Contents

1 While reading: "Lisp Outside the Box" by Nick Levine 1

2 Memory 1

2.1 Which questions we want to answer: 1

2.2 Garbage Collector 1

2.3 When does allocation *almost* always occur in Lisp? And what is the exception? 2

2.3.1 What's the case for SBCL? 2

2.3.2 (SBCL) Why tagged pointers? 4

2.4 What does *consing* mean? 5

2.5 How long are memeory references valid? 6

2.6 How to recycle memory? 6

2.7 The Stack 6

2.7.1 What is the Stack used for in Lisp? 6

2.7.2 Short history 6

2.7.3 How can we tell the compiler, a variable is not accessible once a functions exits? 6

1 While reading: "Lisp Outside the Box" by Nick Levine

`https://www.nicklevine.org/lisp-book/`

2 Memory

2.1 Which questions we want to answer:

- how to request to allocate memory?
- how long are memory references valid?
- how to recycle memory when you done with it?

2.2 Garbage Collector

- Every Lisp implementation has a GC, no exception
- The author calls the garbace collector, GC, a build-in library

- Pretty much everything said about the Common Lisp GC, in this chapter of the book, applies to all the Lisp GCs, and is also relevant to other language GCs as well

2.3 When does allocation *almost* always occur in Lisp? And what is the exception?

Whenever something new gets created, that is not `#'EQ` to anything that is there already!

```
(cons 1 2)
(make-instance 'animal)
(1+ most-positive-fixnum) ;; why? see below
(read-line)
```

The *notable* Exception:

- *in most implementations* allocation doesn't occur when performing any arithmetic involving **fixnums**, those are integers betwee `most-negative-fixnum` and `most-positive-fixnum`, and so long the final result are all within this range, you won't have allocated

2.3.1 What's the case for SBCL?

Before consulting the SBCL Documentation the constants are:

```
;; tested on a 64bit machine, with 64bit SBCL version "1.4.1"
(= most-positive-fixnum 4611686018427387903 (1- (expt 2 62))) ;; ==> T
(= most-negative-fixnum -4611686018427387904 (- (expt 2 62))) ;; ==> T
```

So the range spans 2^{63} values! (2^{63}-1 + 1, because we count 0), the question then is if we have 64bit registers:

Why is it 63bits rather than 64!?

- According to the implementation, atleast one bit, as in this case, is reserved for the **n-fixnum-tag-bits**, which is used to determine whether it is a fixnum or not!

Then, why do we take a bit, to test if something is a fixnum?

- Ok, so this is really cool. In 64bit SBCL, every lisp object has type information, this information at least for some instances, is already part of the pointer to them.

For a fixnum a pointer ending (the least significant bit) with a '0' bit reads the other 63bits and interprets them as an immediate fixnum value!!!

If the least significant bit is not a '0' then the first 4 bits are interpreted as the type! And the rest of the pointer bits are the address to a dword in the lisp heap!

Why can we take 4bits of the address and use them for type information?

- Like aren't we sacrificing a lot of addresses this way? This is fixed by having data be double word alligned in memory. In sbcl a double word is 2x64=128bits or 16 bytes. So the address points to those dword aligned 'pages' and, not to the individual 16 bytes within, so the least significant 4bits (that now can't point to the bytes) are used to store the type information for this dword lisp-data!

  This concept is known as **Tagged pointers** !!!

A cool example of the immediate pointer=fixnum concept

- **TODO** I think it is worth trying to understand the assembly of this microexample!

```
;; Fixnums are immediately stored in the pointer information

(defun foo (x)
  (+ x 7))

(disassemble 'foo)
;; ==> ; disassembly for FOO:
;; (...)
;; The immediate value 14 is in binary #b1110, because 7 is in binary #b111, the 0 is
;; the fixnum immediate pointer tag information!!
;; TODO: I think it is worth trying to understand the assembly of this microexample!
; 39:       BF0E000000       MOV EDI, 14                      ; no-arg-parsing entry p
; 3E:       488BD1           MOV RDX, RCX
; 41:       41BBF004B021     MOV R11D, #x21B004F0             ; GENERIC-+
; 47:       41FFD3           CALL R11
; 4A:       488BE5           MOV RSP, RBP
; 4D:       F8               CLC
; 4E:       5D               POP RBP
; 4F:       C3               RET
```

Another Tagged Pointer Example

```
(setf *x* (1+ most-positive-fixnum)
      *y* (1+ most-positive-fixnum))

(eq *x* *y*) ;; ==> NIL

;; But, use a value that is within the fixnum range:
(setf *x* 20
      *y* 20)

(eq *x* *y*) ;; ==> T

;; Becaseu EQ compares pointers, and fixnums are immediately encoded
;; in the pointer, so in effect "there is no pointer" !

;; the function (sb-kernel:get-lisp-obj-address 123) returns the object a

(sb-kernel:get-lisp-obj-address 123) ;; ==> 246  ...?
;; because the binary representation of 123 is:
(format nil "~b" 123) ==> "1111011"
;; an thus adding the lowtag '0':"1111011" + '0' => 11110110
#b11110110 ==> 246 ;; !!!
```

Since this is a rather interesting topic lets detour a little bit further into it along the SBCL path.

2.3.2 (SBCL) Why tagged pointers?

First of all because Common Lisp is dynamically typed, the language implementation must distinguish between different types (dynamically = at runtime). A solution to this is boxing.

What is Boxing?

- Boxing means grouping a value with an overhead with type and optionally size information, and referring to it through a pointer.

What is unboxing then?

- The inverse of the above insofar as we get the value that is boxed directly. Those values usually the primitive machine types (int, float, byte), in other language also referred to as: primitive types (java), value types (NET, as opposed to reference types) or just "unboxed" (in haskell)

What are the disadvantages of Boxing?

1. adds overhead to the values
2. Must be allocated
3. so it uses up heap space
4. and thus is garbage collected (or destroyed/freed), if necessary
5. To access the value, two lookups are needed: (1) getting the pointer address, (2) using the pointer address to look up the datum on the lisp heap

Can we do it without the disadvantags?

- Having the cake and eating it too can indeed work, by using the immediate tagged pointer representation. Just like fixnums in SBCL are represented by a 64bit word, where the least significant bit is 0 and the folloing 63bit represent the fixnum. So they're not even pointers (!) but immediate values.
- **This way:**
  - the first '0' bit is all the overhead we need, no allocation, heap space wasted, gc need, and a single lookup of the pointer itself is all we need to get at the value.
- **The trade off is:**
  - the value must fit into a 64bit word and needs to relinquish some of its bits for the tag (called lowtag in sbcl). These lowtag bits implicate what kind of immediate type representation is used (characters are also immediate in sbcl), and can still use boxing when necessary by having some of the lowtag bits indicate that the pointer truly is to be used to lookup the actual boxed value on the heap.

2.4 What does *consing* mean?

In Lisp consing refers to allocation of memory in general. It does therefore not just mean the use of `CONS` or cons cells.

2.5 How long are memeory references valid?

For as long as your code can access it by any means whatsoever. Only once this is not the case, the memory becomes *potential garbage*, and will be freed by the GC somewhere in the future.

2.6 How to recycle memory?

Once the GC freed up some space because of the reasons stated under "How long are memory references valid", the memory automatically be reclaimed.

2.7 The Stack

2.7.1 What is the Stack used for in Lisp?

At the bottom of the Stack is where the program started, usually the main() function and its local environment. Now each function call adds an element on top of the stack which holds the local variables and the return address of the function call. Such that on success the stack is popped and the program continues working from the calling function, and its enviornment. Maybe adding some more data to it, from the call.

But the crucial thing is, that the local memory of a function call, only exist in the stack frame. They don't have to be heap allocated! This is typically true for all the (let ((variables in)) a function body).

Another bonus is that the top of the stack can be extended (push) very cheaply. Thus its easy to give the program as much temporary storage, as it needs.

2.7.2 Short history

TODO verify, exapand on this: The Stack and heap where originally invented for Lisp back in the 1950! Over the years some languages got by perfectly well without a heap, and with only temporary memory.

2.7.3 How can we tell the compiler, a variable is not accessible once a functions exits?

By declaring it to have `dynamic-extent`

```
(let* ((x (cons nil nil))
  (y x))
  (declare (dynamic-extent x))
;; (...)
)
```