GHC Commentary: The Layout of Heap Objects

Terminology

- A *lifted* type is one that contains bottom (_|_), conversely an *unlifted* type does not contain _|_. For example, `Array` is lifted, but `ByteArray#` is unlifted.
- A *boxed* type is represented by a pointer to an object in the heap, an *unboxed* object is represented by a value. For example, `Int` is boxed, but `Int#` is unboxed.

The representation of _|_ must be a pointer: it is an object that when evaluated throws an exception or enters an infinite loop. Therefore, only boxed types may be lifted.

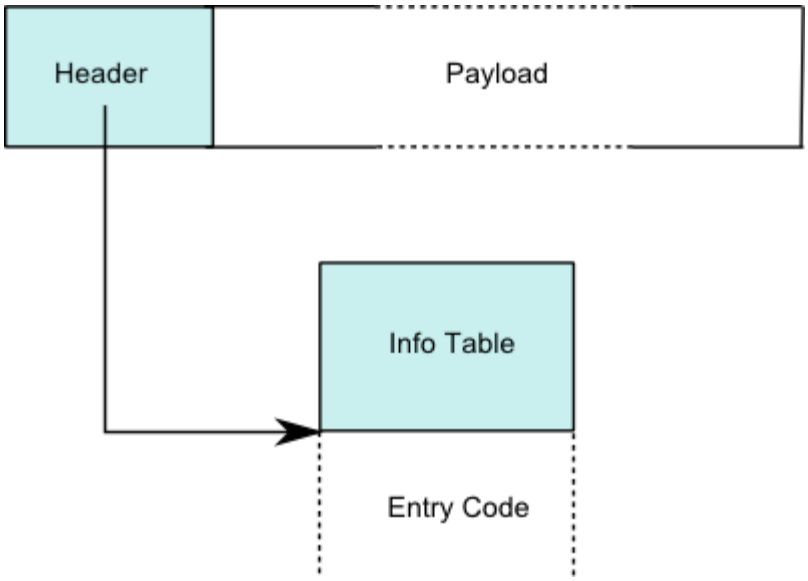
There are boxed unlifted types: eg. `ByteArray#`. If you have a value of type `ByteArray#`, it definitely points to a heap object with type `ARR_WORDS` (see below), rather than an unevaluated thunk.

Unboxed tuples `(#...#)` are both unlifted and unboxed. They are represented by multiple values passed in registers or on the stack, according to the return convention.

Unlifted types cannot currently be used to represent terminating functions: an unlifted type on the right of an arrow is implicitly lifted to include `_|_`.

Heap Objects

All heap objects have the same basic layout, embodied by the type `StgClosure` in Closures.h. The diagram below shows the layout of a heap object:



A heap object always begins with a *header*, defined by `StgHeader` in Closures.h:

```
typedef struct {
    const struct StgInfoTable* info;
#ifdef PROFILING
    StgProfHeader         prof;
#endif
} StgHeader;
```

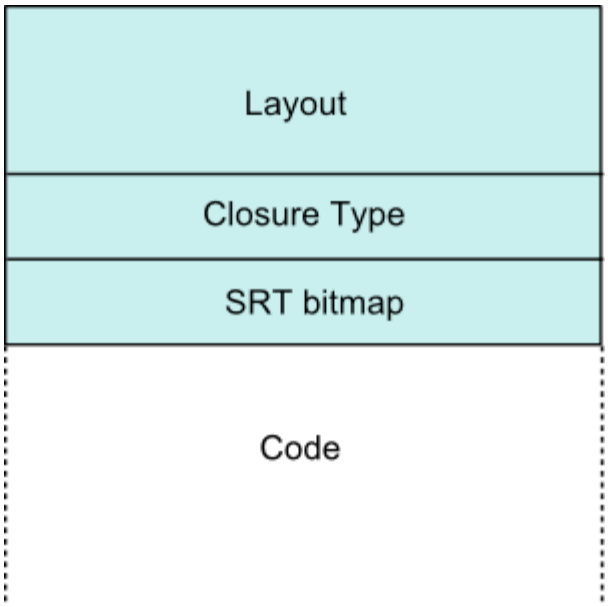
The most important part of the header is the *info pointer*, which points to the info table for the closure. In the default build, this is all the header contains, so a header is normally just one word. In other builds, the header may contain extra information: eg. in a profiling build it also contains information about who built the closure.

Most of the runtime is insensitive to the size of `StgHeader`; that is, we are careful not to hardcode the offset to the payload anywhere, instead we use C struct indexing or `sizeof(StgHeader)`. This makes it easy to extend `StgHeader` with new fields if we need to.

The compiler also needs to know the layout of heap objects, and the way this information is plumbed into the compiler from the C headers in the runtime is described here: Commentary/Compiler/CodeGen.

Info Tables

The info table contains all the information that the runtime needs to know about the closure. The layout of info tables is defined by `StgInfoTable` in InfoTables.h. The basic info table layout looks like this:



Where:

- The *closure type* is a constant describing the kind of closure this is (function, thunk, constructor etc.). All the closure types are defined in ClosureTypes.h, and many of them have corresponding C struct definitions in Closures.h.
- The *SRT bitmap* field is used to support garbage collection of CAFs.
- The *layout* field describes the layout of the payload for the garbage collector, and is described in more detail in Types of Payload Layout below.
- The *entry code* for the closure is usually the code that will *evaluate* the closure. There is one exception: for functions, the entry code will apply the function to the arguments given in registers or on the stack, according to the calling convention. The entry code assumes all the arguments are present - to apply a function to fewer arguments or to apply an unknown function, the generic apply functions must be used.

Some types of object add more fields to the end of the info table, notably functions, return addresses, and thunks.

Space in info tables is a premium: adding a word to the standard info table structure increases binary sizes by 5-10%.

`TABLES_NEXT_TO_CODE`

Note that the info table is followed immediately by the entry code, rather than the code being at the end of an indirect pointer. This both reduces the size of the info table and eliminates one indirection when jumping to the entry code.

GHC can generate code that uses the indirect pointer instead; the `TABLES_NEXT_TO_CODE` turns on the optimised layout. Generally `TABLES_NEXT_TO_CODE` is turned off when compiling unregisterised.

When `TABLES_NEXT_TO_CODE` is off, info tables get another field, `entry`, which points to the entry code. In a generated object file, each

symbol `X_info` representing an info table will have an associated symbol `X_entry` pointing to the entry code (in `TABLES_NEXT_TO_CODE`, the entry symbol is omitted to keep the size of symbol tables down).

Types of Payload Layout

The GC needs to know two things about the payload of a heap object: how many words it contains, and which of those words are pointers. There are two basic kinds of layout for the payload: *pointers-first* and *bitmap*. Which of these kinds of layout is being used is a property of the *closure type*, so the GC first checks the closure type to determine how to interpret the layout field of the info table.

Pointers-first layout

The payload consists of zero or more pointers followed by zero or more non-pointers. This is the most common layout: constructors, functions and thunks use this layout. The layout field contains two half-word-sized fields:

- Number of pointers
- Number of non-pointers

Bitmap layout

The payload consists of a mixture of pointers and non-pointers, described by a bitmap. There are two kinds of bitmap:

Small bitmaps. A small bitmap fits into a single word (the layout word of the info table), and looks like this:

Size (bits 0-4)	Bitmap (bits 5-31)

(for a 64-bit word size, the size is given 6 bits instead of 5).

The size field gives the size of the payload, and each bit of the bitmap is 1 if the corresponding word of payload contains a pointer to a live object.

The macros `MK_BITMAP`, `BITMAP_SIZE`, and `BITMAP_BITS` in InfoTables.h provide ways to conveniently operate on small bitmaps.

Large bitmaps. If the size of the stack frame is larger than the 27 words that a small bitmap can describe, then the fallback mechanism is the large bitmap. A large bitmap is a separate structure, containing a single word size and a multi-word bitmap: see `StgLargeBitmap` in InfoTables.h.

Dynamic vs. Static objects

Objects fall into two categories:

- *dynamic* objects reside in the heap, and may be moved by the garbage collector.
- *static* objects reside in the compiled object code. They are never

moved, because pointers to such objects are scattered through the object code, and only the linker knows where.

To find out whether a particular object is dynamic or static, use the HEAP_ALLOCED() macro, from rts/sm/HeapAlloc.h. This macro works by consulting a bitmap (or structured bitmap) that tells for each megablock of memory whether it is part of the dynamic heap or not.

Dynamic objects

Dynamic objects have a minimum size, because every object must be big enough to be overwritten by a forwarding pointer (Forwarding Pointers) during GC. The minimum size of the payload is given by `MIN_PAYLOAD_SIZE` in includes/rts/Constants.h.

Static objects

All static objects have closure types ending in `_STATIC`, eg. `CONSTR_STATIC` for static data constructors.

Static objects have an additional field, called the *static link field*. The static link field is used by the GC to link all the static objects in a list, and so that it can tell whether it has visited a particular static object or not - the GC needs to traverse all the static objects in order to garbage collect CAFs.

The static link field resides after the normal payload, so that the static variant of an object has compatible layout with the dynamic variant. To access the static link field of a closure, use the `STATIC_LINK()` macro from includes/rts/storage/ClosureMacros.h.

Types of object

Data Constructors

All data constructors have pointers-first layout:

Header	Pointers...	Non-pointers...

Data constructor closure types:

- `CONSTR`: a vanilla, dynamically allocated constructor
- `CONSTR_p_n`: a constructor whose layout is encoded in the closure type (eg. `CONSTR_1_0` has one pointer and zero non-pointers. Having these closure types speeds up GC a little for common layouts.
- `CONSTR_STATIC`: a statically allocated constructor.
- `CONSTR_NOCAF_STATIC`: TODO Needs documentation

The entry code for a constructor returns immediately to the topmost stack frame, because the data constructor is already in WHNF. The return convention may be vectored or non-vectored, depending on the type (see Commentary/Rts/HaskellExecution/CallingConvention).

Symbols related to a data constructor X:

- X_`con_info`: info table for a dynamic instance of X
- X_`static_info`: info table for a static instance of X

- X_`info`: the *wrapper* for X (a function, equivalent to the curried function `X` in Haskell, see Commentary/Compiler/EntityTypes).
- X_`closure`: static closure for X's wrapper

Function Closures

A function closure represents a Haskell function. For example:

```
f = \x -> let g = \y -> x + y
          in g x
```

Here, `f` would be represented by a static function closure (see below), and `g` a dynamic function closure. Every function in the Haskell program generates a new info table and entry code, and top-level functions additionally generate a static closure. All function closures have pointers-first layout:

Header	Pointers...	Non-pointers...

The payload of the function closure contains the free variables of the function: in the example above, a closure for `g` would have a payload containing a pointer to `x`.

Function closure types:

- `FUN`: a vanilla, dynamically allocated function
- `FUN_p_n`: same, specialised for layout (see constructors above)
- `FUN_STATIC`: a static (top-level) function closure

Symbols related to a function `f`:

- `f_info`: f's info table and code
- `f_closure`: f's static closure, if f is a top-level function. The static closure has no payload, because there are no free variables of a top-level function. It does have a static link field, though.

Thunks

A thunk represents an expression that is not obviously in head normal form. For example, consider the following top-level definitions:

```
range = between 1 10
f = \x -> let ys = take x range
          in sum ys
```

Here the right-hand sides of `range` and `ys` are both thunks; the former is static while the latter is dynamic.

Thunks have pointers-first layout:

Header	(empty)	Pointers...	Non-pointers...

As for function closures, the payload contains the free variables of the expression. A thunk differs from a function closure in that it can be updated.

There are several forms of thunk:

- `THUNK`, `THUNK_p_n`: vanilla, dynamically allocated thunks. Dynamic thunks are overwritten with normal indirections `IND` when evaluated.
- `THUNK_STATIC`: a static thunk is also known as a *constant applicative form*, or *CAF*. Static thunks are overwritten with static indirections (`IND_STATIC`).

The only label associated with a thunk is its info table:

- `f_info` is f's info table.

The empty padding is to allow thunk update code to overwrite the target of an indirection without clobbering any of the saved free variables. This means we can do thunk update without synchronization, which is a big deal.

Selector thunks

`THUNK_SELECTOR` is a (dynamically allocated) thunk whose entry code performs a simple selection operation from a data constructor drawn from a single-constructor type. For example, the thunk

```
x = case y of (a,b) -> a
```

is a selector thunk. A selector thunk is laid out like this:

Header	Selectee pointer

The `layout` word contains the byte offset of the desired word in the selectee. Note that this is different from all other thunks.

The garbage collector "peeks" at the selectee's tag (in its info table). If it is evaluated, then it goes ahead and does the selection, and then behaves just as if the selector thunk was an indirection to the selected field. If it is not evaluated, it treats the selector thunk like any other thunk of that shape.

This technique comes from the Phil Wadler paper Fixing some space leaks with a garbage collector, and later Christina von Dorrien who called it "Stingy Evaluation".

There is a fixed set of pre-compiled selector thunks built into the RTS, representing offsets from 0 to `MAX_SPEC_SELECTOR_THUNK`, see rts/StgStdThunks.cmm. The info tables are labelled `__sel_n_upd_info` where `n` is the offset. Non-updating versions are also built in, with info tables labelled `_sel_n_noupd_info`.

These thunks exist in order to prevent a space leak. For example, if y is a thunk that has been evaluated, and y is unreachable, but x is reachable, the risk is that x keeps both the a and b components of y live. By making the selector thunk a special case, we make it possible to reclaim the memory associated with b. (The situation is further complicated when selector thunks point to other selector thunks; the garbage collector sees all, knows all.)

Partial applications

Partial applications are tricky beasts.

A partial application, closure type `PAP`, represents a function applied to too few arguments. Partial applications are only built by the generic apply

functions in rts/Apply.cmm.

Header	Arity	No. of words	Function closure	Payload...

Where:

- *Arity* is the arity of the PAP. For example, a function with arity 3 applied to 1 argument would leave a PAP with arity 2.
- *No. of words* refers to the size of the payload in words.
- *Function closure* is the function to which the arguments are applied. Note that this is always a pointer to one of the `FUN` family, never a `PAP`. If a `PAP` is applied to more arguments to give a new `PAP`, the arguments from the original `PAP` are copied to the new one.
- The payload is the sequence of arguments already applied to this function. The pointerhood of these words are described by the function's bitmap (see `scavenge_PAP_payload()` in rts/sm/Scav.c for an example of traversing a PAP).

There is just one standard form of PAP. There is just one info table too, called `stg_PAP_info`. A PAP should never be entered, so its entry code causes a failure. PAPs are applied by the generic apply functions in `AutoApply.cmm`.

Generic application

An `AP` object is very similar to a `PAP`, and has identical layout:

Header	Arity	No. of words	Function closure	Payload...

The difference is that an `AP` is not necessarily in WHNF. It is a thunk that represents the application of the specified function to the given arguments.

The arity field is always zero (it wouldn't help to omit this field, because it is only half a word anyway).

`AP` closures are used mostly by the byte-code interpreter, so that it only needs a single form of thunk object. Interpreted thunks are always represented by the application of a `BCO` to its free variables.

Stack application

An `AP_STACK` is a special kind of object:

Header	Size	Closure	Payload...

It represents computation of a thunk that was suspended midway through evaluation. In order to continue the computation, copy the payload onto the stack (the payload was originally the stack of the suspended computation), and enter the closure.

Since the payload is a chunk of stack, the GC can use its normal stack-walking code to traverse it.

`AP_STACK` closures are built by `raiseAsync()` in rts/RaiseAsync.c when an asynchronous exception is raised. It's fairly typical for the end of an AP_STACK's payload to have another AP_STACK: you'll get one per update

frame.

Indirections

Indirection closures just point to other closures. They are introduced when a thunk is updated to point to its value. The entry code for all indirections simply enters the closure it points to.

The basic layout of an indirection is simply

Header	Target closure

There are several variants of indirection:

- `IND`: is the vanilla, dynamically-allocated indirection. It is removed by the garbage collector. An `IND` only exists in the youngest generation. The update code (`stg_upd_frame_info` and friends) checks whether the updatee is in the youngest generation before deciding which kind of indirection to use.
- `IND_STATIC`: a static indirection, arises when we update a `THUNK_STATIC`. A new `IND_STATIC` is placed on the mutable list when it is created (see `newCaf()` in rts/sm/Storage.c).

Byte-code objects

`BCO`

Black holes

`BLACKHOLE`, `CAF_BLACKHOLE`

Black holes represent thunks which are under evaluation by another thread (that thread is said to have claimed the thunk). Attempting to evaluate a black hole causes a thread to block until the thread who claimed the thunk either finishes evaluating the thunk or dies. You can read more about black holes in the paper 'Haskell on a Shared-Memory Multiprocessor'. Black holes have the same layout as indirections.

Header	Target closure

Sometimes black holes are just ordinary indirection. Check `stg_BLACKHOLE_info` for the final word: if the indirectee has no tag, then we assume that it is the TSO that has claimed the thunk; if the indirectee is tagged, then it is just a normal indirection. (EZY: I think this optimization is to avoid having to do two memory writes on thunk update; we don't bother updating the header, only the target.)

When eager blackholing is enabled, the black hole that is written is not a true black hole, but an eager black hole. True black holes are synchronized, and guarantee that only one black hole is claimed (this property is used to implement non-dupable unsafePerformIO). Eager black holes are not synchronized; eager black hole are converted into true black holes in ThreadPaused.c. Incidentally, this facility is also used to convert update frames to black holes; this is important for eliminating a space leak caused by the thunk under evaluation retaining too much data (overwriting it with a black hole frees up variable.)

Arrays

`ARR_WORDS`, `MUT_ARR_PTRS_CLEAN`, `MUT_ARR_PTRS_DIRTY`, `MUT_ARR_PTRS_FROZEN0`, `MUT_ARR_PTRS_FROZEN`

Non-pointer arrays are straightforward:

| Header | Bytes | Array payload |

Arrays with pointers are a little more complicated, they include a card table, which is used by the GC to know what segments of the array to traverse as roots (the card table is modified by the GC write barrier):

| Header | Ptrs | Size | Array payload + card table |

You can access the card table by using `mutArrPtrsCard(array, element index)`, which gives you the address of the card for that index.

MVars

`MVar`

MVars have a queue of the TSOs blocking on them along with their value:

| Header | Head of queue | Tail of queue | Value |

An MVar can be in several states. It can be empty (in which case the value is actually just a `stg_END_TSO_QUEUE_closure`) or it can be full. When it is full, the queue of TSOs are those waiting to put; when it is empty, the queue of TSOs are those waiting to read and take (with readers first). Like many mutable objects, MVars have CLEAN and DIRTY headers to avoid reapplying a write barrier when an MVar is already dirty.

Weak pointers

`Weak`

Stable Names

`STABLE_NAME`

Thread State Objects

Closure type `TSO` is a Thread State Object. It represents the complete state of a thread, including its stack.

TSOs are ordinary objects that live in the heap, so we can use the existing allocation and garbage collection machinery to manage them. This gives us one important benefit: the garbage collector can detect when a blocked thread is unreachable, and hence can never become runnable again. When this happens, we can notify the thread by sending it the `BlockedIndefinitely` exception.

GHC keeps divides stacks into stack chunks, with logic to handle stack underflow and overflow: http://hackage.haskell.org/trac/ghc/blog/stack-chunks

The TSO structure contains several fields. For full details see includes/rts/storage/TSO.h. Some of the more important fields are:

- *link*: field for linking TSOs together in a list. For example, the threads blocked on an `MVar` are kept in a queue threaded through the link

field of each TSO.
- *global_link*: links all TSOs together; the head of this list is `all_threads` in rts/Schedule.c.
- *what_next*: how to resume execution of this thread. The valid values are:
  - `ThreadRunGhc`: continue by returning to the top stack frame.
  - `ThreadInterpret`: continue by interpreting the BCO on top of the stack.
  - `ThreadKilled`: this thread has received an exception which was not caught.
  - `ThreadRelocated`: this thread ran out of stack and has been relocated to a larger TSO; the link field points to its new location.
  - `ThreadComplete`: this thread has finished and can be garbage collected when it is unreachable.
- *why_blocked*: for a blocked thread, indicates why the thread is blocked. See includes/rts/Constants.h for the list of possible values.
- *block_info*: for a blocked thread, gives more information about the reason for blockage, eg. when blocked on an

  MVar, block_info will point to the MVar.

- *bound*: pointer to a Task if this thread is bound
- *cap*: the Capability on which this thread resides.

STM objects

These object types are used by STM: `TVAR_WAIT_QUEUE`, `TVAR`, `TREC_CHUNK`, `TREC_HEADER`.

Forwarding Pointers

Forwarding pointers appear temporarily during garbage collection. A forwarding pointer points to the new location for an object that has been moved by the garbage collector. It is represented by replacing the info pointer for the closure with a pointer to the new location, with the least significant bit set to 1 to distinguish a forwarding pointer from an info pointer.

How to add new heap objects

There are two “levels” at which a new object can be added. The simplest way to add a new object is to simply define a custom new info table:

- includes/stg/MiscClosures.h: Declare your info table with `RTS_ENTRY`.
- rts/StgMiscClosures.cmm: Define the info table, also, provide entry points if they represent runnable code (though this is pretty uncommon if you’re not also adding a new closure type; most just error). To find out what `INFO_TABLE` and all its variants do, check the C-- parser at compiler/cmm/CmmParse.y The most important thing is to pick the correct closure type.

You also will need to define primops to manipulate your new object type, if you want to manipulate it from Haskell-land (and not have it be RTS-only). See wiki:Commentary/PrimOps for more details.

An object defined this way is completely unknown to the code generator, so it tends to be pretty inflexible. However, GHC defines lots of these, esp. of the

nullary kind; they're a convenient way of getting non-NULL sentinel values for important pieces of the runtime (e.g. `END_TSO_QUEUE`). A particularly complicated example of an object of this kind is `StgMVarTSOQueue`, which even has a definition in includes/rts/storage/Closures.h, but is simply has closure type `PRIM`. When you use a pre-existing closure type, you must follow their layout; for example, `PRIM` must have pointers first, non-pointers after, and you can't do anything fancy (like have attached variable size payloads, e.g. for arrays.)

To go the whole hog, you need to add a new closure type. This is considerably more involved:

- includes/rts/storage/ClosureTypes.h: Add the new closure type
- includes/rts/storage/ClosureMacros.h: Add a case to `closure_sizeW` for your struct. However, if your structure is really simple (i.e. can be completely described by the info table, an entry here is not necessary.
- includes/rts/storage/Closures.h: Define a struct for the closure, including the *header* as well as your payloads. Sometimes, you will have more than one info table per struct, e.g. if you have `DIRTY` and `CLEAN` variants. As a general rule, GC'd pointers should go before general fields.
- utils/deriveConstants/Main.hs: Tell the deriveConstants script to generate a bunch of accessors so you can manipulate the struct from C-- code.
- rts/ClosureFlags.c: Update the closure flags (see includes/rts/storage/InfoTables.h for info on what the flags mean "Closure flags") and the sanity check at the bottom of the file
- rts/Linker.c: Add your info tables so they are linked correctly
- rts/Printer.c: Print out a description of the closure. You need to handle all of the info tables you defined.
- rts/sm/Sanity.c: Update sanity checks so they know about your new closure type
- rts/sm/Scav.c, rts/sm/Evac.c, rts/sm/Compact.c: teach the garbage collector how to follow live pointers from your object.
- rts/LdvProfile.c, rts/RetainerProfile.c, rts/ProfHeap.c: teach the profiler how to recognize your closure

When in doubt, look at how an existing heap object similar to the one you are implementing is implemented. (Of course, if they're identical, why are you defining a new heap object...)

Last modified on Feb 27, 2016 5:12:46 PM

Attachments (2)