N. Corbyn

Practical Static Memory Management

Computer Science Tripos - Part II

King's College

May 8, 2020

Declaration of Originality

I, Nathan Corbyn of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Nathan Corbyn of King's College, am content for my dissertation to be made available to the students and staff of the University.

Signed

Date

Proforma

Candidate number:	**2402B**
College:	**King's College**
Project Title:	**Practical Static Memory Management**
Examination:	**Computer Science Tripos - Part II**
Year:	**2020**
Word Count:	**11996**[1]
Final Line Count:	**10403**[2]
Project Originator:	The dissertation author
Supervisor:	Prof A. Mycroft

Original Aims of the Project

The original aim of the project was to investigate the practical viability of a recently published approach to static memory management, ASAP. This involved implementing three data-flow analysis passes, a compiler transformation, an LLVM code generator and carrying out a direct performance comparison between code managed using this approach and code managed by the Boehm-Demers-Weiser garbage collector.

Work Completed

I produced a near-complete implementation of ASAP: two of its three analysis passes and its compile-time transformation were implemented in full. I implemented an LLVM back-end for the technique, and used this to gather performance data on three separate benchmarks when managed using ASAP, the Boehm-Demers-Weiser garbage collector and no memory-management strategy. I also investigated ASAP's usability when compared to Rust and developed an extension to the theory behind ASAP to reduce the complexity of its analyses. Success criteria were met up to implementing the third data-flow analysis pass, which was not necessary for reasons discovered during the project.

Special Difficulties

None.

[1]Word-count produced using `texcount` (`https://app.uio.no/ifi/texcount/`).
[2]Lines of Rust, Python and μ-Mitten counted using `loc` (`https://github.com/cgag/loc`).

Contents

List of Figures

1 — Introduction

A recently published PhD thesis (Proust, 2017) describes a novel approach to memory management, *as-static-as-possible memory management* (ASAP)[1]. Proust's work is primarily a theoretical contribution with some initial results from simulation. In his concluding remarks, Proust states

> **Future work** In order to test ASAP on real world programs, a full implementation (rather than the prototype described in Chapter 7) would need to be implemented.

My project is an attempt to carry out this future work. That is, to build a complete machine-level implementation of ASAP, and to test it on real-world programs.

This chapter presents background material in the wider topic of memory management and describes the principal contributions of this project. Chapter 2 gives an in-depth exposition of ASAP and the static analysis techniques on which it is built. Chapter 3 discusses how these techniques were instantiated in a native-code compiler alongside an extension I made to Proust's theory. Finally, chapters 4 and 5 present data collected from code generated by this compiler and summarise my findings.

1.1 Background

In order for programmers to implement and interact with dynamic data-structures (i.e., linked-lists and trees), programming languages must allow for the dynamic allocation of memory. Typically, this is done through a language's runtime system. Despite enabling the use of dynamic data-structures, dynamic allocation comes with its own problems. Specifically, as the memory of a physical system is always finite, a process cannot continue to dynamically allocate indefinitely without previously allocated memory being reclaimed. However, if memory is reclaimed while still in use, a wide variety of unintended behaviours can occur. Thus, it is critical that memory blocks are only reclaimed when a process no longer requires them. Strategies for determining which memory blocks are safe to reclaim and when to reclaim them are known collectively as *memory-management strategies*. Many such strategies exist and are instantiated in a variety of programming languages, but are not without trade-offs.

The designers of C/C++ opted to place the burden of memory management on the programmer. That is, the designers exposed primitives (`malloc()`/`free()` in C, and `new`/`delete` in C++) for explicitly allocating and deallocating memory. It is then up to the programmer to use these primitives to correctly reclaim allocated memory. This approach has many performance advantages over alternative approaches; however, the misuse of these primitives has been the root cause of many high-profile software bugs. In summary, the designers of C/C++ traded safety for performance. However, with a rising demand for correctness in software, language designers have been pushed to search for safer memory-management strategies which alleviate this burden from the programmer.

This search led to the development of *automatic garbage collectors*, which form the basis of the memory-management strategies of the majority of modern programming languages. Automatic garbage collectors operate as part of a language's runtime system. Typically, they are instantiated as sub-processes which automatically detect unreachable memory blocks (*garbage*) and reclaim (*collect*) them. However, there is a huge body of research concerned with automatic garbage collection and there are many possible architectures. These include reference counting, tracing, generational and hybrid schemes.

Although automatic collectors are safe (i.e., never deallocate memory that is still in use), and are generally[1] complete (i.e., eventually deallocate all unused memory), they come with a high performance cost. This cost can include spawning dedicated sub-processes and useful threads being suspended while large collections occur. In addition to this, garbage collectors are themselves very complex pieces of software and therefore massively increase the size of a programming language's runtime when incorporated. This makes garbage-collected languages unsuitable for systems programming, particularly for embedded systems or systems where verifiable real-time behaviour is required.

Rust[3] is a relatively young systems programming language which adopts another alternative approach. As a systems programming language, one of its key design requirements was to have a minimal runtime system, thus preventing the use of a large automatic collector. Rather than leave programmers to manage memory manually (as in C/C++), Rust's designers applied a hybrid of type-theoretic techniques (namely, linear typing and region-based programming) to allow the Rust compiler to statically infer when values are ready to be deallocated and insert code to do so. However, this type-driven approach to memory management, termed Rust's *ownership system*, is not without cost. It places many restrictions on what can be done with data, making programming in Rust difficult. In fact, the learning curve associated with the ownership system is frequently criticised.

ASAP presents a potential solution to this issue. That is, a compiler-driven approach to memory management that places no restrictions on how data is used in a program. Future programming languages employing such an approach would be able to avoid the issues creating Rust's steep learning curve while, hypothetically, maintaining applicability to systems programming. Determining if this is truly the case is the primary motivation of this project.

1.2 Contribution

The principal contribution of this work is to provide the first machine-level implementation of ASAP. This implementation is based on my own extension to Proust's theoretical work and supports multiple memory-management strategies, from which one can be selected via a compiler flag. This has enabled direct performance comparison over a collection of real-world benchmarks between ASAP and other memory-management techniques for the first time. I find that, as-is, ASAP is not yet ready for adoption; however, the data I present indicates potential moving forwards.

[1]Simple reference counting schemes do not achieve completeness but are safe. Conservative collectors, by their nature, trade completeness for safety[4].

2 — Preparation

This chapter discusses work done in preparation for the project. As this project is based on recently published research, the vast majority of this work was centred on gaining familiarity with the source material. However, it also included making several key design decisions regarding the nature of the compiler.

Section 2.1 discusses design decisions surrounding native-code generation, while section 2.2 describes the reasoning behind my choice of source language and the design of my compiler's intermediate representation (IR). Section 2.3 introduces *data-flow analysis*, a popular static-analysis technique used heavily by ASAP. A detailed exposition of ASAP itself is then given in section 2.4. Finally, sections 2.5 and 2.6 give some closing remarks regarding my engineering methodology and the project's starting point.

2.1 Generating Native Code

From the outset, I decided that my compiler should target LLVM[2] as a back-end. This immediately solved the problem of generating native code, as open source tools such as `clang` support generating highly-optimised native code from LLVM's IR. However, there was another key motivator for this decision. ASAP is a phase in the middle-end of a compiler. This leaves all code it generates open to standard optimisations (e.g., tail-call elimination). This is something Proust observes, but does not investigate. Hand-writing such optimisations was clearly beyond the scope of this project and therefore, in order to investigate the impact of optimisation, LLVM, with its suite of in-built optimisations, seemed like an obvious choice.

2.2 μ-Mitten

For my source language, I chose to adopt a subset of the Rust programming language. Acknowledging Proust's IR (μL), I named this μ-Mitten. At this point, one might ask 'if Rust can already manage memory statically, what use does a fragment of Rust have for ASAP?'. Rust's ownership system enforces immensely strong compile-time invariants in order to permit static memory management. With this project, I was particularly interested in showing that ASAP would permit static management of this fragment of Rust, without requiring these invariants. To do this, I wanted to exhibit correct μ-Mitten programs which are rejected by Rust's type system due to simplifications μ-Mitten makes to the programmer's view of memory. Taking μ-Mitten to be a syntactic fragment of Rust made this trivial.

A BNF grammar for the abstract syntax of μ-Mitten is given in appendix A. As shown, the only primitive type supported by μ-Mitten is the unsigned 64-bit integer type `u64`. All other types must be defined as a `struct` (record) or `enum` (sum-type). μ-Mitten does not permit higher-order functions, nested matching or mutability (as Rust does), but does support mutually recursive functions (as μL does not).

Once I had determined my source language, I began designing a suitable IR. The IR I settled on is a *static single assignment* (SSA) form; however, as the control flow structures of μ-Mitten do not permit merging control-flow paths, the IR has no need for the ϕ-nodes usually present in SSA IRs. The decision to use SSA was driven by my knowledge that LLVM's IR is an SSA form, thus simplifying the task of generating LLVM code from my IR. A BNF grammar for the IR is given in appendix B.

To illustrate the relationship between the IR and μ-Mitten, a side-by-side comparison of the two is given in figure 2.1. The IR presented in figure 2.1 was generated by the compiler from the source code shown; however, temporary names (*local indices*) have been altered to aid readability. As can be seen, many of the high-level constructs present in μ-Mitten have been compiled away as syntactic sugar, while local indices have been fixed to all intermediate results.

```
fn fib(n: u64) -> u64 {
  if n == 0 {
    0
  } else if n == 1 {
    1
  } else {
    fib(n - 1) + fib(n - 2)
  }
}
```

```
.fib {
  %n <- arg_0
  %t0 <- 0
  %t1 <- %n == %t0
  match %t1 {
    0 => {
      %t2 <- 1
      %t3 <- %n == %t2
      match %t3 {
        0 => {
          %t6 <- %n - %t2
          %t7 <- .fib(%t6)
          %t8 <- 2
          %t9 <- %n - %t8
          %t10 <- .fib(%t9)
          %t11 <- %t7 + %t10
          ret %t11
        },
        %t5 => {
          ret %t2
        },
      }
    },
    %t4 => {
      ret %t0
    },
  }
}
```

Figure 2.1: Comparison between high-level syntax and equivalent IR

2.3 Data-Flow Analysis

Data-flow analysis is a well-known technique in compiler construction for statically inferring information about the runtime properties of a program[6]. The core idea behind data-flow analysis is that it is possible to build a set of *data-flow equations* relating data-flow information at each program point which are then solved iteratively. I adopt the same terminology as Proust and refer to the information collected at a given program point as the *decoration* at that point.

An example of a typical data-flow analysis pass is *live-variable analysis* (LVA). An LVA determines the set of variables that may be live at each program point. The decorations of an LVA are therefore sets of the names of live variables.

To ensure that solutions exist, we require two properties of an analysis. First, its decorations must form a domain under some partial-order $\sqsubseteq$ with bottom element $\bot$. Second, the application of its data-flow equations must be monotone with respect to $\sqsubseteq$. That is, assuming the analysis is captured as the set of data-flow equations ψ, when viewed as an operator, the application of ψ to any partial solution d satisfies

$$d \sqsubseteq \psi(d)$$

Using this property, we can build an ascending chain of partial solutions as follows

$$\bot \sqsubseteq \psi(\bot) \sqsubseteq \psi(\psi(\bot)) \cdots \sqsubseteq \psi^n(\bot) \sqsubseteq \cdots \tag{2.1}$$

Tarski's fixed-point theorem tells us that the least solution to ψ is then the least upper-bound of this chain. This can be computed iteratively by initialising some state to the value of $\bot$ and repeatedly applying ψ; however, this does not guarantee termination.

For termination, we further require that each ascending chain, such as 2.1, is eventually constant. That is, there exists some N such that for all $j \geq N$, $\psi^j(\bot) = \psi^{j+1}(\bot)$. This is referred to as the *ascending-chain condition* and is equivalent to asserting the domain of an analysis is of finite-height[7]. When iterating, we use this fact to inform termination: if an application of ψ results in no change, we can terminate.

Returning to LVA as an example, it is clear that its decorations form a domain with $\subseteq$ as the order and $\varnothing$ as the bottom element. Furthermore, the decoration at a program point π can never exceed the set of variables in scope at π, hence for any finite program this domain is finite-height. Thus, termination is guaranteed.

2.3.1 Inter-Procedural Analysis

In its usual form, data-flow analysis is applied to procedures in isolation. This is known as *intra-procedural* data-flow analysis. However, a whole-program variant known as *inter-procedural* data-flow analysis exists[1]. It is this variant that Proust uses to define ASAP's data-flow properties. Rather than simply relating the decorations within a procedure, inter-procedural analysis also propagates data-flow information between procedures across call and return points.

Inter-procedural data-flow information is carried in two pieces of information computed for each procedure. The first is a procedure's *summary* which carries information from callees to callers. In the case of an inter-procedural LVA, when analysing a call instruction, we only need to consider the formal parameters that are live in the callee as live in the caller. Thus the LVA summary of a procedure would carry information about which of its formal parameters are live at its entry point. The second is a procedure's *amalgamated call-context* which carries information from callers to callees. Again, taking an LVA as an example, we only need to consider the return value of a procedure as live if it is live in any of the procedure's callers. Thus, the amalgamated LVA-context of a procedure carries information about whether its return value is live immediately after any of its call points.

2.4 ASAP

In his thesis, Proust develops a theoretical framework within which he defines three data-flow properties from which heap liveness can be statically approximated. ASAP uses this static approximation to inform a whole-program transformation which statically inserts code to free inaccessible heap blocks. Presented here is a brief exposition of the theory developed by Proust and its instantiation in ASAP.

2.4.1 Paths

Memory management, as described in chapter 1, is concerned with management of data in the heap. Stack-allocated values are considered as roots from which heap blocks may be accessed; however, we assume that stack memory will be implicitly managed by a language's calling convention and is therefore of no concern.

For ASAP to reason statically about the structure of data in the heap, it requires an efficient compile-time abstraction of the heap. To this end, Proust developed his theory of *paths*, which I have made an attempt to describe here. I have moved to simplify some of Proust's formalisms where appropriate; however, this exposition is substantially the same as Proust's.

Informally, paths capture the idea of a pattern of access in the heap, originating at a value of one type and yielding a value of another. Given two μ-Mitten types $\tau, \tau' \in \mathbf{Type}$, we can define the set of paths beginning at a value of type τ and ending at a value of type τ' ($\mathbf{Path}_{\tau \to \tau'}$) inductively as follows.

$$\frac{\tau = \tau'}{\epsilon \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Empty}) \qquad \frac{p \in \mathbf{Path}_{\tau\to\tau'} \quad \tau = \tau'}{p^* \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Star})$$

$$\frac{\tau = \cdots + D(\tau') + \cdots}{D \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Variant}) \qquad \frac{\tau = \{\cdots, F : \tau', \cdots\}}{F \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Field})$$

$$\frac{p \in \mathbf{Path}_{\tau\to\tau''} \quad q \in \mathbf{Path}_{\tau''\to\tau'}}{p \cdot q \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Seq}) \qquad \frac{p \in \mathbf{Path}_{\tau\to\tau'} \quad q \in \mathbf{Path}_{\tau\to\tau'}}{p + q \in \mathbf{Path}_{\tau\to\tau'}}\ (\textsc{Alt})$$

As can be seen, paths have the same syntax as regular expressions, complete with sequencing ($\cdot$), alternation ($+$), and repetition ($*$). In fact, paths can be viewed as regular expressions with the key difference that rather than the atoms of a path being symbols in an arbitrary alphabet, they represent atomic de-structurings of structured data. Variant captures projecting a variant from a sum-type, and Field captures projecting a field from a record. A string in the language denoted by a path represents a sequence of projections, with the output from each becoming the input to the next.

Given a heap configuration $\eta \in \mathbf{Heap}$, a stack configuration $\sigma \in \mathbf{Stack}$, a field projection operator $\pi_F : \mathbf{Loc} \to \mathbf{Loc}$ and a variant projection operator $\pi_D : \mathbf{Loc} \to \mathbf{Loc}$, we can determine the set of reachable locations denoted by a path when starting from some location $l \in \mathbf{Loc}$. The staged function $Z : \mathbf{Loc} \times \mathbf{Path}_{\tau\to\tau'} \to \mathbf{Heap} \times \mathbf{Stack} \to \mathcal{P}(\mathbf{Loc})$ performs this operation and is defined as follows.

$$\begin{aligned}
Z(l, \epsilon)(\eta, \sigma) &= \{l\} \\
Z(l, F)(\eta, \sigma) &= \begin{cases} \varnothing & \text{if } \tau' = \mathit{u64} \\ \{\pi_F(l)\} & \text{otherwise} \end{cases} \\
Z(l, D)(\eta, \sigma) &= \begin{cases} \varnothing & \text{if } \tau' = \mathit{u64} \\ \{\pi_D(l)\} & \text{otherwise} \end{cases} \\
Z(l, p \cdot q)(\eta, \sigma) &= \bigcup \{Z(l', q)(\eta, \sigma) \mid l' \in Z(l, p)(\eta, \sigma)\} \\
Z(l, p + q)(\eta, \sigma) &= Z(l, p)(\eta, \sigma) \cup Z(l, q)(\eta, \sigma) \\
Z(l, p^*)(\eta, \sigma) &= \bigcup_{n \in \omega} Z_n \text{ where } Z_0 = \{l\} \text{ and } Z_{n+1} = \bigcup_{l' \in Z_n} Z(l', p)(\eta, \sigma)
\end{aligned}$$

Although η and σ are never explicitly referenced in the definition, they are implicit in the application of π_F and π_D. Also note that the sets **Heap** and **Stack** are infinite, and the subset of possible configurations at each program point π ($\mathbf{Config}_\pi = \mathbf{Heap}_\pi \times \mathbf{Stack}_\pi$) is, in general, undecidable. However, given any variable $x \in \mathbf{Var}$, we know its stack location $\sigma(x) \in \mathbf{Loc}$. Thus, given a path $p \in \mathbf{Path}_{\tau\to\tau'}$ with $\tau, \tau' \in \mathbf{Type}$ such that $\Gamma \vdash x : \tau$, we can compute $Z(\sigma(x), p)$ at compile time, and fill in η and σ at runtime when they are known.

To more succinctly refer to these concepts, Proust introduces some terminology and notation. Given types $\tau, \tau' \in \mathbf{Type}$ and a path $p \in \mathbf{Path}_{\tau\to\tau'}$, Proust writes $\tau.p$ to stand for τ'

(i.e., p can be applied to τ to yield τ'). Further, given a program variable $x \in \mathbf{Var}$ such that $\Gamma \vdash x : \tau$, Proust refers to the pair $z = (x, p)$ as a *zone*, with the set of all zones written as $\mathbf{Zone}_{\tau \to \tau'} \subseteq \mathbf{Var} \times \mathbf{Path}_{\tau \to \tau'}$. Finally, rather than write $Z(\sigma(x), p)$ in full, Proust typically writes $Z(x, p)$ or just $Z(z)$.

The principal use of Z is to define the path subsumption relation $\preceq$. Strictly speaking, for each $\tau, \tau' \in \mathbf{Type}$, we have a subsumption relation $\preceq_{\tau \to \tau'} \subseteq \mathbf{Path}_{\tau \to \tau'} \times \mathbf{Path}_{\tau \to \tau'}$ defined over $p, p' \in \mathbf{Path}_{\tau \to \tau'}$ as follows.

$$p \preceq_{\tau \to \tau'} p' \iff \forall l, \eta, \sigma.\ Z(l, p)(\eta, \sigma) \subseteq Z(l, p')(\eta, \sigma)$$

More intuitively, this says that in any runtime configuration, the set of locations reachable by following any string accepted by p is included within that of p' (read p' *subsumes* p). In general, the subscript of $\preceq$ is clear from the context and is omitted.

```
struct Unit {}

struct Cell {
  head: u64,
  tail: List,
}

enum List {
  Nil(Unit),
  Cons(Cell),
}
```

Figure 2.2: μ-Mitten type declaration for a linked list

To give some concrete examples of paths, consider the μ-Mitten type declaration for a linked list given in figure 2.2 (or alternatively its corresponding type-graph shown in figure 2.3). Possible paths originating from the type *List* would then include

$$\begin{aligned}
\mathit{Cons} \cdot \mathit{head} \quad & \text{(a list's immediate head)} \\
(\mathit{Cons} \cdot \mathit{tail})^* \cdot \mathit{Cons} \quad & \text{(a list's cons cells)} \\
\mathit{Cons} \cdot \mathit{head} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons} \cdot \mathit{head} \quad & \text{(the first two elements of a list)} \\
(\mathit{Cons} \cdot \mathit{tail})^* \cdot \mathit{Cons} \cdot \mathit{head} \quad & \text{(all elements of a list)} \\
(\mathit{Cons} \cdot \mathit{tail})^* \cdot \mathit{Nil} \quad & \text{(the list's terminator)}
\end{aligned}$$

$$\begin{tikzcd}
\mathit{Unit} & \mathit{List} \arrow[l, "\mathit{Nil}"'] \arrow[r, bend left, "\mathit{Cons}"] & \mathit{Cell} \arrow[l, bend left, "\mathit{tail}"] \arrow[r, "\mathit{head}"] & \mathit{u64}
\end{tikzcd}$$

Figure 2.3: Type-graph corresponding to the declaration in figure 2.2

2.4.2 *Shape*, *Share* and *Access*

Within this theory of paths, Proust defines three data-flow properties: *Shape*, *Share* and *Access*. *Shape* and *Share* are closely related, interdependent properties, while *Access* is dependent on

Shape. As defined by Proust, these properties are three-valued; however, I observed that as used by ASAP, all three are used as 'may' analyses and I treat them exclusively as such. As described, each of these properties is undecidable but can be safely over-approximated as described by Proust. It is these approximations that ASAP uses to inform its transformation.

Let π be an arbitrary program point. The *Shape* decoration at π, $\mathit{Shape}(\pi) \subseteq \mathbf{Zone} \times \mathbf{Zone}$, is a relation over zones such that for all zones $(x, p), (y, q) \in \mathbf{Zone}$ the following holds.

$$((x,p),(y,q)) \in \mathit{Shape}(\pi) \iff \exists(\eta,\sigma) \in \mathbf{Config}_\pi.\ Z(x,p)(\eta,\sigma) \cap Z(y,q)(\eta,\sigma) \neq \varnothing$$

That is to say, there is some machine configuration possible at π such that the set of locations reachable from (x, p) and (y, q) overlap (*external* aliasing). Note that this is only possible for zones whose destination types are equal. The *Shape* summary of a function f captures the possibility of aliasing between the return values of f and its formal parameters at the points at which f returns. Equally, the amalgamated *Shape*-context of f captures the possibility of aliasing between the formal parameters of f at the points at which f is called.

Similarly, the *Share* decoration at π, $\mathit{Share}(\pi) \subseteq \mathbf{Zone}$, is a set of zones such that for any zone $(x, p) \in \mathbf{Zone}$ the following holds.

$$\begin{aligned}(x,p) \in \mathit{Share}(\pi) \iff & \exists p', p'' \preceq p.\ \exists(\eta,\sigma) \in \mathbf{Config}_\pi. \\ & p' \neq p'' \wedge Z(x,p')(\eta,\sigma) \cap Z(x,p'')(\eta,\sigma) \neq \varnothing\end{aligned}$$

Intuitively, this says that there is some aliasing within p (*internal* aliasing). The *Share* summary of a function f captures internal aliasing within its return values, and the amalgamated *Share*-context of f captures internal aliasing within its formal parameters. Note that external aliasing at one program point can imply internal aliasing at another (e.g., when two externally aliasing zones become fields of the same record) and vice versa, hence the interdependence of *Shape* and *Share*.

The *Access* decoration at π is simply the set of zones that may be accessed at any program point π' reachable from π. Note that explicit access to one zone constitutes an implicit access to any aliasing zone, thus *Access* depends on *Shape*. The amalgamated *Access*-context of a function f records access to the return values and formal parameters of f at the points at which f returns, while the *Access* summary of f records access to the formal parameters of f at the entry point of f. This includes all zones present in its amalgamated *Access*-context.

2.4.3 Scan and Clean

Clean is the name given to ASAP's whole-program transformation. In its simplest form, Clean operates by computing two sets at each program point: the *matter* set, which contains all zones that must not be deallocated; and the *anti-matter* set which contains those which may be deallocated. With these sets computed, Clean generates code implementing the following series of operations:

1. For each zone in the matter set, mark each reachable location within that zone as 'safe'.
2. For each zone z in the anti-matter set, for each location $l \in Z(z)(\eta, \sigma)$ not marked as 'safe' or 'freed', free l and mark it as 'freed'.
3. Reset the marks.

Scan is a more primitive compile-time function used as part of this transformation. Its role is to generate the code required to traverse the memory associated with a given zone and perform marking and freeing operations. My implementation of SCAN differs greatly from the definition given by Proust for reasons discussed in chapter 3 and therefore I do not present his definition here.

Given successive program points π_1 and π_2, the matter set M and the anti-matter set A at π_2 can be computed as follows.

$$\begin{aligned} M &= \mathit{Access}(\pi_2) \\ A &= \mathit{Access}(\pi_1) \setminus (\mathit{Access}(\pi_2) \cup \mathit{protected}(\pi_1, \pi_2)) \end{aligned}$$

Intuitively, this reads as saying that nothing that may be accessed at or after π_2 may be deallocated, while anything that may have been accessed at π_1 but not at or after π_2 is open to deallocation. The set $\mathit{protected}(\pi_1, \pi_2)$ is used to exclude zones that have already been deallocated. For example, if π_1 and π_2 are the program points immediately before and after a call instruction, their protected set will include zones that have already been deallocated by the target of the call.

In practice it is enough to take M as the set of zones accessed at π_2 that may alias with those in the anti-matter set, as they are not at risk fo deallocation otherwise. This is captured mathematically as follows.

$$M = \{z \in \mathit{Access}(\pi_2) \mid \exists z' \in \mathbf{Zone}.\ (z, z') \in \mathit{Shape}(\pi_2)\}$$

Beyond this trivial optimisation, Proust describes an advanced optimisation for matter and anti-matter sets he calls *trimming*; however, I worked on the basis that all such optimisations would be ignored until the success criteria of the project had been met. In this case, the optimisation is left as future work.

2.5 Software Engineering Methodology

I chose to adopt an agile approach to implementation. Key issues and development milestones were tracked as *epics* and each two weeks, I self-assigned tasks which I would then review at the end of the two-week cycle (*sprint*).

I made use of standard software engineering tools such as Git version control and continuous integration. I maintained a collection of regression tests which were run on every commit. These were usually sample programs which could be compiled and run against an expected outcome or malformed programs that the compiler should correctly reject.

Where appropriate, I made use of test-driven techniques. That is, I would write initially failing tests and then extend the compiler to support the necessary features to make them pass, while ensuring no other tests began to fail. However, this was not always possible, particularly when developing analyses for which reference results could not be computed by hand. In such cases, I wrote end-to-end tests after the fact (e.g., testing code generated by transformations dependent on the analyses).

2.6 Starting Point and Legal Concerns

No code included in the final version of this project's repository was written before the project was started.

Before starting this project, I had limited interaction with LLVM as part of a small preliminary investigation into this project's viability. My investigation informed the decision that, rather than interact with LLVM's C API directly, I should use the library Inkwell[1], which safely wraps LLVM's C API in an ergonomic Rust API. This decision was made to avoid having to deal with managing the memory associated with LLVM and instead focus on the core ideas behind the project. Both LLVM and Inkwell are made available under the Apache License 2.0[2], and thus use is freely permitted.

[1]`https://github.com/TheDan64/inkwell/`

[2]`http://www.apache.org/licenses/`

3 — Implementation

This chapter describes the implementation work completed as part of the project. For this project to make a meaningful comparison between ASAP's performance and that of other memory-management strategies, it was essential that all factors beyond the memory-management strategy were tightly controlled. Fortunately, ASAP sits entirely within the middle-end of a compiler. This means that it is possible to construct a conventional compiler pipeline which can then be retroactively extended to support ASAP without modification. In doing so, we ensure the only differences between code generated with and without ASAP are attributable to ASAP's transformations, guaranteeing fairness.

Thus, implementation work was broken down into two key phases. First, I implemented a 'core' pipeline, compiling μ-Mitten code into LLVM directly, only supporting memory management via the Boehm-Demers-Weiser garbage collector[5]. This work is described in section 3.1. I then worked on extending this pipeline to support ASAP. Section 3.2 describes my implementation of a general-purpose data-flow analysis framework, while section 3.3 describes how this was used to implement ASAP's analyses specifically. Section 3.4 describes the treatment of paths and includes an extension I made to Proust's theory. The implementations of SCAN and CLEAN are then described in section 3.5. Finally, work done to automate testing and benchmarking of the project is described in section 3.6.

An overview of the high-level structure of the repository is given in figure 3.1 (red indicates open source libraries and tools).

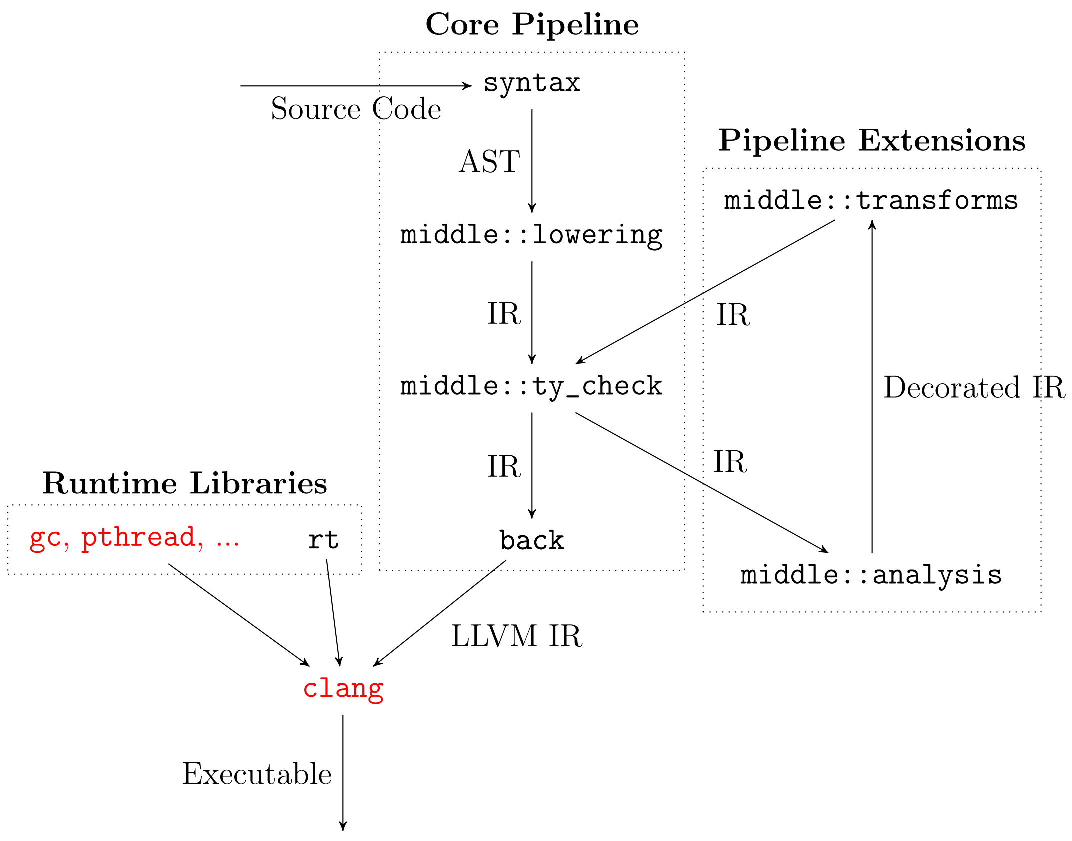

Figure 3.1: Repository overview

3.1 Core Pipeline

As illustrated, the core pipeline is broken down into four modules: the front-end (`src/syntax/`); a lowering module (`src/middle/lowering/`), which constructs the IR from a given AST; a simple type system (`src/middle/ty_check.rs`); and a code generating back-end (`src/back/`). Although not the focus of the project, each presented its own engineering challenges and I have made an effort to highlight the most interesting.

3.1.1 Front-End

When building a front-end for a research language, it is typical to use a parser generator rather than give a hand implementation. However, as a de-facto parser generator has not yet emerged in the Rust ecosystem and as the diagnostics produced from such generators are typically somewhat cryptic, I opted to hand write the compiler's front-end. The parser (`src/syntax/parse.rs`) is a simple recursive-descent parser producing an AST as defined in `src/syntax/ast.rs`. The lexer (`src/syntax/lex.rs`) is similarly conventional and produces tokens as defined in `src/syntax/token.rs`.

Although more time-consuming in the short term, the decision to hand write the front-end proved invaluable in the long term. By having the parser produce spanning information and passing this to the Rust diagnostic formatting library `codespan`[1], I was able to have the compiler produce rich diagnostics with minimal effort. Producing the benchmark programs used for evaluation involved writing several hundred lines of μ-Mitten code and the diagnostics produced by the compiler were indispensable while doing so. An example is given in figure 3.2 (all formatting is performed by `codespan`).

```
error: malformed struct initialiser

   +-- src/test/compile-fail/multiple_field_instantiations.mmtn:8:15 ---
   |
 8 |     let foo = Foo { bar: 0, bar: 1 };
   |               ^^^^^^^^^^^^^^^^^^^^^^ attempted to initialise the same field twice
   .
 8 |     let foo = Foo { bar: 0, bar: 1 };
   |                                  - attempted to initialise 'bar' here
   .
 8 |     let foo = Foo { bar: 0, bar: 1 };
   |                          - but it was already initialised here
   |
```

Figure 3.2: A sample diagnostic

3.1.2 Lowering

Once the AST has been built, it must be lowered into the compiler's IR. Lowering consists of two stages: lowering type declarations to build a *type session* (`src/middle/lowering/ty_ctx.rs`) and lowering function declarations to build IR function definitions (`src/middle/lowering/ctx.rs`). The type session carries information about the structures of types and is essential throughout type-checking, analysis and code generation. Its underlying data-structure (`src/ty.rs`) takes care to structurally unique function types and the `u64` type, while handling record and sum-types nominally.

[1] `https://github.com/brendanzab/codespan`

In order for the compiler to efficiently reference components of the IR throughout analysis and transformation, the in-memory representation of the IR (`/src/middle/ir.rs`) makes heavy use of indices in place of names used in the AST. For example, rather than referring the fields of a record by name, the IR refers to them using an integral index. As the IR introduces indices for a variety of purposes, it was not reasonable to deal with them as pure integers. Thus, I employed a Rust design pattern: opaque index types.

First, a trait[2] is introduced (as shown in figure 3.3), which can be implemented by a type to indicate that it is an index type.

```rust
pub trait Idx: 'static + Copy + Eq + Hash + fmt::Debug {
  fn index(&self) -> usize;

  fn new(index: usize) -> Self;
}
```

Figure 3.3: The `Idx` trait (`src/common/idx.rs`)

Using this trait, it is possible to define a wrapper around the standard library's vector type, `Vec`, as shown in figure 3.4. All of the methods supported by `Vec` can be implemented by this opaque wrapper. However, this is done in such a way that only its type parameter can be used to index the underlying vector. The result is an efficient type-safe map data-structure. As a concrete example, an implementation of `IdxVec::push` is given in figure 3.5.

```rust
#[derive(Clone, PartialEq, Eq, Hash)]
pub struct IdxVec<I, T> {
  raw: Vec<T>,
  // Rust does not permit type parameters to go unused,
  // so we use a marker to ignore the index type
  _phantom: PhantomData<fn(&I)>,
}
```

Figure 3.4: The `IdxVec` data structure (`src/common/idx_vec.rs`)

```rust
impl<I, T> IdxVec<I, T>
where
  I: Idx,
{
  pub fn push(&mut self, elem: T) -> I {
    let index = self.raw.len();
    self.raw.push(elem);
    I::new(index)
  }
}
```

Figure 3.5: `IdxVec::push` (`src/common/idx_vec.rs`)

As Rust statically resolves calls to trait methods, after inlining, use of this pattern compiles away to uses of `usize` and the `Vec` type. This means that performance is preserved while allowing Rust's type system to catch a variety of common mistakes.

[2]Traits are similar to interfaces or type classes in other languages.

3.1.3 Type-Checking

The next phase in the core pipeline is type-checking. The implementation (`src/middle/ty_-check.rs`) checks definitions recursively using the type session built by the lowering phase.

Rather than operating over the front-end's AST, the type-checker works purely with the IR. A consequence of this is that for meaningful diagnostics to be generated, the lowering phase must carry forward spanning information generated by the front-end into the IR. However, there is great incentive to type-check the IR rather than the AST. As ASAP's cleaning transformation is type-safe, by type-checking the IR, the type-checker doubles as a general-purpose IR verifier which can be applied to transformation results as shown in figure 3.1.

3.1.4 Generating LLVM IR

Due to the similarities between μ-Mitten's IR and LLVM IR, the back-end for the core pipeline was also straightforward to implement.

LLVM IR is strongly typed. Thus, it was necessary to determine how μ-Mitten types should be represented in LLVM. The following definition of $\mathit{compile} : \mathbf{Type} \to \mathbf{LLVM}$ captures the approach I took. `i64` denotes LLVM's 64-bit signed integer type; braces are used in indicate structures; and the post-fix type-constructor `*` builds the type of a pointer to its operand type.

$$\begin{aligned}
\mathit{compile}(u64) &= \texttt{i64} \\
\mathit{compile}(D_1(\tau_1) + \cdots + D_n(\tau_n)) &= \texttt{\{ i64, i64 \}*} \\
\mathit{compile}(\{F_1 : \tau_1 + \cdots + F_n : \tau_n\}) &= \texttt{\{ } \mathit{compile}(\tau_1)\texttt{, } \cdots \texttt{, } \mathit{compile}(\tau_n) \texttt{ \}*}
\end{aligned}$$

Note that the representation of sum-types is independent of the types participating in the sum. As each μ-Mitten value is, as in Java, either a primitive or a reference, values are guaranteed to have a fixed 64-bit width. Thus, the representation of any given sum-type contains one `i64` to hold its discriminant and another which can be cast to a pointer to the appropriate LLVM type when projecting one of its variants. Often, it is necessary to know the LLVM representation of an un-boxed value of a given type $\tau \in \mathbf{Type}$, which I will write as $\mathit{compile}_U(\tau)$.

Let-bindings and the binary operators supported by μ-Mitten have direct analogues in LLVM and therefore compiling these was trivial. However, when initialising a record, memory must first be allocated and the values of its fields must be copied individually. Similarly, memory must be allocated for sum-types; however, in this case, the sum's body must be explicitly cast to an `i64` to satisfy LLVM's type system. When using Boehm-Demers-Weiser, memory is allocated using the allocation function `GC_malloc()` provided by `libgc`, while `libc`'s `malloc()` is used otherwise. Throughout this description, I will use *allocfn* to denote an arbitrary allocation function and $\mathit{sizeof} : \mathbf{Type} \to \omega$ to denote the compile-time function computing the un-boxed size of the LLVM representation of a given type in bytes.

Given a record initialisation of the form

$$x \texttt{ <- } \tau \texttt{ \{}F_1\texttt{:}f_1\texttt{, } \cdots \texttt{, } F_n\texttt{:}f_n\texttt{,\}}$$

the compiler will generate the LLVM IR shown in figure 3.6. Similarly, given a variant initialisation of the form

$$x \texttt{ <- } \tau\texttt{::}D\texttt{(}d\texttt{)}$$

the compiler will generate the LLVM IR shown in figure 3.7. Destructuring in match cases is achieved using LLVM's `switch` construction. As μ-Mitten does not support nested matching, no back-tracking is required. When matching the value of a word, the `switch` construct is applied to the word directly. When matching variants, the `switch` construct is instead applied to the variant's discriminant. In this case, code to cast the projected body of the variant to the appropriate LLVM type must be generated at the head of each of the basic-blocks targeted by the `switch`. Finally, when destructuring records, an unconditional switch is used (LLVM optimises this to straight line code). Code is then generated in the `switch`'s branch target to project the appropriate fields.

```
%alloc = tail call i8* @allocfn(i32 sizeof(τ))
x = bitcast i8* %alloc to compile(τ)

%field_1_ptr = getelementptr compile_U(τ), compile(τ) x, i32 0, i32 0
store compile(τ.F_1) f_1, compile(τ.F_1)* %field_1_ptr
⋮
%field_n_ptr = getelementptr compile_U(τ), compile(τ) x, i32 0, i32 n
store compile(τ.F_n) f_n, compile(τ.F_n)* %field_n_ptr
```

Figure 3.6: Expansion of record initialisation into LLVM

```
%alloc = tail call i8* @allocfn(i32 sizeof(τ))
x = bitcast i8* %alloc to compile(τ)

%discriminant_ptr = getelementptr compile_U(τ), compile(τ) x, i32 0, i32 0
store i64 i, i64* %discriminant_ptr
%body_ptr = getelementptr compile_U(τ), compile(τ) x, i32 0, i32 1
%cast_body = ptrtoint compile(τ.D_i) d to i64
store i64 %cast_body, i64* %body_ptr
```

Figure 3.7: Expansion of variant initialisation into LLVM

As a concluding example, the LLVM generated from the IR shown in figure 2.1 is shown in figure 3.8. Again, temporary names and labels have been altered to improve readability. As illustrated in figure 3.1, once generated, output LLVM IR is passed onto `clang` which functions as both an LLVM IR compiler and a linker, thus completing the core pipeline.

3.2 Analysis Framework

Once the core pipeline was complete, I began working on the pipeline extensions implementing ASAP. As ASAP relies on not one but three data-flow properties, I chose to implement a reusable framework within which each of the analyses could be implemented. This had another important benefit. As the results from ASAP's analyses are often extremely complex, testing them directly was impractical. However, by implementing a simpler analysis within the framework, the bulk of the analysis code could be tested without needing to verify results from ASAP's analyses.

At the core of the analysis framework is the `DataFlow` trait, given in appendix C. Each type implementing this trait must specify the direction of analysis (either forwards, or backwards). The types to be used for the decorations, summaries and amalgamated call-contexts of the analysis must also be specified along with how to initialise them. Next, implementers are required to specify how data flows through entry points, expressions, match cases and return points. The methods providing this information are given a reference to the analysis engine from which they are called such that they can query information about the procedure under analysis and update contexts and summaries of other procedures. Additionally, each is provided with

```
define i64 @fib(i64) {
entry:
  %t1 = icmp eq i64 %0, 0
  %t1_cast = zext i1 %t1 to i64
  switch i64 %t1_cast, label %outer_then [
    i64 0, label %outer_else
  ]

outer_else:
  %t3 = icmp eq i64 %0, 1
  %t3_cast = zext i1 %t3 to i64
  switch i64 %t3_cast, label %inner_then [
    i64 0, label %inner_else
  ]

inner_else:
  %t6 = sub i64 %0, 1
  %t7 = call i64 @fib(i64 %t6)
  %t9 = sub i64 %0, 2
  %t10 = call i64 @fib(i64 %t9)
  %t11 = add i64 %t7, %t10
  ret i64 %t11

inner_then:
  ret i64 1

outer_then:
  ret i64 0
}
```

Figure 3.8: LLVM IR generated from the IR given in figure 2.1

the location of the program point being analysed to enable cross-referencing between analysis passes. Finally, the data-flow confluence operator must be specified. This operator describes how to treat data-flow decorations at merging program points. It is only necessary to define this operation for backwards analyses as the structure of the analysis IR ensures there are no merging program points on forwards paths through the control-flow graph. Using this trait, I developed a simple inter-procedural LVA (`src/middle/analysis/passes/lva.rs`), some sample results of which are given in figure 3.9. As with figure 2.1, I have made some subtle changes to the formatting to aid readability.

```
                                   // (context) true
.fib {                             // (summary) {arg_0}
  %n <- arg_0                      // {%n}
  %t0 <- 0                         // {%n, %t0}
  %t1 <- %n == %t0                 // {%n, %t1}
  match %t1 {
    0 => {                         // {%n}
      %t2 <- 1                     // {%n, %t2}
      %t3 <- %n == %t2             // {%n, %t3}
      match %t3 {
        0 => {                     // {%n}
          %t6 <- %n - %t2          // {%n, %t6}
          %t7 <- .0(%t6, )         // {%n, %t7}
          %t8 <- 2                 // {%n, %t8, %t7}
          %t9 <- %n - %t8          // {%t9, %t7}
          %t10 <- .0(%t9, )        // {%t10, %t7}
          %t11 <- %t7 + %t10       // {%t11}
          ret %11
        },
        %t5 => {                   // {%t2}
          ret %t2
        },
      }
    },
    %t4 => {                       // {%t0}
      ret %t0
    },
  }
}
```

Figure 3.9: An LVA of the function shown in figure 2.1

The framework built around this trait is split into three components: the *analysis runtime*, *the analysis engine*, and the *local analysis engine*. At the highest level, the analysis runtime is responsible for handling invocations of analysis passes and caching results. As the data-flow trait prevents implementers from holding state, the analysis runtime can cache results on a per-type basis. This cache is used to make the point at which analyses are run transparent. That is, whenever a piece of code requires the results of an analysis pass, it asks the analysis runtime as if it were for the first time. If the analysis results have not yet been generated, the runtime spins up an analysis engine to compute them, otherwise, the cached results are returned immediately. A concrete illustration is given in figure 3.10.

Below the analysis runtime sits the analysis engine, overseeing the inter-procedural component of analyses. When first instantiated, the analysis engine will initialise empty contexts and summaries for all defined procedures and build a queue of procedures to be analysed. So

```
let runtime = middle::analysis::runtime(/* ... */);
// LVA has not yet been computed so this call will trigger the analysis
let lva = runtime.pass::<Lva>();
// Now the LVA is cached, all future calls will return the cached result
// Thus, no analysis will be triggered by this call
let lva = runtime.pass::<Lva>();
```

Figure 3.10: An example interaction with the analysis runtime

long as this queue is non-empty, the analysis engine will instantiate a local analysis engine for the procedure at the front of the queue. It is this local engine that is responsible for the intra-procedural component of the analysis.

Throughout analysis, the summaries and contexts of procedures may be updated. If a procedure's summary changes, the decorations computed for each of its callers are invalidated and must be recomputed. Similarly, if a procedure's context changes, its own decorations are invalidated and must be recomputed. The methods for updating contexts and summaries lie within the analysis engine. This ensures it can enqueue procedures for re-analysis as and when these changes occur. To guarantee analysis reaches a fixed point, procedures are only enqueued for re-analysis when there is a genuine change due to an update, rather than whenever an update occurs. It is this approach to inter-procedural analysis that allows the compiler to fix-point data-flow information for mutually recursive procedures, something which Proust's implementation was unable to do.

In normal circumstances, data-flow analysis engines will only record data-flow information at the entry and exit of basic blocks in the control-flow graph. The reasoning behind this is that storing data-flow information on a per-instruction basis is too memory intensive, particularly considering that the per-instruction information can be efficiently recovered from the block-level information. However, in the case of ASAP's analyses, I quickly found that the overhead associated with re-computing the per-instruction information could not be justified in terms of the memory saving and thus all data-flow information is recorded.

As with the inter-procedural analysis, the local analysis engine maintains a queue of basic blocks to be analysed. So long as this queue is non-empty, the data-flow information for the block at the front of the queue is computed. This is done by one of two methods: one corresponding to a forwards analysis pass and one corresponding to a backwards pass. These methods are largely symmetric with the key differences being in their treatment of entry blocks, return points and the order in which blocks are enqueued. Note that the local analysis engine will never recompute the data-flow information for a basic-block more than once in a single analysis of a procedure, as the IR ensures there are no cycles in the control-flow graph. That is, it is not necessary to fix-point the analysis information for basic blocks, only procedure contexts and summaries.

Finally, an instance of the local analysis engine can provide references to its parent analysis engine and further to the overarching analysis runtime. This allows analysis passes to request the results from other passes and update the contexts and summaries of other procedures. The one caveat is that there can be no cycles in the dependence graph for analyses. The reason for this is the analysis runtime waits for a pass to be completed before caching its results. Any cycle in the dependence graph will, therefore, cause unbound instantiations of the analysis engine and result in a crash.

3.3 *Shape*, *Share* and *Access*

As described in chapter 2, ASAP's analyses are interdependent; figure 3.11 illustrates the nature of this interdependence as a dependency graph. This presents an obvious problem: the dependency graph is not acyclic, as required by my analysis framework. However, this issue is easily circumvented by realising that any cyclic dependency graph can be reduced to an acyclic one by replacing each cycle with a pseudo-node constructed by taking the product of the analyses in that cycle. This idea is also illustrated in figure 3.11. For ASAP's case, this means that rather than build three analysis passes, I planned to build two: *Access*, as before; and *ImpliedAccess*, the product of *Shape* and *Share*.

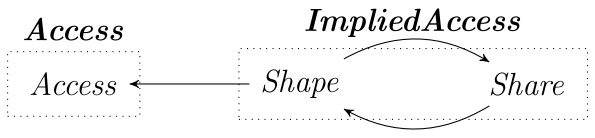

Figure 3.11: Dependency graph of ASAP's analyses

Proust observes that so long as values are used linearly, the *Share* property of a program will be void and therefore has no impact on its *Shape* property. This observation led me, at first, to a decision to implement the *Shape* component of *ImpliedAccess* before its *Share* component. However, ultimately I realised that an implementation of *Share* was simply not necessary to benchmark the algorithms I was interested in, as each of them treats its data linearly. Thus, in the final repository, only the *Shape* component of *ImpliedAccess* is implemented although it could easily be extended to further support *Share*.

Although the decorations for *Access* and *Shape* are defined mathematically as sets and relations of zones respectively, I built specialised data-structures to carry this information. For *Access*, I developed a data-structure `ZoneSet` (`src/middle/analysis/zones.rs`) representing a set of mutually incompatible zones rooted at a single local index. When a new zone is added to the set, I check for a compatible zone. If one is found, I merge the compatible zone with the new zone. Otherwise, the new zone's path is appended to a vector of paths underpinning the `ZoneSet`. The `AccessDecoration` data-structure defined in `src/middle/analysis/passes/access.rs` wraps this data-structure to track the accessed `ZoneSet` for each local index.

An efficient representation of *Shape*'s decorations was significantly harder to find. The data-structure I settled on maps pairs of local indices and paths to maps from local indices to paths. This allows for questions such as 'which zones alias with this zone?' and 'do these two zones alias?' to be answered efficiently; however, the implementation of the data-structure (`src/middle/analysis/passes/implied.rs`) is somewhat non-trivial.

The data-structures used to carry the summaries and contexts of *Shape* and *Access* were derived from those used to carry their decorations by making the data-structures generic in the index type they carry data for.

Having defined the data-structures used to carry the decorations, contexts and summaries of *Shape* and *Access*, building the analyses within my framework was straightforward. It was simply a case of translating Proust's mathematical definitions into code. However, all of this work depended on an implementation of paths. This was a major hurdle. In fact, efficient treatment of paths is something Proust's thesis highlights as an area for future investigation.

3.4 Implementing Paths

Paths are, as mentioned, regular expressions extended to support the subsumption operator $\preceq$. This gives the impression that an efficient treatment of paths would be to compile them into regular expressions and use an off-the-shelf regular expression library to implement operations over them. The problem with this approach becomes apparent when considering the role paths play in Proust's data-flow analyses.

```
fn len(xs: List) -> u64 {
  match xs {
    List::Nil(_) => 0,
    List::Cons(cell) => match cell {
      Cell { head: _, tail: xs } => 1 + len(xs),
    },
  }
}
```

Figure 3.12: Function to compute the length of a list

Consider the function `len` shown in figure 3.12, computing the length of a linked list. It is clear that given a list l, evaluating `len(`l`)` will involve traversing all of the cons cells constituting the memory representation of l (the *spine* of l). Thus, we should expect that the path representing the spine of the list

$$(\mathit{Cons} \cdot \mathit{tail})^* \cdot \mathit{Cons} \tag{3.1}$$

would appear in the *Access* summary computed for `len`. However, recall that the solutions to data-flow equations are computed iteratively. If we assume that the summary is initially empty, after a single iteration, the summary will contain *Cons*; after two iterations, the summary will contain $\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons}$; and, in general, after n iterations the summary will contain the path

$$\sum_{i=0}^{n} \left[(\mathit{Cons} \cdot \mathit{tail})^i \cdot \mathit{Cons}\right]$$

Although in the infinite case, this is equivalent to 3.1, for any finite n, this is not so. In essence, we have found an infinite ascending chain and therefore violated the ascending-chain condition described in chapter 2.

For the more theoretically inclined reader, it suffices to say that although for any $\tau, \tau' \in \mathbf{Type}$, $\mathbf{Path}_{\tau\to\tau'}$ ordered by $\preceq$ forms a meet-semilattice (with meets given by $+$), this semilattice is not of finite height. Thus, when using this semilattice to build partial solutions for our analyses, we cannot expect to converge on a least upper bound in a finite number of iterations.

Proust's solution to this problem is to constrain the set of paths he is interested in for each type to what he termed the *wild path set* for that type[3]. For each $\tau \in \mathbf{Type}$, he gave a construction $\mathit{Wild}(\tau)$ with the property that for any finite program, $\mathit{Wild}(\tau)$ is also finite, yet collectively its paths would explore all of the heap blocks reachable from a value of type τ. Given any $\tau, \tau' \in \mathbf{Type}$, by considering only $\mathbf{Path}_{\tau\to\tau'} \cap \mathit{Wild}(\tau)$ ordered by $\preceq$, Proust obtains a finite meet-semilattice and thus guarantees his analyses reach a fixed-point in a finite number of iterations. At the cost of some information that can be gained from straight line code[4], this

[3]This is an instantiation of a more general technique used in abstract interpretation known as *widening*.
[4]See p. 59 of Proust's thesis for a discussion.

is a neat solution to the problem in theory, but one that presents some significant challenges when put into practice.

The difficulty arises when attempting to compute the meet of a set of paths in these constrained semilattices. As wild paths sets are, in general, not closed under $+$, the $+$ operator no longer serves to compute the meet of two paths. Proust terms the operator computing the meet in one of his semilattices as the *widening* operator for that semilattice; however, his thesis proposes no algorithm for computing the application of such an operator. A naïve solution would be to compute $\mathit{Wild}(\tau)$ for every program type τ and discover meets by brute force (potentially with some form of memoisation to improve performance). This is an approach I put considerable effort into realising, but quickly discovered, is implausible for all but the smallest of programs[5]. In the end, I moved to develop an alternative method for constraining the domain of the analyses which I call *path compaction*.

3.4.1 Compact Paths

Presented here is the notion of *compact paths* and their applicability to data-flow analysis. Compact paths are my own extension to Proust's theory of paths. To the best of my knowledge, they are a novel concept and constitute a key contribution of this dissertation.

To gain a complete understanding of compact paths, it is important to understand their origin, which lies in automata theory. It is a well-known result from Kleene that the set of regular expressions (**RegExp**) and that of deterministic finite automata (**DFA**) are equivalent. That is to say, given any $r \in \mathbf{RegExp}$ there exists an automaton $\mathcal{M}(r) \in \mathbf{DFA}$ such that $\mathcal{L}(r) = \mathcal{L}(\mathcal{M}(r))$ (i.e., r and $\mathcal{M}(r)$ denote the same language). In general, there are many such automata and I will use $[\![\mathcal{M}(r)]\!]$ to refer to them collectively.

As paths are, at their core, regular expressions, Kleene's theorem allows us to further view them as DFAs. Figure 3.13 illustrates a DFA that is equivalent to the path $\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons}$ (the start state is indicated by an arrow and the accepting states are indicated by a double outline).

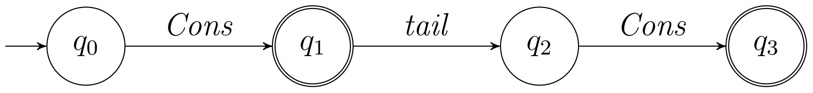

Figure 3.13: $\mathcal{M}(\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons})$

Although they are in essence regular expressions, paths contain a significant amount of information about the structure of the types in a program which is not captured by DFAs. Specifically, as each atom in a path $p \in \mathbf{Path}_{\tau \to \tau'}$ corresponds to the de-structuring of a value of one type to yield a value of another, we can usefully annotate each state in $\mathcal{M}(p)$ with the type of value that will be obtained when that state is reached.

Given a path $p \in \mathbf{Path}_{\tau \to \tau'}$ and its corresponding DFA $\mathcal{M}(p) = (\mathcal{Q}, s, \mathcal{F}, \Delta)$, we define the type-annotated DFA $\mathcal{M}^+(p) \in \mathbf{DFA}^+$ to be the 5-tuple $(\mathcal{Q}, s, \mathcal{F}, \Delta, \gamma)$ where $\gamma : \mathcal{Q} \to \mathbf{Type}$ is a type-annotation function satisfying

$$\gamma(s) = \tau$$
$$\forall q \in \mathcal{F}.\ \gamma(q) = \tau'$$
$$\forall q, q' \in \mathcal{Q}.\ (q \xrightarrow{\alpha} q' \implies \gamma(q).\alpha = \gamma(q'))$$

As such, $\mathcal{M}^+ : \mathbf{Path} \to \mathbf{DFA}^+$ represents an extension of Kleene's theorem between paths and type-annotated DFAs. The result of applying this new mapping to the path $\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons}$ is shown in figure 3.14 (annotations are written above the states).

[5]Wild path sets grow faster than exponentially in the size of the program type-graph.

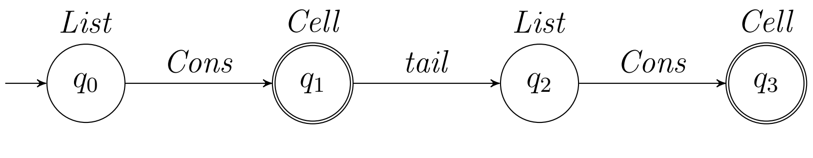

Figure 3.14: $\mathcal{M}^+(\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons})$

Using this definition of $\mathcal{M}^+$, we are now able to give a formal definition of compact paths. A path p is compact if and only if there exists an annotated DFA $m = (\mathcal{Q}, s, \mathcal{F}, \Delta, \gamma) \in [\![\mathcal{M}^+(p)]\!]$ such that γ is an injection. More intuitively, this means that each $\tau \in \mathbf{Type}$ has at most one representative state in m. I will write $\mathbf{Compact}_{\tau\to\tau'} \subseteq \mathbf{Path}_{\tau\to\tau'}$ for the set of compact paths between $\tau, \tau' \in \mathbf{Type}$.

Compact paths have several properties that make them amenable for use in data-flow analysis. Firstly, for any finite program, given program types τ, τ', we have that $\mathbf{Compact}_{\tau\to\tau'}$ is finite. This is because for any compact path $p \in \mathbf{Compact}_{\tau\to\tau'}$, its annotated DFA $\mathcal{M}^+(p)$ can be seen as a proper sub-graph of the program's type-graph of which there are finitely many (as the program type-graph is finite).

In fact, this correspondence between compact paths and sub-graphs of program type-graphs yields an efficient model of compact paths, as it is only necessary to track the subset of the type-graph's edges that appear in the path. Within this model, the *path-compaction* operator $\lfloor\cdot\rfloor_{\tau\to\tau'} : \mathbf{Path}_{\tau\to\tau'} \to \mathbf{Compact}_{\tau\to\tau'}$ can be defined as

$$\begin{aligned}
\lfloor \epsilon \rfloor_{\tau\to\tau'} &= \varnothing \\
\lfloor F \rfloor_{\tau\to\tau'} &= \{\tau \xrightarrow{F} \tau'\} \\
\lfloor D \rfloor_{\tau\to\tau'} &= \{\tau \xrightarrow{D} \tau'\} \\
\lfloor p \cdot q \rfloor_{\tau\to\tau'} &= \lfloor p \rfloor_{\tau\to\tau.p} \cup \lfloor q \rfloor_{\tau.p\to\tau'} \\
\lfloor p + q \rfloor_{\tau\to\tau'} &= \lfloor p \rfloor_{\tau\to\tau'} \cup \lfloor q \rfloor_{\tau\to\tau'} \\
\lfloor p^* \rfloor_{\tau\to\tau'} &= \lfloor p \rfloor_{\tau\to\tau'}
\end{aligned}$$

with the property that for any path $p \in \mathbf{Path}_{\tau\to\tau'}$, $p \preceq \lfloor p \rfloor_{\tau\to\tau'}$. As is shown, the behaviour of this operator in all cases is either trivial or a simple application of set union. This makes implementation simple. The annotated DFA corresponding to the result of applying the path compaction operator to our example path $\mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons}$ can be seen in figure 3.15.

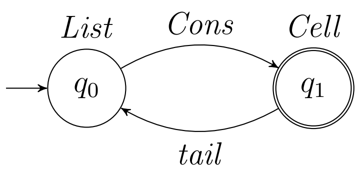

Figure 3.15: $\mathcal{M}^+(\lfloor \mathit{Cons} + \mathit{Cons} \cdot \mathit{tail} \cdot \mathit{Cons} \rfloor_{\mathit{List}\to\mathit{Cons}})$

Furthermore, as every compact path is still a path, given two compact paths $p, q \in \mathbf{Compact}_{\tau\to\tau'}$, we have that the path $p + q$ subsumes both p and q. As the compaction of this path $\lfloor p + q \rfloor_{\tau\to\tau'}$ subsumes $p + q$, by the transitivity of $\preceq$, we also have that $\lfloor p + q \rfloor_{\tau\to\tau'}$ subsumes both p and q. This gives an efficient way to compute meets of compact paths. Finally, recall that $\mathbf{Compact}_{\tau\to\tau'}$ is finite for any finite program, and thus $\mathbf{Compact}_{\tau\to\tau'}$ ordered by $\preceq$ forms a finite meet-semilattice within which we can efficiently build solutions to our data-flow equations.

3.4.2 Practical Implications

What is really happening here is that any time a path visits a type more than once, the compaction of that path jumps to the conclusion that that type could be visited an infinite number of times. This is a safe over-approximation of the original path; however, it has the obvious drawback that the precision of our analyses is reduced. This is because compaction throws away information that could have been gathered from straight-line code.

```rust
fn tail(xs: List) -> List {
  match xs {
    List::Nil(unit) => List::Nil(unit),
    List::Cons(cell) => match cell {
      Cell { head: _, tail: xs } => xs,
    },
  }
}
```

Figure 3.16: Function to compute the tail of a list

For example, consider the function `tail`, shown in figure 3.16, computing the tail of a list. Under Proust's wild path sets, the *Access* summary of `tail` would contain the path

$$Cons \cdot tail$$

However, after compaction, the summary would instead contain the path

$$(Cons \cdot tail)^*$$

Even though `tail` is non-recursive, compaction has discarded that information and assumed that every element in the list may be accessed.

To minimise complexity, in my implementation, all paths are compacted. As my benchmarks are recursive, I believe this will have had minimal impact. However, as a potential solution, I believe it would be possible to develop a hybrid between Proust's symbolic treatment of paths and path compaction to minimise the impact on precision. While fix-pointing analyses, paths would be compacted, but for straight-line code, paths would be treated symbolically. Hypothetically, this would preserve precision in straight-line code while maintaining the benefits of path compaction for fix-pointing but is left as future work.

3.4.3 Concrete Implementation

To capture the model of compact paths as sub-graphs of the program type-graph, I used the definition given in figure 3.17, taken from `src/middle/analysis/paths.rs`.

At the lowest level, the `Atom` type represents an edge label, as either a field projection or a variant projection. The `Edge` type then describes complete edges, with a source type (`src`), destination type (`dest`) and a label (`atom`). Finally, the `Path` type represents a compact path between its source type (`src`) and its destination type (`dest`). The set of edges making up the path is represented using the Rust standard library's hash-set implementation, which allows for path operations to be efficiently implemented using standard library functions.

As path operators are only defined given that some requirements on the source and destination types of the operand paths are met, it would have been desirable to have the compiler check that these requirements are met statically. However, as Rust is not dependently typed, this was simply not possible. Instead, as a correct implementation of ASAP's analyses should never violate the type restrictions on path operations, I made heavy use of debug assertions

```rust
#[derive(Copy, Clone, PartialEq, Eq, Hash)]
enum Atom {
  Field(ty::FieldIdx),
  Variant(ty::VariantIdx),
}

#[derive(Copy, Clone, PartialEq, Eq, Hash)]
struct Edge {
  src: ty::Ty,
  dest: ty::Ty,
  atom: Atom,
}

#[derive(Clone, PartialEq, Eq)]
pub struct Path {
  pub src: ty::Ty,
  pub dest: ty::Ty,
  edges: HashSet<Edge>,
}
```

Figure 3.17: The data-structure representing compact paths

to enforce restrictions at runtime. These assertions are disabled in release builds and therefore have no impact on the compiler's performance, but were invaluable while developing ASAP's analyses. An example of how debug assertions were used while implementing path union is given in figure 3.18.

```rust
impl Path {
  pub fn or(&self, other: &Path) -> Path {
    debug_assert!(self.compatible_with(other));
    Path {
      src: self.src,
      dest: self.dest,
      edges: self.union_edges(other),
    }
  }
}
```

Figure 3.18: Path union using debug assertions to enforce type restrictions

A drawback of this approach to implementing paths is that the Rust standard library's `HashSet` type cannot itself be collected in a hash-set, as it does not implement the required `Hash` trait. As the `Path` type contains a `HashSet`, it too cannot implement `Hash`. As such, when dealing with sets of paths, it is necessary to use a set implementation which does not make use of hashing. This also carries over to situations using maps with paths as keys: it is necessary to use a map implementation which does not hash its keys. Such implementations are not typically required or desired, so I was forced to develop these myself. My implementations of vector-backed sets and maps are given in `src/common/set.rs` and `src/common/map.rs` respectively. They support the majority of the public API of their hash-based alternatives in the standard library and therefore serve as drop-in replacements for working with paths.

3.5 SCAN and CLEAN

At this point, I began implementing CLEAN. The approach I took was broken down into two key components: implementing the transformation (code generation) and animating its output (runtime support).

3.5.1 Code Generation

The first step towards generating cleaning code was to extend the IR's instruction set with primitives for marking, un-marking and freeing pointers. I also included a 'reset' instruction which could be invoked after a cleaning pass to tidy up any state required during cleaning. This is illustrated in figure 3.19.

$$
\begin{aligned}
\langle \textit{instruction} \rangle ::= &\ \langle \textit{local-idx} \rangle \text{ `}\texttt{<-}\text{' } \langle \textit{expr} \rangle \\
| &\ \text{`}\texttt{free}\text{' } \langle \textit{type-idx} \rangle\ \langle \textit{local-idx} \rangle \\
| &\ \text{`}\texttt{mark}\text{' } \langle \textit{type-idx} \rangle\ \langle \textit{local-idx} \rangle \\
| &\ \text{`}\texttt{unmark}\text{' } \langle \textit{type-idx} \rangle\ \langle \textit{local-idx} \rangle \\
| &\ \text{`}\texttt{reset}\text{'}
\end{aligned}
$$

Figure 3.19: Extensions to the IR

With these IR extensions I began my implementation of the compile-time function SCAN. As mentioned, the implementation differs significantly from Proust's definition. The reason for this is path compaction. In Proust's thesis, SCAN is defined by structural recursion on the syntax of paths; however, the model of paths used by my implementation does not allow for this. Instead, when generating scanning code for a given path, my implementation generates a single scanning function for each reachable type in that path. For each outgoing edge from these types, code is generated to perform the corresponding projection and call the scanning function for the resulting type. The exception is the *u64* type which is silently ignored.

Scanning functions are generated to perform one of three roles: marking, un-marking and freeing. I refer to this as a function's *scanning mode*. When generating scanning functions to free a zone, a `free` instruction is inserted at the entry to the scanning function for the destination type of the zone's path. On the other hand, when marking and un-marking a zone, a `mark` or `unmark` instruction is inserted at the entry of each scanning function.

When CLEAN makes use of SCAN to scan a particular zone, a call to the scanning function generated for the zone's source type is inserted. In cases where the zone to be scanned is empty, no functions are generated and a single instruction corresponding to the scanning mode is inserted instead.

My implementation of CLEAN directly generates code at each program point implementing the series of operations described in chapter 2. However, naïvely generating fresh scanning code at each program point is highly likely to result in duplication of code. Thus, the data-structure overseeing CLEAN ensures that scanning functions are unique up to the zone they scan, the type they scan from and their scanning mode. However, this uniquing is purely nominal. That is to say, given code defining two distinct types with precisely the same structure, the scanning code generated for one cannot be used for the other as this would violate the type-safety of CLEAN.

3.5.2 Runtime Support

Although Proust's definition CLEAN makes use of two sets of marks, he suggests inlining mark-bits in the memory representations of heap objects to avoid incurring the cost of managing these sets explicitly. In this implementation, the marks applied to the matter set are inlined, while the anti-matter mark-set is managed by a runtime system.

In order to inline mark-bits and support the new IR primitives, it was necessary to extend the back-end. First, I altered type compilation under ASAP to add a mark bit to each record and sum-type. This was simply a case of adding a field with type `i1` to their LLVM representations. Given a type $\tau \in \textbf{Type}$, I will use $compile_M(\tau)$ to denote its LLVM representation with the additional mark-bit added, and $markptr_\tau(x)$ to denote a pointer to the mark-field of a variable x of type τ. As words are not freed, none of the primitives are defined when $\tau = u64$. Thus, I only consider cases where τ is a constructed type.

After extending the compilation of types, I implemented the primitives themselves. When compiled into LLVM, the `mark` and `unmark` primitives operate as expected, writing the mark field of their operand as shown in figures 3.20 and 3.21.

```
%mark_ptr = markptr_τ(x)
store i1 true, i1* %mark_ptr ; set mark-bit to 1
```

Figure 3.20: Expansion of `mark` into LLVM

```
%mark_ptr = markptr_τ(x)
store i1 false, i1* %mark_ptr ; set mark-bit to 0
```

Figure 3.21: Expansion of `unmark` into LLVM

On the other hand, the `free` and `reset` primitives expand to calls into a runtime system as shown in figures 3.22 and 3.23. Observe that the expansion of `free` will only invoke the runtime if the mark-bit of its operand is zero.

```
  %mark_ptr = markptr_τ(x)
  %mark = load i1, i1* %mark_ptr          ; load the mark
  br i1 %mark, label %merge, label %free ; if unmarked free, otherwise skip

free:
  %raw = bitcast compile_M(τ) x to i8* ; bit-cast x to a byte pointer
  call void @mitten_free(i8* %raw) ; invoke the runtime
  br label %merge                  ; continue

merge:
```

Figure 3.22: Expansion of `free` into LLVM

The runtime system was also developed as part of this project. When developing the runtime system, I worked to keep it as small as possible. The source code for the runtime is given in appendix D, it uses advanced features of Rust in order to safely interface with μ-Mitten executables as a dynamically-linked C library. As can be seen, the final implementation of `mitten_free` does not free memory directly, but adds the target pointer to a hash-set of pointers waiting to be freed. At each call to `mitten_reset()`, the hash-set is *drained* (cleared, while retaining the underlying memory allocation) and each pointer it contained is freed. By collecting pointers in a hash-set, we avoid double-frees which could otherwise occur when aliasing zones are scanned in a single cleaning pass. Furthermore, if memory is directly freed while scanning, the un-marking phase of a cleaning pass can dereference freed data, resulting in undefined behaviour. Thus all deallocations are pushed to the end of the cleaning pass in its call to `mitten_reset()`.

```
call void @mitten_reset() ; invoke the runtime
```

Figure 3.23: Expansion of `reset` into LLVM

3.6 Testing and Benchmarking Tools

As mentioned in chapter 2, I made use of regression testing throughout development. In order to minimise the overheads associated with adding new tests, I built a testing framework (`src/tools/mitten-test/`). Once pointed at a directory, it recursively discovers all `.mmtn` files and compiles and executes each of them. In order to make assertions about code it is running, the framework interprets the first comment of each `.mmtn` file it finds as a JSON object representing a test configuration. This configuration might assert that the test should compile and run to produce an expected value as shown in figure 3.24, or that compilation fails due to an expected error as shown in figure 3.25.

```
// { "compile_status": "ok", "expectation": 42 }

fn add(a: u64, b: u64) -> u64 {
  a + b
}

fn main() -> u64 {
  add(20, 22)
}
```

Figure 3.24: An example of a passing test

```
// { "compile_status": "fail" }

struct Foo {
  bar: u64,
}

fn main() -> u64 {
  let foo = Foo { bar: 0, bar: 1 };
  0
}
```

Figure 3.25: An example of a test with an expected failure

When beginning data-collection, I took a similar approach to implementing a benchmarking tool (`src/tools/mitten-bench/`). The most important requirement of the benchmarking tool was that it should be able to execute the same benchmark at a variety of problem sizes. However, as μ-Mitten code does not accept arguments from the command line, there appeared to be no way to achieve this without duplicating test code. This was not a satisfactory solution for the number of problem sizes I hoped to test. Instead, I extended the compiler to support a macro, `env`, which expands to the value of a given environment variable at compile time. An example of how this is used is given in figure 3.26. Benchmarks make use of this macro to pick up on changes to one of a number of environment variables set by the benchmarking tool. The benchmarking tool recompiles code as it varies these environment variables enabling measurement of the impact of executing the same code at a variety of problem sizes.

```
// { "compile_status": "ok", "expectation": 100, "env": { "MITTEN_ENV": "100" } }

fn main() -> u64 {
  env!("MITTEN_ENV")
}
```

Figure 3.26: A test invoking the `env` macro

3.7 Summary

To summarise, this chapter has described the implementation of the two key components of the compiler: its core pipeline, implementing compilation to LLVM; and the pipeline extensions, implementing ASAP's analyses and transformations. It also introduced path compaction, my own extension to Proust's theory, guaranteeing ASAP's analyses terminate efficiently. Finally, it discussed the automation of testing and data-collection, the results from which are discussed in the next chapter.

4 — Evaluation

This chapter evaluates ASAP in three aspects: its impact on execution, discussed in section 4.1; its compile-time overheads, described in section 4.2; and finally, its usability, analysed in section 4.3. Throughout sections 4.1 and 4.2, ASAP is compared to the Boehm-Demers-Weiser garbage collector. As a control, I also present data for when no memory-management strategy is used (i.e., situations when all memory is leaked). Section 4.4 summarises my findings.

The data presented in this chapter was collected over a number of days using a hosted virtual machine. The majority of the data collection was repeated 100 times to allow for estimates of error to be computed; however, the data regarding heap and cache performance could only be collected once due to high instrumentation overheads.

4.1 Impact on Execution

This section investigates ASAP's impact on execution. This investigation is not limited to absolute execution time, but also studies factors such as memory footprint and cache performance. I discuss my findings in the context of predictions made by Proust about the possible performance characteristics of ASAP when compared to other memory-management strategies.

The data I collected is centred around three μ-Mitten benchmarks: `list_len.mmtn`, which builds a list of length n and computes its length; `depth_first.mmtn`, which builds a binary tree with n internal nodes and traverses it depth first; and `quick_sort.mmtn`, which builds a list of length n containing the numbers 0 to $n-1$ in descending order, and sorts it using a quick sort into ascending order[1]. In each case, n is the problem size.

In real world programs, it is rare for a single algorithm to be run in isolation. Thus, in order to better capture how ASAP and Boehm-Demers-Weiser impact long-running processes with many isolated invocations of an algorithm, each benchmark iterates its algorithm i times, where i is determined by the environment variable `MITTEN_ITERS`. All of the data presented here was collected with i equal to 1000. The benchmark skeleton which performs this iteration is shown in figure 4.1.

```
fn kernel() -> u64 {
  // The actual algorithm would be invoked here...
}

fn iter(n: u64, acc: u64) -> u64 {
  if n == 0 {
    acc
  } else {
    kernel() & iter(n - 1, acc)
  }
}

fn main() -> u64 {
  iter(env!("MITTEN_ITERS"), 1)
}
```

Figure 4.1: The skeleton of each benchmark

It is important to note that although my implementation of ASAP produced correct cleaning code for each of my regression tests, `list_len.mmtn` and `depth_first.mmtn`, the more

[1]Note that this is always the $O(n^2)$ worst case for quick sort, but is of great interest due to the memory allocated for temporary lists during append operations.

complex `quick_sort.mmtn` contains a residual memory leak. I also planned to gather data from a fourth benchmark, `breadth_first.mmtn`, a breadth-first tree traversal; however, its generated cleaning code contains a double-free which I was unable to repair. The consequence of this is that various aspects of the data pertaining to `quick_sort.mmtn` are attributable to its memory leak and therefore do not coincide with those of the data collected for the other two benchmarks. I have highlighted situations where I believe this to be the case.

4.1.1 Execution Times

Figure 4.2 shows how execution time per iteration grows with problem size for each strategy on each benchmark. Dashed lines are used to indicate executions when LLVM's optimisations have been disabled, while solid lines indicate executions compiled with LLVM's highest optimisation level. The 95% confidence interval for plotted means was computed using a T-distribution; however, the resulting error bars are too small to be observed on the plot.

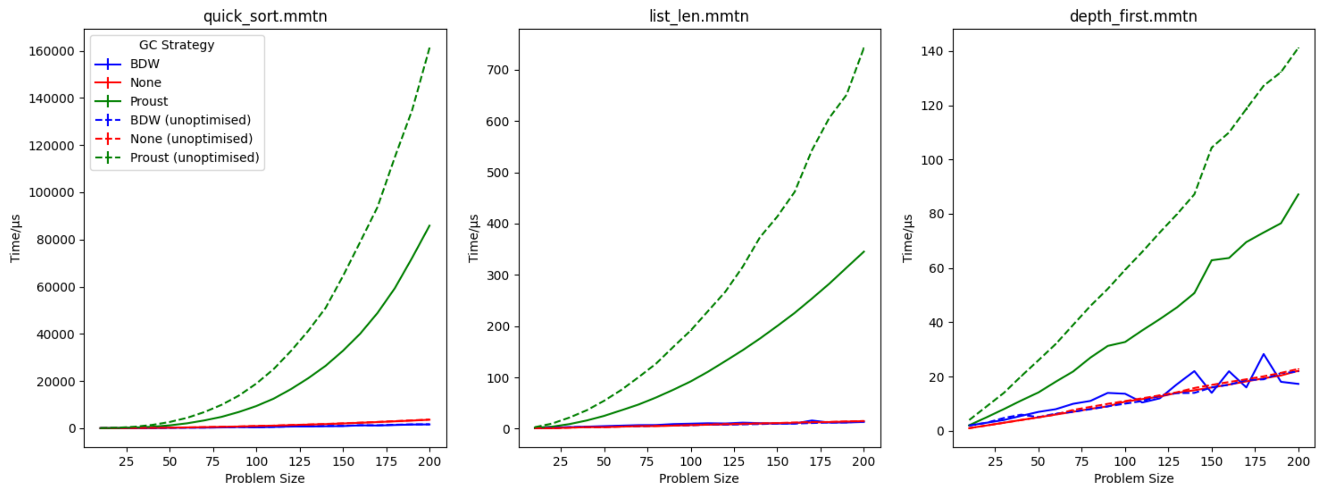

Figure 4.2: Time per iteration by problem size by strategy for each benchmark

It is clear from this data that application of my implementation of ASAP incurs a high cost when compared to both the control strategy and Boehm-Demers-Weiser. However, the data also highlights some other interesting facts about ASAP.

As mentioned, Proust observes that ASAP's position in the compiler pipeline leaves its generated code open to optimisation by later stages. The huge impact disabling optimisations has on ASAP's performance shown in figure 4.2 indicates that scanning code heavily benefits from standard optimisations. This is in contrast to Boehm-Demers-Weiser and the control strategy, which benefited minimally by comparison.

From this data-set alone, it is impossible to determine what portion of this is due to the relatively naïve approach I took to generating it, or ASAP's amenability to optimisation. I see it as likely that the impact of LLVM's optimisations on better-optimised scanning code is reduced; however, I believe that even with better-optimised scanning code, programs using ASAP would remain more amenable to optimisation than those using Boehm-Demers-Weiser.

4.1.2 Heap Usage

In embedded systems, available memory is often heavily constrained and therefore for ASAP to have applicability to systems programming languages as discussed in chapter 1, it is important that its generated code does not introduce space overheads.

To obtain heap performance data, I made use of a tool know as Massif, which belongs to the wider memory instrumentation framework Valgrind[8]. Figure 4.3 gives the maximum heap size in bytes observed by Massif at a range of problem sizes broken down by strategy for each benchmark.

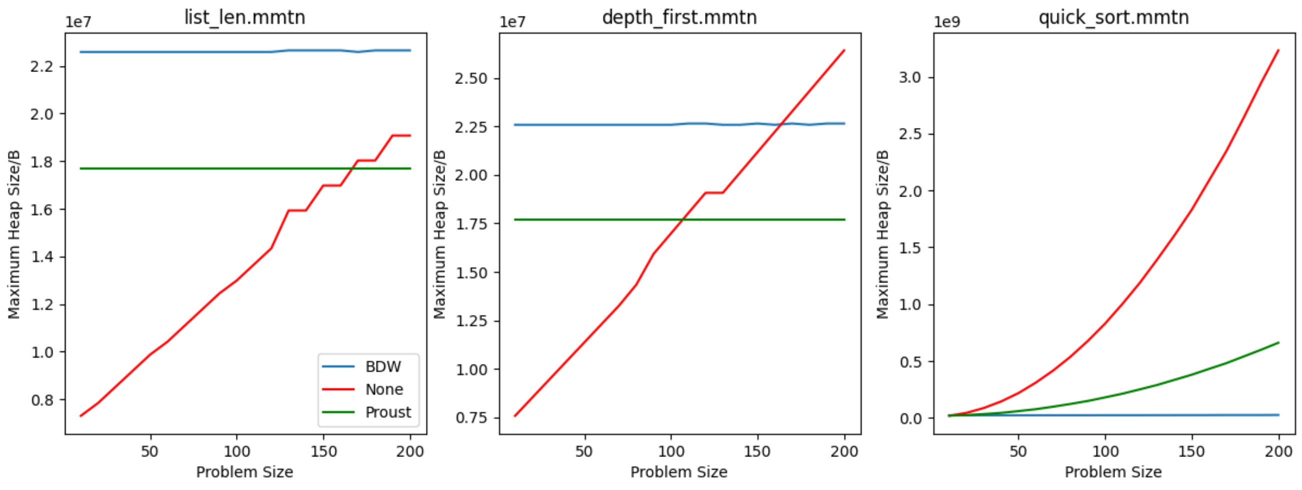

Figure 4.3: Max heap size by problem size by strategy for each benchmark

In the cases of `list_len.mmtn` and `depth_first.mmtn`, ASAP is able to outperform Boehm-Demers-Weiser at all problem sizes, while outperforming the control strategy at larger problem sizes. As my ASAP implementation makes use of the Rust runtime and Boehm-Demers-Weiser manages its own internal heap, both pre-emptively allocate pools of memory using `mmap()`. In these examples, this pre-allocated memory was never exceeded and hence there was no observed growth in the heap. As for the case of `quick_sort.mmtn`, I believe that the majority of the upwards trend in maximum heap size under ASAP is due to its residual memory leak, rather than ASAP itself. It is regrettable that the other benchmarks do not exhibit the effect of a single iteration exceeding the size of the pre-allocated memory pool; however, as previously mentioned, collecting this data was too computationally expensive for me to trial larger problem sizes.

4.1.3 Cache Miss Rates

Another matter of great interest is ASAP's impact on the cache performance of programs. Modern CPU architectures are aggressively optimised to exploit spatial and temporal locality. A feature of modern high-performance CPU architectures is a hierarchy of cache memories, designed to mask the huge latency associated with access to main memory. CPU caches are extremely high performance, but are heavily constrained in their capacity, meaning only a fraction of the memory associated with a process can be cached at any one time. When a process attempts to access a word which is not currently held in the cache, a *cache miss* occurs and the CPU is forced to wait for the word to be returned from main memory. This is hugely expensive and thus, there is great incentive to minimise cache miss rates.

Mark-and-sweep garbage collectors struggle here, as they regularly interrupt process execution to scan vast chunks of the heap. This will often result in many cache misses, as only a fragment of the heap will be cached at the point of interruption. Furthermore, active process data is likely to be evicted from cache causing further cache misses when execution is restarted.

Proust argues that ASAP's tendency to scan recently accessed data will result in a significantly lower data-cache miss rate when compared to other memory-management strategies. This is something I was keen to investigate.

To obtain cache performance data, I made use of another component of Valgrind: Cachegrind. Cachegrind emulates a two-level cache hierarchy with a combined L2 cache, but separate L1 instruction and data caches. From Cachegrind's output, I as able to compute both instruction and data-cache miss rates. Figure 4.4 illustrates the observed data-cache miss rate at a variety of problem sizes, broken down by strategy for each benchmark, and figure 4.5 illustrates the same for the instruction-cache miss rate.

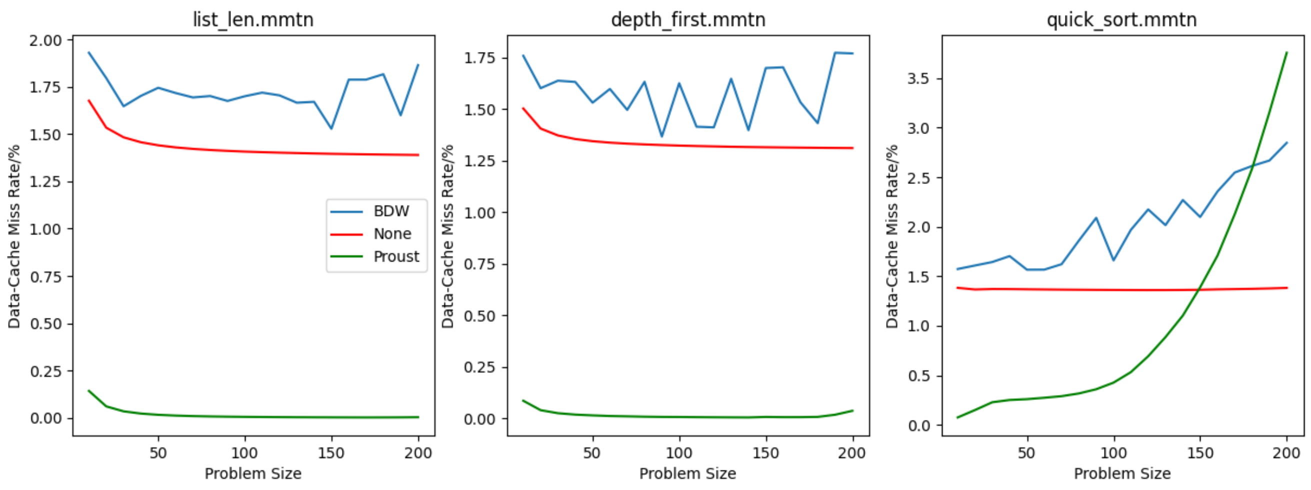

Figure 4.4: Data-cache miss rates by problem size by strategy for each benchmark

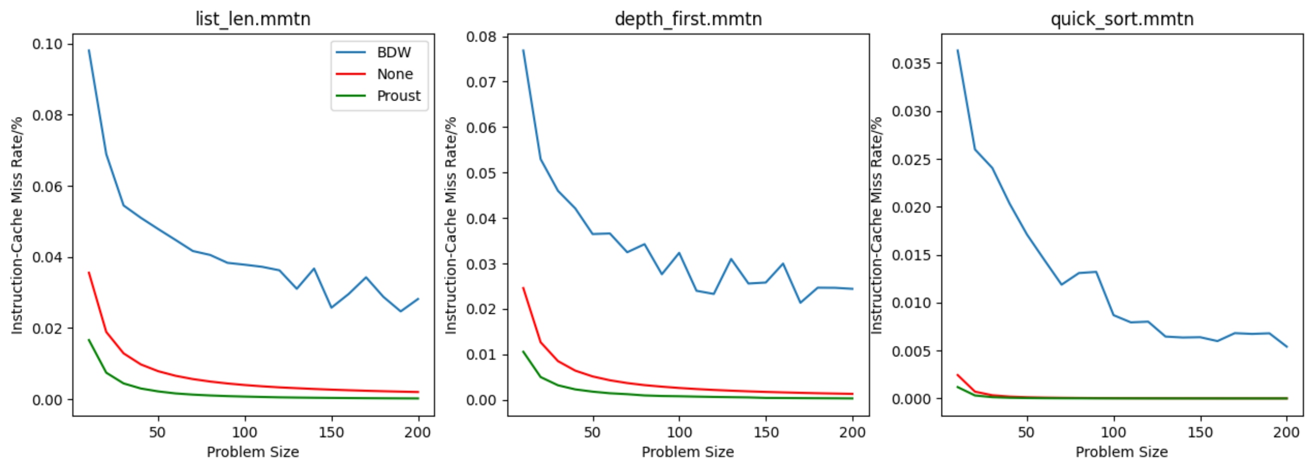

Figure 4.5: Instruction-cache miss rates by problem size by strategy for each benchmark

Given the disparity in execution times between ASAP and the other approaches, a valid criticism of figures 4.4 and 4.5 is that they may mask the absolute number of cache misses caused by ASAP behind a huge volume of memory traffic. In actual fact, this is not the case. Even the absolute number of cache misses caused by ASAP is very competitive. Figure 4.6 gives the absolute number of observed data-cache misses for the same data set, and figure 4.7 gives that of instruction-cache misses.

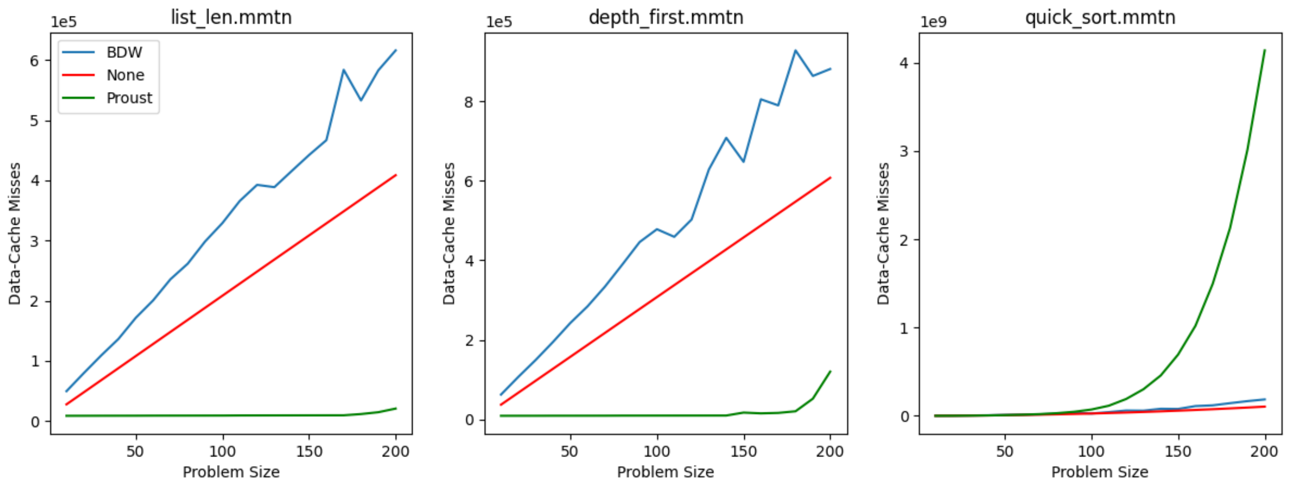

Figure 4.6: Data-cache misses by problem size by strategy for each benchmark

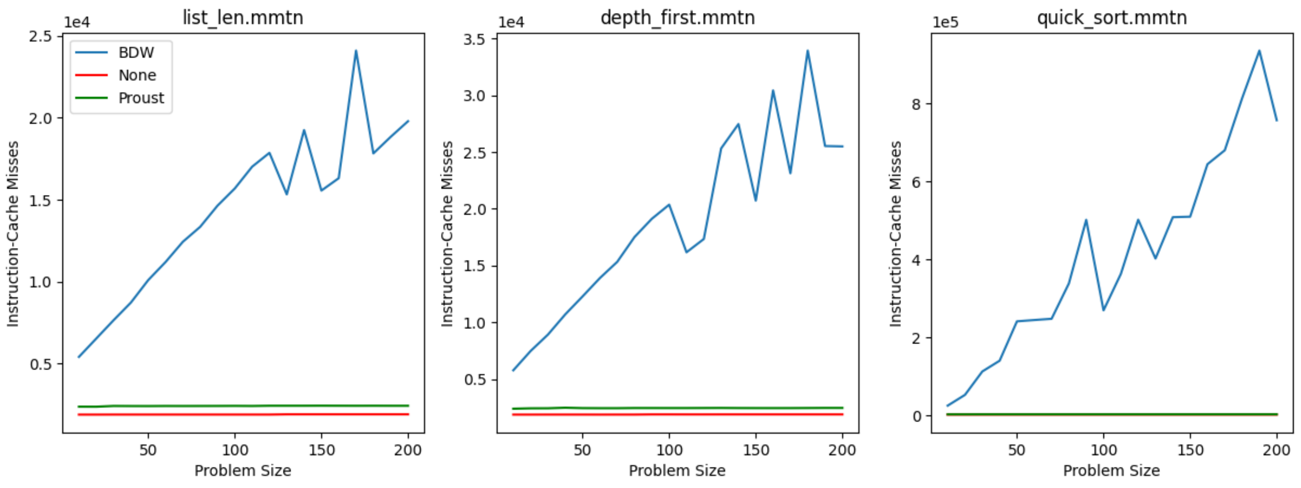

Figure 4.7: Instruction-cache misses by problem size by strategy for each benchmark

As can be seen, with the exception of the quick-sort benchmark (likely due to its residual memory leak), ASAP generates significantly less data-cache misses than Boehm-Demers-Weiser and the control strategy. Furthermore, it has a comparable number of instruction-cache misses with the control strategy while significantly outperforming Boehm-Demers-Weiser.

I believe this data constitutes strong evidence to support of Proust's prediction that ASAP would reduce data-cache misses when compared with mark-and-sweep garbage collectors. As for the improvement in instruction-cache performance compared with Boehm-Demers-Weiser, I believe that this is attributable to the fact that ASAP's cleaning code is contained almost entirely within its binary. In these examples, the binaries are small enough that they can fit within the instruction-cache in their entirety. This means there are at most as many instruction-cache misses as words in the binary, hence the independence of instruction-cache misses on problem size observed in 4.5. In contrast, Boehm-Demers-Weiser will load and evict large quantities of library code throughout execution resulting in the observed increase in instruction-cache misses with problem size.

4.2 Compile-Time Overheads

Beyond ASAP's impact on execution, it is important to consider its impact on compile-time factors such as binary size and compilation time. As ASAP relies on several static-analysis passes and code generation, there is a risk that ASAP's impact on compilation is too great for it to be practical in any industrial programming language. In this section I investigate whether this is really the case.

4.2.1 Compilation Time

Figure 4.8 shows the mean compilation time for each benchmark broken down by strategy over a sample of 100 compilations. The error bars indicate the 95% confidence interval for the mean as computed from a T-distribution.

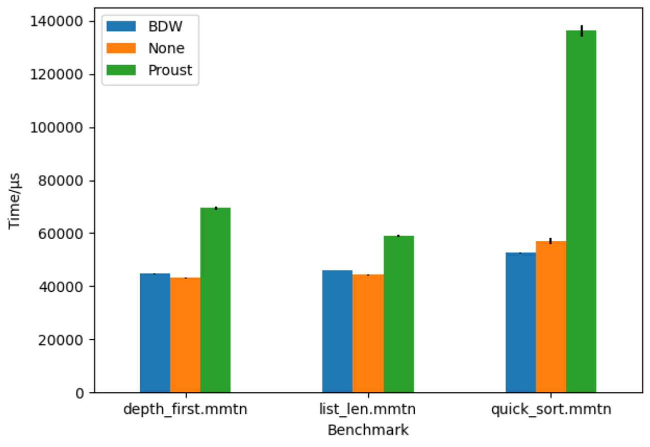

Figure 4.8: Compile time by strategy by benchmark

Although clearly noticeable, the impact of ASAP on compilation of these benchmarks was significantly less than I was anticipating given the amount of work done by its analyses. Quick sort, the benchmark with the most complex analysis results, suffered the most and this leads me to question how ASAP's impact scales when compared with other techniques. Proust's thesis sheds some light on the scalability of ASAP's analyses in isolation; however, does not provide similar data for untransformed code.

4.2.2 Binary Size

Figure 4.9 shows the output binary size for each benchmark broken down by strategy. As compilation is deterministic, there are no errors in these measurements.

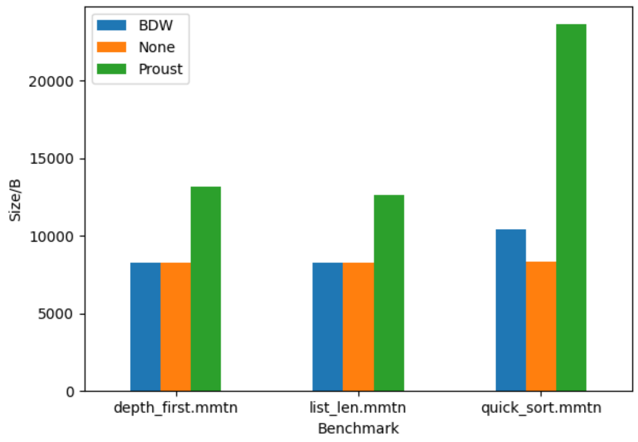

Figure 4.9: Output binary size by strategy by benchmark

Again, although noticeable, the growth in binary size due to ASAP is clearly within reason. From this data, it is impossible to determine how well ASAP's impact on binary size scales to larger programs; however, with more careful optimisation of scanning code such as structural uniquing and trimming, I believe further reductions in the impact are possible. For these benchmarks specifically, ASAP's impact on binary size is clearly offset by its huge reduction in instruction-cache misses when compared with Boehm-Demers-Weiser.

4.3 Usability

Finally, this section studies ASAP's usability when compared with Rust's ownership system. Specifically, I illustrate that ASAP is successful in eliminating the need for Rust's compile-time invariants and programmer annotations, while retaining the ability to manage memory statically. To this end, I present examples of μ-Mitten code taken from `quick_sort.mmtn` with adaptations to make them Rust compatible. These adaptations are highlighted in orange to draw attention to the simplifications ASAP was able to make to μ-Mitten's type system, without introducing the need for an automatic garbage collector.

The first key simplification, is to permit recursive types without explicit indirection. By default, Rust assumes that all records are un-boxed, and therefore in order to declare recursive types, programmers are required to explicitly mark fields as boxed. Figure 4.10 gives the μ-Mitten type declaration for a list altered in this way.

Secondly, when initialising these explicitly boxed fields, Rust requires the programmer to explicitly instantiate a box within which to move data. Figure 4.11 illustrates this in an adapted μ-Mitten function that builds a list of a given length.

Finally, when working with data structures containing boxed data, unless the data is to be consumed, Rust requires explicit *borrowing* (reference taking) of the contents of the box. The

```
struct Unit {}

struct Cell {
  head: u64,
  tail: Box<List>,
}

enum List {
  Nil(Unit),
  Cons(Cell),
}
```

Figure 4.10: Recursive type requiring explicit indirection

```
fn list_with_len(n: u64) -> List {
  match n {
    0 => List::Nil(Unit {}),
    _ => List::Cons(Cell {
      head: n,
      tail: Box::new(list_with_len(n - 1)),
    }),
  }
}
```

Figure 4.11: Explicit boxing when initialising boxed fields

noise introduced by this borrowing is highlighted in figure 4.12. The code written in grey is required to cast the Rust `bool` type to the μ-Mitten `u64` type and should be ignored.

Overall, μ-Mitten's use of ASAP enables it to separate the programmer from explicit notions of memory, while still managing allocations statically. In contrast, Rust gives explicit fine-grained control over the memory representation of data at the cost of introducing noise into the code as shown in figures 4.10, 4.11 and 4.12.

4.4 Summary

To summarise, my findings show a somewhat disappointing slowdown associated with application of ASAP. However, ASAP was also shown to have clear benefits in terms of reducing the memory footprint of processes and reducing cache miss rates over Boehm-Demers-Weiser. Furthermore, it is not susceptible to any kind of stop-the-world latency, while remaining fully automatic. The compile-time impact of ASAP was negligible for the benchmarks I considered, while it yielded a clear reduction in noise when instantiated in a Rust-like language when compared to Rust's ownership system.

```
fn sorted(list: &List) -> u64 {
  match list {
    List::Nil(_) => 1,
    List::Cons(cell) => match cell {
      Cell {
        head: head,
        tail: tail,
      } => match &**tail {
        List::Nil(_) => 1,
        List::Cons(cell) => match cell {
          Cell {
            head: other_head,
            tail: _,
          } => (head <= &other_head) as u64 & sorted(&tail),
        },
      },
    },
  }
}
```

Figure 4.12: Explicit address-taking and dereferencing when working with boxed data

5 — Conclusion

The project was a success. It implemented Proust's ASAP memory-management strategy in a native-code compiler and enabled the first real-world evaluation of ASAP's performance. Technically, the success criteria included implementing Proust's three data-flow analyses, but the data-flow analysis *Share* was not needed for this evaluation. As such, the project met its success criteria. Moreover, I extended the project by comparing ASAP's usability with that of Rust's ownership system and by developing Proust's theory to reduce the complexity of his analyses.

In terms of performance, implementing Proust's ideas in μ-Mitten showed that ASAP does not match its initial promise. In particular, section 4.1.1 illustrates a super-linear slowdown with increase in problem size. From this, I would argue that, as-is, ASAP is not ready for real-world adoption. However, the avenues of investigation highlighted in section 5.2 might begin to close this gap.

ASAP clearly has potential. Section 4.2 demonstrates, somewhat surprisingly, negligible compile-time costs across the set of benchmarks tested. Furthermore, ASAP exhibited impressive reductions in space overheads and cache miss-rates when compared to the Boehm-Demers-Weiser conservative garbage collector. By giving the first machine-level implementation of ASAP, this project has opened the door for further investigation into possible optimisations and improvements. Thus, I believe that this project has had extremely positive research outcomes.

5.1 Lessons Learned

Projects based on recent research work will inevitably run into issues with the source material. This is particularly the case when dealing with work that has not been widely peer-reviewed, such as Proust's thesis. There were many points during development where I felt my understanding was at its limit, or that a particular theoretical challenge was insurmountable. Spending more time absorbing the source material to avoid this kind of disruption would have smoothed development significantly.

The other main challenge of the project was the sheer amount of time spent debugging memory corruptions. Although this is expected in a project of this nature, I did not anticipate quite how time-consuming this would be. I believe a good portion of this time could have been saved if I had invested time earlier in the project to ensure that my compiler could emit debugging symbols for its generated LLVM IR. However, by the end of the project it was too late to consider. Were I to attempt this project again, I would focus on my approach to debugging as a central issue.

5.2 Future Work

It would be interesting to see an instantiation of ASAP in a more fully-featured language, such as one supporting parametric polymorphism and mutability. Proust's thesis gives indications of how his analyses may be extended to support such language features, but implementing them was beyond the scope of this project.

In terms of reducing ASAP's costs, further investigation into the impact of better-optimised scanning code is necessary. For example, the hybrid between Proust's paths and compact paths suggested in chapter 3 may well lead to higher-performance straight-line code. Other possibilities include the aggressive inlining of scanning functions, further trimming of matter and anti-matter sets and implementing new optimisation passes driven by Proust's analyses such as removing unaccessed allocations entirely.

Bibliography

[1] Proust, R. (2017) *ASAP: As Static As Possible memory management.* PhD thesis. University of Cambridge. `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-908.pdf` (Accessed: October 2019).

[2] The LLVM Foundation (2019) *The LLVM Compiler Infrastructure.* Available at: `https://llvm.org/` (Accessed: October 2019).

[3] The Rust Programming Language (2019) *Rust Programming Language.* Available at: `https://rust-lang.org/` (Accessed: October 2019).

[4] Boehm, H. (1993) 'Space efficient garbage collection', ACM SIGPLAN Notices, Vol. 28, No. 6, pp. 197-206.

[5] Boehm, H. (2016) *A garbage collector for C and C++.* Available at: `https://www.hboehm.info/gc/` (Accessed: October 2019).

[6] Khedker, U., Sanyal, A., and Sathe, B. (2009) *Data Flow Analysis: Theory and Practice.* Florida, USA: CRC Press.

[7] Cousot, P., and Cousot, R. (1977) 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints', in Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. California, USA: Association for Computing Machinery, pp. 238-252.

[8] Valgrind Developers (2019) *Valgrind.* Available at: `https://valgrind.org/` (Accessed: January 2020).

A — μ-Mitten

```
⟨type⟩ ::= ‘u64’ | ⟨ident⟩

⟨op⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘&’ | ‘|’ | ‘^’ | ‘<<’ | ‘>>’ | ‘==’ | ‘!=’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’

⟨field⟩ ::= ⟨ident⟩ ‘:’ ⟨ident⟩ ‘,’

⟨arg⟩ ::= ⟨expr⟩ ‘,’

⟨expr⟩ ::= ⟨literal⟩
  |  ⟨ident⟩
  |  ‘!’ ⟨expr⟩
  |  ⟨expr⟩ ⟨op⟩ ⟨expr⟩
  |  ⟨ident⟩ ‘(’ ⟨arg⟩* ‘)’
  |  ⟨type⟩ ‘{’ ⟨field⟩* ‘}’
  |  ⟨type⟩ ‘::’ ⟨ident⟩ ‘(’ ⟨expr⟩ ‘)’

⟨pattern⟩ ::= ⟨literal⟩
  |  ⟨ident⟩
  |  ⟨type⟩ ‘{’ ⟨field⟩* ‘}’
  |  ⟨type⟩ ‘::’ ⟨ident⟩ ‘(’ ⟨ident⟩ ‘)’

⟨case⟩ ::= ⟨pattern⟩ ‘=>’ ⟨term⟩ ‘,’

⟨term⟩ ::= ‘let’ ⟨ident⟩ ‘=’ ⟨expr⟩ ‘;’ ⟨term⟩
  |  ‘let’ ⟨ident⟩ ‘:’ ⟨type⟩ ‘=’ ⟨expr⟩ ‘;’ ⟨term⟩
  |  ‘if’ ⟨expr⟩ ‘{’ ⟨term⟩ ‘}’ ‘else’ ‘{’ ⟨term⟩ ‘}’
  |  ‘match’ ⟨expr⟩ ‘{’ ⟨case⟩* ‘}’
  |  ⟨expr⟩

⟨binding⟩ ::= ⟨ident⟩ ‘:’ ⟨type⟩ ‘,’

⟨variant⟩ ::= ⟨ident⟩ ‘(’ ⟨type⟩ ‘)’ ‘,’

⟨decl⟩ ::= ‘fn’ ⟨ident⟩ ‘(’ ⟨binding⟩* ‘)’ ‘->’ ⟨type⟩ ‘{’ ⟨term⟩ ‘}’
  |  ‘struct’ ⟨ident⟩ ‘{’ ⟨binding⟩* ‘}’
  |  ‘enum’ ⟨ident⟩ ‘{’ ⟨variant⟩* ‘}’

⟨program⟩ ::= ⟨decl⟩*
```

B — The Analysis IR

```
⟨op⟩ ::= '+' | '-' | '*' | '/' | '&' | '|' | '^' | '<<' | '>>' | '==' | '!=' | '<' | '>' | '<=' | '>='

⟨arg⟩ ::= ⟨local-idx⟩ ','

⟨field⟩ ::= ⟨field-idx⟩ ':' ⟨local-idx⟩ ','

⟨expr⟩ ::= ⟨literal⟩
  |  ⟨local-idx⟩
  |  '!' ⟨local-idx⟩
  |  ⟨local-idx⟩ ⟨op⟩ ⟨local-idx⟩
  |  ⟨def-idx⟩ '(' ⟨arg⟩* ')'
  |  ⟨type-idx⟩ '{' ⟨field⟩* '}'
  |  ⟨type-idx⟩ '::' ⟨variant-idx⟩ '(' ⟨local-idx⟩ ')'

⟨pattern⟩ ::= ⟨literal⟩
  |  ⟨local-idx⟩
  |  ⟨type-idx⟩ '{' ⟨field⟩* '}'
  |  ⟨type-idx⟩ '::' ⟨variant-idx⟩ '(' ⟨local-idx⟩ ')'

⟨case⟩ ::= ⟨pattern⟩ '=>' '{' ⟨block⟩ '}' ','

⟨instruction⟩ ::= ⟨local-idx⟩ '<-' ⟨expr⟩

⟨terminator⟩ ::= 'ret' ⟨local-idx⟩
  |  'match' ⟨local-idx⟩ '{' ⟨case⟩* '}'

⟨block⟩ ::= ⟨instruction⟩* ⟨terminator⟩

⟨param⟩ ::= ⟨local-idx⟩ '<-' ⟨param-idx⟩

⟨entry⟩ ::= ⟨param⟩* ⟨block⟩

⟨def⟩ ::= ⟨def-idx⟩ '{' ⟨entry⟩ '}'

⟨program⟩ ::= ⟨def⟩*
```

C — The DataFlow Trait

```
pub trait DataFlow: Sized + 'static {
  const DIR: DataFlowDir;

  type Context: Eq + fmt::Debug;
  type Summary: Eq + fmt::Debug;
  type Decoration: Clone + fmt::Debug;

  fn empty_context() -> Self::Context;

  fn empty_summary() -> Self::Summary;

  fn empty_decoration() -> Self::Decoration;

  fn entry(
    engine: &LocalAnalysisEngine<'_, '_, Self>,
    loc: AnalysisLoc,
    dec: &mut Self::Decoration,
  );

  fn expr(
    engine: &LocalAnalysisEngine<'_, '_, Self>,
    loc: AnalysisLoc,
    dec: &mut Self::Decoration,
    binding: ir::LocalIdx,
    expr: &ir::ExprKind,
  );

  fn pattern(
    engine: &LocalAnalysisEngine<'_, '_, Self>,
    loc: AnalysisLoc,
    dec: &mut Self::Decoration,
    source: ir::LocalIdx,
    pattern: &ir::PatternKind,
  );

  fn ret(
    engine: &LocalAnalysisEngine<'_, '_, Self>,
    loc: AnalysisLoc,
    dec: &mut Self::Decoration,
    idx: ir::LocalIdx,
  );

  fn conflate(
    engine: &LocalAnalysisEngine<'_, '_, Self>,
    loc: AnalysisLoc,
    decorations: Vec<Self::Decoration>,
  ) -> Self::Decoration;
}
```

D — The Runtime System

```rust
#![crate_type = "cdylib"]

use libc::c_void;
use std::cell::RefCell;
use std::collections::HashSet;

thread_local! {
  static ANTI_MARKS: RefCell<HashSet<*mut c_void>> = RefCell::new(HashSet::new());
}

#[no_mangle]
pub extern "C" fn mitten_free(ptr: *mut c_void) {
  ANTI_MARKS.with(|marks| marks.borrow_mut().insert(ptr));
}

#[no_mangle]
pub unsafe extern "C" fn mitten_reset() {
  ANTI_MARKS.with(|marks| {
    for ptr in marks.borrow_mut().drain() {
      libc::free(ptr);
    }
  })
}
```

Computer Science Part II Project Proposal

Practical Static Memory Management

Introduction and Description of the Work

It is well known that memory management is a necessary component of many programs, as without it, dynamically allocated memory would be left unreclaimed and cause a resource deficit, eventually resulting in a crash.

In his thesis, PROUST describes a novel approach to memory management wherein the compiler is responsible for statically inferring which heap blocks are safe to be freed, and inserting instructions to free them [1]. This is as opposed to other approaches taken by major programming languages such as placing the burden on the programmer to manually free allocated memory (as in C/C++); requiring an automatic garbage collector to run as part of a runtime system (as in Java); or requiring a linear/region-based type system (as in Rust).

PROUST's approach hinges on an abstraction of a heap access he refers to as a *path*. Using this abstraction, he defines three whole-program data-flow analyses (*Shape*, *Share*, and *Access*), which collectively gather enough information to statically infer which heap blocks are candidates for deallocation, and which are required to remain live.

Data-flow analysis is a family of techniques for statically approximating information about the runtime behaviour of programs by collecting information available at each program point and propagating this to surrounding program points [2]. Whole-program data-flow analysis, specifically, considers whole programs (rather than isolated procedures) and propagates data-flow information across call points (i.e. inter-procedurally).

Based on the results of his analyses, PROUST describes when and how code should be generated to descend into the heap along candidate *path*s in order to deallocate unused heap blocks in a mark-and-sweep style operation.

Unfortunately, the real-world performance characteristics of PROUST's approach are as yet unknown, as his work only went so far as to verify the approach's correctness in simulation. Thus, this project aims to investigate the practical viability of the approach by building the technology into a real compiler in such a way that permits empirical evaluation of its performance relative to alternative approaches on a real machine and to carry out such evaluation.

Starting Point

Rust appears to be a sound choice for the implementation language as it is well suited to compiler implementation and I have prior familiarity with it from a number of contributions

I have made to the language's compiler. I have also had limited interaction with the LLVM compiler infrastructure as part of a small preliminary investigation into the viability of this project. Furthermore, prior to starting, I have studied PROUST's thesis to the extent that I understand what needs to be done to provide a complete implementation of his work, but have made no attempt to implement a data-flow analysis framework or any of his analyses.

Resources Required

The key libraries I intend to work with (the LLVM compiler infrastructure and the BOEHM-DEMERS-WEISER GC) are open-source and widely distributed.

As for the development platform of my project, I intend to use my own laptop (quad-core Intel Core i7 CPU 1.8GHz; 8GB RAM; 256GB SSD). All source code (including the Dissertation LaTeX source) will be kept under `git(1)` version control, and back-ups will be pushed to GitHub and an external hard drive on a daily basis. Should my laptop become incapacitated, any remaining work can be completed on the Computing Service's MCS, as I have verified it has all necessary software pre-installed or readily available.

Substance and Structure of the Project

In order to carry out a detailed empirical analysis of PROUST's approach, it will be necessary to implement a compiler that takes source programs written in a language similar to that used by PROUST and produces executable binaries which either manage memory as described by PROUST, or link into an automatic collector as part of the runtime system, based on a compiler flag.

The idea is that it will then be possible to directly compare the performance characteristics of the binaries produced with and without applying PROUST's approach, as this will be the only differing stage in the compilation pipeline. With intent to draw practically applicable conclusions from this analysis, a collection of standard algorithms and data structures will be used as benchmarks, but determining which and how many is left as part of the project.

As a starting point, I only intend to profile against the BOEHM-DEMERS-WEISER GC as it represents a state of the art implementation of a popular class of garbage collectors, namely, conservative garbage collectors [3]. However, there is no reason that further collectors could not be trialed as extensions to the project.

PROUST states that his approach is complete. That is, all eligible heap blocks will be freed before program termination. Thus, any failure to implement PROUST's approach correctly should result in detectable memory leakage. Hence, the use of the BOEHM-DEMERS-WEISER GC also enables the evaluation of the correctness of any implementation of PROUST's approach, as it doubles as a leak detector and will therefore catch any such incorrectness.

Finally, in order to realistically profile the performance of the approach on a real machine, it is necessary to generate efficient machine code targeting that machine. Particularly, as PROUST's approach inserts code before compiler optimisation, freeing code may be optimised in each location it is inserted. The performance impact of this is something that I am also very interested in investigating. It is important to note here that the nature of PROUST's approach means that it does not require any form of special support from the compiler back-end, and hence any existing compiler back-end may be chosen as a target for the implementation. Thus, I am proposing to make use of the LLVM compiler infrastructure as the back-end for the implementation, as it is open-source, well maintained and known to generate efficient machine code[4]. Furthermore, LLVM supports several levels of optimisation, each of which could be trialed in order to determine if compiler optimisations are indeed impactful in improving the performance of deallocations.

From this, it follows that the project should have the following main sections:

1. Selecting and implementing the data structures necessary to represent the IR over which the analyses will be carried out.

2. A small front-end targeting the IR, to simplify the process of testing. This will include a simple lexer and parser capable of preparing an AST; a simple type-checker and a collection of de-sugaring operations if determined to be necessary.

3. A data flow analysis framework for the IR. The framework should support decorating program points with information gathered from PROUST's 3VL analyses, and thus, implementations of data structures for 3VL sets and relations will also be required.

4. Implementing the three analyses defined in PROUST's thesis (*Shape*, *Share* and *Access*) using the constructed analysis framework. This will require a representation of the *path*s over which he defines his analyses, and implementations of the operations he defines over them.

5. Implementing a code generation pass that generates freeing code based on the results of the analyses, or inserts instructions required to drive the BOEHM-DEMERS-WEISER GC.

6. A back-end targeting the LLVM IR.

7. Writing a collection of representative programs to profile with and without the statically generated freeing code.

8. Profiling the executions of these programs against a constructed set of benchmarks.

9. Writing the Dissertation.

Success Criteria

The following should be achieved:

- Implement each of the three data flow analyses defined in PROUST's thesis
- Implement the transformation that inserts cleaning code based on the results of these analyses
- Output LLVM IR corresponding to the chosen internal IR
- Perform a direct performance comparison between code generated using PROUST's approach, and code generated to use the BOEHM-DEMERS-WEISER garbage collector

Possible Extensions

The IR over which PROUST defines his analyses is simple in nature and does not support a variety of higher-level features that are commonplace in modern high-level programming languages. Examples include parametric polymorphism, mutually recursive functions, and mutability. However, towards the end of his thesis, he describes modifications to the analyses which permit such features. Thus, potential extensions to the project include extending the implementation to cover these modifications.

Furthermore, additional garbage collectors and memory management strategies could be compared in order to extend the empirical evaluation. For example, the OCaml garbage collector differs greatly in design from the BOEHM-DEMERS-WEISER collector, and in so doing, avoids some of the pitfalls of conservative garbage collection [5]. Thus, making it an excellent candidate to compare against.

References

[1] Raphaël L. Proust: ASAP: As Static As Possible memory management. Technical report, Computer Laboratory, University of Cambridge, 2017.

[2] U. Khedkar, A. Sanyal, and B. Sathe: Data Flow Analysis: Theory and Practice. CRC Press, 2009.

[3] A garbage collector for C and C++. `https://www.hboehm.info/gc/`, 2019.

[4] The LLVM Compiler Infrastructure. `https://www.llvm.org/`, 2019.

[5] Damien Doligez, Georges Gonthier: Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In Principles of Programming Languages, 1994

Timetable and Milestones

The planned starting date is 26th October 2019.

1. 26th October - 8th November *Front-end*

   Design and implement the in memory representation of the IR.

   Implement a lexer, parser, type-checker and simple lowering mechanism targeting this representation.

   Write a small test-suite to test the correctness of the constructed IR.

   Milestone: Completed front-end

2. 9th November - 22nd November *Back-end*

   Implement the LLVM IR generation pass.

   Set-up linker to produce executables linked to the BOEHM-DEMERS-WEISER GC.

   Enable the IR to be configured to interact with the GC in leak detection mode.

   Begin implementation of benchmarks.

   Extend the test-suite to test the correctness of the generated code.

   Milestone: Compiler now capable of producing garbage collected executables

3. 23rd November - 6th December *Analysis framework*

   Implement the data flow analysis framework and any supporting data structures (such as 3VL sets and relations).

   Write a sample data-flow analysis using the framework to verify the correctness and usability of the framework (i.e. a live-variable analysis).

   Continue implementation of benchmarks.

   Milestone: Can now perform a data-flow analysis on the IR

4. 7th December - 20th December *Shape, Share and Access*

   Implement each of Proust's analyses using the constructed framework and any supporting data structures (such as *path*s and *zone*s).

   Finish implementing benchmarks.

   Milestone: Completed implementation of all of PROUST's three analyses and benchmarking suite

5. 21st December - 3rd January SCAN *and* CLEAN

   Implement the compile-time function SCAN for scanning *zone*s.

   Implement the compile-time function CLEAN using the implementations of SCAN and *Access* in order to free candidate heap blocks.

   Further extend test-suite to ensure generated cleaning code is in fact cleaning all necessary heap blocks (issues with the analysis discovered as a result of this will also have to be fixed here).

Milestone: Completed implementation of PROUST's static memory management approach

6. 4th January - 17th January [*Slack*]

   Draft progress report and project presentation.

   If time allows, begin implementation of one or more extensions.

   Begin collection of data for evaluation.

   Milestone: Progress report drafted and delivered to Director of Studies and supervisor.

7. 18th January - 31st January *Complete progress report*

   If time allows, continue implementation of one or more extensions.

   Continue collection of data for evaluation.

   Milestone: Progress report submitted

8. 1st February - 14th February [*Slack*]

   If time allows, complete implementation of one or more extensions.

   Finish collection of data for evaluation.

   Milestone: Collected all data required for evaluation

9. 15th February - 28th February *Begin write-up*

   Complete a draft of the introduction and preparation sections.

   Milestone: Introduction and preparation sections drafted and delivered to supervisor and Director of Studies for feedback.

10. 29th February - 13th March *Continue write-up*

    Respond to feedback on the introduction and preparation sections.

    Complete a draft of the implementation section.

    Milestone: Implementation section drafted and delivered to supervisor and Director of Studies for feedback.

11. 14th March - 27th March *Continue write-up*

    Respond to feedback on the implementation section.

    Complete a draft of the evaluation section.

    Milestone: Evaluation section drafted and delivered to supervisor and Director of Studies for feedback.

12. 28th March - 10th April *Complete write-up*

    Respond to feedback on evaluation section.

    Milestone: Dissertation completed and delivered as PDF to supervisor and Director of Studies for feedback.

13. 11th April - 24th April [*Slack*]

    Respond to any final comments on Dissertation as a whole.

    Milestone: Dissertation submitted