BOSC2013 OO3 - Garbage Collection

Sigurt Bladt Dinesen
sidi@itu.dk

December 9, 2013

Contents

1 Re-handin **2**

2 Exercise 1 **2**
2.1 Exercise 1 (i) 2
2.2 Exercise 1 (ii) 2
2.3 Exercise 1 (iii) 3
2.4 Exercise 1 (iv) 3

3 Exercise 2 **3**
3.1 The "mark" phase 3
3.2 The "sweep" phase 3

4 Exercise 3 and 4 **4**

5 Exercise 5 **4**

6 Exercise 6 **4**

7 Testing **5**

8 Appendix/code **6**

1 Re-handin

Exercise 5 is now described in section 5.

Exercise 6 has been completed, and is described in section 6.

2 Exercise 1

2.1 Exercise 1 (i)

Write 3-10 line descriptions of how the abstract machine executes ADD, CSTI i, NIL, IFZERO, CONS, CAR , and SETCAR.

sp denotes the stack pointer, and $s[i]$ the element in the i'th slot of of the stack.

- ADD - $s[sp-1]$ is set to the sum of $s[sp]$ and $s[sp-1]$, sp is decremented by 1.
- CSTI i - the stack machine pushes the integer i onto the stack, and increments sp. To indicate that i is an integer rather than a reference, i's binary representation is shifted one bit to the left, and the rightmost bit is set to 1. We refer to this as "tagging".
- NIL - the stack machine pushes the untagged integer (I.E. a reference) 0 onto the stack.
- IFZERO - the stack machine checks if $s[sp]$ is 0 or NIL, by first untagging it if it is an integer. It then sets the program counter to the "argument" of IFZERO if $s[sp]$ is zero or NIL, and increments it by one otherwise. The stack pointer is decremented by one regardless.
- CONS - the stack machine allocates three words on the heap, and sets the non-header words to $s[sp-1]$ and $s[sp]$ respectively.
- CAR - interprets $s[sp]$ as a word array, raising an error if it is zero, and otherwise replaces it with the first word in the array. sp is decremented by one.
- SETCAR - the stack machine assumes $s[sp]$ is a word and that $s[sp-1]$ is a word array. Then sets the first non-header word of $s[sp-1]$ to $s[sp]$ and decrements sp.

2.2 Exercise 1 (ii)

Describe the C macros Length, Color, and Paint, and their effect on the 32-bit header blocks

- Length(hdr) - shifts hdr 2 bits to the right, in effect ignoring the color bits, and applies the binary and operation to the result and the binary value consisting of 22 ones. The result is the value of the bits 9 though 30 i.e the length bits.

- Color(hdr) - By applying binary and to hdr and 3 (11 in binary) the value of the last two bits is obtained - the color bits.
- Paint(hdr, color) - applying binary and to hdr and the result of not 3 returns hdr with both color bits set to zero. Applying binary or to that, and color, returns hdr with the color bits set to the input color.

2.3 Exercise 1 (iii)

When does the abstract machine (or mutator) interact with the garbage collector?

The stack machine directly calls the `allocate` method when it puts cons cells on the heap. The `allocate` method may then call `collect`, which frees up heap space.

2.4 Exercise 1 (iv)

Under what circumstances is `collect` *called*

`collect` is called whenever the list machine attempts to allocate more data than can be fit into a block on the free-list. I.e. when there is no more memory available on the heap.

3 Exercise 2

Implement the "mark and sweep" algorithm for the listmachine

3.1 The "mark" phase

My first implementation of the mark phase stepped through the stack, looking for references to the heap. It did so under the assumption that if a stack element is not an integer, it is a heap reference. For each heap reference on the stack, a recursive function `mark` was called to color the referenced block white (mark it), and do the same to any heap references that block may have contained.

3.2 The "sweep" phase

The sweep phase steps though every block on the heap, and puts white (marked) blocks on the free-list. Black blocks (in-use blocks) are colored white, so that the next sweep will consider them free unless they are reached in the next mark phase.

An auxiliary function called `freeBlock` is used for putting blocks on the free-list. It does so by treating the list as a linked list (because it is) and then using the standard way of inserting elements into an ordered linked list. This means the time complexity of `freeBlock` is $\Theta(n)$ in the number of blocks allready on the free-list.[1] This makes the `sweepPhase` function $\Theta(n^2)$ in the

[1] Θ as in Bachmann–Landau

number of free blocks. Alternatively, the `sweepPhase` function could discard and rebuild the free-list anew, allowing a complexity of $\Theta(n)$.

4 Exercise 3 and 4

Joining adjacent free blocks

This is done by by `sweepPhase` letting the function `joinFree` determine the how big a step to take through the heap. `joinFree` increases the size bits of the header-block it takes as argument, so that it is joined with any free blocks immediately following it. It then returns the size of the new block to be the stepping value of `sweepPhase`.

5 Exercise 5

Implement the `markPhase` in a non-recursive fashion

The non-recursive `markPhase` function iterates through the stack, painting referred headers grey. It then invokes the `traverse` function until there are no grey headers on the heap.

`traverse` iterates through the heap, painting grey headers black, and painting any headers they refer grey. It returns `TRUE` or `FALSE`, indicating whether there are still any grey blocks on the heap.

6 Exercise 6

Implement "stop and copy" garbage collection

The "stop and copy" allocate method, `allocateStopAndCopy` works simply by putting the requested block in the first available slot in the free-list (which is just the last, free, part of the from-heap). If there are not enough free words, `collectStopAndCopy` is called.

`collectStopAndCopy` iterates through the heap, invoking `copyBlock` on each heap reference it finds, replacing the stack content with the result.

`copyBlock` takes a pointer to a block header, copies the entire block to the free-list (which is now on the to-heap), and returns it new address on the to-heap.

`collectStopAndCopy` then invokes `copyFromTo`, which iterates through every occupied block on the to-heap, looking for references to the to-heap (live blocks that have not yet been copied). `copyFromTo` then moves such refered blocks to the to-heap, marking the block on the from-heap with a "forward" pointer to the to-heap, so that it can avoid copying the same block more than once. Finally, `copyFromTo` swaps the to- and from-heap.

7 Testing

To test the stop and copy implementation, simply make the default make target, and run the executable listmachine on the exXX.out files.

Testing the mark and sweep implementation takes a little work (I know, I'm sorry). To test mark and sweep, the following lines in listmachine.c need to be commented in/out respectively: 638/639 and 303/304. Then the make/run process is the same as for stop and copy.

8 Appendix/code

listmachine.c

```
1  /* File ListC/listmachine.c
2     A unified-stack abstract machine and garbage collector
3     for list-C, a variant of micro-C with cons cells.
4     sestoft@itu.dk * 2009-11-17, 2012-02-08
5
6     Compile like this, on ssh.itu.dk say:
7        gcc -Wall listmachine.c -o listmachine
8
9     If necessary, force compiler to use 32 bit integers:
10       gcc -m32 -Wall listmachine.c -o listmachine
11
12    To execute a program file using this abstract machine,
          do:
13       ./listmachine <programfile> <arg1> <arg2> ...
14    To get also a trace of the program execution:
15       ./listmachine -trace <programfile> <arg1> <arg2>
             ...
16
17    This code assumes -- and checks -- that values of type
18    int, unsigned int and unsigned int* have size 32 bits.
19 */
20
21 /*
22    Data representation in the stack s[...] and the heap:
23     * All integers are tagged with a 1 bit in the least
          significant
24       position, regardless of whether they represent
             program integers,
25       return addresses, array base addresses or old base
             pointers
26       (into the stack).
27     * All heap references are word-aligned, that is, the
          two least
28       significant bits of a heap reference are 00.
29     * Integer constants and code addresses in the program
          array
30       p[...] are not tagged.
31    The distinction between integers and references is
          necessary for
32    the garbage collector to be precise (not conservative)
          .
33
34    The heap consists of 32-bit words, and the heap is
          divided into
35    blocks.  A block has a one-word header block[0]
          followed by the
```

```
36      block's contents: zero or more words block[i], i=1..n.
37
38      A header has the form ttttttttnnnnnnnnnnnnnnnnnnnnnngg
39      where tttttttt is the block tag, all 0 for cons cells
40            nn....nn is the block length (excluding header)
41            gg       is the block's color
42
43      The block color has this meaning:
44      gg=00=White: block is dead (after mark, before sweep)
45      gg=01=Grey:  block is live, children not marked (
            during mark)
46      gg=10=Black: block is live (after mark, before sweep)
47      gg=11=Blue:  block is on the freelist or orphaned
48
49      A block of length zero is an orphan; it cannot be used
50      for data and cannot be on the freelist.  An orphan is
51      created when allocating all but the last word of a
            free block.
52  */
53
54  #include <stdlib.h>
55  #include <string.h>
56  #include <stdio.h>
57  #include <sys/time.h>
58  #include <sys/resource.h>
59  #include <assert.h>
60
61  typedef unsigned int word;
62
63  #define IsInt(v) (((v)&1)==1)
64  #define Tag(v) (((v)<<1)|1)
65  #define Untag(v) ((v)>>1)
66
67  #define White 0
68  #define Grey  1
69  #define Black 2
70  #define Blue  3
71
72  #define BlockTag(hdr) ((hdr)>>24)
73  #define Length(hdr)   (((hdr)>>2)&0x003FFFFF)
74  #define Color(hdr)    ((hdr)&3)
75  #define Paint(hdr, color)  (((hdr)&(~3))|(color))
76
77  #define CONSTAG 0
78
79  // Heap size in words
80
81  #define HEAPSIZE 1000
82
83  #define TRUE 1
```

```
84  #define FALSE 0
85
86  word* heapFrom;
87  word* heapTo;
88  word* afterFrom;
89  word* afterTo;
90  word* heap;
91  word* afterHeap;
92  word *freelist;
93
94  // These numeric instruction codes must agree with ListC/
        Machine.fs:
95  // (Use #define because const int does not define a
        constant in C)
96
97  #define CSTI 0
98  #define ADD 1
99  #define SUB 2
100 #define MUL 3
101 #define DIV 4
102 #define MOD 5
103 #define EQ 6
104 #define LT 7
105 #define NOT 8
106 #define DUP 9
107 #define SWAP 10
108 #define LDI 11
109 #define STI 12
110 #define GETBP 13
111 #define GETSP 14
112 #define INCSP 15
113 #define GOTO 16
114 #define IFZERO 17
115 #define IFNZRO 18
116 #define CALL 19
117 #define TCALL 20
118 #define RET 21
119 #define PRINTI 22
120 #define PRINTC 23
121 #define LDARGS 24
122 #define STOP 25
123 #define NIL 26
124 #define CONS 27
125 #define CAR 28
126 #define CDR 29
127 #define SETCAR 30
128 #define SETCDR 31
129
130 #define STACKSIZE 1000
131
```

```
132 // Print the stack machine instruction at p[pc]
133
134 void printInstruction(int p[], int pc) {
135   switch (p[pc]) {
136   case CSTI:   printf("CSTI %d", p[pc+1]); break;
137   case ADD:    printf("ADD"); break;
138   case SUB:    printf("SUB"); break;
139   case MUL:    printf("MUL"); break;
140   case DIV:    printf("DIV"); break;
141   case MOD:    printf("MOD"); break;
142   case EQ:     printf("EQ"); break;
143   case LT:     printf("LT"); break;
144   case NOT:    printf("NOT"); break;
145   case DUP:    printf("DUP"); break;
146   case SWAP:   printf("SWAP"); break;
147   case LDI:    printf("LDI"); break;
148   case STI:    printf("STI"); break;
149   case GETBP:  printf("GETBP"); break;
150   case GETSP:  printf("GETSP"); break;
151   case INCSP:  printf("INCSP %d", p[pc+1]); break;
152   case GOTO:   printf("GOTO %d", p[pc+1]); break;
153   case IFZERO: printf("IFZERO %d", p[pc+1]); break;
154   case IFNZRO: printf("IFNZRO %d", p[pc+1]); break;
155   case CALL:   printf("CALL %d %d", p[pc+1], p[pc+2]);
        break;
156   case TCALL:  printf("TCALL %d %d %d", p[pc+1], p[pc+2],
        p[pc+3]); break;
157   case RET:    printf("RET %d", p[pc+1]); break;
158   case PRINTI: printf("PRINTI"); break;
159   case PRINTC: printf("PRINTC"); break;
160   case LDARGS: printf("LDARGS"); break;
161   case STOP:   printf("STOP"); break;
162   case NIL:    printf("NIL"); break;
163   case CONS:   printf("CONS"); break;
164   case CAR:    printf("CAR"); break;
165   case CDR:    printf("CDR"); break;
166   case SETCAR: printf("SETCAR"); break;
167   case SETCDR: printf("SETCDR"); break;
168   default:     printf("<unknown>"); break;
169   }
170 }
171
172 // Print current stack (marking heap references by #) and
      current instruction
173
174 void printStackAndPc(int s[], int bp, int sp, int p[],
     int pc) {
175   printf("[ ");
176   int i;
177   for (i=0; i<=sp; i++)
```

```
178     if (IsInt(s[i]))
179       printf("%d ", Untag(s[i]));
180     else
181       printf("#%d ", s[i]);
182   printf("]");
183   printf("{%d:", pc); printInstruction(p, pc); printf("}\
        n");
184 }
185
186 // Read instructions from a file, return array of
      instructions
187
188 int* readfile(char* filename) {
189   int capacity = 1, size = 0;
190   int *program = (int*)malloc(sizeof(int)*capacity);
191   FILE *inp = fopen(filename, "r");
192   int instr;
193   while (fscanf(inp, "%d", &instr) == 1) {
194     if (size >= capacity) {
195       int* buffer = (int*)malloc(sizeof(int) * 2 *
            capacity);
196       int i;
197       for (i=0; i<capacity; i++)
198         buffer[i] = program[i];
199       free(program);
200       program = buffer;
201       capacity *= 2;
202     }
203     program[size++] = instr;
204   }
205   fclose(inp);
206   return program;
207 }
208
209 word* allocate(unsigned int tag, unsigned int length, int
      s[], int sp);
210 word* allocateStopAndCopy(unsigned int tag, unsigned int
      length, int s[], int sp);
211
212 // The machine: execute the code starting at p[pc]
213
214 int execcode(int p[], int s[], int iargs[], int iargc,
      int /* boolean */ trace) {
215   int bp = -999;          // Base pointer, for local
        variable access
216   int sp = -1;            // Stack top pointer
217   int pc = 0;             // Program counter: next
        instruction
218   for (;;) {
219     if (trace)
```

```
220       printStackAndPc(s, bp, sp, p, pc);
221     switch (p[pc++]) {
222     case CSTI:
223       s[sp+1] = Tag(p[pc++]); sp++; break;
224     case ADD:
225       s[sp-1] = Tag(Untag(s[sp-1]) + Untag(s[sp])); sp--;
              break;
226     case SUB:
227       s[sp-1] = Tag(Untag(s[sp-1]) - Untag(s[sp])); sp--;
              break;
228     case MUL:
229       s[sp-1] = Tag(Untag(s[sp-1]) * Untag(s[sp])); sp--;
              break;
230     case DIV:
231       s[sp-1] = Tag(Untag(s[sp-1]) / Untag(s[sp])); sp--;
              break;
232     case MOD:
233       s[sp-1] = Tag(Untag(s[sp-1]) % Untag(s[sp])); sp--;
              break;
234     case EQ:
235       s[sp-1] = Tag(s[sp-1] == s[sp] ? 1 : 0); sp--;
            break;
236     case LT:
237       s[sp-1] = Tag(s[sp-1] < s[sp] ? 1 : 0); sp--; break
            ;
238     case NOT: {
239       int v = s[sp];
240       s[sp] = Tag((IsInt(v) ? Untag(v) == 0 : v == 0) ? 1
              : 0);
241     } break;
242     case DUP:
243       s[sp+1] = s[sp]; sp++; break;
244     case SWAP:
245       { int tmp = s[sp];  s[sp] = s[sp-1];  s[sp-1] = tmp
            ; } break;
246     case LDI:                   // load indirect
247       s[sp] = s[Untag(s[sp])]; break;
248     case STI:                   // store indirect, keep
          value on top
249       s[Untag(s[sp-1])] = s[sp]; s[sp-1] = s[sp]; sp--;
            break;
250     case GETBP:
251       s[sp+1] = Tag(bp); sp++; break;
252     case GETSP:
253       s[sp+1] = Tag(sp); sp++; break;
254     case INCSP:
255       sp = sp+p[pc++]; break;
256     case GOTO:
257       pc = p[pc]; break;
258     case IFZERO: {
```

```
259       int v = s[sp--];
260       pc = (IsInt(v) ? Untag(v) == 0 : v == 0) ? p[pc] :
              pc+1;
261     } break;
262     case IFNZRO: {
263       int v = s[sp--];
264       pc = (IsInt(v) ? Untag(v) != 0 : v != 0) ? p[pc] :
              pc+1;
265     } break;
266     case CALL: {
267       int argc = p[pc++];
268       int i;
269       for (i=0; i<argc; i++)                  // Make room
              for return address
270         s[sp-i+2] = s[sp-i];                  // and old
                base pointer
271       s[sp-argc+1] = Tag(pc+1); sp++;
272       s[sp-argc+1] = Tag(bp);   sp++;
273       bp = sp+1-argc;
274       pc = p[pc];
275     } break;
276     case TCALL: {
277       int argc = p[pc++];                     // Number of
              new arguments
278       int pop  = p[pc++];                     // Number of
              variables to discard
279       int i;
280       for (i=argc-1; i>=0; i--)     // Discard variables
281         s[sp-i-pop] = s[sp-i];
282       sp = sp - pop; pc = p[pc];
283     } break;
284     case RET: {
285       int res = s[sp];
286       sp = sp-p[pc]; bp = Untag(s[--sp]); pc = Untag(s[--
              sp]);
287       s[sp] = res;
288     } break;
289     case PRINTI:
290       printf("%d ", IsInt(s[sp]) ? Untag(s[sp]) : s[sp]);
              break;
291     case PRINTC:
292       printf("%c", Untag(s[sp])); break;
293     case LDARGS: {
294       int i;
295       for (i=0; i<iargc; i++) // Push commandline
              arguments
296         s[++sp] = Tag(iargs[i]);
297     } break;
298     case STOP:
299       return 0;
```

```
300       case NIL:
301         s[sp+1] = 0; sp++; break;
302       case CONS: {
303         //word* p = allocate(CONSTAG, 2, s, sp);
304         word* p = allocateStopAndCopy(CONSTAG, 2, s, sp);
305         p[1] = (word)s[sp-1];
306         p[2] = (word)s[sp];
307         s[sp-1] = (int)p;
308         sp--;
309       } break;
310       case CAR: {
311         word* p = (word*)s[sp];
312         if (p == 0)
313           { printf("Cannot take car of null\n"); return -1;
                }
314         s[sp] = (int)(p[1]);
315       } break;
316       case CDR: {
317         word* p = (word*)s[sp];
318         if (p == 0)
319           { printf("Cannot take cdr of null\n"); return -1;
                }
320         s[sp] = (int)(p[2]);
321       } break;
322       case SETCAR: {
323         word v = (word)s[sp--];
324         word* p = (word*)s[sp];
325         p[1] = v;
326       } break;
327       case SETCDR: {
328         word v = (word)s[sp--];
329         word* p = (word*)s[sp];
330         p[2] = v;
331       } break;
332       default:
333         printf("Illegal instruction %d at address %d\n", p[
              pc-1], pc-1);
334         return -1;
335       }
336     }
337   }
338
339   // Read program from file, and execute it
340
341   int execute(int argc, char** argv, int /* boolean */
        trace) {
342     int *p = readfile(argv[trace ? 2 : 1]);          //
          program bytecodes: int[]
343     int *s = (int*)malloc(sizeof(int)*STACKSIZE);    //
          stack: int[]
```

```
344   int iargc = trace ? argc - 3 : argc - 2;
345   int *iargs = (int*)malloc(sizeof(int)*iargc);   //
          program inputs: int[]
346   int i;
347   for (i=0; i<iargc; i++)                         //
          Convert commandline arguments
348     iargs[i] = atoi(argv[trace ? i+3 : i+2]);
349   // Measure cpu time for executing the program
350   struct rusage ru1, ru2;
351   getrusage(RUSAGE_SELF, &ru1);
352   int res = execcode(p, s, iargs, iargc, trace);  //
          Execute program proper
353   getrusage(RUSAGE_SELF, &ru2);
354   struct timeval t1 = ru1.ru_utime, t2 = ru2.ru_utime;
355   double runtime = t2.tv_sec-t1.tv_sec+(t2.tv_usec-t1.
          tv_usec)/1000000.0;
356   printf("\nUsed %7.3f cpu seconds\n", runtime);
357   return res;
358 }
359
360 // Garbage collection and heap allocation
361
362 word mkheader(unsigned int tag, unsigned int length,
       unsigned int color) {
363   return (tag << 24) | (length << 2) | color;
364 }
365
366 int inHeapTo(word* p) {
367   return heapTo <= p && p < afterTo;
368 }
369
370 int inHeapFrom(word* p) {
371   return heapFrom <= p && p < afterFrom;
372 }
373
374 int inHeap(word* p) {
375   return heap <= p && p < afterHeap;
376 }
377
378 // Call this after a GC to get heap statistics:
379
380 void heapStatistics() {
381   int blocks = 0, free = 0, orphans = 0,
382     blocksSize = 0, freeSize = 0, largestFree = 0;
383   word* heapPtr = heap;
384   while (heapPtr < afterHeap) {
385     if (Length(heapPtr[0]) > 0) {
386       blocks++;
387       blocksSize += Length(heapPtr[0]);
388     } else
```

```
389         orphans++;
390       word* nextBlock = heapPtr + Length(heapPtr[0]) + 1;
391       if (nextBlock > afterHeap) {
392         printf("HEAP ERROR: block at heap[%d] extends
              beyond heap\n",
393                heapPtr-heap);
394         exit(-1);
395       }
396       heapPtr = nextBlock;
397     }
398     word* freePtr = freelist;
399     while (freePtr != 0) {
400       free++;
401       int length = Length(freePtr[0]);
402       if (freePtr < heap || afterHeap < freePtr+length+1) {
403         printf("HEAP ERROR: freelist item %d (at heap[%d],
              length %d) is outside heap\n",
404                free, freePtr-heap, length);
405         exit(-1);
406       }
407       freeSize += length;
408       largestFree = length > largestFree ? length :
            largestFree;
409       if (Color(freePtr[0])!=Blue)
410         printf("Non-blue block at heap[%d] on freelist\n",
              (int)freePtr);
411       freePtr = (word*)freePtr[1];
412     }
413     printf("Heap: %d blocks (%d words); of which %d free (%
          d words, largest %d words); %d orphans\n",
414            blocks, blocksSize, free, freeSize, largestFree,
                 orphans);
415 }
416
417 void initHeapStopAndCopy(){
418   heapFrom = (word*)malloc(sizeof(word)*HEAPSIZE*2);
419   heapTo = &heapFrom[(HEAPSIZE/2)-1];
420   afterFrom = heapTo;
421   afterTo = &heapTo[HEAPSIZE];
422   freelist = heapFrom;
423 }
424
425 void initheap() {
426   heap = (word*)malloc(sizeof(word)*HEAPSIZE);
427   afterHeap = &heap[HEAPSIZE];
428   // Initially, entire heap is one block on the freelist:
429   heap[0] = mkheader(0, HEAPSIZE-1, Blue);
430   heap[1] = (word)0;
431   freelist = &heap[0];
432 }
```

```
433
434 int traverse(){
435   int greys = FALSE;
436   word* header = heap;
437   while(inHeap(header)){
438     if(Grey == Color(*header)){
439       *header = Paint(*header, Black);
440       int len = Length(*header);
441       int i;
442       for(i=1; i <= len; i++){
443         if (IsInt(header[i]) || header[i] == 0)
444           continue;
445         if (White == Color(*(word*)header[i])){
446           greys = TRUE;
447           *(word*)header[i] = Paint(*(word*)header[i],
                Grey);
448         }
449       }
450     }
451       header += Length(*header) +1;
452   }
453   return greys;
454 }
455
456 void markPhase(int s[], int sp) {
457   printf("marking ...\n");
458   int i;
459   for (i=0; i<=sp; i++) {
460     if (IsInt(s[i]) || 0 == s[i]) //only treat heap
         references
461       continue;
462     *(word*)s[i] = Paint(*(word*)s[i], Grey);
463   }
464   int greys = TRUE;
465   while (greys) {
466     greys=traverse();
467   }
468 }
469
470 void freeBlock(word* block){
471   word* free = freelist;
472   word** prev = &freelist;
473   while(free < block && inHeap(free)){
474     prev = (word**)&free[1];
475     free = (word*) free[1];
476   }
477   assert(free != block);
478   block[1] = (int) free;
479   *prev = block;
480   *block = Paint(*block, Blue);
```

```
481 }
482
483 int joinFree(word* header)
484 {
485   int size = 0;
486   word* block = header;
487   do {
488     int len = Length(*block)+1;
489     size += len;
490     block += len;
491     assert(len>0);
492   } while(inHeap((word*)block) && White == Color(*block))
        ;
493   size--;
494   *header = mkheader(CONSTAG, size, White); //the tag
        doesn't matter, so CONSTAG is used as a convenience
495   return size;
496 }
497
498
499 void sweepPhase() {
500   printf("sweeping ...\n");
501   word* header=heap;
502   while(header < afterHeap){
503     int color=Color(*header);
504     int stepNext = Length(*header) +1;
505     switch (color)
506     {
507       case White:
508         assert(Length(*header) >= 2);
509         freeBlock(header);
510         header += joinFree(header)+1;
511         break;
512       case Black:
513         *header = Paint(*header, White);
514         header += stepNext;
515         break;
516       default:
517         header += stepNext;
518         break;
519     }
520   }
521 }
522
523 word* copyBlock(word* block){
524   int j;
525   int len = Length(*block);
526   for(j=0; j<= len; j++){
527     freelist[j] = block[j];
528   }
```

```
529   block[1] = (int)freelist;
530   freelist += len +1;
531   return (word*)block[1];
532 }
533
534 void copyFromTo(){
535   word* header = heapTo;
536   while(header < freelist){ //all blocks in to-space
537     int i;
538     int len = Length(*header);
539     for(i=1; i<=len; i++){ //all words in block
540       if(IsInt(header[i]) || 0 == header[i]){ //only
              treat references, specifically non-nil
              references
541         continue;
542       } else if(inHeapFrom((word*)header[i])){ //pointer
            to the from-space
543         word* oldHeader = (word*)header[i];
544         if(!IsInt(oldHeader[1]) && inHeapTo((word*)
              oldHeader[1])){ //contains a forward pointer
              to the to-space
545           header[i] = oldHeader[1];
546         } else { //un-copied  value, needs to be copied
547           header[i] = (int) copyBlock(oldHeader);
548         }
549       }
550     }
551       header += len +1;
552   }
553   word* tmp = heapFrom;
554   heapFrom = heapTo;
555   heapTo = tmp;
556   tmp = afterFrom;
557   afterFrom = afterTo;
558   afterTo = tmp;
559 }
560
561 void collectStopAndCopy(int s[], int sp){
562   freelist = heapTo;
563   int i;
564   for(i=0; i<sp; i++){
565     if(!IsInt(s[i]) && s[i] != 0)
566       s[i] = (int) copyBlock((word*)s[i]);
567   }
568   copyFromTo();
569 }
570
571 void collect(int s[], int sp) {
572   markPhase(s, sp);
573   heapStatistics();
```

```
574   sweepPhase();
575   heapStatistics();
576 }
577
578 word* allocateStopAndCopy(unsigned int tag, unsigned int
        length, int s[], int sp){
579   int attempt = 1;
580   do {
581     word* newBlock = freelist;
582     freelist += length +1;
583     if(freelist <= afterFrom){ //is on heap
584       newBlock[0] = mkheader(tag, length, White);
585       return newBlock;
586     }
587     if(attempt == 1){
588       collectStopAndCopy(s, sp);
589     }
590   } while(attempt++ == 1);
591   printf("Out of memory\n");
592   exit(1);
593 }
594
595 word* allocate(unsigned int tag, unsigned int length, int
        s[], int sp) {
596   int attempt = 1;
597   do {
598     word* free = freelist;
599     word** prev = &freelist;
600     while (free != 0) {
601       int rest = Length(free[0]) - length;
602       if (rest >= 0)  {
603         if (rest == 0) // Exact fit with free block
604           *prev = (word*)free[1];
605         else if (rest == 1) { // Create orphan (unusable)
              block
606           *prev = (word*)free[1];
607           free[length+1] = mkheader(0, rest-1, Blue);
608         } else { // Make previous free block point to
              rest of this block
609           *prev = &free[length+1];
610           free[length+1] = mkheader(0, rest-1, Blue);
611           free[length+2] = free[1];
612         }
613         free[0] = mkheader(tag, length, White);
614         return free;
615       }
616       prev = (word**)&free[1];
617       free = (word*)free[1];
618     }
619     // No free space, do a garbage collection and try
```

```
        again
620     if (attempt==1)
621       collect(s, sp);
622   } while (attempt++ == 1);
623   printf("Out of memory\n");
624   exit(1);
625 }
626
627 // Read code from file and execute it
628
629 int main(int argc, char** argv) {
630   if (sizeof(word)!=4 || sizeof(word*)!=4 || sizeof(int)
        !=4) {
631     printf("Size of word, word* or int is not 32 bit,
          cannot run\n");
632     return -1;
633   } else if (argc < 2) {
634     printf("Usage: listmachine [-trace] <programfile> <
          arg1> ...\n");
635     return -1;
636   } else {
637     int trace = argc >= 3 && 0==strncmp(argv[1], "-trace
          ", 7);
638     //initheap();
639     initHeapStopAndCopy();
640     return execute(argc, argv, trace);
641   }
642 }
643 // vim: set ts=2 sw=2 et:
```