Obligatorisk opgave 3

Operativsystemer og C

Bachelor in Software Development,
IT-University of Copenhagen

Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

November 30th, 2012

Contents

1 Forord **2**

2 Opgavebesvarelse og implementation **3**

2.1 Opgave 1 3

2.2 Opgave 2, 3 og 4 4

2.3 Opgave 5 5

2.4 Opgave 6 5

A Kode **7**

A.1 Makefile 7

A.2 listmachineV1.c 8

A.3 listmachineV2.c 11

A.4 listmachineV3.c 14

1 Forord

I denne rapport dokumenterer vi vores valg i forhold til implementationen af opgaverne i Obligatorisk opgave 3.

Når vi refererer til "Programming Language Concepts", mener vi "Peter Sestoft, Programming Language Concepts, ISBN 978-1-4471-4156-3 (eBook)".

Vi har testet vores tre implementationer af garbage collection algoritmer på eksemplerne "ex30.lc", "ex35.lc" og "ex36.lc", som beskrevet i opgaven. Disse fulgte med "listc.zip"[1] mappen, som blev givet med opgaven.

For at skære ned på størrelsen af appendix, har vi kun vedlagt kode, som vi selv har ændret i / lavet.

Kildekode og testdokumentation kan findes i appendix på side 6. Vores git repository kan findes på https://github.com/esfdk/BOSC/tree/master/oo3.

[1] https://blog.itu.dk/BOSC-E2012/files/2012/11/listc.zip

2 Opgavebesvarelse og implementation

2.1 Opgave 1

I dette afsnit vil vi nævne "tagging" ofte. Med det mener vi, at "Tag" og "Untag" funktionerne er blevet anvendt på tallene. Et tal bliver tagget for at sørge for, at det ikke misforstås for en heap-reference.

2.1.1 Del 1

ADD untagger de to øverste elementer på stakken, ligger dem sammen, tagger den nye værdi og ligger den på toppen af stakken.

CSTI I tager den næste værdi i p[ ] arrayet og ligger den på toppen af stakken.

NIL Ligger 0 på toppen af stakken. Da nullet ikke bliver tagget, betyder det nil og ikke nul.

IFZERO tager det øverste element af stakken og decrementere stackpointeren med en. Den tjekker om v er en int. Hvis v er en int, untagges v og sammenlignes med nul, ellers sammenlignes v med NIL. Hvis sammenligningen er sand i tilfældet med nul bliver program counter sat til den nuværende værdi på p[pc] ellers bliver næste instruktion udført.

CONS laver en cons celle ud af de to øverste elementer på stakken, og decrementere så stack pointeren med en.

CAR henter et word fra stakken og tjekker om det er NIL, hvis det ikke er NIL tages det første element af cons cellen og ligges på toppen af stakken i stedet for det hentede word.

SETCAR henter det øverste element på stakken og et word. Den sætter wordets første værdi til at være den udhentede værdi på stakken.

2.1.2 Del 2

Length laver to bitwise right shifts, hvilket fjerner de to garbage collection bites. Derefter bruger den bitwise AND til at sammenligne length bitsne med 0x003FFFF. Dette giver os værdien af n bitsne, som er det tal, der repræsentere længden på en blok.

Color går ind og bruger bitwise AND til at sammenligne farven på cellen med 11, hvorved den finder cellens faktiske farve.

Paint producerer et tal, hvor garbage collection bitsne i inputtet er blevet lavet om til de bits, som svarer til farven, der er givet som input argument.

2.1.3 Del 3

"allocate" funktionen bliver kun kaldt i CONS casen i "execcode" funktionen.

Der er direkte interaktion mellem abstract machinen og garbage collectoren, da abstract machinen har en stack, hvor garbage collectoren skal gå ind og opdatere heap referencer. Hvis det ikke bliver gjort, vil casene

CAR,CDR,SETCAR og SETCDR i "execcode" ikke altid virke, da deres referencer kan være forældet og derfor pege et forkert sted i heapen.

2.1.4 Del 4

Hvis der ikke er nok plads tilbage på "freelist", når "allocate" funktionen bliver kaldt, bliver "collect" funktionen kaldt.

2.2 Opgave 2, 3 og 4

Igennem opgaverne har vi skulle lave tre forskellige implementationer af garbage collection. En mark-sweep med rekursive funktionskald, en mark-sweep uden rekursive funktions kald og en stop-and-copy med rekursive funktions kald. For at undgå at en fil indeholdte flere implementationer med nogle af dem udkommenteret, valgte vi at dele dem ud i flere filer. Opgave 2, 3 og 4 findes i "oo3/listmachineV1.c" samt side 8, opgave 5 i "oo3/listmachineV2.c" samt side 14 og opgave 6 i "oo3/listmachineV3.c" samt side **??** i appendix.

Vi har valgt at beskrive vores implementation af opgave 2, 3 og 4 i samme afsnit, da 3 og 4 går ud på at forbedre algoritmen, vi implementerede i opgave 2. Den version, som findes i "listmachineV1.c", har fået ændringerne fra opgave 3 og 4, så derfor vil der være nogle kodeelementer, som vi først forklarer i 2.2.2 og 2.2.3.

Vores implementation of mark-sweep algoritmen består af tre dele; en "mark" funktion, en "markPhase" funktion og en "sweepPhase" funktion. Algoritmen er beskrevet på side 180-181 i "Programming Language Concepts".

2.2.1 Opgave 2

Implementationen af den simple, rekursive "mark-sweep" algoritme har følgende steps:

Step A Lav step B for alle heap-referencer i stakken, som er hvide

Step B Farv heap-referencen sort og udfør step B på alle heap-referencer fra heap-referencen, som er hvide.

Step C Løb heap igennem for at "rengøre" den.

- **Step C.1** Hvis en block er hvid, mail den blå og add den til freelist.
- **Step C.2** Hvis en block er sort, mal den hvid.

Step C.1 For at adde et ord til "freelist", opretter vi en temporary pointer til addressen i heapen, hvor headeren på blokken ligger. Denne temporary pointer anvender vi så som en blok, hvor vi sætter det først ord (efter headeren) til at pege på freelist. Herefter sætter vi freelist til at pege på addressen af headeren af ordet.

2.2.2 Opgave 3

For at mindske risikoen for, at der opstår mange små blokke i freelisten, kan man slå hvide blokke, som ligger ved siden af hinanden, sammen. Dette gør vi ved at tjekke, om den næste blok er hvid og stadig ligger inden for heapen. Hvis den gør, øger vi længden på den hvide blok, vi arbejder med, sætter headeren på den næste blok til at være en "junk-value" og bruger "mkheader" til at lave en ny blok med den nye længde.

2.2.3 Opgave 4

For at optimere ændringen fra 2.2.2 endnu mere, kan man udnytte muligheden for at slå alle hvide blokke, som ligger ved siden af hinanden, sammen. Dette gør vi ved at gøre det samme, som i opgave 3, men i denne

version itererer vi bare videre, indtil vi støder på en næste blok, som enten ikke er hvid eller som ligger uden for heapen. Når dette sker, laver vi en ny blok på samme måde som i opgave3 (med mkheader) på længden af den første hvide blok, vi arbejdede med samt alle de hvide blokke, som ligger ved siden af den.

2.3 Opgave 5

For at implementere den ikke-rekursive version af Mark and Sweep[1], har vi fjernet vores "mark" funktion og udvidet "markPhase" funktionen. "sweepPhase" er den samme som i 2.2.

Vores kode kan findes i "oo3/listmachineV2.c" samt på side 14 i appendix.

Step A Mal alle blokke, som kan nås fra stakken, grå.

Step B Løb heap igennem og marker alle grå blokke sort.

Step B.1 Hvis en grå blok pegede på en hvid blok, males den hvide blok grå. Hvis der er grå blokke i heap, gentag step B, ellers gå til step C.

Step C Sweep på samme måde som i opgave 2.

2.4 Opgave 6

Vi har implementeret algoritmen til opgave 6 (two-space stop and copy) som en række steps. Algoritmen er forklaret i "Programming Language Concepts"[2]. Vores implementation kan findes i "oo3/listmachineV3.c" samt på side **??** i appendix.

Vi har delt algoritmen op i følgende steps:

Step A Peg freelisten på starten af heapTo.

Step B Iteration over stacken for at finde heap referencer.

Step B.1 Undersøg om blokken er flyttet allerede.

Step B.2 Lav en ny blok og flyt freelist pointeren.

Step B.3 Kør blokken igennem og kopier ord fra blokken.

Step B.4 Returner pointer til den nye blok i heapTo.

Step C Iteration over heapTo for at finde eventuelle referencer til heapFrom.

Step D Ombytning af heapFrom og heapTo.

De to steps er delt ud på to funktioner, som foreslået i opgavebeskrivelsen. "void copyFromTo(int s[], int sp)" og "word* copy(word* oldBlock)". Desuden har vi ændret i "allocate" funktionen, som foreslået i "Programming Language Concepts" og "initheap" funktionen, hvilket vi var nødt til pga. at vi nu har to heaps. Desuden har vi ændret "inHeap" funktionen til to funktioner - "inFromHeap" og "inToHeap".

Vi har valgt kun at beskrive step B, B.1, B.3 og C i detaljer, da vi føler, at de andre er beskrevet grundigt nok i listen over steps.

2.4.1 Step B

Step B er algoritmens mest komplekse step. Derfor har vi delt det op i fire mindre steps.

Step B løber stacken igennem og undersøger, om den indeholder nogen heap referencer. Hvis der er, bliver de kopieret og stacken bliver opdateret med addresserne på de kopierede blokke.

[1] "Programming Language Concepts", side 192-193

[2] "Programming Language Concepts", s. 181-182

Step B.1 Hvis en blok allerede er kopieret, returnerer vi bare "to-heap" addressen på blokken.
Vi ved, at en blok er kopieret, hvis det holder sandt at "(oldBlock[1] != 0 && !IsInt(oldBlock[1]) && inToHeap((word*) oldBlock[1]))".
Vi bruger det første ord efter headeren i "from-heap" til at holde en "forwarding pointer" til blokken i "to-heap".

Step B.3 Step B.3 løber ordene i blokken fra "from-heap" igennem og kopierer dem til "to-heap".
Hvis et ord er en heap reference, bliver "copy()" kaldt rekursivt, så vi kan få referencen til den blokken i "to-heap".
Hvis det ikke er en heap reference, bliver indholdet af ordet bare kopieret.
Vi kontrollerer desuden om det første ord efter headeren er kopieret - hvis den er, så sætter vi "forwarding pointeren" i "from-heap"s "oldBlock[1]" til addressen på den nyligt kopierede blok.

Step C I step C løber vi heap'en igennem og ser, om der er blokke, som indeholder ord, der peger i "from-heap". Hvis der er, så opdaterer vi referencen ved at hente forwarding pointeren.

A Kode

A.1 Makefile

```
1  all: lm1 lm2 lm3
2
3  LIBS= -m32 -Wall
4  CC = gcc
5
6  lm1: listmachineV1.o
7          ${CC} -o $@ listmachineV1.o ${LIBS}
8
9  listmachineV1.o: listmachineV1.c
10         ${CC} -c listmachineV1.c ${LIBS}
11
12 lm2: listmachineV2.o
13         ${CC} -o $@ listmachineV2.o ${LIBS}
14
15 listmachineV2.o: listmachineV2.c
16         ${CC} -c listmachineV2.c ${LIBS}
17
18 lm3: listmachineV3.o
19         ${CC} -o $@ listmachineV3.o ${LIBS}
20
21 listmachineV3.o: listmachineV3.c
22         ${CC} -c listmachineV3.c ${LIBS}
23
24 clean:
25         rm -rf *.o lm1 lm2 lm3
```

A.2 listmachineV1.c

```
1  /* Recursively runs through a block to paint white blocks black. */
2  void mark(word* block){
3
4    // If the block is already colored...
5    if(Color(block[0]) != White)
6    {
7            // ... return to stop the function.
8      return;
9    }
10   /*Step B*/
11   // Paint the block black
12   block[0] = Paint(block[0], Black);
13
14   int i;
15   for(i = 1; i <= Length(block[0]); i++) // Go through every word in the block
16   {
17     if(!IsInt(block[i]) && block[i] != 0) // If word is not an integer and is not nil,
18                                           then mark the block the word points to
19           {
20       mark((word*)block[i]); // Mark a referenced block
21     }
22   }
23 }
24
25 /* Marks heap references in the stack */
26 void markPhase(int s[], int sp) {
27   printf("\nmarking ...\n");
28   /*Step A*/
29   int i;
30   for(i = 0; i < sp; i++)
31   {
32         if(!IsInt(s[i]) && (s[i]) != 0) /* If item on stack is not an integer and is
33                                            not nil, convert it to a word reference
34                                                  and mark it*/
35         {
36           mark((word*) s[i]);
37         }
38   }
39 }
40
41 /* Sweeps the heap and  */
42 void sweepPhase() {
43   printf("sweeping ...\n");
44
45   /*Step C*/
46   int i;
47   word w;
48
49   for(i = 0; i < HEAPSIZE; i += Length(w) + 1) // Increase i by the length of the
```

```
50                                                          previous block + 1.
51   {
52     w = heap[i]; // The word in the heap.
53         int extra_space;
54
55         switch(Color(w))
56         {
57           case White:
58             /*Step C.1*/
59             extra_space = 0;
60             word* next = &heap[i + Length(w) + 1]; // Get next word from heap.
61
62                 // While adjecent blocks are white, put them together.
63                 while(Color(*next) == White && next < afterHeap)
64                 // While the colour of the next block is white and
65                       the next block is still in the heap
66                 {
67                   // Increase length of free space
68                   extra_space += Length(*next) + 1;
69
70                   // Set block header to a junk value
71                   *next = Tag(9999);
72
73                   next = &heap[i + extra_space + Length(w) + 1];
74                 }
75
76                 if(extra_space > 0) // If there are more than one white
77                                           block in succession.
78                 {
79                   // Set first block to block length + extra length and paint blue
80                   heap[i] = mkheader(Tag(w), Length(w) + extra_space, Blue);
81                 }
82                 else
83                 {
84                   // Just paint blue
85                   heap[i] = Paint(w, Blue);
86                 }
87
88                 // Add word to freelist
89                 word* wo = (word*) &heap[i];
90         wo[1] = (int) freelist;
91         freelist = &wo[0];
92
93                 break;
94
95           case Black:
96             /*Step C.2*/
97             // Paint black blocks white
98             w = Paint(w, White);
99                 break;
100
```

```
101             case Blue:
102               // Ignore blue blocks
103                   break;
104
105             case Grey:
106               // Should not happen
107                   break;
108
109             default:
110               // Should not happen
111                   break;
112           }
113     }
114 }
```

A.3 listmachineV2.c

```
1  /* Marks heap references in the stack */
2  void markPhase(int s[], int sp) {
3  printf("\nmarking ...\n");
4
5  /* Step A */
6  int i;
7  for(i = 0; i < sp; i++)
8  {
9          if(!IsInt(s[i]) && (s[i]) != 0) /* If item on stack is not an integer
10                                             and is not nil, convert it to a word
11                                                     reference and mark it */
12         {
13                 word* block = ((word*) s[i]);
14                 block[0] = Paint(block[0], Grey);
15         }
16 }
17
18
19 /* Step B */
20 int goAgain = 1;
21 word* b;
22 int j;
23
24 while(goAgain)
25 {
26 goAgain = 0;
27 // To iterate over all the blocks in the heap we increment i by length of b + 1.
28 for(i = 0; i < HEAPSIZE; i += Length(b[0]) + 1)
29 {
30         b = (word*) &heap[i]; // Get address of the block in the heap
31         if(Color(b[0]) == Grey)
32         {
33                 b[0] = Paint(b[0], Black);
34                 for(j = 1; j <= Length(b[0]); j++) // Iterate over words in b
35                 {
36                         if(!IsInt(b[j]) && b[j] != 0)
37                         {
38                 // We need to get the word pointer to be able to colour the header
39                                 word* rblock = (word*) b[j];
40                                 if(Color(rblock[0]) == White)
41                                 {
42                                         rblock[0] = Paint(rblock[0], Grey);
43                 // Every time we paint a block grey, we need to check recheck heap for grey
44                                         goAgain = 1;  blocks.
45                                 }
46                         }
47                 }
48         }
49 }
```

```
50 }
51 }
52
53 /* Sweeps the heap and  */
54 void sweepPhase() {
55 printf("sweeping ...\n");
56
57 /* Step C */
58 int i;
59 word w;
60  // Increase i by the length of the previous block + 1.
61 for(i = 0; i < HEAPSIZE; i += Length(w) + 1)
62 {
63         w = heap[i]; // The word in the heap.
64         int extra_space;
65
66         switch(Color(w))
67         {
68         case White:
69                 extra_space = 0;
70                 word* next = &heap[i + Length(w) + 1]; // Get next word from heap.
71
72                 // While adjecent blocks are white, put them together.
73 // While the colour of the next block is white and the next block is still in the heap
74                 while(Color(*next) == White && next < afterHeap)
75                 {
76                         // Increase length of free space
77                         extra_space += Length(*next) + 1;
78
79                         // Set block header to a junk value
80                         *next = Tag(9999);
81
82                         next = &heap[i + extra_space + Length(w) + 1];
83                 }
84
85                 if(extra_space > 0) // If there are more than one white block in a row.
86                 {
87                         // Set first block to word length + extra length and paint blue
88                         heap[i] = mkheader(Tag(w), Length(w) + extra_space, Blue);
89                 }
90                 else
91                 {
92                         // Just paint blue
93                         heap[i] = Paint(w, Blue);
94                 }
95
96                 // Add word to freelist
97                 word* wo = (word*) &heap[i];
98                 wo[1] = (int) freelist;
99                 freelist = &wo[0];
100
```

```
101                     break;
102
103             case Black:
104                     // Paint black blocks white
105                     w = Paint(w, White);
106                     break;
107
108             case Blue:
109                     // Ignore blue blocks
110                     break;
111
112             case Grey:
113                     // Should not happen
114                     break;
115
116             default:
117                     // Should not happen
118                     break;
119             }
120     }
121     }
```

A.4 listmachineV3.c

```
1  int inToHeap(word* p) {
2    return heapTo <= p && p < afterTo;
3  }
4  int inFromHeap(word* p) {
5    return heapFrom <= p && p < afterFrom;
6  }
7
8  void initheap() {
9    heapFrom = (word*)malloc(sizeof(word)*HEAPSIZE);
10   heapTo = (word*)malloc(sizeof(word)*HEAPSIZE);
11   afterFrom = &heapFrom[HEAPSIZE];
12   afterTo = &heapTo[HEAPSIZE];
13   // Initially, entire heap is one block on the freelist:
14   heapFrom[0] = mkheader(0, HEAPSIZE-1, Blue);
15   heapFrom[1] = (word)0;
16   heapTo[0] = mkheader(0, HEAPSIZE-1, Blue);
17   heapTo[1] = (word)0;
18   freelist = &heapFrom[0];
19 }
20
21 // Copies a block and returns the new to-space address
22 word* copy(word* oldBlock)
23 {
24         /*Step B.1*/
25         // If block is already copied, return forwarding pointer
26         if(oldBlock[1] != 0 && !IsInt(oldBlock[1]) && inToHeap((word*)oldBlock[1]))
27         {
28                 return (word*) oldBlock[1];
29         }
30
31         /*Step B.2*/
32         word* toBlock = freelist; // Create new block at freelist pointer
33
34         int length = Length(oldBlock[0]);
35 // Increase freelist pointer by (block length + 1) such that it points at
36     the first free space in the "to-heap".
37         freelist += (length + 1);
38
39         /*Step B.3*/
40         int i;
41         for(i = 0; i <= length; i++)
42         {
43                 if(oldBlock[i] != 0 && !IsInt(oldBlock[i]) && i != 0) //If a heap ref.
44                 {
45 // Copy the referenced block from the "from-heap" to the "to-heap".
46                         word* p = copy((word *) oldBlock[i]);
47 // Update the pointer to the copied block in the "to-heap".
48                         toBlock[i] = (word) p;
49                 }
```

```
50                  else // If a normal block, just copy.
51                  {
52                          toBlock[i] = oldBlock[i];
53                  }
54
55                  if(i == 1) // If first word in block has been copied
56                  {
57  // Set first word in old block to be a reference to the newly copied block.
58                          oldBlock[1] = (word) &toBlock[0];
59                  }
60          }
61
62          /*Step B.4*/
63          return toBlock; // Return reference to this block.
64  }
65
66  void copyFromTo(int s[], int sp)
67  {
68          /*Step A*/
69          freelist = &heapTo[0]; // Move the freelist pointer to address 0 of the "to-heap"
70
71          /*Step B*/
72          int i;
73          for(i = 0; i < sp; i++)
74          {
75                  if(!IsInt(s[i]) && (s[i]) != 0)
76                  {
77                          word* block = ((word*) s[i]);
78                          s[i] = (int) copy(block); // Update reference of i in the stack
79                  }
80          }
81
82          /*Step C*/
83          word* b;
84          int j;
85          for(i = 0; i < HEAPSIZE; i += Length(b[0]) + 1)
86          {
87                  b = (word*) &heapTo[i]; // Gets the address of the block header
88                  for(j = 1; j <= Length(b[0]); j++)
89                  {
90                          if(!IsInt(b[j]) && b[j] != 0)
91                          {
92                                  word* rblock = (word*) b[j]; //
93                                  /* If word is a reference to the "from-heap",
94                                      then update reference to be address of block
95                                      in the "to-heap".*/
96                                  if(inFromHeap(rblock))
97                                  {
98                                          b[j] = rblock[1];
99                                  }
100                         }
```

```
101                     }
102             }
103
104             /*Step D*/
105             word* heapTemp = heapTo;
106             heapTo = heapFrom;
107             heapFrom = heapTemp;
108
109             word* afterTemp = afterTo;
110             afterTo = afterFrom;
111             afterFrom = afterTemp;
112     }
```