# Contents

1 Introduction .................................................. 3

2 Planned Work ................................................. 4
   2.1 Schedule .................................................. 4
   2.2 Deliverables .............................................. 5
   2.3 Evaluation ............................................... 5

3 Actual Work ................................................. 6
   3.1 Week 1-2 Project - Optimizer (Value Numbering) ........... 6
   3.2 Week 1-2 Reading - Ch. 1-4 ................................ 8
   3.3 Week 3-4 Project - VM Code Generator ....................... 9
   3.4 Week 3-4 Reading - Ch. 5-8 ................................ 10
   3.5 Week 5-6 Project - Copying Collector ....................... 11
   3.6 Week 5-6 Reading - Ch. 9-12 ................................ 12
   3.7 Week 7-8 Project - Mark-Compact Collector .................. 14
   3.8 Week 7-8 Reading - Ch. 13-15 .............................. 15
   3.9 Week 9-10 Project - Generational Collector ................. 16
   3.10 Week 9-10 Reading - Ch. 16-19 ......................... 17

4 Idle Time Garbage Collection Scheduling (PLDI ’16) ........... 19
   4.1 Scheduling Preliminaries .................................... 19
   4.2 Scheduling Details ......................................... 20
      4.2.1 Scheduling Minor GC .................................. 20
      4.2.2 Scheduling Major GC .................................. 20
   4.3 Performance results ....................................... 21

5 Concurrent Marking of Shape-Changing Objects (ISMM’19) ...... 22
   5.1 On the Write Barrier ...................................... 22
      5.1.1 Under the relaxed memory model ....................... 23
      5.1.2 Synchronizing on the mark bits via CAS ................. 24
5.2 On the Shape Changes
5.2.1 Safe shape changes
5.2.2 Unsafe shape changes
5.3 On the Result

6 Hierarchical memory management for mutable state (PPoPP’18)
6.1 Disentanglement Invariant and Promotion
6.2 Challenges of Promotion and Concurrency
6.3 Thoughts on overall design
6.3.1 The similarity of multi-spaces collector
6.3.2 Prioritize the common use cases

7 Course Project - Generational GC with Incremental Old-Gen Collector
7.1 Proposal
7.1.1 Overview
7.1.2 Details, Proposed Extensions and Challenges
7.2 Writeup
7.2.1 Plan
7.2.2 Status
7.2.3 What’s completed

8 Appendix A: LangF

9 Appendix B: Number of hours spent
Chapter 1

Introduction

The advantages that garbage collected languages offer to software development are legion. It eliminates whole classes of bugs, such as attempting to follow dangling pointers that still refer to memory that has been reclaimed or worse, reused in another context. It is no longer possible to free memory that has already been freed. It reduces the chances of programs leaking memory, although it cannot cure all errors of this kind. It greatly simplifies the construction and use of concurrent data structures. Above all, the abstraction offered by garbage collection provides for better software engineering practice. It simplifies user interfaces and leads to code that is easier to understand and to maintain, and hence more reliable. By removing memory management worries from interfaces, it leads to code that is easier to reuse.


This independent study will primarily focus on garbage collection and follow the The Garbage Collection Handbook [1] textbook developed by professors Richard Jones, Antony Hosking, and Eliot Moss during the first 10 weeks, and will also examine other tools and research literature describing the importance and state-of-the art of garbage collection research during the second half of the semester.

To supplement reading with some implementation activities, the study will also contain two projects. The first would use professor Fluet’s compiler construction course project LangF and its VM as a vehicle to practise on variant garbage collection algorithm. To fast forward, the front-end works are skipped. The second would be a non-trivial final project, tentatively via research vehicles like Jikes RVM, MLton, etc.
Chapter 2

Planned Work

2.1 Schedule

A tentative schedule of this independent study is given below.

<table>
<thead>
<tr>
<th>Weeks</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LangF Project 4 (value numbering)</td>
</tr>
<tr>
<td>2</td>
<td>GC Ch. 1-4 (LangF Project 4 due)</td>
</tr>
<tr>
<td>3</td>
<td>LangF Project 5 (vm codegen)</td>
</tr>
<tr>
<td>4</td>
<td>GC Ch. 5-8 (LangF Project 5 due)</td>
</tr>
<tr>
<td>5</td>
<td>LangF copying</td>
</tr>
<tr>
<td>6</td>
<td>GC Ch. 9-12 (LangF copying due)</td>
</tr>
<tr>
<td>7</td>
<td>LangF mark-compact</td>
</tr>
<tr>
<td>8</td>
<td>GC Ch. 13-16 (LangF mark-compact due)</td>
</tr>
<tr>
<td>9</td>
<td>LangF ref count or generational copying</td>
</tr>
<tr>
<td>10</td>
<td>GC Ch. 17-19 (LangF wk9 due)</td>
</tr>
<tr>
<td>11</td>
<td>Research Papers</td>
</tr>
<tr>
<td>12</td>
<td>Research Papers (Project proposal due)</td>
</tr>
<tr>
<td>13</td>
<td>Research Papers</td>
</tr>
<tr>
<td>14-15</td>
<td>Project work</td>
</tr>
<tr>
<td>16</td>
<td>Project presentation</td>
</tr>
</tbody>
</table>
2.2 Deliverables

The student and faculty sponsor will meet weekly to discuss progress. Deliverables would include this independent study reports, reading and research paper writeups, and two coding projects.

2.3 Evaluation

A tentative grading rubric is given below.

- Weekly Meetings ("preparation and participation"): 10%
- Reading and Research Paper writeups: 15%
- LangF Projects: 50%
  - LangF Project4: 6.25%
  - LangF Project5: 6.25%
  - LangF mark-sweep GC: 12.5%
  - LangF copying GC: 12.5%
  - LangF ref count or generational copying GC: 12.5%
- Project: 25%
  - Proposal: 5%
  - Writeup: 5%
  - Presentation: 5%
  - Code/Effort: 10%
Chapter 3

Actual Work

In this section I summarize the actual work completed over the course of this independent study. Summaries are provided on a *pragmatism* manner. I will describe the projects and readings biweekly for the first 10 weeks, with the writeups for research paper readings attached, and end with the course project. The focus for book readings will be theories and knowledge that I learned from books and meetings, and for projects will be background knowledge, implementation status, challenges, solutions and feedback.

3.1 Week 1-2 Project - Optimizer (Value Numbering)

The project is to implement a *value numbering* optimization for the A-Normal Form Intermediate Representation (AnfIR) in the LangF compiler/interpreter.

*Value numbering* is an optimization that analyzes a program for expressions that are *congruent*, meaning that the expressions are guaranteed to compute the same value at runtime, and transforms the program to eliminate later expressions that are congruent with an earlier expression.

— *LangF Project4 Optimizer (Value Numbering)*

This is a heavy start. Prof. Fluet provided me the materials for *LangF* Project 1-4 and project seed code for Project 4 which contains the completed *lexer, parser and type checker* from *LangF* Project 1-3. I spent roughly a dozen of hours on going over the previous projects; taking notes; and getting me familiar with the codebase. The codebase is very well engineered and I really enjoyed exploring it. Some of the implementation details that I learned and noted are:

- The **Control** module for managing compiler *passes* in a central place.
- The infrastructure for managing source files, positions, errors, prettier printing.
Fundamental representation for each passes including tokens, parse tree, (typed) AST.

Common arrangement such as Environment with Map/Tbl; ID for generating UUID.

Many Standard ML idioms, such as the heavy use of the module system and functors.

Then I embarked on the actual Project 4 works, which required me to first get familiar with the two IRs that being introduced:

- The CoreIR is closer to SystemF. It’s lowered from typed AST by lowering if orelse, andalso to case; unary/binary/ternary operations to primitive calls to runtime; sequence to nested let; multiple argument functions to curried CoreIR.Lam for both term-level (λ) and type-level (Λ), etc.

- The AnfIR is similar with SSA (from some perspective). It helps with making optimization easier by giving names to intermediate results and making the dataflow more explicit.

One valuable technique that I learned is that both IRs need to be first validated (type-checked) before being used so that compiler internal bugs can be found earlier. That’s one of the benefits of using typed IR during compilation.

I started the implementation of value numbering optimization on the LangF VM after completing reading of Prof. Fluet’s materials and some of the relevant sections in the chapter 8 of Engineer A Compiler [2] pointed out in Prof. Fluet’s materials. The implementation process was not entirely smooth and the main roadblocks and gotchas are:

- My first pass was using shared mutable hash table for implementing maps but found it’s tricky to handle out-of-the-scope cases. The second pass switches to immutable tree map and then it went smoothly — Functional programming for the win!

- I mis-thought first implementing a LVN (Local Value Numbering) that only applied to the level of BB (Basic Block) would be easier but later found implementing a SLVN (Superlocal Value Numbering) that directly applied to the level of EBB (Extended Basic Block) is actually much more natural.

I wasn’t got sufficient time to do much for the extra credit: Extending value numbering on strings was completed; Elimination commutative primitives was implemented wrong; Constant founding was not completed. The automatic test suites yield 100.99% grades.
3.2 Week 1-2 Reading - Ch. 1-4

Unfortunately, *liveness* is an *undecidable* property of programs: there is no way to decide for an arbitrary program whether it will ever access a particular heap object or not. Just because a program continues to hold a pointer to an object does not mean it will access it. Fortunately, we can approximate liveness by a property that is decidable: *pointer reachability*. An object $N$ is *reachable* from an object $M$ if $N$ can be reached by following a chain of pointers, starting from some field $f$ of $M$. By extension, an object is only usable by a mutator if there is a chain of pointers from one of the mutator’s roots to the object.


The first bi-week of reading is trying to cover the first four chapters:

*Ch1. Introduction* covers the background, benefits of having GC comparing to manual memory management, and lots of preliminaries including metrics for comparing GC algorithms, terminology and notations, etc. The most important takeaways to me was that garbage collection problem can be reduced into a graph reachability problem where heap memory is represented as a directed graph in which nodes are objects in the heap and edges are pointers (references).

*Ch2. Mark-sweep* starts with the famous mark-sweep algorithms used in Lisp back to 1960 [3], which is known as the very first garbage collector. This chapter also introduce the concept of *tracing* and *indirect collection* that can differentiate this category of GC methods out of the direct collection method e.g. *reference counting*. Many optimization was discussed, such as *Segregated-fit free list*, *Bitmap marking*, *BiBOP (Big Bag of Pages)*, *Lazy sweeping*, etc. It goes into much more details on comparing how would different design affect performance impacts, such as different iterating order (DFS vs. BFS) on *locality*.

*Ch3. Mark-compact* discussed the importance of *fragmentation* problem and *compacting* to the rescue. In contrast with *copying collection*, mark-compact algorithms perform *in-place compaction*. The *compaction order* has locality implications. Several algorithms were discussed:

- *Lisp2*, widely used, *sliding* order, 3 passes on the heap.
- *Two-Finger*, required fixed size object, *arbitrary* order, 2 passes.
- *Threaded* [4], used in *MLton*, *sliding* order, 2 passes.
- One-pass algorithms could be done with tables and constraints.

*Ch4. Copying* presented the lovely *semispace copying* [5] algorithms such as *Cheney’s Copying* [6] which leverage the *to-space* as the worklist (to-be-scanned set). Copying collectors have many very attractive advantages:
• Supporting bump pointer (or linear) allocation, great throughput.
• It compact the heap while during the copying (a.k.a. evacuating or scavenging).
• Require only one pass in total via forwarding pointers, good pause time.

The only downside is the heap utilization because of the requirements of copy reserve. As usual, traversal order (copying order) have locality impacts.

I should admit that I was not able to understand lots of the in-depth details from the book back to my first reading. I was too new to this domain and was lack of related background knowledge about things such as CPU cache and OS virtual memory (I have to pick up those whenever I encounter some). The book contents also have many forward references as dependencies that confused me a lot. Fortunately, more and more details gradually "clicked" along with me reading more chapters and doing more projects.

It also worth to mention that I also supplement the study with another garbage collection book, 垃圾回收的算法与实现 (ガベジコレクションのアルゴリズムと装) [7] (originally published in Japanese by people behind Ruby’s GC, but there happen to be a Chinese translation that I can understand). It covers a subset of algorithms discussed in The Garbage Collection Handbook [1], as well as some early (relative to the writing of this report) versions of GC in Python, Davlik VM, Rubinius and V8. It is intended to be more casual thus more friendly to newcomers. It helped me getting over some initial barriers during the study of The Garbage Collection Handbook [1].

3.3 Week 3-4 Project - VM Code Generator

The project is to implement a simple code generator for LangF, which takes a high-level intermediate representation and produces an object file for interpretation by a virtual machine.

The virtual machine is a stack-based interpreter. Code generation consists of two phases. The first phase translates the high-level intermediate representation (CoreIR) to a lower-level intermediate representation in which data representations and variable locations have been determined. The second phase translates this low-level intermediate representation to bytecode instructions.

— LangF Project5 VM Code Generator

I really enjoyed this project which uncovered many mysteries in virtual machine implementations even though this is just one of the instances of them. I expect many of the interesting background knowledge that I learned would be universally applied in other similar systems in someways. Some of them are:
- How does data in higher-level programming languages represent in memory.
- Better understandings on VM configuration: stack; heap; registers, FFIs...
- Closures representation and calling convention.
- Pattern matching implementation.
- Pointers are *word-aligned* and we can use this fact to steal bits for VM-internal meta data, such as *tagged integers* and *mark bits*.

I also enjoyed the *formalization* of the VM a lot. It described the semantics of each instruction precisely by stepping between configurations and it has been very useful as a reference manual during the implementation.

Having been understood how does *LangF* VM work, the code generations for its bytecode itself is a fairly straightforward work. One of the stupid bug that I made was incidentally use a `loadlocal` for a place that should have be `storelocal`, and the debugging process of tracing bytes was *fun*.

I completed one improvement for *extra credit*: the *self application* improvement. Many of the improvements mentioned in the material are very interesting. They give me a sense of the direction of optimizations that could be made into a VM.

### 3.4 Week 3-4 Reading - Ch. 5-8

We have shown that tracing and reference counting garbage collection, which were previously thought to be very different, in fact share the exact same structure and can be viewed as *duals* of each other. This in turn allowed us to demonstrate that all high-performance garbage collectors are in fact hybrids of tracing and reference counting techniques.

In the process we discovered some interesting things: a write barrier is fundamentally a feature of reference counting; and the existence of cycles is what makes garbage collection inherently non-incremental: cycle collection is “trace-like”.

— *A unified theory of garbage collection, OOPSLA ’04* [8]

In the second bi-week of reading, we went through another four chapters. Again, I should admit that my first pass over Ch7. *Allocation* and Ch8. *Partitioning the heap* was terrible. I only able to understand most of them when I went back later. The most gain to me from this week was the 6.6 *A unified theory of garbage collection* and the original in-proceedings paper.

*Ch5. Reference counting.* The RC algorithm is classified as *direct collection*. It is "notorious" for its inability of handling *cyclic garbage*. Implementing RC requires the uses of
write barrier to maintain the reference count, which imposes significant throughput overhead. However, there are some advantages of RC:

- It can be implemented as a library without assistance from runtime system.
- It has relatively more deterministic behavior than tracing collectors by giving programmers more control. If controlled well, it might have better maximal pause time.
- It has better memory utilization comparing to lots of tracing collector.

Some of the variants were discussed: Lazy, Deferred, Coalescing, Buffered, as well as approaches to handle cyclic garbage: Backup tracing; Trial deletion; Partial tracing. Most of them tend to introduce some amount of tracing to bring some advantages (as well as disadvantages) of tracing in, while trading-off some disadvantages (as well as advantages) of RC.

Ch6. Comparing garbage collectors presented a high-level comparison (over many metrics) on previously discussed four basic collectors. N.B. All advanced collectors can be seen as some kind of combination of these four basic ones.

Ch7. Allocation discussed in detail on the allocation because allocation and reclamation of memory are tightly linked: how memory is reclaimed places constraints on how it is allocated. (I was not able to get this until having implemented some). Specifically, sequential allocation and different strategies of free-list allocation; fragmentation; segregated-fits (size classes) were discussed.

Ch8. Partitioning the heap discussed different arrangements on splitting the heap, in which the best-known one is generational collection (another instance of forward references). This chapter opened my mind on optimizing collector in this level, where one can exploit any possible properties of a group of objects to partition the heap and gain some benefits.

During this week, I also spent half of the time going through chapters of Conservative GC, Genrational GC, Incremental GC in 垃圾回收的算法与实现（ガベージコレクションのアルゴリズムと装）[7] to have a better expectation on what’s next after these four basic collectors.

### 3.5 Week 5-6 Project - Copying Collector

Semispace copying [5] [6] compacts the heap, thus allowing fast allocation, yet requires only a single pass over the live objects in the heap. Its chief disadvantage is that it reduces the size of the available heap by half.

The elegant Cheney scanning algorithm [6] uses the grey objects in tospace as a first-in, first-out queue. It requires no additional storage other than a single pointer.
Copying is the first collector that I implemented due to its simplicity. The project seed code is my completed LangF Project 5, in which the heap simply grow (double its size then copy all the data and adjust all the pointers) whenever it is exhausted.

Besides of the core algorithms, what I felt interesting to note down are:

- **Forwarding pointers** are tagged to be distinguishable and stored by reusing bits of the header field. It will be written into the old object after it being forwarded (copied) from the from-space to to-space. I used 0x5 (0101) for the tag to tell it apart from plain heap objects (either 0x2 (0010) for record or 0x6 (0110) for string).
- **Tracing** is implemented as DFS directly in recursion for convenience (see below).
- **Bug-fix** on Word_t was mis-treated as byte.

Although lots of the functionality are correctly implemented, there are 1 minor issue and 2 significant issues that I wasn’t even thought about until Prof. Fluet pointed out later:

- It would be better to use 0x3 (0011) as the tag for forwarding pointers because in 32-bit platforms, objects would be aligned on 4-byte boundaries so only the low two bits would be guaranteed to be 0 for a pointer.
- It should not use recursive function! It might overflow the C stack. The original Cheney’s Copying use to-space as the FIFO queue (as in our quote for this section). N.B. that would be a BFS.
- I did not implemented the allocation fail and memory exhaustive cases.

That’s why I need to take this independent study!

### 3.6 Week 5-6 Reading - Ch. 9-12

By concentrating reclamation effort on the youngest objects in order to exploit the weak generational hypothesis that most objects die young, they hope to maximise yield (recovered space) while minimising effort. Generational collectors segregate objects by age into generations, typically physically distinct areas of the heap. Younger generations are collected in preference to older ones, and objects that survive long enough are promoted (or tenured) from the generation being collected to an older one.

Third bi-week was mostly focusing on Ch 9. *Generational garbage collection*. I had been aware of this technique for a long time. One of the key motivations of being generational is that we can only collect part of the heap and reduce expected pause time, especially for the minor GC. Approaching this requires not only heap being partitioned as new spaces and old spaces, but also ages of objects being recorded by either object or space; and the uses of remember set which remember inter-generational pointers so we can regard them as roots for collecting one generation.

The chapter presented lots of variants of generational collectors that really demonstrate the fact that advanced collectors are really combinations of basic collectors and different choices of heap partitioning and allocator. Two of the most notable ones are:

- **Ungar’s** [9] partition the heap as one new space and two survivor spaces and ages are recorded per object. The benefits of such arrangement are that young generation objects can be collected via copying arbitrary times between the two survivor spaces. So the collector can easily scaled to arbitrary steps. Sun’s HotSpot JVM (2006) even take advantage of this to use a dynamic promotion policy that does not impose a fixed age limit but instead attempts to keep the survivor space half empty.

- **Appel’s** [10] compute the copy reserve on-the-fly by always partitioning the rest of the heap (in terms of the old space) into two semispaces, one as the nursery space for new allocation and one as the copy reserve for the nursery space. The collector maintain the invariant that there is always reserved space for a copying. It’s technically 1-step (considered it’s designed for ML).

This style of collector is also used by MLton by further extending with a threaded compaction collector to form a dual-mode collector [11].

**Ch10. Other partitioned schemes** extended the previous chapters and describe other segregation including Large object spaces, Topological collectors, and Hybrid mark-sweep, copying collectors. Two of the famous examples are Garbage-First, or G1 [12], introduced in Sun’s HotSpot VM in JDK 7; Immix [13] and its later variation RC-Immix [14]. Each of them is quite sophisticated and has a full research paper published dedicated for it. I will have to take them as the future works for this independent study.

**Ch11. Run-time interface** described many concerns on the interface between the collector and the rest of the system. Recalled that most collectors would love to have assistance from run-times to be complete. In particular, Allocation and initialization need to be coherent; Finding pointers can either be conservative or accurate via tagged values or type information, etc.; External references created by FFI need be specially took cared; Barriers for stack, write, and read; Card table and Crossing maps for implementing remembered
set...lots of things need to be concerned and those concerns tend to be specific to a particular programming language or virtual machine. This chapter is greatly practical as a reference.

Ch12. Language-specific concerns described the impact of Finalization and Weak reference could have on collector design. I only got time to skim this chapter.

3.7 Week 7-8 Project - Mark-Compact Collector

The Lisp 2 collector is widely used, either in its original form or adapted for parallel collection [Flood et al, 2001]. It can be used with objects of varying sizes and, although it makes three passes over the heap, each iteration does little work (compared, for example, with threaded compactors).

The first pass over the heap (after marking) computes the location to which each live object will be moved, and stores this address in the object’s field...The second pass updates the roots of mutator threads and references in marked objects so that they refer to the new locations of their targets, using the forwarding address stored in each about-to-be-relocated object’s header by the first pass. Finally, in the third pass, relocate moves each live (marked) object in a region to its new destination.


The second collector that we chose is Lisp2. It’s a very representative compaction collector and although it required 3 passes, each of them is just responsible for one small and clean job.

The core algorithms followed the The Garbage Collection Handbook [I] heavily. The own idea are mostly about the run-time interfaces such as object shape, data structure implementation and better engineering.

- Forwarding pointers are stored in its dedicated field (-2 relative to object pointer).
- Marking bits are stole from the header field using 0x1 (the lowest bit).
- Marking work-list is implemented via a new stack.c.
- C macros are used to help with better engineering.
- Tracing messages followed the original VM setting to help with debugging.
- Growing heap when memory is exhausted.

Most of the pain that I had was related on debugging bugs caused by pointer arithmetic and wrong index calculation.
There are still one quite significant issue that I wasn’t thought about that was pointed out in the grading: Marking work-list should not use extra memory during GC. This point was not presented from my reading on *The Garbage Collection Handbook* [1], but it’s quite intuitive. The alternative (better) solution without using an external work list data structure are:

- Schorr-Waite pointer reversal marking scheme (used by MLton).
- Threaded objects as linked list stack by reusing the forwarding pointer field.

### 3.8 Week 7-8 Reading - Ch. 13-15

An obvious way to reduce pause times is to have all processors cooperate to collect garbage (while still stopping all mutator threads). This parallel collection is the topic of this chapter.


Another way to reduce pause times on a uniprocessor is to interleave mutator execution with collector execution...called incremental collection...Mostly-concurrent collection avoids some synchronisation overhead by assuming that the mutators are all stopped together for a brief period during each collector cycle...Relaxing the need for a global stop-the-world phase yields purely concurrent on-the-fly collection,


The fourth bi-week was tough. This part required lots of background knowledge to follow and significant amount of time was spent on that.

*Ch13. Concurrency preliminaries* discussed tons of details on concurrency. It almost like a small concurrency textbook to me. I had no idea on this domain of computer science at all previously. Topics includes *Process synchronization*; Constraints and supports from hardware such as *Hardware primitives* from chips and instruction sets; Atomic *read-modify-write* operations such as *Compare-and-swap* and its *ABA problem*; Variance of locks including *Spin lock, Optimistic locking*; Memory consistency model such as *strict (strong) and relaxed (weak)* which allowed *reordering; Progress guarantees; Lock-free; Wait-free*...I was definitely not able to understand all the details but still, I slowly and gradually learned a lot with the help of Wikipedia and Google.

*Ch14. Parallel garbage collection* and *Ch 15. Concurrent garbage collection* discussed the ideas and challenges of concurrency and parallelism in garbage collection in general. Challenges mostly came from the external complexities while the collector need to still maintain
the same invariant as their uniprocessor equivalence to ensure the correctness without losing objects.

The *Strong/weak tricolor abstraction* is used to describe the GC status of each heap object in a concurrent setting. Such colors are often recorded concretely in the object to help the system out under the non-determinism brought by concurrency. Different style of *concurrent write barriers* are invented to detect the occurrences of problematic cases so it can perform some "fixes" and recover the invariant.

This two chapters also clarify the terminologies and categories of parallel/concurrent collectors and point me many directions for future study on this topic.

### 3.9 Week 9-10 Project - Generational Collector

Card table (*card marking*) schemes divide the heap conceptually into fixed size, contiguous areas called *cards*...Whenever a pointer is written, the write barrier *dirties* an entry in the card table corresponding to the card containing the source of the pointer.

As a card table is searched, each dirty card discovered must be processed, which requires finding the modified objects and slots somewhere in the card. This is not straightforward since the start of the card is not necessarily aligned with the start of an object but in order to scan fields we must start at an object. In order to be able to find the start of an object, we need a *crossing map* that decodes how objects span cards.


It took us a while to decide how should the third collector be like. There are quite a lot of different settings can be choose, each has its own easy parts and challenges. After discussions, we decided our generational collector project would be arranged as such:

- The young generation collector would be a (correct) implementation of Cheney’s Copying, as most of the generational setting.
- The old generation collector would be reusing the Lisp2 from last project.
- The focus would be the handling of *inter-generation pointers*. In particular, *write barrier* (only the case of `OP_UPDATE`) and *remembered set*. To make the project more interesting, we will use *card table* and *crossing map* to implement the *remembered set*.

Some of the non-trivial considerations are:

- *Copy reserve* is assigned for new space in the old space to make minor GC easier.
• **Object shape** is constant so there is one word *wasted* for young generation object.

• **Write barrier** is strict and only dirty the card table in the old -> new cases.

• **Crossing map** records the offset for "lowest dirtied objects" by one comparison.

• **Pre-tenuring** for objects that is larger than the entire new space.

The feedback on this project was decent. However there is always room for improving:

• We could be more *conservative* on the write barrier to increase the throughput, e.g. ignoring the isNew check for the written object. Essentially, we are trading the pause time (spent on scanning dirtied card) for throughput. It’s also possible to even ignore the isOld check and expand the card table to the whole heap as well. This would be ideal for the native compilation case. (VM has more control over things)

• We could *postpone* updating the crossing map until either minor or major GC instead of updating it eagerly during the card marking (dirtying).

• It’s possible to combine the two tables by using an "invalid" offset as "unmarked" flag.

3.10 Week 9-10 Reading - Ch. 16-19

The mark-sweep family are the simplest of the concurrent collectors. Because they do not change pointer fields, the mutator can freely read pointers from the heap without needing to be protected from the collector. Thus, there is no inherent need for a read barrier for non-moving collectors. Read barriers are otherwise generally considered too expensive for use in maintaining the strong invariant for a non-moving collector, since heap reads by the mutator are typically much more frequent than writes.


The fifth bi-week is the last one scheduled for reading. As the book went nearly to the end, the contents also became much harder and denser. Most of the examples are quite non-trivial and the book contents are more like summaries of those research papers cited. However, being aware of those projects and topics definitely broaden my views and help me setting the goal of future study.
Ch16. *Concurrent mark sweep* went deeper in the *Initialization* phase and *Termination* phase of each collection cycle for mostly concurrent collectors, as well as more details on allocation and concurrent marking and sweeping. Finally, it introduced the novel *on-the-fly* collectors such as the Doligez-Leroy-Gonthier collector used by CAML [15] [16], and later adopted for Java and the Manticore project [17].

Ch17. *Concurrent copying & compaction* are both moving collectors so they would suffer much more from collecting concurrently with mutating. *Read barriers* thus need to be used to maintain the *tricolor abstraction*. One interesting instance of them is the *self-erasing* read barriers used in *Glasgow Haskell Compiler (GHC)*.

Ch18. *Concurrent reference counting* and Ch 19. *Real-time garbage collection* are skimmed and would remain as future study.
Chapter 4

Idle Time Garbage Collection Scheduling (PLDI ’16) [18]

Interactive programs such as GUI applications or games are critical to be responsive to users’ inputs. The long pause time introduced by automatic garbage collection has been blamed to be a main source of latency that cause jank (noticeable frame drop), meaning the refresh rate could not keep up with 60 frames per second.

This paper presented a novel way of scheduling garbage collections in the idle time during the rendering in Chrome, by cooperating V8 JavaScript engine and the task scheduler in Blink rendering engine. This writeup will discuss about key challenges and solutions of the scheduling decisions in my own words and orders based on my own understandings.

4.1 Scheduling Preliminaries

JavaScript environment includes the core JavaScript engine (such as V8) that implemented the ECMAScript standard and the hosting system to provide some runtime supports and more functionalities. N.B. ECMAScript standard doesn’t specify the underlying concurrency model but only ask for a "job queue" that it can emit "job" into. The hosting runtime system, notably server-side JavaScript system (such as Node.js) or web browser engine (such as Blink) will actually implement the task queue and fire those tasks. As of today, both Node and Blink use event loop to steal work from task queue and schedule any JavaScript scripting as tasks.

The Blink task scheduler, the implementation of this paper content, then responsible for not only JavaScript event callbacks but also all kinds of browser tasks such as layout and
painting, thus have centralized knowledge about how busy is the whole system, and able to
tell how many idle time do we have, after having performed all necessary works, before
the deadline, i.e. the next frame need to be completed in 16.67 ms. The scheduler also have
the power to "inject" idle task as it want, including JavaScript requestIdleCallback and
garbage collection work.

Different with many other offline profiling system, the scheduling decisions made by
scheduler heavily relied on online profiling. The scheduler knows how long does each kind
of task take, as well as detailed data such as the observed allocation rate and average idle time
of the current application running and observed marking speeds on the current hardware, so
the scheduling can take advantage of those profiling data to confidently estimate the time of
garbage collection task and make sure they meet the the deadline.

4.2 Scheduling Details

V8 uses a generational collector with young generation using semi-space copying, named
minor GC, and old generation using incremental mark-sweep-compact (as the time of time
paper), named major GC.

4.2.1 Scheduling Minor GC

The minor GC tend to be short and easy to fit into even short idle time. However, scheduling
minor GC too early would unnecessarily promote objects and too late would make the young
generation too large to be collectable within the idle time. The system uses a clever heuristics
to make sure the minor GC task is in between of "being able to complete in this idle time"
and "not leaving too many works to next idle time", by using knowledge of previous average
minor GC speed and average idle times.

4.2.2 Scheduling Major GC

The major GC works are categorized as starting, incremental marking, and finalization. Since
starting major GC is a critical decision to make, so it’s controlled by a dedicated memory
reducer, which also use allocation limits heuristics and even take advantage when the webpage
is detected to be "inactive" at all.

Incremental marking and finalization are scheduled as normal idle task and made sure
their estimated time can fit into the allotted idle time window. Besides, the incremental
marking task can be linearly scaled by the number of bytes, based on previous average
marking speed (observed on current hardware).
4.3 Performance results

The paper also introduced a new metric called *Frame Time Discrepancy* to measure the frame rate regularity. The novelty of this metric is that it takes the distance of two frame drops into account and can better represent the requirement of interactive GUI applications and games.

In terms of the benchmark result, the idle time scheduling can significantly improve the frame time discrepancy in the WebGL game scenarios, where the application is constantly rendering, striving to hit 60 FPS. In other web pages or applications scenarios that the frame rate is less a problem, idle time scheduling are mainly help with reducing the memory usages overall, since the system is smarter at scheduling garbage collection more frequently at the idle time without concerning interrupting users’ interaction.
Chapter 5

Concurrent Marking of Shape-Changing Objects (ISMM’19) [19]

V8, the well-known JavaScript virtual machine used in Google Chrome and Node.js has recently upgraded its garbage collector from an incremental GC into a mostly concurrent and parallel GC. The paper discussed technical details of the concurrent-marking phase of its (old space) collector, especially concerning the extra constraints where the interpretation of object, guided by the object shape, can change, due to how V8 optimize the dynamic nature of JavaScript language. This writeup will focus on the interesting parts of the design, specifically, the write barrier, the shape changes, and how does the concurrency affect the design of them.

5.1 On the Write Barrier

The write barrier used is a Dijkstra-style advancing-wavefront barrier.

In a single-threaded setting (e.g. in an incremental GC or in a generational GC), to protect from creating black-to-white pointers (and violate the strong tri-color invariant), one can shade the target object only if both the color of the source object and the target object able to pass the check:

```
// Called after `object.field = value;`.
write_barrier(object, field_offset, value) {
    if (color(object) == black && color(value) == white) {
        set_color(value, grey);
        marking_worklist.push(value);
    }
```

1Pseudo code from the V8 blog https://v8.dev/blog/concurrent-marking
5.1.1 Under the relaxed memory model

However, omitting the extra check on the source object allows the memory order to be relaxed, with the cost of the loss of precision (the write barrier becomes more conservative) and may result in more floating garbage.

```c
// Called after `atomic_relaxed_write(&object.field, value);`
write_barrier(object, field_offset, value) {
    if (color(value) == white && atomic_color_transition(value, white, grey)) {
        marking_worklist.push(value);
    }
}
```

That’s see why the strict write barrier (which checks the source object color) would be problematic under a relaxed memory model by looking at an example:

```c
atomic_relaxed_write(&object.field, value);
write_barrier(object, field_offset, value); // inlined
```

The problem is that the memory model might reorder the color load operation (in the write barrier body but inlined) before the actual store operation. If a marking thread happened to race here and scan the object to the black before the actual write performed, then we creates a black-to-white pointer.

```c
mutator load G <- color(obj)  // Gray, thus no shade and worklist pushing happened
marker store B -> color(obj)
marker follow references in obj
mutator store value -> obj[i]  // Created black to white pointer
```

Alternative approach would be to insert a memory fence to make sure the store always performed ahead of the color load operation so we can perform the more precise check. V8 team prefer to omit this check and avoid this expensive fence.

```c
atomic_relaxed_write(&object.field, value);
memory_fence();
write_barrier(object, field_offset, value);
```
5.1.2 Synchronizing on the mark bits via CAS

Another subtle race is that, after the actual store happened, the marker and the write barrier might simultaneously discover the new pointer and want to shade it and push it into the worklist. The algorithm protect from this race by synchronizing on the mark bits via a pair of CAS operations.

marker FollowReference(value):
  if HasRefTag(value):
    load color ← Markbits(value)
    if color = white and CAS (white → gray) → Markbits(value): --- (1)
      push value → Worklist

mutator WriteBarrier(value, type):
  if type = Tagged and HasRefTag(value):
    load color ← Markbits(value)
    if color = white and CAS (white → grey) → Markbits(value): --- (2)
      push value → Worklist

mutator WriteField(obj, i, value, type)
  store value → obj[i] --- actual store
  WriteBarrier(value, type)

This ensured the target object would be shaded and pushed into worklist exactly once.

5.2 On the Shape Changes

There are two categories of shape changes that might happened in V8: safe and unsafe, in terms of could those shape changes work with concurrent marking without any special synchronization. The paper mentioned five type of shape changes: Field Type Change, Append Field, Remove Field, Right Trim, Left Trim. Essentially, the unsafe changes are those sequence of shape changes that can cause a field type changed from tagged to untagged (or vice versa), either via a direct Field Type Change or a sequence of appending/removing.

5.2.1 Safe shape changes

Safe shape changes exclude the unsafe case by having conditions that "an object filed does not appear as as a tagged field in one shape and as a raw field in another shape" in its

\[^2\]Pseudo code from the paper
definition, which is formalized in the paper as:

$$\forall a \in B^W, a \mod W = 0 : \{Tagged, Raw\} \not\subset types(a), where \text{types}(a) = \bigcup_{i=0}^{n} f_i(a)$$

Most of other safe shape changes are trivially handled, there are two tricky cases still: changes from tagged to \( \epsilon \) used \textit{FillerClass} to maintain the invariant \textit{4.4 (Reference Safety)}, and changes from \( \epsilon \) to tagged are protected by write barrier to maintain the tri-color invariant.

N.B. each shape change operation emits a write barrier for each written references as code below:

```python
mutator FieldTypeChange(obj, new_class, affected_fields):
    release store new_class → obj[0]
    WriteBarrier(new_class, Tagged)
    for (index, value, type) in affected_fields:
        store value → obj[index]
        WriteBarrier(value, type)
```

### 5.2.2 Unsafe shape changes

Unsafe shape changes can be destructed as two cases: pointer addition (untagged to tagged) and pointer deletion (tagged to untagged).

Pointer addition cases are protected by write mutator same as the \( \epsilon \) to tagged case in safe shape changes. The assumption is that pointer addition changes would always performed with a references writing protected by write barrier.

Pointer deletion cases are handled by a snapshotting algorithm as below\(^3\). Intuitively, a snapshotting algorithm means it will firstly take a snapshot to the object before some changes, and later use that snapshot to scan and find transitive references, which is conservative but handle the cases safely. Together we maintain the strong tri-color invariants for unsafe shape changes

```python
snapshot = [];
hidden_class = atomic_relaxed_load(&object.hidden_class);
for (field_offset in pointer_field_offsets(hidden_class)) {
    pointer = atomic_relaxed_load(object + field_offset);
    snapshot.add(field_offset, pointer);
}
```

\(^3\)Code from paper Figure 9
\(^4\)We used the blog version for simplicity.
To handle the possible race between marker and mutator, the algorithm again use a pair of CAS to synchronize on the mark bits (with a slightly more constrained acquire-release model than full relaxed model) to ensure the mutator would only perform the shape changes when the object is black, either scanned by any concurrent marker thread, or by the mutator performing the snapshotting visiting:

**mutator UnsafeShapeChange_S(obj, ...):**

```java
load color ← Markbits(obj)
if color != black:
    CAS (white → grey) → Markbits(obj)
    if acquire CAS (grey → black) → Markbits(obj): --- (1)
    VisitObject(obj, class)
UnsafeShapeChange(obj, ...)
```

**marker VisitObject_S(obj):**

```java
let (snapshot, success) = TakeSnapshot(obj)
if success and release CAS (grey → black) → Markbits(obj): --- (2)
for value in snapshot:
    FollowReference(value)
```

This ensures the *Invariant 4.4 (Reference Safety)* holds during the shape changing.

### 5.3 On the Result

We briefly talk about the result. The V8 team evaluate the result by comparing the main (S) wait-free concurrent marking algorithms discussed in the paper with another four configurations:

- (I) original incremental marking (as baseline)
- (N) concurrent marking with unsafe shape change optimization and snapshotting turned off
- (LW) lock-based concurrent marking using per-word locks (fine-grained)
- (LP) lock-based concurrent marking using per-page locks (coarse-grained)

Here are some of the takeaways from the benchmarks:

1. All concurrent configurations improve "cumulative marking times on main thread" from 35% – 100% and no significant difference between different concurrent settings.

   - concurrent marking is always better in reducing pause time in main thread!
2. In terms of "cumulative marking time on all threads" \((I) < (N) < (S) < (LW) < (LP)\) – this strictly show the overhead of synchronization.

3. The memory consumption are all quite similar – faster marking mediated the floating garbage due to conservativity.

4. All concurrent configurations improve minimum mutator utilization (MMU) in most cases – because MMU measure the worst case, it’s likely that the main pause dominated the statistics when it occurred.

Concurrency and multi-processor for the win!
Chapter 6

Hierarchical memory management for mutable state (PPoPP’18) [20]

This paper presented a novel way to incorporate in-place updates, i.e. mutation, with hierarchical heaps, a memory management technique developed in prior work, that organize memory so that it mirrors the structure of the parallel computation, for implementing efficient parallelism in functional languages. The write-up will briefly discuss the important invariant, challenges and solutions, and some of my own thoughts on the overall design.

6.1 Disentanglement Invariant and Promotion

The most important invariant of this algorithm is to maintain that the hierarchy is disentangled, which means that the hierarchy must not contain down-pointer that point from ancestor to descendant, nor may it contain cross-pointer that point between unrelated heaps (e.g. two siblings).

Purely functional programming style without (arbitrary, or user-land) mutation naturally maintains this invariant. The problem raised in the presence of arbitrary mutation: a child task can easily create a down-pointer by writing a child object (from the child heap) into a mutable reference it holds that points to an parent heap object. To make things worse, a sibling child could read from the same mutable reference to create a cross-pointer.

The solution is to use write barriers to detect those problematic cases and promote (copy) the lower object to the super heap. Promotion would also eagerly promote all transitively reachable data, otherwise down pointers would be created by the promotion itself.
6.2 Challenges of Promotion and Concurrency

The challenges are how to preserve the semantics under promotions and concurrency.

The first one is about triggering promotions on the same target object multiple times and creates duplicates of it, due to repeated writes of a same pointer to objects located in heaps of decreasing depth. To make things worse, we might lose updates if they are mutable. The solution is to thread all copies as singly-linked list and always distinguish the highest copy as the \textit{master copy} and redirect all reads/writes to that object through \textit{forwarding pointers}.

The second one is to prevent data-race pitfalls since tasks are mutating simultaneously. Immutable reads are safe. Mutable reads, non-pointer writes, and non-promoting writes are relatively safe since they won’t trigger promotion, thus we only need to synchronize the heap one at a time during the process of finding master copy. The most tricky cases are the promoting writes, where we have to lock the entire path from the pointer located heap all the way to its master copy then perform another copy (and create a new master copy), so that no other thread can read a false forward pointer in the middle of promotion or performing another promotion at the same time.

6.3 Thoughts on overall design

6.3.1 The similarity of multi-spaces collector

One of the main motivations of having hierarchical heap that mirror the computations are allowing that each leaf heap (or sub-tree in theory) can be collected individually and in parallel. Besides of taking advantages of multi-processors as parallel collectors, this also reminds me generational collectors in general. The idea of partitioning the heap as multiple spaces is similar: each heap is designed to has its dedicated functionality, so that we can have fine-grained control on the time and space costs that we need to pay for collections. The hierarchical setting can be seen as a generational setting whose ages are depth, and the leaf heaps correspond to the nursery spaces. During each collection cycle, the survivors are promoted.

Then the natural concerns would be handling cross-space pointers, more specifically, older-to-younger pointers, or black-to-white pointers. In this case, they correspond to the \textit{down-pointers}. We don’t have this problem in purely functional settings, however it raises in the presence of mutable references. Again, the solution is somewhat "standard": we use a write barrier to detect the occurrences of problematic cases so we can perform some "fixes": In generational collector we record those pointers in remembered set, while here we eagerly perform a promotion.
6.3.2 Prioritize the common use cases

Many performance optimizations in automatic memory management come with prioritizing some more specialized cases and trade-off other cases. Mostly functional programming languages like ML, though having mutable references, still mostly be used functionally and mutate much much rarer than imperative languages. Thus, the design focus on having no or low costs on immutable operations, i.e. optimize for the common use cases of the object languages.

Another places reflecting this kind of prioritization is to assign local heap to each task (thread), so common use cases such as local access can be very efficient. As the related work mentioned, the DLG collector introduced the idea of having processor-local nursery heaps that provide fast local access and users only pay for the synchronization cost of global heap as their uses.
Chapter 7

Course Project - Generational GC with Incremental Old-Gen Collector

7.1 Proposal

7.1.1 Overview

We planned to implement a generational GC on the LangF VM, where the new-gen collector will inherit the Chenny’s copying implementation from the 5th homework project "Generational GC" and we will focus on making old-gen collector a incremental collector (piggy-backed on the mutator thread) with incremental marking and lazy sweeping. Several proposed extensions are discussed below and we will choose some of them to finalize.

7.1.2 Details, Proposed Extensions and Challenges

Incrementality → Mark-Sweep?

Incrementality (and also concurrency) is easier to corporate into a non-moving collector, that probably explained why most of the incremental/concurrent GC builds upon mark-sweep. I guess we will go with mark-sweep as well.

Free-lists Allocation

Using mark-sweep means we have to gave up linear (bump-pointer) allocation and use free-list. We need to choose lookup strategy from either first-fit, next-fit or best-fit.
Write Barriers (WB)

We will tweak then reuse the card-table/cross-map implementation of generational WB from project 5th and corporate a concurrent WB into it.

One optimization would be thinking about how to "merge" them in someways.

Proposed Extension - Size-segregated Free-lists

To optimize the free list lookup speed, it would be beneficial to use multiple free-lists separated by the size of memory chunk they managed (a.k.a segregated-fits allocation).

Proposed Extension - Multiple Pages

Instead of a liner consecutive heap, we might prefer partitioning the heap into many smaller pages, e.g. dedicated large-object space, page size matching some virtual memory size...

I am not exactly sure what’s exact heuristic we used by but I guess this gave us a finer-grained control and opened up some optimization space in general.

Proposed Extension - Backup Mark-Compact

Due to the fragmentaion problem of mark-sweep. It would be a natural extension to have a "full major GC" that stop the world (for simplicity) and perform the compactation based on a heap fragmentaion heuristic.

A optimization (with multiple-paged heap) would be only compact heap page by page, which will reduce the pause time.

Proposed Extension - 2-step Young Generation

Split the young generation into "nursery" and "intermediate" spaces, i.e. a young generational object now need to survive 2 collection cycle (a.k.a 2-steps or 2-aged) to be promoted.

This would be useful to test the "Survival rates with a copy count of 1 or 2." mentioned in GC handbook Figure 9.3 at P118.

Configurability

Providing interfaces to configure heap layout and collectors (e.g. full-sweep vs. lazy-sweep).

Perf Evaluation

Providing performance evaluation with different configurations (and also previous homework projects) to see whether our incremental collector worth the efforts.
Some other profile data specific to this collector (fragmentation ratio) would be necessary as well.

7.2 Writeup

7.2.1 Plan

For simplicity, we chose to use single contiguous heap and single free-list. My plan was set as two phases:

- **Phase 1**: Implemented an non-incremental mark-sweep collectors as the major GC, with important infrastructure including freelist-based allocation, and *Hotspot CMS* style threaded worklist for minor GC (Cheney’s Copying).

- **Phase 2**: Implemented the incremental marking for the mark-sweep collectors, and ideally lazy-sweeping and a backup mark-compact collector.

7.2.2 Status

The current project code can collect new space with the newly-implemented *Hotspot-CMS* style Cheney’s Copying backed by threaded worklist without problems; survivors would be promoted into old space via freelist-based allocation partitioned from left, with first-fit strategy; and the major GC (non-incremental mark-sweep) collector will triggered when the the freelist is exhausted during either pre-tenuring allocation or minor GC (Cheney’s Copying). The collector can reclaim old spaces garbage back to freelist via an eager sweeping without problems.

The problem kicked in when the major GC could still not able to provide enough space, then we need either:

- directly grow the heap on the assumption of fragmented heap.

- trigger the backup GC (Lisp2) and if it’s still not suffice, we then grow the heap on the assumption of after compaction.

However, either case required some "moving" and "pointer adjustments" in the middle of the minor GC. N.B. The current codebase is experimenting with the first approach. The VM will crush after the growing.
7.2.3 What’s completed

Here are a detailed list on what was completed to demonstrate the efforts.

- **Object shape**: a complete set of macros was introduced to work with object shape, including the new 2-bits color tags and utility functions used by freelist-based allocation. They have been moved to *vm-internal.h*.

- **Freelist-based allocation** is completed via threading on the NEXT/FP field of unreachable objects. The freelist is initialized during VM initialization (*vm.c*) by making a placeholder object as the first node (free chunk) in the list, which fit the current first-fit strategy well. It currently implemented a left-partition allocation to be better working with Lisp2 which slides to left during compaction. It help with tracking the *heap utilization* from allocation side.

- **Hotspot-CMS style Cheney’s Copying** The Cheney’s Copying collector used in new spaces has been changed to using a worklist that is threaded via the NEXT/FP field of promoted-but-unscanned objects. A static pointer *Cheney_todo* is introduced as the head pointer. This technique is inspired by Hotspot CMS paper.

- **Non-incremental mark-sweep collector** is completed by reusing the marking part of previous Lisp2 collector. The *MarkSweep_sweep* is the new sweeping phase that supports a simple coalescing. It help with tracking the *heap utilization* from reclamation side.

- **Heap parsing** procedures including Lisp2 and heap growing have been adopted to the new object shape, new color tags and new macros.

- **An effort of heap growing in the middle of minor GC** is presented.

- **Engineering**: code are refactored to be better structured, e.g. macros and traces are both extracted.

- **Comment**: code are well documented, with thoughts and challenges within the comments.
Chapter 8

Appendix A: *LangF*

*LangF* is a strongly-typed, call-by-value, higher-order, polymorphic, functional programming language. The syntax and semantics of *LangF* are similar to other functional programming languages (e.g., Standard ML, Haskell), but with many simplifications and a more explicit type system. *LangF* does not have type inference, exceptions, or a module system. *LangF* does have first-class functions, datatypes, mutable arrays, and explicit and first-class polymorphism.

```plaintext
1 (* list.lgf *)
2 datatype List ['a] = Nil | Cons {'a, List ['a]}
3
4 fun foldl ['a] ['b] (f: 'a -> 'b -> 'b) (b: 'b) (l: List ['a]) : 'b =
5   case l of
6     Nil ['a] => b
7   | Cons ['a] {hd, tl} => foldl ['a] ['b] f (f hd b) tl
8 end
9
10 val rev = fn ['a] =>
11   foldl ['a] [List ['a]]
12     (fn (hd: 'a) (tl: List ['a]) => Cons ['a] {hd, tl})
13     (Nil ['a])
14
15 val n = 4999
16 val sum_n =
17   foldl [Integer] [Integer]
18     (fn (x: Integer) (y: Integer) => x + y)
19     0
20     (tabulate [Integer] n (fn (i: Integer) => i))
```
Chapter 9

Appendix B: Number of hours spent

This appendix includes a list of major tasks completed over the course of this independent study and the approximate number of hours spent.

<table>
<thead>
<tr>
<th>Goals</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>LangF Project4 (value numbering) &amp; other reading</td>
<td>20</td>
</tr>
<tr>
<td>GC Ch. 1-4 &amp; other reading</td>
<td>11</td>
</tr>
<tr>
<td>LangF Project5 (vm codegen) &amp; other reading</td>
<td>30</td>
</tr>
<tr>
<td>GC Ch. 5-8 &amp; other reading</td>
<td>13</td>
</tr>
<tr>
<td>LangF copying GC</td>
<td>16</td>
</tr>
<tr>
<td>GC Ch. 9-12</td>
<td>13</td>
</tr>
<tr>
<td>LangF mark-compact GC</td>
<td>16</td>
</tr>
<tr>
<td>GC Ch. 13-15</td>
<td>12</td>
</tr>
<tr>
<td>LangF generational GC</td>
<td>25</td>
</tr>
<tr>
<td>GC Ch. 16-19</td>
<td>20</td>
</tr>
<tr>
<td>Research Paper 1</td>
<td>5</td>
</tr>
<tr>
<td>Research Paper 2 + Writeup</td>
<td>26</td>
</tr>
<tr>
<td>Research Paper 3</td>
<td>19</td>
</tr>
<tr>
<td>Writeup for Paper 1 and 3</td>
<td>12</td>
</tr>
<tr>
<td>Course Project Proposal</td>
<td>2</td>
</tr>
<tr>
<td>Course Project</td>
<td>22</td>
</tr>
<tr>
<td>This Report</td>
<td>22</td>
</tr>
<tr>
<td>Sum</td>
<td>284</td>
</tr>
</tbody>
</table>


[7] [Online]. Available: https://www.amazon.com/%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6%E7%9A%84%E7%AE%97%E6%B3%95%E4%B8%8E%E5%AE%9E%E7%8E%B0-%E6%97%A5-%E4%B8%AD%E6%9D%91%E6%88%90%E6%B4%8B%E6%89%B8%E7%9B%B8%E5%B7%9D%E5%85%89/dp/B01JZS0AO8


