The STG runtime system (revised)

Simon Peyton Jones
Microsoft Research Ltd., Cambridge

Simon Marlow
Microsoft Research Ltd., Cambridge

Alastair Reid
Yale University

February 26, 1999

Contents

Part I
Introduction

1 Overview

This document describes the GHC/Hugs run-time system. It serves as a Glasgow/Yale/Nottingham "contract" about what the RTS does.

1.1 New features compared to GHC 3.xx

- The RTS supports mixed compiled/interpreted execution, so that a program can consist of a mixture of GHC-compiled and Hugs-interpreted code.
- The RTS supports concurrency by default. This has some costs (eg we can't do hardware stack checks) but reduces the number of different configurations we need to support.
- CAFs are only retained if they are reachable. Since they are referred to by implicit references buried in code, this means that the garbage collector must traverse the whole accessible code tree. This feature eliminates a whole class of painful space leaks.
- A running thread has only one stack, which contains a mixture of pointers and non-pointers. Section 9.9 describes how we find out which is which. (GHC has used two stacks for some while. Using one stack instead of two reduces register pressure, reduces the size of update frames, and eliminates "stack-stubbing" instructions.)
- The "return in registers" return convention has been dropped because it was complicated and doesn't work well on register-poor architectures. It has been partly replaced by unboxed tuples (Section 2.2) which allow the programmer to explicitly state where results should be returned in registers (or on the stack) instead of on the heap.
- Exceptions are supported by the RTS.
- Weak Pointers generalise the previously available Foreign Object interface.
- The garbage collector supports a number of new features, including a dynamically resizable heap and multiple generations with aging within a generation.

1.2 Wish list

Here's a list of things we'd like to support in the future.

- Interrupts, speculative computation.
- The SM could tune the size of the allocation arena, the number of generations, etc taking into account residency, GC rate and page fault rate.

- We could trigger a GC when all threads are blocked waiting for IO if the allocation arena (or some of the generations) are nearly full.

1.3 Configuration

Some of the above features are expensive or less portable, so we envision building a number of different configurations supporting different subsets of the above features.

You can make the following choices:

- Support for parallelism. There are three mutually-exclusive choices.

  `SEQUENTIAL` Support for concurrency but not for parallelism.

  `GRANSIM` Concurrency support and simulated parallelism.

  `PARALLEL` Concurrency support and real parallelism.

- `PROFILING` adds cost-centre profiling.
- `TICKY` gathers internal statistics (often known as "ticky-ticky" code).
- `DEBUG` does internal consistency checks.
- Persistence. (well, not yet).
- Which garbage collector to use. At the moment we only anticipate one, however.

1.4 Glossary

ToDo: *This terminology is not used consistently within the document. If you find something which disagrees with this terminology, fix the usage.*

In the type system, we have boxed and unboxed types.

- A *pointed* type is one that contains $\perp$. Variables with pointed types are the only things which can be lazily evaluated. In the STG machine, this means that they are the only things that can be *entered* or *updated* and it requires that they be boxed.
- An *unpointed* type is one that does not contain $\perp$. Variables with unpointed types are never delayed — they are always evaluated when they are constructed. In the STG machine, this means that they cannot be *entered* or *updated*. Unpointed objects may be boxed (like `Array#`) or unboxed (like `Int#`).

In the implementation, we have different kinds of objects:

- *boxed* objects are heap objects used by the evaluators
- *unboxed* objects are not heap allocated

- *stack* objects are allocated on the stack
- *closures* are objects which can be *entered*. They are always boxed and always have boxed types. They may be in WHNF or they may be unevaluated.
- A *thunk* is a (representation of) a value of a *pointed* type which is *not* in WHNF.
- A *value* is an object in WHNF. It can be pointed or unpointed.

At the hardware level, we have *word*s and *pointer*s.

- A *word* is (at least) 32 bits and can hold either a signed or an unsigned int.
- A *pointer* is (at least) 32 bits and big enough to hold a function pointer or a data pointer.

Occasionally, a field of a data structure must hold either a word or a pointer. In such circumstances, it is *not safe* to assume that words and pointers are the same size. **ToDo:***GHC currently makes words the same size as pointers to reduce complexity in the code generator/RTS. It would be useful to relax this restriction, and have eg. 32-bit Ints on a 64-bit machine.*

1.5 Subtle Dependencies

Some decisions have very subtle consequences which should be written down in case we want to change our minds.

- If the garbage collector is allowed to shrink the stack of a thread, we cannot omit the stack check in return continuations (Section 11.7).
- When we return to the scheduler, the top object on the stack is a closure. The scheduler restarts the thread by entering the closure.

  Section 10.4 discusses how Hugs returns an unboxed value to GHC and how GHC returns an unboxed value to Hugs.
- When we return to the scheduler, we need a few empty words on the stack to store a closure to reenter. Section 11.7 discusses who does the stack check and how much space they need.
- Heap objects never contain slop — this is required if we want to support mostly-copying garbage collection.

  This is a big problem when updating since the updatee is usually bigger than an indirection object. The fix is to overwrite the end of the updatee with "slop objects" (described in Section 9.8). This is hard to arrange if we do *lazy* blackholing (Section **??**) so we currently plan to blackhole an object when we push the update frame.
- Info tables for constructors contain enough information to decide which return convention they use. This allows Hugs to use a single piece of entry code for all constructors and insulates Hugs from changes in the choice of return convention.

2 Source Language

2.1 Explicit Allocation

As in the original STG machine, (almost) all heap allocation is caused by executing a let(rec). Since we no longer support the return in registers convention for data constructors, constructors now cause heap allocation and so they should be let-bound.

For example, we now write

```
> cons = \ x xs -> let r = (:) x xs in r
```

instead of

```
> cons = \ x xs -> (:) x xs
```

Note:*For historical reasons, GHC doesn't use this syntax — but it should.*

2.2 Unboxed tuples

Functions can take multiple arguments as easily as they can take one argument: there's no cost for adding another argument. But functions can only return one result: the cost of adding a second "result" is that the function must construct a tuple of "results" on the heap. The assymetry is rather galling and can make certain programming styles quite expensive. For example, consider a simple state transformer monad:

```
> type S a      = State -> (a,State)
> bindS m k s0 = case m s0 of { (a,s1) -> k a s1 }
> returnS a s  = (a,s)
> getS s       = (s,s)
> setS s _     = ((),s)
```

Here, every use of `returnS`, `getS` or `setS` constructs a new tuple in the heap which is instantly taken apart (and becomes garbage) by the case analysis in `bind`. Even a short state-transformer program will construct a lot of these temporary tuples.

Unboxed tuples provide a way for the programmer to indicate that they do not expect a tuple to be shared and that they do not expect it to be allocated in the heap. Syntactically, unboxed tuples are just like single constructor datatypes except for the annotation `unboxed`.

```
> data unboxed AAndState# a = AnS a State
> type S a = State -> AAndState# a
> bindS m k s0 = case m s0 of { AnS a s1 -> k a s1 }
> returnS a s  = AnS a s
> getS s       = AnS s s
> setS s _     = AnS () s
```

Semantically, unboxed tuples are just unlifted tuples and are subject to the same restrictions as other unpointed types.

Operationally, unboxed tuples are never built on the heap. When an unboxed tuple is returned, it is returned in multiple registers or multiple stack slots. At first sight, this seems a little strange but it's no different from passing double precision floats in two registers.

Notes:

- Unboxed tuples can only have one constructor and that thunks never have unboxed types — so we'll never try to update an unboxed constructor. The restriction to a single constructor is largely to avoid garbage collection complications.
- The core syntax does not allow variables to be bound to unboxed tuples (ie in default case alternatives or as function arguments) and does not allow unboxed tuples to be fields of other constructors. However, there's no harm in allowing it in the source syntax as a convenient, but easily removed, syntactic sugar.
- The compiler generates a closure of the form

  ```
  > c = \ x y z -> C x y z
  ```

  for every constructor (whether boxed or unboxed).

  This closure is normally used during desugaring to ensure that constructors are saturated and to apply any strictness annotations. They are also used when returning unboxed constructors to the machine code evaluator from the bytecode evaluator and when a heap check fails in a return continuation for an unboxed-tuple scrutinee.

2.3 STG Syntax

ToDo:*Insert STG syntax with appropriate changes.*

Part II

System Overview

This part is concerned with defining the external interfaces of the major components of the system; the next part is concerned with their inner workings.

The major components of the system are:

- The evaluators (Section 5) are responsible for evaluating heap objects. The system supports two evaluators: the machine code evaluator; and the bytecode evaluator.
- The scheduler (Section 4) acts as the coordinator for the whole system. It is responsible for switching between evaluators, switching between threads, garbage collection, communication between multiple processors, etc.
- The storage manager (Section 3) is responsible for allocating blocks of contiguous memory and for garbage collection.

- The loader (Section 7) is responsible for loading machine code and bytecode files from the file system and for resolving references between separately compiled modules.
- The compilers (Section 6) generate machine code and bytecode files which can be loaded by the loader.

ToDo:*Insert diagram showing all components underneath the scheduler and communicating only with the scheduler*

3 The Evaluators

There are two evaluators: a machine code evaluator and a bytecode evaluator. The evaluators task is to evaluate code within a thread until one of the following happens:

- heap overflow
- stack overflow
- it is preempted
- it blocks in one of the concurrency primitives
- it performs a safe ccall
- it needs to switch to the other evaluator.

The evaluators expect to find a closure on top of the thread's stack and terminate with a closure on top of the thread's stack.

3.1 Evaluation Model

Whilst the evaluators differ internally, they share a common evaluation model and many object representations.

3.1.1 Heap objects

The choice of heap and stack objects used by the evaluators is tightly bound to the evaluation model. This section provides an overview of the most important heap and stack objects; further details are given later.

All heap objects look like this:

Header	*Payload*

The headers vary between different kinds of object but they all start with a pointer to a pair consisting of an *info table* and some *entry code*. The info table is used both by the evaluators and by the storage manager and contains a `type` field which identifies which kind of heap object uses it and determines the interpretation of the payload and of the other fields of the info table. The entry code is some machine code used by the machine code evaluator to evaluate closures and raises an error for other kinds of objects.

The major kinds of heap object used are as follows. (For simplicity, this description omits certain optimisations and extra fields required by the garbage collector.)

Constructors are used to represent data constructors. Their payload consists of the fields of the constructor; the tag of the constructor is stored in the info table.

`CONSTR`	*Fields*

Primitive objects are used to represent objects with unlifted types which are too large to fit in a register (or stack slot) or for which sharing must be preserved. Primitive objects include large objects such as multiple precision integers and immutable arrays and mutable objects such as mutable arrays, mutable variables, MVar's, IVar's and foreign object pointers. Since primitive objects are not lifted, they cannot be entered. Their payload varies according to the kind of object.

Function closures are used to represent functions. Their payload (if any) consists of the free variables of the function.

`FUN`	*Free Variables*

Function closures are only generated by the machine code compiler.

Thunks are used to represent unevaluated expressions which will be updated with their result. Their payload (if any) consists of the free variables of the function. The entry code for a thunk starts by pushing an *update frame* onto the stack. When evaluation of the thunk completes, the update frame will cause the thunk to be overwritten again with an *indirection* to the result of the thunk, which is always a constructor or a partial application.

`THUNK`	*Free Variables*

Thunks are only generated by the machine code evaluator.

Byte-code Objects (`BCO`s) are generated by the bytecode compiler. In conjunction with *updatable applications* and *non-updatable applications* they are used to represent functions, unevaluated expressions and return addresses.

`BCO`	*Constant Pool*	*Bytecodes*

Non-updatable (Partial) Applications are used to represent the application of a function to an insufficient number of arguments. Their payload consists of the function and the arguments received so far.

`PAP`	*Function Closure*	*Arguments*

`PAP`s are used when a function is applied to too few arguments and by code generated by the lambda-lifting phase of the bytecode compiler.

Updatable Applications are used to represent the application of a function to a sufficient number of arguments. Their payload consists of the function and its arguments.

Updateable applications are like thunks: on entering an updateable application, the evaluators push an *update frame* onto the stack and overwrite the application with a *black hole*; when evaluation completes, the evaluators overwrite the application with an *indirection* to the result of the application.

`AP`	*Function Closure*	*Arguments*

`AP`s are only generated by the bytecode compiler.

Black holes are used to mark updateable closures which are currently being evaluated. "Black holing" an object cures a potential space leak and detects certain classes of infinite loops. More imporantly, black holes act as synchronisation objects between separate threads: if a second thread tries to enter an updateable closure which is already being evaluated, the second thread is added to a list of blocked threads and the thread is suspended.

When evaluation of the black-holed closure completes, the black hole is overwritten with an indirection to the result of the closure and any blocked threads are restored to the runnable queue.

Closures are overwritten by black-holes during a "lazy black-holing" phase which runs on each thread when it returns to the scheduler. **ToDo:***section describing lazy black-holing*.

`BLACKHOLE`	*Blocked threads*

ToDo:*In a single threaded system, it's trivial to detect infinite loops: reentering a BLACKHOLE is always an error. How easy is it in a multi-threaded system?*

Indirections are used to update an unevaluated closure with its (usually fully evaluated) result in situations where it isn't possible to perform an update in place. (In the current system, we always update with an indirection to avoid duplicating the result when doing an update in place.)

`IND`	*Closure*

Indirections needn't always point to a closure in WHNF. They can point to a chain of indirections which point to an evaluated closure.

Thread State Objects (TSOs) represent Haskell threads. Their payload consists of some per-thread information such as the Thread ID and the status of the thread (runnable, blocked etc.), and the thread's stack. See `TSO.h` for the full story. `TSO`s may be resized by the scheduler if its stack is too small or too large.

The thread stack grows downwards from higher to lower addresses.

`TSO`	*Thread info*	*Stack*

3.1.2 Stack objects

The stack contains a mixture of *pending arguments* and *stack objects*.

Pending arguments are arguments to curried functions which have not yet been incorporated into an activation frame. For example, when evaluating `let { g x y = x + y; f x = g{x} } in f{3,4}`, the evaluator pushes both arguments onto the stack and enters `f`. `f` only requires one argument so it leaves the second argument as a *pending argument*. The pending argument remains on the stack until `f` calls `g` which requires two arguments: the argument passed to it by `f` and the pending argument which was passed to `f`.

Unboxed pending arguments are always preceeded by a "tag" which says how large the argument is. This allows the garbage collector to locate pointers within the stack.

There are three kinds of stack object: return addresses, update frames and seq frames. All stack objects look like this

Header	*Payload*

As with heap objects, the header starts with a pointer to a pair consisting of an *info table* and some *entry code*.

Return addresses are used to cause selection and execution of case alternatives when a constructor is returned. Return addresses generated by the machine code compiler look like this:

`RET_XXX`	*Free Variables of the case alternatives*

The free variables are a mixture of pointers and non-pointers whose layout is described by a bitmask in the info table.

There are several kinds of `RET_XXX` return address - see Section 9.9.1 for the details.

Return addresses generated by the bytecode compiler look like this:

`BCO_RET`	*BCO*	*Free Variables of the case alternatives*

There is just one `BCO_RET` info pointer. We avoid needing different `BCO_RET`s for each stack layout by tagging unboxed free variables as though they were pending arguments.

Update frames are used to trigger updates. When an update frame is entered, it overwrites the updatee with an indirection to the result, restarts any threads blocked on the `BLACKHOLE` and returns to the stack object underneath the update frame.

`UPDATE_FRAME`	*Next Update Frame*	*Updatee*

Seq frames are used to implement the polymorphic `seq` primitive. They are a special kind of update frame, and are linked on the update frame list.

`SEQ_FRAME`	*Next Update Frame*

Stop frames are put on the bottom of each thread's stack, and act as sentinels for the update frame list (i.e. the last update frame points to the stop frame). Returning to a stop frame terminates the thread. Stop frames have no payload:

`SEQ_FRAME`

3.1.3 Case expressions

In the STG language, all evaluation is triggered by evaluating a case expression. When evaluating a case expression `case e of alts`, the evaluators pushes a return address onto the stack and evaluate the expression `e`. When `e` eventually reduces to a constructor, the return address on the stack is entered. The details of how the constructor is passed to the return address and how the appropriate case alternative is selected vary between evaluators.

Case expressions for unboxed data types are essentially the same: the case expression pushes a return address onto the stack before evaluating the scrutinee; when a function returns an unboxed value, it enters the return address on top of the stack.

3.1.4 Function applications

In the STG language, all function calls are tail calls. The arguments are pushed onto the stack and the function closure is entered. If any arguments are unboxed, they must be tagged as unboxed pending arguments. Entering a closure is just a special case of calling a function with no arguments.

3.1.5 Let expressions

In the STG language, almost all heap allocation is caused by let expressions. Filling in the contents of a set of mutually recursive heap objects is simple enough; the only difficulty is that once the heap space has been allocated, the thread must not return to the scheduler until after the objects are filled in.

3.1.6 Primitive operations

ToDo:

Most primops are simple, some aren't.

4 Scheduler

The Scheduler is the heart of the run-time system. A running program consists of a single running thread, and a list of runnable and blocked threads. A thread is represented by a *Thread Status Object* (TSO), which contains a few words status information and a stack. Except for the running thread, all threads have a closure on top of their stack; the scheduler restarts a thread by entering an evaluator which performs some reduction and returns to the scheduler.

4.1 The scheduler's main loop

The scheduler consists of a loop which chooses a runnable thread and invokes one of the evaluators which performs some reduction and returns.

The scheduler also takes care of system-wide issues such as heap overflow or communication with other processors (in the parallel system) and thread-specific problems such as stack overflow.

4.2 Creating a thread

Threads are created:

- When the scheduler is first invoked.
- When a message is received from another processor (I think). (Parallel system only.)
- When a C program calls some Haskell code.
- By `forkIO`, `takeMVar` and (maybe) other Concurrent Haskell primitives.

4.3 Restarting a thread

When the scheduler decides to run a thread, it has to decide which evaluator to use. It does this by looking at the type of the closure on top of the stack.

- `BCO` ⇒ bytecode evaluator
- `FUN` or `THUNK` ⇒ machine code evaluator
- `CONSTR` ⇒ machine code evaluator
- other ⇒ either evaluator.

The only surprise in the above is that the scheduler must enter the machine code evaluator if there's a constructor on top of the stack. This allows the bytecode evaluator to return a constructor to a machine code return address by pushing the constructor on top of the stack and returning to the scheduler. If the return address under the constructor is `HUGS_RET`, the entry code for `HUGS_RET` will rearrange the stack so that the return `BCO` is on top of the stack and return to the scheduler which will then call the bytecode evaluator. There is little point in trying to shorten this slightly indirect route since it is will happen very rarely if at all.

Note:*As an optimisation, we could store the choice of evaluator in the TSO status whenever we leave the evaluator. This is required for any thread, no matter what state it is in (blocked, stack overflow, etc). It isn't clear whether this would accomplish anything.*

4.4 Returning from a thread

The evaluators return to the scheduler when any of the following conditions arise:

- A heap check fails, and a garbage collection is required.
- A stack check fails, and the scheduler must either enlarge the current thread's stack, or flag an out of memory condition.
- A thread enters a closure built by the other evaluator. That is, when the bytecode interpreter enters a closure compiled by GHC or when the machine code evaluator enters a BCO.
- A thread returns to a return continuation built by the other evaluator. That is, when the machine code evaluator returns to a continuation built by Hugs or when the bytecode evaluator returns to a continuation built by GHC.
- The evaluator needs to perform a "safe" C call (Section 4.6).
- The thread becomes blocked. This happens when a thread requires the result of a computation currently being performed by another thread, or it reads a synchronisation variable that is currently empty (Section 9.7.2).
- The thread is preempted (the preemption mechanism is described in Section 4.5).
- The thread terminates.

Except when the thread terminates, the thread always terminates with a closure on the top of the stack. The mechanism used to trigger the world switch and the choice of closure left on top of the stack varies according to which world is being left and what is being returned.

4.4.1 Leaving the bytecode evaluator

Entering a machine code closure When it enters a closure, the bytecode evaluator performs a switch based on the type of closure (`AP`, `PAP`, `Ind`, etc). On entering a machine code closure, it returns to the scheduler with the closure on top of the stack.

Returning a constructor When it enters a constructor, the bytecode evaluator tests the return continuation on top of the stack. If it is a machine code continuation, it returns to the scheduler with the constructor on top of the stack.

Note: *This is why the scheduler must enter the machine code evaluator if it finds a constructor on top of the stack.*

Returning an unboxed value **Note:** *Hugs doesn't support unboxed values in source programs but they are used for a few complex primops.*

When it returns an unboxed value, the bytecode evaluator tests the return continuation on top of the stack. If it is a machine code continuation, it returns to the scheduler with the tagged unboxed value and a special closure on top of the stack. When the closure is entered (by the machine code evaluator), it returns the unboxed value on top of the stack to the return continuation under it.

The runtime library for GHC provides one of these closures for each unboxed type. Hugs cannot generate them itself since the entry code is really very tricky.

Heap/Stack overflow and preemption The bytecode evaluator tests for heap/stack overflow and preemption when entering a BCO and simply returns with the BCO on top of the stack.

4.4.2 Leaving the machine code evaluator

Entering a BCO The entry code for a BCO pushes the BCO onto the stack and returns to the scheduler.

Returning a constructor We avoid the need to test return addresses in the machine code evaluator by pushing a special return address on top of a pointer to the bytecode return continuation. Figure 5 shows the state of the stack just before evaluating the scrutinee.

```
| stack    |
+----------+
| bco      |--> BCO
+----------+
| HUGS_RET |
+----------+
```

Figure 1: Stack layout for evaluating a scrutinee

This return address rearranges the stack so that the bco pointer is above the constructor on the stack (as shown in Figure 2) and returns to the scheduler.

Returning an unboxed value We avoid the need to test return addresses in the machine code evaluator by pushing a special return address on top of a pointer to the bytecode return

```
| stack    |
+----------+
| con      |--> Constructor
+----------+
| bco      |--> BCO
+----------+
```

Figure 2: Stack layout for entering a Hugs return address

continuation. This return address rearranges the stack so that the bco pointer is above the tagged unboxed value (as shown in Figure 3) and returns to the scheduler.

```
| stack    |
+----------+
| 1#       |
+----------+
| I#       |
+----------+
| bco      |--> BCO
+----------+
```

Figure 3: Stack layout for returning an unboxed value

Heap/Stack overflow and preemption **ToDo:**

4.5 Preempting a thread

Strictly speaking, threads cannot be preempted — the scheduler merely sets a preemption request flag which the thread must arrange to test on a regular basis. When an evaluator finds that the preemption request flag is set, it pushes an appropriate closure onto the stack and returns to the scheduler.

In the bytecode interpreter, the flag is tested whenever we enter a closure. If the preemption flag is set, it leaves the closure on top of the stack and returns to the scheduler.

In the machine code evaluator, the flag is only tested when a heap or stack check fails. This is less expensive than testing the flag on entering every closure but runs the risk that a thread will enter an infinite loop which does not allocate any space. If the flag is set, the evaluator returns to the scheduler exactly as if a heap check had failed.

4.6 "Safe" and "unsafe" C calls

There are two ways of calling C:

"Unsafe" C calls are used if the programer is certain that the C function will not do anything dangerous. Unsafe C calls are faster but must be hand-checked by the programmer.

Dangerous things include:

- Call a system function such as `getchar` which might block indefinitely. This is dangerous because we don't want the entire runtime system to block just because one thread blocks.
- Call an RTS function which will block on the RTS access semaphore. This would lead to deadlock.
- Call a Haskell function. This is just a special case of calling an RTS function.

Unsafe C calls are performed by pushing the arguments onto the C stack and jumping to the C function's entry point. On exit, the result of the function is in a register which is returned to the Haskell code as an unboxed value.

"Safe" C calls are used if the programmer suspects that the thread may do something dangerous. Safe C calls are relatively slow but are less problematic.

Safe C calls are performed by pushing the arguments onto the Haskell stack, pushing a return continuation and returning a *C function descriptor* to the scheduler. The scheduler suspends the Haskell thread, spawns a new operating system thread which pops the arguments off the Haskell stack onto the C stack, calls the C function, pushes the function result onto the Haskell stack and informs the scheduler that the C function has completed and the Haskell thread is now runnable.

The bytecode evaluator will probably treat all C calls as being safe.

ToDo:*It might be good for the programmer to indicate how the program is unsafe. For example, if we distinguish between C functions which might call Haskell functions and those which might block, we could perform an unsafe call for blocking functions in a single-threaded system or, perhaps, in a multi-threaded system which only happens to have a single thread at the moment.*

5 The Storage Manager

The storage manager is responsible for managing the heap and all objects stored in it. It provides special support for lazy evaluation and for foreign function calls.

5.1 SM support for lazy evaluation

- Indirections are shorted out.
- Update frames pointing to unreachable objects are squeezed out.
- Adjacent update frames (for different closures) are compressed to a single update frame pointing to a single black hole.

5.2 SM support for foreign function calls

- Stable pointers allow other languages to access Haskell objects.
- Weak pointers and foreign objects provide finalisation support for Haskell references to external objects.

5.3 Misc

- If the stack contains a large amount of free space, the storage manager may shrink the stack. If it shrinks the stack, it guarantees never to leave less than `MIN_SIZE_SHRUNKEN_STACK` empty words on the stack when it does so.
- For efficiency reasons, very large objects (eg large arrays and TSOs) are not moved if possible.

6 The Compilers

Need to describe interface files, format of bytecode files, symbols defined by machine code files.

6.1 Interface Files

Here's an example - but I don't know the grammar - ADR.

```
_interface_ Main 1
_exports_
Main main ;
_declarations_
1 main _:_ IOBase.IO PrelBase.();;
```

6.2 Bytecode files

(All that matters here is what the loader sees.)

6.3 Machine code files

(Again, all that matters is what the loader sees.)

7 The Loader

In a batch mode system, we can statically link all the modules together. In an interactive system we need a loader which will explicitly load and unload individual modules (or, perhaps, blocks of mutually dependent modules) and resolve references between modules.

While many operating systems provide support for dynamic loading and will automatically resolve cross-module references for us, we generally cannot rely on being able to load mutually dependent modules.

A portable solution is to perform some of the linking ourselves. Each module should provide three global symbols:

- An initialisation routine. (Might also be used for finalisation.)
- A table of symbols it exports. Entries in this table consist of the symbol name and the address of the names value.
- A table of symbols it imports. Entries in this table consist of the symbol name and a list of references to that symbol.

On loading a group of modules, the loader adds the contents of the export lists to a symbol table and then fills in all the references in the import lists.

References in import lists are of two types:

References in machine code The most efficient approach is to patch the machine code directly, but this will be a lot of work, very painful to port and rather fragile.

Alternatively, the loader could store the value of each symbol in the import table for each module and the compiled code can access all external objects through the import table. This requires that the import table be writable but does not require that the machine code or info tables be writable.

References in data structures (SRTs and static data constructors) Either we patch the SRTs and constructors directly or we somehow use indirections through the symbol table. Patching the SRTs requires that we make them writable and prevents us from making effective use of virtual memories that use copy-on-write policies (this only makes a difference if we want to run several copies of the same program simultaneously). Using an indirection is possible but tricky.

Note: We could avoid patching machine code if all references to external references went through the SRT — then we just have one thing to patch. But the SRT always contains a pointer to the closure rather than the fast entry point (say), so we'd take a big performance hit for doing this.

Using the above scheme, all accesses to "external" objects involve a layer of indirection. To avoid this overhead, the machine code compiler might provide a way for the programmer to specify which modules will be statically linked and which will be dynamically linked — the idea being that statically linked code and data will be accessed directly.

Part III

Internal details

This part is concerned with the internal details of the components described in the previous part.

The major components of the system are:

- The scheduler (Section 9)
- The storage manager (Section 9)
- The evaluators
- The loader
- The compilers

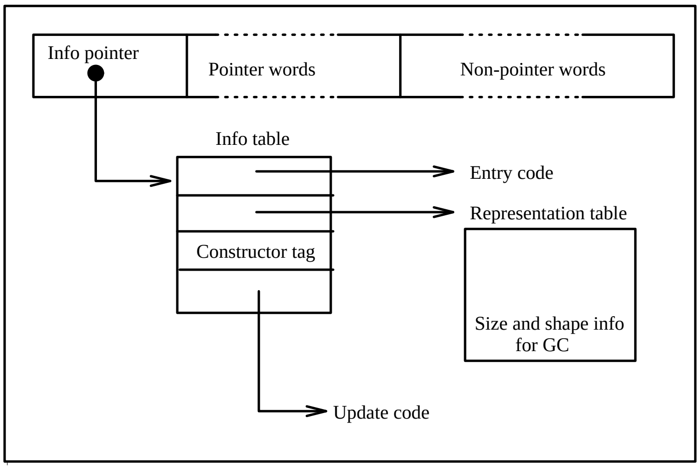
8 The Scheduler

ToDo:*Detailed description of scheduler*

Many heap objects contain fields allowing them to be inserted onto lists during evaluation or during garbage collection. The lists required by the evaluator and storage manager are as follows.

- 4 lists of threads: runnable threads, sleeping threads, threads waiting for timeout and threads waiting for I/O.
- The *mutables list* is a list of all objects in the old generation which might contain pointers into the new generation. Most of the objects on this list are indirections (Section 9.6.7) or “mutable.” (Section 9.7.2.)
- The *Foreign Object list* is a list of all foreign objects which have not yet been deallocated. (Section 9.7.3.)
- The *Spark pool* is a doubly(?) linked list of Spark objects maintained by the parallel system. (Section 9.8.)
- The *Blocked Fetch list* (or lists?). (Section 9.8.)
- For each thread, there is a list of all update frames on the stack. (Section 11.5.)
- The Stable Pointer Table is a table of pointers to objects which are known to the outside world and must be retained by the garbage collector even if they are not accessible from within the heap.

ToDo:*The links for these fields are usually inserted immediately after the fixed header except ...*



ToDo:*Fix this picture*

Figure 4: A closure

9 The Storage Manager

9.1 Misc Text looking for a home

A *value* may be:

- *Boxed*, i.e. represented indirectly by a pointer to a heap object (e.g. foreign objects, arrays); or
- *Unboxed*, i.e. represented directly by a bit-pattern in one or more registers (e.g. `Int#` and `Float#`).

All *pointed* values are *boxed*.

9.2 Heap Objects

Every *heap object* is a contiguous block of memory, consisting of a fixed-format *header* followed by zero or more *data words*.

The header consists of the following fields:

- A one-word *info pointer*, which points to the object's static *info table*.
- Zero or more *admin words* that support
  - Profiling (notably a *cost centre* word). **Note:***We could possibly omit the cost centre word from some administrative objects.*
  - Parallelism (e.g. GranSim keeps the object's global address here, though GUM keeps a separate hash table).

– Statistics (e.g. a word to track how many times a thunk is entered.).

  We add a Ticky word to the fixed-header part of closures. This is used to indicate if a closure has been updated but not yet entered. It is set when the closure is updated and cleared when subsequently entered. [1]

Most of the RTS is completely insensitive to the number of admin words. The total size of the fixed header is given by `sizeof(StgHeader)`.

9.3 Info Tables

An *info table* is a contiguous block of memory, laid out as follows:

Parallelism Info	variable
Profile Info	variable
Debug Info	variable
Static reference table	pointer word (optional)
Storage manager layout info	pointer word
Closure flags	8 bits
Closure type	8 bits
Constructor Tag / SRT length	16 bits
entry code	
⋮	

On a 64-bit machine the tag, type and flags fields will all be doubled in size, so the info table is a multiple of 64 bits.

An info table has the following contents (working backwards in memory addresses):

- The *entry code* for the closure. This code appears literally as the (large) last entry in the info table, immediately preceded by the rest of the info table. An *info pointer* always points to the first byte of the entry code.
- A 16-bit constructor tag / SRT length. For a constructor info table this field contains the tag of the constructor, in the range $0..n-1$ where n is the number of constructors in the datatype. Otherwise, it contains the number of entries in this closure's Static Reference Table (Section **??**).
- An 8-bit *closure type field*, which identifies what kind of closure the object is. The various types of closure are described in Section 9.4.
- an 8-bit flags field, which holds various flags pertaining to the closure type.
- A single pointer or word — the *storage manager info field*, contains auxiliary information describing the closure's precise layout, for the benefit of the garbage collector and the code that stuffs graph into packets for transmission over the network. There are three kinds of layout information:

[1] an "entries-after-update count." The commoning up of `CONST`, `CHARLIKE` and `INTLIKE` closures is turned off(?) if this is required. This has only been done for 2s collection.

- Standard layout information is for closures which place pointers before non-pointers in instances of the closure (this applies to most heap-based and static closures, but not activation records). The layout information for standard closures is
  * Number of pointer fields (16 bits).
  * Number of non-pointer fields (16 bits).
- Activation records don't have pointers before non-pointers, since stack-stubbing requires that the record has holes in it. The layout is therefore represented by a bitmap in which each '1' bit represents a non-pointer word. This kind of layout info is used for `RET_SMALL` and `RET_VEC_SMALL` closures.
- If an activation record is longer than 32 words, then the layout field contains a pointer to a bitmap record, consisting of a length field followed by two or more bitmap words. This layout information is used for `RET_BIG` and `RET_VEC_BIG` closures.
- Selector Thunks (Section 9.6.3) use the closure layout field to hold the selector index, since the layout is always known (the closure contains a single pointer field).

- A one-word *Static Reference Table* field. This field points to the static reference table for the closure (Section **??**), and is only present for the following closure types:
  - `FUN_*`
  - `THUNK_*`
  - `RET_*`
- *Profiling info*

  ToDo: *The profiling info is completely bogus. I've not deleted it from the document but I've commented it all out.*
- *Parallelism info* **ToDo:**
- *Debugging info* **ToDo:**

9.4 Kinds of Heap Object

Heap objects can be classified in several ways, but one useful one is this:

- *Static closures* occupy fixed, statically-allocated memory locations, with globally known addresses.
- *Dynamic closures* are individually allocated in the heap.
- *Stack closures* are closures allocated within a thread's stack (which is itself a heap object). Unlike other closures, there are never any pointers to stack closures. Stack closures are discussed in Section 9.9.

A second useful classification is this:

- *Executive objects*, such as thunks and data constructors, participate directly in a program's execution. They can be subdivided into three kinds of objects according to their type:
  - *Pointed objects*, represent values of a *pointed* type (¡.pointed types launchbury.¿) –i.e. a type that includes *bottom* such as `Int` or `Int# -> Int#`.
  - *Unpointed objects*, represent values of a *unpointed* type –i.e. a type that does not include *bottom* such as `Int#` or `Array#`.
  - *Activation frames*, represent "continuations". They are always stored on the stack and are never pointed to by heap objects or passed as arguments. **Note:***It's not clear if this will still be true once we support speculative evaluation.*
- *Administrative objects*, such as stack objects and thread state objects, do not represent values in the original program.

Only pointed objects can be entered. If an unpointed object is entered the program will usually terminate with a fatal error.

This section enumerates all the kinds of heap objects in the system. Each is identified by a distinct closure type field in its info table.

closure type	Section
Pointed	
CONSTR	9.6.2
CONSTR_p_n	9.6.2
CONSTR_STATIC	9.6.2
CONSTR_NOCAF_STATIC	9.6.2
FUN	9.6.1
FUN_p_n	9.6.1
FUN_STATIC	9.6.1
THUNK	9.6.3
THUNK_p_n	9.6.3
THUNK_STATIC	9.6.3
THUNK_SELECTOR	9.6.3
BCO	9.6.4
AP_UPD	9.6.6
PAP	9.6.5
IND	9.6.7
IND_OLDGEN	9.6.7
IND_PERM	9.6.7
IND_OLDGEN_PERM	9.6.7
IND_STATIC	9.6.7
CAF_UNENTERED	9.11
CAF_ENTERED	9.11
CAF_BLACKHOLE	9.11
Unpointed	
BLACKHOLE	9.6.8
BLACKHOLE_BQ	9.6.8
MVAR	9.7.2
ARR_WORDS	9.7.1
MUTARR_PTRS	9.7.2
MUTARR_PTRS_FROZEN	9.7.2
MUT_VAR	9.7.2
WEAK	9.7.4
FOREIGN	9.7.3
STABLE_NAME	9.7.5

Activation frames do not live (directly) on the heap — but they have a similar organisation.

closure type	Section
RET_SMALL	9.9.1
RET_VEC_SMALL	9.9.1
RET_BIG	9.9.1
RET_VEC_BIG	9.9.1
UPDATE_FRAME	9.9.1
CATCH_FRAME	9.9.1
SEQ_FRAME	9.9.1
STOP_FRAME	9.9.1

There are also a number of administrative objects. It is an error to enter one of these objects.

closure type	Section
TSO	9.9
SPARK_OBJECT	9.8
BLOCKED_FETCH	9.8
FETCHME	9.6.9

9.5 Predicates

The runtime system sometimes needs to be able to distinguish objects according to their properties: is the object updateable? is it in weak head normal form? etc. These questions can be answered by examining the closure type field of the object's info table.

We define the following predicates to detect families of related info types. They are mutually exclusive and exhaustive.

- `isCONSTR` is true for `CONSTR`s.
- `isFUN` is true for `FUN`s.
- `isTHUNK` is true for `THUNK`s.
- `isBCO` is true for `BCO`s.
- `isAP` is true for `AP`s.
- `isPAP` is true for `PAP`s.
- `isINDIRECTION` is true for indirection objects.
- `isBH` is true for black holes.
- `isFOREIGN_OBJECT` is true for foreign objects.
- `isARRAY` is true for array objects.
- `isMVAR` is true for `MVAR`s.
- `isIVAR` is true for `IVAR`s.

- `isFETCHME` is true for `FETCHME`s.
- `isSLOP` is true for slop objects.
- `isRET_ADDR` is true for return addresses.
- `isUPD_ADDR` is true for update frames.
- `isTSO` is true for `TSO`s.
- `isSTABLE_PTR_TABLE` is true for the stable pointer table.
- `isSPARK_OBJECT` is true for spark objects.
- `isBLOCKED_FETCH` is true for blocked fetch objects.
- `isINVALID_INFOTYPE` is true for all other info types.

The following predicates detect other interesting properties:

- `isPOINTED` is true if an object has a pointed type.

  If an object is pointed, the following predicates may be true (otherwise they are false). `isWHNF` and `isUPDATEABLE` are mutually exclusive.

  - `isWHNF` is true if the object is in Weak Head Normal Form. Note that unpointed objects are (arbitrarily) not considered to be in WHNF.

    `isWHNF` is true for `PAP`s, `CONSTR`s, `FUN`s and all `BCO`s.

    ToDo: *Need to distinguish between whnf BCOs and non-whnf BCOs in their closure type*
  - `isUPDATEABLE` is true if the object may be overwritten with an indirection object.

    `isUPDATEABLE` is true for `THUNK`s, `AP`s and `BH`s.

  It is possible for a pointed object to be neither updatable nor in WHNF. For example, indirections.

- `isUNPOINTED` is true if an object has an unpointed type. All such objects are boxed since only boxed objects have info pointers.

  It is true for `ARR_WORDS`, `ARR_PTRS`, `MUTVAR`, `MUTARR_PTRS`, `MUTARR_PTRS_FROZEN`, `FOREIGN` objects, `MVAR`s and `IVAR`s.

- `isACTIVATION_FRAME` is true for activation frames of all sorts.

  It is true for return addresses and update frames.

  - `isVECTORED_RETADDR` is true for vectored return addresses.
  - `isDIRECT_RETADDR` is true for direct return addresses.

- `isADMINISTRATIVE` is true for administrative objects: `TSO`s, the stable pointer table, spark objects and blocked fetches.

- `hasSRT` is true if the info table for the object contains an SRT pointer.

  `hasSRT` is true for `THUNK`s, `FUN`s, and `RET`s.

- `isSTATIC` is true for any statically allocated closure.
- `isMUTABLE` is true for objects with mutable pointer fields: `MUT_ARR`s, `MUTVAR`s, `MVAR`s and `IVAR`s.
- `isSparkable` is true if the object can (and should) be sparked. It is true of updateable objects which are not in WHNF with the exception of `THUNK_SELECTOR`s and black holes.

As a minor optimisation, we might use the top bits of the `INFO_TYPE` field to "cache" the answers to some of these predicates.

An indirection either points to HNF (post update); or is result of overwriting a FetchMe, in which case the thing fetched is either under evaluation (BLACKHOLE), or by now an HNF. Thus, indirections get NoSpark flag.

9.6 Closures (aka Pointed Objects)

An object can be entered iff it is a closure.

9.6.1 Function closures

Function closures represent lambda abstractions. For example, consider the top-level declaration:

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

Both `f` and `g` are represented by function closures. The closure for `f` is *static* while that for `g` is *dynamic*.

The layout of a function closure is as follows:

Fixed header	*Pointers*	*Non-pointers*

The data words (pointers and non-pointers) are the free variables of the function closure. The number of pointers and number of non-pointers are stored in `info->layout.ptrs` and `info->layout.nptrs` respecively.

There are several different sorts of function closure, distinguished by their closure type field:

- `FUN`: a vanilla, dynamically allocated on the heap.
- `FUN_`p`_`np: to speed up garbage collection a number of specialised forms of `FUN` are provided, for particular (p, np) pairs, where p is the number of pointers and np the number of non-pointers.
- `FUN_STATIC`. Top-level, static, function closures (such as `f` above) have a different layout than dynamic ones:

Fixed header	Static object link

Static function closures have no free variables. (However they may refer to other static closures; these references are recorded in the function closure's SRT.) They have one field that is not present in dynamic closures, the *static object link* field. This is used by the garbage collector in the same way that to-space is, to gather closures that have been determined to be live but that have not yet been scavenged.

Note:*Static function closures that have no static references, and hence a null SRT pointer, don't need the static object link field. We don't take advantage of this at the moment, but we could. See* `CONSTR_NOCAF_STATIC`*.*

Each lambda abstraction, f, in the STG program has its own private info table. The following labels are relevant:

- f`_info` is f's info table.
- f`_entry` is f's slow entry point (i.e. the entry code of its info table; so it will label the same byte as f`_info`).
- f`_fast_`k is f's fast entry point. k is the number of arguments f takes; encoding this number in the fast-entry label occasionally catches some nasty code-generation errors.

9.6.2 Data constructors

Data-constructor closures represent values constructed with algebraic data type constructors. The general layout of data constructors is the same as that for function closures. That is

Fixed header	Pointers	Non-pointers

There are several different sorts of constructor:

- `CONSTR`: a vanilla, dynamically allocated constructor.
- `CONSTR_`p`_`np: just like `FUN_`p`_`np.
- `CONSTR_INTLIKE`. A dynamically-allocated heap object that looks just like an `Int`. The garbage collector checks to see if it can common it up with one of a fixed set of static int-like closures, thus getting it out of the dynamic heap altogether.
- `CONSTR_CHARLIKE`: same deal, but for `Char`.
- `CONSTR_STATIC` is similar to `FUN_STATIC`, with the complication that the layout of the constructor must mimic that of a dynamic constructor, because a static constructor might be returned to some code that unpacks it. So its layout is like this:

Fixed header	Pointers	Non-pointers	Static object link

The static object link, at the end of the closure, serves the same purpose as that for `FUN_STATIC`. The pointers in the static constructor can point only to other static closures.

The static object link occurs last in the closure so that static constructors can store their data fields in exactly the same place as dynamic constructors.

- `CONSTR_NOCAF_STATIC`. A statically allocated data constructor that guarantees not to point (directly or indirectly) to any CAF (Section 9.11). This means it does not need a static object link field. Since we expect that there might be quite a lot of static constructors this optimisation makes sense. Furthermore, the `NOCAF` tag allows the compiler to indicate that no CAFs can be reached anywhere *even indirectly*.

For each data constructor *Con*, two info tables are generated:

- *Con*`_con_info` labels *Con*'s dynamic info table, shared by all dynamic instances of the constructor.
- *Con*`_static` labels *Con*'s static info table, shared by all static instances of the constructor.

Each constructor also has a *constructor function*, which is a curried function which builds an instance of the constructor. The constructor function has an info table labelled as `$Con$_info`, and entry code pointed to by `$Con$_entry`.

Nullary constructors are represented by a single static info table, which everyone points to. Thus for a nullary constructor we can omit the dynamic info table and the constructor function.

9.6.3 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.

The layout of a thunk is the same as that for a function closure. However, thunks must have a payload of at least `MIN_UPD_SIZE` words to allow it to be overwritten with a black hole and an indirection. The compiler may have to add extra non-pointer fields to satisfy this constraint.

Fixed header	*Pointers*	*Non-pointers*

The layout word in the info table contains the same information as for function closures; that is, number of pointers and number of non-pointers.

A thunk differs from a function closure in that it can be updated.

There are several forms of thunk:

- `THUNK` and `THUNK_`p`_`np: vanilla, dynamically allocated thunks. Dynamic thunks are overwritten with normal indirections (`IND`), or old generation indirections (`IND_OLDGEN`): see Section 9.6.7.
- `THUNK_STATIC`. A static thunk is also known as a *constant applicative form*, or *CAF*. Static thunks are overwritten with static indirections.

Fixed header	*Static object link*

- `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:

Fixed 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. [Implementation notes. Copying: only the evacuate routine needs to be special. Compacting: only the PRStart (marking) routine needs to be special.]

  There is a fixed set of pre-compiled selector thunks built into the RTS, representing offsets from 0 to `MAX_SPEC_SELECTOR_THUNK`. The info tables are labelled `sel_info_$n$` where n is the offset.

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

f`_info` is f's info table.

9.6.4 Byte-code objects

A Byte-Code Object (BCO) is a container for a a chunk of byte-code, which can be executed by Hugs. The byte-code represents a supercombinator in the program: when Hugs compiles a module, it performs lambda lifting and each resulting supercombinator becomes a byte-code object in the heap.

BCOs are not updateable; the bytecode compiler represents updatable thunks using a combination of `AP`s and `BCO`s.

The semantics of BCOs are described in Section 10.2. A BCO has the following structure:

Fixed Header	Layout	Offset	Size	Literals	Byte code

where:

- The entry code is a static code fragment/info table that returns to the scheduler to invoke Hugs (Section 4.4.2).
- *Layout* contains the number of pointer literals in the *Literals* field.
- *Offset* is the offset to the byte code from the start of the object.
- *Size* is the number of words of byte code in the object.
- *Literals* contains any pointer and non-pointer literals used in the byte-codes (including jump addresses), pointers first.
- *Byte code* contains *Size* words of non-pointer byte code.

9.6.5 Partial applications

A partial application (PAP) represents a function applied to too few arguments. It is only built as a result of updating after an argument-satisfaction check failure. A PAP has the following shape:

Fixed header	No of words of stack	Function closure	Stack chunk ...

The "Stack chunk" is a copy of the chunk of stack above the update frame; "No of words of stack" tells how many words it consists of. The function closure is (a pointer to) the closure for the function whose argument-satisfaction check failed.

In the normal case where a PAP is built as a result of an argument satisfaction check failure, the stack chunk will just contain "pending arguments", ie. pointers and tagged non-pointers. It may in fact also contain activation records, but not update frames, seq frames, or catch frames. The reason is the garbage collector uses the same code to scavenge a stack as it does to scavenge the payload of a PAP, but an update frame contains a link to the next update frame in the chain and this link would need to be relocated during garbage collection. Revertible black holes and asynchronous exceptions use the more general form of PAPs (see Section **??**).

There is just one standard form of PAP. There is just one info table too, called `PAP_info`. Its entry code simply copies the arg stack chunk back on top of the stack and enters the function closure. (It has to do a stack overflow test first.)

There is just one way to build a PAP: by calling `stg_update_PAP` with the function closure in register `R1` and the pending arguments on the stack. The `stg_update_PAP` function will build the PAP, perform the update, and return to the next activation record on the stack. If there are *no* pending arguments on the stack, then no PAP need be built: in this case `stg_update_PAP` just overwrites the updatee with an indirection to the function closure.

PAPs are also used to implement Hugs functions (where the arguments are free variables). PAPs generated by Hugs can be static so we need both `PAP` and `PAP_STATIC`.

9.6.6 `AP_UPD` objects

`AP_UPD` objects are used to represent thunks built by Hugs. The only distintion between an `AP_UPD` and a `PAP` is that an `AP_UPD` is updateable.

Fixed Header	*No of stack words*	*Function closure*	*Stack chunk*

The entry code pushes an update frame, copies the arg stack chunk on top of the stack, and enters the function closure. (It has to do a stack overflow test first.)

The "stack chunk" is a block of stack not containing update frames, seq frames or catch frames (just like a PAP). In the case of Hugs, the stack chunk will contain the free variables of the thunk, and the function closure is (a pointer to) the closure for the thunk. The argument stack may be empty if the thunk has no free variables.

Note:*Since* `AP_UPD`*s are updateable, the* `MIN_UPD_SIZE` *constraint applies here too.*

9.6.7 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.

There are several forms of indirection:

`IND` is the vanilla, dynamically-allocated indirection. It is removed by the garbage collector. It has the following shape:

Fixed header	*Target closure*

An `IND` only exists in the youngest generation. In older generations, we have `IND_OLDGEN`s. The update code (`Upd_frame_$n$_entry`) checks whether the updatee is in the youngest generation before deciding which kind of indirection to use.

`IND_OLDGEN` is the vanilla, dynamically-allocated indirection. It is removed by the garbage collector. It has the following shape:

Fixed header	*Target closure*	*Mutable link field*

It contains a *mutable link field* that is used to string together mutable objects in each old generation.

`IND_PERM` for lexical profiling, it is necessary to maintain cost centre information in an indirection, so "permanent indirections" are retained forever. Otherwise they are just like vanilla indirections. **Note:***If a permanent indirection points to another permanent indirection or a* `CONST` *closure, it is possible to elide the indirection since it will have no effect on the profiler.*

Note:*Do we still need* `IND` *in the profiling build, or do we just need* `IND` *but its behaviour changes when profiling is on?*

`IND_OLDGEN_PERM` Just like an `IND_OLDGEN`, but sticks around like an `IND_PERM`.

`IND_STATIC` is used for overwriting CAFs when they have been evaluated. Static indirections are not removed by the garbage collector; and are statically allocated outside the heap (and should stay there). Their static object link field is used just as for `FUN_STATIC` closures.

Fixed header	*Target closure*	*Static link field*

9.6.8 Black holes and blocking queues

Black hole closures are used to overwrite closures currently being evaluated. They inform the garbage collector that there are no live roots in the closure, thus removing a potential space leak.

Black holes also become synchronization points in the concurrent world. When a thread attempts to enter a blackhole, it must wait for the result of the computation, which is presumably in progress in another thread.

Note:*In a single-threaded system, entering a black hole indicates an infinite loop. In a concurrent system, entering a black hole indicates an infinite loop only if the hole is being entered by the same thread that originally entered the closure. It could also bring about a deadlock situation where several threads are waiting circularly on computations in progress.*

There are two types of black hole:

`BLACKHOLE` A straightforward blackhole just consists of an info pointer and some padding to allow updating with an `IND_OLDGEN` if necessary. This type of blackhole has no waiting threads.

Fixed header	*Padding*	*Padding*

If we're doing *eager blackholing* then a thunk's info pointer is overwritten with `BLACKHOLE_info` at the time of entry; hence the need for blackholes to be small, otherwise we'd be overwriting part of the thunk itself.

`BLACKHOLE_BQ` When a thread enters a `BLACKHOLE`, it is turned into a `BLACKHOLE_BQ` (blocking queue), which contains a linked list of blocked threads in addition to the info pointer.

Fixed header	*Blocked thread link*	*Mutable link field*

The *Blocked thread link* points to the TSO of the first thread waiting for the value of this thunk. All subsequent TSOs in the list are linked together using their `tso->link` field, ending in `END_TSO_QUEUE_closure`.

Because new threads can be added to the *Blocked thread link*, a blocking queue is *mutable*, so we need a mutable link field in order to chain it on to a mutable list for the generational garbage collector.

9.6.9 FetchMes

In the parallel systems, FetchMes are used to represent pointers into the global heap. When evaluated, the value they point to is read from the global heap.

ToDo:*Describe layout*

Because there may be offsets into these arrays, a primitive array cannot be handled as a FetchMe in the parallel system, but must be shipped in its entirety if its parent closure is shipped.

9.7 Unpointed Objects

A variable of unpointed type is always bound to a *value*, never to a *thunk*. For this reason, unpointed objects cannot be entered.

9.7.1 Immutable objects

`ARR_WORDS` is a variable-sized object consisting solely of non-pointers. It is used for arrays of all sorts of things (bytes, words, floats, doubles... it doesn't matter).

Strictly speaking, an `ARR_WORDS` could be mutable, but because it only contains non-pointers we don't need to track this fact.

Fixed Hdr	*No of non-pointers*	*Non-pointers...*

9.7.2 Mutable objects

Some of these objects are *mutable*; they represent objects which are explicitly mutated by Haskell code through the `ST` or `IO` monads. They're not used for thunks which are updated precisely once. Depending on the garbage collector, mutable closures may contain extra header information which allows a generational collector to implement the "write barrier."

Notice that mutable objects all have the same general layout: there is a mutable link field as the second word after the header. This is so that code to process old-generation mutable lists doesn't need to look at the type of the object to determine where its link field is.

`MUT_VAR` is a mutable variable.

Fixed Hdr Pointer	*Mutable link*	

`MUT_ARR_PTRS` is a mutable array of pointers. Such an array may be *frozen*, becoming an `MUT_ARR_PTRS_FROZEN`, with a different info-table.

Fixed Hdr	*No of ptrs*	*Mutable link*	*Pointers...*

`MUT_ARR_PTRS_FROZEN` This is the immutable version of `MUT_ARR_PTRS`. It still has a mutable link field for two reasons: we need to keep it on the mutable list for an old generation at least until the next garbage collection, and it may become mutable again via `thawArray`.

Fixed Hdr	*No of ptrs*	*Mutable link*	*Pointers...*

`MVAR`	*Fixed header*	*Head*	*Mutable link*	*Tail*	*Value*

ToDo:*MVars*

9.7.3 Foreign objects

Here's what a ForeignObj looks like:

Fixed header	*Data*

A foreign object is simple a boxed pointer to an address outside the Haskell heap, possible to `malloc`ed data. The only reason foreign objects exist is so that we can track the lifetime of one using weak pointers (see Section 9.7.4) and run a finaliser when the foreign object is unreachable.

9.7.4 Weak pointers

Fixed header	*Key*	*Value*	*Finaliser*	*Link*

ToDo:*Weak poitners*

9.7.5 Stable names

Fixed header	*Index*

ToDo:*Stable names*

The remaining objects types are all administrative — none of them may be entered.

9.8 Other weird objects

`BlockedFetch` **heap objects ('closures')** (parallel only)

`BlockedFetch`s are inbound fetch messages blocked on local closures. They arise as entries in a local blocking queue when a fetch has been received for a local black hole. When awakened, we look at their contents to figure out where to send a resume.

A `BlockedFetch` closure has the form:

Fixed header	link	node	gtid	slot	weight

Spark Closures (parallel only)

Spark closures are used to link together all closures in the spark pool. When the current processor is idle, it may choose to speculatively evaluate some of the closures in the pool. It may also choose to delete sparks from the pool.

Fixed header	*Spark pool link*	*Sparked closure*

Slop Objects Slop objects are used to overwrite the end of an updatee if it is larger than an indirection. Normal slop objects consist of an info pointer a size word and a number of slop words.

Info Pointer	*Size*	*Slop Words*

This is too large for single word slop objects which consist of a single info table.

Note that slop objects only contain an info pointer, not a standard fixed header. This doesn't cause problems because slop objects are always unreachable — they can only be accessed by linearly scanning the heap.

Note:*Currently we don't use slop objects because the storage manager isn't reliant on objects being adjacent, but if we move to a "mostly copying" style collector, this will become an issue.*

9.9 Thread State Objects (TSOs)

In the multi-threaded system, the state of a suspended thread is packed up into a Thread State Object (TSO) which contains all the information needed to restart the thread and for the garbage collector to find all reachable objects. When a thread is running, it may be "unpacked" into machine registers and various other memory locations to provide faster access.

Single-threaded systems don't really *need* TSOs — but they do need some way to tell the storage manager about live roots so it is convenient to use a single TSO to store the mutator state even in single-threaded systems.

Rather than manage TSOs' alloc/dealloc, etc., in some *ad hoc* way, we instead alloc/dealloc/etc them in the heap; then we can use all the standard garbage-collection/fetching/flushing/etc machinery on them. So that's why TSOs are "heap objects," albeit very special ones.

Fixed header
Link field
Mutable link field
What next
State
Thread Id
Exception Handlers
Ticky Info
Profiling Info
Parallel Info
GranSim Info
Stack size
Max Stack size
Sp
Su
SpLim
Stack

The contents of a TSO are:

Link field This is a pointer used to maintain a list of threads with a similar state (e.g. all runnable, all sleeping, all blocked on the same black hole, all blocked on the same MVar, etc.)

Mutable link field Because the stack is mutable by definition, the generational collector needs to track TSOs in older generations that may point into younger ones (which is just about any TSO for a thread that has run recently). Hence the need for a mutable link field (see Section 9.7.2).

What next This field has five values:

- `ThreadEnterGHC` The thread can be started by entering the closure pointed to by the word on the top of the stack.
- `ThreadRunGHC` The thread can be started by jumping to the address on the top of the stack.
- `ThreadEnterHugs` The stack has a pointer to a Hugs-built closure on top of the stack: enter the closure to run the thread.
- `ThreadKilled` The thread has been killed (by `killThread#`). It is probably still around because it is on some queue somewhere and hasn't been garbage collected yet.
- `ThreadComplete` The thread has finished. Its `TSO` hasn't been garbage collected yet.

Thread Id This field contains a (not necessarily unique) integer that identifies the thread. It can be used eg. for hashing.

Ticky Info Optional information for "Ticky Ticky" statistics: `TSO_STK_HWM` is the maximum number of words allocated to this thread.

Profiling Info Optional information for profiling: `TSO_CCC` is the current cost centre.

Parallel Info Optional information for parallel execution.

GranSim Info Optional information for gransim execution.

Stack Info Various fields contain information on the stack: its current size, its maximum size (to avoid infinite loops overflowing the memory), the current stack pointer (*Sp*), the current stack update frame pointer (*Su*), and the stack limit (*SpLim*). The latter three fields are loaded into the relevant registers when the thread is run.

Stack This is the actual stack for the thread, *Stack size* words long. It grows downwards from higher addresses to lower addresses. When the stack overflows, it will generally be relocated into larger premises unless *Max stack size* is reached.

The garbage collector needs to be able to find all the pointers in a stack. How does it do this?

- Within the stack there are return addresses, pushed by `case` expressions. Below a return address (i.e. at higher memory addresses, since the stack grows downwards) is a chunk of stack that the return address "knows about", namely the activation record of the currently running function.
- Below each such activation record is a *pending-argument section*, a chunk of zero or more words that are the arguments to which the result of the function should be applied. The return address does not statically "know" how many pending arguments there are, or their types. (For example, the function might return a result of type α.)
- Below each pending-argument section is another return address, and so on. Actually, there might be an update frame instead, but we can consider update frames as a special case of a return address with a well-defined activation record.

The game plan is this. The garbage collector walks the stack from the top, traversing pending-argument sections and activation records alternately. Next we discuss how it finds the pointers in each of these two stack regions.

9.9.1 Activation records

An *activation record* is a contiguous chunk of stack, with a return address as its first word, followed by as many data words as the return address "knows about". The return address is actually a fully-fledged info pointer. It points to an info table, replete with:

- entry code (i.e. the code to return to).

- closure type is either `RET_SMALL/RET_VEC_SMALL` or `RET_BIG/RET_VEC_BIG`, depending on whether the activation record has more than 32 data words (**Note:***64 for 8-byte-word architectures*) and on whether to use a direct or a vectored return.
- the layout info for `RET_SMALL` is a bitmap telling the layout of the activation record, one bit per word. The least-significant bit describes the first data word of the record (adjacent to the fixed header) and so on. A "`1`" indicates a non-pointer, a "`0`" indicates a pointer. We don't need to indicate exactly how many words there are, because when we get to all zeros we can treat the rest of the activation record as part of the next pending-argument region.

  For `RET_BIG` the layout field points to a block of bitmap words, starting with a word that tells how many words are in the block.
- the info table contains a Static Reference Table pointer for the return address (Section **??**).

The activation record is a fully fledged closure too. As well as an info pointer, it has all the other attributes of a fixed header (Section 9.2) including a saved cost centre which is reloaded when the return address is entered.

In other words, all the attributes of closures are needed for activation records, so it's very convenient to make them look alike.

9.9.2 Pending arguments

So that the garbage collector can correctly identify pointers in pending-argument sections we explicitly tag all non-pointers. Every non-pointer in a pending-argument section is preceded (at the next lower memory word) by a one-word byte count that says how many bytes to skip over (excluding the tag word).

The garbage collector traverses a pending argument section from the top (i.e. lowest memory address). It looks at each word in turn:

- If it is less than or equal to a small constant `MAX_STACK_TAG` then it treats it as a tag heralding zero or more words of non-pointers, so it just skips over them.
- If it points to the code segment, it must be a return address, so we have come to the end of the pending-argument section.
- Otherwise it must be a bona fide heap pointer.

9.10 The Stable Pointer Table

A stable pointer is a name for a Haskell object which can be passed to the external world. It is "stable" in the sense that the name does not change when the Haskell garbage collector runs—in contrast to the address of the object which may well change.

A stable pointer is represented by an index into the `StablePointerTable`. The Haskell garbage collector treats the `StablePointerTable` as a source of roots for GC.

In order to provide efficient access to stable pointers and to be able to cope with any number of stable pointers (eg $0 \ldots 100000$), the table of stable pointers is an array stored on the heap and can grow when it overflows. (Since we cannot compact the table by moving stable pointers about, it seems unlikely that a half-empty table can be reduced in size—this could be fixed if necessary by using a hash table of some sort.)

In general a stable pointer table closure looks like this:

Fixed header	*No of pointers*	*Free*	SP_0	...	SP_{n-1}

The fields are:

`NPtrs`**:** number of (stable) pointers.

`Free`**:** the byte offset (from the first byte of the object) of the first free stable pointer.

SP_i**:** A stable pointer slot. If this entry is in use, it is an "unstable" pointer to a closure. If this entry is not in use, it is a byte offset of the next free stable pointer slot.

When a stable pointer table is evacuated

1. the free list entries are all set to `NULL` so that the evacuation code knows they're not pointers;
2. The stable pointer slots are scanned linearly: non-`NULL` slots are evacuated and `NULL`-values are chained together to form a new free list.

There's no need to link the stable pointer table onto the mutable list because we always treat it as a root.

9.11 Garbage Collecting CAFs

A CAF (constant applicative form) is a top-level expression with no arguments. The expression may need a large, even unbounded, amount of storage when it is fully evaluated.

CAFs are represented by closures in static memory that are updated with indirections to objects in the heap space once the expression is evaluated. Previous version of GHC maintained a list of all evaluated CAFs and traversed them during GC, the result being that the storage allocated by a CAF would reside in the heap until the program ended.

Treating CAFs this way has two problems:

- It can cause a very large space leak. For example, this program should run in constant space but, instead, will run out of memory.

  ```
  > main :: IO ()
  > main = print nats
  >
  > nats :: [Int]
  > nats = [0..maxInt]
  ```

- Expressions with no arguments have very different space behaviour depending on whether or not they occur at the top level. For example, if we make `nats` a local definition, the space leak goes away and the resulting program runs in constant space, as expected.

```
> main :: IO ()
> main = print nats
>  where
>   nats :: [Int]
>   nats = [0..maxInt]
```

  This huge change in the operational behaviour of the program is a problem for optimising compilers and for programmers. For example, GHC will normally flatten a set of let bindings using this transformation:

```
let x1 = let x2 = e2 in e1   ==>   let x2 = e2 in let x1 = e1
```

  but it does not do so if this would raise `x2` to the top level since that may create a CAF. Many Haskell programmers avoid creating large CAFs by adding a dummy argument to a CAF or by moving a CAF away from the top level.

Solving the CAF problem requires different treatment in interactive systems such as Hugs than in batch-mode systems such as GHC

- In a batch-mode the program the runtime system is terminated after every execution of the runtime system. In such systems, the garbage collector can completely “destroy” a CAF when it is no longer live — in much the same way as it “destroys” normal closures when they are no longer live.

- In an interactive system, many expressions are evaluated without restarting the runtime system between each evaluation. In such systems, the garbage collector cannot completely “destroy” a CAF when it is no longer live because, whilst it might not be required in the evaluation of the current expression, it might be required in the next evaluation.

  There are two possible behaviours we migth want:

  1. When a CAF is no longer required for the current evaluation, the CAF should be reverted to its original form. This behaviour ensures that the operational behaviour of the interactive system is a reasonable predictor of the operational behaviour of the batch-mode system. This allows us to use Hugs for performance debugging (in particular, trying to understand and reduce the heap usage of a program) — an area of increasing importance as Haskell is used more and more to solve “real problems” in “real problem domains”.
  2. Even if a CAF is no longer required for the current evaluation, we might choose to hang onto it by collecting it in the normal way. This keeps the space leak but might be useful in a teaching environment when trying to teach the difference between call by name evaluation (which doesn’t share work) and lazy evaluation (which does share work).

It turns out that it is easy to support both styles of use, so the runtime system provides a switch which lets us turn this on and off during execution. **ToDo:***What is this switch called?* It would also be easy to provide a function `RevertCAF` to let the interpreter revert any CAF it wanted between (but not during) executions, if we so desired. Running `RevertCAF` during execution would lose some sharing but is otherwise harmless.

An easy but inefficient fix to the CAF problem would be to make a complete copy of the heap before every evaluation and discard the copy after evaluation. This works but is inefficient.

An efficient way to achieve a similar effect is to revert all updatable thunks to their original form as they become unnecessary for the current evaluation. To do this, we modify the compiler to ensure that the only updatable thunks generated by the compiler are CAFs and we modify the garbage collector to revert entered CAFs to unentered CAFs as their value becomes unnecessary.

9.11.1 New Heap Objects

We add three new kinds of heap object: unentered CAF closures, entered CAF objects and CAF blackholes. We first describe how they are evaluated and then how they are garbage collected.

- Unentered CAF closures contain a pointer to closure representing the body of the CAF. The "body closure" is not updatable.

  Unentered CAF closures contain two unused fields to make them the same size as entered CAF closures — which allows us to perform an inplace update. **ToDo:***Do we have to add another kind of inplace update operation to the storage manager interface or do we consider this to be internal to the SM?*

`CAF_unentered`	*body closure*	*unused*	*unused*

  When an unentered CAF is entered, we do the following:

  - allocate a CAF black hole;
  - push an update frame (to update the CAF black hole) onto the stack;
  - overwrite the CAF with an entered CAF object (see below) with the same body and whose value field points to the black hole;
  - add the CAF to a list of all entered CAFs (called "the CAF list"); and
  - the closure representing the value of the CAF is entered.

  When evaluation of the CAF body returns a value, the update frame causes the CAF black hole to be updated with the value in the normal way.

  ToDo:*Add a picture*

- Entered CAF closures contain two pointers: a pointer to the CAF body (the same as for unentered CAF closures); a pointer to the CAF value (this is initialised with a CAF blackhole, as previously described); and a link to the next CAF in the CAF list

  ToDo:*How is the end of the list marked? Null pointer or sentinel value?.*

`CAF_entered`	*body closure*	*value*	*link*

When an entered CAF is entered, it enters its value closure.

- CAF blackholes are identical to normal blackholes except that they have a different infotable. The only reason for having CAF blackholes is to allow an optimisation of lazy blackholing where we stop scanning the stack when we see the first *normal blackhole* but not when we see a *CAF blackhole*. **ToDo:***The optimisation we want to allow should be described elsewhere so that all we have to do here is describe the difference.*

  Instead of allocating a blackhole to update with the value of the CAF, it might seem simpler to update the CAF directly. This would require a new kind of update frame which would update the value field of the CAF with a pointer to the value and wouldn't catch blackholes caused by CAFs that depend on themselves so we chose not to do so.

9.11.2 Garbage Collection

To avoid the space leak, each run of the garbage collector must revert the entered CAFs which are not required to complete the current evaluation (that is all the closures reachable from the set of runnable threads and the stable pointer table).

It does this by performing garbage collection in three phases:

1. During the first phase, we "mark" all closures reachable from the scheduler state.

   How we "mark" closures depends on the garbage collector. For example, in a 2-space collector, closures are "marked" by copying them into "to-space", overwriting them with a forwarding node and "marking" all the closures reachable from the copy. The only requirements are that we can test whether a closure is marked and if a closure is marked then so are all closures reachable from it.

   ToDo:*At present we say that the scheduler state includes any state that Hugs may have. This is not true anymore.*

   Performing this phase first provides us with a cheap test for execution closures: at this stage in execution, the execution closures are precisely the marked closures.

2. During the second phase, we revert all unmarked CAFs on the CAF list and remove them from the CAF list.

   Since the CAF list is exactly the set of all entered CAFs, this reverts all entered CAFs which are not execution closures.

3. During the third phase, we mark all top level objects (including CAFs) by calling `MarkHugsRoots` which will call `MarkRoot` for each top level object known to Hugs.

To implement the second style of interactive behaviour (where we deliberately keep the CAF-related space leak), we simply omit the second phase. Omitting the second phase causes the third phase to mark any unmarked CAF value closures.

So far, we have been describing a pure Hugs system which contains no machine generated code. The main difference in a hybrid system is that GHC-generated code is statically allocated in memory instead of being dynamically allocated on the heap. We split both `CAF_unentered` and `CAF_entered` into two versions: a static and a dynamic version. The static and dynamic versions of each CAF differ only in whether they are moved during garbage collection. When reverting CAFs, we revert dynamic entered CAFs to dynamic unentered CAFs and static entered CAFs to static unentered CAFs.

10 The Bytecode Evaluator

This section describes how the Hugs interpreter interprets code in the same environment as compiled code executes. Both evaluation models use a common garbage collector, so they must agree on the form of objects in the heap.

Hugs interprets code by converting it to byte-code and applying a byte-code interpreter to it. Wherever possible, we try to ensure that the byte-code is all that is required to interpret a section of code. This means not dynamically generating info tables, and hence we can only have a small number of possible heap objects each with a statically compiled info table. Similarly for stack objects: in fact we only have one Hugs stack object, in which all information is tagged for the garbage collector.

There is, however, one exception to this rule. Hugs must generate info tables for any constructors it is asked to compile, since the alternative is to force a context-switch each time compiled code enters a Hugs-built constructor, which would be prohibitively expensive.

We achieve this simplicity by forgoing some of the optimisations used by compiled code:

- Whereas compiled code has five different ways of entering a closure (Section 11.1.2), interpreted code has only one. The entry point for interpreted code behaves like slow entry points for compiled code.
- We use just one info table for *all* direct returns. This introduces two problems:
    1. How does the interpreter know what code to execute?

       Instead of pushing just a return address, we push a return BCO and a trivial return address which just enters the return BCO.

       (In a purely interpreted system, we could avoid pushing the trivial return address.)
    2. How can the garbage collector follow pointers within the activation record?

       We could push a third word —a bitmask describing the location of any pointers within the record— but, since we're already tagging unboxed function arguments on the stack, we use the same mechanism for unboxed values within the activation record.

       ToDo:*Do we have to stub out dead variables in the activation frame?*
- We trivially support vectored returns by pushing a return vector whose entries are all the same.

- We avoid the need to build SRTs by putting bytecode objects on the heap and restricting BCOs to a single basic block.

10.1 Hugs Info Tables

Hugs requires the following info tables and closures:

`HUGS_RET` .

Contains both a vectored return table and a direct entry point. All entry points are the same: they rearrange the stack to match the Hugs return convention (Section 10.4) and return to the scheduler. When the scheduler restarts the thread, it will find a BCO on top of the stack and will enter the Hugs interpreter.

`UPD_RET` .

This is just the standard info table for an update frame.

Constructors .

The entry code for a constructor jumps to a generic entry point in the runtime system which decides whether to do a vectored or unvectored return depending on the shape of the constructor/type. This implies that info tables must have enough info to make that decision.

`AP` **and** `PAP` .

Indirections .

Selectors .

Hugs doesn't generate them itself but it ought to recognise them

Complex primops .

Some of the primops are too complex for GHC to generate inline. Instead, these primops are hand-written and called as normal functions. Hugs only needs to know their names and types but doesn't care whether they are generated by GHC or by hand. Two things to watch:

1. Hugs must be able to enter these primops even if it is working on a standalone system that does not support genuine GHC generated code.
2. The complex primops often involve unboxed tuple types (which Hugs does not support at the source level) so we cannot specify their types in a Haskell source file.

10.2 Hugs Heap Objects

10.2.1 Byte-code objects

Compiled byte code lives on the global heap, in objects called Byte-Code Objects (or BCOs). The layout of BCOs is described in detail in Section 9.6.4, in this section we will describe their semantics.

Since byte-code lives on the heap, it can be garbage collected just like any other heap-resident data. Hugs arranges that any BCO's referred to by the Hugs symbol tables are treated as live objects by the garbage collector. When a module is unloaded, the pointers to its BCOs are removed from the symbol table, and the code will be garbage collected some time later.

A BCO represents a basic block of code — the (only) entry points is at the beginning of a BCO, and it is impossible to jump into the middle of one. A BCO represents not only the code for a function, but also its closure; a BCO can be entered just like any other closure. Hugs performs lambda-lifting during compilation to byte-code, and each top-level combinator becomes a BCO in the heap.

10.2.2 Thunks and partial applications

A thunk consists of a code pointer, and values for the free variables of that code. Since Hugs byte-code is lambda-lifted, free variables become arguments and are expected to be on the stack by the called function.

Hugs represents updateable thunks with `AP_UPD` objects applying a closure to a list of arguments. (As for `PAP`s, unboxed arguments should be preceded by a tag.) When it is entered, it pushes an update frame followed by its payload on the stack, and enters the first word (which will be a pointer to a BCO). The layout of `AP_UPD` objects is described in more detail in Section 9.6.6.

Partial applications are represented by `PAP` objects, which are non-updatable.

ToDo:*Hugs Constructors.*

10.3 Calling conventions

The calling convention for any byte-code function is straightforward:

- Push any arguments on the stack.
- Push a pointer to the BCO.
- Begin interpreting the byte code.

In a system containing both GHC and Hugs, the bytecode interpreter only has to be able to enter BCOs: everything else can be handled by returning to the compiled world (as described in Section 4.4.1) and entering the closure there.

This would work but it would obviously be very inefficient if we entered a `AP` by switching worlds, entering the `AP`, pushing the arguments and function onto the stack, and entering the function which, likely as not, will be a byte-code object which we will enter by *returning* to the byte-code interpreter. To avoid such gratuitious world switching, we choose to recognise certain closure types as being "standard" — and duplicate the entry code for the "standard closures" in the bytecode interpreter.

A closure is said to be "standard" if its entry code is entirely determined by its info table. *Standard Closures* have the desirable property that the byte-code interpreter can enter the closure by simply "interpreting" the info table instead of switching to the compiled world. The standard closures include:

Constructor To enter a constructor, we simply return (see Section 10.4).

Indirection To enter an indirection, we simply enter the object it points to after possibly adjusting the current cost centre.

`AP` To enter an `AP`, we push an update frame, push the arguments, push the function and enter the function. (Not forgetting a stack check at the start.)

`PAP` To enter a `PAP`, we push the arguments, push the function and enter the function. (Not forgetting a stack check at the start.)

Selector To enter a selector (Section 9.6.3), we test whether the selectee is a value. If so, we simply select the appropriate component; if not, it's simplest to treat it as a GHC-built closure — though we could interpret it if we wanted.

The most obvious omissions from the above list are `BCO`s (which we dealt with above) and GHC-built closures (which are covered in Section 4.4.1).

10.4 Return convention

When Hugs pushes a return address, it pushes both a pointer to the BCO to return to, and a pointer to a static code fragment `HUGS_RET` (this is described in Section 4.4.2). The stack layout is shown in Figure 5.

```
| stack    |
+----------+
| bco      |--> BCO
+----------+
| HUGS_RET |
+----------+
```

Figure 5: Stack layout for a Hugs return address

```
| stack    |
+----------+
| con      |--> CON
+----------+
```

Figure 6: Stack layout on enterings a Hugs return address

When a Hugs byte-code sequence enters a closure, it examines the return address on top of the stack.

```
| stack    |
+----------+
| 3#       |
+----------+
| I#       |
+----------+
```

Figure 7: Stack layout on entering a Hugs return address with an unboxed value

```
| stack    |
+----------+
| ghc_ret  |
+----------+
| con      |--> CON
+----------+
```

Figure 8: Stack layout on enterings a GHC return address

- If the return address is `HUGS_RET`, pop the `HUGS_RET` and the bco for the continuation off the stack, push a pointer to the constructor onto the stack and enter the BCO with the current object pointer set to the BCO (Figure 6).
- If the top of the stack is not `HUGS_RET`, we need to do a world switch as described in Section 4.4.1.

ToDo:*This duplicates what we say about switching worlds (Section* **??***) - kill one or t'other.*

ToDo:*This was in the evaluation model part but it really belongs in this part which is about the internal details of each of the major sections.*

10.5 Addressing Modes

To avoid potential alignment problems and simplify garbage collection, all literal constants are stored in two tables (one boxed, the other unboxed) within each BCO and are referred to by offsets into the tables. Slots in the constant tables are word aligned.

ToDo:*How big can the offsets be? Is the offset specified in the address field or in the instruction?*

Literals can have the following types: char, int, nat, float, double, and pointer to boxed object. There is no real difference between char, int, nat and float since they all occupy 32 bits — but it costs almost nothing to distinguish them and may improve portability and simplify debugging.

10.6 Compilation

To make sense of the instructions, we need a sense of how they will be used. Here is a small compiler for the STG language.

```
> cg (f{a1, ... am}) = do
```

```
| stack    |
+----------+
| ghc_ret  |
+----------+
| 3#       |
+----------+
| I#       |
+----------+
| restart  |--> id_Int#_closure
+----------+
```

Figure 9: Stack layout on enterings a GHC return address with an unboxed value

```
>   pushAtom am; ... pushAtom a1
>   pushVar f
>   SLIDE (m+1) |env|
>   ENTER
> cg (let {x1=rhs1; ... xm=rhsm} in e) = do
>   ALLOC x1 |rhs1|, ... ALLOC xm |rhsm|
>   build x1 rhs1,   ... build xm rhsm
>   cg e
> cg (case e of alts) = do
>   PUSHALTS (cgAlts alts)
>   cg e

> cgAlts { alt1; ... altm }  = cgAlt alt1 $ ... $ cgAlt altm pmFail
>
> cgAlt (x@C{xs} -> e) fail = do
>   TEST C fail
>   HEAPCHECK (heapUse e)
>   UNPACK xs
>   cg e

> build x (C{a1, ... am}) = do
>   pushUntaggedAtom am; ... pushUntaggedAtom a1
>   PACK x C
> -- A useful optimisation
> build x ({v1, ... vm} \ {}. f{a1, ... am}) = do
>   pushVar am; ... pushVar a1
>   pushVar f
>   MKAP x m
> build x ({v1, ... vm} \ {}. e) = do
>   pushVar vm; ... pushVar v1
>   PUSHBCO (cgRhs ({v1, ... vm} \ {}. e))
>   MKAP x m
```

```
> build x ({v1, ... vm} \ {x1, ... xm}. e) = do
>   pushVar vm; ... pushVar v1
>   PUSHBCO (cgRhs ({v1, ... vm} \ {x1, ... xm}. e))
>   MKPAP x m

> cgRhs (vs \ xs. e) = do
>   ARGCHECK   (xs ++ vs)  -- can be omitted if xs == {}
>   STACKCHECK min(stackUse e,heapOverflowSlop)
>   HEAPCHECK  (heapUse e)
>   cg e

> pushAtom x  = pushVar x
> pushAtom i# = PUSHINT i#

> pushVar x = if isGlobalVar x then PUSHGLOBAL x else PUSHLOCAL x

> pushUntaggedAtom x  = pushVar x
> pushUntaggedAtom i# = PUSHUNTAGGEDINT i#

> pushVar x = if isGlobalVar x then PUSHGLOBAL x else PUSHLOCAL x
```

ToDo:*Is there an easy way to add semi-tagging? Would it be that different?*

ToDo:*Optimise thunks of the form* `f{x1,...xm}` *so that we build an AP directly*

10.7 Instructions

We specify the semantics of instructions using transition rules of the form:

	is	s	su	h	hp	σ
$\Longrightarrow$	is'	s'	su'	h'	hp'	σ

where is is an instruction stream, s is the stack, su is the update frame pointer and h is the heap.

10.8 Stack manipulation

Push a global variable .

	PUSHGLOBAL o : is	s	su	h	hp	σ
$\Longrightarrow$	is	$\sigma!o : s$	su	h	hp	σ

Push a local variable .

	PUSHLOCAL o : is	s	su	h	hp	σ
$\Longrightarrow$	is	$s!o : s$	su	h	hp	σ

Push an unboxed int .

	PUSHINT o : is	s	su	h	hp	σ
$\Longrightarrow$	is	$I\# : \sigma!o : s$	su	h	hp	σ

The $I\#$ is a tag included for the benefit of the garbage collector. Similar rules exist for floats, doubles, chars, etc.

Push an unboxed int .

	PUSHUNTAGGEDINT $o : is$	s	su	h	hp	σ
$\Longrightarrow$	is	$\sigma!o : s$	su	h	hp	σ

Similar rules exist for floats, doubles, chars, etc.

Delete environment from stack — ready for tail call .

	SLIDE $m\ n : is$	$as ++ bs ++ cs$	su	h	hp	σ
$\Longrightarrow$	is	$as ++ cs$	su	h	hp	σ

where $|as| = m$ and $|bs| = n$.

Push a return address .

	PUSHALTS $o{:}is$	s	su	h	hp	σ
$\Longrightarrow$	is	`HUGS_RET` $: \sigma!o : s$	su	h	hp	σ

Push a BCO .

	PUSHBCO $o : is$	s	su	h	hp	σ
$\Longrightarrow$	is	$\sigma!o : s$	su	h	hp	σ

10.9 Heap manipulation

Allocate a heap object .

	ALLOC $m : is$	s	su	h	hp	σ
$\Longrightarrow$	is	$hp : s$	su	h	$hp + m$	σ

Build a constructor .

	PACK $o\ o' : is$	$ws ++ s$	su	h	hp	σ
$\Longrightarrow$	is	s	su	$h[s!o \mapsto PackC\{ws\}]$	hp	σ

where $C = \sigma!o'$ and $|ws| = arityC$.

Build an AP or PAP .

	MKAP $o\ m{:}is$	$f : ws ++ s$	su	h	hp	σ
$\Longrightarrow$	is	s	su	$h[s!o \mapsto AP(f, ws)]$	hp	σ

where $|ws| = m$.

	MKPAP $o\ m{:}is$	$f : ws ++ s$	su	h	hp	σ
$\Longrightarrow$	is	s	su	$h[s!o \mapsto PAP(f, ws)]$	hp	σ

where $|ws| = m$.

Unpacking a constructor .

	UNPACK $: is$	$a : s$	su	$h[a \mapsto C\ ws]$	hp	σ
$\Longrightarrow$	is'	$(reverse\ ws) ++ a : s$	su	h	hp	σ

The *reverse ws* looks expensive but, since the stack grows down and the heap grows up, that's actually the cheap way of copying from heap to stack. Looking at the compilation rules, you'll see that we always push the args in reverse order.

10.10 Entering a closure

Enter a BCO .

$$\begin{array}{|llllllll|}\hline & [\text{ENTER}] & a:s & su & h[a \mapsto BCO\{is\}] & hp & \sigma \\ \Longrightarrow & is & a:s & su & h & hp & a \\ \hline\end{array}$$

Enter a PAP closure .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:s & su & h[a \mapsto PAP(f,ws)] & hp & \sigma \\ \Longrightarrow & [\text{ENTER}] & f:ws +\!\!+\, s & su & h & hp & ??? \\ \hline\end{array}$$

Entering an AP closure .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:s & su & h[a \mapsto AP(f,ws)] & hp & \sigma \\ \Longrightarrow & [\text{ENTER}] & f:ws +\!\!+\, \texttt{UPD\_RET}:su:a:s & su' & h & hp & ??? \\ \hline\end{array}$$

Optimisations:

- Instead of blindly pushing an update frame for a, we can first test whether there's already an update frame there. If so, overwrite the existing updatee with an indirection to a and overwrite the updatee field with a. (Overwriting a with an indirection to the updatee also works.) This results in update chains of maximum length 2.

Returning a constructor .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:\texttt{HUGS\_RET}:alts:s & su & h[a \mapsto C\{ws\}] & hp & \sigma \\ \Longrightarrow & alts.entry & a:s & su & h & hp & \sigma \\ \hline\end{array}$$

Entering an indirection node .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:s & su & h[a \mapsto Ind\ a'] & hp & \sigma \\ \Longrightarrow & [\text{ENTER}] & a':s & su & h & hp & \sigma \\ \hline\end{array}$$

Entering GHC closure .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:s & su & h[a \mapsto GHCOBJ] & hp & \sigma \\ \Longrightarrow & [\text{ENTERGHC}] & a:s & su & h & hp & \sigma \\ \hline\end{array}$$

Returning a constructor to GHC .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:GHCRET:s & su & h[a \mapsto C\,ws] & hp & \sigma \\ \Longrightarrow & [\text{ENTERGHC}] & a:GHCRET:s & su & h & hp & \sigma \\ \hline\end{array}$$

10.11 Updates

Updating with a constructor .

$$\begin{array}{|lllllll|}\hline & [\text{ENTER}] & a:\texttt{UPD\_RET}:ua:s & su & h[a \mapsto C\{ws\}] & hp & \sigma \\ \Longrightarrow & [\text{ENTER}] & a +\!\!+\, s & su & h[au \mapsto Ind\ a & hp & \sigma \\ \hline\end{array}$$

Argument checks .

$$\begin{array}{|lllllll|}\hline & \text{ARGCHECK}\ m{:}is & a:as +\!\!+\, s & su & h & hp & \sigma \\ \Longrightarrow & is & a:as +\!\!+\, s & su & h' & hp & \sigma \\ \hline\end{array}$$

where $m \geq (su - sp)$

	ARGCHECK m:is	$a : as ++$ `UPD_RET` $: su : ua : s$	su	h	hp	σ
$\Longrightarrow$	is	$a : as ++ s$	su	h'	hp	σ

where $m < (su - sp)$ and $h' = h[ua \mapsto Ind\ a', a' \mapsto PAP(a, reverse\ as)]$

Again, we reverse the list of values as we transfer them from the stack to the heap — reflecting the fact that the stack and heap grow in different directions.

10.12 Branches

Testing a constructor .

	TEST $tag\ is' : is$	$a : s$	su	$h[a \mapsto C\ ws]$	hp	σ
$\Longrightarrow$	is	$a : s$	su	h	hp	σ

where $C.tag = tag$

	TEST $tag\ is' : is$	$a : s$	su	$h[a \mapsto C\ ws]$	hp	σ
$\Longrightarrow$	is'	$a : s$	su	h	hp	σ

where $C.tag \neq tag$

10.13 Heap and stack checks

	STACKCHECK stk:is	s	su	h	hp	σ
$\Longrightarrow$	is	s	su	h	hp	σ

if s has stk free slots.

	HEAPCHECK hp:is	s	su	h	hp	σ
$\Longrightarrow$	is	s	su	h	hp	σ

if h has hp free slots.

If either check fails, we push the current bco (σ) onto the stack and return to the scheduler. When the scheduler has fixed the problem, it pops the top object off the stack and reenters it.

Optimisations:

- The bytecode CHECK1000 conservatively checks for 1000 words of heap space and 1000 words of stack space. We use it to reduce code space and instruction decoding time.
- The bytecode HEAPCHECK1000 conservatively checks for 1000 words of heap space. It is used in case alternatives.

10.14 Primops

ToDo:*primops take m words and return n words. The expect boxed arguments on the stack.*

11 The Machine Code Evaluator

This section describes the framework in which compiled code evaluates expressions. Only at certain points will compiled code need to be able to talk to the interpreted world; these are discussed in Section **??**.

11.1 Calling conventions

11.1.1 The call/return registers

One of the problems in designing a virtual machine is that we want it abstract away from tedious machine details but still reveal enough of the underlying hardware that we can make sensible decisions about code generation. A major problem area is the use of registers in call/return conventions. On a machine with lots of registers, it's cheaper to pass arguments and results in registers than to pass them on the stack. On a machine with very few registers, it's cheaper to pass arguments and results on the stack than to use "virtual registers" in memory. We therefore use a hybrid system: the first n arguments or results are passed in registers; and the remaining arguments or results are passed on the stack. For register-poor architectures, it is important that we allow $n = 0$.

We'll label the arguments and results $\texttt{arg}_1 \ldots \texttt{arg}_m$ — with the understanding that $\texttt{arg}_1 \ldots \texttt{arg}_n$ are in registers and $\texttt{arg}_{n+1} \ldots \texttt{arg}_m$ are on top of the stack.

Note that the mapping of arguments $\texttt{arg}_1 \ldots \texttt{arg}_n$ to machine registers depends on the *kinds* of the arguments. For example, if the first argument is a Float, we might pass it in a different register from if it is an Int. In fact, we might find that a given architecture lets us pass varying numbers of arguments according to their types. For example, if a CPU has 2 Int registers and 2 Float registers then we could pass between 2 and 4 arguments in machine registers — depending on whether they all have the same kind or they have different kinds.

11.1.2 Entering closures

To evaluate a closure we jump to the entry code for the closure passing a pointer to the closure in $\texttt{arg}_1$ so that the entry code can access its environment.

11.1.3 Function call

The function-call mechanism is obviously crucial. There are five different cases to consider:

1. *Known combinator (function with no free variables) and enough arguments.*

   A fast call can be made: push excess arguments onto stack and jump to function's *fast entry point* passing arguments in $\texttt{arg}_1 \ldots \texttt{arg}_m$.

   The *fast entry point* is only called with exactly the right number of arguments (in $\texttt{arg}_1 \ldots \texttt{arg}_m$) so it can instantly start doing useful work without first testing whether it has enough registers or having to pop them off the stack first.

2. *Known combinator and insufficient arguments.*

   A slow call can be made: push all arguments onto stack and jump to function's *slow entry point.*

   Any unpointed arguments which are pushed on the stack must be tagged. This means pushing an extra word on the stack below the unpointed words, containing the number of unpointed words above it.

The *slow entry point* might be called with insufficient arguments and so it must test whether there are enough arguments on the stack. This *argument satisfaction check* consists of checking that `Su-Sp` is big enough to hold all the arguments (including any tags).

- If the argument satisfaction check fails, it is because there is one or more update frames on the stack before the rest of the arguments that the function needs. In this case, we construct a PAP (partial application, Section 9.6.5) containing the arguments which are on the stack. The PAP construction code will return to the update frame with the address of the PAP in $\texttt{arg}_1$.
- If the argument satisfaction check succeeds, we jump to the fast entry point with the arguments in $\texttt{arg}_1 \ldots \texttt{arg}_{arity}$.

  If the fast entry point expects to receive some of $\texttt{arg}_i$ on the stack, we can reduce the amount of movement required by making the stack layout for the fast entry point look like the stack layout for the slow entry point. Since the slow entry point is entered with the first argument on the top of the stack and with tags in front of any unpointed arguments, this means that if $\texttt{arg}_i$ is unpointed, there should be space below it for a tag and that the highest numbered argument should be passed on the top of the stack.

  We usually arrange that the fast entry point is placed immediately after the slow entry point — so we can just "fall through" to the fast entry point without performing a jump.

3. *Known function closure (function with free variables) and enough arguments.*

   A fast call can be made: push excess arguments onto stack and jump to function's *fast entry point* passing a pointer to closure in $\texttt{arg}_1$ and arguments in $\texttt{arg}_2 \ldots \texttt{arg}_{m+1}$.

   Like the fast entry point for a combinator, the fast entry point for a closure is only called with appropriate values in $\texttt{arg}_1 \ldots \texttt{arg}_{m+1}$ so we can start work straight away. The pointer to the closure is used to access the free variables of the closure.

4. *Known function closure and insufficient arguments.*

   A slow call can be made: push all arguments onto stack and jump to the closure's slow entry point passing a pointer to the closure in $\texttt{arg}_1$.

   Again, the slow entry point performs an argument satisfaction check and either builds a PAP or pops the arguments off the stack into $\texttt{arg}_2 \ldots \texttt{arg}_{m+1}$ and jumps to the fast entry point.

5. *Unknown function closure, thunk or constructor.*

   Sometimes, the function being called is not statically identifiable. Consider, for example, the `compose` function:

   ```
   compose f g x = f (g x)
   ```

Since f and g are passed as arguments to compose, the latter has to make a heap call. In a heap call the arguments are pushed onto the stack, and the closure bound to the function is entered. In the example, a thunk for (g x) will be allocated, (a pointer to it) pushed on the stack, and the closure bound to f will be entered. That is, we will jump to fs entry point passing f in $\texttt{arg}_1$. If $\texttt{arg}_1$ is passed on the stack, it is pushed on top of the thunk for (g x).

The *entry code* for an updateable thunk (which must have arity 0) pushes an update frame on the stack and starts executing the body of the closure — using $\texttt{arg}_1$ to access any free variables. This is described in more detail in Section 11.5.

The *entry code* for a non-updateable closure is just the closure's slow entry point.

In addition to the above considerations, if there are *too many* arguments then the extra arguments are simply pushed on the stack with appropriate tags.

To summarise, a closure's standard (slow) entry point performs the following:

Argument satisfaction check. (function closure only)

Stack overflow check.

Heap overflow check.

Copy free variables out of closure.

Eager black holing. (updateable thunk only)

Push update frame.

Evaluate body of closure.

11.2 Case expressions and return conventions

The *evaluation* of a thunk is always initiated by a case expression. For example:

```
case x of (a,b) -> E
```

The code for a case expression looks like this:

- Push the free variables of the branches on the stack (fv(E) in this case).
- Push a *return address* on the stack.
- Evaluate the scrutinee (x in this case).

Once evaluation of the scrutinee is complete, execution resumes at the return address, which points to the code for the expression E.

When execution resumes at the return point, there must be some *return convention* that defines where the components of the pair, a and b, can be found. The return convention varies according to the type of the scrutinee x:

- (A space for) the return address is left on the top of the stack. Leaving the return address on the stack ensures that the top of the stack contains a valid activation record (Section 9.9.1) — should a garbage collection be required.
- If `x` has a boxed type (e.g. a data constructor or a function), a pointer to `x` is returned in $\texttt{arg}_1$.

  ToDo: *Warn that components of E should be extracted as soon as possible to avoid a space leak.*
- If `x` is an unboxed type (e.g. `Int#` or `Float#`), `x` is returned in $\texttt{arg}_1$
- If `x` is an unboxed tuple constructor, the components of `x` are returned in $\texttt{arg}_1 \ldots \texttt{arg}_n$ but no object is constructed in the heap.

  When passing an unboxed tuple to a function, the components are flattened out and passed in $\texttt{arg}_1 \ldots \texttt{arg}_n$ as usual.

11.3 Vectored Returns

Many algebraic data types have more than one constructor. For example, the `Maybe` type is defined like this:

```
data Maybe a = Nothing | Just a
```

How does the return convention encode which of the two constructors is being returned? A `case` expression scrutinising a value of `Maybe` type would look like this:

```
case E of
  Nothing -> ...
  Just a  -> ...
```

Rather than pushing a return address before evaluating the scrutinee, `E`, the `case` expression pushes (a pointer to) a *return vector*, a static table consisting of two code pointers: one for the `Just` alternative, and one for the `Nothing` alternative.

- The constructor `Nothing` returns by jumping to the first item in the return vector with a pointer to a (statically built) Nothing closure in $\texttt{arg}_1$.

  It might seem that we could avoid loading $\texttt{arg}_1$ in this case since the first item in the return vector will know that `Nothing` was returned (and can easily access the Nothing closure in the (unlikely) event that it needs it. The only reason we load $\texttt{arg}_1$ is in case we have to perform an update (Section 11.5).
- The constructor `Just` returns by jumping to the second element of the return vector with a pointer to the closure in $\texttt{arg}_1$.

In this way no test need be made to see which constructor returns; instead, execution resumes immediately in the appropriate branch of the `case`.

11.4 Direct Returns

When a datatype has a large number of constructors, it may be inappropriate to use vectored returns. The vector tables may be large and sparse, and it may be better to identify the constructor using a test-and-branch sequence on the tag. For this reason, we provide an alternative return convention, called a *direct return*.

In a direct return, the return address pushed on the stack really is a code pointer. The returning code loads a pointer to the closure being returned in $\texttt{arg}_1$ as usual, and also loads the tag into $\texttt{arg}_2$. The code at the return address will test the tag and jump to the appropriate code for the case branch. If $\texttt{arg}_2$ isn't mapped to a real machine register on this architecture, then we don't load it on a return, instead using the tag directly from the info table.

The choice of whether to use a vectored return or a direct return is made on a type-by-type basis — up to a certain maximum number of constructors imposed by the update mechanism (Section 11.5).

Single-constructor data types also use direct returns, although in that case there is no need to return a tag in $\texttt{arg}_2$.

ToDo:*for a nullary constructor we needn't return a pointer to the constructor in* $\texttt{arg}_1$.

11.5 Updates

The entry code for an updatable thunk (which must be of arity 0):

- copies the free variables out of the thunk into registers or onto the stack.
- pushes an *update frame* onto the stack.

  An update frame is a small activation record consisting of

Fixed header	*Update Frame link*	*Updatee*

  Note:*In the semantics part of the STG paper (section 5.6), an update frame consists of everything down to the last update frame on the stack. This would make sense too — and would fit in nicely with what we're going to do when we add support for speculative evaluation.* **ToDo:***I think update frames contain cost centres sometimes*
- If we are doing "eager blackholing," we then overwrite the thunk with a black hole (Section 9.6.8). Otherwise, we leave it to the garbage collector to black hole the thunk.
- Start evaluating the body of the expression.

When the expression finishes evaluation, it will enter the update frame on the top of the stack. Since the returner doesn't know whether it is entering a normal return address/vector or an update frame, we follow exactly the same conventions as return addresses and return vectors. That is, on entering the update frame:

- The value of the thunk is in $\texttt{arg}_1$. (Recall that only thunks are updateable and that thunks return just one value.)
- If the data type is a direct-return type rather than a vectored-return type, then the tag is in $\texttt{arg}_2$.
- The update frame is still on the stack.

We can safely share a single statically-compiled update function between all types. However, the code must be able to handle both vectored and direct-return datatypes. This is done by arranging that the update code looks like this:

```
|       ^       |
| return vector |
|---------------|
|  fixed-size   |
|  info table   |
|---------------|  <- update code pointer
|  update code  |
|       v       |
```

Each entry in the return vector (which is large enough to cover the largest vectored-return type) points to the update code.

The update code:

- overwrites the *updatee* with an indirection to $\texttt{arg}_1$;
- loads `Su` from the Update Frame link;
- removes the update frame from the stack; and
- enters $\texttt{arg}_1$.

We enter $\texttt{arg}_1$ again, having probably just come from there, because it knows whether to perform a direct or vectored return. This could be optimised by compiling special update code for each slot in the return vector, which performs the correct return.

11.6 Semi-tagging

When a `case` expression evaluates a variable that might be bound to a thunk it is often the case that the scrutinee is already evaluated. In this case we have paid the penalty of (a) pushing the return address (or return vector address) on the stack, (b) jumping through the info pointer of the scrutinee, and (c) returning by an indirect jump through the return address on the stack.

If we knew that the scrutinee was already evaluated we could generate (better) code which simply jumps to the appropriate branch of the `case` with a pointer to the scrutinee in $\texttt{arg}_1$. (For direct returns to multiconstructor datatypes, we might also load the tag into $\texttt{arg}_2$).

An obvious idea, therefore, is to test dynamically whether the heap closure is a value (using the tag in the info table). If not, we enter the closure as usual; if so, we jump straight to the appropriate alternative. Here, for example, is pseudo-code for the expression `(case x of { (a,_,c) -> E }`:

```
      \Arg{1} = <pointer to x>;
      tag = \Arg{1}->entry->tag;
      if (isWHNF(tag)) {
          Sp--;  \\ insert space for return address
          goto ret;
      }
      push(ret);
      goto \Arg{1}->entry;

      <info table for return address goes here>
ret:  a = \Arg{1}->data1; \\ suck out a and c to avoid space leak
      c = \Arg{1}->data3;
      <code for E2>
```

and here is the code for the expression `(case x of { [] -> E1; x:xs -> E2 }`:

```
      \Arg{1} = <pointer to x>;
      tag = \Arg{1}->entry->tag;
      if (isWHNF(tag)) {
          Sp--;  \\ insert space for return address
          goto retvec[tag];
      }
      push(retinfo);
      goto \Arg{1}->entry;

      .addr ret2
      .addr ret1
retvec:           \\ reversed return vector
      <return info table for case goes here>
retinfo:
      panic("Direct return into vectored case");

ret1: <code for E1>

ret2: x  = \Arg{1}->head;
      xs = \Arg{1}->tail;
      <code for E2>
```

There is an obvious cost in compiled code size (but none in the size of the bytecodes). There is also a cost in execution time if we enter more thunks than data constructors.

Both the direct and vectored returns are easily modified to chase chains of indirections too. In the vectored case, this is most easily done by making sure that `IND = TAG_1 - 1`, and adding an extra field to every return vector. In the above example, the indirection code would be

```
ind:  \Arg{1} = \Arg{1}->next;
      goto ind_loop;
```

where `ind_loop` is the second line of code.

Note that we have to leave space for a return address since the return address expects to find one. If the body of the expression requires a heap check, we will actually have to write the return address before entering the garbage collector.

11.7 Heap and Stack Checks

The storage manager detects that it needs to garbage collect the old generation when the evaluator requests a garbage collection without having moved the heap pointer since the last garbage collection. It is therefore important that the GC routines *not* move the heap pointer unless the heap check fails. This is different from what happens in the current STG implementation.

Assuming that the stack can never shrink, we perform a stack check when we enter a closure but not when we return to a return continuation. This doesn't work for heap checks because we cannot predict what will happen to the heap if we call a function.

If we wish to allow the stack to shrink, we need to perform a stack check whenever we enter a return continuation. Most of these checks could be eliminated if the storage manager guaranteed that a stack would always have 1000 words (say) of space after it was shrunk. Then we can omit stack checks for less than 1000 words in return continuations.

When an argument satisfaction check fails, we need to push the closure (in R1) onto the stack - so we need to perform a stack check. The problem is that the argument satisfaction check occurs *before* the stack check. The solution is that the caller of a slow entry point or closure will guarantee that there is at least one word free on the stack for the callee to use.

Similarly, if a heap or stack check fails, we need to push the arguments and closure onto the stack. If we just came from the slow entry point, there's certainly enough space and it is the responsibility of anyone using the fast entry point to guarantee that there is enough space.

ToDo:*Be more precise about how much space is required - document it in the calling convention section.*

11.8 Handling interrupts/signals

```
May have to keep C stack pointer in register to placate OS?
May have to revert black holes - ouch!
```

12 The Loader

13 The Compilers

Part IV
History

We're nuking the following:

- Two stacks
- Return in registers. This lets us remove update code pointers from info tables, removes the need for phantom info tables, simplifies semi-tagging, etc.
- Threaded GC. Careful analysis suggests that it doesn't buy us very much and it is hard to work with.

  Eliminating threaded GCs eliminates the desire to share SMReps so they are (once more) part of the Info table.
- RetReg. Doesn't buy us anything on a register-poor architecture and isn't so important if we have semi-tagging.

  ```
  - Probably bad on register poor architecture
  - Can avoid need to write return address to stack on reg rich arch.
    - when a function does a small amount of work, doesn't
      enter any other thunks and then returns.
      eg entering a known constructor (but semitagging will catch this)
  - Adds complications
  ```
- Update in place

  This lets us drop CONST closures and CHARLIKE closures (assuming we don't support Unicode). The only point of these closures was to avoid updating with an indirection.

  We also drop `MIN_UPD_SIZE` — all we need is space to insert an indirection or a black hole.
- STATIC SMReps are now called CONST
- `MUTVAR` is new
- The profiling "kind" field is now encoded in the `INFO_TYPE` field. This identifies the general sort of the closure for profiling purposes.
- Various papers describe deleting update frames for unreachable objects. This has never been implemented and we don't plan to anytime soon.