Lilliput 2 - 4-byte headers

Header layout

```
32                13     6   2 0
 [CCCCCCCCCCCCCCCCCCCCHHVVVVAAASLL]

C - Compressed Class-Pointer
H - Compact Identity Hash-Code
V - Valhalla Bits
A - GC Age Bits
S - Self-Forwarding Bits
L - Lock-Bits
```

Class-Pointer Compression

The compressed class-pointer will use 19 bits in total. This would allow to address a maximum of ~500,000 classes. We expect that this is enough range for the vast majority of workloads. However, we also expect that some workloads would require more classes than this. Therefore we need a fallback mechanism. If the first N bits are all set, then we interpret the remaining 19-N bits as offset (in words) into the object, and find the uncompressed class-pointer there. A reasonble value of N is to be determined. When a class is loaded, we detect whether or not the encoding range is exhausted or not. If the encoding range is exhausted, then the field layouter will append a pointer-sized field at the end of superclass fields, but before the loaded-class fields (to keep the offset small). Assuming the fallback-marker uses 8 bits, then we have 11 bits remaining for the offset, which allows offsets up to 16KB into the object. The resulting compressed-class-pointer is then computed as `11111111oooooooooooo` and stored into the Klass structure as the prototype header. When class encoding range is not exhausted at class-loading, we need to be careful to not accidentally generate an encoded class that has the fallback-pattern.

(We can not use a simpler approach that uses all-zeroes and a fixed Klass* offset, because a superclass may be loaded normally, and later a subclass requires fallback-encoding, in which case that fixed location would already be used.)

Identity-Hash-Code

The identity hash-code will no longer be stored into the object header. Instead, we expand objects on-demand and only use 2 bits in the header to track the state of i-hashing for the object. The approach is outlined in detail in Compact Identity Hashcode.

Sliding-Forwarding

We need to preserve the compressed-class and i-hash bits **during** full-GCs (because the GC needs both information to determine object's sizes). We need to preserve the lock-bits only across full-GCs, we can temporarily use those bits during full-GC. That means we effectively have 11 bits for forward-pointer addressing during full-GC. Those bits will be used as follows:

Bit 0: Distinguish between forwarded and not forwarded. Serial GC also uses this bit for marking.

Bit 1: Select target block

Bits 2 - 10: Forwarding offset within target block, in words. That's 9 offset bits, allowing for blocks of 2^9 = 512 words, or 4KB.

The algorithm then works as follows (only describes notable differences to the sliding-forwarding as implemented in Lilliput 1):

1. Divide the heap into 4KB blocks.
2. Allocate a side-table with 2 pointer-sized entries per block. That means that this side-table will be maximum 1/256th of the heap-size.
3. During marking: If we encounter a locked object, then preserve its header as usual. For all live objects, clear the lowest header-bit to indicate 'not-forwarded'. In Serial-GC, install a 1, this doubles as 'marked', but that's ok. (Note that we don't need to use the lowest 2 header bits for this.)
4. During relocation, encode the forwarding-pointer as we do in Lilliput 1, and set the lowest bit to indicate 'forwarded'. In SerialGC, that bit will already be set, but that's ok because we're only using a single thread. In-fact, it's questionable if we even need the 'forwarded' bit at all in non-Serial GCs - the GC threads could simply forward all marked objects during the 'prepare' phase, and assume that all marked objects are forwarded during the 'adjust' and 'compact' phases.
5. Some GCs (G1 and maybe Serial) need a fallback mechanism, because the assumption that objects in each block can only be forwarded to one of two possible target blocks may be broken. We can use an all-ones bit pattern to indicate 'fallback', and if we encounter this, we would look-up the forwarding in a hash-table instead. This means that we cannot address the last word in each block. When we would need to forward an object to the last word of a block, then it would need to use the fallback table, also.

Parallel Full-GC

The Parallel Full GC is not compatible with Compact Identity Hash-Code, because it uses an forwarding-encoding-scheme that assumes a fixed object size. The Parallel Full GC has to be re-implemented to use a parallel mark-compact algorithm instead, similar to what G1 and Shenandoah are doing. In Parallel GC this is complicated by the fact that Parallel GC doesn't partition the heap into regions, which is the basis for work-division between GC threads. It can be solved by dividing the heap into regions in other ways, though. This needs to be figured out.