Architecture of the Haskell Execution Platform (HEP)
Version 6

Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, Simon Peyton Jones
Microsoft Research, Cambridge, UK
{v-julsew,simonmar,v-sfinne,simonpj}@microsoft.com, andy@cse.ogi.edu

29 July 1999

1 What this document is for

As the combined GHC–Hugs system comes ever closer to running compiled code, it becomes clearer than an architectural overhaul is needed. We at MSRC, together with Andy Gill, have been discussing this at length.

Those wishing to go directly to the all-important HEP interface will find it in section 6.1.

2 Names

One frequent point of confusion in our discussions so far has been what the names mean. Here's some defns:

- "GHC" is the standalone native-code compiler.
- "Hugs" denotes the version built from sources in the Glasgow tree, using an STG back end and GHC runtime support. On supported architectures, Hugs will be able to load binaries compiled by GHC.
- "Hugs98" is the current public distribution. This document is not concerned with it. Further changes to Hugs98 will be bugfixes/maintenance, and we expect that Hugs will supercede Hugs98 at some point.

3 Rationale and motivation

As of 10 June, Hugs is able to load and run extremely simple functions compiled by GHC. To get to this stage has required drastic changes to the original Hugs98 from which it was derived:

- dumping the original back end and runtime support, and using an STG-based code generator and GHC's RTS instead.
- adding a new parser to read GHC's interface files (easy) and a significant amount of code to manufacture appropriate symbol table entries (not so easy).
- modifying the import chasing mechanism to follow dependencies through both source and interface files.

These changes, particularly the latter two, are inelegantly integrated into the original structure. It is clear that whatever Hugs emerges as a result of my current hackery will be more a proof-of-concept vehicle than as something which we can carry forwards. Some of the design decisions I made on the way are, in hindsight, bad. A decent system will need a cleaned-up architecture.

Hugs is becoming more complex as more parties modify it for their own diverse purposes. There are now various user interfaces (WinHugs, the "standard" text UI, the HaskellScript stuff, and RunHugs). Not far ahead will come the further complexity of supporting multiple code generators. We already have the original and STG codegens, and further codegens are proposed for Hugs and GHC.

A powerful motivating factor for redoing the architecture is to try and modularise Hugs so that

- supporting these various extensions is simpler.
- supporting different platforms is simpler. Hugs is not too bad, but GHC is very Unix oriented.
- new extensions don't involve grappling with so many parts of the system.
- building customised variants is simpler.

Finally, it would be nice to at least review, and possibly redo, some aspects of the existing code base:

- The conservative garbage collector (now reduced to compile-time only duty in Hugs) has been much discussed. Perhaps we could do away with it and use the GHC heap instead.
- Symbol table (names, classes, values, instances, modules) management in Hugs is too inflexible to support loading and unloading of arbitrary mixtures of compiled and interpreted code in arbitrary orders. It needs redoing.
- The import chasing mechanism is hard to understand, and has proven difficult to adapt for use in Hugs. Redesign is unavoidable.

4 Issues

Here are the major architectural difficulties which have been encountered.

- What mixtures of compiled and interpreted code should be supported? Currently there are at three proposals, in increasing order of complexity to implement:
  - "Downward closure": if module M is compiled, then all modules reachable from M, including Prelude, must be too.
  - "Prelude + any": arbitrary mixtures are allowed, with the proviso that if any module is compiled, then the Prelude must be compiled too.
  - "Any": any mixture at all.

  Underlying problems are:

  - Run-time linking of object files. Use of the Unix `dlopen` facility mandates a downward closure approach, since there is no way to resolve references to interpreted code from compiled code. How the windows DLL loaders would behave I don't know. To be more flexible seems to require writing one's own linkers.
  - Primops. Operations on, and assumptions about representation of primops have to be compatible in compiled and interpreted code. One possibility is to ensure that Hugs's and GHC's primops exactly correspond. That seems wasteful, and difficult and bug-prone to maintain. Another approach it to route all primop calls to GHC, probably by routing them all via a GHC-compiled Prelude.
  - Interface files. All but the downward closure option require Hugs to generate interface files for interpreted modules so GHC can compile against them.

- When the user asks ":reload", how should Hugs go about bringing the execution environment up-to-date? How intelligent should it be? (how accurately do we track changes in the module dependencies?) Where do we put the intelligence? Is Hugs allowed to invoke GHC, or must the user do it? Do the different user interfaces require different levels of sophistication?

- For platforms on which GHC is not supported, we still desire to build a standalone Hugs without object loading facilities. The main trickyness is that a standalone Hugs will have to supply its own implementation of primops, since it can't rely on use of primop facilities in a GHC-compiled prelude.

Some less troublesome issues are

- We might want a form of BCO which can be dumped into a file and reloaded/linked later. One use would be to give an architecture-neutral way to distribute libraries in binary form. Another is for shipping code across a network. Finally, for platforms on which GHC is not supported, it offers the possibility of fast startup, by loading `Prelude.bco` and reading `Prelude.hi`. One consequence of doing dumpable BCOs is that Hugs would need to create interface files.

- Multiple code generators. If Hugs is to acquire other code generators, it may be desirable to create an interface at the boundary between STGland and the code generators.

  Since GHC may also want to target new platforms, work on new code generators should aim to maximise compatibility between Hugs and GHC.

- Loading object files. Hugs is fairly platform independent, except for its need to load object files. We can create an object file loading/linking generic API, and hook specific loaders, for example ELF and Win32 DLL, under that. However, as mentioned above, use of the native facilities may be too inflexible, and so necessitate writing our own linkers.

- Object vs source-level symbol tables. For each function `f` that GHC compiles, a whole bunch of symbols are spewed into the object code: `f_closure`, `f_entry`, `f_info` and `f_fastk`, at the very least.

  On the one hand, you need to remember all these names because other object modules will reference them. On the other hand, the bytecode compiler is only interested in source-level facts about `f`, such as its type, and not about the addresses of its derivative symbols. We propose to have a separate symbol table for object code symbols, with only minimal connection to the source-level symbol table (a pointer to `f_closure` ?)

- Replacement of the conservative garbage collector. It doesn't make much sense to have two heaps, allocators and GCs in the new Hugs. Either we can move compile-time allocation into the execution heap, or move to an arena-based allocator. Hugs allocates heap when compiling so slowly that the latter possibility is quite practical. Either way, the main criteria is to reimplement the `fst`, `snd`, `pair`, &c, macros, so that client code doesn't have to be changed.

  Changing the heap representation would require a new scheme for dealing with Hugs' symbol tables, since pointers to places outside the (Hugs) heap are interpreted in various ways, including as symbol table references.

  It's also unclear what the consequences would be of any client code which dynamically changes the type of a (eg) pair field from pointer to non-pointer, or vice versa.

- Dictionary layouts. Hugs and GHC need to agree on the order in which dictionary arguments are passed, and on the order of methods within a dictionary. We also need to make GHC regard selector functions as possibly overloaded, if necessary, and pass dictionaries appropriately.

- Currently GHC, and therefore Hugs, is very Unix oriented. This is not acceptable for a standalone Hugs. The main blocking points for development on WinNT are the GHC driver (Perl) and the build environment (GNU Make).

- Profiling. If Hugs is to be able to do profiling, it needs to be able to handle scc annotations and probably implement auto-sccification. Will it be difficult to make sure the implemented cost semantics are the same as that of GHC ?

- Type system extensions. If Hugs is to implement them, Hugs and GHC must agree on them. Currently: multiparam type classes, local universal quantification, existential quantification.

- Issues to do with startup and loading the Prelude, so as to minimise code duplication and complexity in Hugs. Interacts with the primops issue.

- Andy is interested in looking into some hooks for debugging.

5 Proposed structure

```
             TextUI    WinUI  HaskellScript  etcUI    VisualStudio
                |        |         |          |           |
                +--------+----+----+----------+ ..........+
                              |
                         ..... HEP interface
      /~                      |
      |                    Session                     ((Compile-time
      |                    Manager                       storage mgr))
      |                       |
 HEP -+         +---------+----+----+----------+---------+
      |         |         |    |    |          |         |
      |    Bytecode    Source  | Object     Object    IFace
      |    Compiler    SymTab  | Loader     SymTab    Reader
      \_                       |
                            StorMgr
                            & Eval
```

This crude picture depicts the main blocks, with lines indicating flow of control, not data. Underlying all components is the compile-time storage manager. The session manager is central to the design, so I'll start there.

The parts from the Session Manager on downwards, including the compile-time storage manager, are collectively known as the Haskell Execution Platform (HEP). HEP has to support a diversity of clients, both those which offer human user interfaces (TextUI, WinUI, etc) or those offering higher level programmatic interfaces (HaskellScript, VisualStudio). Because of this, the HEP interface is critical. It is described in detail in Section 6.1.

- Session manager (SM): Responsible for building an up-to-date executable image (mixture of byte and native code) when the user interfaces request a particular Haskell module to be made available for execution. Responsible for arranging evaluation of Haskell expressions passed from the user interfaces.

  To build an up-to-date image, SM needs to keep track of module dependencies and source/object in-or-out-of-dateness, so as to determine when to reload/recompile modules. It's fair to say that SM will have to incorporate the functionality of (`hbc|nhc|`ϵ)`make`.

  The RTS can support multiple Haskell threads running concurrently, so SM offers that service too. This might be useful for a GUI based Hugs in which there are multiple read-eval windows. Further, SM needs to offer GUIs a way to check their own event queues whilst the evaluator or compiler is running. We have devised what we hope is a flexible scheme to support this. The same mechanism allows UIs to interrupt compilation or evaluation in a portable manner.

- Bytecode Compiler (BC): the previous core of Hugs. Reads Haskell source and translates it via STG code to bytecode, which it places in the (runtime) heap. Updates the Source SymTab

(SSym) entry for this module and references SSym entries for other modules. Optionally emits a simple, GHC-readable interface file for the module.

In order that SM can determine the dependencies of a given source file without attempting full compilation, BC has a facility to parse a file and return the import list, but do nothing more.

- IFace Reader (RdIF): reads GHC created interface files and manufactures symbol table entries in SSym for the specified module.

  Analogously to BC, has a submode for finding just the dependencies of an interface file.

  When called upon to load an interface, RdIF must engage Object Loader (OLoad) to load the corresponding object file. It is OLoad's responsibility to relocate and link this file, since that's platform dependent. However, RdIF must add some minimal info about the object code to this module's SSym entry, namely the address of the `_closure` entry points.

- Source Symbol Table (SSym): is the global source-level symbol table, containing info on modules (imports, exports), classes (types, members, etc), instances, tycons and functions. This is what BC will have to consult and update in order to static-check and typecheck source files. SSym only contains enough info about object files to be able to create calls to GHC compiled functions. At the moment this would appear to be the `f_closure` addresses.

  SSym must be designed so that modules can be thrown out of the system in any order, rather than just the stack-like order in the current Hugs. Fixed limits on table sizes should also be avoided.

  SSym must be designed so client code doesn't have to change. I think most accesses go via the `name`, `isName`, `findName`, `newName` macros (ditto `module`, `cclass`, `inst`, `tycon`), so it's ok.

- Object Symbol Table (OSym): global object-level symbol table. Contains a binding for every symbol in every object currently loaded. New objects can be linked merely by querying OSym; no reference to SSym is needed. As motivation, GHC compiled functions have dozens of symbols not referred to at the source level but which are referred to from other objects, and also internally between code and data sections, so we need to record their addresses.

- Object Loader (OLoad) has two duties: (1) bring an object file into memory and create OSym entries for it. (2) resolve references in an object file in memory by consulting OSym (and possibly SSym?).

  OLoad is obviously object-format dependent. It should be structured so that it has a format independent interface, and implementations of (1) and (2) for each format (ELF, DLL, COFF, etc). The ELF implementation is already done and takes only about 300 lines of C.

  Extra complication: object files can refer to symbols in the RTS, and to symbols like `printf`. These symbols will be bound into the executable Hugs image at the time that is built. So we need a way to find the addresses of symbols "in this process image". On one hand, logically it is up to OSym to supply these addresses. But this is also object/executable format dependent, so OLoad needs to be involved. Blargh. Possibly have an OLoad call to preload OSym with these symbols at Hugs startup time.

- Storage Manager and Evaluator (RTS): This is the GHC RTS, along with the bytecode interpreter already in StgHugs. Executes the native/bytecode mixture. (Not sure what else to say. This is what Simon and Alastair created. It works and is more or less in the required form).
- The user interfaces, TextUI, WinUI, RunHugsUI, etc. These wrap up the services of SM and present them to the end-user, either human or programmatic, in some appropriate way. The pictures show multiple interfaces built into the system, but in practice we expect only one to be linked in to any particular system. The requests that they can pass to SM are:
  - Initialise system, shut down.
  - Change system settings (implements :set in Hugs)
  - Prepare a module for use, returning a module handle.
  - Translate expressions in the context of a given module.
  - Evaluate a translated expression.
  - Query SSym (so that :info, :type, etc can be implemented)
- Underlying everything is the compile-time storage manager. Details currently hazy.

Possible variant 1: multiple code generators

Chop up BC, so it becomes a single source-to-STGcode converter plus some number of STGcode-to-whatever code generators. Hopefully the code generators would all have the same interface.

```
  TextUI    WinUI   RunHugsUI  etcUI    VisualStudio
     |        |         |        |           |
     +--------+----+----+--------+ ..........+
                   |
                Session                   ((Compile-time
                Manager                     storage mgr))
                   |
  +----------+----+----+----------+---------+--------+----------+
  |          |    |    |          |         |        |          |
Haskell   Source  |  Object     Object    IFace    STG to     STG to
to STG    SymTab  |  Loader     SymTab    Reader   Bytecode   OTHER
                  |
               StorMgr
               & Eval
```

Possible variant 2: dumpable BCOs

If BCOs are dumpable to file, they can be regarded as just another flavour of object format. Then the Haskell to BCO (BC) component can be factored off into another process. Loading of BCOs would be done via OLoad, with RdIF reading the interface files which would have to be created by BC. It also means BC would have to read interface files.

This scheme has overheads in both compile speed and implementational complexity. The point of mentioning it is to emphasise the idea that there's no particularly fundamental reason why the BC component should be compiled into SystemC whilst GHC remains a separate entity. If we need to do this, it's not difficult. However, nor is it at present of the highest priority.

6 The component interfaces

Many of the calls can fail. It would be good to think about a consistent error-recovery strategy. I've ignored any error cases in the signatures below. I'm working with Variant 1 (multiple code generators) above.

6.1 Session manager (SM)

These interfaces are presented in IDLishly, with Haskell types. Regard the return types as really being `IO` types.

```
interface IUnknown {
   addRef, release :: ()
}
```

All interfaces inherit reference counting methods in `IUnknown`. When a client copies a pointer, it should `addRef` it, and conversely `release` it when done. Once a pointer is `release`d, the thing it points at may or may not still be alive, but in any case the owner of the pointer must assume it is dead. **ToDo:** *Are these the COM semantics?*

6.1.1 Servers

```
getServer :: HsServer

interface HsServer : IUnknown {
   loadModule   :: LoadHooks -> String -> Maybe HsModule

   setOptions   :: [(String,String)] -> ()
   getOptions   :: [(String,String)]
   canUseObject :: Bool
}
```

For simplicity, we assume there is only one server, obtained via `getServer`. In practice they might be manufactured by a class factory (???), but that's too COM specific here.

A client can ask a `HsServer` object whether it is capable of loading object code with `canUseObject`. HEPs hosted on non-GHC-supporting platforms will return `False` here. The HEP will still work but must interpret everything.

Clients can query and set options for this server using `getOptions` and `setOptions`. `getOptions` returns all the available options and their current values. `setOptions` can supply new values for any subset, or all, of them.

`HsServer`'s main purpose is to supply `loadModule`. Clients supply a string such as "`List`", indicating a module name, with no filename extension or directory. HEP tries to return a `HsModule` handle reflecting the current state of the source/object code base. In doing so, it may need to load and/or compile arbitrary numbers of subsidiary modules. This operation may fail, hence the `Maybe` return type.

Once a `HsModule` handle has been successfully acquired, that handle remains valid until the client calls `release` on it. What the handle refers to is the state of the object/source code base at the time the handle was created. Subsequent changes to the code base have no effect on the handle; the executable images to which it refers are unchanged.

For example, assuming `s :: HsServer` and `h :: LoadHooks`, then given the following sequence of events

1. `m1 = s.loadModule("M", h)`, and this call succeeds
2. The source/object for `M` changes
3. `m2 = s.loadModule("M", h)`

`m1` will continue to be valid and will refer to the original version of `M`, whilst `m2` refers to the new version. Furthermore, if `M` depends on some other modules, `m1` will still be based on the original version of those modules, even if their sources/objects change. In short, once a `HsModule` comes into existence, its meaning never changes.

6.1.2 Load-time hooks

HEP decides for itself what modules it needs to load, when, and in what order. It is up to the client to supply the actual source/object code for modules. To facilitate this, clients must pass a `LoadHooks` object to `loadModule`:

```
interface LoadHooks : IUnknown {
   findModule :: String
         -> (Maybe InStream, Maybe (InStream, InStream))
      -- .hs file, (.hi file, .o file)

   setProgress :: String -> Float -> Bool
      -- True <=> abandon compilation please

   error :: Error -> ()
}
```

When HEP needs a module, it calls `findModule`, passing it the unadorned module name, unadorned in the same way as names passed to `loadModule`. The client should attempt to locate source, interface and object streams for the module in any way it pleases. The returned pair is a possible stream for the source text, and a possible pair of streams for the interface and object text. This latter pairing exists because there's no point in producing an object stream without the corresponding interface stream, or vice versa.

An `InStream` is an abstract handle with which to read a finite stream. `getChar` reads the next character or returns an end-of-file marker. `fprint` returns a fingerprint, perhaps a checksum, or a file timestamp, which characterises the stream's contents. `eqFPrint` compares this stream's fingerprint against a supplied one. The fingerprinting semantics requires that two identical streams have identical fingerprints. **ToDo:** *Do we care that this isn't implementable, strictly speaking?* The intention is to provide HEP with a way to determine to whether a given stream has changed since it was last looked at. `FPrint` is regarded as an abstract type supplied by the client.

```
interface InStream : IUnknown {
   getChar  :: Int
   fprint   :: FPrint
   eqFPrint :: FPrint -> Bool
}
```

HEP notifies the client of compilation/loading progress by calling `setProgress`, giving the name of a goal, eg "Typechecking", and a value, increasing monotonically over a sequence of such calls, from 0.0 to 1.0, indicating progress towards that goal. If the client returns `True` to such a call, HEP abandons compilation as soon as possible. In this way, clients can interrupt compilation in a portable manner.

If a compilation error occurs, HEP creates a `Error` object and passes it to the client with `error`. For the moment, the error object only contains the source coordinates of the error, the `InStream` from which the source originated, and a method `show` which produces the text of the error message. Later versions may contain more information.

```
interface Showable : IUnknown {
   show :: String
}

interface Error : Showable {
   source    :: InStream
   line, col :: Int
}
```

6.1.3 Modules

Once a client has obtained a `HsModule` handle, it can translate and run expressions in the context of that module. Translated values are `HsVal` objects.

```
interface HsModule : IUnknown {
   exprVal :: String -> Bool -> Maybe HsVal
      -- A Haskell expression, treated as if
      -- it was written in the module
      -- Bool==True <=> wrap in `print' if not :: IO t

   idVal :: String -> Maybe HsVal
      -- The name of a top-level value in
      -- scope in the module

   -- takes a regexp, gives list of things
   -- defined in (Bool==True) or in scope in (Bool==False) this mod
   getNames     :: Bool -> String -> [Name]
   getTypes     :: Bool -> String -> [Type]
   getTycons    :: Bool -> String -> [Tycon]
   getClasses   :: Bool -> String -> [Class]
   getInstances :: Bool -> String -> [Instance]

   -- query a specific thing.  String is a name.  Bool as above.
   findName     :: Bool -> String -> Maybe Name
   findType     :: Bool -> String -> Maybe Type
   findTycon    :: Bool -> String -> Maybe Tycon
   findClass    :: Bool -> String -> Maybe Class
   findInstance :: Bool -> String -> String -> Maybe Instance
}
```

The two important methods are `exprVal` and `idVal`. `exprVal` takes an arbitrary Haskell expression and tries to return a translation of it in the context of that module. The caller can optionally ask to have the value wrapped in `print`, since that is often convenient.

`idVal` is simpler. It simply regards the supplied string as the name of a top-level function in scope in the module, and returns a `HsVal` referring to that name. It's conceivable that a skeletal HEP which just loaded object files could implement `idVal` but not `exprVal`. Such a HEP would not need a bytecode compiler or interpreter.

The `get*` and `find*` methods allow clients to consult the HEP's symbol tables. In all cases, the `Bool` parameter facilitates choosing between "defined in this module" and "in scope in this module" interpretation of queries.

`getNames` returns the names of all top-level values matching the wildcard in the supplied string, defined in or in scope in this module. `findName` returns the corresponding information for a specific name; the supplied string must be a real name and not a wildcard. The `Type`, `Tycon`, `Class` and `Instance` calls function analogously for types, type constructors, classes and instances.

As with error messages, currently the only possible action with a name, type, tycon, class or instance is to print it. This may change later.

```
interface Class    : Showable { }
```

```
interface Type     : Showable { }
interface Instance : Showable { }
interface Tycon    : Showable { }
interface Name     : Showable { }
```

6.1.4 Values

A Haskell value is represented by a `HsVal` object. `HsVal`s carry their type, which is obtained with `type`.

New values can be created by application of a value to a list of argument values; `apply` does this. The application is typechecked, and will fail if it is type-incorrect. **ToDo:** *Say more about the rules and extent of this.*

For convenience, new values can be manufactured from integers, using `mkIntValue`. The inverse operation is `intValueOf`, which will fail if the type is wrong. Such mistakes can be avoided by first testing with `isIntValue`. **ToDo:** *What does* `intValueOf` *do if the arg is not in whnf?* Similar functions exist for other primitive types.

```
interface HsVal : IUnknown {
   type  :: Type

   apply :: [HsVal] -> Maybe HsVal
      -- can fail because of type error

   eval   :: RunHooks -> WhyIStopped
      -- To WHNF
   evalIO :: RunHooks -> Maybe WhyIStopped
      -- Runs a value of type IO t, returning the t
      -- the result may or may not be evaluated

   mkIntValue :: Int -> HsVal
   isIntValue :: Bool
   intValueOf :: Maybe Int
      -- ditto Bool Char Word Float Double
}
```

The main purpose of `HsVal` is to supply `eval` and `evalIO`. `eval` evaluates a value, which may be of any type, to WHNF. The client must supply a `RunHooks` object to be used during evaluation. `evalIO` is similar, except that the supplied value must have type `IO t`. The possibly unevaluated `t` is then returned.

```
data WhyIStopped = Done
                 | DoneIO HsVal
                 | YouAskedMeToStop
                 | NoThreadsToRun
```

```
interface RunHooks : IUnknown {
   continue       :: Bool
   stdout, stderr :: Char -> ()
   stdin          :: Char
   -- if the last three block, the entire HEP does
}
```

A `RunHooks` object allows the client to capture `stdout` and `stderr` from the evaluated expression, and supply a `stdin` stream for it. If any of these calls blocks, the entire HEP will too. **ToDo:** *Are we sure?*

When running an expression, HEP will call `continue` on a fairly regular basis, to find out if the client wants to interrupt evaluation. If the client returns `True`, `eval/evalIO` then will return with the value `YouAskedMeToStop`. The client can resume execution later by calling `eval/evalIO` again on that value.

`eval/evalIO` may also return bearing `Done` or `DoneIO` respectively, indicating that the value has reached WHNF. Lastly, a return value of `NoThreadsToRun` indicates that the RTS's scheduler could not find any Haskell threads to run. This could indicate deadlock.

6.1.5 Threading issues for the HEP

There are two kinds of threads to consider: Haskell threads, managed and scheduled by the RTS's scheduler, and operating-system threads.

Haskell threads are easy. An `eval/evalIO` call starts a single "main" thread, and that can create new threads using the Haskell primitive `forkIO`.

The complication lies in OS threads. Rather than attempt to design a multithreaded HEP, we place a mutex around the entire HEP, and allow only one OS thread in at a time. To try and create some fairness, the HEP can, at times convenient to it, check whether any other OS threads are waiting to enter, in which case it may yield, so as to let others enter.

Unfortunately this will cause deadlocks for Haskell programs using callbacks. Typically, a callback function will be supplied to some other subsystem (eg, graphics) using a `ccall` to that subsystem, bearing the `HsVal` of the callback. To run the callback, the subsystem will later call `eval` on the callback. But if the same OS thread is still "inside" the HEP, this call will block.

A solution is to release the lock when an OS thread `ccall`s out of the HEP. This allows other threads, or callbacks in the same thread, to successfully enter the HEP – not necessarily immediately, but eventually, assume the OSs thread scheduling is fair.

6.2 Haskell to STG compiler (Hs2Stg)

```
-- look at text of module and get import list
getImportList :: ModuleName -> [ModuleName]
```

```
-- convert .hs to stg trees
compileModule :: ModuleName -> [STG]
```

6.3 Interface reader (RdIF)

Loading of mutually recursive groups of objects is allowed even though Hugs can't handle that at the source level. Internally, `loadObjectGroup` has to make two passes through the interface files, the first to create partially-filled entries in SSym, and the second to complete those entries.

```
-- look at interface file for module and get import list
getImportList   :: ModuleName -> [ModuleName]

-- load a group of modules, resolving references between them
-- update OSym and SSym
loadObjectGroup :: [ModuleName] -> ()
```

6.4 Source symbol table (SSym)

```
-- create/delete entries for a new module
newModule :: ModuleName -> ()
delModule :: ModuleName -> ()

-- various functions for adding/querying vars, tycons, classes, instances
-- and in particular
setClosureOf :: Name -> Pointer -> ()
getClosureOf :: Name -> Pointer
```

6.5 Object symbol table (OSym)

```
-- add, delete module-level entries
addOMod  :: ModuleName -> ()
delOMod  :: ModuleName -> ()
-- add, find symbols
addOSym  :: ModuleName -> Name -> Pointer -> ()
findOSym :: ModuleName -> Name -> Pointer
```

6.6 Object loader (OLoad)

```
-- check object for correct format, etc
validateObject :: ModuleName -> Bool
-- get object into memory and update OSym, but don't link it
loadObject :: ModuleName -> ()
-- resolve refs in previously loaded object
```

```
linkObject :: ModuleName -> ()
-- remove object from memory
delObject :: ModuleName -> ()
```

6.7 Storage manager and evaluator (RTS)

```
-- eval this ptr to WHNF
enter :: Pointer -> ()
```

6.8 Code generators, STG to bytecode (Stg2BC) and STG to OTHER (Stg2Com)

```
stg2BC, stg2OTHER :: [STG] -> ()
```

6.9 The user interfaces

... don't have a programmatic interface.

7 Tentative design details looking for a home

These sections record how the various bits of the new system should work. In may places it merely records what the existing system does.

7.1 Compile-time storage management

ToDo: draw some pictures of how the tables interrelate.

Relevant structures are

- compile-time heap
- Source symbol table, comprising: name table, tycon table, class table, instance table, module table
- Text table (not sure where this logically belongs)
- Object symbol table

Needs to be redone. Design goals are:

- Should be able to delete an arbitrary module from the system and scrub all references to it. The current Hugs scheme only allows deleting of modules in a LIFO fashion.
- No fixed table sizes for anything! Everything to by dynamically resizable on-the-go. This is a long-overdue change.

- No changes to the client code. No way do we want to rewrite the typechecker, static analyser, etc. That means that any reimplementation will have to supply suitable versions of the many macros used to access storage in current Hugs. Indeed, the viability of this enterprise depends on the observation that the current Hugs consistently uses these macros/functions and does not go directly to the structures.

For the time being, it seems best not to mix the compile-time and run-time heaps. It might be possible, but there are a bunch of unknown factors, and fixing them seems to have a low brownie-points/effort ratio:

- Hugs assumes the existence of a "generic" pointer-ish entity. This might point into the heap, but if it doesn't, then it can be a reference to a symbol table, an integer, a piece of text, or various other things. This simplifies the typechecker and static analysis. (I guess you could say it's a union type).

  Let's call these values "compile-time pointers".

  GHC's storage manager would need to be modified to deal with fields which might or not might be pointers.

- Assignment. Hugs frequently (?) mutates heap objects. I'm unsure what effect this would have on the RTS if a pointer field is changed to a non-pointer one, or vice versa.

  Also, when doing assignments, the old-to-new pointer tables would need to be checked and updated. Changes to client code at assignment points seem unavoidable.

Name, tycon, class and instance tables

Each name table entry describes a top-level value in a Haskell module. It holds a pointer to the textual name, the type, pointers to the STG trees and compiled bytecode, and various other things. A compile-time pointer can point directly at a name table entry.

Like existing Hugs, we continue to organise the name table as an array of name table entries. Unlike existing Hugs, the name entries for a given module do not occupy a contiguous range of the name table. To do so makes it impossible to delete modules, since deletion of a module would create a large hole in the table. To fill this hole, we'd have to slide other entries around, but then we'd need to find and change all compile-time pointers to the moved entries (viz, the usual compacting-GC problem). The latter could be very difficult.

Instead, each name table entry has a link field, which is used to chain together all entries for a given module. For unused entries, the link fields implement a standard freelist. Allocation of new entries consults the freelist. If that's empty, the table is expanded as follows: a bigger array, perhaps twice the size, is malloc'd. The existing name table is copied into it, and the new entries are put on the free list.

When a module is deleted, all its name table entries are put back on the free list.

The tycon, instance and class tables function in exactly the same way. The module table is different.

Notion of the "current module's scope"

When translating a module, Hugs needs to have a way, given the text of a symbol, to (a) find out if it is in scope in this module, and (b) if so, what value/type it is bound to. Because a module can import symbols from other modules, the current scope is not just the contents of the current module.

To support this, name table entries have a second link field. This field is used to chain together all entries in the current scope. Unlike the module-link fields, this chain necessarily visits entries from many different modules.

Because each name table only has one scope-link field, only one current scope can be supported at any time. When Hugs starts processing a new module, the current scope chain has to be rebuilt for that module, by processing its import declarations, and recursing down through other modules as necessary. This operation is expensive and should be done infrequently.

Having a single chain for the current scope makes name lookup terribly inefficient. The current Hugs uses a small 256 entry hash table to help. Each hash table entry points to a chain of name-table entries with the same hash value, chained as described above through the current-scope field. In other words, there are 256 current-scope chains, not just one, and they all begin at the hash table. So, when changing current module, Hugs has to rebuild the hash table as well as the chains.

Tycons, classes and instances use exactly the same scheme.

The module table

Unlike the name, tycon, class and instance table, this one has exactly one entry per module. Each entry contains pointers to: the textual name of the module, an import list, an export list, the start of the name-table chain for this module, ditto for tycons, classes and instances, and perhaps a pointer to a list of STG trees denoting the code generated for the module. When a module is deleted, its module table entry is marked as unused. This table can be resized by copying it into a larger array if needed, in the usual way.

The text table

This remains pretty much the same as before, with the proviso that entries in it cannot be ejected when a module is deleted. If it gets full, we expand it by copying it onto a bigger array in the usual way. In practice this is unlikely to be a problem, since loading a revised version of a recently-deleted module is very likely to present almost the same set of strings as the old version, thereby not increasing the size of the table very much.

One final detail: the current hashing scheme to find stuff in the table will need to be carefully revised so that it can still work when the table size is, say, doubled. Apparently `rts/Hash.c` contains a suitable algorithm.

The object symbol table

We may as well make the object symbol table work in the same way as the name, tycon, class and instance tables. That is, an array of symbol records, with each record having a link field to chain together entries in the same module, and another field linking together entries with the same hash values. The table can expand dynamically, and modules can be thrown out of the table in any order. There is a top-level hash table holding the start point of the hash chains.

Unlike the name, class, tycon and instance tables, the hash table and chains for the object symbol table reflects all the available symbols, not just those regarded as in scope for the current module being translated.

Dividing up the compile-time-pointer space

Hugs has always had the notion of a pointer which *might* point into the compile-time heap, or it might not, denoting a name, piece of text, etc. This is a great idea, but needs redoing.

Problem is that the address spaces for the various kinds of entities are arranged contiguously, with no gaps in between. This make it impossible to expand the address range for some particular kind of entity without disturbing the other address ranges.

We propose instead to use a tag-and-value scheme. A compile-time-pointer is still a 32-bit integer, but divided up thusly:

```
<-1-> <--7--> <--24-->
  0     tag    value
```

The tag space is arranged like this (pretty arbitrary; requires a trawl through existing storage.h to pick up all stuff):

```
000 0000    is an illegal tag
0xx xxxx    denotes a heap number.  value field is address in that heap.
100 0000    Text
100 0001    Name
100 0010    Class
100 0011    Instance
100 0100    an integer; 24-bit value field gives value
100 0101    Object symbol table entry
100 0110    Tuple constructor; value field gives tuple number
100 0111    Offset; value in value field
........
101 0000    ``Box cell tag'' (TAGMIN .. BCSTAG);
            actual value is in value & 0xff
101 0001    ``Constructor tag'' (LETREC .. SPECMIN-1);
            value is in value & 0xff
101 0010    ``Special tag'' (SPECMIN .. PREDEFINED);
            value is in value & 0xff
```

`000 0000` is an illegal tag so that we can continue the convention that a 32-bit zero word means NIL. We could have redefined NIL, but it is frequently tested for, and tests against zero are fast on most (RISC) architectures.

There are up to 63 possible heaps, each with up to 16 M entries. This facilitates a dynamically expanding heap. The idea is to start off with a single small heap of say 20000 cells. When this fills, a new lump of memory can be malloc'd and used as a second heap, of perhaps 40000 cells, giving 60000 in total. Hugs needs to know the size of each heap for GC purposes, but there are no required size relationships between them.

In principle, if Hugs can guarantee that there are no references to a given heap, it can return that memory to the operating system (or at least `free` it).

This is a tradeoff. The `fst` and `snd` macros take more instructions:

```
#define fst(p)    heap[p >> 24][p & 0xffffff].fst
#define snd(p)    heap[p >> 24][p & 0xffffff].snd
```

Each of these translate to 5 Intel x86 insns; I have measured it. In the current Hugs, `fst` and `snd` are just array references, possibly just 1 x86 insn. On the other hand, if `fst(p)` is referenced close in time to `snd(p)`, for the same `p`, it's likely that the second reference will still be in the CPU's data cache. The original Hugs has separate arrays for `fst` and `snd`.

Further compensation is that the ubiquitous `whatIs` call is a lot cheaper this way, since we just shift out the tag:

```
#define whatIs(x)     ((x) >> 24)
```

which is just a single instruction, instead of a whole sequence implementing a binary search tree, as at present.

Texts, names, classes, and other entities with only one tag value, can have up to 16 M entries, since the value field is 24 bits long. After some discussion, we feel that (eg) 16 M names is enough for programs at least ten times the size of GHC, around half a million lines of code.

I'm going to call these entities `HugsPtr` in pseudocode bits.

7.2 The code generator interface

More details on how this hangs together.

```
stg2BC, stg2OTHER :: [(Name, STG_TREE)] -> ()
markRTSobjects_BC,
markRTSobjects_OTHER :: Name -> ()
```

The result of translating a Haskell module to STG trees is a *code list*: a list of pairs `(Name, STG_TREE)`. Each tree represents a function or CAF, either written by the programmer in the source module, or generated automatically, by desugaring, `deriving` clauses, or lambda lifting.

The `Name` component of the pairs points to a name table entry for the tree. And that name table entry points back at the pair. In previous hacking, I often needed to get a name table entry from a tree, and vice versa. Without a bidirectional link, this requires a search of all the name tables or all the trees, which is inefficient.

So each tree has a name table entry, even if it isn't part of the source given by the user. That makes life simpler and more uniform, even if it wastes a little space in the name table.

When the bytecode compiler translates source to trees, it generates a code list, and sets up the links from the code list to the name table and back.

To generate bytecode objects in the GHC-RTS- or OTHER-managed heap, `stg2BC` or `stg2OTHER` is passed the code list. A simple interface, but what goes on inside could be quite complex.

- For one thing, each STG tree can turn into multiple BCOs (I guess I should be more general and say "code blocks"), because of the STG way of translating `case` expressions. At GC time, all of these need to be found and kept alive.

  I propose that each name table entry include the following fields:

  ```
  code_list_entry :: HugsPtr
  rts_primary     :: HugsPtr
  rts_secondaries :: [HugsPtr]
  ```

  `code_list_entry` is a pointer to the relevant code list pair. The STG tree lives, of course, in the compile-time heap.

  After code generation, `rts_primary` and `rts_secondaries` hold pointers into the code blocks in the *runtime* heap. (NB – these pointers are `HugsPtr`s pointing to boxed pointers in the compile-time heap, as at present.) `rts_primary` holds the address of the code-block (or whatever one enters) generated from the tree as a whole. `rts_secondaries` is a list of pointers to subsidiary code blocks, such as `case` return continuations.

  Only the specific code generators needs to understand the precise meaning and layout of what `rts_primary` and `rts_secondaries` point at. Each code generator must also supply a `markRTSobjects` function, which examines and marks the `rts_` entries in the specified name table entry.

- Linking. Code generators will have to traverse the code list twice, once to generate code, and once to fix inter-tree references. ((Blargh – unresolved details ahead)) The first pass translates each tree into a collection of BCOs. Each BCO has unresolved references to other BCOs, but how are they recorded? Erk. But I don't think this is v. important?

  In the second pass, the unresolved refs are fixed.

  It seems that the BCOs can't be constructed directly in the heap, because the intermediate BCOs need a fixup table which the final ones don't. Current StgHugs has `AsmBCO`s for the intermediaries, living in mallocville, and `StgBCOs` for the Real Thing in the RTS heap. The `AsmBCO`s are freed at the end of code generation for a module. Because Hugs doesn't support mutually recursive modules, the entire codegen-resolve cycle can happen on a per-module basis.

8 ToDo:

Clarify how mutually recursive object modules are loaded.