STG Survival Sheet

A poor wee soul

April 30, 2001

Abstract

This document summarises crucial information about the RTS. It focusses on the parallel version of the RTS, also known as GUM. It is meant to be used as a kind of survival sheet for the RTS hacker.

1 Papers You Need to Survive

The design of the new ($\geq$ 4.00) GHC runtime system (RTS) is discussed in [8]. It discusses sequential and concurrent compilation and execution. The documentation on the new RTS in the GHC distribution [6] is far more detailed and a must-read for the serious RTS hacker.

The design of GUM itself is discussed in detail in [9]. Several other papers on the GPH web page [3] discuss details of the RTS such as graph packing etc. Information on how to install and use GPH can be found on the GPH web page [3] and the GHC User's Guide [1].

A good general overview of the compiler is given in [5], although that paper is rather old by now. Several important GHC transformations are discussed in [7]. The STG machine, i.e. the abstract graph reduction machine we extend in order to perform parallel computation, is discussed in [4].

Most of these papers are available from Simon PJ's web page [2] or from the GPH web page [3].

2 A Roadmap of the GUM code in the RTS

2.1 Memory management:

Several include files in `ghc/includes` describe the closure layout (discussed in more detail in Section 3):

`ClosureTypes.h`	List of all available closures
`Closures.h`	The closure layout as `typedefs`

Some crucial properties of closures are specified in the global array `ClosureFlags` defined in `ghc/rts/ClosureFlags.h`. The routines for checking these flags are macros defined in `ghc/includes/InfoTables`. The macros have the form `closure_HNF(c)` or `ip_HNF(ip)` etc.

The layout of InfoTables (`StgInfoTable`) is defined in `ghc/includes/InfoTables`. Remember that the info table contains information common to all closures of one particular type (e.g. number of pointer, entry code etc).

The entry code for the new closures in GUM is defined in `StgMiscClosures.hc` and `FetchMe.hc`.

2.2 Communication:

The communication primitives are split into the high-level communication routines in `ghc/rts/parallel/HLComms.c` and the low-level communication primitives in `ghc/rts/parallel/LLComms.c`. The former contain routines implementing the packing/sending/unpacking/processing of the messages in the GUM abstract machine (see GUM paper). The latter contain routines for manipulating the PVM buffer and performing message passing, currently only on top of PVM. In order to port GUM to MPI, only the routines in `LLComms.c` and the relvant include files have to be modified.

The layout of the pack buffer is defined in `ghc/includes/Parallel.h` as `rtsPackBuffer`.

The (un)packing routines are specified in `ghc/rts/parallel/Pack.c`. The top-level packing function is `PackNearbyGraph`. The function `PackClosure` contains the info how to pack each individual closure. `PackGeneric` is used for most closures and contains the code for copying the contents of a closure into a packet. The corresponding routines for unpacking are: `UnpackGraph`, `UnpackClosure`, and `FillInClosure`.

The (un)acking functions use hash tables to maintain a list of closures that have to be packed (`ClosureQueue`). These functions, as well as the functions for maintaining the GALA and LAGA tables `ghc/rts/parallel/Global.c`, use the hash routines in `ghc/rts/Hash.c`.

2.3 Scheduling:

The top-level scheduling file is `ghc/rts/Schedule.c` and it is a mess. It contains different code for the different setups. The head of the main scheduling loop varies, based on the way of building the RTS, as does most of the body. However the core part, that contains the call into the graph reducer is shared for all versions.

The GUM scheduler is *unfair*, i.e. a thread continues with graph reduction until it is blocked on remote data or local data that is under evaluation by another thread. This can be seen in the code in the case `ThreadYielding` (when checking for the return value): GUM uses a `PUSH_ON_RUN_QUEUE(t)`, i.e. puts the thread at the *beginning* of the run-queue, whereas the std (concurrent) RTS uses `APPEND_TO_RUN_QUEUE(t)` to put it at the end of the run-queue.

At the beginning of the loop some checks have to be done that are specific to the parallel code. If the `PendingFetches` list is non-empty, fetch messages are sent to every PE referred to in the elements of the list (all elements are of type `BLOCKED_FETCH`). If the processor is idle, i.e. has an empty run-queue, we

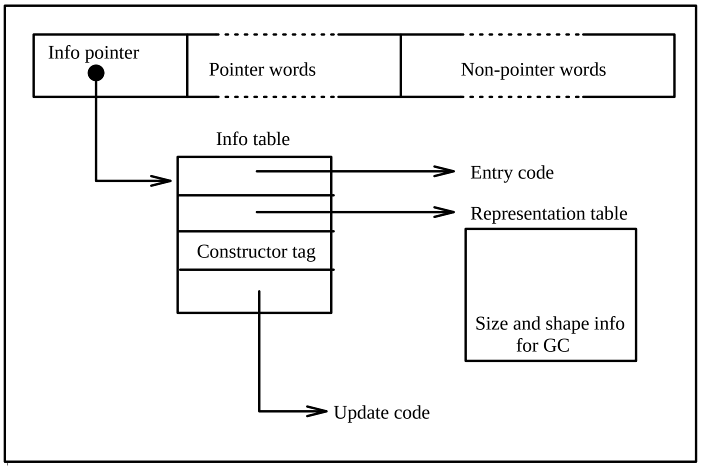

Figure 1: Closure Layout

search for a spark (`findLocalSpark`) and turn it into a thread. If we cannot find a spark on this processor, we send a fish message to a random PE, which is supposed to return a spark.

The file `ghc/rts/Sparks.c` contains routines for spark pool management.

3 Heap Closures

Closure are blobs of memory allocated from within the RTS. Simon et al's paper on the new RTS [8] discusses the details for the memory management. In GUM we have to know what kinds of closures exist and we have to know the closure layout.

Figure 1 shows the general layout of a closure in the heap. Details of the layout of individual closures can be found in the following sections and in [6].

The fixed header has the same structure for all closures. In GUM it only consists of the info pointer (thus FHS, the fixed header size, is 1), in GranSim it also contains a bitmask indicating on which PEs this closure is living. The variable header contains header info specific to one kind of closure. In the new RTS this part is mostly unused. After that we have the pointers related to that closure, then the non-ptr words.

Normally, the size of a closure is stored in its info table, because normally all closures of one type have the same size (PAPs and ARRs are exceptions from that rule). Typically you have to use `sizeW_fromITBL` to get the size of the closure (see `ClosureMacros.h`). In GUM we often use a function `get_closure_info` (in `Parallel.c`) to obtain the crucial info on a closure. It's interface is

```
StgInfoTable* get_closure_info(StgClosure* node, nat *size, nat *ptrs, nat *nonptrs,
                               nat *vhs, char *info_hdr_ty)
```

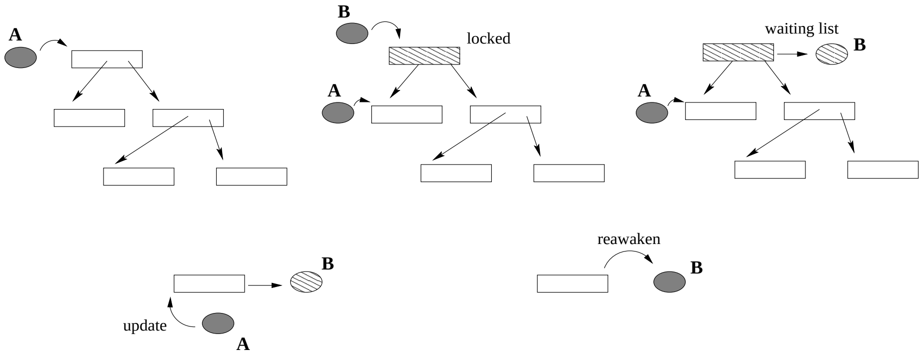

Figure 2: Locking closures and blocking TSOs in GUM

It returns the info pointer of that closure and sets the size, ptrs and non-ptr parameters. It also copies a string with the name of the closure type to the last param. For obtaining just this name you can also use `info_type(closure)` — this is often used in the debugging code.

NB: When you add a new closure type you have give it a number and define its structure in the include files. Then you have to define an info table, which contains among other things the entry code of the closure. See `ghc/rts/StgMiscClosures.hc` as an example. And you have to define evacuation and scavenging routines in `ghc/rts/GC.c` so that the garbage collector knows how to deal with the new closures.

4 Blocking Queues

Figure 2 shows the locking of closures and blocking of TSOs in GUM. Upon reaching an unevaluated closure TSO A locks it (by overwriting the info pointer of the closure) and continues evaluation by jumping to the children of that node. The next TSO B is added to the blocking queue, or waiting list, of the locked closure. When TSO A has finished evaluation, it updates the locked closure with an indirection to the result value. As part of the update operation the blocking queue is awakened, and TSO B can continue evaluation.

The code for adding the current to a blocking queue can be found in the entry code of closures such as `FETCH_ME` (`FetchMe.hc`):

```
STGFUN(stg_FETCH_ME_entry)
{ ...
    CurrentTSO->link = END_BQ_QUEUE;
    ((StgFetchMeBlockingQueue *)R1.cl)->blocking_queue =
      (StgBlockingQueueElement *)CurrentTSO;
  ... }
```

The code for awakening a blocking queue is called from the generic update routine and is defined in `ghc/includes/Updates.h` as the macro `AWAKEN_BQ`.

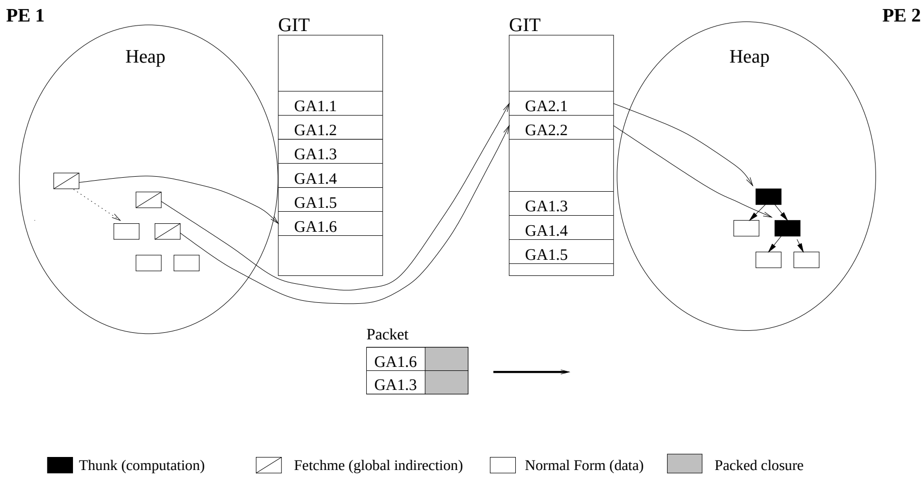

Figure 3: Communication of a graph structure

This code appears in the compiled code, and therefore has to be in the include files. **NB:** If you change code in here, or in any include file, you have to recompile all libraries!.

Only a small set of closure can have a blocking queue: `BLACKHOLE_BQ`, `BLOCKED_FETCH`, `RBH`. In the std RTS (with concurrency) only `TSO` closure can appear in the RTS. In GUM, three different closure can appear there: `TSO`, `BLOCKED_FETCH` (records a request for data from another PE; stores the GA of the `FETCH_ME` on the remote PE), `CONSTR` (words that have been extracted from the closure to make space for the blocking queue; this is only necessary for `RBH` closures in order to make the "revertible"; it can be discarded if the data transfer has been successful). The RTS routines `awakenBlockedQueue` and `unblockOneLocked` in `Schedule.c` does the actual work.

5 Graph Packing

Graph packing traverses a graph in a breadth first fashion, using a queue of closures that still have to be packed. Only the data components are copied into the packet, the pointers can be reconstructed when unpacking the closure. We allocate, by default, GAs for every closure that we pack and put the GA at the beginning of an entry in the packet. Special cases for a GA are a marker for indirections into the packet (in order to maintain sharing in the graph) and a marker for PLCs, i.e. data that is available on all PEs (loaded as part of the text segment). Figure 3 shows a snapshot after communicating a 5-node graph and during the transfer of a 2-node graph that shares one closure. Note that all thunks have been overwritten with `FETCH_ME`s whereas data has been copied.

6 Threaded vs. RTS code

All files ending with `.hc` represent "threaded code" i.e. "assember code" for the abstract graph reduction machine (STG machine). Such code must not modify registers that are under control of the STG machine. The safest way to guarantee this is to have only small C code segments, without block structures and without function call. Every call to a RTS function must be wrapped into a `STGCALL` macro, which saves/restores all STG registers (see `StgMiscClosures.hc` for examples). This remark holds for all entry codes as well as for the code that is generated for builtin primitives.

In the other direction, calling the graph reducer from the RTS is done via the function `StgRun`, which is used in the `Schedule.schedule`. It returns the code indicating whether GC is necessary etc. Normally you shouldn't fiddle with this part of the code.

NB: When compiling the RTS with debugging option, it is important to use the option `-optc-g` only for `.c` but not for `.hc` files (`GhcRtsHcOpts`, `GhcRtsCcOpts`)!! See the commented lines in the `build.mk` file (Appendix D).

7 Profiling

RTS

Profiling information is entered as events into a `.gr` file. The relevant routines are called like this (see `Parallel.c`):

```
DumpRawGranEvent(CURRENT_PROC, CURRENT_PROC,
                 GR_SCHEDULE, t, (StgClosure *)NULL, 0, 0);
```

The format is the same as for GranSim style profiles, therefore the use of the same events such as `GR_SCHEDULE`. To add a new event, modify `GranSimRts.h`, `GranSim.c`, and the dumping routines in `Parallel.c`.

Visualisation tools:

See the User's Guide. The main routines are `gr2ps` and `gr2ap` in `ghc/utils/parallel`. Both translate a `.gr` file into a `.qp` file (`gr2qp`) and then into a postscript file (`qp2ps`, `qp2ap`). The latter are simple Perl scripts — not the fastest way of processing the file but rather easy to hack.

8 Builtin Primitives

Compiler:

Functions such as `par` are treated as builtin primitives by the compiler. The compiler has to know the type and crucial information such as the strictness. These are specified in a few compiler modules: `ghc/compiler/prelude/PrimOps.lhs` and `ghc/compiler/prelude/primops.txt` (the .txt file is automatically processed when building the compiler!) contain the essential information and specify how to translate the code; e.g. `ParOp` expands to a call to `parzh` indicated by the component `‘‘par#’’` in `ghc/compiler/prelude/primops.txt`.

RTS:

The parallelism primitives usually expand to code specified in `ghc/includes/PrimOps.h`. E.g. `parzh` expands to C code that checks the spark pool and inserts a new spark.

9 Useful Tools

Several emacs packages are very useful when hacking the RTS:

- `hideif` mode automatically preprocesses a file and hides code that is not used. Very useful when working on `Schedule.c`.
- `fakeinfo` mode (`(require 'qaHleS)`) allows you to define a tree structure in a file by automatically adding comments (such as `//@menu`) that already exist in most files. Sectioning comments and menus are highlighted. Special keybindings allow navigation in the code.
- `ange-ftp` to directly access files on remote machines.

For enabling these modes, check my `~hwloidl/.emacs` file and my elisp directory `~hwloidl/Elisp`.

An English-Klingon glossary of the most important Klingon phrases used in the GUM documentation exists somewhere in my filespace.

10 Running a parallel program

User-level information on how to compile and run a parallel program can be found in the GHC users guide and in the modified User's Guide that lies on the GPH web page. This section explains the internals of the startup procedure of the parallel code.

The code generated when compiling with option -parallel is in fact a wee Perl script. This script calls at the end the GUM system manager, which starts and coordinates the PEs. The code for the system manager is in `ghc/rts/parallel/SysMan.c`. The main routine in this file starts all PEs and controls the execution and termination. The binary that is generated by GHC actually lives in a central PVM directory. Each PE starts executing the `main` function in `ghc/rts/Main.c` and then initiates graph reduction using the code in the binary file, by calling the top level closure on its stack. The main initialisation routines, that are called from main, are `startupHaskell` in `RtsStartup.c` and more specifically for the parallel system `synchroniseSystem` in `ParInit.c`.

```
foo_mp (Perl) -> SysMan (RTS) -> Main.main (RTS) -> reduce mainIO_closure
```

11 Debugging a parallel program

In order to get more info about the execution of the program out of the RTS, you have to compile the RTS with debugging options by adding the following to your `build.mk` file:

```
GhcRtsCcOpts= -optc-g -optc-DDEBUG
```

In order to debug the RTS you can get more and more information on the following three levels:

1. Use *sanity checks* when running the program! There is a whole range of flags (see `ghc/rts/RtsFlags.h`) encoded as a base 2 number. My favourite RTS flag is `-D161`, which prints info about the (concurrent) scheduler, prints GC info and does (rather aggressive, and thus expensive) sanity checks such as checking entire heap for consistency after GC.

2. Use *parallel debug flags* to produce tracing information in GUM! You can, for example, print the contents of every packet that is sent, the contents of blocking queues etc (see Table 1). The encoding is as a base 2 number as above. The RTS flag is `-qD<n>`. The following table summarises the available flags. Choose e.g. `-qD513` to print general verbose messages and packet contents.

   Code in the RTS, that is only enabled with flag foo, has the structure:

   ```
   IF_PAR_DEBUG(foo, ... C code ...)
   ```

   NB: especially flags 8 and above produce a *lot* of info and are very good in filling an entire partition.

3. Use *gdb* to trace the behaviour the prg and RTS! Start the parallel prgram as usual with the additional RTS option `-qw<n>`. This adds a wait loop at the beginning of the program. `<n>` has to be large enough so that you can attach a gdb process to each PE. On each machine that you use, do a `ps` to get PID of the process and the full prg name of the binary that is being executed. Then start gdb with `gdb <pid> <prg>`. Now you can set breakpoints, trace execution etc in the RTS in each of your gdb windows. I use a few scripts that automatically attach a gdb to a running GUM PE. If you really have to go to that level of debugging, get in touch with me and I can give you some more hints/scripts/etc.

Useful debugging routines, such as `PrintGraph`, can be found in `ghc/rts/parallel/ParallelDebug.c`. Sanity checks are defined in `ghc/rts/Sanity.c`.

Table 1: GUM RTS debug flags (see `RtsFlags.c`)

Flag	Name	Comment
0	verbose	general comments
1	bq	blocking queue info and contents
2	schedule	scheduling info
3	free	free messages (distrib GC)
4	resume	resume messages
5	weight	info on GA weight (distrib GC)
6	fetch	fetch messages
7	fish	fish messages (searching for work)
8	tables	contents of GALA etc tables
9	packet	contents of packets
10	pack	info on graph packing
11	paranoia	*very* detailed info

Table 2: Register mapping (see `MachRegs.h`)

Reg	STG-Reg	Comment
ebx	Base	Ptr to the base of the current TSO
ebp	Sp	Stack pointer
esi	R1	Node register (ptr to the current closure)
edi	Hp	Heap pointer

12 Registers

Once you start debugging the RTS with gdb you have to know the mapping of STG registers to hardware registers in order to figure out what's going on. Depending on the architecture a subset of the STG registers are mapped onto hardware registers. All registers are stored in the TSO when a context switch occurs.

The mapping of STG registers to machine registers is defined in `MachRegs.h`. For x86 architectures we have only 4 registers available, which are mapped as shown in Table 2.

Note that important STG registers such as Su, SpLim, and HpLim are not mapped to machine registers — there are just too few of them.

13 Web Pages

GHC page (with sections on CVS repository and CVS cheat sheet):
`http://haskell.cs.yale.edu/ghc`

GPH page:
http://www.cee.hw.ac.uk/~dsg/gph
GUM announcement with info how to install and run it:
ftp://ftp.cee.hw.ac.uk/pub/gph/ANNOUNCE.GUM
Updated GHC User's Guide (see Parallel Haskell sections):
http://www.cee.hw.ac.uk/~dsg/gph/docs/4.06/users_guide/book1.html
GranSim User's Guide (partially relevant for GUM):
http://www.dcs.gla.ac.uk/fp/software/gransim/user_toc.html

14 References

References

[1] The Glasgow Haskell Compiler. WWW page, January 1998.

[2] Simon Peyton Jones' Web Page. WWW page, April 2001. URL: http://research.microsoft.com/Users/simonpj/.

[3] GPH Web Page. WWW page, April 2001. URL: http://www.cee.hw.ac.uk/~dsg/gph.

[4] S. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming 2(2)*, 2(2):127–202, Apr. 1992.

[5] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, UK, Mar. 1993. URL: http://research.microsoft.com/Users/simonpj/Papers/grasp-jfit.ps.Z.

[6] S. P. Jones, S. Marlow, and A. Reid. The STG runtime system (revised). Technical report, Microsoft Research Ltd., Cambridge, 2001. Part of GHC-5.00: fptools/ghc/docs/rts/rts.tex.

[7] S. P. Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming 32(1-3), pp3-47, September 1998.*, 32(1–3):3–47, Sept. 1998. URL: http://research.microsoft.com/Users/simonpj/Papers/comp-by-trans-scp.ps.gz.

[8] S. Marlow and S. Peyton. The New GHC/Hugs Runtime System. Unpublished. URL: http://research.microsoft.com/Users/simonpj/Papers/new-rts.ps.gz.

[9] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable implementation of Haskell. In *International Workshop on the Implementation of Functional Languages*, Bastad, Sweden, Sept. 1995. URL: http://www.cee.hw.ac.uk/~dsg/gph/papers/ps/gum-ifl95.ps.

Table 3: Closure Types (see `ClosureTypes.h`)

INVALID_OBJECT	0	THUNK	18	RET_BCO	37
CONSTR	1	THUNK_1_0	19	RET_SMALL	38
CONSTR_1_0	2	THUNK_0_1	20	RET_VEC_SMALL	39
CONSTR_0_1	3	THUNK_2_0	21	RET_BIG	40
CONSTR_2_0	4	THUNK_1_1	22	RET_VEC_BIG	41
CONSTR_1_1	5	THUNK_0_2	23	RET_DYN	42
CONSTR_0_2	6	THUNK_STATIC	24	UPDATE_FRAME	43
CONSTR_INTLIKE	7	THUNK_SELECTOR	25	CATCH_FRAME	44
CONSTR_CHARLIKE	8	BCO	26	STOP_FRAME	45
CONSTR_STATIC	9	AP_UPD	27	SEQ_FRAME	46
CONSTR_NOCAF_STATIC	10	PAP	28	BLACKHOLE	47
FUN	11	IND	29	BLACKHOLE_BQ	48
FUN_1_0	12	IND_OLDGEN	30	SE_BLACKHOLE	49
FUN_0_1	13	IND_PERM	31	SE_CAF_BLACKHOLE	50
FUN_2_0	14	IND_OLDGEN_PERM	32	MVAR	51
FUN_1_1	15	IND_STATIC	33	ARR_WORDS	52
FUN_0_2	16	CAF_UNENTERED	34	MUT_ARR_PTRS	53
FUN_STATIC	17	CAF_ENTERED	35	MUT_ARR_PTRS_FROZEN	54
THUNK	18	CAF_BLACKHOLE	36	MUT_VAR	55
WEAK	56	**TSO**	59	EVACUATED	64
FOREIGN	57	**BLOCKED_FETCH**	60		
STABLE_NAME	58	**FETCH_ME**	61		
		FETCH_ME_BQ	62		
		RBH	63		

A ClosureTypes.h

Table 3 shows the types of closures available in the RTS. At the end of the list you'll find the closures specific to the foreign language interface and to parallelism (boldface).

B Closures.h

This section summarises the contents of the individual structures. As a general rule there is one C-typedef for each type of closure. Additionally, there are kind-of collective types, which reflect a layout common to several types of closure. Logically, this should be a union type, but making it another struct and typecasting where necessary is usually easier to do in the code. As an example, `StgClosure` is such a collective closure, reflecting the common layout of all closures. Note, that in GUM we need a `StgBlockingQueueElement` as a collective type over, `StgTSO`, `StgBlockedFetch` and `StgConstr`, which may all appear in a blocking queue (in the std RTS only `StgTSOs` can occur in the RTS).

```
typedef struct StgClosure_ {
    StgHeader   header;
    struct StgClosure_ *payload[0];
} StgClosure;

/* What a stroke of luck - all our mutable closures follow the same
 * basic layout, with the mutable link field as the second field after
 * the header.  This means the following structure is the supertype of
 * mutable closures.
 */
```

```
typedef struct StgMutClosure_ {
    StgHeader   header;
    StgPtr     *padding;
    struct StgMutClosure_ *mut_link;
    struct StgClosure_ *payload[0];
} StgMutClosure;

typedef struct {
    StgHeader   header;
    StgClosure *selectee;
} StgSelector;

typedef struct {
    StgHeader   header;
    StgWord     n_args;
    StgClosure *fun;
    StgPtr      payload[0];
} StgPAP;

typedef struct {
    StgHeader   header;
    StgWord     n_args;
    StgClosure *fun;
    StgPtr      payload[0];
} StgAP_UPD;

typedef struct {
    StgHeader  header;
    StgWord    n_ptrs;
    StgWord    n_words;
    StgWord    n_instrs;
    StgWord    stgexpr;
    StgPtr     payload[0];
} StgBCO;

typedef struct {
    StgHeader   header;
    StgClosure *indirectee;
} StgInd;

typedef struct {
    StgHeader   header;
    StgClosure *indirectee;
    StgMutClosure *mut_link;
} StgIndOldGen;

typedef struct {
    StgHeader   header;
    StgClosure *indirectee;
```

```
    StgClosure *static_link;
} StgIndStatic;

typedef struct StgCAF_ {
    StgHeader     header;
    StgClosure    *body;
    StgMutClosure *mut_link;
    StgClosure    *value;
    struct StgCAF_ *link;
} StgCAF;

typedef struct {
    StgHeader  header;
    StgWord    words;
    StgWord    payload[0];
} StgArrWords;

typedef struct {
    StgHeader   header;
    StgWord     ptrs;
    StgMutClosure *mut_link;    /* mutable list */
    StgClosure *payload[0];
} StgMutArrPtrs;

typedef struct {
    StgHeader   header;
    StgClosure *var;
    StgMutClosure *mut_link;
} StgMutVar;

typedef struct _StgUpdateFrame {
    StgHeader  header;
    struct _StgUpdateFrame *link;
    StgClosure *updatee;
} StgUpdateFrame;

typedef struct {
    StgHeader  header;
    struct _StgUpdateFrame *link;
} StgSeqFrame;

typedef struct {
    StgHeader  header;
    struct _StgUpdateFrame *link;
    StgInt      exceptions_blocked;
    StgClosure *handler;
} StgCatchFrame;

typedef struct {
    StgHeader  header;
```

```
} StgStopFrame;

typedef struct {
    StgHeader   header;
    StgClosure *evacuee;
} StgEvacuated;

typedef struct {
  StgHeader header;
  StgWord data;
} StgIntCharlikeClosure;

/* statically allocated */
typedef struct {
  StgHeader  header;
} StgRetry;

typedef struct _StgForeignObj {
  StgHeader      header;
  StgAddr        data;          /* pointer to data in non-haskell-land */
} StgForeignObj;

typedef struct _StgStableName {
  StgHeader      header;
  StgWord        sn;
} StgStableName;

typedef struct _StgWeak {       /* Weak v */
  StgHeader header;
  StgClosure *key;
  StgClosure *value;            /* v */
  StgClosure *finalizer;
  struct _StgWeak *link;
} StgWeak;

typedef struct _StgDeadWeak {   /* Weak v */
  StgHeader header;
  struct _StgWeak *link;
} StgDeadWeak;

/* Dynamic stack frames - these have a liveness mask in the object
 * itself, rather than in the info table.  Useful for generic heap
 * check code.
 */

typedef struct {
  const struct _StgInfoTable* info;
  StgWord        liveness;
  StgWord        ret_addr;
  StgWord        payload[0];
```

```
} StgRetDyn;

/* Concurrent communication objects */

typedef struct {
  StgHeader       header;
  struct StgTSO_ *head;
  StgMutClosure  *mut_link;
  struct StgTSO_ *tail;
  StgClosure*     value;
} StgMVar;

#if defined(PAR) || defined(GRAN)
/*
  StgBlockingQueueElement represents the types of closures that can be
  found on a blocking queue: StgTSO, StgRBHSave, StgBlockedFetch.
  (StgRBHSave can only appear at the end of a blocking queue).
  Logically, this is a union type, but defining another struct with a common
  layout is easier to handle in the code (same as for StgMutClosures).
*/
typedef struct StgBlockingQueueElement_ {
  StgHeader                         header;
  struct StgBlockingQueueElement_  *link;
  StgMutClosure                    *mut_link;
  struct StgClosure_               *payload[0];
} StgBlockingQueueElement;

typedef struct StgBlockingQueue_ {
  StgHeader                 header;
  struct StgBlockingQueueElement_ *blocking_queue;
  StgMutClosure            *mut_link;
} StgBlockingQueue;

/* this closure is hanging at the end of a blocking queue in (par setup only) */
typedef struct StgRBHSave_ {
  StgHeader    header;
  StgPtr       payload[0];
} StgRBHSave;

typedef struct StgRBH_ {
  StgHeader                                header;
  struct StgBlockingQueueElement_         *blocking_queue;
  StgMutClosure                           *mut_link;
} StgRBH;

#else

typedef struct StgBlockingQueue_ {
  StgHeader          header;
  struct StgTSO_    *blocking_queue;
```

```
  StgMutClosure     *mut_link;
} StgBlockingQueue;

#endif

#if defined(PAR)
/* global indirections aka FETCH_ME closures */
typedef struct StgFetchMe_ {
  StgHeader              header;
  globalAddr            *ga;            /* type globalAddr is abstract here */
  StgMutClosure         *mut_link;
} StgFetchMe;

/* same contents as an ordinary StgBlockingQueue */
typedef struct StgFetchMeBlockingQueue_ {
  StgHeader                          header;
  struct StgBlockingQueueElement_   *blocking_queue;
  StgMutClosure                     *mut_link;
} StgFetchMeBlockingQueue;

/* entry in a blocking queue, indicating a request from a TSO on another PE */
typedef struct StgBlockedFetch_ {
  StgHeader                         header;
  struct StgBlockingQueueElement_  *link;
  StgMutClosure                    *mut_link;
  StgClosure                       *node;
  globalAddr                        ga;
} StgBlockedFetch;
#endif
```

C ClosureFlags.c

The basic properties of the closure types are defined in an array `closure_flags` indexed with the type of the closure (as int). The abbreviations stand for (see `InfoTables.h`):

HNF head normal form ie. the closure is evaluated and cannot be evaluated further

BTM bitmap ie. this closure has a bitmap attached to it indicating which elements of the closure are pointers and which are non-pointers

NS non-sparkable ie. no parallel process is generated for a graph with an NS closure as root (GUM only)

STA static ie. is this a statically allocated closure; note that such closures live in a different part of the memory; functions such as `LOOKS_LIKE_STATIC` can be used to determine whether it lives in static space, indeed.

THU thunk ie. this is something that can be evaluated further.

MUT mutable ie. a closure that is mutable

UPT unpointed

SRT a closure with an SRT attached to it

		0	1	2	3	4	5	6	7
		HNF	BTM	NS	STA	THU	MUT	UPT	SRT
0	INVALID_OBJECT								
1	CONSTR	HNF		NS					
2	CONSTR_1_0	HNF		NS					
3	CONSTR_0_1	HNF		NS					
4	CONSTR_2_0	HNF		NS					
5	CONSTR_1_1	HNF		NS					
6	CONSTR_0_2	HNF		NS					
7	CONSTR_INTLIKE	HNF		NS	STA				
8	CONSTR_CHARLIKE	HNF		NS	STA				
9	CONSTR_STATIC	HNF		NS	STA				
10	CONSTR_NOCAFSTATIC	HNF		NS	STA				
11	FUN	HNF		NS	STA			SRT	
12	FUN_1_0	HNF		NS					
13	FUN_0_1	HNF		NS					
14	FUN_2_0	HNF		NS					
15	FUN_1_1	HNF		NS					
16	FUN_0_2	HNF		NS					
17	FUNSTATIC	HNF		NS	STA			SRT	
18	THUNK		BTM			THU			SRT
19	THUNK_1_0		BTM			THU			SRT
20	THUNK_0_1		BTM			THU			SRT
21	THUNK_2_0		BTM			THU			SRT
22	THUNK_1_1		BTM			THU			SRT
23	THUNK_0_2		BTM			THU			SRT
24	THUNKSTATIC		BTM		STA	THU			SRT
25	THUNK_SELECTOR		BTM			THU			SRT
26	BCO	HNF	NS						
27	AP_UPD		BTM			THU			
28	PAP	HNF		NS					
29	IND			NS					
30	IND_OLDGEN			NS					
31	IND_PERM			NS					
32	IND_OLDGEN_PERM			NS					
33	INDSTATIC			NS	STA				
34	CAF_UNENTERED								
35	CAF_ENTERED								
36	CAF_BLACKHOLE		BTM	NS			MUT	UPT	
37	RET_BCO		BTM						
38	RET_SMALL		BTM						SRT
39	RET_VEC_SMALL		BTM						SRT
40	RET_BIG								SRT
41	RET_VEC_BIG								SRT
42	RET_DYN								SRT

43	UPDATE_FRAME		BTM			
44	CATCH_FRAME		BTM			
45	STOP_FRAME		BTM			
46	SEQ_FRAME		BTM			
47	BLACKHOLE			NS	MUT	UPT
48	BLACKHOLE_BQ			NS	MUT	UPT
49	SE_BLACKHOLE			NS		UPT
50	SE_CAF_BLACKHOLE			NS		UPT
51	MVAR	HNF		NS	MUT	UPT
52	ARR_WORDS	HNF		NS		UPT
53	MUT_ARR_PTRS	HNF		NS	MUT	UPT
54	MUT_ARR_PTRS_FROZEN	HNF		NS	MUT	UPT
55	MUT_VAR	HNF		NS	MUT	UPT
56	WEAK	HNF		NS		UPT
57	FOREIGN	HNF		NS		UPT
58	STABLE_NAME	HNF		NS		UPT
59	TSO	HNF		NS	MUT	UPT
60	BLOCKED_FETCH	HNF		NS	MUT	UPT
61	FETCH_ME	HNF		NS	MUT	UPT
62	FETCH_ME_BQ			NS	MUT	UPT
63	RBH			NS	MUT	UPT
64	EVACUATED					
65	N_CLOSURE_TYPES					

D HOWTO checkout and build GHC.

CVS checkout:

See the CVS cheat sheet on the GHC page for more details.

Set your CVSROOT environment variable to :pserver:anoncvs@glass.cse.ogi.edu:/cvs

Only the first time around do login:

```
$ cvs login
```

(password is cvs). For a checkout of the sources do:

```
$ cvs checkout fpconfig
$ cd fptools
$ cvs checkout ghc
```

Setup a build dir:

```
$ lndir <srcdir> <builddir>
```

e.g

```
$ lndir /export/tmp/SRCS/ghc-5.00/fptools /export/tmp/BUILDS/ghc-5.00/i386-unknown-li
```

Configure GHC:

Create a file build.mk in ¡builddir¿/mk. See attached example file. Important to set GhcLibWays to build libraries for GUM (way mp).

```
$ cd /export/tmp/BUILDS/ghc-5.00/i386-unknown-linux
$ autoconf
$ cd ghc
$ autoconf
$ cd ..
$ ./configure
```

Build GHC:

```
$ make
```

Pitfalls:

Before doing a checkout set CVS_RSH to ssh (if available) or rsh.

Example build.mk

```
# Time-stamp: <Tue Apr 24 2001 15:02:13 stardate: Command not found. hwloidl>
#
# Build file for a parallel setup:
#   GUM + GranSim
# Enabling parallel ticky RTS monitoring on an otherwise optimised RTS
# -----------------------------------------------------------------------------

# Directory used by GHC (and possibly other tools) for storing
# temporary files.  If your /tmp isn't big enough, either override
# this in build.mk or set your environment variable "TMPDIR" to point
# to somewhere with more space.  (TMPDIR=. is a good choice).

TMPDIR                  = /tmp

# GhcOpts = -pgmC/net/dazdak/ghc-4.02/fptools/lib/i386-unknown-linux/ghc-4.02/hsc

# Compiler to use for building ghc (same as ./configure --with-hc=... )
WithGhcHc = /app/lang/functional/ghc-4.08.2/bin/i386-unknown-linux/ghc-4.08.2

# Option flags to pass to GHC when it's compiling RTS modules
# This is a good way to set things like -optc-g and -optc-DDEBUG for the RTS.
# GhcRtsHcOpts is used when compiling .hc files.
# GhcRtsCcOpts is used when compiling .c  files.

GhcRtsHcOpts=
```

```
#GhcRtsCcOpts= -optc-g -optc-DDEBUG
#GhcRtsCcOpts= -optc-O

WAY_mp_CC_OPTS= -optc-O

# Extra ways in which to build the compiler (for example, you might want to
# build a profiled compiler so you can see where it spends its time)
# GhcCompilerWays= mg

# What extra ways to build the libraries in
# In addition to the normal sequential way, the default is to also build
# profiled prelude libraries.
# When booting from .hc files, turn this off.
GhcLibWays= mp
# ToDo: add mg s

# ushas option: be verbose when building the libraries to see where we crash
# HWL: for debugging GranSim generate .hc and .s files
# GhcLibHcOpts += -keep-hc-files-too -keep-s-files-too
```