GHC’s Runtime System

Ben Gamari — ben@well-typed.com

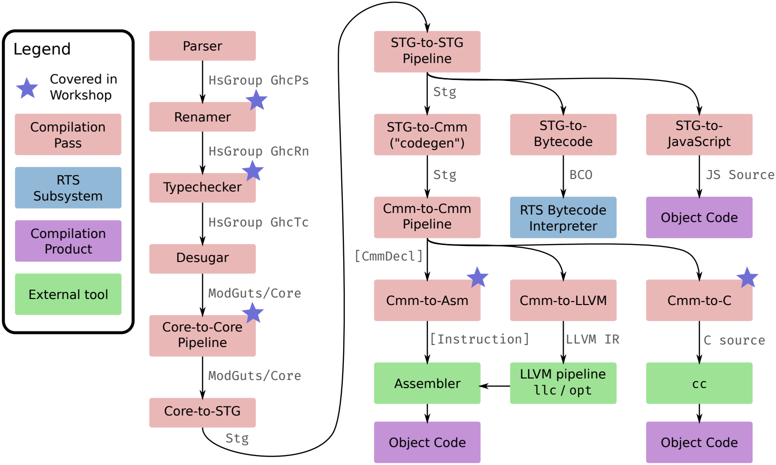
Well-Typed

Overview

- Review of RTS's responsibilities
- Heap structure
- Storage manager
  - Block allocator
  - Garbage collector
- Concurrency
- Bytecode interpreter
- Linking
- Debugging techniques

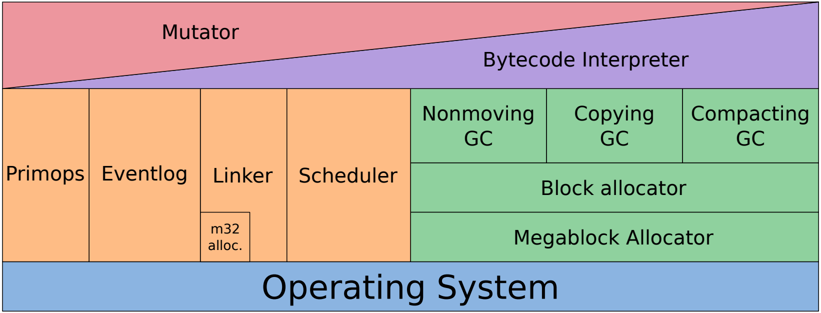
The Big Picture

GHC Overview



Well-Typed

The Runtime System



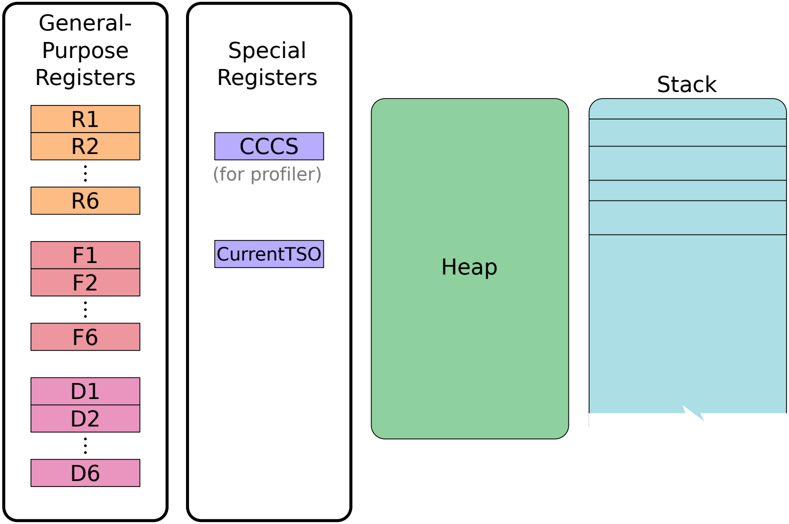
- Provides a multitude of services:
  - Allocation, garbage collection
  - Green threads, sparks
  - Various types and primops: `StableName#`, `StaticPtr#`, `MVar#`
  - `WeakPtr#` and finalization
  - Dynamic code loading
  - Bytecode interpreter
  - Exceptions & stack unwinding
  - STM, . . .

Well-Typed

The GHC/Haskell Execution Model

Abstract machine

A refinement of the **STG Machine** from [7].



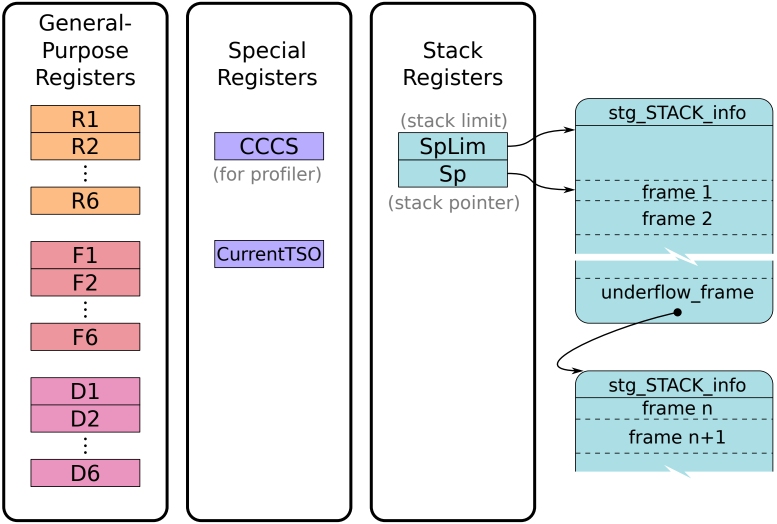
The Stack

Stack usage

- Excess argument passing
- Excess result passing
- Continuation tracking
- Tracking thunk updates
- Exception handling

Abstract machine (stack representation)

A refinement of the **STG Machine** from [7].



Well-Typed

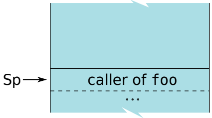
Example: Function calls and case analysis

```
foo = \a b ->
  case f a of x { _ -> g x b }
```

Well-Typed

Example: Function calls and case analysis

```
foo = \a b ->
  case f a of x { _ -> g x b }
```



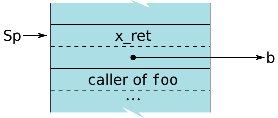
Will be lowered to

```
foo() {
  StgPtr a=R1, b=R2;

  // Push return frame
  Sp = Sp - 2;
  Sp(0) = x_ret;
  Sp(1) = b;
  // Enter scrutinee
  R1 = a;
  call f;
}
```

```
x_ret() {
  StgPtr x = R1;
  StrPtr b = Sp(1)
  Sp = Sp + 2;
  R1 = x; R2 = b;
  call g;
}
```

Example: Function calls and case analysis

```
foo = \a b ->
  case f a of x { _ -> g x b }
```



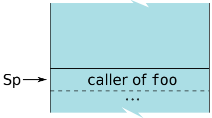
Will be lowered to

```
foo() {
  StgPtr a=R1, b=R2;

  // Push return frame
  Sp = Sp - 2;
  Sp(0) = x_ret;
  Sp(1) = b;
  // Enter scrutinee
  R1 = a;
  call f;
}
```

```
x_ret() {
  StgPtr x = R1;
  StrPtr b = Sp(1)
  Sp = Sp + 2;
  R1 = x; R2 = b;
  call g;
}
```

Example: Function calls and case analysis

```
foo = \a b ->
  case f a of x { _ -> g x b }
```



Will be lowered to

```
foo() {
  StgPtr a=R1, b=R2;

  // Push return frame
  Sp = Sp - 2;
  Sp(0) = x_ret;
  Sp(1) = b;
  // Enter scrutinee
  R1 = a;
  call f;
}
```

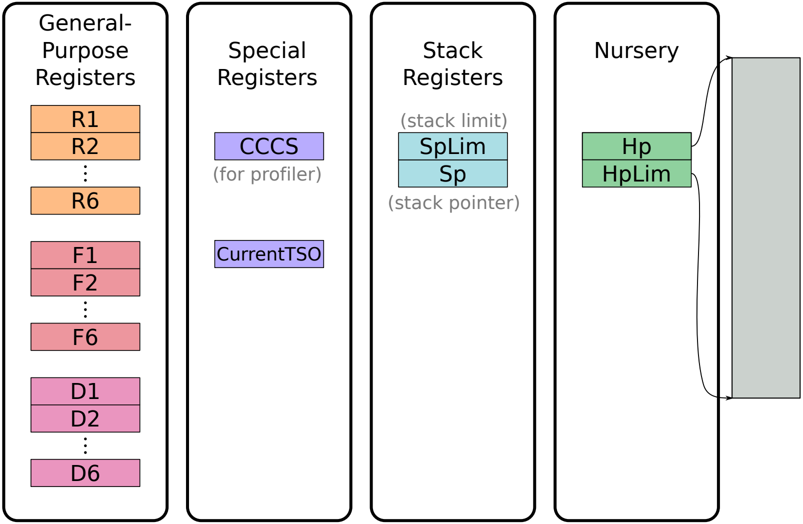
```
x_ret() {
  StgPtr x = R1;
  StrPtr b = Sp(1)
  Sp = Sp + 2;
  R1 = x; R2 = b;
  call g;
}
```

Well-Typed

The Heap

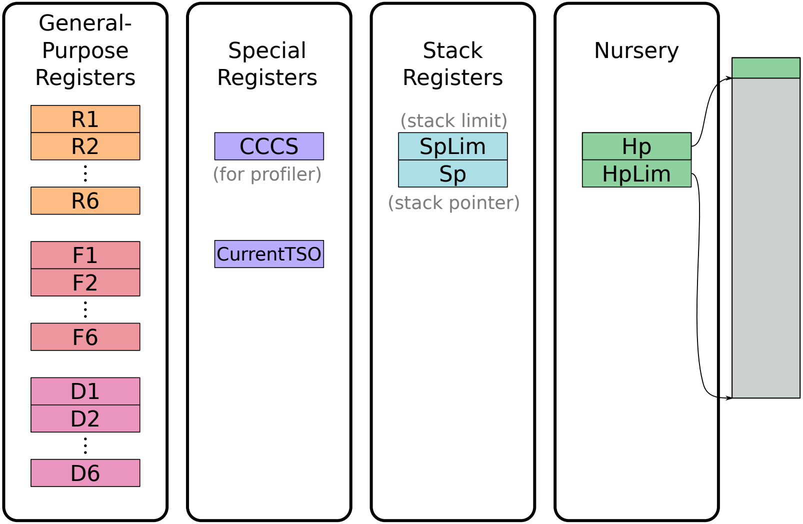
Abstract machine (heap representation)

A refinement of the **STG Machine** from [7].



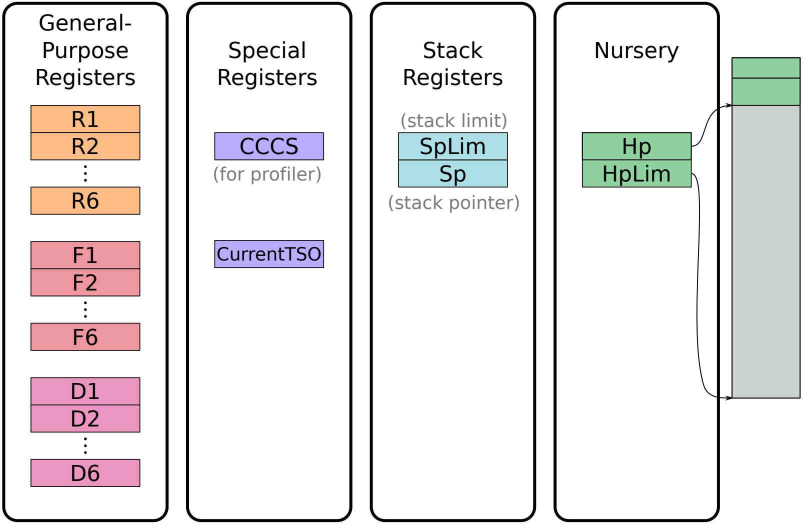
Abstract machine (heap representation)

A refinement of the **STG Machine** from [7].



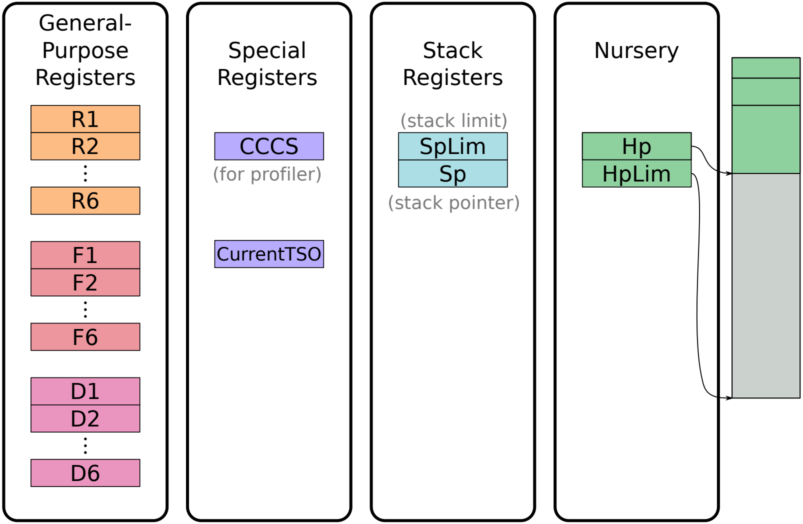
Abstract machine (heap representation)

A refinement of the **STG Machine** from [7].



Abstract machine (heap representation)

A refinement of the **STG Machine** from [7].



Well-Typed

Theme: Nearly everything is a heap object

- Threads (`StgTSO`)
- Stacks (`StgStack`)
- Messages (`Message`)
- Bytecode objects (`StgBCO`)
- STM transactions (`StgTRecHeader`, `StgTVarWatchQueue`)
- Compact regions (`StgCompactNFData`)

Well-Typed

Heap Objects (closures)

From `rts/include/rts/storage/Closures.h`:

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

Heap Objects (closures)

From `rts/include/rts/storage/Closures.h`:

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

// Closure header
typedef struct {
    const StgInfoTable*   info;
#if defined(PROFILING)
    StgProfHeader         prof;
#endif
} StgHeader;
```

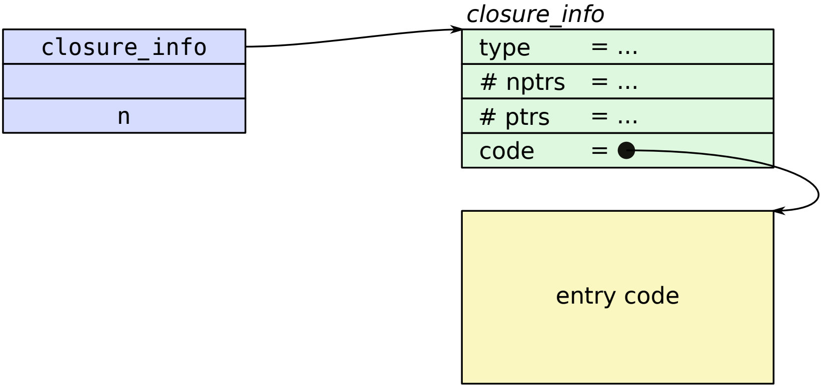
Info Tables

The structure of a closure is described by its **info table**:

- closure type (e.g. constructor, `Weak#`, thunk, indirection)
- payload layout
- function arity
- entry code
- for thunks and functions: pointer to static reference table (SRT)

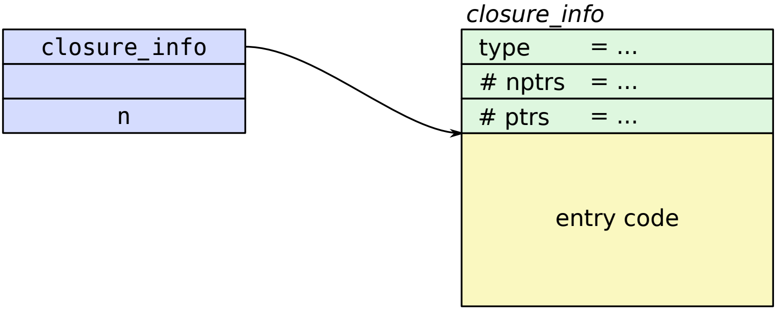
See definition of `StgInfoTable` in
`rts/include/rts/storage/InfoTables.h`.

Entry Code: Naive model



Well-Typed

Entry Code: Tables-next-to-code

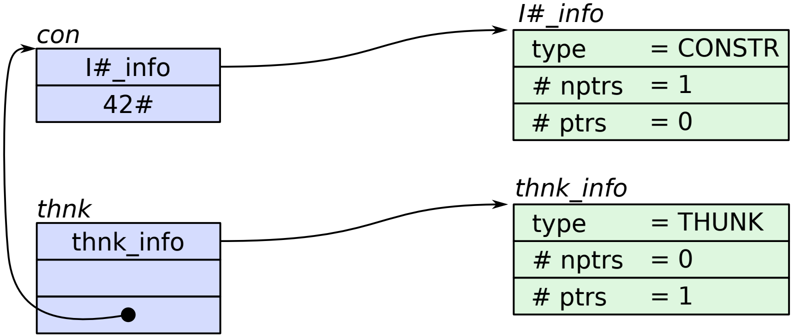


Heap Objects: Some Examples

```
let con  = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel  = fst pair
in ...
```

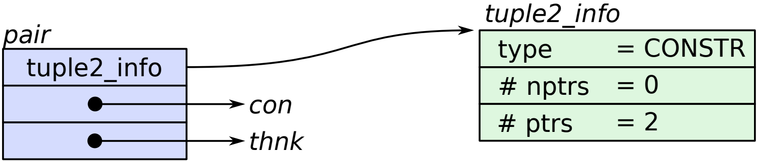
Heap Objects: Some Examples

```
let con  = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel  = fst pair
in ...
```



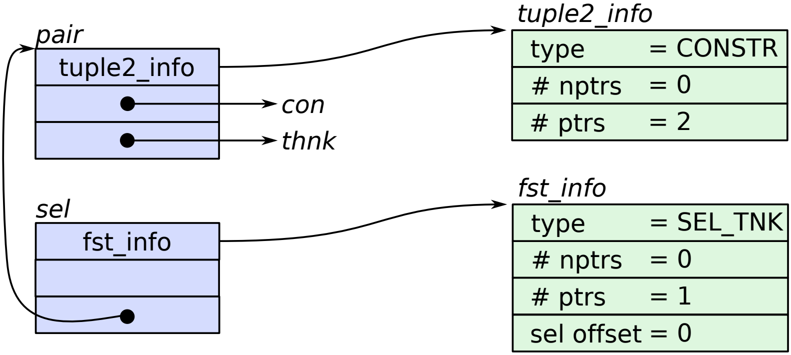
Heap Objects: Some Examples

```
let con  = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel  = fst pair
in ...
```



Heap Objects: Some Examples

```
let con  = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel  = fst pair
in ...
```



Partial Applications

Consider an undersaturated function application:

```
ap :: (a -> b -> c) -> a -> (b -> c)
ap f x = f x
```

This will compile to

```
{
  StgPtr f = R2;
  StgPtr x = R3;
  R2 = x;
  R1 = f;
  call stg_ap_p_fast(R2, R1)
      args: 8, res: 0, upd: 8;
}
```

Partial Applications

`stg_ap_p_fast` is an **application function**. These are generated for various call patterns by `utils/genapply`.

This function will:

1. Inspect the closure type of the applied function
2. Determine whether the given number of arguments has saturated the function
   - If so, call the function
   - If not, allocate a PAP closure

See `_build/stage1/rts/build/cmm/AutoApply.cmm`

Partial Applications

Applying one argument to an unknown arity-3 function:

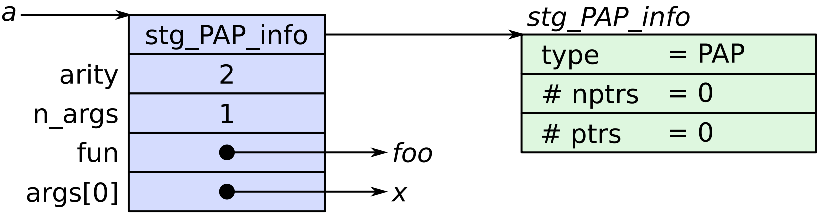
```
foo :: a -> b -> c -> d

a = foo x
```

Partial Applications

Applying one argument to an unknown arity-3 function:

```
foo :: a -> b -> c -> d

a = foo x
```

Will give rise to



Closure Types: Haskell Constructs

Closure type	Description
`CONSTR`	A saturated data constructor application. `x = Just y`
`FUN`	A function. `f = \x -> ...`
`THUNK`	A thunk `x = fib 42`
`THUNK_SELECTOR`	A selector thunk `x = fst pair`
`AP`	A saturated function application.
`PAP`	A partially-applied function application. `z = compare x`
`WEAK`	A `Weak#`
`CONTINUATION`	A `Continuation#`

Well-Typed

Closure Types: Arrays and mutable variables

Closure type	Description
MUT_VAR †	A MutVar# (i.e. IORef or STRef).
MVAR †	An MVar#.
TVAR	An TVar#.
ARR_WORDS	A ByteArray#.
MUT_ARR_PTRS †	An MutableArray#
MUT_ARR_PTRS_FROZEN †	An Array#
SMALL_MUT_ARR_PTRS †	An MutableSmallArray#
SMALL_MUT_ARR_PTRS_FROZEN †	An SmallArray#

† denotes that the type has _CLEAN and _DIRTY variants.

Closure Types: Book-keeping

Closure type	Description
AP_STACK	A computation suspended due to thrown exception.
IND	An indirection.
BCO	A byte-code object
BLACKHOLE	A thunk which is currently under evaluation.
BLOCKING_QUEUE	Records that a thread is blocked on a blackhole.
TSO	An thread state object.
STACK	An thread stack chunk.
WHITEHOLE	A general placeholder used for synchronization.

Case study: Thunk allocation and entry

To see how these pieces fit together, consider the following program:

```haskell
-- examples/thunk.hs

foo :: Int -> Solo Int
foo n =
    let thnk = fib n
    in Solo thnk
```

Let’s trace the execution of an entry into `foo` and then `thnk`…

Case study: Thunk allocation and entry (Core)

```
-- ghc examples/thunk.hs -ddump-simpl

foo :: Int -> Solo Int
[...]
foo = \ (n_aCE :: Int) -> Solo (fib n_aCE)
```

Well-Typed

Background: Reading STG syntax

Core | **STG**

function binding

```
let
  foo :: Int -> Int
  foo = \x -> rhs
in ...
```

```
let
  foo :: Int -> Int =
    \r [x] rhs
in ...
```

arguments

update flag
r ≡ "reentrant"

updateable thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \u [] bar 42
in ...
```

free variable list

u ≡ "updatable"

single-entry (non-updatable) thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \s [] bar 42
in ...
```

s ≡ "single-entry"

Background: Reading STG syntax

Core | **STG**

function binding

```
let
  foo :: Int -> Int
  foo = \x -> rhs
in ...
```

```
let
  foo :: Int -> Int =
    \r [x] rhs
in ...
```

arguments

update flag
r ≡ "reentrant"

updateable thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \u [] bar 42
in ...
```

free variable list

u ≡ "updatable"

single-entry (non-updatable) thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \s [] bar 42
in ...
```

s ≡ "single-entry"

Well-Typed

Background: Reading STG syntax

Core | **STG**

function binding

```
let
  foo :: Int -> Int
  foo = \x -> rhs
in ...
```

```
let
  foo :: Int -> Int =
    \r [x] rhs
in ...
```

arguments

update flag
r ≡ "reentrant"

updateable thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \u [] bar 42
in ...
```

free variable list

u ≡ "updatable"

single-entry (non-updatable) thunk

```
let
  foo :: Int
  foo = bar 42
in ...
```

```
let
  foo :: Int =
    {bar} \s [] bar 42
in ...
```

s ≡ "single-entry"

Well-Typed

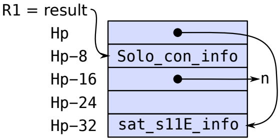
Case study: Thunk allocation (STG)

```
-- ghc examples/thunk.hs -ddump-stg-final

Hi.foo :: GHC.Types.Int -> Solo GHC.Types.Int
[GblId, Arity=1, Str=<MP(ML)>, Cpr=1, Unf=OtherCon []] =
    \r [n_s11D]
        let {
          sat_s11E [Occ=Once1] :: GHC.Types.Int
          [LclId] =
              \u [] Hi.fib n_s11D;
        } in  Solo [sat_s11E];
```

Case study: Thunk allocation (Cmm)

```
// ghc examples/thunk.hs -ddump-opt-cmm

Hi.foo_entry() // [R2]
{
  c12S:
      // N.B. R2 is the first argument to `foo`
      Hp = Hp + 40;

      // Heap check:
      if (Hp > HpLim) (likely: False) {
        goto heap_chk_failed;
      } else {
        goto heap_chk_ok;
      }

  heap_chk_failed:
      HpAlloc = 40;
      R1 = Hi.foo_closure;
      call (I64[BaseReg - 8])(R2, R1)
          args: 8, res: 0, upd: 8;

  heap_chk_ok:
      I64[Hp - 32] = sat_s11E_info;
      P64[Hp - 16] = R2;
      I64[Hp - 8] = Solo_con_info;
      P64[Hp] = Hp - 32;
      R1 = Hp - 7; // due to pointer tagging
      call (P64[Sp])(R1)
          args: 8, res: 0, upd: 8;
}
```

```
-- ghc examples/thunk.hs -ddump-stg-final

Hi.foo :: GHC.Types.Int -> Solo GHC.Types.Int
[...] =
  \r [n_s11D]
      let {
        sat_s11E [Occ=Once1] :: GHC.Types.Int
        [LclId] =
            \u [] Hi.fib n_s11D;
      } in  Solo [sat_s11E];
```



Case study: Thunk entry

Recall our example program:

```
foo :: Int -> Solo Int
foo n =
    let thnk = fib n
    in Solo thnk
```

… where the STG was:

```
Hi.foo :: Int -> Solo Int =
  \r [n_s11D]
    let {
      sat_s11E [Occ=Once1]
        :: GHC.Types.Int =
          \u [] Hi.fib n_s11D;
    } in  Solo [sat_s11E];
```

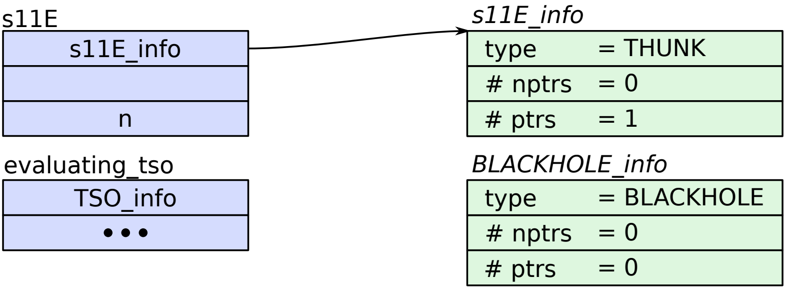
Well-Typed

Case study: Thunk entry (Cmm)

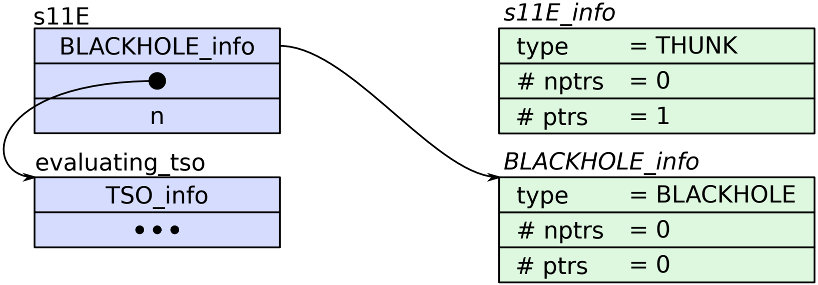
```
// ghc examples/thunk.hs -ddump-opt-cmm

sat_s11E_entry() { //  [R1]
  c12O:
      // N.B. on entry R1 is the address of `thnk`

      // Stack check:
      if ((Sp + -16) < SpLim) (likely: False) {
        goto stack_chk_failed;
      } else {
        goto stack_chk_ok;
      }

  stack_chk_failed:
      call (I64[BaseReg - 16])(R1)
          args: 8, res: 0, upd: 8;

  stack_chk_ok:
      // Push update frame
      I64[Sp - 16] = stg_upd_frame_info;
      P64[Sp - 8] = R1;
      Sp = Sp - 16;

      // Setup call to `fib`
      R2 = P64[R1 + 16];  // === n
      call Hi.fib_info(R2)
          args: 24, res: 0, upd: 24;
}
```

Well-Typed

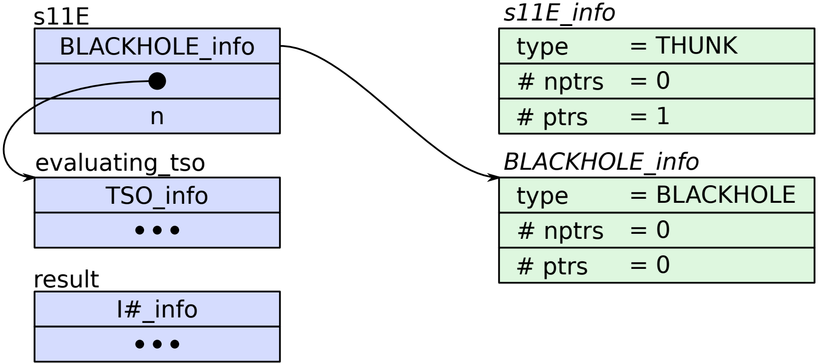
Case study: Thunk update



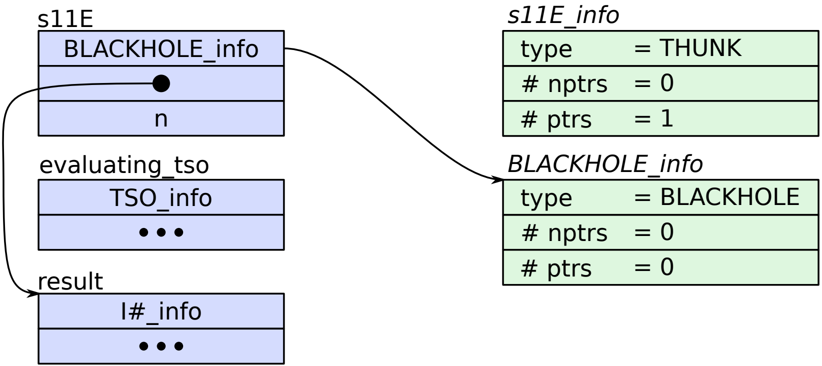
Case study: Thunk update



Case study: Thunk update



Case study: Thunk update



The Storage Manager

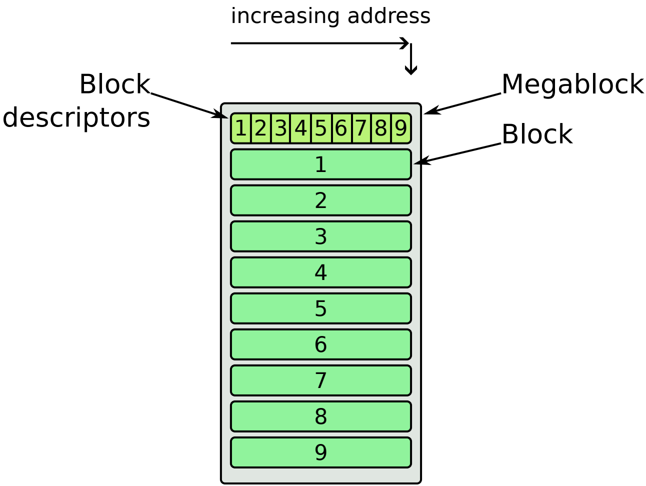
Storage management

Requirements:

- Incremental address-space commit
- Allocation, freeing, and reuse
- Efficient membership query
- $O(1)$ lookup of metadata by address
- NUMA-domain awareness

Block allocator

GHC bases its storage manger on a block allocator [5]



Block descriptor

```c
// From rts/include/rts/storage/Block.h

typedef struct bdescr_ {
    StgPtr start;                 // [READ ONLY] start addr of block
    union {
        StgPtr free;              // First free byte of block
        struct NonmovingSegmentInfo nonmoving_segment;
    };
    struct bdescr_ *link;         // used for chaining blocks together
    union {
        struct bdescr_ *back;     // sometimes used for doubly-linked lists
        StgWord *bitmap;          // bitmap for mark/compact GC
        StgPtr  scan;             // scan pointer for copying GC
    } u;
    struct generation_ *gen;      // generation
    StgWord16 gen_no;             // gen->no, cached
    StgWord16 dest_no;            // number of destination generation
    StgWord16 node;               // which NUMA node does this block live?
    StgWord16 flags;              // block flags, see below
    StgWord32 blocks;             // [READ ONLY] no. of blocks in a group
} bdescr;
```

Well-Typed

Mutator Allocation

Each STG machine is allocated a nursery by the GC (`Storage.c:resetNurseries`):

```
typedef struct nursery_ {
    bdescr *        blocks;
    memcount        n_blocks;
} nursery;
```

`blocks` is a chain of free blocks which the mutator will allocate into in bump-pointer manner.

Exception: Arrays are allocated via `Storage.c:allocate` or `Storage.c:allocatePinned`.

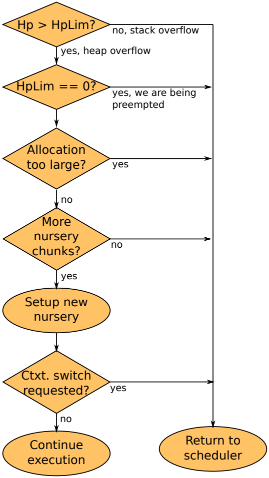
Mutator Allocation: Heap Check

Each function which allocates is responsible for performing a heap check:

```
Hp = Hp + bytes_needed;
if (Hp > HpLim) {
    // jump to GC
} else {
    // proceed...
}
```

Mutator Allocation: Heap Check

If the heap check fails we end up in `stg_gc_noregs` (`HeapStackCheck.cmm`).



From the scheduler, control passes to `Schedule.c:scheduleDoGC` and finally `GC.c:GarbageCollect`.

Well-Typed

Threading and Concurrency

Threading

GHC/Haskell provides threads with an $M : N$ threading model.

Supports “bound” threads (e.g. `forkOS`).

Threading

GHC/Haskell provides threads with an $M : N$ threading model.

Supports "bound" threads (e.g. `forkOS`).

Two principle abstractions:

- **Task**: An OS thread used for Haskell execution.
- **Capability**: A Haskell execution context.

Threading

GHC/Haskell provides threads with an $M : N$ threading model.

Supports "bound" threads (e.g. `forkOS`).

Two principle abstractions:

- **Task**: An OS thread used for Haskell execution.
- **Capability**: A Haskell execution context.

There are a fixed number of capabilities in a program; set by:

- passing `+RTS -N<n>` on the command-line, or
- calling `Control.Concurrent.setNumCapabilities`

Capability: Basics

```c
// From rts/Capability.h

struct Capability_ {
    ...
    StgRegTable r;  // STG machine registers

    uint32_t no;    // capability number.

    // The NUMA node on which this capability resides.
    uint32_t node;

    // true if this Capability is currently running Haskell
    bool in_haskell;
    ...
```

Capability: Ownership

```
// From rts/Capability.h

struct Capability_ {
    ...

    // The Task currently holding this Capability.
    Task *running_task;

    Mutex lock;
    ...
```

Each capability may be **owned** by a task, implying exclusive access to most of its fields.

Capabilities are acquired and released with

```
void releaseCapability  (Capability* cap);
void waitForCapability (Capability **cap, Task *task);
```

Capability: The run queue

```c
// From rts/Capability.h

struct Capability_ {
    ...

    // The queue of Haskell threads waiting to run
    // on the capability.
    StgTSO *run_queue_hd;
    StgTSO *run_queue_tl;
    uint32_t n_run_queue;

    ...
```

Capability: GC bits

```
// From rts/Capability.h

struct Capability_ {
    ...

    // Various remembered sets for the GCs
    bdescr **mut_lists, **saved_mut_lists;
    UpdRemSet upd_rem_set;

    ...
```

Well-Typed

Capability: Allocation areas

```
// From rts/Capability.h

struct Capability_ {
    ...

    // Array of current segments for the non-moving collector.
    // Of length NONMOVING_ALLOCA_CNT.
    struct NonmovingSegment **current_segments;

    // block for allocating pinned objects into
    bdescr *pinned_object_block;
    // full pinned object blocks allocated since the last GC
    bdescr *pinned_object_blocks;
    // empty pinned object blocks, to be allocated into
    bdescr *pinned_object_empty;

    ...
```

Capability: Context switch flags

```c
// From rts/Capability.h

struct Capability_ {
    // Context switch flag.  When non-zero, this means:
    // stop running Haskell code, and switch threads.
    int context_switch;

    // Interrupt flag.  Like the context_switch flag, this als
    // indicates that we should stop running Haskell code
    // but we do *not* switch threads.
    //
    // This is used to stop a Capability in order to do GC,
    // for example.
    int interrupt;

    ...
```

Inter-capability communication: Messages

Capabilities at times need to notify their peers of events:

- `MessageBlackhole`: "I am blocking on a thunk you are currently evaluating"
- `MessageThrowTo`: "I am throwing an asynchronous exception to your thread *t*"

Messages are delivered by setting the recipient `Capability`'s `inbox` field.

Haskell Threads

Each Haskell thread is represented by a **Thread State Object**:

```c
// from rts/include/rts/storage/TSO.h

typedef struct StgTSO_ {
    StgHeader        header;
    StgTSO*          _link;       /* content-dependent list */
    StgTSO*          global_link; /* per-generation list of all threads */
    StgStack*        stackobj;    /* the top of the thread's stack */
    StgWord16        what_next;   /* the thread's run-state */
    StgWord16        why_blocked; /* What is the thread blocked on? */
    StgTSOBlockInfo  block_info;
    StgWord32        flags;
    StgThreadID      id;          /* numeric identifier */
    StgWord32        saved_errno;
    StgWord32        dirty;       /* non-zero => dirty */
    InCall*          bound;       /* is the thread bound to a task? */
    Capability*      cap;         /* owning capability */
    StgTRecHeader*   trec;        /* Active STM transaction */
    StgArrBytes*     label;       /* Thread label */

    /* List of threads blocked on this TSO waiting to throw exceptions. */
    struct MessageThrowTo_ * blocked_exceptions;

    /* Threads blocked on thunks that are under evaluation by this thread. */
    struct StgBlockingQueue_ *bq;

    StgInt64  alloc_limit;     /* Allocation limiit in bytes */

    /* Sum of the sizes of all stack chunks in words */
    StgWord32  tot_stack_size;
} StgTSO;
```

Well-Typed

Scheduling and work-pushing

Thread scheduling is handled by `Schedule.c:schedule`. The threaded RTS's scheduler uses a work-pushing scheme to distribute TSOs to idle capabilities:

- Every scheduler iteration checks whether it has "excess" threads
- If so: look for idle capabilities, move excess to their run queues
- Wake-up target capabilities

Linker

Linker: What?

GHC's RTS includes static runtime linker/loader implementations for:

- COFF (Windows)
- ELF (Linux, BSDs)
- MachO (Darwin)

Goal: Load object files (e.g. `.o` files) and static archives (e.g. `.a` files) for execution.

Linker: Why?

- **Portability**: Dynamic linking implementations tend to vary drastically in what they support; on Windows it’s not supported at all.
- **Performance**: Dynamic linking requires position-independent code which can come at a performance penalty
- **Functionality**: Things like code unloading/reloading are near impossible given the constraints of POSIX/Win32’s interfaces.

Linker: Phases

The primary abstraction of the linker is `ObjectCode`, representing a loaded object file.

Linking begins with a call to `Linker.c:loadObj`.

This proceeds in several phases:

1. Indexing
   - Verify integrity of object (`ocVerifyImage`)
   - enumerate defined symbols (`ocGetNames`)
2. Resolution:
   - Map object contents into address space
   - Resolve and perform relocations (`ocResolve`)
3. Initialization
   - Run static initializers (`ocRunInit`)

After loading, symbols can be resolved to addresses with `Linker.c:lookupSymbol`.

See `Note [runtime-linker-phases]`.

Well-Typed

Linker: Unloading and GC

Objects can be unloaded using `unloadObj`.

When there are objects pending unload the GC will mark reachable `ObjectCodes`.

After GC the linker will unload any unmarked objects.

Linker: Mapping

Linking non-relocatable code is tricky due to, e.g., jump displacement restrictions.

The `m32` allocator is a special-purpose allocator specifically for object-code mappings which manages low-memory for use by the linker.

`m32` also handles memory protection (e.g. W^X)

Bytecode Interpreter

Bytecode Interpreter

Compiling and loading object code is expensive.

For interactive usage we generally prefer bytecode.

- Closures compiled to bytecode take the form of **bytecode objects** (BCOs)
- Stack machine, instruction stream of 16-bit words
- Bytecode documented in `GHC.ByteCode.Instr`
- Interpreter found in `rts/Interpreter.c`

Working on the Runtime System

Code Structure

`rts/linker` The RTS linker; used for dynamic code loading in GHCi

`rts/sm/{MBlock,BlockAlloc}.c` The (mega-)block allocator

`rts/sm/{GC,Evac,Scav}.c` The copying garbage collector

`rts/StgCRun.c` Responsible for transitions between Haskell and C execution.

`rts/{js,posix,wasm,win32}/` Platform-dependent bits

`rts/adjustor` Adjustor thunk implementations (for `foreign exports`)

Header structure

There are two classes of RTS functions:

- **private** symbols, which are declared in `rts/*.h` and are not exposed
- **public** symbols, which are declared in `rts/include/...`

To use the public interface one should `#include <Rts.h>`, not the individual headers in `rts/include`.

The "stable" interface to the RTS appropriate for use by end-users is defined in `rts/include/RtsAPI.h`.

Validating RTS behavior

- Assertions:
  - `ASSERT`s are only asserted in the `DEBUG` runtime
  - `CHECK`s are always asserted
- `valgrind`
  - Sometimes useful for diagnosing C-side leaks
- `ThreadSanitizer`
  - Quite useful for catching data races; see `Note [ThreadSanitizer]` in `rts/includes/rts/TSANUtils.h`.

Observing RTS behavior

- `debugBelch()`: Simple `printf` debugging
- Eventlog (`trace()`): Sometimes more useful than `debugBelch`
- `+RTS -D*` (with `-debug` RTS): Useful tracing output
- `strace`
- gdb
  - `rr`: Time travelling debugging
  - `ghc-utils/gdb`[1]: Useful gdb extensions for inspecting RTS state
  - Always build with `+debug_info` flavour transformer

[1] https://gitlab.haskell.org/bgamari/ghc-utils

Symbol names: Conventions

GHC uses a set of prefixes to identify compiler-generated symbols:

Prefix	Meaning
`$d`	Dictionary
`$f`	Dictionary function
`$w`	Worker function
`$s`	Specialised function
`$m`	Pattern synonym matcher
`$dm`	Default method
`$tc`, `$tr`	Typeable evidence
`D:`	Dictionary data constructor

See `Note [Making system names]`.

Symbol names: Z-encoding

GHC-generated symbol names use a Z-encoding[2] to escape non-alphanumeric characters.

Character	Z-encoding
`.`	`zi`
`+`	`zp`
`_`	`zu`
`h`	`zh`
`$`	`zd`

For instance,

`base_GHCziBase_zpzp_closure`

decodes to

`base_GHC.Base_++_closure`

[2] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/symbol-names

Recommended Reading

- "Mathematizing C++ Concurrency" [1]: Concurrency and memory
- "Runtime Support for Multicore Haskell" [6]
- "Haskell on a Shared-Memory Multiprocessor" [4]
- "Composable Memory Transactions" [3]: STM
- "A Concurrent Garbage Collector for the Glasgow Haskell Compiler" [2]
- Pointer tagging

Appendix

References

[1] Batty, M. et al. 2011. Mathematizing c++ concurrency[3]. **Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages** (New York, NY, USA, 2011), 55–66.

[2] Gamari, B. and Dietz, L. 2020. Alligator collector: A latency-optimized garbage collector for functional programming languages[4]. **Proceedings of the 2020 ACM SIGPLAN international symposium on memory management** (New York, NY, USA, 2020), 87–99.

[3] Harris, T. et al. 2008. Composable memory transactions. **Commun. ACM**. 51, 8 (Aug. 2008), 91–100. DOI:https://doi.org/10.1145/1378704.1378725[5].

[4] Harris, T. et al. 2005. Haskell on a shared-memory multiprocessor[6]. **Proceedings of the 2005 ACM SIGPLAN workshop on haskell** (New York, NY, USA, 2005), 49–61.

[5] Marlow, S. et al. 2008. Parallel generational-copying garbage collection with a block-structured heap[7]. (2008), 11–20.

[6] Marlow, S. et al. 2009. Runtime support for multicore haskell[8]. **Proceedings of the 14th ACM SIGPLAN international conference on functional programming** (New York, NY, USA, 2009), 65–78.

[7] Peyton Jones, S.L. 1992. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. **Journal of Functional Programming**. 2, 2 (1992), 127–202. DOI:https://doi.org/10.1017/S0956796800000319[9].

---

[3] https://doi.org/10.1145/1926385.1926394

[4] https://doi.org/10.1145/3381898.3397214

[5] https://doi.org/10.1145/1378704.1378725

[6] https://doi.org/10.1145/1088348.1088354

[7] https://doi.org/%2010.1145/1375634.1375637%20

[8] https://doi.org/10.1145/1596550.1596563

[9] https://doi.org/10.1017/S0956796800000319