Building a new GC for Haskell: Lessons on abstraction and large systems engineering

A report submitted for the course
COMP3740, Project Work in Computing

6 pt research project, S1 2022

By:

Junming Zhao

Supervisors:
Prof. Steve Blackburn
Ben Gamari

Australian National University

School of Computing
College of Engineering and Computer Science (CECS)

ABSTRACT

Many programming languages nowadays are designed with automatic memory management. As modern software systems are getting more complicated, software developers are also increasingly choosing languages with automatic memory management [1] due to design benefits. MMTk is an efficient and generalized memory management framework. Establishing MMTk-programming language binding allows MMTk to provide the programming language with a wide range of memory management plans.

This report describes the motivation behind the MMTk-GHC binding, and presents the exploration pathway of implementing the binding. The project result provides an essential foundation for further binding development. A discussion on the trade-off between performance and modularity is also included in this report.

1. Background

1.1 Memory management

In computing, memory is essential for storing data or program instructions. Memory management refers to a scheme of allocating memory upon request, and reclaiming memory that is no longer needed.

The process of reclaiming memory basically contains two phases: The first phase is to identify garbage (objects that are no longer used in the running program, or “dead” objects); The second phase is to return the occupied memory to be free memory (Figure 1). There can be a third phase to facilitate more efficient memory management, such as rearranging objects in the memory, recording object age, etc.

This reclaiming process is also called garbage collection (GC).

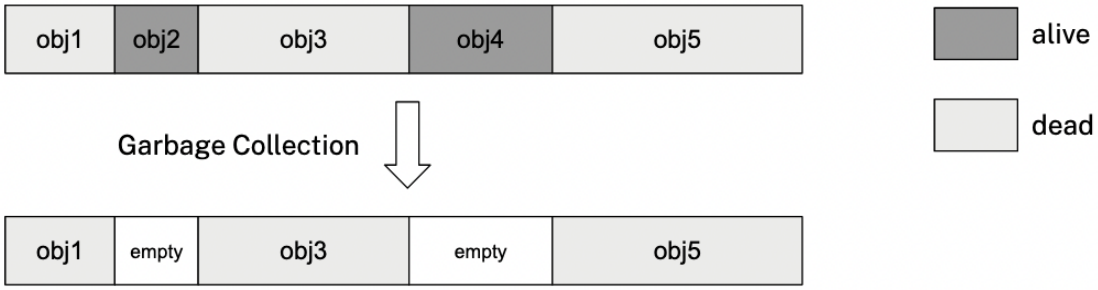

Figure 1. Phase 2 in GC, after identifying garbage

Memory management can be achieved either manually, or automatically. Manual memory management requires the user to explicitly specify the allocation, object destruction and deallocation, such as malloc() and free() in C, and delete() in C++ [2].

In contrast, automatic memory management provides the above functionalities for users, so that users do not have to worry about garbage collection in their programs. Automatic memory management would require a set of data and operations by itself as part of (or as an extension of) the runtime system. Different garbage collectors require different correspondingly different memory management plans.

Many programming languages today provide automatic memory management for users, such as Java and C# (these languages are called “garbage collected languages”). Most of the users (programmers) nowadays also tend to choose garbage collected languages [1] instead of manual memory management.

1.2 MMTk: Memory Management Toolkit

MMTk is a generalized memory management framework, aiming to provide a wide range of high-performance memory management schemes and garbage collectors for different programming languages and runtimes [3].

Connecting MMTk to language runtime will help languages to abstract the memory framework, which benefits languages with succinct architecture and better maintainability. MMTk also provides researchers with a generalized framework to implement novel garbage collectors algorithms, and measure collectors' performance with a multi-runtime platform.

In addition to software design benefits, MMTk collectors also show a higher performance. [1]

MMTk currently has a wide range of collectors including: Copying, mark-sweep, copying generational, hybrid generationals etc. [4]; and supports binding with OpenJDK, JikesRVM, V8 Mu, Ruby and .NET. There are also a number of developing bindings such as Julia, PyPy, etc. [5]

1.3 GHC: The Glasgow Haskell Compiler

Haskell is one of the garbage collected languages. GHC, the Glasgow Haskell Compiler, is the current major, most commonly used compiler for Haskell [6].

GHC has an existing storage manager (GHCSM) that supports a block-structured memory management plan and a number of garbage collectors, including a moving generational collector and concurrent non-moving collector. [7]

1.3.1 Block-structured memory management

Currently GHCSM allocates memory in units of blocks. Each block is associated with a small structure called block descriptor, which records the block information such as its generation, the free pointer, links to previous block and next block, etc.

Compared to conventional contiguous regions of memory, block-structured heap has benefits in flexibility and efficiency: fast memory pool resizing, fast memory recycling and managing large size objects without copying, etc. It also provides a relatively easy extension to support faster parallel copying generational GC. [8]

1.3.2 Generational GC

The current GHCSM performed a generational, copying collector [8]. This means that the heap is divided into several generations, categorized from young to old. Younger generations hold

newer objects; and objects will be promoted to older generations if they survive a few cycles of garbage collections, as illustrated in the below Figure 2.

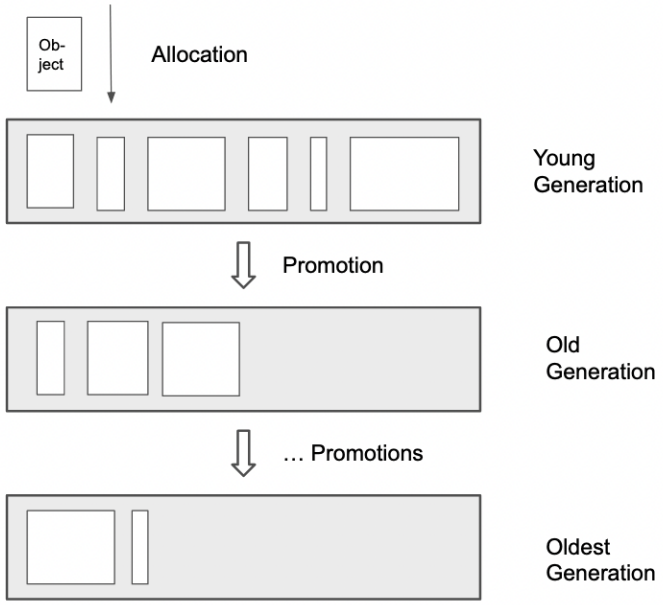

Figure 2. Generational GC:
Objects that survive (i.e. maintain alive, not garbage collected) during GC will be promoted to older generations

Each generation has an associated remembered set, which records all the objects that have pointers pointing towards younger generations. Garbage collection is dependent on the generation age and its remembered set. [9]

2. Introduction

Although currently GHC already has a sophisticated generational collector, it is however highly optimized and integrated into GHC's runtime, making it difficult to switch to a broader range of collectors.

It would be beneficial to generalize GHC's runtime, make it separable from GHCSM, and connect it with a suite of different garbage collections from MMTk, so that users can choose between GHCSM and MMTk for their Haskell programs. A successful MMTk-GHC binding will benefit the Haskell community in at least three ways:

1. Provide GHC with a suite of state-of-the-art garbage collection strategies;
2. Improve the generality of GHC's runtime – the project will need to refactor the runtime to abstract over the differences between MMTK and GHCSM, which will benefit GHC's code in general, even without MMTk;
3. Put in place foundations for a multi-domain garbage collectors in GHC, improving scalability of Haskell programs.

The binding will also benefit MMTk, by:

1. Allow MMTk collectors to be evaluated in a high-performance Haskell runtime;
2. Check the generality of MMTk framework, ensure the framework is compatible with functional languages like Haskell.

However, since GHC is a highly optimized and highly tuned system, any refactoring would potentially affect multiple functionalities of GHC and would require a significant amount of work. Note that the refactoring is common in binding implementation and can be very time consuming due to the language runtime system design [5], as we shall discuss later.

The implementation of the impedance matching code between GHC and MMTk would also require a clear understanding of GHC runtime, which includes more than 50,000 lines of C and C-- code [10]. Therefore this semester project mainly serves as an exploration and introduction to the binding, which involves the below two parts:

- Replace GHC allocation with MMTk allocation;
- Introduce necessary matching code to connect MMTk collectors.

We will discuss each part in more detail.

3. Replace GHC allocation

NoGC is the simplest possible memory management plan, namely just allocating memory without any collection – the name “NoGC” means “no garbage collection”. So in this part, the goal is just to replace every call of GHCSM allocation to call MMTk allocation. Once succeeded, Haskell programs will be able to run as usual with MMTk allocation, as long as there is a large enough heap so that MMTk can keep allocating.

Although NoGC is not an industrially practical plan, it is the very essential first step to connect any further garbage collectors. This means objects’ memory is under MMTk’s control range. After successfully configuring NoGC binding, we can then start to introduce object scanning, write barriers etc., in order for MMTk to arrange and collect the allocated memory.

3.1 Necessity to fix abstraction leakage

However, the NoGC implementation step might be as trivial as it seems, due to the runtime design and complexity. Due to optimization reasons, the memory manager might not be well-encapsulated. For example, the runtime can put assumptions on how the current memory manager works; or use touch the “private data” of the memory manager directly. If the runtime has a strong dependency on its memory manager, then swapping the allocation call can break the system.

Therefore, in order to call MMTk allocation, we need to refactor the runtime, such that it is separable from its memory manager. In the case of V8, the refactoring within V8 required to get a simple NoGC plan working was substantial, touching over 100 files [5].

In the case of GHC, the memory manager is relatively well encapsulated, except a few abstraction leakages:

- The runtime assumes that the memory manager has a block structure, and explicitly uses references to block descriptors.
- The runtime assumes that the memory manager holds a generational garbage collector, and explicitly uses generation.
- The thread state objects (TSO) are recorded and arranged in the memory manager, so the runtime does not explicitly have a direct reference to trace threads except calling its memory manager.

3.2 Abstraction leakage 1

The first and the second abstractions mainly appear in the “capability” data structure, which is an abstraction of the virtual processor. GHC maintains one capability per physical CPU, and each capability contains a set of data that are required to execute Haskell codes, such as the running queue, message queue, worker pool and remembered sets etc.

The problem arises when the capability owns a field of GC remembered sets. Remembered sets essentially belong to the memory manager, and are designed specifically for generational garbage collectors. In the case of GHC, each capability contains a remembered set for every generation, each remembered set contains the objects with pointers pointing towards younger generations.

The main reason behind this design is the locality of remembered sets. A local remembered set can avoid synchronization issues between different capabilities; and also make use of the CPU cache of the current capability. Therefore such design is driven by the runtime efficiency. [11]

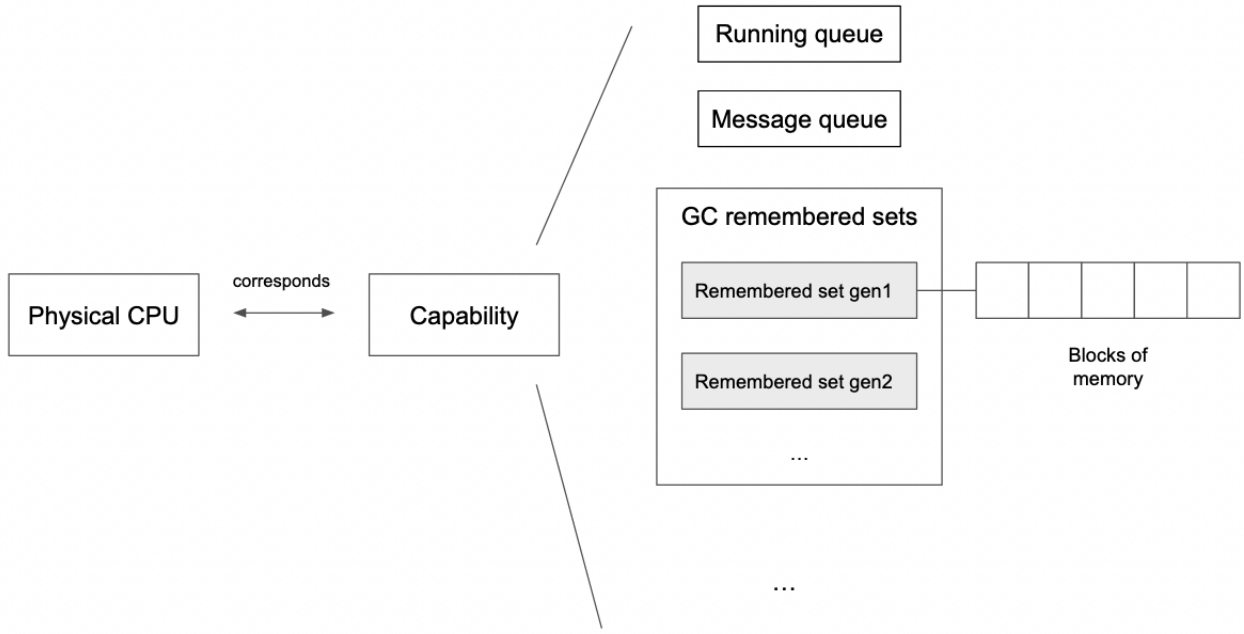

Figure 3. Remembered set for each capability

As illustrated in the above Figure 3, the local remembered sets in GHC capability are using the block structure as well as generation information; and this is a main source of abstraction leakage.

To avoid this abstraction leakage, I proposed a change to store the remembered sets inside the generation structure, as illustrated in the diagram in Figure 4.

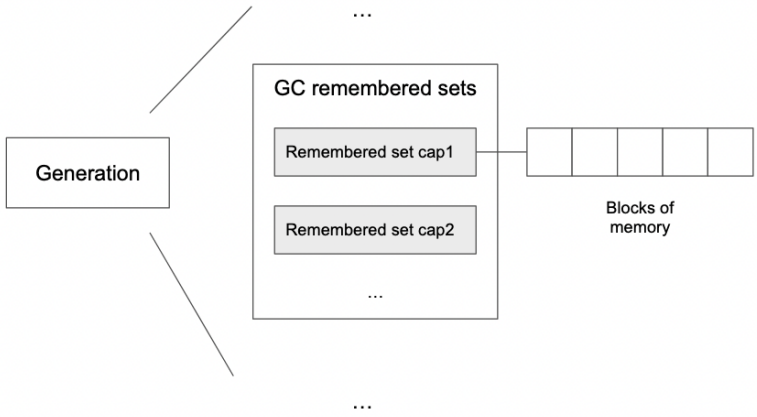

Figure 4. Remembered set for each generation

The proposed change can hide the generation structure inside GHCSM, and ideally should still preserve the locality of the remembered sets for each capability. The code implementation is located at the GitLab repository [12] in branch mmtk/sm2.0. The relative merge request is [13].

3.3 Abstraction leakage 2

There is another major abstraction leakage, namely the TSO arranging issue (as mentioned at the end of section 3.1). TSO, the thread state object, is a representation of the corresponding thread, containing information such as the thread state (running, blocked, finished, etc.), blocked information, and a link to the next TSO etc.

In GHC, TSOs are also heap objects and are garbage collected. To optimize the GC process, each generation holds a list of its TSOs. During GC, the generational garbage collector will detect if a blocked thread is unreachable, and perform corresponding handling. [14] This is also a tacit assumption about the underlying collector's implementation.

To avoid the runtime inspecting inside GHCSM's generations, I proposed a temporary solution to create a global TSO list, shared among all capabilities. GHCSM also needs to handle the global list during GC. The code implementation is located at the GitLab repository [12] in branch mmtk/tso1.0. The modified version of GHC passes most of the GHC built-in testsuite, and discovered a runtime bug of GHC during testing [15].

Note that although such implementation will not be as efficient as the original design, it introduces the correctness and allows the NoGC binding. Optimisation steps can be followed up later once the binding is working.

4. Connect MMTk collectors

After successfully introducing MMTk allocation, the next step would be to connect MMTk collectors.

As discussed in section 1.1, this would require MMTk to be able to identify the garbage objects. Garbage objects are the objects that are no longer referenced in the program – therefore, in order for MMTk to identify the garbage objects, an object tracing function is necessary. Given an object reference as input, the object tracing function should give all the references that the input object contains.

As shown in the below Figure 5, each GHC object has a header (contains object type, size, and other information) and payload (the actual object content). When one object contains another object's reference in its payload, it means that the referrer will likely use the referee later in the program.

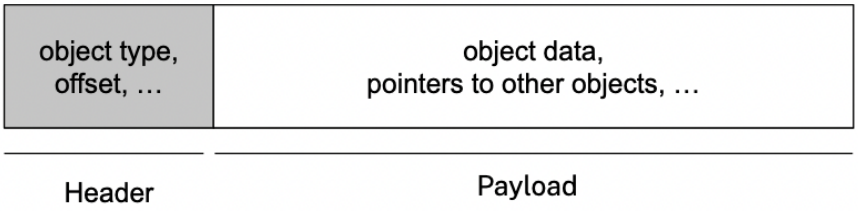

Figure 5. Object's memory structure

To identify pointers' locations, the object tracing function will firstly need to read the object header, which contains object type information. Same type of GHC objects have their pointers residing in the same locations inside its payload, and then we will be able to identify all the pointers. Such design of GHC objects allows us to identify the contained pointers within constant time (O(1)), insteading of scanning through the entire payload (O(n)), if we consider the object size as n.

MMTk will then need to call this object tracing function with some root objects (objects currently on the stack, global variables etc.), and recursively identify all the reachable objects from the roots. The non-reachable objects are then considered as dead, as it will no longer get used in the program, and will be garbage collected. The below Figure 6 is a flow diagram demonstrating how object tracing is used during GC.

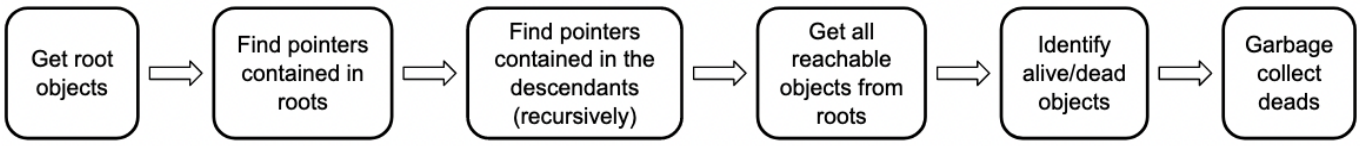

Figure 6. Role of object tracing in GC

However, GHC's current object tracing is integrated inside its memory manager, and inseparable from its collection phase, so there is no feasible object tracing function that MMTk can call from GHC directly.

Therefore I created a separate, Rust-based object tracing function. The function itself includes almost 2,000 lines of code. Since there are many types of GHC objects, we also need to introduce GHC object types using Rust representation. Then we implemented the tracing, depending on the object types.

The code implementation can be found at [16].

The new object tracing function slightly differs from the original version inside GHCSM. We used Rust type features to generalize some objects such as constants, operators, object info tables, etc. The type feature of Rust also helps with error handling at compile time.

5. Outcome

There are two major outcomes of this project:

- Refactorization of some GHC runtime code;
- Introduced MMTk allocation to GHC;
- Implemented object trancing for GHC objects.

The first outcome was mentioned in section 3.2 and 3.3. The project therefore has helped with the generality of GHC's runtime and benefited the community by proposing the above abstraction leakage fix. Note that GHC is a highly optimized and tuned system, so that changing one instance can result in a change in a number of different places, although the proposed changes seem to be minor, the actual refactoring touched about 30 files.

The second outcome is a successful NoGC binding between GHC and MMTk. With a heap size of 100M, the modified GHC (located in [12] branch mmtk/nogc1.0) can successfully run some reasonable programs, including calculating the 1st up to 5000th fibonacci numbers, and a series of small programs in [17].

The third outcome is also pretty successful. The tracing function can identify all the pointers in most of the objects in GHC and correctly (compared with the profiling function in heap.c), including standard closures, byte arrays, functions, stack frames. We created a testing program that generates random GHC objects and performed testing on these objects to observe the output pointers.

6. Discussion

6.1 “Trade-off” between performance and modularity

A major motivation behind the MMTk-GHC binding is to check the generality of MMTk, and develop further features if needed. This gives rise to an open question in software engineering design, about whether software generality, performance and modularity can be achieved at the same time.

MMTk is an example of introducing a modular architecture for runtimes. The resulting runtime after the binding refactorization should be highly modular: the memory manager should be well-encapsulated, and runtime should interact with its memory manager (or with MMTk) only via standardized interface. Such a system would be easier to maintain and develop new features, while having better performance at the same time.

Another example is the current design of the GHC block-structured memory manager, which provides flexibility and efficiency to arrange memory, as well as an easy extension to the copying generational collector. These two examples demonstrate the possibility that a good software engineering design can bring multi-level benefits to software systems without (or with only very little) trade-off.

6.2 Design of large software system

Another food for thought from this project is the design choices when implementing large software systems like GHC. We can see even a large and well-maintained system like GHC has abstraction leakages, which create barriers for adapting to a wider range of memory managers.

Our world is constantly changing and the pace of change is increasing. Technology develops at a frightening rate, and this creates new requirements and abilities. For a system, in order to keep functioning in a constantly changing environment, it needs to have the ability to respond to changes. Systems are required to have an expansion capacity, an ability to grow. If a system is unable to do so, it might not be useful anymore. Ability to grow a system is called scalability, and it is one of the key requirements to the system nowadays [18].

Attaching new components (e.g. MMTk binding) to the existing GHC system is problematic due to the current system design. This can be identified as a scaling issue. GHC is a large and fast evolving system, new components are being developed and integrated. Working on uniformity and atomicity of the components at the same time might be challenging, because different developments might have contradicting requirements. Therefore a practical trade-off has to be considered. In the meantime, developing on a highly tuned component can also introduce technical debt for later refactoring. That’s why it would be good to consider software scalability issues at the early stage of design. [19]

Modular structure with uniform components, such as MMTK framework, allows an easier exchange to a new component, or an upgrade and extension of the current components. Automation and standardization of component integration will also bring an improvement, since this will provide a systematic approach to system development.

7. Future Work

The outcome of the project is meaningful – it fulfills the key preliminary requirements of the MMTk-GHC binding. These steps are necessary for any further development of the binding.

The possible next steps are:

1. Use the object tracing functions in MMTk collectors, to support mark and sweep GC;
2. Introduce write barriers to support generational GC;
3. In object tracing, support more elaborate heap objects like the weak pointers;
4. Once the correctness of any GC has been established, characterize its performance; relative to GHCSM and optimize as necessary;
5. Iteratively support more MMTk collectors in GHC.

A binding with functionalities in step 1-4 implemented can be considered as relatively mature. It would be then possible to upstream such binding to GHC for programmers to use. The performance measurements, along with the MMTk-Haskell platform, can also be used to facilitate GC algorithm research.

Reference

[1] Blackburn, S.M., Cheng, P. and McKinley, K.S. (2004). *Oil and water? High performance garbage collection in Java with MMTk*. [online] IEEE Xplore. doi:10.1109/ICSE.2004.1317436.

[2] Wikipedia. (2022). *Manual memory management*. [online] Available at: https://en.wikipedia.org/wiki/Manual_memory_management [Accessed 26 May 2022].

[3] Memory Management Toolkit. *Memory Management Toolkit*. [online] Available at: https://www.mmtk.io/ [Accessed 26 May 2022].

[4] www.mmtk.io. *mmtk::plan - Rust*. [online] Available at: https://www.mmtk.io/mmtk-core/mmtk/plan/index.html [Accessed 26 May 2022].

[5] www.mmtk.io. *MMTk Porting Guide - MMTk Porting Guide*. [online] Available at: https://www.mmtk.io/mmtk-core/portingguide/prefix.html [Accessed 26 May 2022].

[6] taylor.fausak.me. *2020 State of Haskell Survey results*. [online] Available at: https://taylor.fausak.me/2020/11/22/haskell-survey-results/#s2q0 [Accessed 26 May 2022].

[7] GitLab. *gc · Wiki · Glasgow Haskell Compiler / GHC*. [online] Available at: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/gc [Accessed 26 May 2022].

[8] Marlow, S., Harris, T., James, R.P. and Peyton Jones, S. (2008). Parallel generational-copying garbage collection with a block-structured heap. *Proceedings of the 7th international symposium on Memory management - ISMM '08*. doi:10.1145/1375634.1375637.

[9] Ungar, D. (1984). Generation Scavenging. *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments - SDE 1*. doi:10.1145/800020.808261.

[10] GitLab. *rts · Wiki · Glasgow Haskell Compiler / GHC*. [online] Available at: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts [Accessed 26 May 2022].

[11] Marlow, S., Peyton Jones, S. and Singh, S. (2009). Runtime support for multicore Haskell. *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*. doi:10.1145/1596550.1596563.

[12] GitLab. *Files · mmtk/sm2.0 · Junming Zhao / GHC*. [online] Available at: https://gitlab.haskell.org/JunmingZhao42/ghc/-/tree/mmtk/sm2.0 [Accessed 26 May 2022].

[13] GitLab. *Draft: Porting MMTK: Eliminating usage of `bdescr` and generations in non-sm part of RTS (!7569) · Merge requests · Glasgow Haskell Compiler / GHC*. [online] Available at: https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7569 [Accessed 26 May 2022].

[14] GitLab. *heap objects · Wiki · Glasgow Haskell Compiler / GHC*. [online] Available at: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts/storage/heap-objects#thread-state-objects.

[15] GitLab. *Fix dead threads (!7939) · Merge requests · Glasgow Haskell Compiler / GHC*. [online] Available at: https://gitlab.haskell.org/ghc/ghc/-/merge_requests/7939.

[16] *mmtk-ghc*. [online] GitHub. Available at: https://github.com/JunmingZhao42/mmtk-ghc [Accessed 26 May 2022].

[17] Haskell.org. (2019). *Learn Haskell in 10 minutes - HaskellWiki*. [online] Available at: https://wiki.haskell.org/Learn_Haskell_in_10_minutes.

[18] Wikipedia Contributors (2022). *Scalability*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Scalability#Examples [Accessed 26 May 2022].

[19] Gorton. (2022). *1. Introduction to Scalable Systems - Foundations of Scalable Systems*. [online] Available at: https://learning.oreilly.com/library/view/foundations-of-scalable/9781098106058/ch01.html#ch01_manageability [Accessed 26 May 2022].