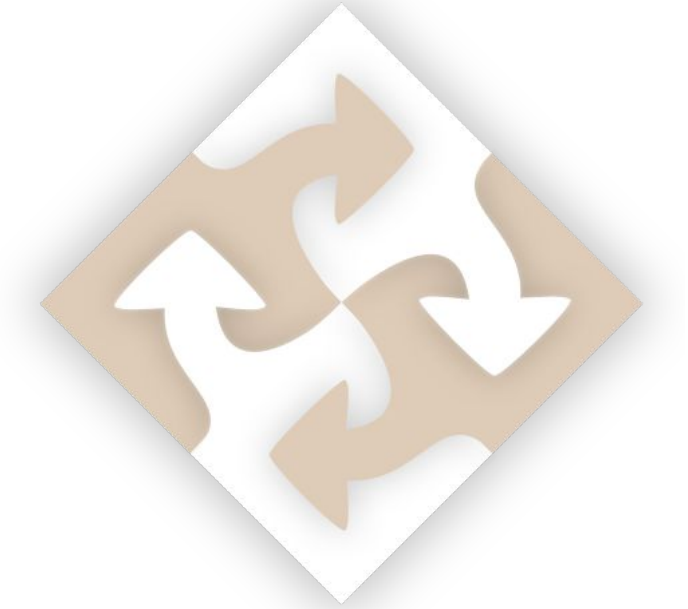


Memory Pool System

Ravenbrook

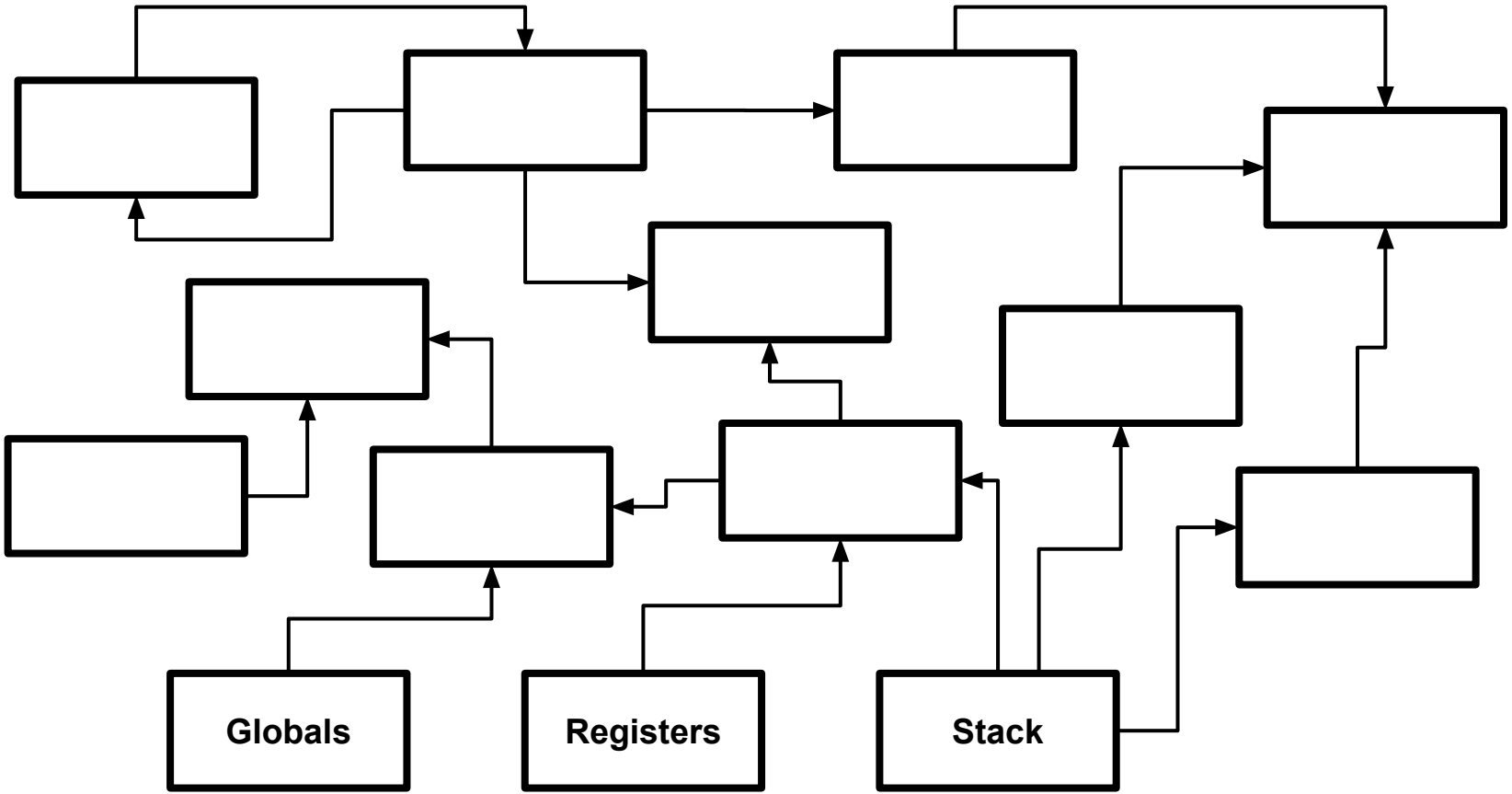
https://ravenbrook.com

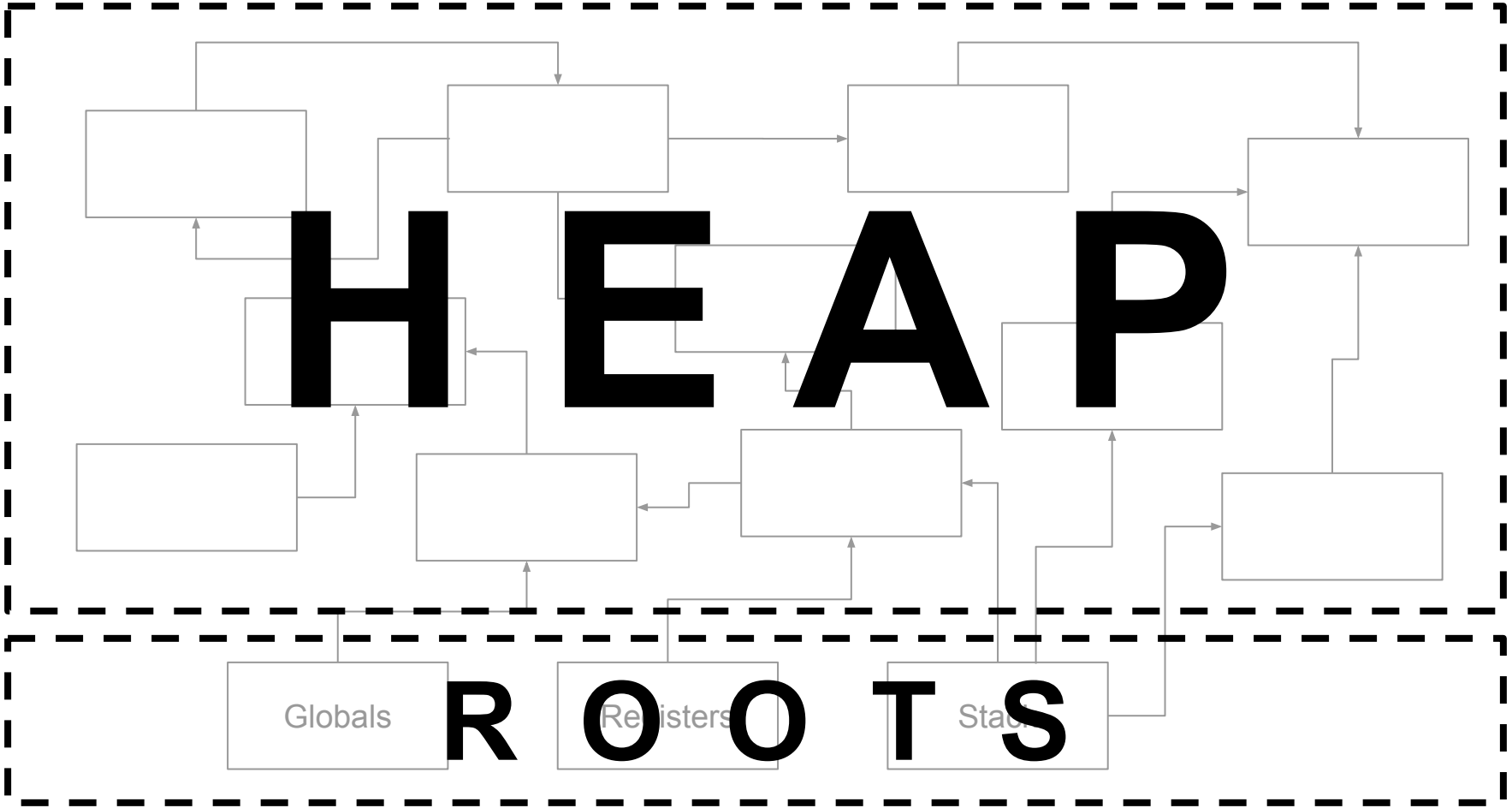
Memory Pool System

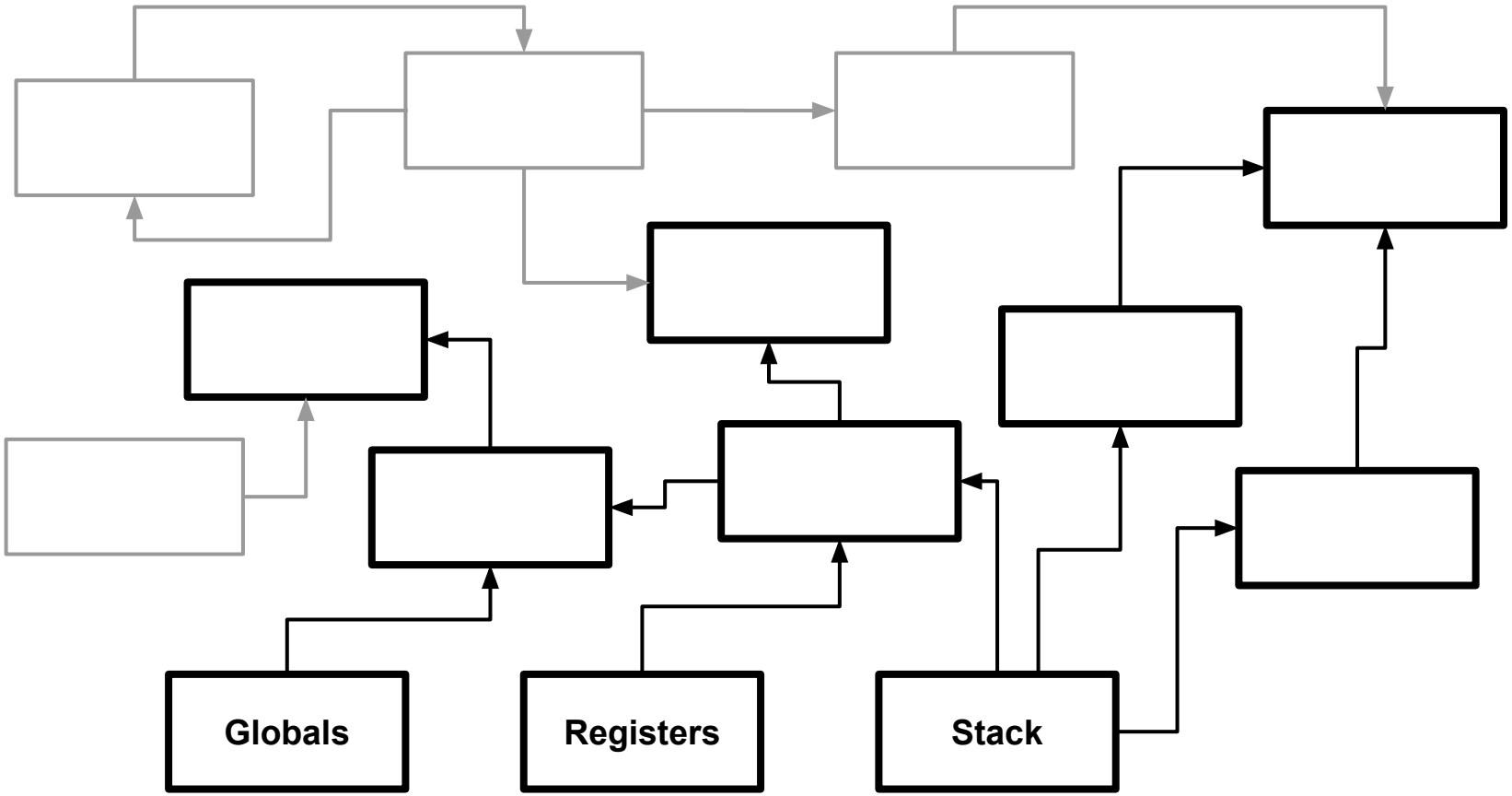
a reliable, soft-real-time, semi-conservative, mostly-copying garbage collector

Memory Pool System

a reliable, soft-real-time, semi-conservative, mostly-copying **garbage collector**

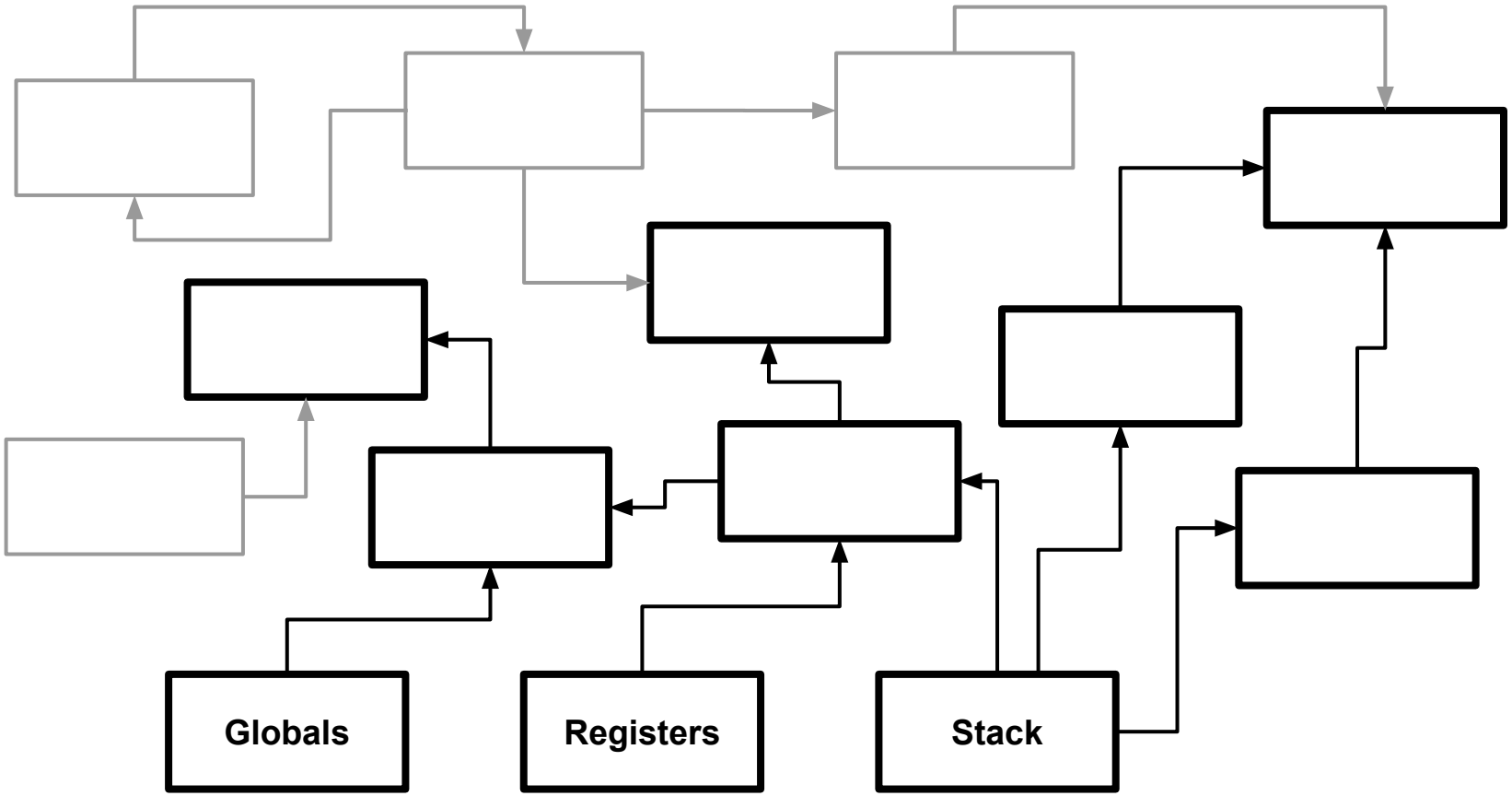

Globals
Registers
Stack

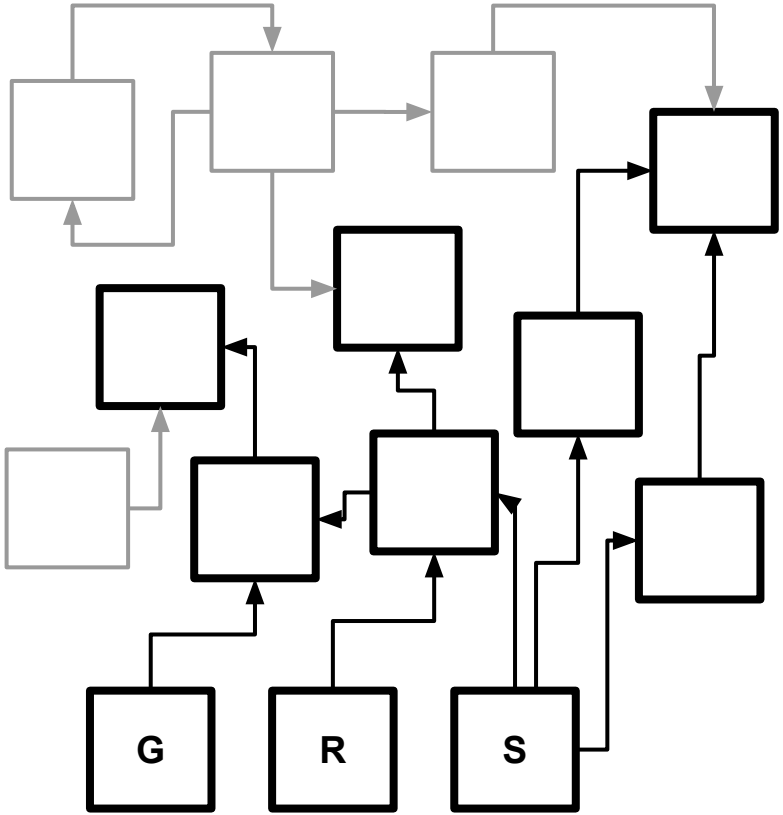

HEAP
Globals
Registers
Stack
ROOTS

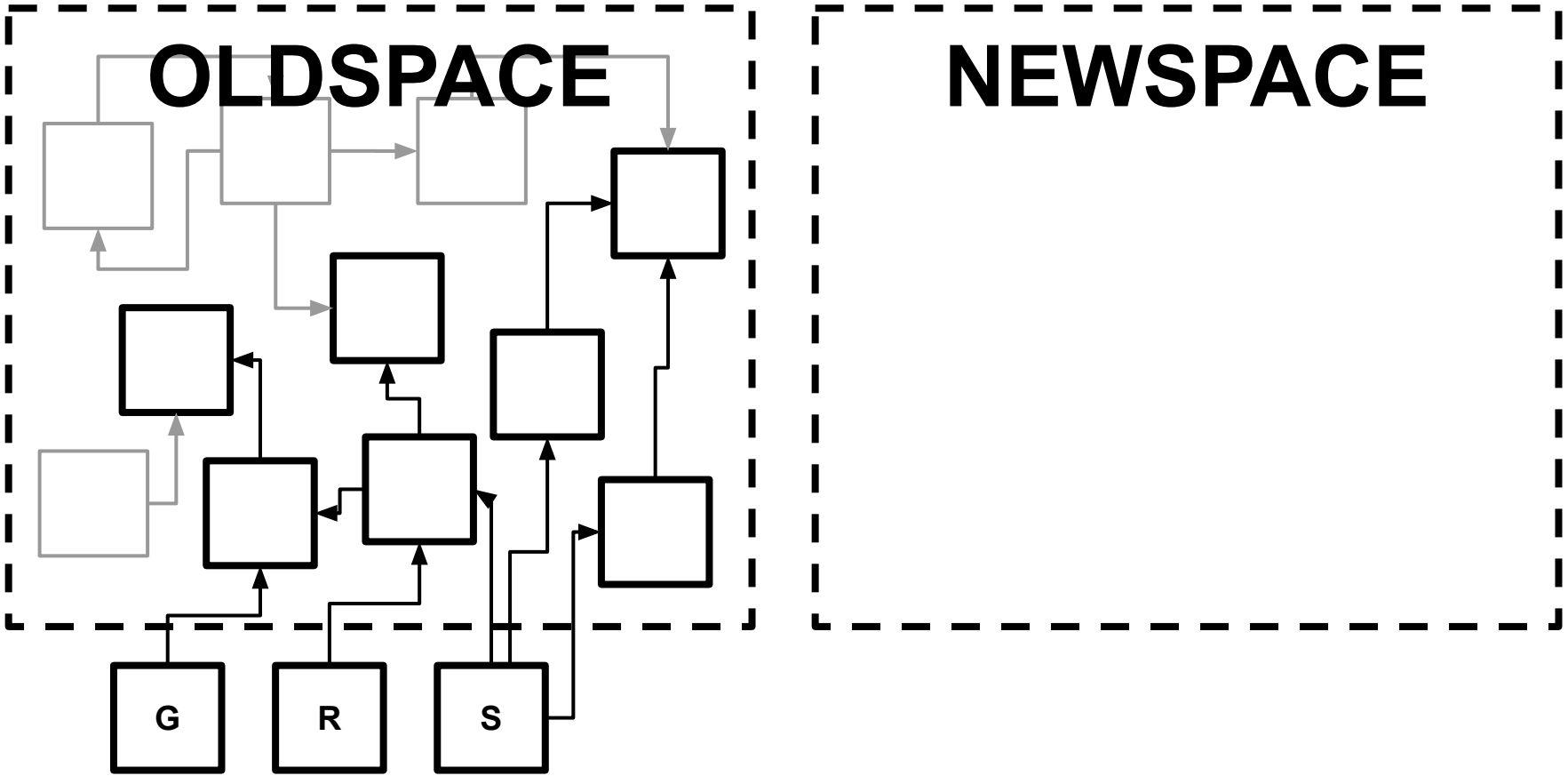

Globals
Registers
Stack

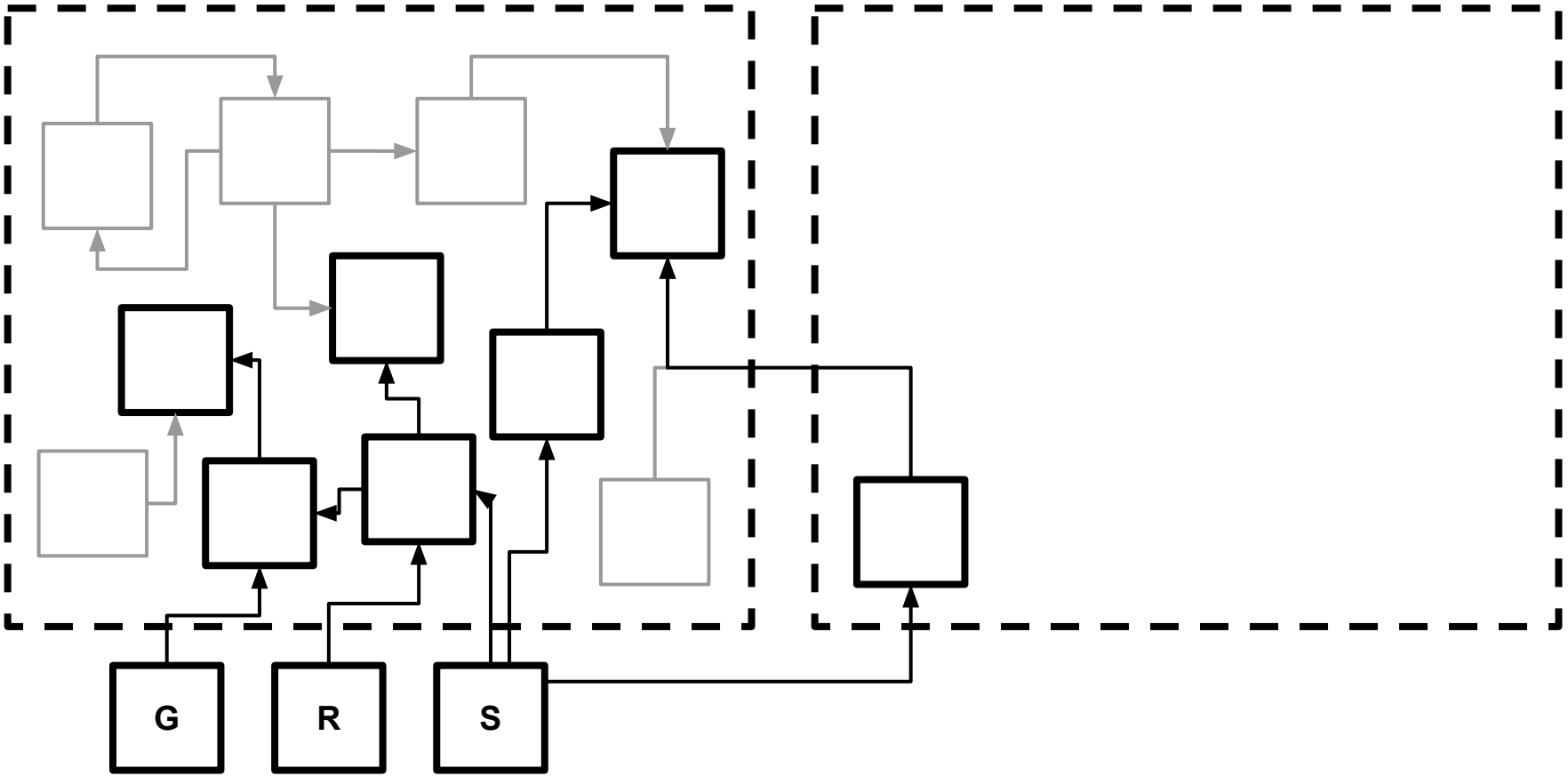
Memory Pool System

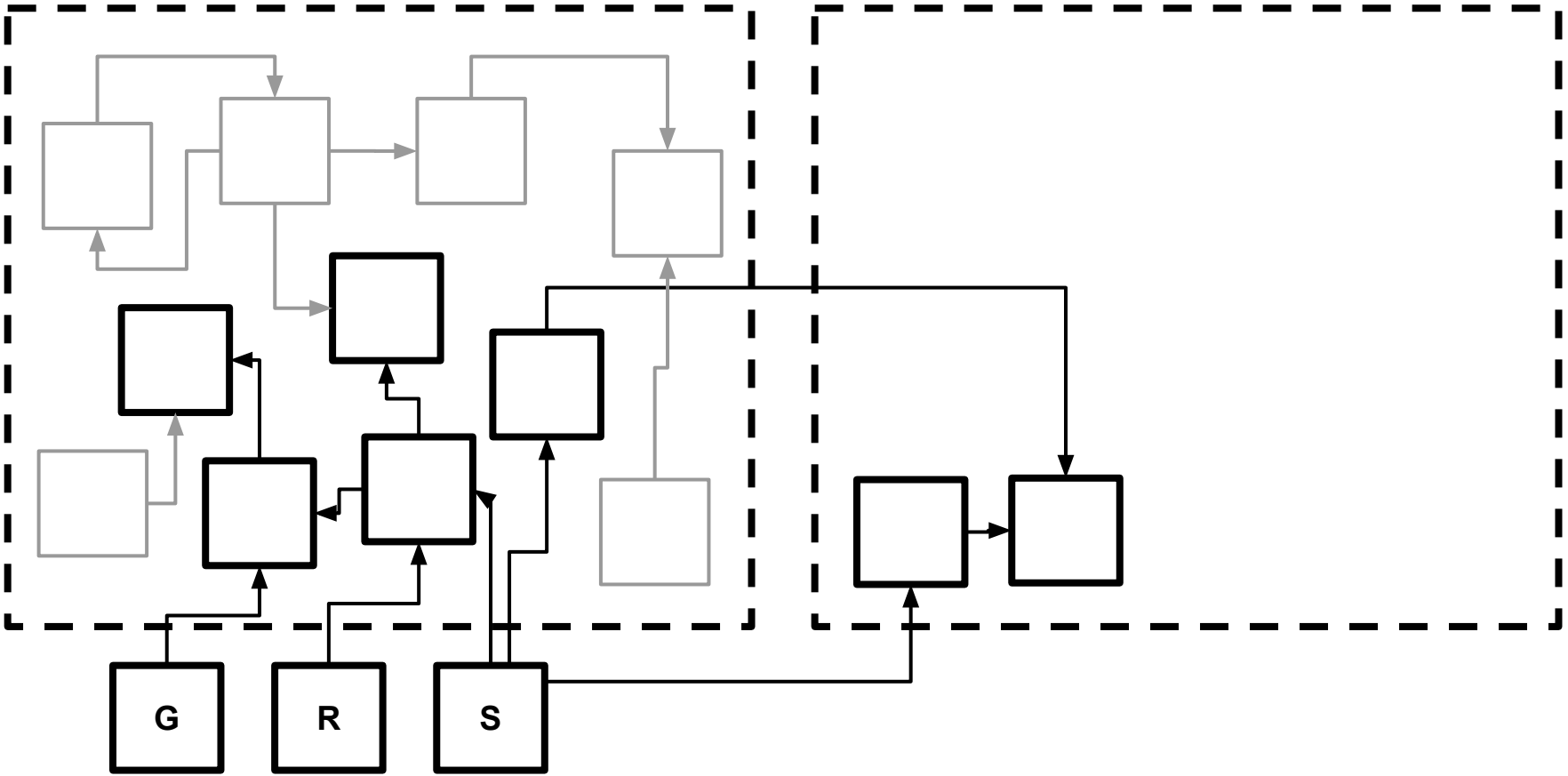
a reliable, soft-real-time, semi-conservative,
mostly-copying garbage collector

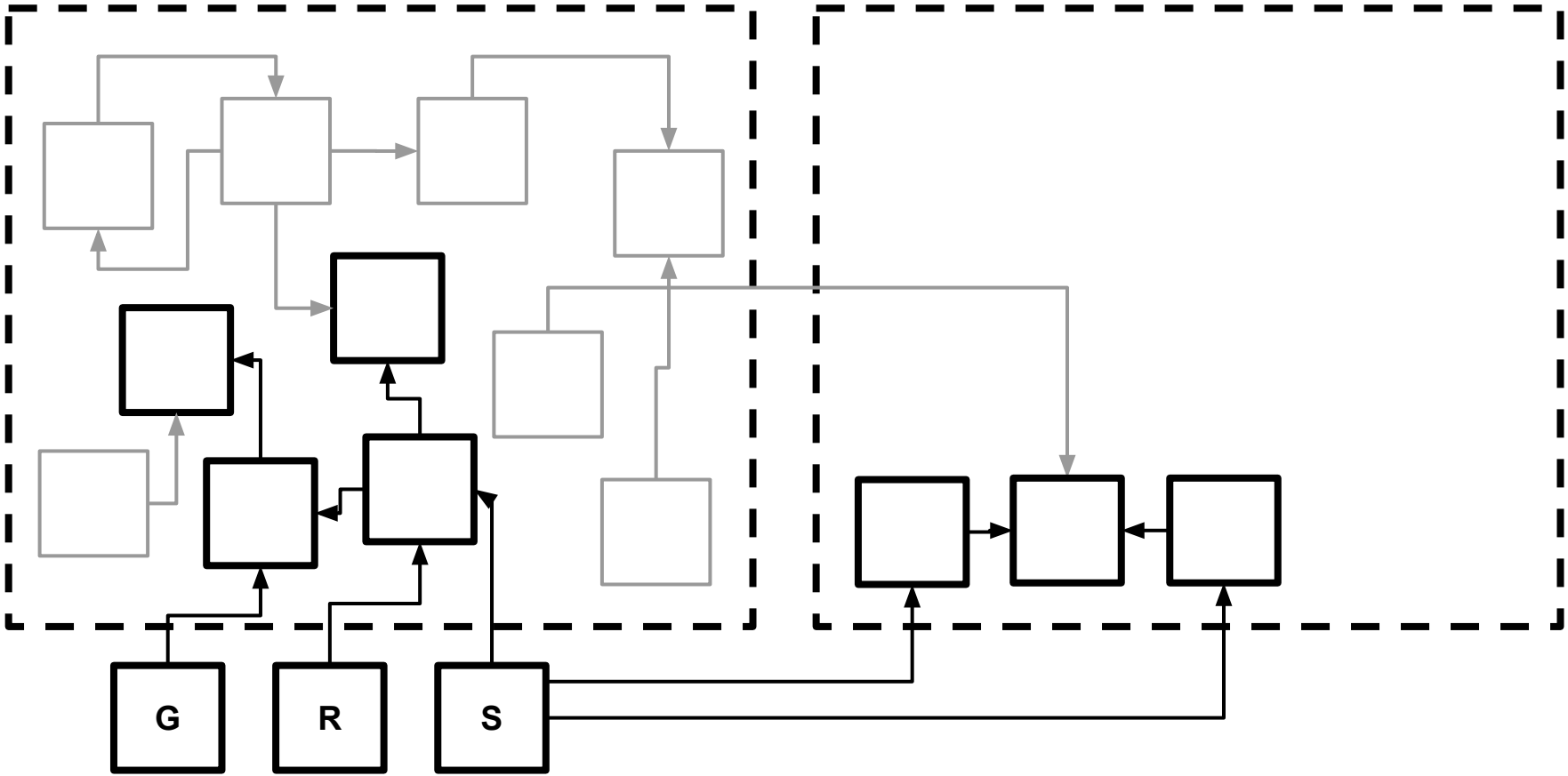

Globals
Registers
Stack

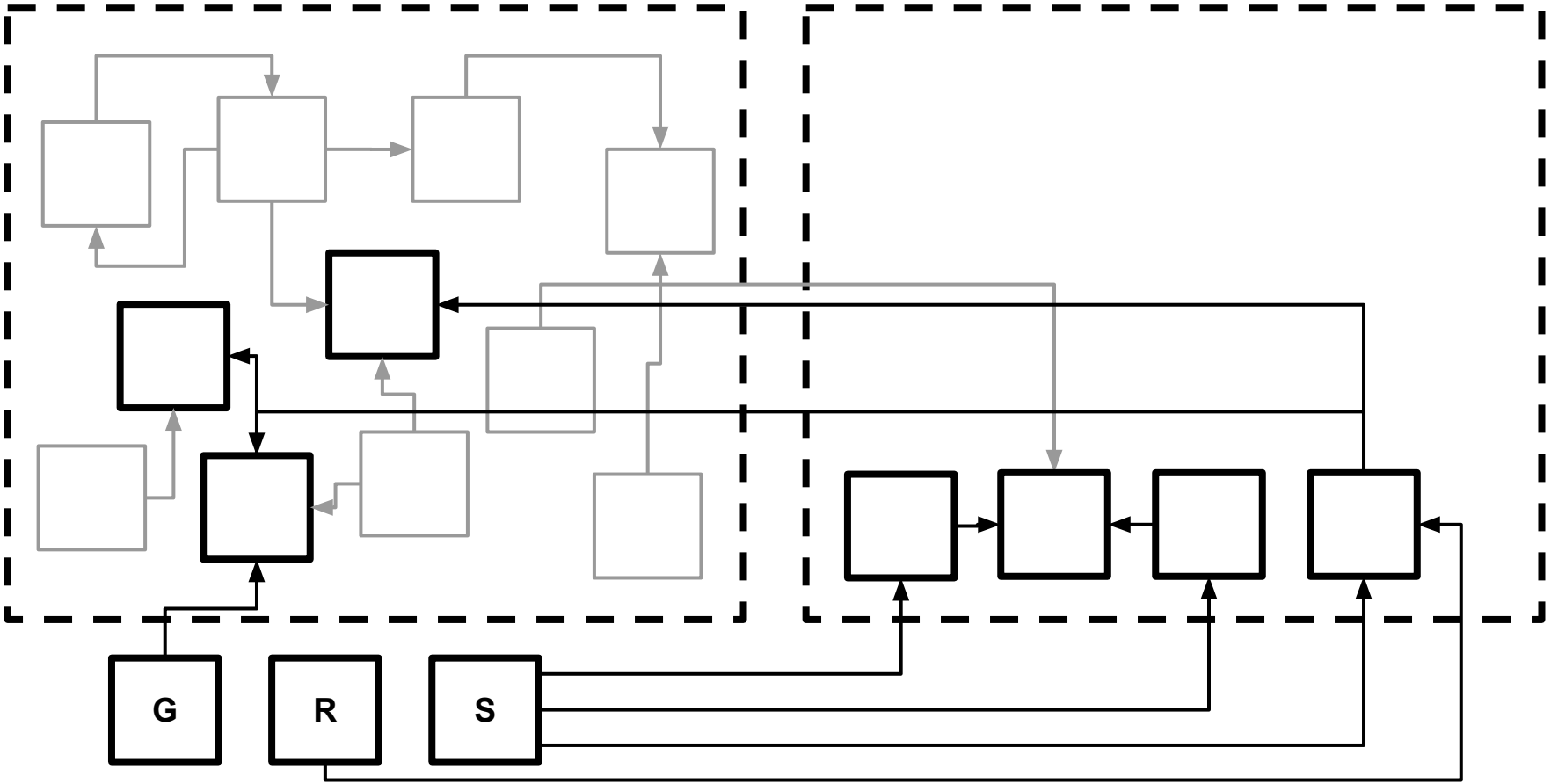

G
R
S

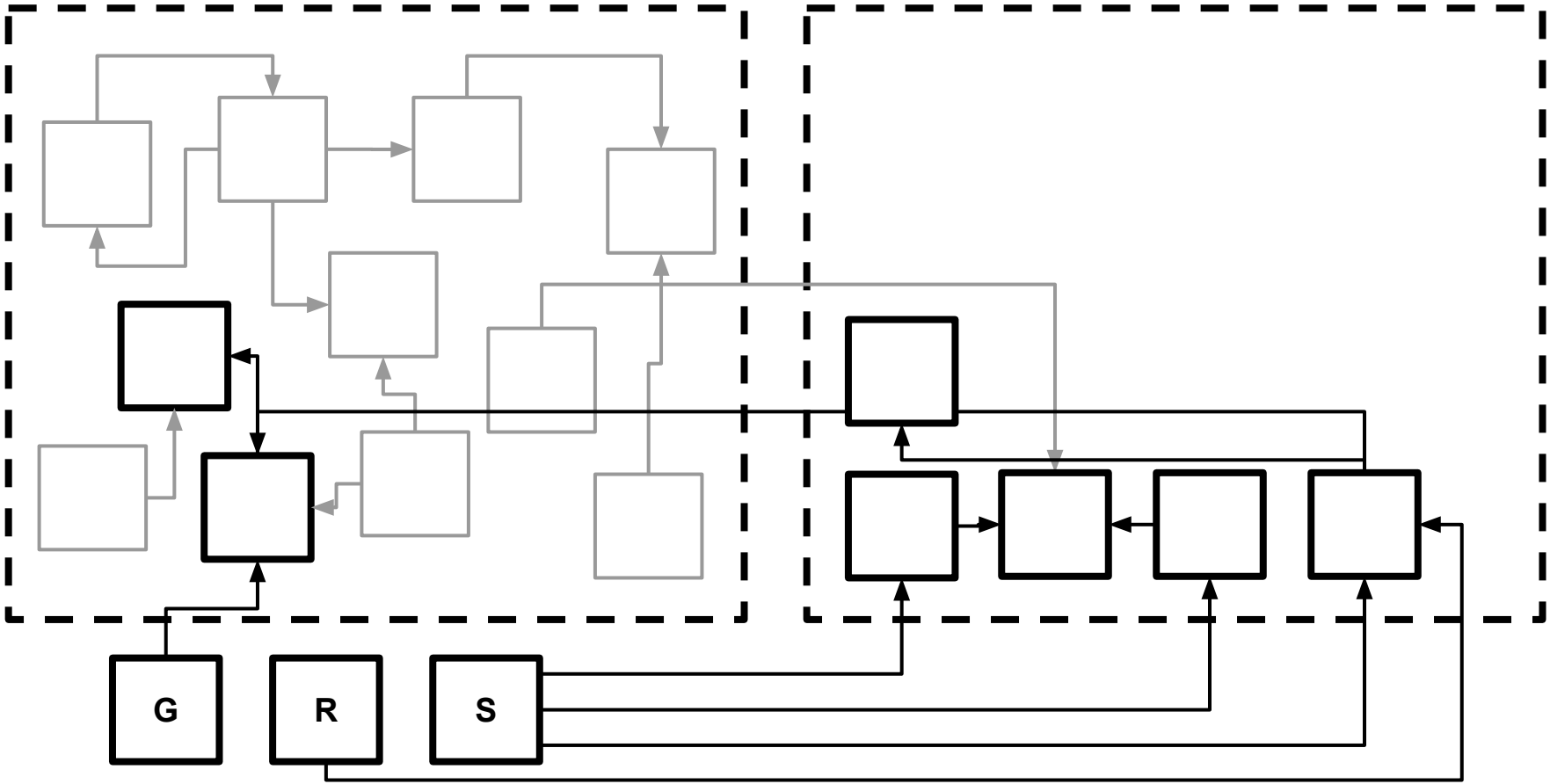

OLDSPACE
NEWSPACE
G
R
S

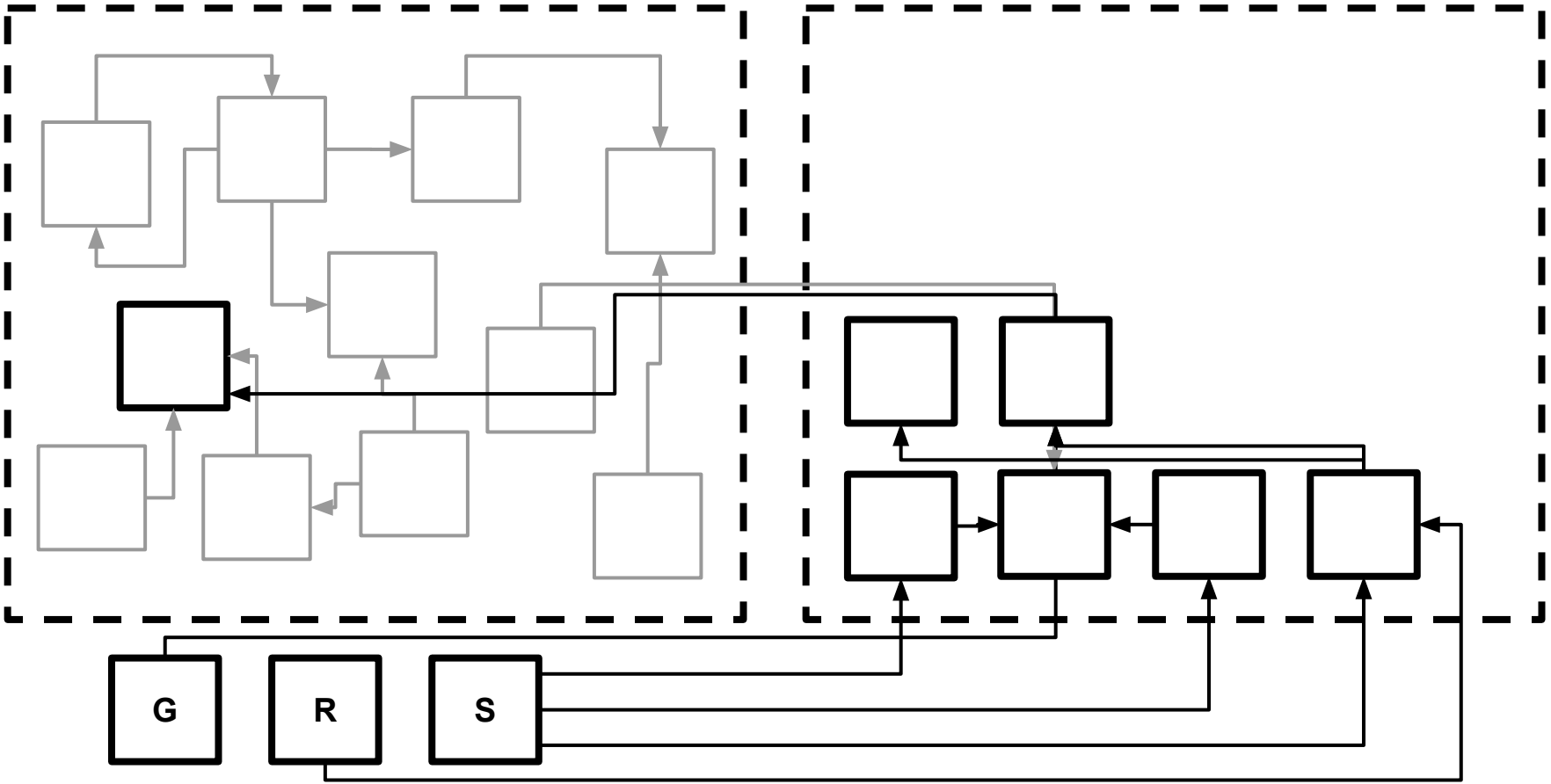

G
R
S

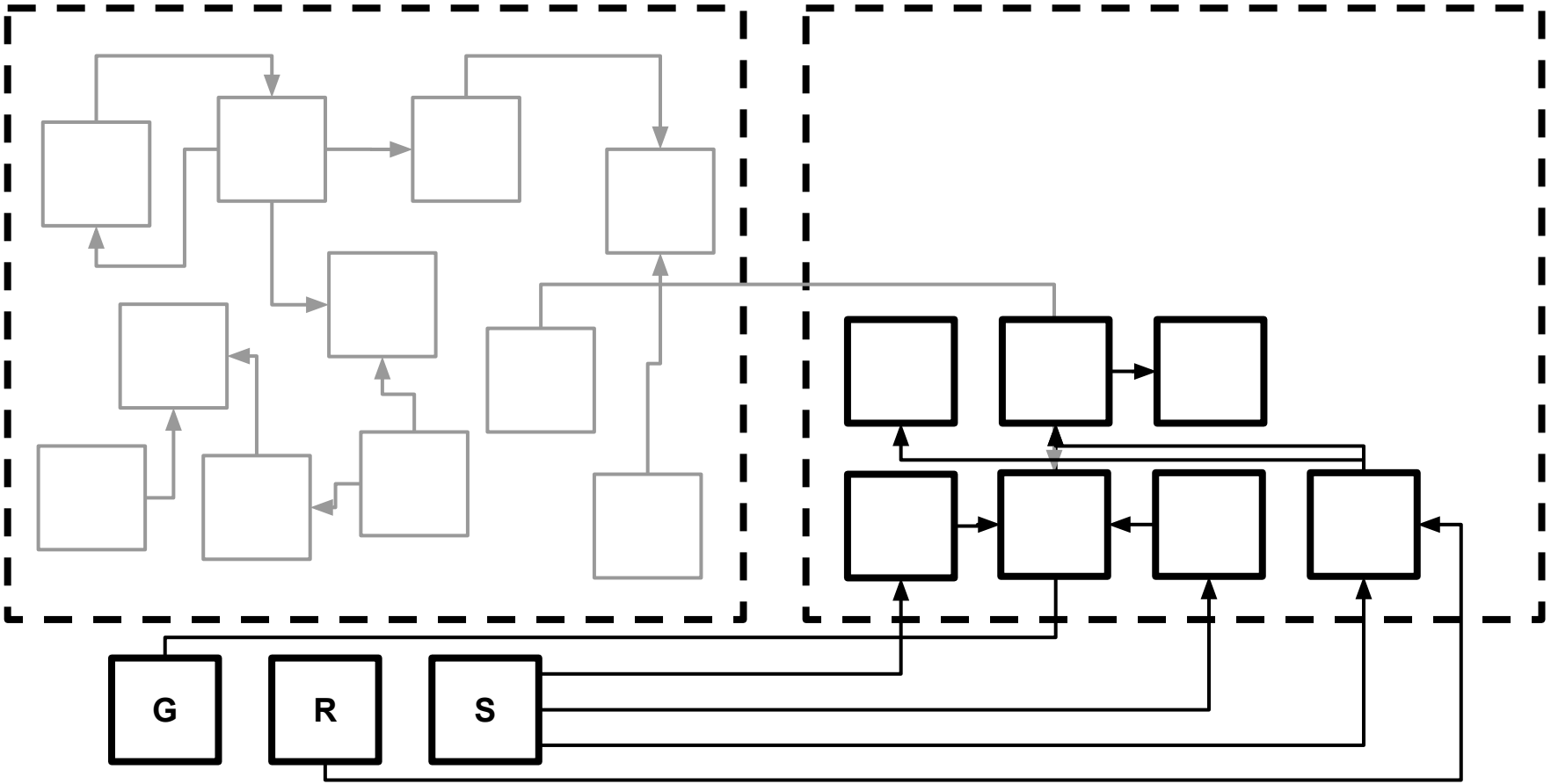

G
R
S

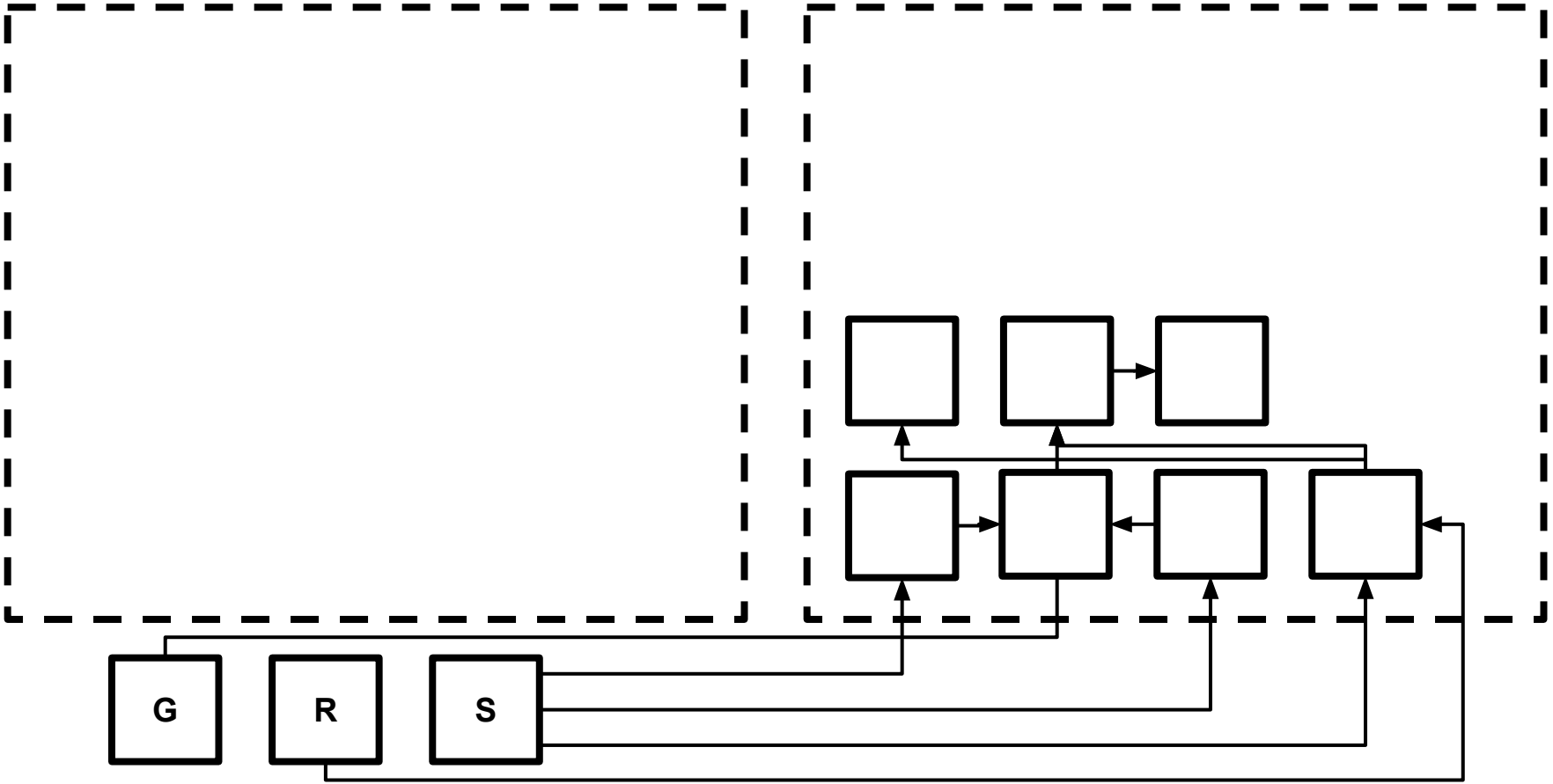

G
R
S


G
R
S


G
R
S


G
R
S

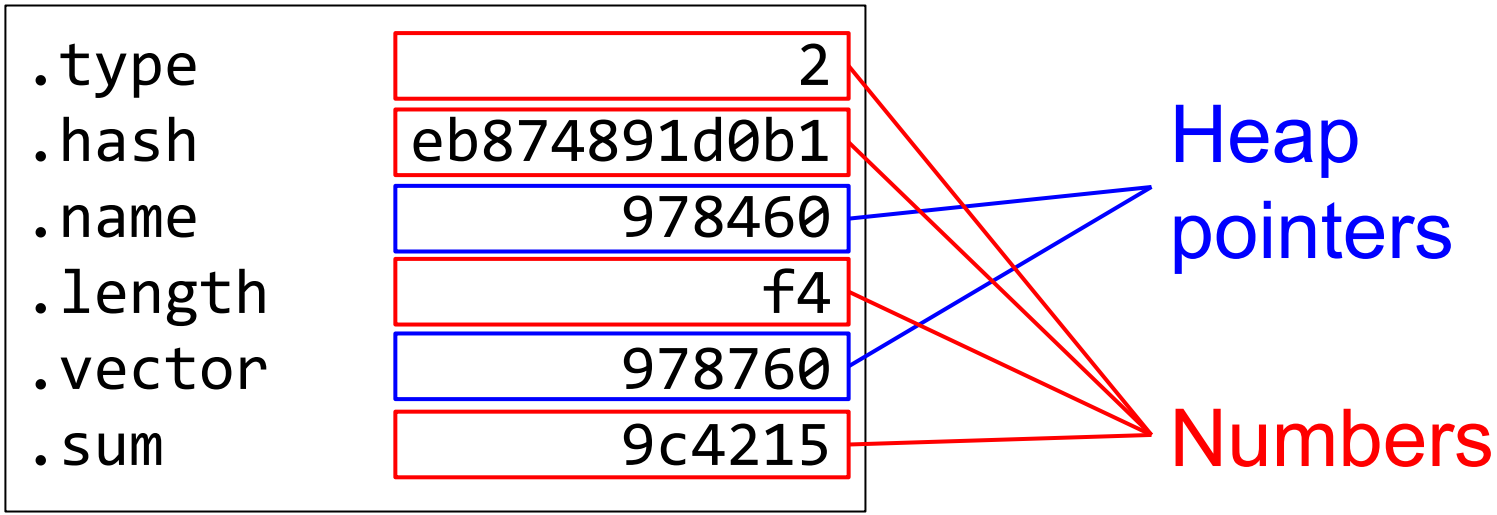

G
R
S


G
R
S

Memory Pool System

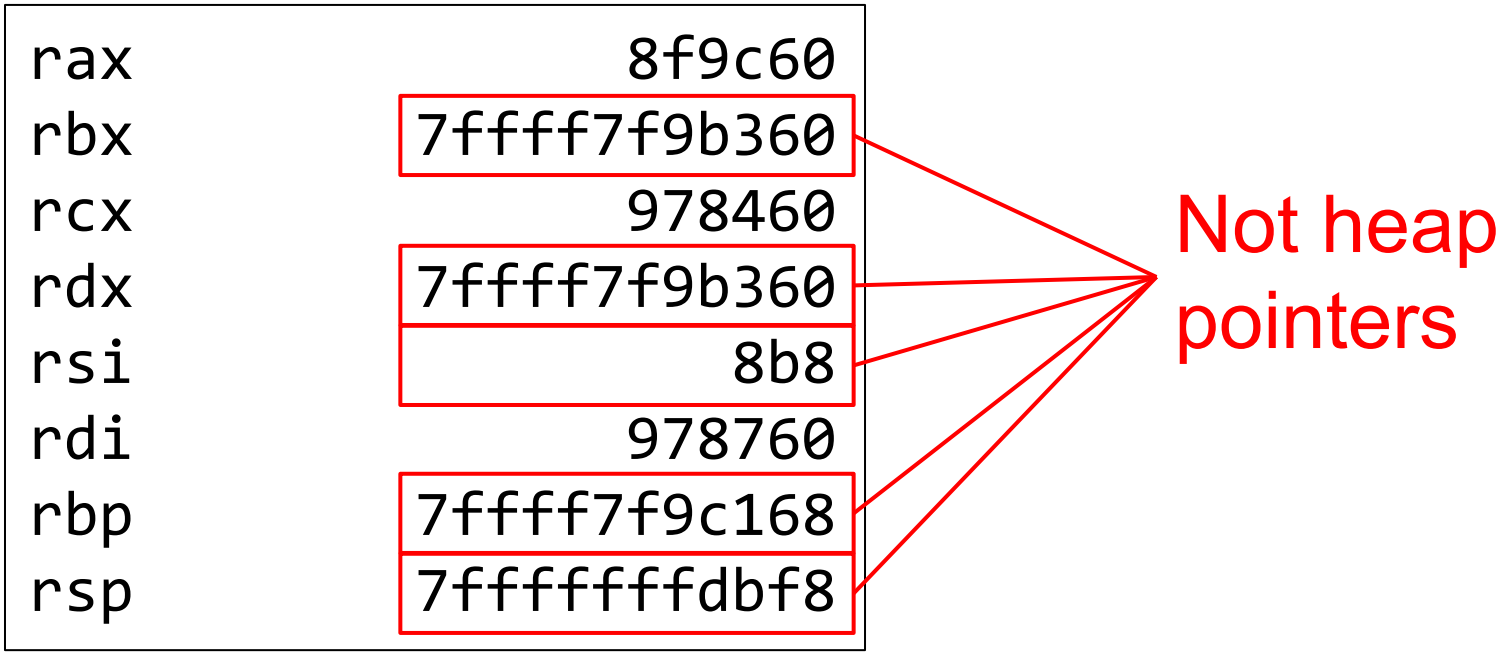
a reliable, soft-real-time, **semi-conservative**,
mostly-copying garbage collector

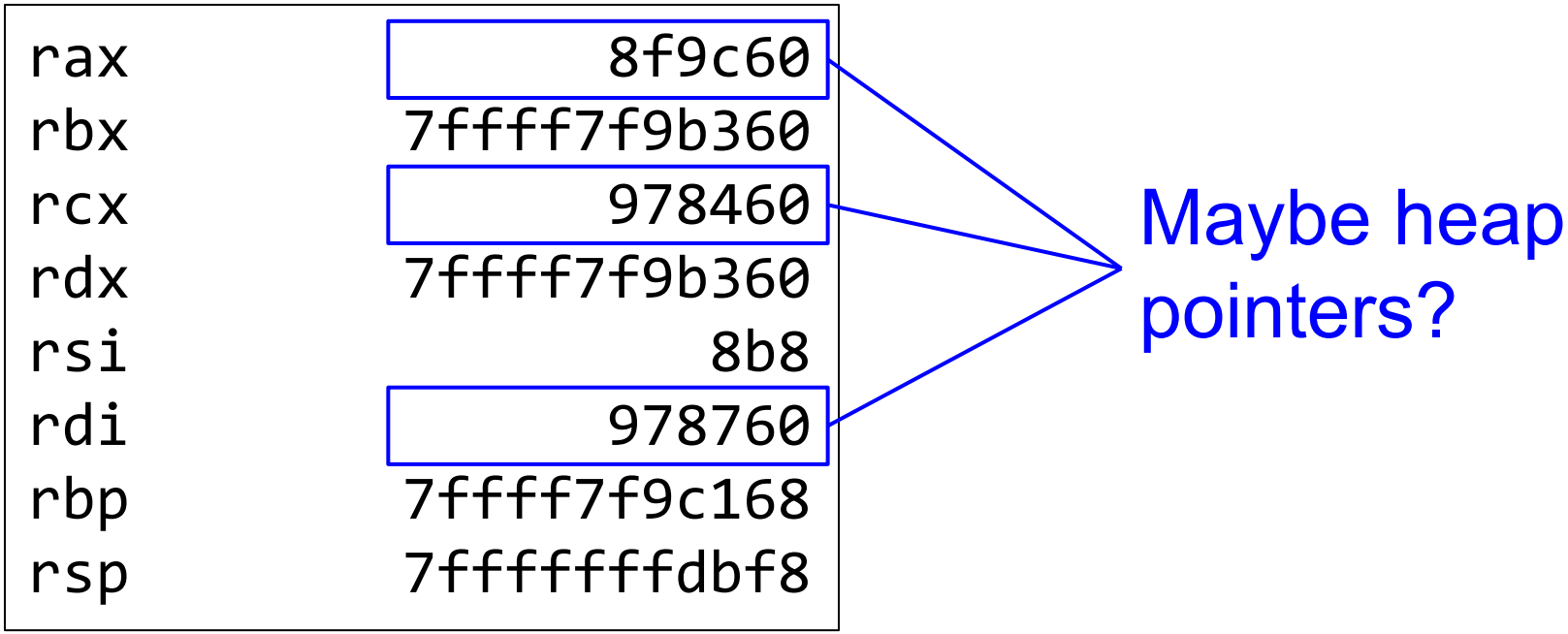
```
.type                    2
.hash        eb874891d0b1
.name              978460
.length                f4
.vector            978760
.sum               9c4215
```



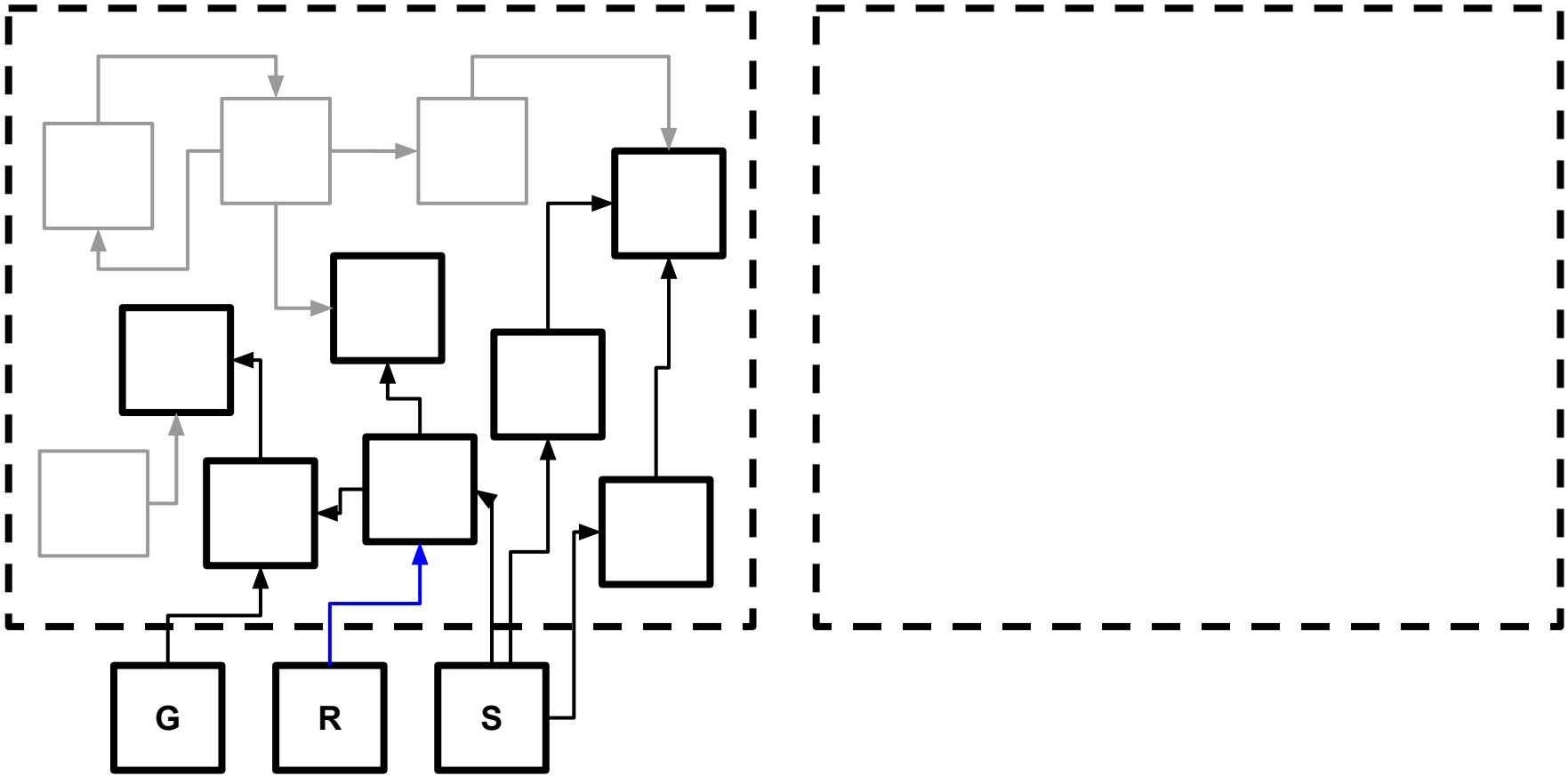
Exact references

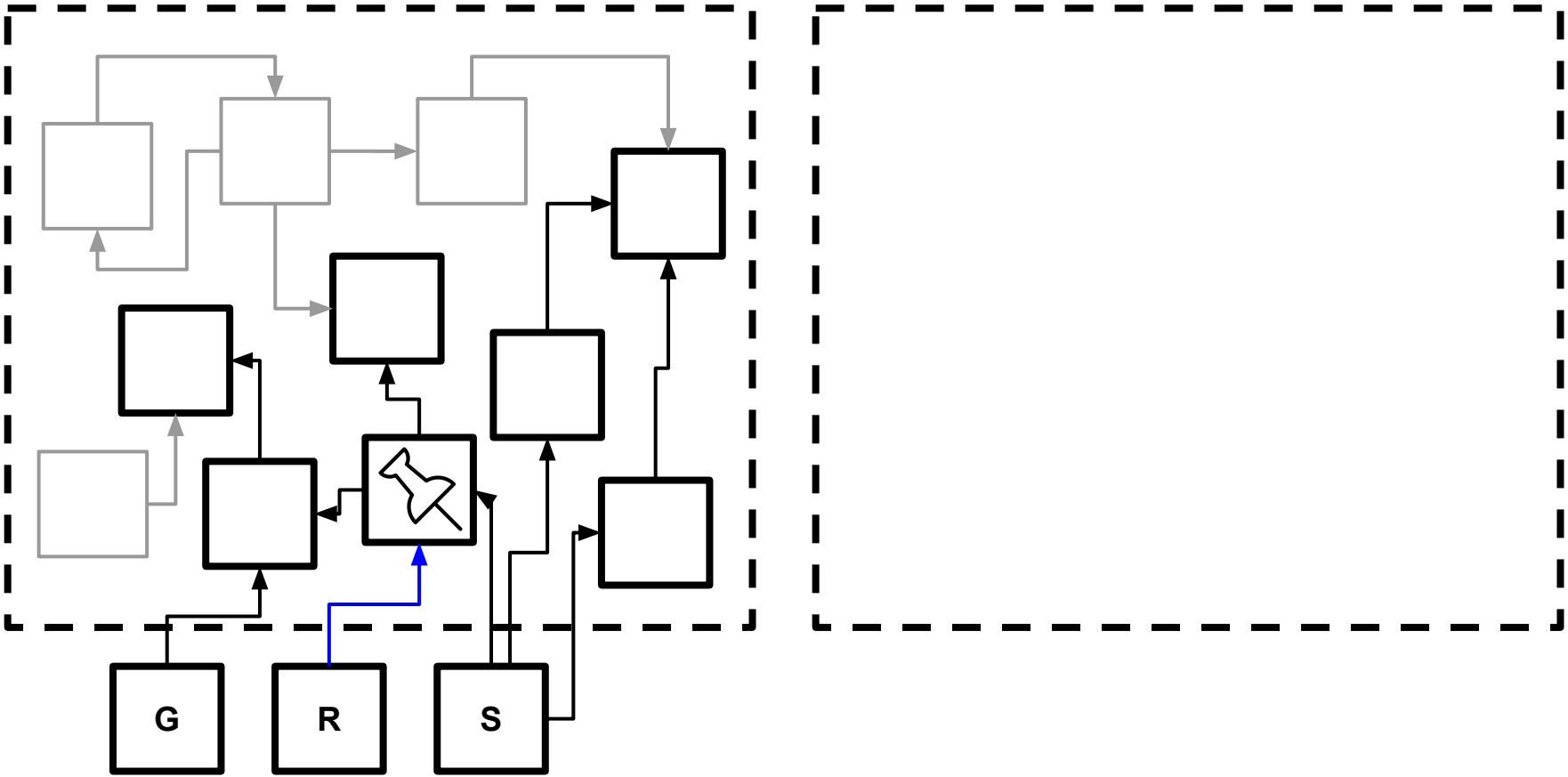
rax	8f9c60
rbx	7ffff7f9b360
rcx	978460
rdx	7ffff7f9b360
rsi	8b8
rdi	978760
rbp	7ffff7f9c168
rsp	7fffffffdbf8

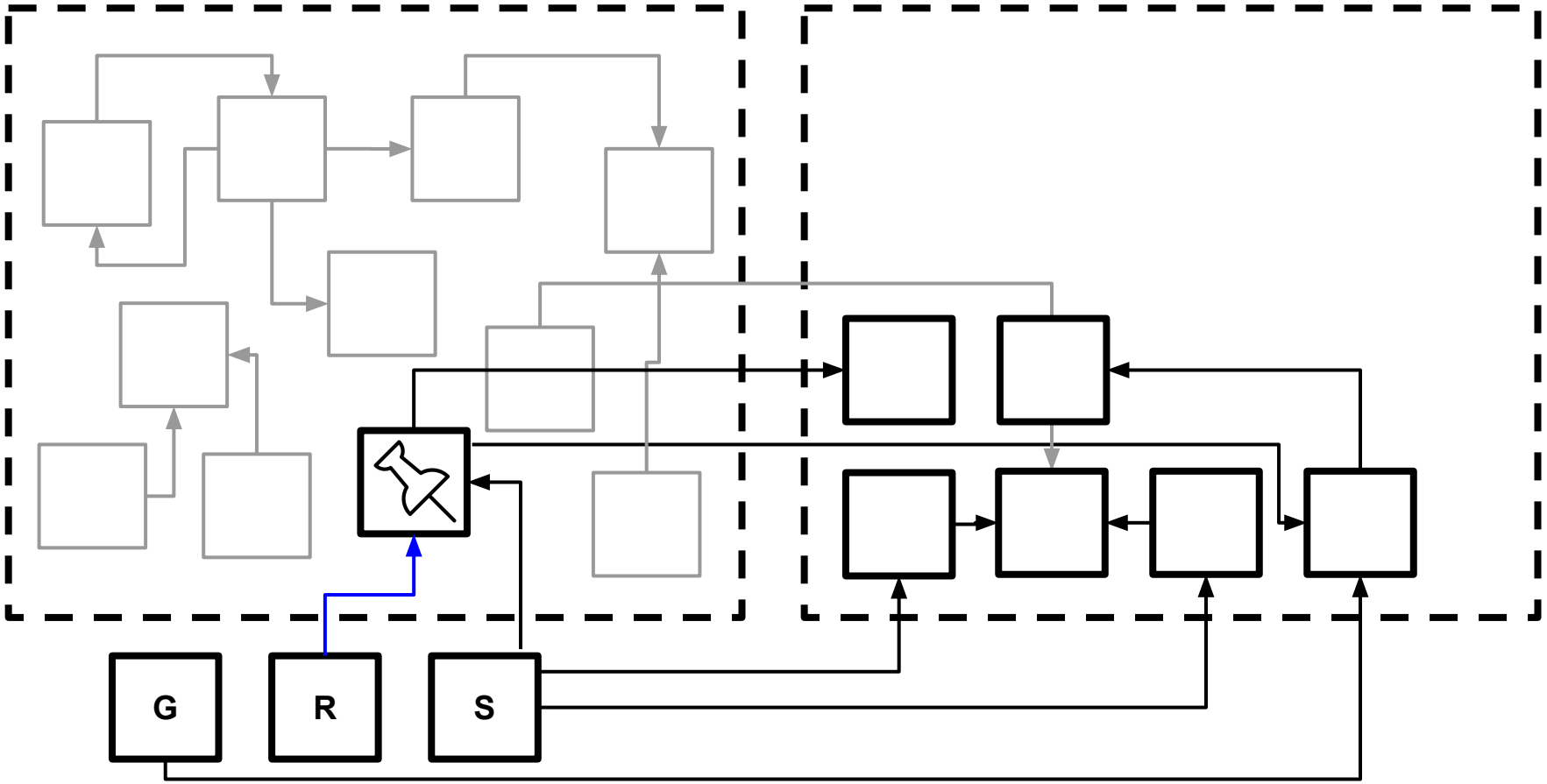

rax 8f9c60
rbx 7ffff7f9b360
rcx 978460
rdx 7ffff7f9b360
rsi 8b8
rdi 978760
rbp 7ffff7f9c168
rsp 7fffffffdbf8
Not heap pointers



Ambiguous references


G
R
S

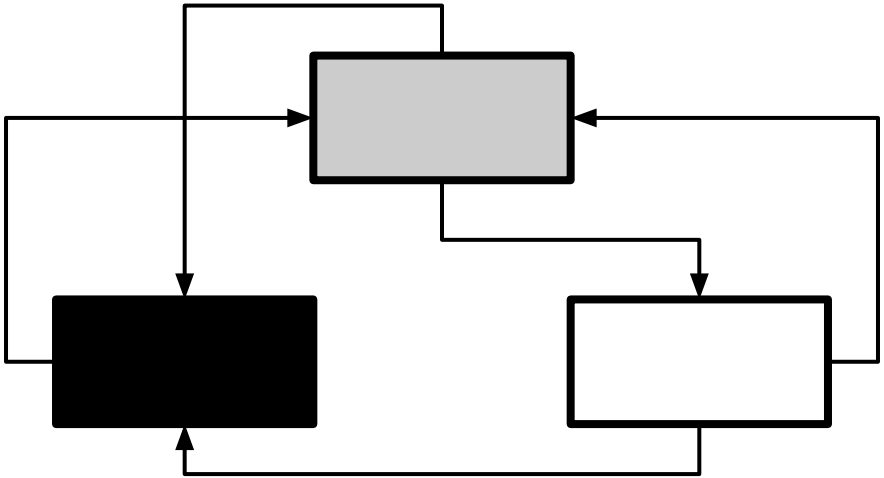

G
R
S


G
R
S

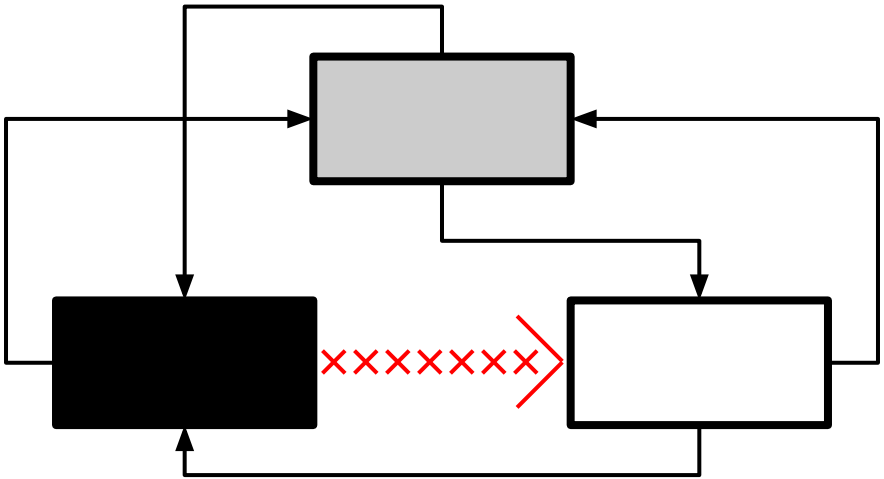
Memory Pool System

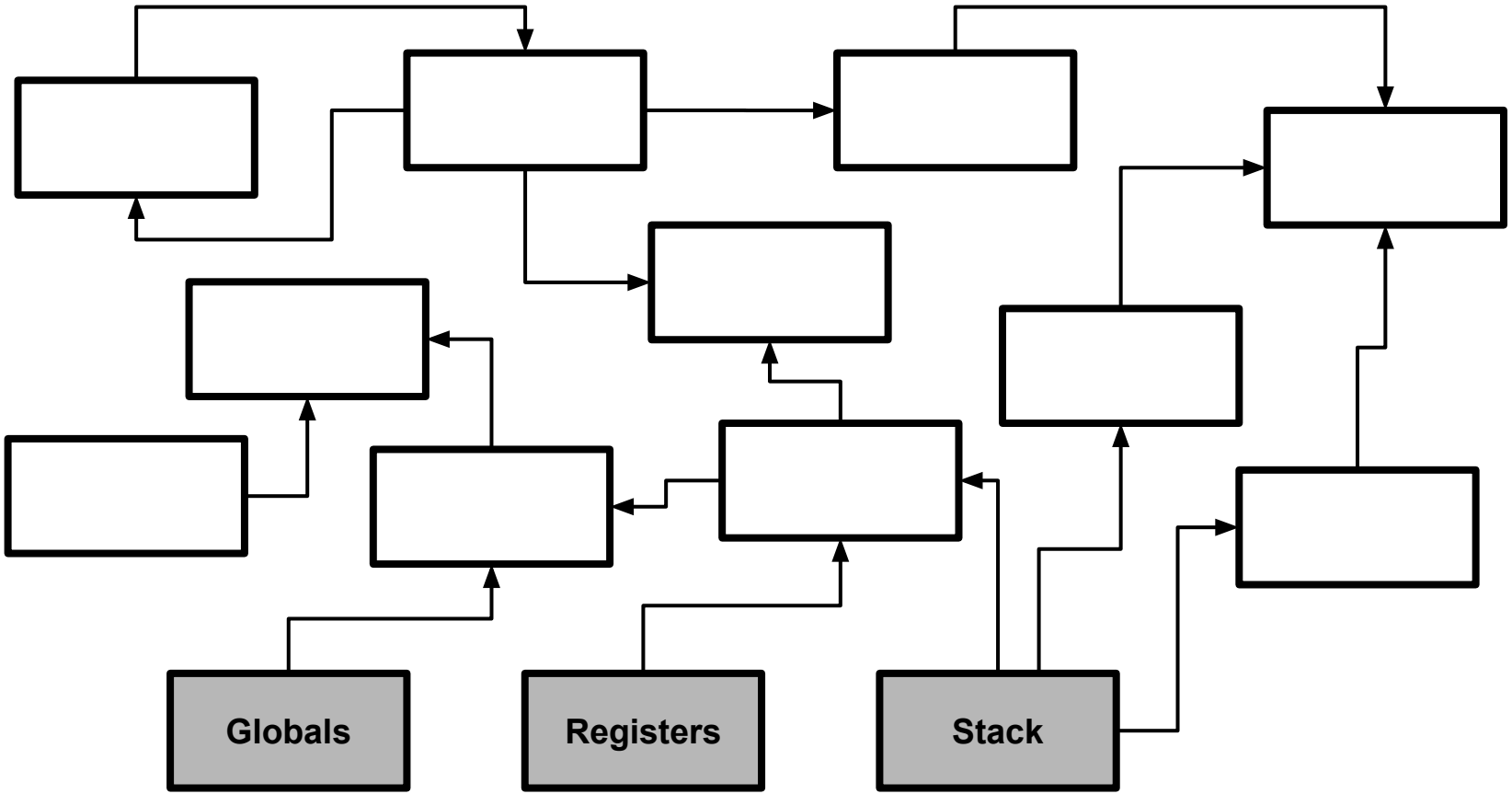
a reliable, **soft-real-time**, semi-conservative, mostly-copying garbage collector

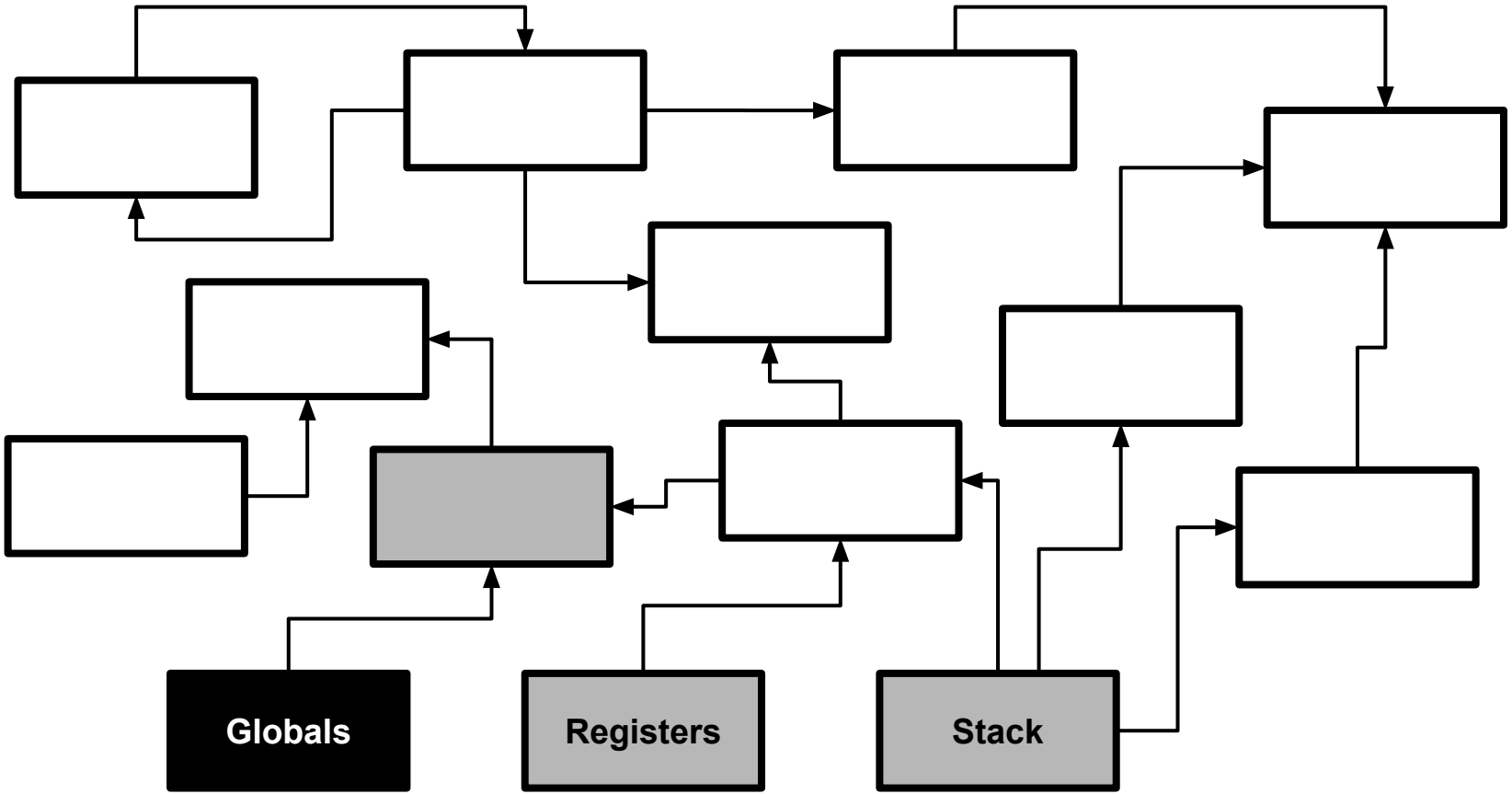
Dijkstra’s Tri-colour Algorithm

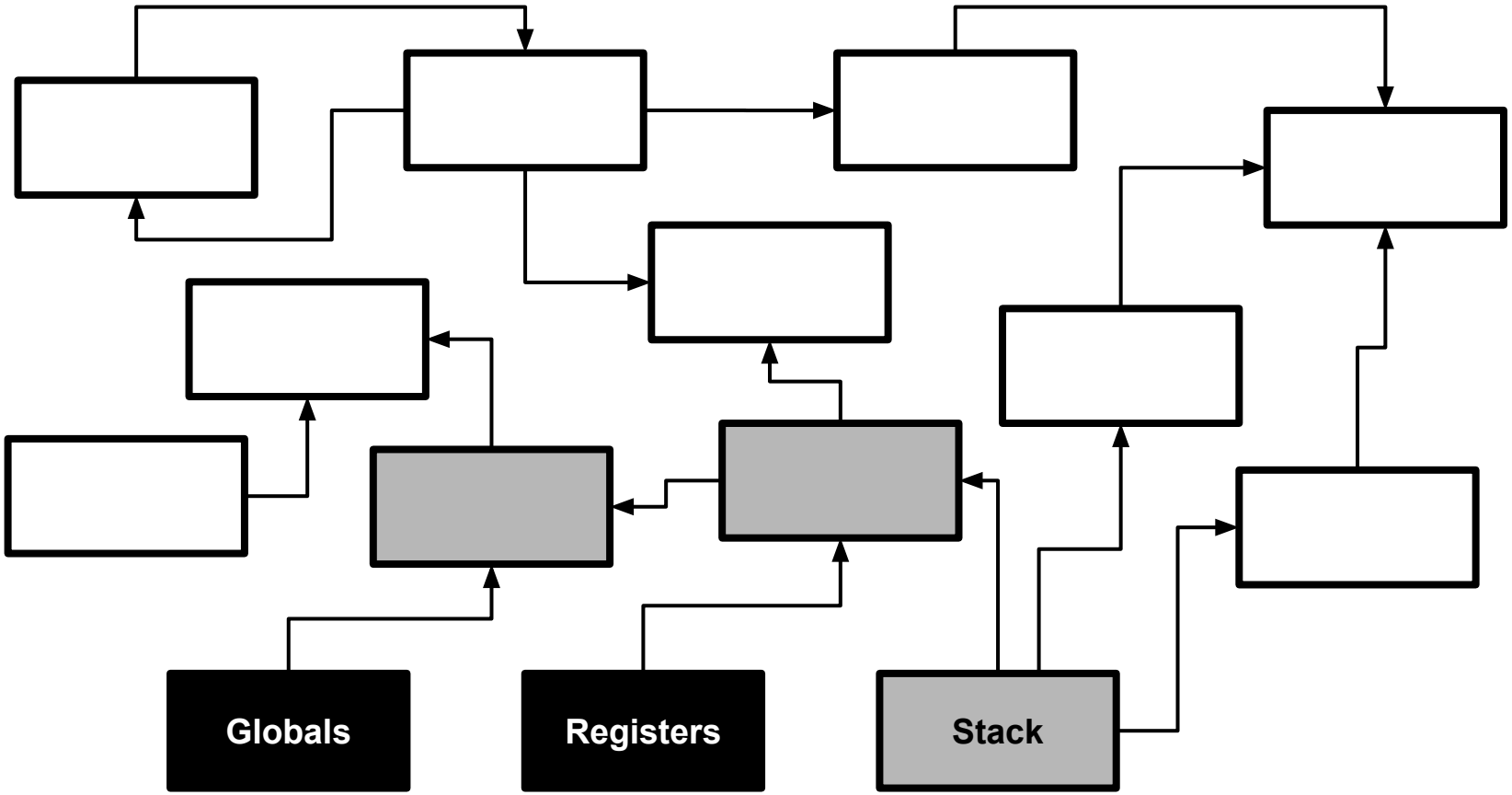


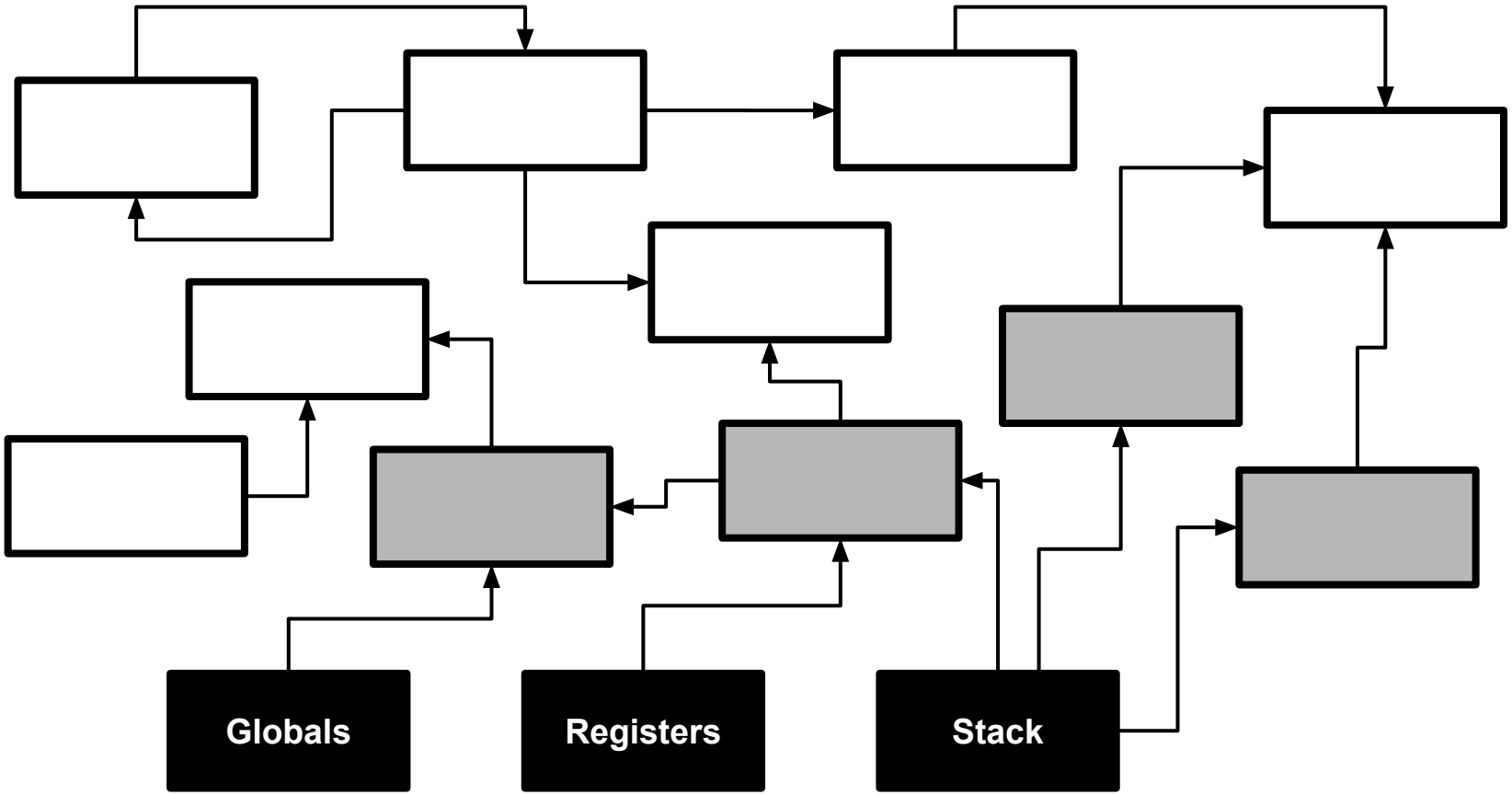
Strong Tri-colour Invariant

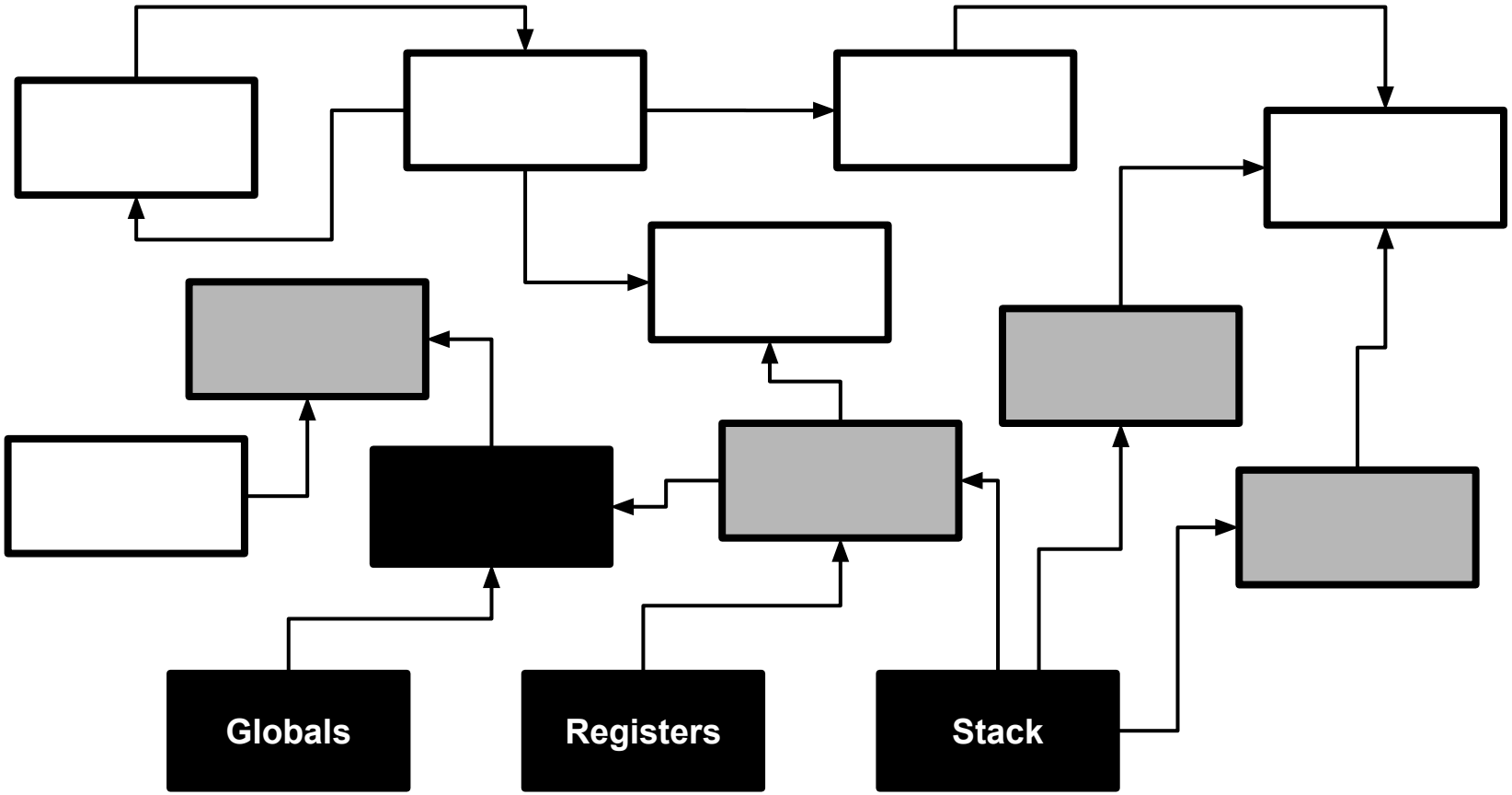


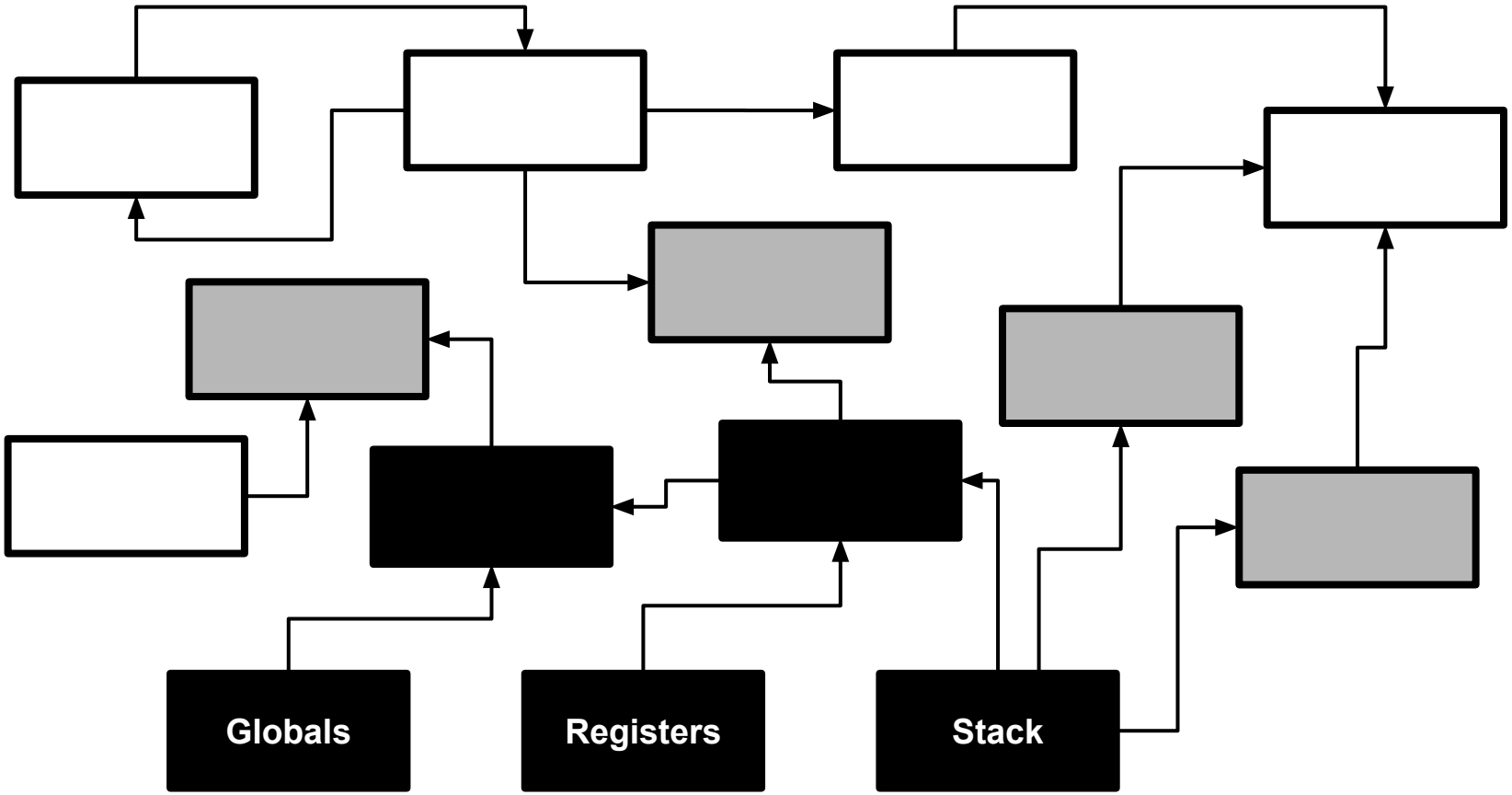

Globals
Registers
Stack

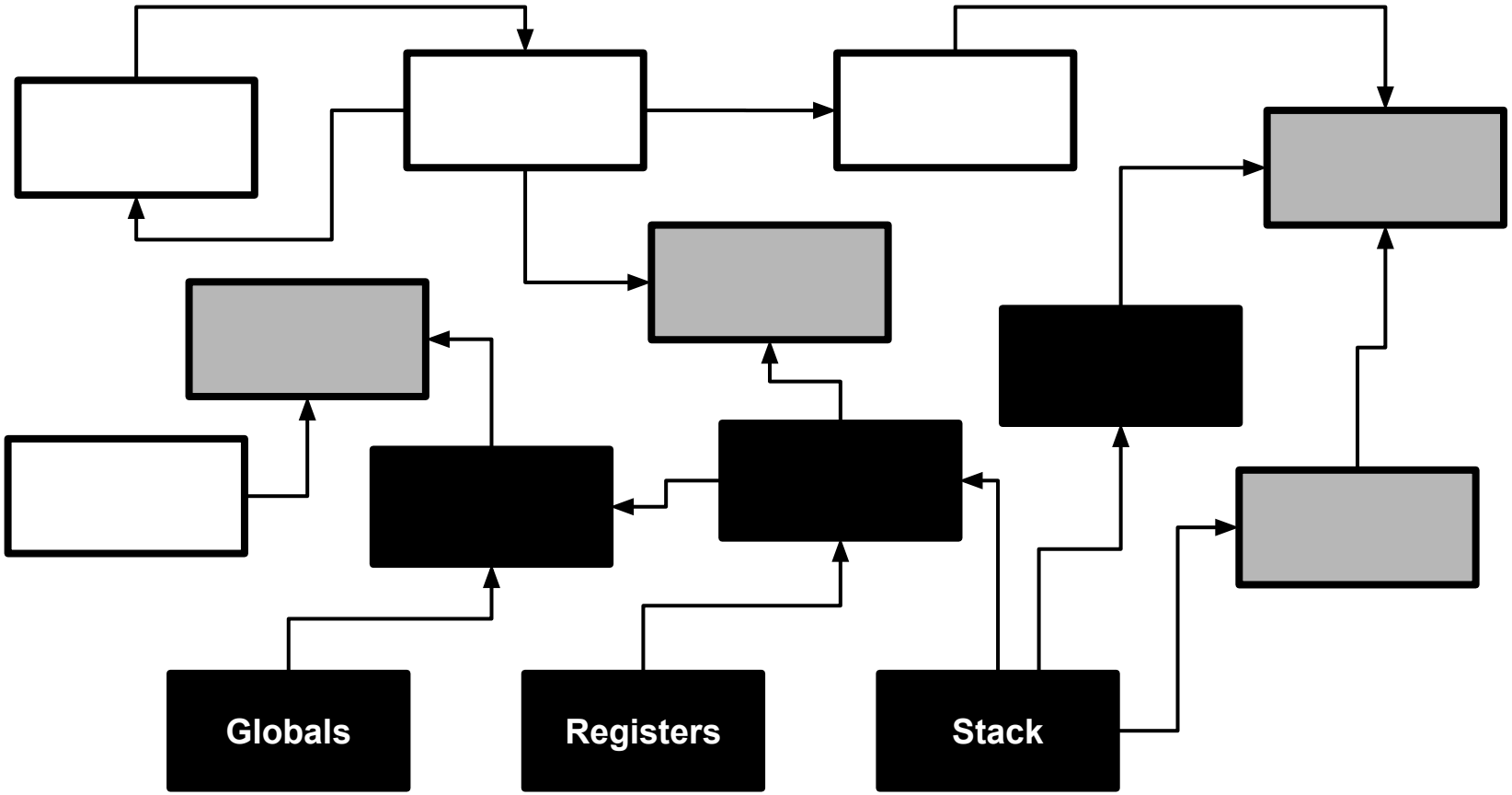

Globals
Registers
Stack

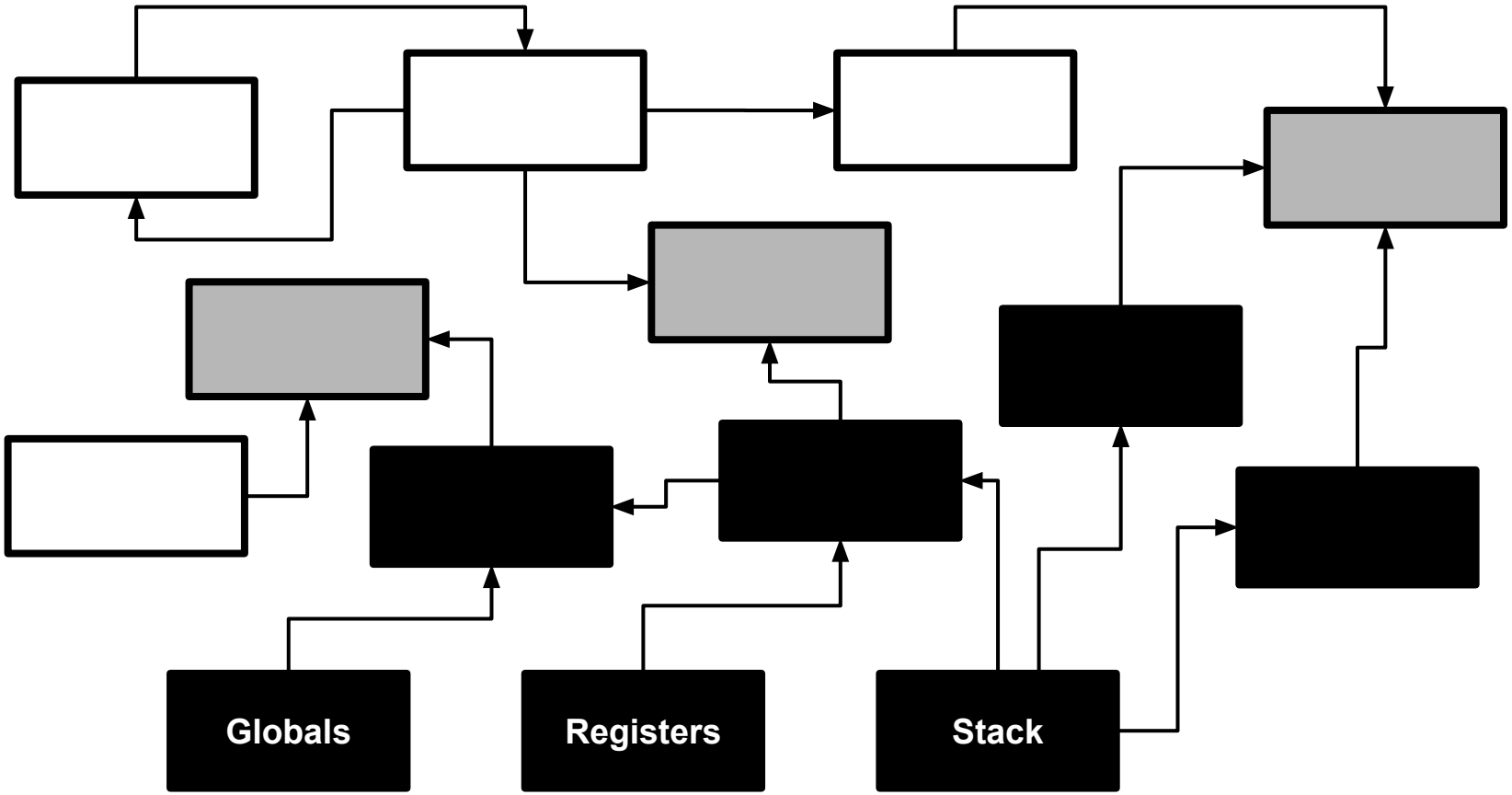

Globals
Registers
Stack

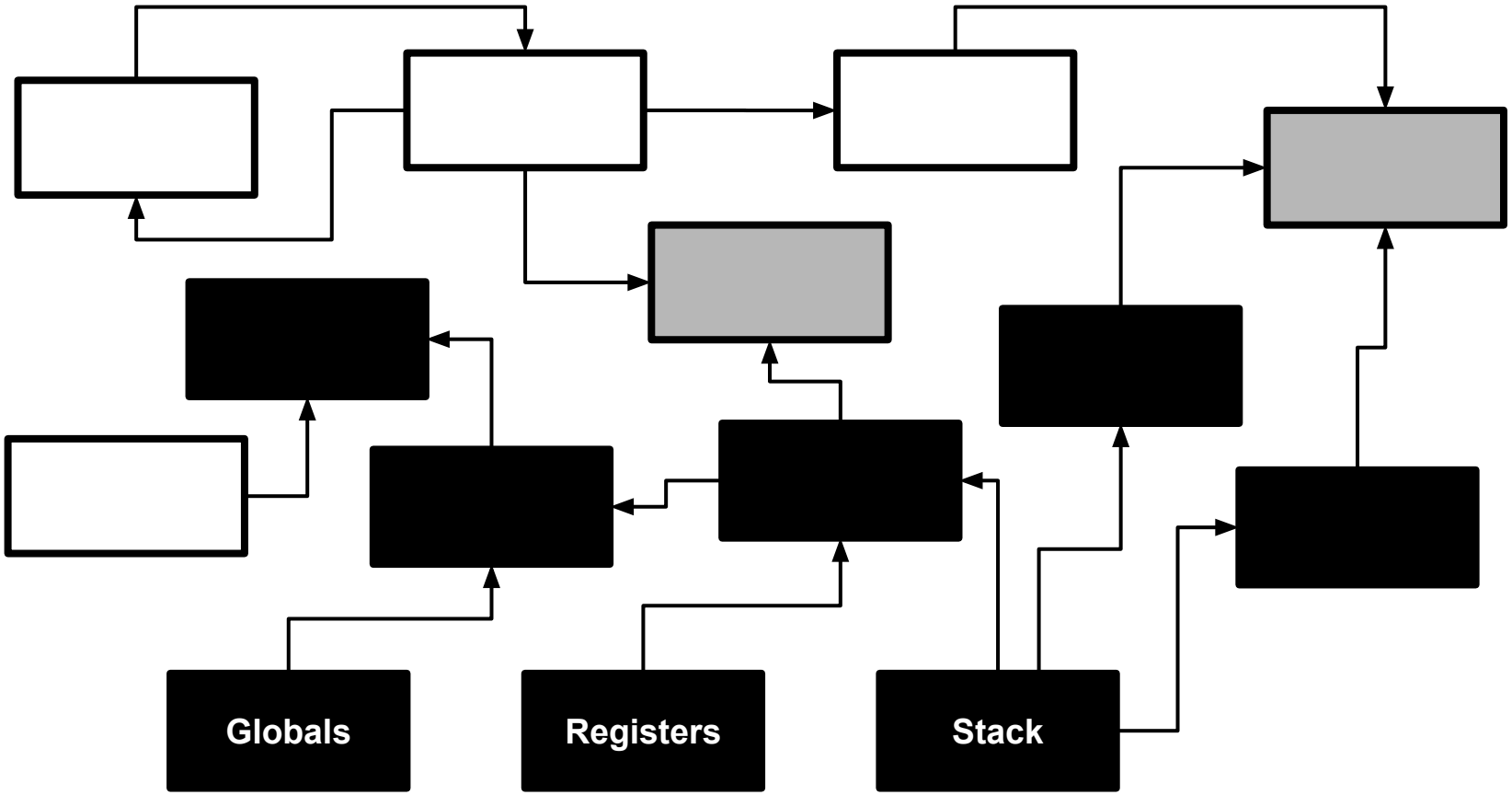

Globals
Registers
Stack

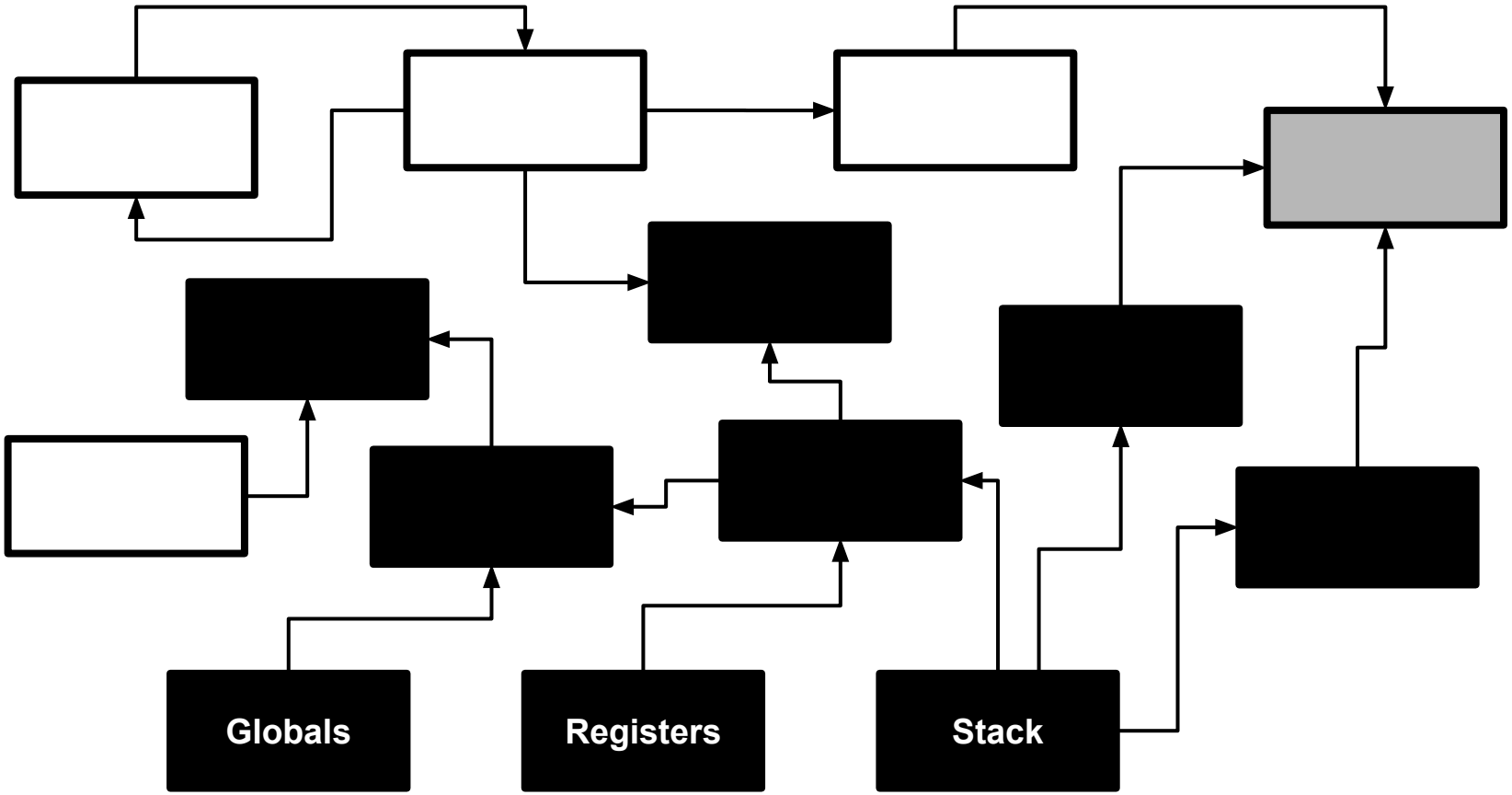

Globals
Registers
Stack

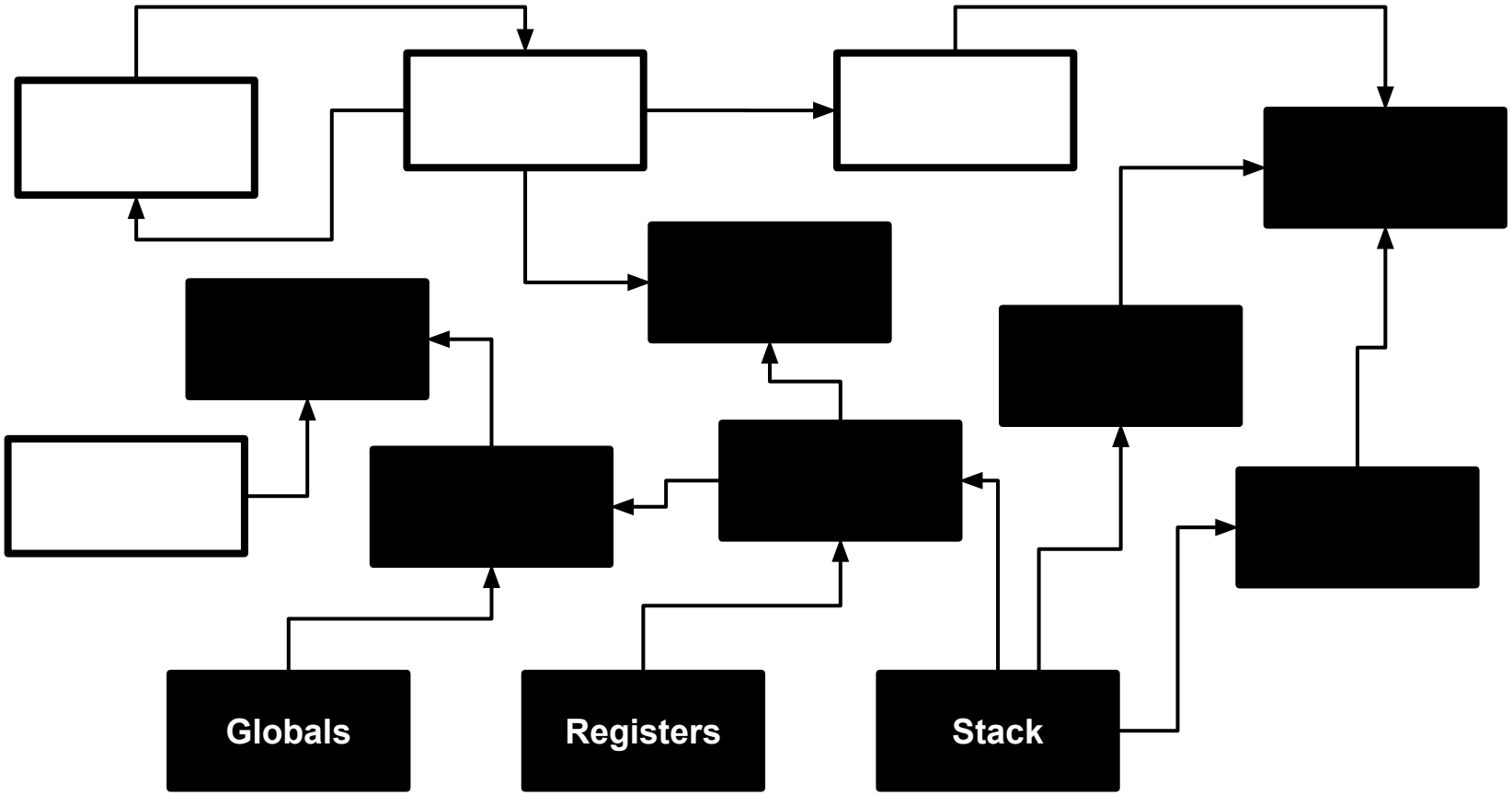

Globals
Registers
Stack

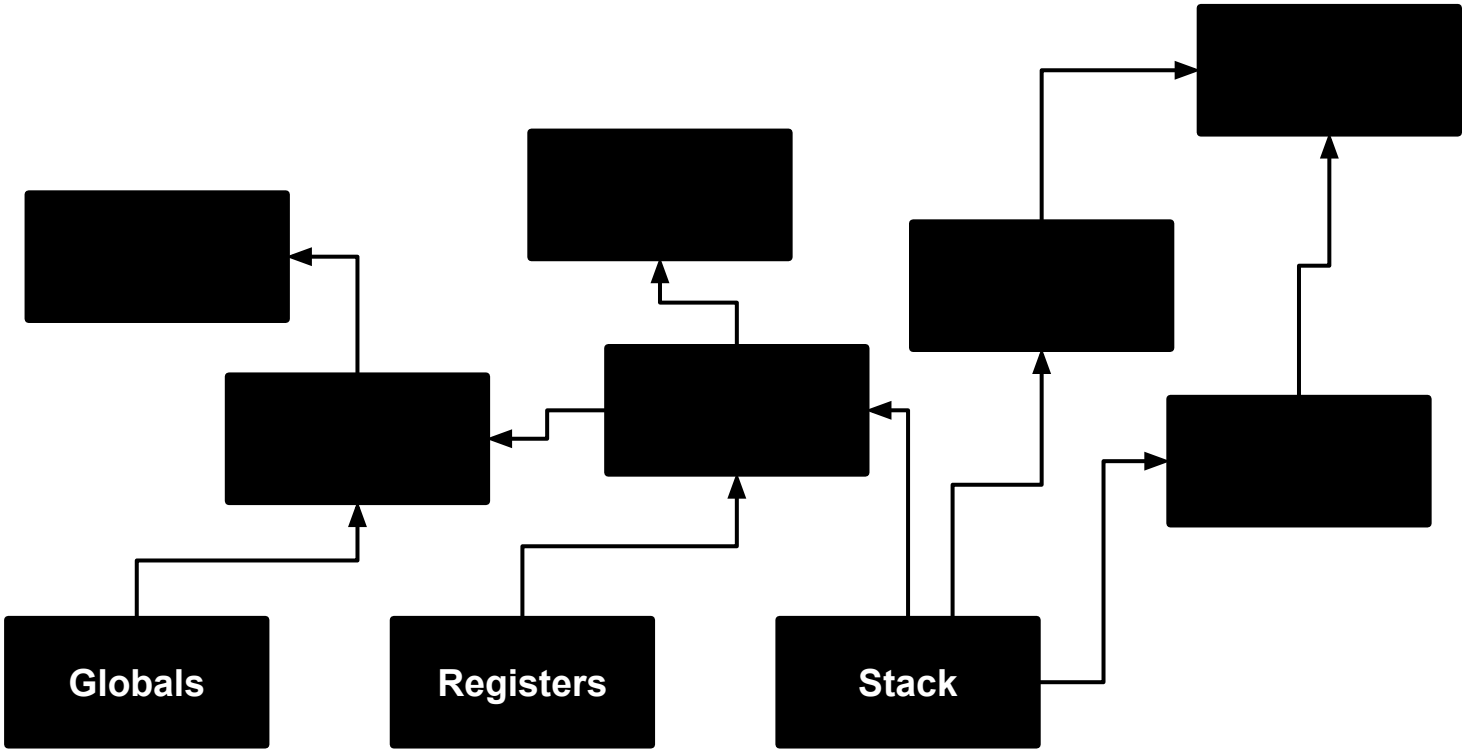

Globals
Registers
Stack

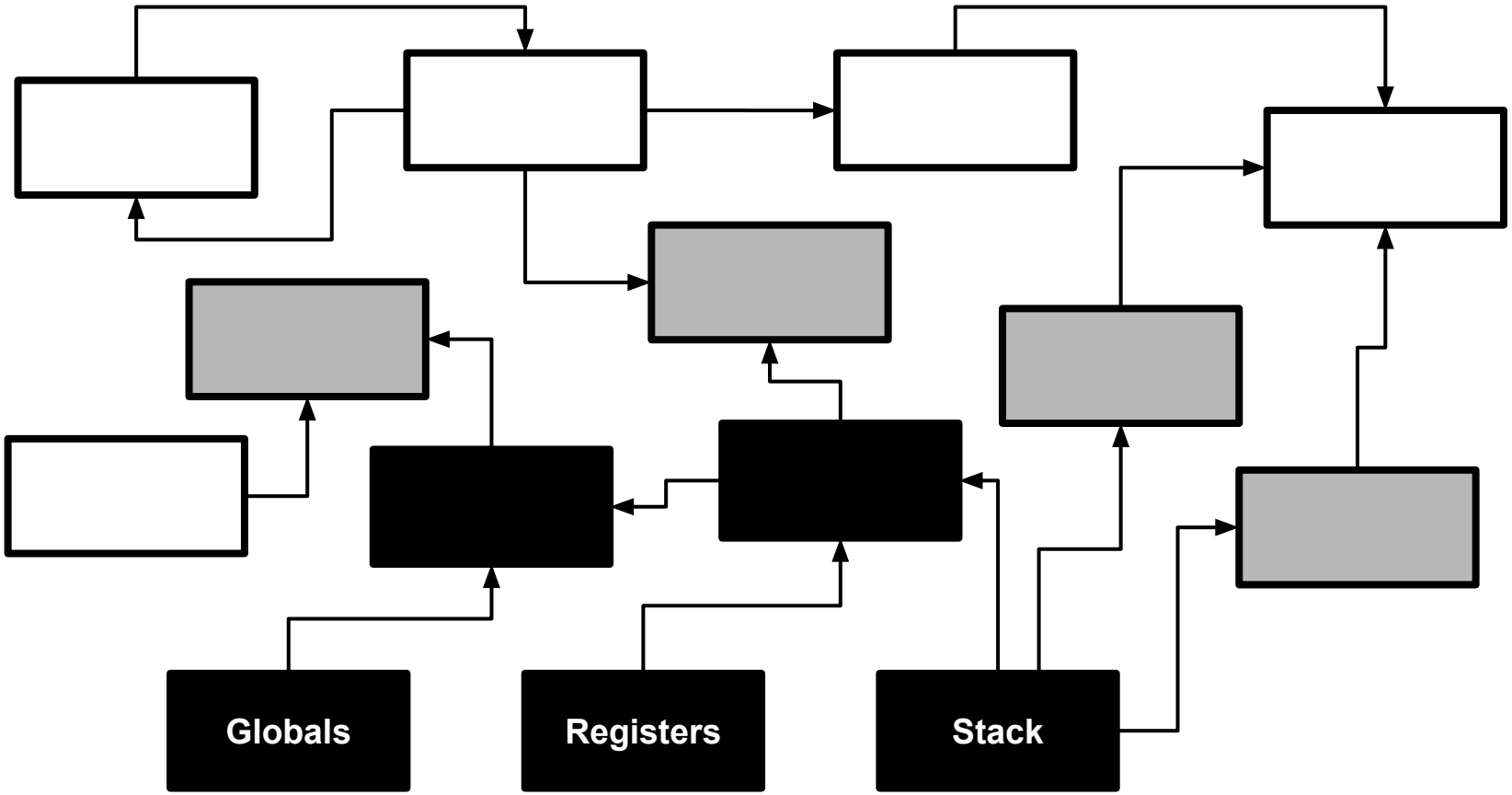

Globals
Registers
Stack

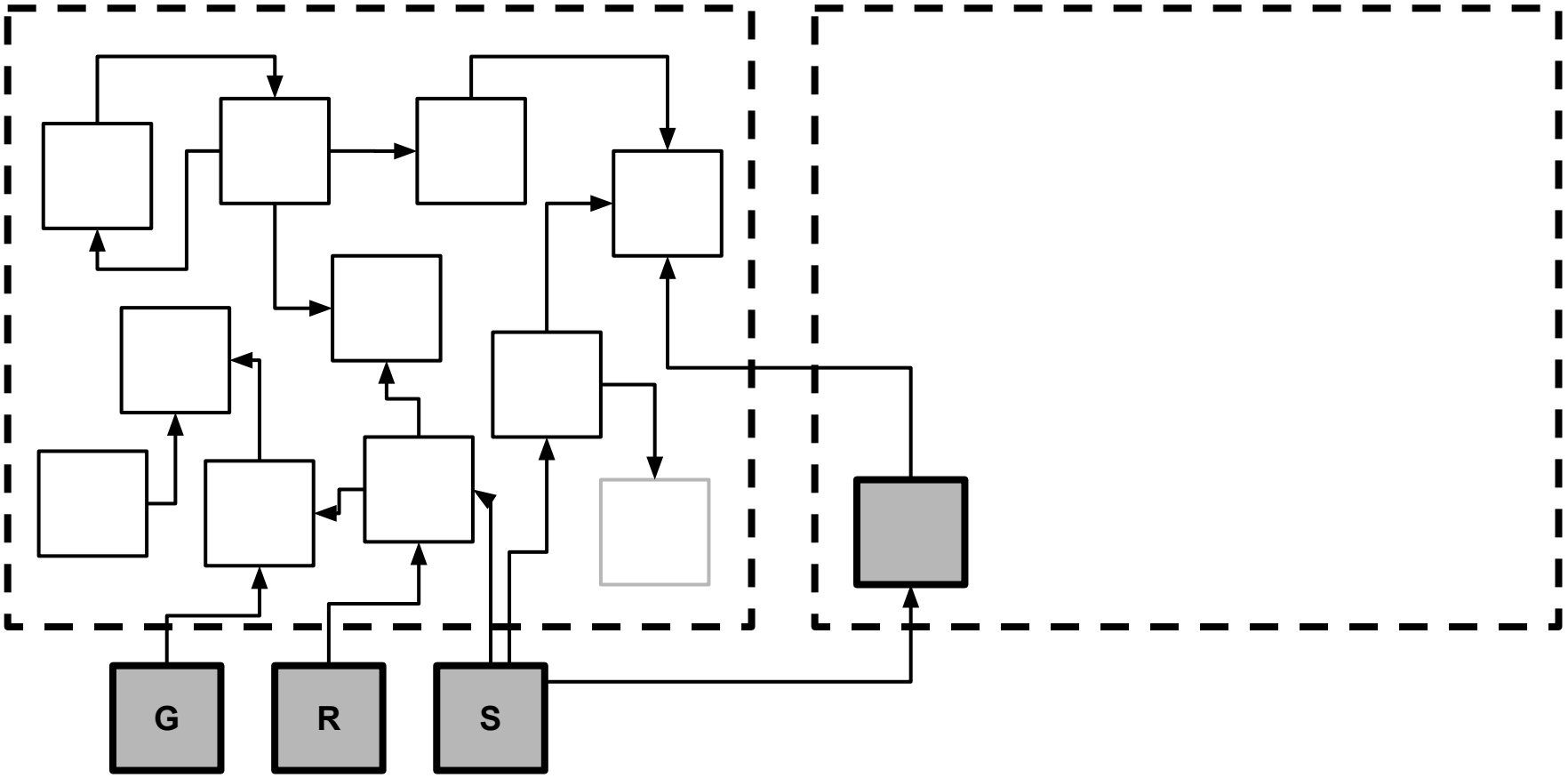

Globals
Registers
Stack

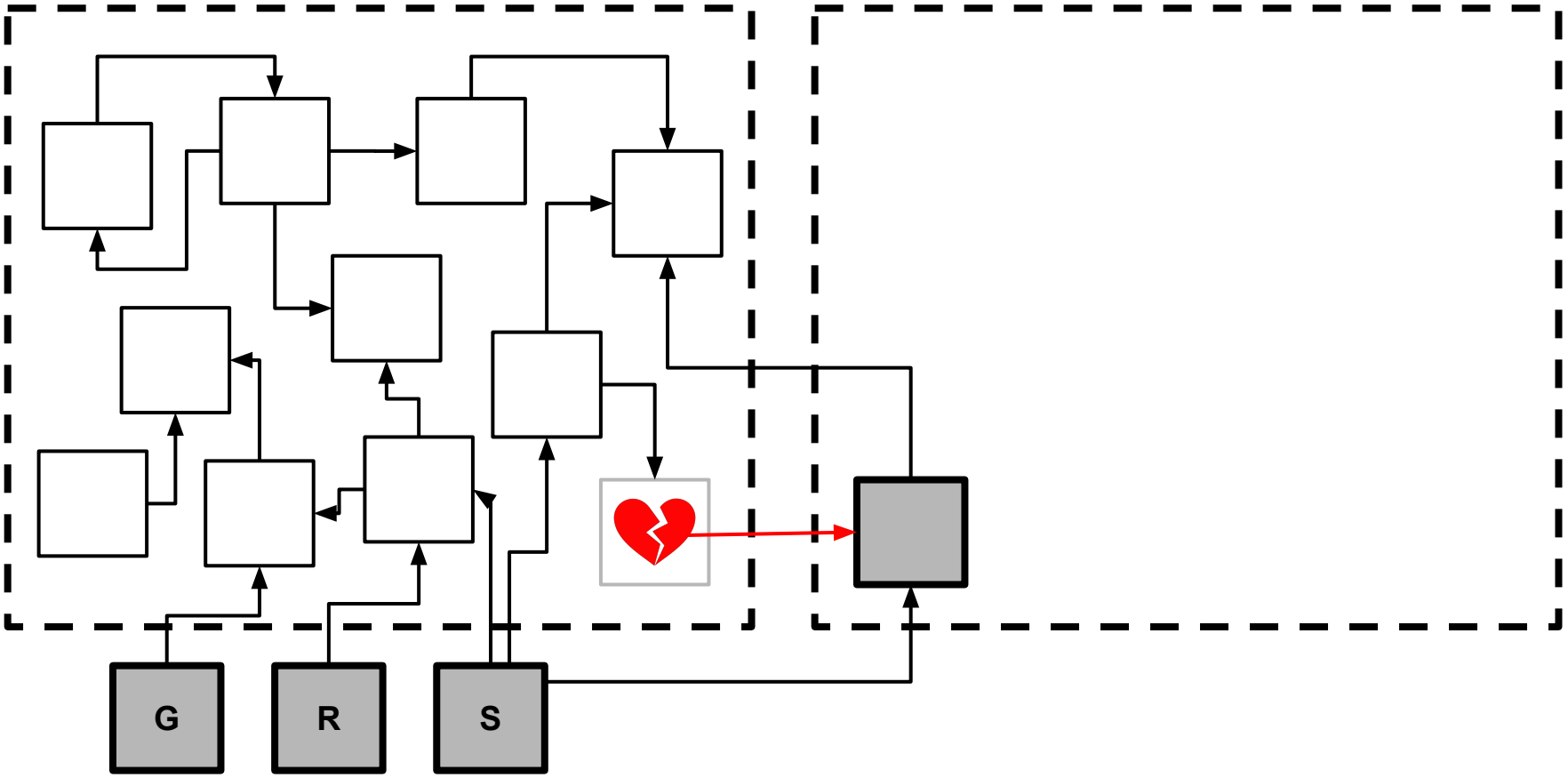

Globals
Registers
Stack


Globals
Registers
Stack

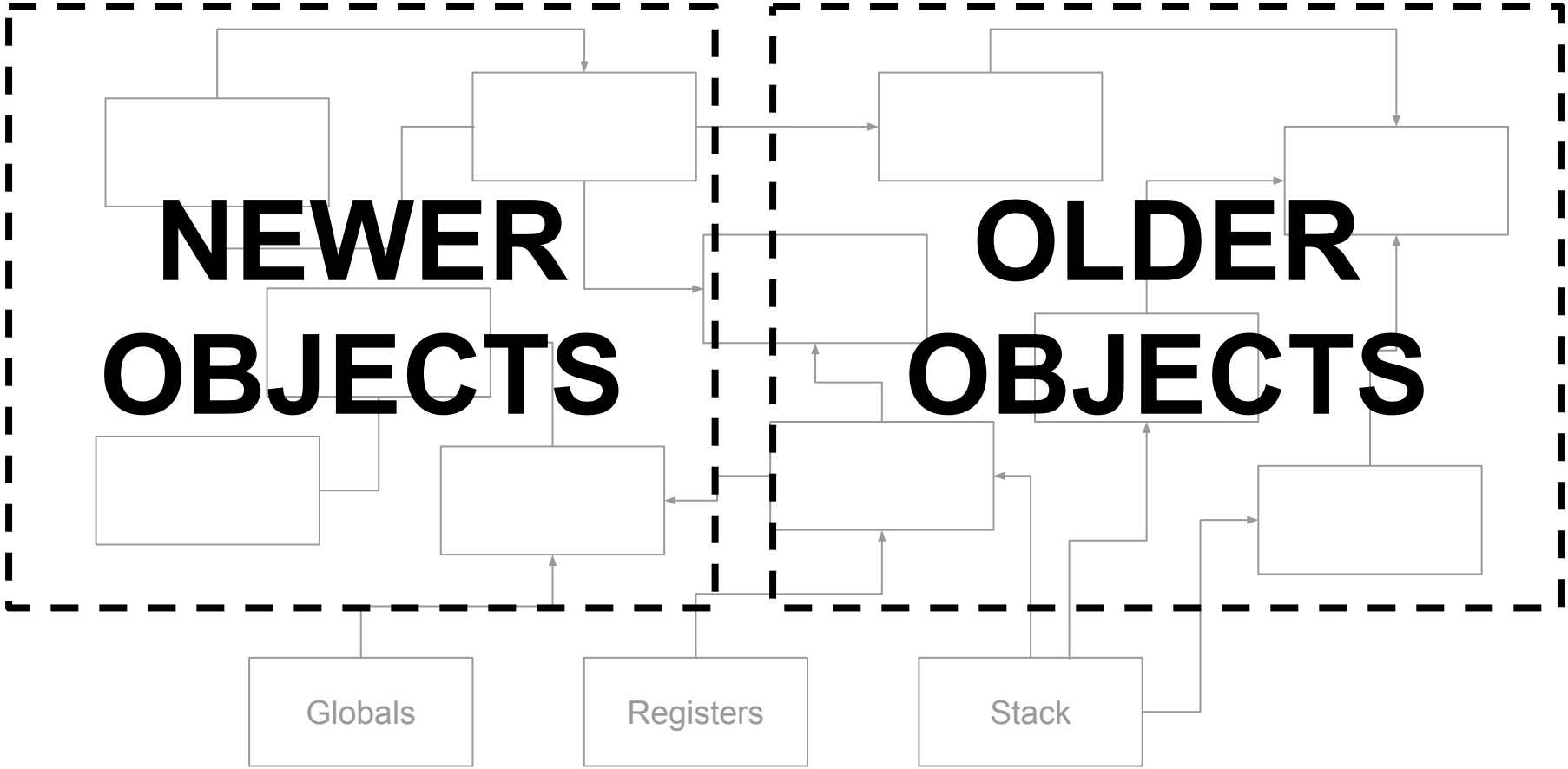

Globals
Registers
Stack

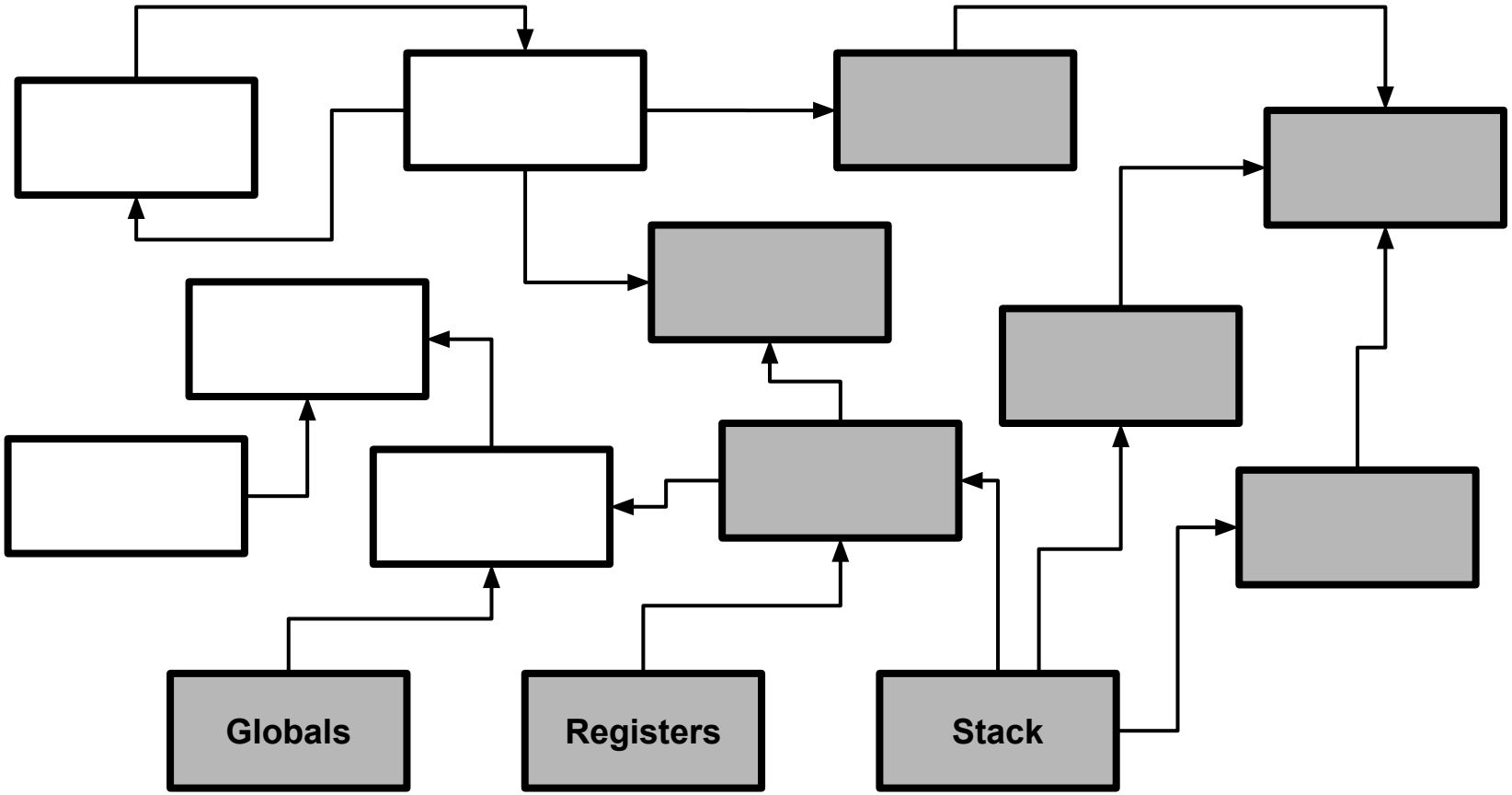

Globals
Registers
Stack


G
R
S

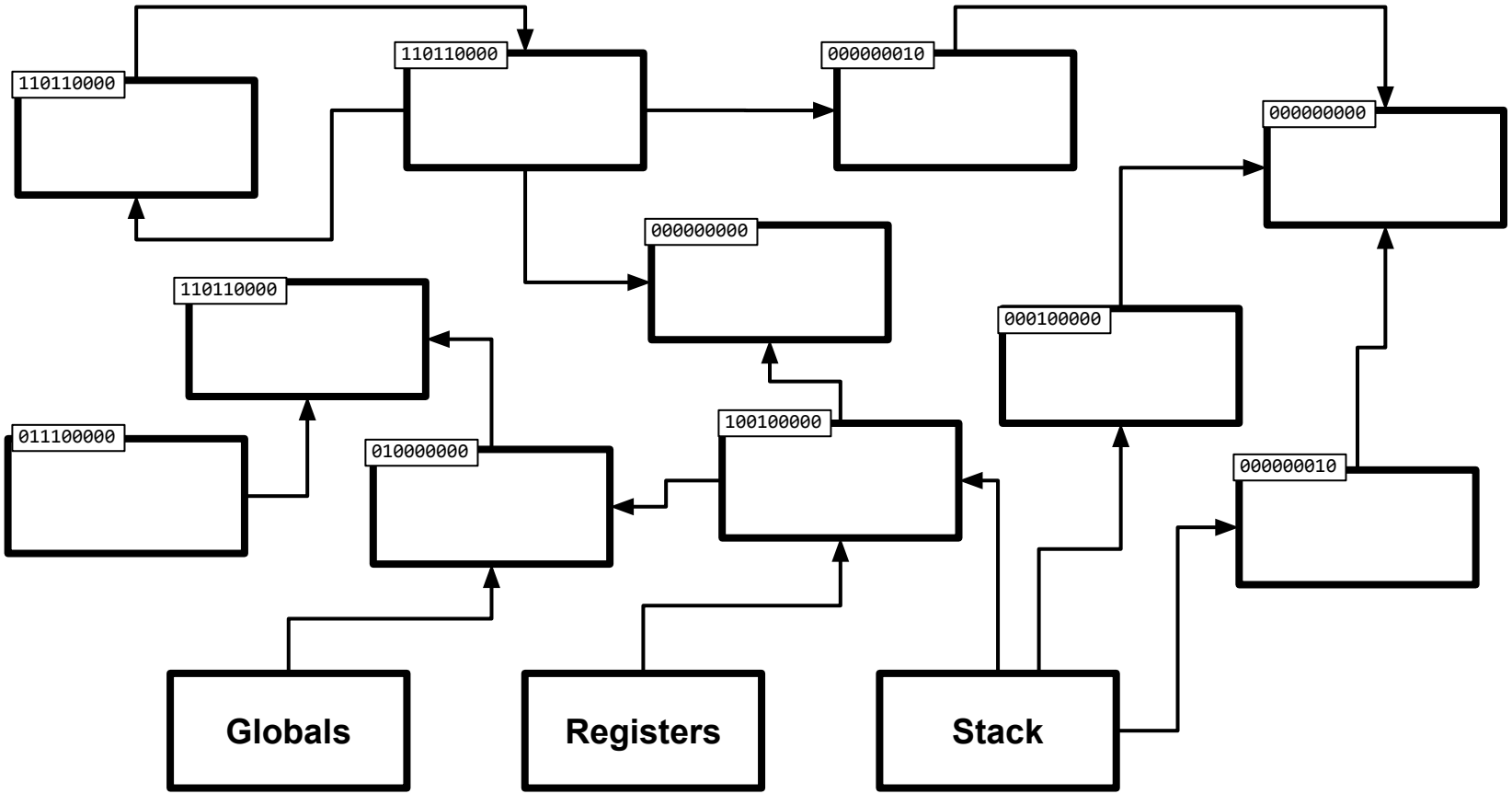

G
R
S

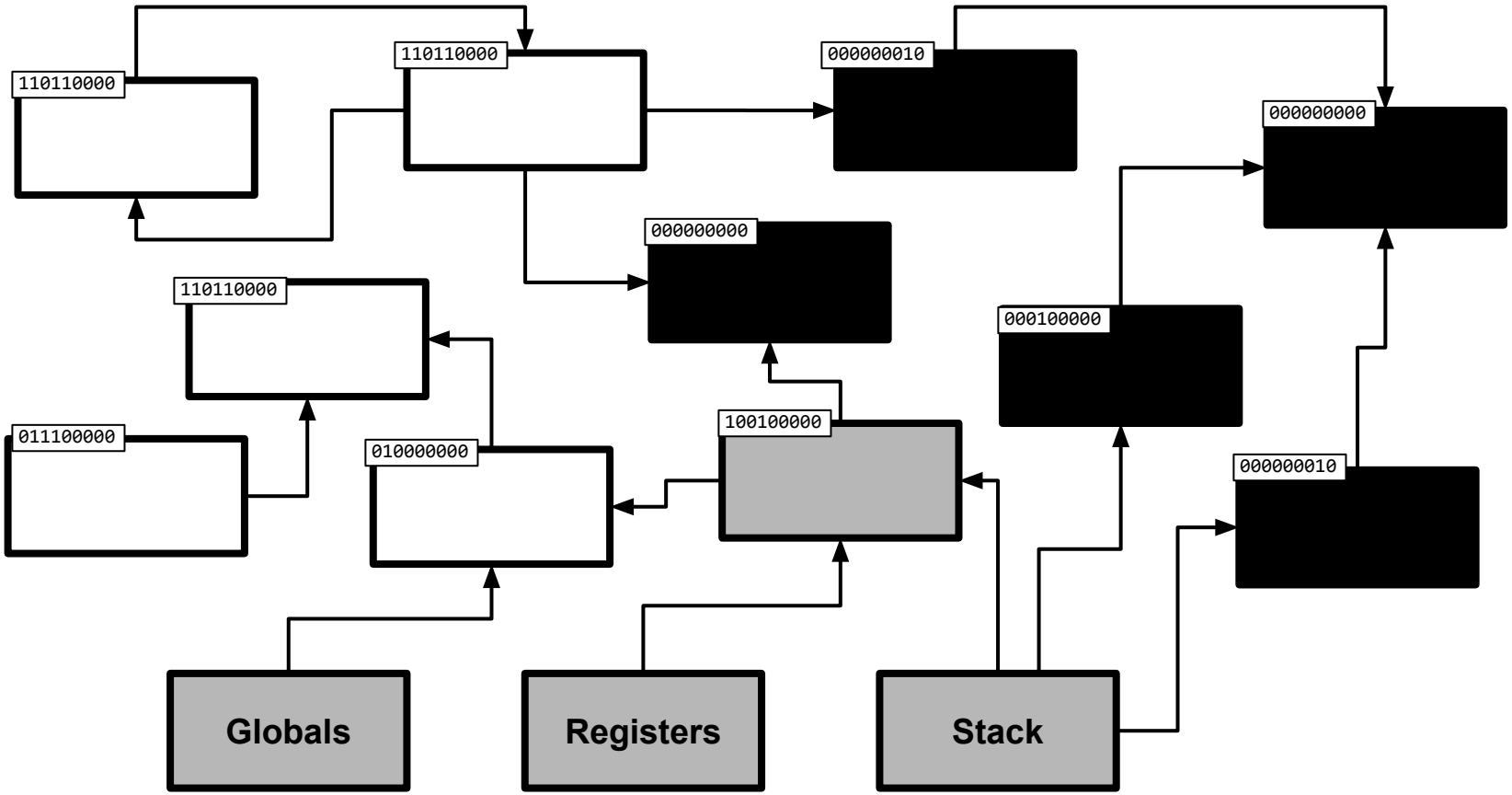
Collecting part of the heap


NEWER OBJECTS
OLDER OBJECTS
Globals
Registers
Stack


Globals
Registers
Stack

Summaries


110110000
110110000
000000010
000000000
000000000
110110000
000100000
011100000
010000000
100100000
000000010
Globals
Registers
Stack


110110000
110110000
000000010
000000000
000000000
110110000
000100000
011100000
010000000
100100000
000000010
Globals
Registers
Stack

Memory Pool System

a **reliable**, soft-real-time, semi-conservative, mostly-copying garbage collector

Abstractions

Make it possible to understand and review the program one function at a time

31. Shield

31.1. Overview

`.overview:` The MPS implements incremental garbage collection using memory barriers implemented by a combination of hardware memory protection and thread control. The MPS needs *separate control* of collector access and mutator (client) access to memory: the collector must be able to incrementally scan objects, without the mutator being able to see them yet.

Unfortunately common operating systems do not support different access levels (protection maps) for different parts of the same process.

The MPS Shield is an abstraction that does extra work to overcome this limitation, and give the rest of the MPS the illusion that we can control collector and mutator access separately.

Design

How the code meets the requirements

31.4. Mechanism

On common operating systems, the only way to allow the MPS access is to allow access from the whole process, including the mutator. So **`ShieldExpose()`** will suspend all mutator threads to prevent any mutator access, and so will `ShieldRaise()` on an unexposed segment. The shield handles suspending and resuming threads, and so the rest of the MPS does not need to worry about it.

The MPS can make multiple sequential, overlapping, or nested calls to **`ShieldExpose()`** on the same segment, as long as each is balanced by a corresponding **`ShieldCover()`** before **`ShieldLeave()`** is called. A usage count is maintained on each segment in `seg->depth`. When the usage count reaches zero, there is no longer any reason the segment should be unprotected, and the shield may reinstate hardware protection at any time.

```
1027        AVER(limit == SegLimit(seg));
1028      } else {
1029        /* Large segment: buffer had only the size requested; job001811. */
1030        AVER(limit <= SegLimit(seg));
1031      }
1032
1033      /* <design/poolamc#.flush.pad> */
1034      if (init < limit) {
1035        ShieldExpose(arena, seg);
1036        (*pool->format->pad)(init, AddrOffset(init, limit));
1037        ShieldCover(arena, seg);
1038      }
1039
1040      /* Any allocation in the buffer (including the padding object just
1041       * created) is white, so needs to be accounted as condemned for all
1042       * traces for which this segment is white. */
1043      TRACE_SET_ITER(ti, trace, seg->white, arena)
```

Tags

Cross-references with no need for special tooling

31.5.3. Invariants

`.inv.outside.running:` The mutator is not suspended while outside the shield.

`.inv.unsynced.suspended:` If any segment is not synced, the mutator is suspended.

`.inv.unsynced.depth:` All unsynced segments have positive depth or are in the queue.

`.inv.outside.depth:` The total depth is zero while outside the shield.

`.inv.prot.shield:` The prot mode is never more than the shield mode.

`.inv.expose.depth:` An exposed segment's depth is greater than zero.

`.inv.expose.prot:` An exposed segment is not protected in the mode it was exposed with.

```
742
743   void (ShieldCover)(Arena arena, Seg seg)
744   {
745     Shield shield;
746
747     /* <design/trace#.fix.noaver> */
748     AVERT_CRITICAL(Arena, arena);
749     shield = ArenaShield(arena);
750     AVERT_CRITICAL(Seg, seg);
751     AVER_CRITICAL(SegPM(seg) == AccessSetEMPTY);
752
753     AVER_CRITICAL(SegDepth(seg) > 0);
754     SegSetDepth(seg, SegDepth(seg) - 1);
755     AVER_CRITICAL(shield->depth > 0);
756     --shield->depth;
757
758     /* Ensure <design/shield#.inv.unsynced.depth>. */
759     shieldQueue(arena, seg);
760   }
761
```

Self-checking (1)

Every abstract type can check itself

```
65  Bool ShieldCheck(Shield shield)
66  {
67    CHECKS(Shield, shield);
68    /* Can't check Boolean bitfields <design/type#.bool.bitfield.check> */
69    CHECKL(shield->queue == NULL || shield->length > 0);
70    CHECKL(shield->limit <= shield->length);
71    CHECKL(shield->next <= shield->limit);
72
73    /* The mutator is not suspended while outside the shield
74       <design/shield#.inv.outside.running>. */
75    CHECKL(shield->inside || !shield->suspended);
76
77    /* If any segment is not synced, the mutator is suspended
78       <design/shield#.inv.unsynced.suspended>. */
79    CHECKL(shield->unsynced == 0 || shield->suspended);
80
81    /* The total depth is zero while outside the shield
82       <design/shield#.inv.outside.depth>. */
83    CHECKL(shield->inside || shield->depth == 0);
```

Self-checking (2)

Every function checks its arguments

```
1016
1017      AVERT(Seg, seg);
1018      AVERT(Buffer, buffer);
1019      base = BufferBase(buffer);
1020      init = BufferGetInit(buffer);
1021      limit = BufferLimit(buffer);
1022      AVER(SegBase(seg) <= base);
1023      AVER(base <= init);
1024      AVER(init <= limit);
1025      if(SegSize(seg) < amc->largeSize) {
1026        /* Small or Medium segment: buffer had the entire seg. */
1027        AVER(limit == SegLimit(seg));
1028      } else {
1029        /* Large segment: buffer had only the size requested; job001811. */
1030        AVER(limit <= SegLimit(seg));
1031      }
1032
```

Self-checking (3)

Multics-style structure signatures

```
119
120     vm->pageSize = pageSize;
121     vm->block = vbase;
122     vm->base = AddrAlignUp(vbase, grainSize);
123     vm->limit = AddrAdd(vm->base, size);
124     AVER(vm->base < vm->limit);  /* .assume.not-last */
125     AVER(vm->limit <= AddrAdd((Addr)vm->block, reserved));
126     vm->reserved = reserved;
127     vm->mapped = 0;
128
129     vm->sig = VMSig;
130     AVERT(VM, vm);
131
```

```
139  void VMFinish(VM vm)
140  {
141    int r;
142
143    AVERT(VM, vm);
144    /* Descriptor must not be stored inside its own VM at this point. */
145    AVER(PointerAdd(vm, sizeof *vm) <= vm->block
146         || PointerAdd(vm->block, VMReserved(vm)) <= (Pointer)vm);
147    /* All address space must have been unmapped. */
148    AVER(VMMapped(vm) == (Size)0);
149
150    EVENT1(VMFinish, vm);
151
152    vm->sig = SigInvalid;
153
154    r = munmap(vm->block, vm->reserved);
155    AVER(r == 0);
156  }
```

Error handling (1)

Always propagate errors upward

```
282     res = VMInit(vm, size, ArenaGrainSize(arena), vmArena->vmParams);
283     if (res != ResOK)
284       goto failVMInit;
285
286     base = VMBase(vm);
287     limit = VMLimit(vm);
288
289     res = BootBlockInit(boot, (void *)base, (void *)limit);
290     if (res != ResOK)
291       goto failBootInit;
292
293     /* .overhead.chunk-struct: Allocate and map the chunk structure. */
294     res = BootAlloc(&p, boot, sizeof(VMChunkStruct), MPS_PF_ALIGN);
295     if (res != ResOK)
296       goto failChunkAlloc;
297     vmChunk = p;
298     /* Calculate the limit of the grain where the chunkStruct resides. */
299     chunkStructLimit = AddrAlignUp((Addr)(vmChunk + 1), ArenaGrainSize(arena));
300     res = vmArenaMap(vmArena, vm, base, chunkStructLimit);
301     if (res != ResOK)
302       goto failChunkMap;
303     vmChunk->overheadMappedLimit = chunkStructLimit;
```

Error handling (2)

Never fail due to lack of memory

```
1218
1219  static Res traceScanSeg(TraceSet ts, Rank rank, Arena arena, Seg seg)
1220  {
1221    Res res;
1222
1223    res = traceScanSegRes(ts, rank, arena, seg);
1224    if(ResIsAllocFailure(res)) {
1225      ArenaSetEmergency(arena, TRUE);
1226      res = traceScanSegRes(ts, rank, arena, seg);
1227      /* Should be OK in emergency mode. */
1228      AVER(!ResIsAllocFailure(res));
1229    }
1230
1231    return res;
1232  }
1233
```

```
130
131     res = LandDelete(&oldRange, fo->primary, range);
132
133     if (res == ResFAIL) {
134       /* Range not found in primary: try secondary. */
135       return LandDelete(rangeReturn, fo->secondary, range);
136     } else if (res != ResOK) {
137       /* Range was found in primary, but couldn't be deleted. The only
138        * case we expect to encounter here is the case where the primary
139        * is out of memory. (In particular, we don't handle the case of a
140        * CBS returning ResLIMIT because its block pool has been
141        * configured not to automatically extend itself.)
142        */
143       AVER(ResIsAllocFailure(res));
144
145       /* Delete the whole of oldRange, and re-insert the fragments
146        * (which might end up in the secondary). See
147        * <design/failover#.impl.assume.delete>.
148        */
149       res = LandDelete(&dummyRange, fo->primary, &oldRange);
150       if (res != ResOK)
151         return res;
152
```

Memory Pool System

https://ravenbrook.com/project/mps/