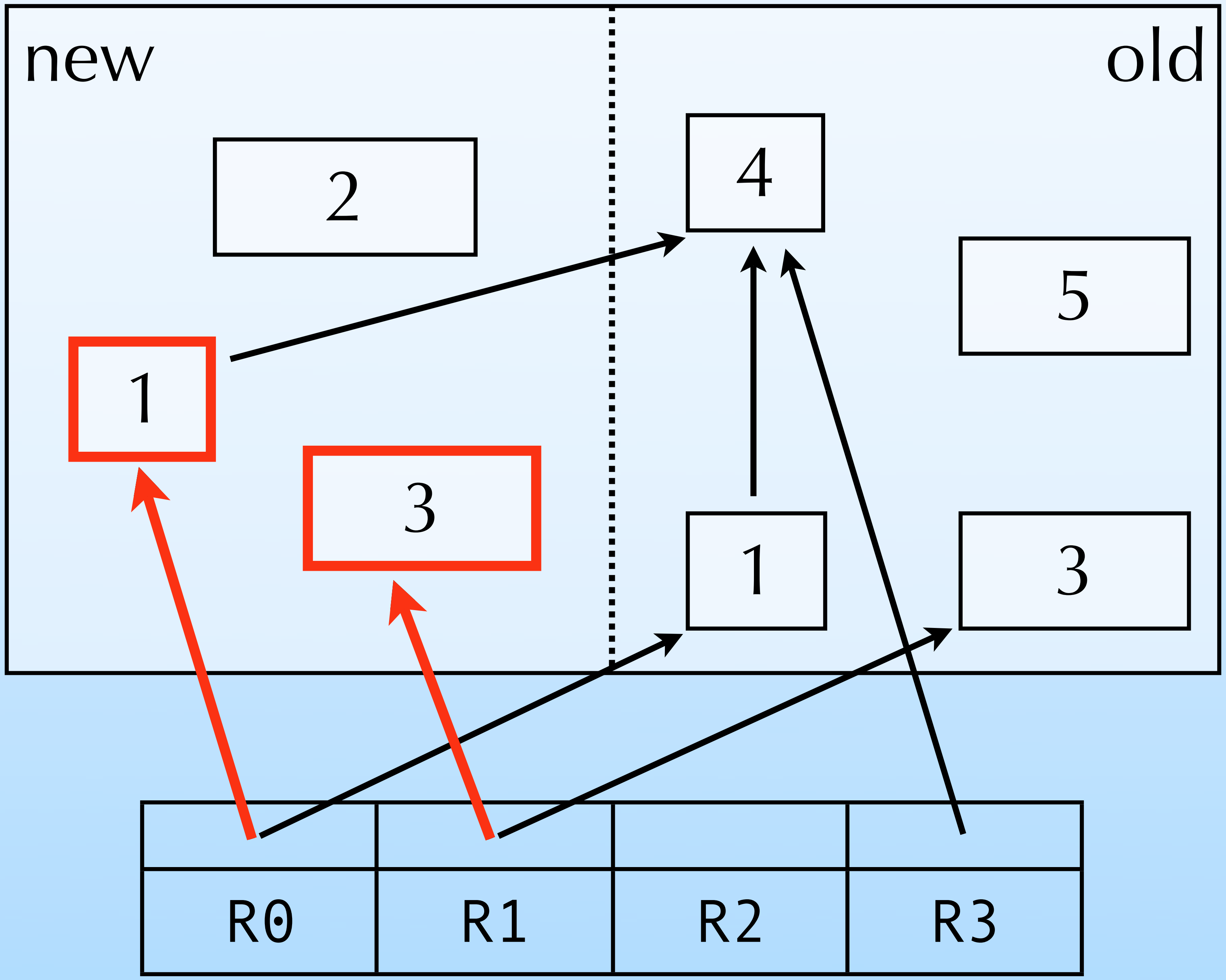
When older generation are full, they also get collected, usually along with the younger one(s).

Kinds of collections

In a generational GC, we distinguish two kinds of collections:

- **minor collections**, during which only the youngest generation is collected,
- **major collections**, during which some old generation, and usually all younger generations, are collected.

Generational GC
minor collection



Promotion policies

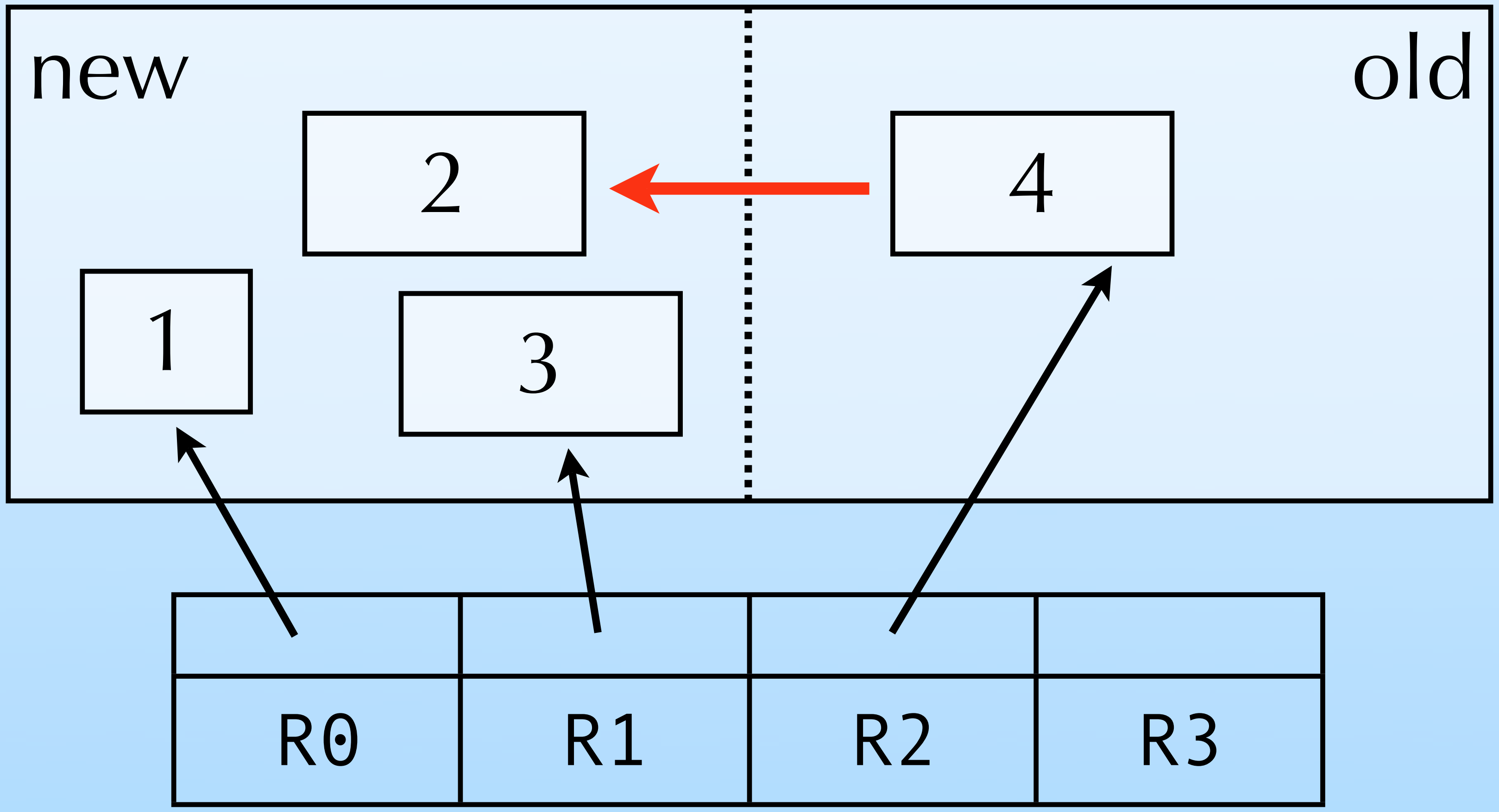
Generational GCs use a **promotion policy** to decide when objects should be advanced to an older generation.

The simplest one – all survivors are advanced – can promote very young objects, but is simple as object age does not need to be recorded.

To avoid promoting very young objects it is sufficient to wait until they survive a second collection before advancing them.

Roots for generational GC

The roots to be used for a minor collection must also include all **inter-generational pointers**, *i.e.* pointers from older generations to younger ones.



Inter-generational pointers

Inter-generational pointers can be handled in two ways:

1. by scanning older generations during minor collection,
2. by detecting pointer writes using a **write barrier** – implemented either in software or through hardware support – and remembering those which create inter-generational pointers.

Inter-generational pointers remembered sets

A **remembered set** contains all old objects pointing to young objects.

The write barrier maintains this set by adding objects to it iff:

- the object into which the pointer is stored is not yet in the remembered set,
- the pointer is stored in an old object, and points to a young one.

can also be done later by collector

Inter-generational pointers card marking

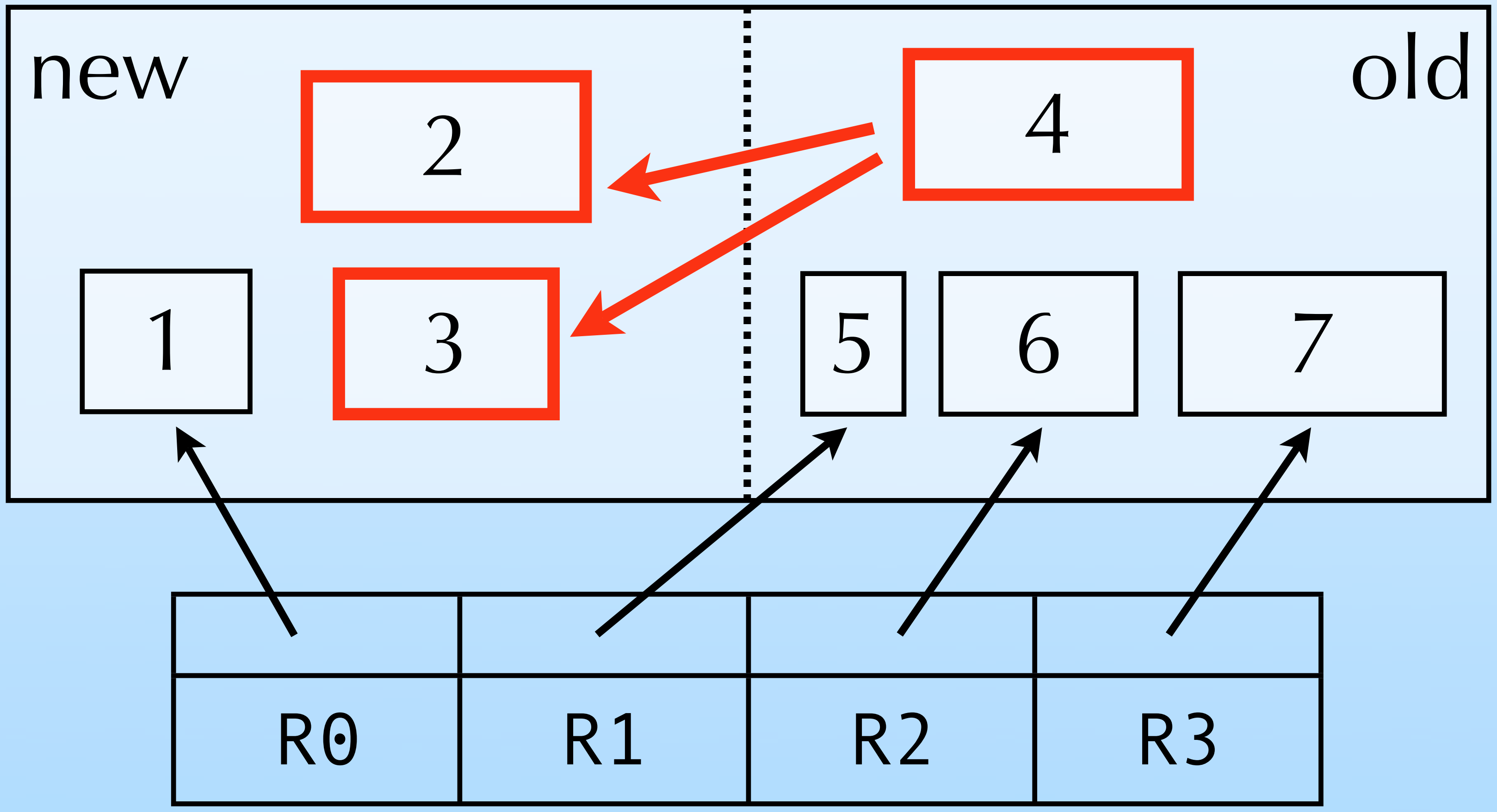
Card marking is another technique to detect inter-generational pointers.

Memory is divided into small, fixed sized areas called cards. A card table remembers, for each card, whether it potentially contains inter-generational pointers.

On each pointer write, the card is marked in the table, and marked cards are scanned for inter-generational pointers during collection.

Nepotism

Since old generations are not collected as often as young ones, it is possible for dead old objects to prevent collection of dead young objects.



Generational GC pros and cons

Generational GC tends to reduce GC pause times since only the youngest generation – which is also the smallest – is collected most of the time. It also avoids copying long-lived objects over and over.

The management of inter-generational pointers has its cost, however, and nepotism is a problem.

Other kinds of garbage collectors

Incremental and concurrent GCs

An **incremental garbage collector** can perform garbage collection in small, incremental steps, thereby reducing the length of GC pauses.

A **concurrent garbage collector** can work in parallel with the main program.

Incremental and concurrent GCs must both be able to deal with modifications to the reachability graph performed by the main program while they attempt to compute it.

Hybrid GCs

The various garbage collection techniques we have seen can be combined in hybrid GCs.

For example, the OCaml garbage collector is a generational GC where allocation happens linearly in the first generation, like in a copying GC. All objects which survive a GC are copied to a second generation, which is collected by an incremental mark & sweep GC.

Additional garbage collector features

Finalisers

Some GCs make it possible to associate **finalisers** with objects.

Finalisers are functions which are called when an object is about to be collected. They are generally used to free “external” resources associated with the object about to be freed.

Since there is no guarantee about when finalisers are invoked, the resource in question should not be scarce.

Finalisers design and implementation issues

Finalisers are tricky:

- what do we do if a finaliser makes the finalised object reachable again – *e.g.* by storing it in a global variable?
- how do finalisers interact with concurrency – *e.g.* in which thread are they run?
- how can they be implemented efficiently in a copying GC, which doesn't visit dead objects?

References

When the GC encounters a reference, it usually treats it as a **strong** reference, meaning that the referenced object will be considered as reachable and survive the collection.

It is sometimes useful to have weaker kinds of references, which can refer to an object without preventing it from being collected.

Weak references

The term **weak reference** designates references which do not prevent an object from being collected.

During a GC, if an object is only reachable through weak references, it is collected, and all (weak) references pointing to it are cleared.

Weak references are useful to implement caches, canonicalising mappings, etc.

Example: Java references

Java provides several kinds of “non-strong” references, which are, from strongest to weakest:

- **soft references**, cleared when memory is low,
- **weak references**, cleared as early as possible,
- **phantom references**, similar to weak references except that the referenced object is *not* available – and therefore cannot be resurrected.

Summary

Copying GCs copy reachable objects from one semi-space to the other on every collection. This avoids all fragmentation, and makes allocation very fast.

Generational GCs put young objects in a separate, smaller area, collected more often. This reduces collection pauses, and avoids the repeated copying of long-lived objects.