Functional Programming

Christopher D. Clack

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 26

REFERENCE COUNTING GARBAGE COLLECTION

© 2020 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

CONTENTS

- Overview
- Assumptions
- Simple imperative pseudo-code
- Optimisations
- For and against Reference Counting GC

In this lecture we will explore the third and final of three canonical garbage collection algorithms – Reference Counting Garbage Collection

The lecture provides simple imperative pseudo-code for the basic reference counting garbage collection functions and for a number of optimisations to basic reference counting:

- non-recursive freeing
- deferred reference counting
- limited-width counts
- cyclic reference counting

The lecture ends with a summary of good and bad characteristics of Reference Counting Garbage Collection

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OVERVIEW

- Reference Counting GC triggered when a block becomes inaccessible
- Program evaluation typically pauses during GC of that block
- Garbage is detected by tracking the number of live pointers to every block
- Garbage is collected / made available for re-use by immediately adding the garbage block to the free list
- Typically malloc() uses a free list and Reference Counting GC adds garbage blocks to the free list
- Live blocks are not copied: fragmentation may result

Triggering the garbage collection

Reference Counting garbage collection is triggered whenever a block becomes inaccessible to the program (no live pointers to the block)

Evaluation of the program typically pauses momentarily until garbage collection of that block is finished

Identifying garbage

The Reference Counting garbage collection algorithm automatically and incrementally identifies garbage by tracking the number of live pointers to every block – if the number becomes zero, the block is garbage

Collecting garbage

The Reference Counting algorithm makes garbage available for re-use immediately – it detects the deletion of the last live pointer to a block, identifies it as garbage, and immediately adds it to the free list (see later for complexities)

Interaction with memory allocation

Typically *malloc()* uses a free list and Reference Counting GC adds garbage blocks to the free list

Fragmentation

Live blocks are not moved: fragmentation may result, though a separate compactor could be used

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

ASSUMPTIONS

- *malloc()* allocates free memory using a free list
- block headers include a reference count of live pointers to that block
- a block is garbage if its reference count is zero
- a new pointer to a block can only be created by copying an existing pointer using the copy() function
- an existing pointer can only be deleted by using the delete() function
- can distinguish pointers from other data
- evaluation pauses during GC of a single block, and then resumes

Assumptions

Reference Counting garbage collection assumes:

- normally, the memory allocator uses free-list allocation (though other methods may be possible)
- block headers are extended to include a count of all the references to that block
  - **aside:** if the reference-count field in the header is at least the same size as a pointer then the former can be used to hold the "next" pointer in the free list (free blocks don't need reference counts), so there is no minimum block size
- a block is garbage if the reference count drops to zero
- a new pointer to a block can only be created by using the *copy()* function to copy an existing pointer, and this increments the block's reference count
- an existing pointer can only be deleted by using the *delete()* function, and this decrements the block's reference count
- it is always possible to distinguish between values that are pointers and those that are other data
- that program evaluation pauses during the GC of a single block, and resumes after the GC has finished (see later for complexities)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

SIMPLE IMPERATIVE PSEUDO-CODE: ALLOCATION

```
malloc(n) =
   if (FLP == null) then abort "Out of memory"
   (next, size) = read(FLP – HEADERSIZE)
   prev = 0, temp = FLP
   while (size<n) {
      if (prev == 0) prev = FLP else prev = temp
      temp = next
      (next, size) = read(temp-HEADERSIZE)
   }
   if (prev == 0) then FLP = next
                  else   write(prev-HEADERSIZE, next)
   write(temp-HEADERSIZE, 1)
   return(temp )
```

Allocation

The code on this slide implements a very simple first-fit free list with no splitting. FLP is the free list pointer

- If the free list is ever empty, the program is out of memory
- *malloc()* reads the header (containing "next" and "size") of the first free block on the free list and initialises local variables *prev* and *temp*
- *malloc()* then checks whether *size < n* and enters a loop (traversing the free list) until the first block is found where *size >=n*
- the found block is removed from the free list, and the free list pointers are set accordingly
- *malloc()* sets the reference count of the found block to be "1" and returns the address of the block

Notice how *malloc()* uses the first location in the header as the "next" pointer while traversing the free list, and then treats it as the reference count (setting it to the value 1) just before returning its address as a live cell

malloc() could also clear (set to zero) the entire data area of the block being allocated (this choice, coordinated with the compiler, must be made regardless of the allocator being used)

You should satisfy yourself that you understand how this code works (and see if you can simulate it in Miranda)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

SIMPLE IMPERATIVE PSEUDO-CODE: DELETE

```
delete(p) =
   refcount = read(p - HEADERSIZE)
   refcount = refcount - 1
   write(p – HEADERSIZE, refcount)
   if (refcount == 0) {
      for each M in children(p) { delete(M) }
      write(p – HEADERSIZE, FLP)
      FLP = p
   }
   return()
```

Pointer deletion

The code on this slide implements the *delete()* function that deletes a pointer to a block

The *delete()* function is passed an argument "p" that is a pointer to (the address of) a block in the heap

delete() must read the current reference count field from the block's header, subtract 1 and write the result back to the reference count field in the block's header

If the new reference count is zero, the delete() function must:

1. delete all pointers found in the data area of the block, and
2. add the block to the free list using a simple LIFO policy

Notice that when the block is added to the free list its reference count field in its header is now treated as the pointer to the next free block

You should satisfy yourself that you understand how this code works (and see if you can simulate it in Miranda)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

SIMPLE IMPERATIVE PSEUDO-CODE: COPY

```
copy(r, p) =
    refcount = read(p - HEADERSIZE)
    refcount = refcount + 1
    write(p – HEADERSIZE, refcount)
    temp = read(r)
    delete(temp)
    write(r,p)
    return()
```

Pointer copy

The code on this slide implements the *copy()* function that makes a copy of a pointer and stores it in a given location. The Jones & Lins book calls this the *update()* function

The *copy()* function is passed two arguments: the first ("r") is the address in memory where the new copy should be stored, overwriting an existing pointer that is assumed to exist in that location; and the second ("p") is the pointer that needs to be copied

copy() does the following:

1. it increments the reference count for the block pointed to by "p"
2. it deletes the current pointer existing in location "r"
3. it writes the copy of "p" into the location "r"

It is important that *copy()* increments the reference count for block "p" first. Thus, if (i) the existing location "r" already contained a pointer to block "x" and (ii) *copy(r,x)* were called to update the contents of "r" with a pointer to the same block "x", then *copy()* first increments the reference count for "x" and then decrements it – doing the increment last could cause "x" to be garbage collected prematurely!

You should satisfy yourself that you understand how this code works (and see if you can simulate it in Miranda)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 1: NON-RECURSIVE FREEING

```
delete1(p) =
   refcount = read(p - HEADERSIZE)
   if (refcount == 1) {
      write(p – HEADERSIZE, FLP)
      FLP = p
   } else {
      refcount = refcount - 1
      write(p – HEADERSIZE, refcount)
   }
   return()
```

Non-recursive freeing

The simple recursive implementation of *delete()* on Slide 7 has a problem: the deletion of a single pointer requires deletion of all of that block's children (all pointers in its data area) – and potentially all its grand-children, and so on

This could create an "embarrassing pause" in evaluation, which can be solved by using non-recursive freeing as shown on this and the next slide (loosely based on Weizenbaum's 1963 algorithm)

Notice how the code for *delete1()* does NOT attempt to call itself recursively to delete the children pointers in the data area of the block – the deletion is now done by a new version of *malloc()* called *malloc1()* just before a block is allocated, and is initially only done for the children – not the grandchildren or further descendants (which are processed later)

The new code for *malloc1()* is given on the next slide

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 1: NON-RECURSIVE FREEING

```
malloc1(n) =
   if (FLP == null) then abort "Out of memory"
   (next, size) = read(FLP – HEADERSIZE)
   prev = 0, temp = FLP
   while (size<n) {
      if (prev == 0) prev = FLP else prev = temp
      temp = next
      (next, size) = read(temp-HEADERSIZE)
   }
   if (prev == 0) then FLP = next
                  else   write(prev-HEADERSIZE, next)
   write(temp-HEADERSIZE, 1)
   for M in children(temp) {  delete1(M)  }
   return(temp )
```

Non-recursive freeing (continued)

Here is the new code for *malloc1()*. It contains only one new line of code (in red) – a for loop that calls *delete1()* on every child pointer found in the data area of the allocated block. This is done just before the pointer to the newly allocated block is returned to the code that called *malloc()*

Remember that delete1() decrements the reference count of the block and then, if it is zero, adds it to the free list

malloc1() and *delete1()* work together so that the descendants of a block have their pointers deleted incrementally. The maximum amount of work done during any deletion is constant, and the maximum amount of work done during any memory allocation is bounded by the maximum number of pointers that can exist in the data area of a block

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 2: DEFERRED REFERENCE COUNTING

```
delete2(p) =
    refcount = read(p - HEADERSIZE)
    refcount = refcount - 1
    write(p – HEADERSIZE, refcount)
    if (refcount == 0) {  add_to_ZCT(p)  }
    return()

copy2(r, p) =
    refcount = read(p - HEADERSIZE)
    refcount = refcount + 1
    write(p – HEADERSIZE, refcount)
    temp = read(r)
    delete(temp)
    remove_from_ZCT(p)
    write(r,p)
    return()
```

Deferred reference counting

The overhead of incrementing and decrementing reference counts in block headers can be very high – the manipulations themselves take time, and they must be done very often – e.g. when pushing a pointer onto, or popping it off the system stack; or when using pointers to traverse a linked data structure. Each data fetch probably incurs a pre-fetch of nearby data into the data cache, and worst of all each reference-update operation could cause a VM page swap

Deutsch and Bobrow (1976) observed that most pointer stores are made into local variables or the stack, and not into heap locations. So they treated heap references differently – reference counts only counted pointers from heap blocks to other heap blocks, and pointers held elsewhere are tracked only periodically

Thus a block can no longer be reclaimed when its reference count drops to zero – instead they are added to a Zero Count Table ("ZCT" – a hash table or bitmap)

On this slide we show the new code for *delete2()* and *copy2()*, and the next slide will show reconciliation of the ZCT and adding garbage to the free list

delete2() detects when no other blocks point to this block and adds it to the ZCT

copy2() detects a new pointer created from heap location r to the heap block p, so removes p from the ZCT (if there)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 2: DEFERRED REFERENCE COUNTING

```
reconcile2() =
   for each N in Roots {
      reference_count = read(N - HEADERSIZE)
      write(N - HEADERSIZE, reference_count + 1)
   }
   for each M in ZCT {
      reference_count = read(M - HEADERSIZE)
      if (reference_count == 0) {
         for each J in children(M) {  delete2(J)  }
         write(M – HEADERSIZE, FLP)
         FLP = M
      }
   }
   for each N in Roots {
      reference_count = read(N - HEADERSIZE)
      write(N - HEADERSIZE, reference_count - 1)
   }
```

Deferred reference counting (continued)

Periodically (e.g. when it is about to become too big) the ZCT is scanned to see whether blocks with zero reference counts are really garbage or not. To do this it checks the ZCT against all pointers held on the stack and in local variables (together, call these "Roots"). Note that the pointers found in Roots may point to blocks whose reference counts are NOT zero and that are NOT on the ZCT

The function *reconcile2()* shown on this slide does this, as follows:

1. increment the reference count for all blocks pointed to from Roots (includes those in ZCT and those not in ZCT)
2. any block in ZCT with zero reference count must be garbage, so delete its children and put it on the free list
3. decrement the reference count for all blocks pointed to from Roots (includes those in ZCT and those not in ZCT)

If deleting a child would itself potentially cause the ZCT to overflow, the reclamation of that object could be deferred until the next reclamation phase (alternatively optimisations 1 and 2 could be combined, so children are only deleted incrementally, on allocation)

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 3: LIMITED WIDTH COUNTS

- Each block header uses memory to hold the count – this is a large memory overhead (say 4 bytes per block)
- However, in practice programs tend not to produce lots of pointers to a single block, so can use fewer bits ("limited width") to hold the count (& manage overflow)
- Limited-width maximum count is "sticky"
  - Mark-scan run periodically to manage overflow
- Extreme: Single-Bit ("1-bit") reference counting
  - Again, mark-scan used to manage the overflow of blocks that once were shared and have since become garbage
- 1-bit RC can be viewed as an incremental method to reduce the frequency of mark-scan collection

Limited width counts

Reference counting requires each block header to set aside memory to hold the count. If every memory location in the heap pointed to the same block then this would require a large number of bits, and if we take into account byte-alignment issues (for speed) then we probably need 4 bytes (the same as a 32-bit pointer) additional memory overhead for every block

In practice programs don't produce lots of pointers to a single block, so Optimisation 3 uses limited-width reference count fields at the cost of needing to handle overflow occasionally

The (limited-width) maximum reference count must be "sticky" – once reached it can't be incremented or decremented and a tracing collector must be run periodically to manage this overflow

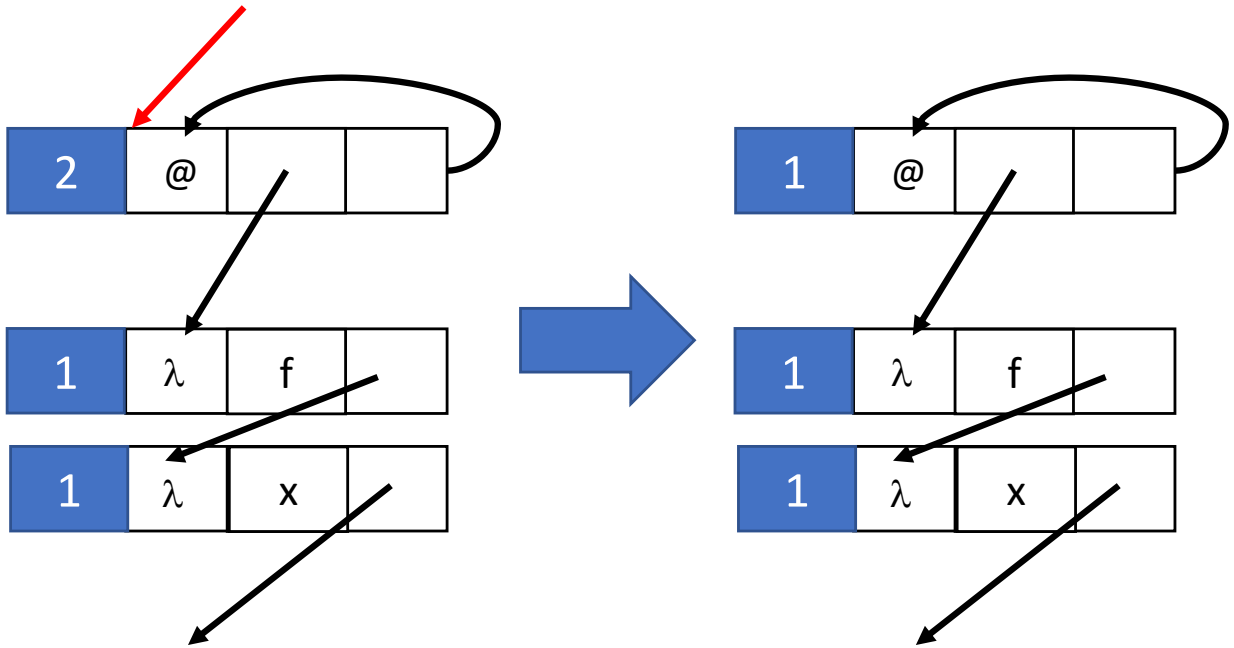
In the extreme, based on empirical observations that most blocks are not shared and only have one reference, "single-bit" reference counting can be used (unique/shared toggle bit, and only useful if you have a spare bit available in the header or pointer). Again, a tracing collector must be used periodically to manage the overflow of blocks that once were shared and have since become garbage

Limited-width reference counting collection can be viewed as an incremental method to reduce the frequency of mark-scan collection

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 4: CYCLIC REFERENCE COUNTING



Cyclic reference counting

Possibly the biggest problem with reference counting garbage collection is the fact that it cannot reclaim cyclic structures

This figure illustrates the problem: here we have the result of the δ reduction of (Y h), giving h (Y h) where the argument (Y h) is implemented as a cyclic pointer to the root of the redex. The full function body is not shown, nor is the other argument to which this function is applied. We are only interested in the topmost block and the red arrow that points to it. The block has a reference count of 2 (the reference count field is shown in blue) and we can see both pointers to the block – one black, the other red. If the red arrow were deleted, there would be no other reference to the topmost block from anywhere else in the program – it should be detectable as garbage, and yet it would still have the black arrow pointing to itself and its reference count would never drop below one. Therefore it can never be detected as garbage and its memory can never be reclaimed

Furthermore, if the only pointers to blocks within the function body arise transitively from this topmost block then they should all be detectable as garbage and yet reference counting will never reclaim them because the topmost block will never be reclaimed

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 4: CYCLIC REFERENCE COUNTING

- Specific solution for functional languages (Friedman & Wise, Information Processing Letters 8(1):41-45, 1979)
- More general solutions:
  - Bobrow: distinguishes pointers internal to a cycle (between members of the same cycle) from external references (pointers to a member of a cycle that does not come from a member of the same cycle)
  - Several other techniques distinguish cycle-closing pointers ("weak" pointers) from other pointers ("strong" pointers)

Cyclic reference counting (continued)

With functional languages cycles are created in very specific ways, and it is possible to extend reference counting to manage these cycles at runtime to ensure that (i) no part of a cycle is created before or survives after any other part, and (ii) when the last external pointer to a distinguished "lead" node of the cycle is deleted, the entire cycle can be reclaimed (Friedman & Wise, Information Processing Letters 8(1):41-45, 1979)

More general techniques distinguish different types of pointers:

- Bobrow's technique distinguishes pointers *internal* to a cycle (between members of the same cycle) from *external* references (pointers to a member of a cycle that do not come from a member of the same cycle)
- Several other techniques distinguish cycle-closing pointers ("weak" pointers) from other pointers ("strong" pointers)

We will explore examples both of these more general techniques in the following slides

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 4: CYCLIC REFERENCE COUNTING

Bobrow's technique

```
copy4(r, p) =
   groupidp = read(p - HEADERSIZE)
   groupidr  = read(r – HEADERSIZE)
   temp = read(r)
   groupidt  = read(temp – HEADERSIZE)
   if (groupidr != groupidp) increment_groupRC(p)
   if (groupidr != groupidt) {
      decrement_groupRC(temp)
      if (groupRC(temp) == 0) reclaim_group(temp)
   }
   write(r,p)
   return()
```

temp is silently deleted

Bobrow's technique

Bobrow (ACM TOPLAS 2(3):269-273, 1980) describes a technique where every allocated block is assigned to a group

- Each group of blocks has its own separate reference-count
- Each block has a link to its group's reference count
- When a pointer is overwritten, the groups of the three blocks involved in the transaction are examined – as illustrated by the code for *copy4(r, p)* shown on this slide
- For simplicity assume r, temp are the first words in their block data areas

This code does not manage reference counts for blocks – only for groups. It checks whether r and p refer to blocks in the same group (if not, a new external pointer to p's group is being created, so increment p's group reference count). It then checks whether r and t refer to blocks in the same group (if not, an external pointer to t's group is being deleted: the group reference count is decremented and if necessary the entire group is reclaimed)

This technique only reclaims intra-group cycles, not inter-group cycles. Thus, ideally, each group should correspond to a strongly connected component (each block reachable from every other block) – expensive in general, but less so for functional languages

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

Weak-pointer algorithms

- A cycle is never created when a block is created:
  - the first pointer to a new block is called *strong*
  - copied pointers may cause a cycle and these are called *weak*
  - All live blocks are reachable by tracing strong pointers
  - The network of strong pointers is never cyclic
- The technique requires both blocks and pointers to be tagged with a single (strong/weak) bit:
  - for blocks this will be in the header
  - for pointers this will either be the top bit (implies reduced address space) or an extra bit in the block holding the pointer

Weak-pointer algorithms

The other kind of technique for collecting cyclic structures with reference counting is to distinguish *weak* and *strong* pointers

A cycle is not created when a block is created, and the first pointer to a new block is called a *strong* pointer. By contrast, when a pointer is copied and the copy saved in a memory location, this might create a cycle, and these copies are called *weak* pointers

- Every live block can always be reached by tracing strong pointers from the Root Set
- The network of strong pointers starting from the Root Set is never cyclic

Weak-pointer algorithms for reference counting have been developed over many years with contributions from different researchers. It is very difficult to ensure accuracy and termination under all scenarios with acceptable efficiency. We present a simple version, despite known limitations

The technique needs both blocks and pointers to be tagged with a *strength* bit:

- for blocks this will be in the header
- for pointers this may be the top bit of the pointer (which implies a reduced address space) or an extra bit held with the pointer

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

Weak-pointer algorithms (continued)

- Each block header holds a count of strong pointers to it – the Strong Reference Count (SRC) – and a count of weak pointers to it – the Weak Reference Count (WRC)
- If SRC=0 and WRC=0 then the block must be reclaimed
- Any block with SRC > 0 cannot be garbage
- Any block with SRC = 0 and WRC > 0 might be garbage
- When the last strong pointer to a block is deleted:
    - turn all the weak pointers into strong pointers
    - recursively follow all children pointers and if a pointer to this block is found then weaken it
    - if SRC = 0, this block must be part of a garbage cycle so reclaim it

Weak-pointer algorithms (continued)

In the most general case, each block header holds a count of strong pointers to it – the Strong Reference Count (SRC) – and a count of weak pointers to it – the Weak Reference Count (WRC)

If SRC=0 and WRC=0 then clearly the block is garbage and must be reclaimed

Any block with SRC > 0 cannot be garbage, and so it should not be reclaimed

However, any block with SRC = 0 and WRC > 0 might be garbage (because it might be part of a cycle, all of whose blocks are garbage) and therefore it requires further analysis

When the last strong pointer to a block is deleted:

- turn all the weak pointers to that block into strong pointers
- recursively follow all children pointers in this block and if a pointer to this block is found then weaken it
- thereafter, if SRC = 0 this block must be part of a garbage cycle so reclaim it

Representing strong/weak pointers

- a *strong* pointer has the same strength tag (0 or 1) as the block it references
- a *weak* pointer has a different strength tag to the block it references

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

```
malloc5(n) =
   if (FLP == null) then abort "Out of memory"
   (strength, rc, next, size) = read(FLP – HEADERSIZE)
   prev = 0, temp = FLP
   while (size<n) {
      if (prev == 0) prev = FLP else prev = temp
      temp = next
      (strength, rc, next, size) = read(temp-HEADERSIZE)
   }
   if (prev == 0) then FLP = next
                  else  write(prev-HEADERSIZE, (0,0,next))
   write(temp-HEADERSIZE, (1,1,0))
   for F in fields(temp) { write(F,0) }
   temp = setstrengthhigh(temp)
   return(temp)
```

resetting the free list

Weak-pointer code: malloc5()

The code for *malloc5()* (often called "new()") is straightforward and almost identical to the code previously given on Slide 6. Differences are shown in red

Assume the header for each block contains the strength bit, two reference counts (if strength = 1, the first is SRC and the second is WRC, but their roles are reversed if strength=0) and the block size. Assume that when the block is on the free list: strength = 0, its first reference count = 0, and the second reference count contains the pointer to the next block on the free list

When a suitable block has been found it is initialised as follows:

- the strength bit is set high (i.e. 1)
- SRC=1
- WRC=0

We need to ensure that no pointers exist in the data area of the newly allocated block, so every field in the data area is set to zero

We assume the pointer strength bit is stored in the pointer itself (at the cost of a reduction in available address space). A special function *setstrengthhigh()* is called to set the strength bit in the pointer *temp* to be high (1)

Because the pointer and the block both have their strength bits set to the same value (1), this is a strong pointer for this block

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

```
copy5(r,p) =
   (strength, rc1, rc2) = read(p - HEADERSIZE)
   if (strength == 1) then rc2 = rc2 + 1
                      else  rc1 = rc1 + 1
   write(p – HEADERSIZE, (strength, rc1, rc2))
   temp = read(r)
   delete5(temp)
   p = setstrengthweak(p, strength)
   write(r,p)
   return()
```

Weak-pointer code: copy5()

This code is very similar to that shown previously on Slide 8, with differences in red. As stated on the previous slide, the block header is assumed to contain a strength bit, two reference count fields, and the size (not used in this code)

When copying a pointer this will increase the WRC for the block pointed to by *p*. Whether *rc1* or *rc2* is the WRC depends on whether *strength* is 0 or 1

As previously, the pointer currently held in the location pointed to by *r* will be deleted

The pointer about to be stored into location *r* must have its strength bit set using the special function *setstrengthweak(p, strength)* – this sets *p*'s strength bit to be the opposite of *strength* (thereby making it weak). An alternative approach, used where more is known about the ways in which cycles are created, is to create a weak pointer only at the point where a cycle is created (this is the approach taken by Brownbridge's 1985 algorithm). The approach taken here is to let all copied pointers be weak including those that create cycles

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

```
delete5(p) =
   (s, rc1, rc2) = read(p - HEADERSIZE)
   if  (getstrength(p) != s)
   then { if (s == 1) then write(p-HEADERSIZE,(s,rc1,rc2-1))
                      else  write(p-HEADERSIZE,(s,rc1-1,rc2)) }
   else {  if (s == 1) then { rc1 = rc1 – 1 } else { rc2 = rc2 – 1 }
           write(p-HEADERSIZE,(s,rc1,rc2))
           (src, wrc) = ([rc2, rc1]!s, [rc1, rc2]!s)
           if (src > 0) then {  return()  }
           if (src + wrc = 0) then {  free5(p); return()  }
           invert_blockstrength(p);
           for each M in children(p) {  suicide5(p, M)  }
           (s, rc1, rc2) = read(p - HEADERSIZE)
           if ([rc2, rc1]!s == 0) then {  free5(P)  }
   }  return()

free5(p) = for each M in children(p) { delete5(M) }
            write(p – HEADERSIZE, (0,0,FLP)
            FLP = p
```

Weak-pointer code: delete5()

Deletion of pointers is entirely new code since it must consider several cases – the code for *delete5()* and *suicide5()* (see next slide) should work together, but their interaction is complex

If the strength of the pointer p is different from that of the block, it is a weak pointer so just decrement the WRC – no further action is needed

<u>If p is a strong pointer</u>

- Decrement SRC – if *s* is 1, that's *rc1*, otherwise it's *rc2*
- If SRC (after decrement) is >0 then no further action is needed
- If SRC (after decrement) and WRC are both zero, free the block and return
- Otherwise SRC must be 0 and WRC must be >0, so do the following:
  - toggle the strength bit held in the block pointed to by *p* (this makes all the weak pointers strong)
  - for each child call the function *suicide(p,M)* to trace pointers and see if this is a cycle that must be deleted (if *suicide()* finds a copy of *p* it toggles that copy's strength bit so it becomes a weak pointer)
  - re-read the block header to see if it has changed – if SRC is zero then free the block
  - otherwise the block strength is NOT re-inverted, since all live blocks must be reachable by at least one strong pointer

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

```
suicide5(start, M) =
   (s, rc1, rc2) = read(M - HEADERSIZE)
   if  (getstrength(M) == s) {
      if (*M=*start) then { setstrengthweak(M,s); return() }
      if (SRC(M)>1) then { setstrengthweak(M,s); return() }
      for each T in children(M) {  suicide5(start, T)  }
   }
   return()
```

Weak-pointer code: suicide5()

The code for *suicide5(p,M)* is subtle because it must both determine whether the block pointed to by p can be freed and it must adjust the strong/weak property of pointers to ensure that every live block remains reachable via strong pointers from the Root Set

- The code takes no action if M is a weak pointer, so first it checks whether it is strong
- Next it checks whether the memory addresses of M and start are the same (i.e. without considering their strengths – this is indicated in the code by using a star) – if so, then a cycle has been found, the pointer M is weakened, and the function ends
- If the strong reference count of the block pointed to by M is greater than one, then M is set to be weak and the function ends. This is done because the other strong pointer indicates the block is definitely live and if the tracing were to be continued along this route (via this known-to-be-live block) and the original block *start* were found then we would NOT want to weaken the final strong pointer to this *start* block
- Finally, if none of the above apply then make a recursive call to *suicide5()* for each of the children pointers in this block

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

OPTIMISATION 5: CYCLIC REFERENCE COUNTING

- known modes of failure
- attempts to "correct" the algorithm in various ways either have introduced further problems:
  - non-termination in pathological cases, or
  - a very complex and expensive algorithm
- weak-pointer cyclic reference counting has good locality properties: attractive for distributed architectures
- but not sufficiently attractive for normal memory architectures
- often considered simpler to couple (i) non-cyclic reference counting with (ii) an occasional mark-scan collection to remove cyclic structures

The weak-pointer algorithm for cyclic reference counting described on the previous slides has some known modes of failure, and attempts to "correct" the algorithm in various ways either have introduced further problems (such as non-termination in pathological cases) or have created an algorithm that is very complex and very expensive

Although the locality property of weak-pointer cyclic reference counting is very attractive for distributed architectures, it is not sufficiently attractive for normal memory architectures and it is often considered simpler to couple non-cyclic reference counting with an occasional mark-scan collection to remove cyclic structures

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING
REFERENCE COUNTING GC

FOR AND AGAINST REFERENCE COUNTING GC

Against	For
	Operates well in tightly-confined heaps
	Has very good locality properties
	Operation is distributed in time and space
The cost of deleting the last pointer to a block may be very large	This is easily solved with non-recursive freeing (Optimisation 1)
The total cost of adjusting reference counts is significantly greater than the total cost of tracing garbage collection methods	This is easily solved with deferred reference counting (Optimisation 2)
It has a substantial space overhead (the counters in each block header take up a lot of room)	This is mostly solved with limited-width reference counting (Optimisation 3)
It is unable to reclaim cyclic structures	Solutions are tricky: special extensions can be complex and expensive It is easy to combine reference counting with a tracing collector, run periodically

For and Against

Reference-counting garbage collection operates well in tightly-confined heaps

It has very good locality properties – only those blocks that are modified are monitored for garbage collection

Its operation is distributed in both time and space

The cost of deleting the last pointer to a block may be very large since any pointers contained within that block must also be deleted, each of which may cause another block to be freed, and so on

This is easily solved with non-recursive freeing (Optimisation 1)

The total cost of adjusting reference counts is significantly greater than the total cost of tracing garbage collection methods

This is easily solved with deferred reference counting (Optimisation 2)

It has a substantial space overhead (the counters in each block header take up a lot of room)

This is mostly solved with limited-width reference counting (Optimisation 3)

It is unable to reclaim cyclic structures

Solutions are tricky: special extensions can be complex and expensive

It is easy to combine reference counting with a tracing collector, run periodically

© 2020-2021 Christopher D. Clack

FUNCTIONAL PROGRAMMING REFERENCE COUNTING GC

SUMMARY

- Overview
- Assumptions
- Simple imperative pseudo-code
- Optimisations
- For and against Reference Counting GC

In summary, this lecture has explored the third and final of three canonical garbage collection algorithms – Reference Counting Garbage Collection

The lecture provided simple imperative pseudo-code for the basic reference counting garbage collection functions and for a number of optimisations to basic reference counting:

- non-recursive freeing
- deferred reference counting
- limited-width counts
- cyclic reference counting

The lecture then ended with a summary of good and bad characteristics of Reference Counting Garbage Collection

© 2020-2021 Christopher D. Clack