KØBENHAVNS UNIVERSITET

Assignment 2 - PLD

Mikkel Willén

3. marts 2023

Indhold

1 Stack allocation

a)

In this case each stack will have its own pointer and grows independently. A possible advantages is that it can reduce memory fragmentation, since each stack can allocate blocks of memory for its own purposes. I can also potentially improve memory access since each stack can be accessed more effectly with fewer cache misses. Since each stack can be scanned separately for live objects, it may simplify garbage collection.
A disadvantage can be, that since each stack needs its own stack pointer and boundary checks, it can increase memory overhead. When coding, it can be more complicated to debug, since we may have to inspect two stacks at a given point.

b)

I can be used to implement stack oriented languages that operate on one or more stacks using postfix or reverse polish notation. They may be hard to read and debug, but they can be more concise and efficient.
It could also be possible to implemnt special stack data structures, that support operations like getMin() in constant time. This could be more effective for certain problems, but might also be more complex or use more memory.
It makes it possible to implement multiple stacks in a single array, where stacks can start at endpoints or be used for divider elements. This can save space, and potentially avoid dynamic allocation.

c)

With an anbounded number of stacks, it is possible to implement a queue. Atleast 3 stacks would be needed, one to store the incoming elements, another stack to store the outgoing elements, and a third stack to transfer elements between the two other stacks.

2 Reference counting

a)

At the first point, both lists have a reference count of 1, since *p* and *q* points to the lists [7; 9] and [13] respectively.
At point 2 a new reference *tmp* is created and points to the same list as *p*. This is increases the reference count of [7; 9] to 2.
At point 3 we assign to *p* the value of *q*. This decreases the reference count of [7; 9] to 1, since it is now only referenced by *tmp*, while the reference count of [13] is increased to 2, since both *p* and *q* points to it.
At point 4 we assign to *q*, the value of *tmp*. This decreases the reference count of [13] to 1, since only *p* points to it now, while it increases the reference count of [7; 9] to 2, since both *tmp* and *q* points to it now.
At point 5 we exit the scope of the *let tmp* expression, which means *tmp* goes out of scope and its value is discarded. This decreases the reference count of [7 9] to 1, since it is now only referenced by *q*, and both lists have a reference count of 1.

b)

The total number of reference modifications are 6, since there are 1 increment and 1 decrement at both points 3 and 4, while there are 1 increment at point 2 and 1 decrement at point 5.

3 Mark-Sweep collection

If objects have different sizes, then the mark-sweep procedure can cause fragmentation and waste of memory. If eg as small object is freed and then a larger object is requested, the allocator wont be able to use the freed space.
This could be fixed, by using a compacting garbage collector, that moves all the live objects together to eliminate the gaps. This approach how other drawbacks though such as increased overhead and complexity.

4 Parameter passing mechanisms

a)

Early computation: In this case the address of $A[i]$ is computed before g executes. $i = 0$ and $A[i] = A[0]$ When g executes it changes $i = 1$ and $j = A[0] = 2$.
Late computation: In this case the address of $A[i]$ is computed when g returns. $i = 0$ and $A[i] = A[0]$. When g executes it changes $i = 1$ and $j = A[1] = 2$.

b)

```
void g(int a, int b) {
    a = a + b;
    b = a - b;
}

void f() {
    int x = 1;
    int y = 2;
    g(x, x);
    System.out.println("x = " + x + ", y = " + y);
}
```

If g writes back a first and b second, x will be 3 and y will be 2.
If g writes back b first and a second, x will be -1 and y will be 2.

5 Scope rules

```
int bingo;
int dingo() {
    return bingo()
}
for (int bingo = 0; bingo <= 17; bingo++) {
    int x;
    x = dingo();
}
```

6 Program transformations

a)

```
h 8 =
    f b 8 8
f b 8 8 =
    let h y = b 8 y
    h 8 =
        b 8 8
b 8 8 =
    let k x =
        3 * 8
    8 + (k 8) =
        8 + 3 * 8
h 8 = 32
```

b)

```
let f' () (g x z) =
                        h'' (g x) z
let h'' (g x)    y =
                        g x y
let b' ()      v w =
                        v + (k' (w) x)
let k' (w)       x =
                        3 * w
let h' ()        u =
                        f' () b' u u
```

c)

```
eval (F ()) (g x z) =
                        h' g x z
eval (H' (g x))    y =
                        g x y
eval (B ())    (u w) =
                        v + (k'(x) w)
eval (K w)         x =
                        3 * w
eval (H ())        u =
                        f' () (b u u)
```

```
eval (F ()) (g x z) =
                        eval (H' (g x)) z
eval (H' (g x))    y =
                        g x y
eval (B ())    (u w) =
                        v + (eval (K w) x)
eval (K w)         x =
                        3 * w
eval (H ())        u =
                        eval (F ()) (b u u)
```