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

Legend
- Covered in Workshop
- Compilation Pass
- RTS Subsystem
- Compilation Product
- External tool

Compilation Pipeline:
1. Parser
2. Renamer
3. Typechecker
4. Desugar
5. Core-to-Core Pipeline
6. Core-to-STG
7. STG-to-STG Pipeline
8. STG-to-Cmm Pipeline
9. Cmm-to-Cmm Pipeline
10. Cmm-to-Asm Pipeline
11. STG-to-Bytecode Pipeline
12. Cmm-to-LLVM Pipeline
13. STG-to-JavaScript Pipeline
14. Cmm-to-C Pipeline
15. Assembler
16. RTS Bytecode Interpreter
17. cc

Conversion Stages:
- Source to Core: Core-to-Core Pipeline
- Core to STG: Core-to-STG Pipeline
- STG to Cmm: STG-to-Cmm Pipeline
- Cmm to Asm: Cmm-to-Asm Pipeline
- Cmm to LLVM: Cmm-to-LLVM Pipeline
- STG to Bytecode: STG-to-Bytecode Pipeline
- STG to JavaScript: STG-to-JavaScript Pipeline
- Cmm to C: Cmm-to-C Pipeline

Conversion Tools:
- Cmm-to-C
- Assembler
- llc
- opt
- cc
- Well-Typed

RTS Subsystem:
- Compilation Pass
- Compilation Product
- External tool

Covered in Workshop:
- Parser
- Renamer
- Typechecker
- Desugar
- Core-to-Core Pipeline
- Core-to-STG
- STG-to-STG Pipeline
- STG-to-Cmm Pipeline
- Cmm-to-Cmm Pipeline
- Cmm-to-Asm Pipeline
- STG-to-Bytecode Pipeline
- Cmm-to-LLVM Pipeline
- STG-to-JavaScript Pipeline
- Cmm-to-C Pipeline
- Assembler
- RTS Bytecode Interpreter
- llc
- opt
- cc

Legend:
- Compilation Pass
- RTS Subsystem
- Compilation Product
- External tool
- Covered in Workshop
- Compilation Product
- Well-Typed
The Runtime System

<table>
<thead>
<tr>
<th>Mutator</th>
<th>Bytecode Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primops</td>
<td>Eventlog</td>
</tr>
<tr>
<td>Linker</td>
<td>Scheduler</td>
</tr>
<tr>
<td>m32 alloc.</td>
<td></td>
</tr>
</tbody>
</table>

### Operating 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, …

- Nonmoving GC
- Copying GC
- Compacting GC

- Block allocator
- Megablock Allocator

Well-Typed
The GHC/Haskell Execution Model
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].

- **General-Purpose Registers**
  - R1
  - R2
  - \ldots
  - R6
  - F1
  - F2
  - \ldots
  - F6
  - D1
  - D2
  - \ldots
  - D6

- **Special Registers**
  - CCCS (for profiler)
  - CurrentTSO

- **Stack Registers**
  - Sp
  - SpLim
  - stg_STACK_info
  - frame 1
  - frame 2
  - underflow_frame
  - frame n
  - frame n+1

Well-Typed
Example: Function calls and case analysis

```haskell
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 ->

\textbf{case} f a \textbf{of} x \{ \_ -> g x b \}

Will be lowered to

\begin{verbatim}
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;
}
\end{verbatim}

\begin{verbatim}
x_ret() {
    StgPtr x = R1;
    StrPtr b = Sp(1)
    Sp = Sp + 2;
    R1 = x; R2 = b;
    call g;
}
\end{verbatim}
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;
The Heap
Abstract machine (heap representation)

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

**General-Purpose Registers**
- R1
- R2
- \(\vdots\)
- R6
- F1
- F2
- \(\vdots\)
- F6
- D1
- D2
- \(\vdots\)
- D6

**Special Registers**
- CCCS (for profiler)
- CurrentTSO

**Stack Registers**
- SpLim (stack limit)
- Sp (stack pointer)

**Nursery**
- Hp
- HpLim

Well-Typed
Abstract machine (heap representation)

A refinement of the STG Machine from [7].

- General-Purpose Registers: R1, R2, ..., R6, F1, F2, ..., F6, D1, D2, ..., D6
- Special Registers: CCCS (for profiler), CurrentTSO
- Stack Registers: (stack limit) SpLim, (stack pointer) Sp
- Nursery: Hp, HpLim

Well-Typed
Abstract machine (heap representation)

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

- **General-Purpose Registers**
  - R1
  - R2
  - ... R6
  - F1
  - F2
  - ... F6
  - D1
  - D2
  - ... D6

- **Special Registers**
  - CCCS (for profiler)
  - CurrentTSO

- **Stack Registers**
  - Sp
  - SpLim
  - (stack limit)
  - (stack pointer)

- **Nursery**
  - Hp
  - HpLim

Well-Typed
Abstract machine (heap representation)

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

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>Special Registers</th>
<th>Stack Registers (stack limit)</th>
<th>Nursery</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>CCCS (for profiler)</td>
<td>SpLim</td>
<td>Hp</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>Sp</td>
<td>HpLim</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Theme: Nearly everything is a heap object

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

```
# closure_info
# nptrs = ...
type = ...
# ptrs = ...
entry code
closure_info
n
code =
```

Well-Typed
Well-Typed

**Entry Code: Tables-next-to-code**

```
<table>
<thead>
<tr>
<th>closure_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
</tbody>
</table>

```

```
closure_info
# nptrs = ...
type = ...
# ptrs = ...
entry code
closure_info
n
```

Well-Typed
let con = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel  = fst pair

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

Well-Typed
let con = I# 42#
    thnk = foo con
    pair = (con, thnk)
    sel = fst pair

in ...

```
<table>
<thead>
<tr>
<th>tuple2_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>con</td>
</tr>
<tr>
<td>thnk</td>
</tr>
</tbody>
</table>
```

```
tuple2_info

<table>
<thead>
<tr>
<th>type</th>
<th>CONSTR</th>
</tr>
</thead>
<tbody>
<tr>
<td># nptrs</td>
<td>0</td>
</tr>
<tr>
<td># ptrs</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Heap Objects: Some Examples

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

in ...
```

Diagram:
- `pair` contains two pointers: `con` and `thnk`.
- `sel` contains a single pointer to another node.

**tuple2_info**:
- Type: CONSTR
- # nptrs: 0
- # ptrs: 2

**fst_info**:
- Type: SEL_TNK
- # nptrs: 0
- # ptrs: 1
- sel offset: 0

Well-Typed
Partial Applications

Consider an undersaturated function application:

\[ ap :: (a \to b \to c) \to a \to (b \to c) \]
\[ ap \; f \; x = f \; x \]

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;
\}
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:

\[ \text{foo} :: a \rightarrow b \rightarrow c \rightarrow d \]

\[ a = \text{foo} \ x \]

Will give rise to

![Diagram showing partial application]

- **arity**: 2
- **n_args**: 1
- **fun**: foo
- **args[0]**: x

**stg_PAP_info**

- `type = PAP`
- `# nptrs = 0`
- `# ptrs = 0`
## Closure Types: Haskell Constructs

<table>
<thead>
<tr>
<th>Closure type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTR</td>
<td>A saturated data constructor application.</td>
</tr>
<tr>
<td></td>
<td>(x = \text{Just } y)</td>
</tr>
<tr>
<td>FUN</td>
<td>A function.</td>
</tr>
<tr>
<td></td>
<td>(f = \lambda x \rightarrow \ldots)</td>
</tr>
<tr>
<td>THUNK</td>
<td>A thunk</td>
</tr>
<tr>
<td></td>
<td>(x = \text{fib } 42)</td>
</tr>
<tr>
<td>THUNK_SELECTOR</td>
<td>A selector thunk</td>
</tr>
<tr>
<td></td>
<td>(x = \text{fst } \text{pair})</td>
</tr>
<tr>
<td>AP</td>
<td>A saturated function application.</td>
</tr>
<tr>
<td></td>
<td>(z = \text{compare } x)</td>
</tr>
<tr>
<td>PAP</td>
<td>A partially-applied function application.</td>
</tr>
<tr>
<td>WEAK</td>
<td>A Weak#</td>
</tr>
<tr>
<td>CONTINUATION</td>
<td>A Continuation#</td>
</tr>
</tbody>
</table>
## Closure Types: Arrays and mutable variables

<table>
<thead>
<tr>
<th>Closure type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUT_VAR †</td>
<td>A MutVar# (i.e. IORef or STRef).</td>
</tr>
<tr>
<td>MVAR †</td>
<td>An MVar#.</td>
</tr>
<tr>
<td>TVAR</td>
<td>An TVar#.</td>
</tr>
<tr>
<td>ARR_WORDS</td>
<td>A ByteArray#.</td>
</tr>
<tr>
<td>MUT_ARR_PTRS †</td>
<td>An MutableArray#</td>
</tr>
<tr>
<td>MUT_ARR_PTRS_FROZEN †</td>
<td>An Array#</td>
</tr>
<tr>
<td>SMALL_MUT_ARR_PTRS †</td>
<td>AnMutableSmallArray#</td>
</tr>
<tr>
<td>SMALL_MUT_ARR_PTRS_FROZEN †</td>
<td>An SmallArray#</td>
</tr>
</tbody>
</table>

† denotes that the type has _CLEAN and _DIRTY variants.
<table>
<thead>
<tr>
<th>Closure type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP_STACK</td>
<td>A computation suspended due to thrown exception.</td>
</tr>
<tr>
<td>IND</td>
<td>An indirection.</td>
</tr>
<tr>
<td>BCO</td>
<td>A byte-code object</td>
</tr>
<tr>
<td>BLACKHOLE</td>
<td>A thunk which is currently under evaluation.</td>
</tr>
<tr>
<td>BLOCKING_QUEUE</td>
<td>Records that a thread is blocked on a blackhole.</td>
</tr>
<tr>
<td>TSO</td>
<td>An thread state object</td>
</tr>
<tr>
<td>STACK</td>
<td>An thread stack chunk</td>
</tr>
<tr>
<td>WHITEHOLE</td>
<td>A general placeholder used for synchronization.</td>
</tr>
</tbody>
</table>
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)
Function binding

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

Updateable thunk

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

Single-entry (non-updatable) thunk

```haskell
let
  foo :: Int
  foo = bar 42
in ...
```
Background: Reading STG syntax

Core

function binding

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

updateable thunk

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

single-entry (non-updatable) thunk

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

STG

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

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

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

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

u ≡ "updatable"

s ≡ "single-entry"
Background: Reading STG syntax

Core

function binding
let
  foo :: Int -> Int =
    \[x\] rhs
in ...

updateable thunk
let
  foo :: Int
  foo = bar 42
in ...

single-entry (non-updatable) thunk
let
  foo :: Int
  foo = bar 42
in ...

STG

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

update flag
  r ≡ "reentrant"

free variable list
  \[

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

update flag
  u ≡ "updatable"

s ≡ "single-entry"
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)

```haskell
// 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

```haskell
Hi.foo :: GHC.Types.Int -> Solo GHC.Types.Int
[...] = \ [n_s11D] = \\
    \u [] Hi.fib n_s11D;
} in Solo [sat_s11E];
```

```
Well-Typed
```
Recall our example program:

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

... where the STG was:

```haskell
Hi.foo :: Int -> Solo Int =
  \ [n_s11D]
  let {
    sat_s11E [Occ=Once1] :: GHC.Types.Int = \\
      \u [] Hi.fib n_s11D;
  } in Solo [sat_s11E];
```
// sat_s11E_entry() { // \[R1\] 
  // 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;
}
Case study: Thunk update

```
<table>
<thead>
<tr>
<th>TSO_info</th>
<th>s11E_info</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>evaluating_tso</th>
</tr>
</thead>
<tbody>
<tr>
<td>s11E_info</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>

| # nptrs = 0 |
| type = THUNK|
| # ptrs = 1  |

<table>
<thead>
<tr>
<th>BLACKHOLE_info</th>
</tr>
</thead>
<tbody>
<tr>
<td># nptrs = 0</td>
</tr>
<tr>
<td>type = BLACKHOLE</td>
</tr>
<tr>
<td># ptrs = 0</td>
</tr>
</tbody>
</table>
```

Well-Typed
Case study: Thunk update

Well-Typed
Case study: Thunk update

Well-Typed
Case study: Thunk update

s11E

BLACKHOLE_info
n

evaluating_tso

TSO_info

result

l#_info

BLACKHOLE_info

# nptrs = 0

# ptrs = 1

s11E_info

type = THUNK

# nptrs = 0

# ptrs = 1

BLACKHOLE_info

type = BLACKHOLE

# nptrs = 0

# ptrs = 0

Well-Typed
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
GHC bases its storage manager on a block allocator [5].

Block descriptors

Megablock

Block

Increasing address
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;
Mutator Allocation

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

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

\[
Hp = Hp + \text{bytes\_needed};
\]

\[
\text{if} \ (Hp > HpLim) \{
\quad \text{// jump to GC}
\}
\text{else} \{
\quad \text{// proceed...}
\}
\]
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`.

- **Hp > HpLim?**
  - yes, heap overflow

- **HpLim == 0?**
  - yes, we are being preempted

- **Allocation too large?**
  - yes

- **More nursery chunks?**
  - no

- **Ctxt. switch requested?**
  - yes

- **Continue execution**
  - no

- **Return to scheduler**
Threading and Concurrency
Threading

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

Supports “bound” threads (e.g. forkOS).
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`
struct Capability_ {
    StgRegTable r; // STG machine registers
    uint32_t no; // capability number.
    uint32_t node; // The NUMA node on which this capability resides.
    bool in_haskell;
    ...

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

Capabilities are acquired and released with

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

...
// From rts/Capability.h

struct Capability_ {
    ...

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

    ...

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;

    ...

Well-Typed
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 also
    // 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;
    StgWord32    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 limit in bytes */
    /* Sum of the sizes of all stack chunks in words */
    StgWord32   tot_stack_size;
} StgTSO;
```
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
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.
**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.
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].
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.
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
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)
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

- **Assertion**
  - ASSERTs are only asserted in the DEBUG runtime
  - CHECKs 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/gdb¹`: Useful gdb extensions for inspecting RTS state
- Always build with +debug_info flavour transformer

¹https://gitlab.haskell.org/bgamari/ghc-utils
GHC uses a set of prefixes to identify compiler-generated symbols:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d</td>
<td>Dictionary</td>
</tr>
<tr>
<td>$f</td>
<td>Dictionary function</td>
</tr>
<tr>
<td>$w</td>
<td>Worker function</td>
</tr>
<tr>
<td>$s</td>
<td>Specialised function</td>
</tr>
<tr>
<td>$m</td>
<td>Pattern synonym matcher</td>
</tr>
<tr>
<td>$dm</td>
<td>Default method</td>
</tr>
<tr>
<td>$tc, $tr</td>
<td>Typeable evidence</td>
</tr>
<tr>
<td>D:</td>
<td>Dictionary data constructor</td>
</tr>
</tbody>
</table>

See Note [Making system names].
Symbol names: Z-encoding

GHC-generated symbol names use a Z-encoding\(^2\) to escape non-alphanumeric characters.

<table>
<thead>
<tr>
<th>Character</th>
<th>Z-encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>zi</td>
</tr>
<tr>
<td>+</td>
<td>zp</td>
</tr>
<tr>
<td>_</td>
<td>zu</td>
</tr>
<tr>
<td>h</td>
<td>zh</td>
</tr>
<tr>
<td>$</td>
<td>zd</td>
</tr>
</tbody>
</table>

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


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