Mukul Rathi

Types for Data-Race Freedom

Computer Science Tripos – Part II

Queens' College

8 May 2020

Declaration

I, Mukul Rathi of Queens' 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, Mukul Rathi of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed Mukul Rathi

Date 8 May 2020

Proforma

Candidate Number:	2417C
Project Title:	**Types for Data-Race Freedom**
Examination:	**Computer Science Tripos – Part II, 2020**
Word Count:	11787 [1]
Line Count:	14065 [2]
Project Originator:	The dissertation author and Prof. Alan Mycroft
Supervisor:	Prof. Alan Mycroft

Original Aims of the Project

The core aims were to implement a minimal concurrent object-oriented language *Bolt* (without methods) which implemented the core subset of the type system *Kappa*. *Kappa* eliminates data-races, a class of concurrency-related bugs which traditional systems programming languages like Java and C++ lack support to prevent. Bolt would be evaluated against a corpus of programs, a subset of which contained data races. Extensions included: implementing the full *Kappa* type system, thereby achieving more fine-grained concurrency than Rust. Implementing an inference algorithm to reduce programmer overhead of *Kappa*, an important consideration for practical languages. Implementing data-race-free data structures in Bolt.

Work Completed

Exceeded core and all extension success criteria. Bolt supports inheritance and polymorphism through generics, method overloading and overriding. Bolt compiles to native code and its performance exceeds Java. I extend *Kappa* to support inheritance and subtype polymorphism and incorporate ideas from Rust to increase expressivity. I transliterate Java programs that are hundreds of lines of code into Bolt with minimal overhead. I show how this extended version of *Kappa* could be incorporated into the Java type-checker, guaranteeing data-race-freedom statically with zero runtime overhead. This approach is backwards-compatible with existing Java code, aiding a gradual adoption of statically guaranteed data-race-freedom.

[1] Calculated using `texcount` (from `https://app.uio.no/ifi/texcount/`)

[2] Calculated using `cloc` (from `https://github.com/AlDanial/cloc`), ignoring autogenerated test output.

Special Difficulties

None.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Since the mid 2000's, single-core clock frequencies in modern processors have remained largely stagnant, with performance gains in hardware achieved through multi-core *parallelism*. Initially only *embarrassingly parallel* programs, which contain several long-running independent sub-units of work, were chosen as candidates to exploit parallelism. However this paradigm shift has since occurred across the hardware spectrum, from mobile devices to servers in large data centres. Embarrassingly parallel programs constitute only a tiny fraction of programs, thus to fully exploit these performance gains, programmers have had to specify parallelism in the presence of *shared state* between threads.

Thread scheduling and execution is inherently *non-deterministic*, resulting in *race conditions*, where the program's result depends on the interleaving of individual operations between threads. These introduce bugs in the presence of mutable shared state, one major class being *data races*. Data races occur when two threads concurrently access the same memory location, where at least one of the accesses is a write. Consider a bank application where users can view their account balance and deposit money into other accounts. The pseudocode below illustrates a data race when a user reads their bank balance whilst another user concurrently writes to it – if they read their balance when the write has only overwritten the lower bytes, they will read an invalid value that is neither the balance before nor after the deposit:

```
  user_1_thread{                    user_2_thread{
   read(user1.bankBalance)             user1.bankBalance += depositAmount
}                                   }
```

In general, data races are notoriously difficult to detect and reproduce due to the combinatorial explosion in the number of program execution paths: two threads executing a 150 line program concurrently have more interleavings than atoms in the universe. A given path may contain a data race but only be executed years after the code ships, and adding debugging instructions would alter its timing and instruction trace. Data races can result in unexpected behaviour that makes it impossible to reason about *correctness*: programs may no longer appear *sequentially consistent* (execution occurs in order specified by program) and values may appear out of thin air.

Despite the severity of bugs and difficulty in debugging them, programming languages designed in the 1980's and 1990's still have little support for ensuring correctness in concurrent programs. Programs with data races have *undefined behaviour* in the C17 and C++17 standards [13]. This means compilers do not provide *any* correctness guarantees when optimising source code if data races are present. Instead, the onus is on the *programmer* to prevent data races by correctly locking any concurrent accesses. This is only a partial solution: locking is itself error-prone. Programmers can forget to acquire or release locks and neither Java nor C++ provide any compile-time checks to prevent this. Moreover, manually tracking lock acquisition is a cognitive burden that does not scale to a large codebase.

The state space is so large we cannot *dynamically* guarantee the absence of data races through testing and test coverage. A more attractive approach is to prevent data races via *static analysis* of programs. Type systems such as Castegren and Wrigstad's *Kappa* [11] use static analysis to enforce data-race freedom. Proving the absence of data races is *undecidable*, so they conservatively reject some safe programs in order to ensure soundness. Rust is a prominent example of a new generation of programming languages whose rich type systems offer programmers stronger correctness guarantees. Rust's *ownership* type system [23] *eliminates* data races and guarantees memory safety.

However these type systems require different mental models (such as *ownership*) and specialised language constructs (such as *traits*) that are not present in the previous generation of programming languages. These older languages are still widely used for concurrent programming, with millions of lines of code that cannot easily be ported to newer languages. Programmers using these languages must instead rely on industry static analysis tools, such as RacerD [9] for Java and ThreadSanitizer [20] for C++, which sacrifice soundness to minimise false positives, so still miss data races. Therefore whilst programmers may have a high confidence data races aren't present, until there is language support for data-race freedom they do not have *certainty*.

1.1 Project Summary

In sections 2.3 to 2.4, I explain how Rust and *Kappa* guarantee data-race freedom; the latter type system forms the basis of my project. I demonstrate the following in my dissertation:

- I implemented *Bolt*, a concurrent object-oriented programming language modelled after Java. Bolt supports inheritance and polymorphism through *generics*, method *overloading* and method *overriding*. Bolt compiles to native code and implements concurrency through hardware threads. In section 4.4 I show that **Bolt's performance exceeds Java**.
- I extend Kappa to support inheritance and *sub-type polymorphism* (section 3.2.4). I designed an inference algorithm (section 3.2.3) to reduce *Kappa*'s annotation overhead, making it suitable to be used in a practical language.
- I provide empirical evidence for data-race freedom by evaluating Bolt's typechecker on a corpus of programs, demonstrating that Bolt supports **more fine-grained concurrency** than Rust (section 4.3).

- In section 4.2 I demonstrate that Java programs can be easily transliterated into Bolt with minimal overhead, and that the Bolt compiler scales to programs that are hundreds of lines of code.
- I provide a possible syntax through which *Kappa* could be incorporated into the Java type-checker with **zero runtime overhead** whilst being **backwards-compatible** with existing Java code.

Chapter 2

Preparation

Type systems are used in programming languages to *prove* the absence of undesirable program behaviour. Type systems consists of a set of rules (*typing judgements*) that constrain the behaviour of expressions based on the *types* they are tagged with – a `string` cannot be multiplied, nor can an `int` be concatenated. Richer type systems can prove stronger invariants about programs: that they will terminate, access memory safely, or (in our case) that they do not contain data races. In sections 2.2 to 2.4, I present three type systems that guarantee data-race freedom by constraining how references to objects are used, each building upon the the last: linear type systems, Rust's *ownership* type system [23] and *Kappa* [11]. I use *Kappa* as a basis for Bolt (section 3.2) and also incorporate ideas from Rust (section 3.2.2). Section 2.5 details the requirements for Bolt and how I used the spiral model of software development to achieve this.

2.1 Type Systems

Typing judgements in a type system are of the form $\Gamma \vdash e : \tau$. This can be read as: *expression e has the type τ under the typing environment Γ*. The typing environment Γ assigns a type t to each free variable x in e. Types *abstract* away the concrete values of the expressions and let us reason about their *properties*: here we are not concerned with the result of the computation, only *how* references to objects are used, ensuring that any object accesses are data-race-free.

Judgements in a type system can be composed using inference rules:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \ \ldots \ \Gamma_k \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}(\textit{rule-name})$$

This can be read as: *if $\Gamma_i \vdash e_i : \tau_i$ hold for all $i = 1 \ldots k$, then $\Gamma \vdash e : \tau$ holds.* These inference rules allow us to compose typing judgements about sub-expressions to reason about the overall expression.

2.2 Linear Type Systems

To simplify type-checking, the type systems presented in sections 2.2 to 2.4 impose an additional constraint: each reference can only be used in a **single thread** at once (I show this constraint is not necessary in section 3.2).

The simplest approach to preventing data races under this constraint is to prevent *aliasing*, where multiple references point to the same value. A *linear* type system only allows one reference to a value at any time, so objects can only be accessed from one thread at any time. Programmers transfer values between references via a **destructive read** (the original reference no longer points to the value). However, many operations (such as function calls) implicitly involve aliasing, requiring onerous destructive reads (denoted by the `consume` operation) and then reassigning the value to the original reference to continue using that reference as below:

```
function squared(int z){ // z gets a reference to argument passed in
  return (consume z , z * z);
}
let x = 1;
(x , result) = squared(consume x); // so we must destructively read x
// we return the result and reassign value to x
```

2.3 Rust's Ownership Type System

Rust's type system refers to linearity as *ownership*: references *own* values, and we *transfer ownership* (via a destructive read) between references. To reduce the programmer burden imposed by linearity, Rust allows *borrowing* – temporarily aliasing a value. When the owning reference is *borrowed*, we can have multiple referencing accessing the value concurrently. To prevent data races, Rust mandates that neither the owner nor the aliasing references can mutate the value whilst a reference is borrowed. Rust's borrow checker thus enforces the *Multiple Readers Single Writer* concurrency pattern.

Determining exactly when a reference is borrowed is intractable so Rust's borrow checker conservatively approximates this. The earlier implementations of the borrow checker marked a reference as *borrowed* if an alias was in the same lexical scope. This analysis is approximate: if the alias is in scope, then it *could* be used concurrently, so to guarantee data-race freedom, the checker does not allow either of the references to be written to. However, it is an *overapproximation* and rejects safe programs.

We can use *liveness analysis* as a better approximation. The value of a variable is *live* at a given point in the program if it will be used later in some execution flow of the program, and *dead* otherwise. We mark a reference as borrowed if it has a live alias. The safe assignment `x=2` below is rejected by the lexical scope analysis but accepted by liveness analysis:

```
let mut x = 1; // owning reference (denoted by mut)
{
  let y = &x; // y is an alias of x
  ...  // y is live & in scope
  println("{}", y); // y is dead (no longer used from now on)
  x = 2; // alias y is in scope but dead so safe to assign
}
x = 3; // this is fine since the alias y is not in scope
```

Key Takeaway. *Rust distinguishes between **two** modes of references: borrowed (aliased) and unaliased. Rust's borrow checker can accept more safe programs if we use a more nuanced approximation – shifting the burden from the programmer to the typechecker. I adopt liveness analysis in Bolt in section 3.2.2.*

2.4 Kappa – Capabilities for Access Control

Rust offers either mutability or aliasing, but not both. This is a conservative approximation: we should allow aliases to concurrently mutate an object so long as they access *disjoint* state (for example `x.f` and `x.g` access disjoint fields `f`, `g`). However in order to do this the type system needs to constrain which parts of the object a reference can access. *Kappa* assigns references *capabilities* which control how each reference can be used to access its object's internal state. Access modifiers (*public, private, protected, package*) in Java can be viewed as capabilities used to facilitate *encapsulation*. We augment our type system to track capabilities: we represent types in *Kappa* as *pairs* of the form (τ, κ) where τ corresponds to a traditional type (such as `int`) and κ corresponds to the capability held by the reference. Kappa's judgements (fig. 2.1) therefore extend traditional typing judgements with capabilities: $\Gamma \vdash e : \tau, \kappa$. Each capability grants a *mode* of operation, with its own constraints and corresponding typing judgements (denoted by *(e-_)*):

Linear: we can only have one reference to an object. This means we cannot directly access references *(e-var)*, but only destructively read (`consume`) them *(e-consume-var)*.

Read: We can freely alias the object, however aliases are read-only: we cannot assign to a field using a read capability *(e-assign-field)*.

Locked: the compiler inserts locking instructions around accesses (similar semantics to Java's `synchronised`), so all accesses are safe (we don't need additional judgements).

Thread-local: We can freely alias and modify the object, so long as all aliases are accessed in a single thread. Forked threads (listing 2) cannot access thread-local state *(e-finish-async)*.

Subordinate: we can freely alias and mutate the object, so long as we only access the object via an encapsulating object. An example of this would be a `LinkedListNode` object which is only accessed within the methods of a `LinkedList` object. To prevent subordinate state leaking out of an object, we only pass in or return subordinate state in internal methods (`this.m()`) *(e-subord-this)*, or if the object is encapsulated within another object *(e-subord-*

encaps). Finally, as subordinate state can be freely mutated and aliased within an object, we do not allow it to be concurrently accessed *(e-finish-async)*.

```
finish{
  async{ // fork a thread
    ...  // which executes expression in this block
  }
  async{ // fork another thread
    ... // which concurrently executes this expression
  }
  ... // concurrently execute this expression on current thread
}; // all forked threads join by the end of the finish block
... // continue executing on current thread.
```

Listing 2: We represent concurrency through the `finish-async` construct, which clearly defines the lifetimes of forked threads. The construct is discussed further in section 3.1.

Each of these modes guarantees data-race freedom. The *linear* and *read* modes are akin to those in Rust. *Thread-local* references are *exclusive* to a single thread so cannot be accessed concurrently. The *locked* mode guards all accesses with locks, so can be safely used concurrently – both *read* and *locked* capabilities are **thread-safe**. The *subordinate* mode inherits the data-race freedom from the encapsulating object. The *subordinate* mode is used to enforce the object-oriented principle of *encapsulation* in a concurrent context. Since object state is not exposed to users of the object, as long as the object's accesses are data-race free, any subordinate state is also protected.

Key Takeaway. *By introducing additional modes, each with different constraints, Kappa provides programmers with more flexibility, however this means our type system must track both types and* ***capabilities****.*

2.4.1 Traits and Capability Composition

As well as offering more modes, *Kappa* takes a more *fine-grained* approach to access control than Rust. Rather than associating capabilities with classes, *Kappa* associates capabilities with *traits* [19]: sub-units of classes that group together methods that implement a given *behaviour*. Classes in *Kappa* are composed from multiple traits. A reference with a given capability therefore can only access *part* of an object: the fields and methods associated with its capability's trait. Traits provide an alternative mechanism of code reuse in object-oriented languages: classes can share behaviour by reusing traits rather than via inheritance.

Type systems in traditional languages treat an expression as well-typed if its subexpressions are, regardless of whether the subexpressions are being executed sequentially or concurrently. This does not hold for data races: it is safe for a reference to use multiple capabilities sequentially, but using capabilities concurrently could introduce data races. For example, consider a thread-local capability and a read capability that access the same state. We have a data race if we use the

read capability to read from the state in another thread while using the thread-local capability to mutate it in the current thread.

Just as *Kappa* encoded partial access to an object through *traits*, *Kappa* encodes concurrent and sequential use in its type system through two binary operators to compose capabilities: the *conjunction* (can use the capabilities concurrently, denoted by $\otimes$) and *disjunction* (can only use the capabilities sequentially, denoted by $\oplus$) operators. Programmers specify how traits in a class will be used by composing the traits' capabilities using these operators. *Kappa* then checks these operators are valid (fig. 2.1): we can use any capabilities sequentially *(cap-disj)* but can only use capabilities that access disjoint state or that any overlapping state is thread-safe *(cap-conj)*. The example below illustrates three different traits that are composed to form a Binary Search Tree (`BST` class). `Left` and Right are used to concurrently mutate the left and right subtrees, whilst `Search` is used for a different behaviour: read-only traversal of the entire BST:

```
// Left and Right access disjoint subtrees so can be used concurrently
trait Left {
  // declare fields used
  require var leftChild:Tree;
  ... // setter and getter methods for leftChild
}
trait Right {
  require var rightChild:int;
  ... // setter and getter methods for rightChild
}
// Search accesses all fields so cannot
// be used concurrently with Left and Right.
trait Search {
  require var leftChild:Tree;
  require var value: int;
  require var rightChild:Tree;
  def search(int key) : int {
    ...
  }
  def inOrderTraversal() : void {
    ...
  }
}
class BST = (linear Left * linear Right) + read Search {
  var leftChild:Tree;
  var key: int;
  var value: int;
  var rightChild:Tree;
}
```

To determine whether overlapping state is *thread-safe*, we need to encode capability information about fields accessed into our overall capability. If a capability κ_1 accesses nested object fields that have capability κ_2, we encode this as the *composite* $\kappa_1\langle\kappa_2\rangle$ in the typing judgements. This composite also captures the requirements for accessing that field: we must type-check both κ_1's and κ_2's constraints. This is analogous to access control in a file system: to access a file we require permissions for both the directory (κ_1) and the file (κ_2). Below we illustrate a potential data race if we do not satisfy both κ_1's and κ_2's constraints. The example shows a *thread-local* mailbox server, that uses a *linear* message priority queue. If we alias the server with a replica, we implicitly alias the message queue, potentially causing a data race if we concurrently add messages to the queue:

```
trait PriorityQueue {
  require var messageQueue: BST; // linear
  ... // setter + getter methods
}
class MailBoxServer = local PriorityQueue { // thread-local
  var messageQueue : BST;
}
main(){
  let server = new MailBoxServer();
  server.setMessageQueue(new BST());
  let replica = server; // replica aliases server
  let serverMessages  = consume server.messageQueue;
  let replicaMessages = consume replica.messageQueue;
  // serverMessages aliases replicaMessages (violating linearity)
   ... // so could have data race if concurrently adding messages
}
```

Each mode in a composite capability introduces its own constraints so to type-check these we introduce the helper predicate `hasmode(capability, mode)`. This predicate uses two additional modes: *thread-safe* (can be safely accessed in multiple threads: that is, either *read* and *locked*), and *encapsulated* (no part of the object can be accessed from outside its encapsulating object).

We define the `hasmode(capability, mode)` on a mode-by-mode basis:

- **linear, subordinate, thread-local:** holds if any sub-capabilities have that mode, or if any of the fields in their traits have it, since if we violate the mode for the object, we violate it for the field as well.
- **read, thread-safe** holds if all sub-capabilities have that mode, since a capability is not read-only or thread-safe if sub-capabilities can be written to or are not thread-safe respectively.
- **locked** holds if all sub-capabilities are thread-safe, and at least one is locked, since this implies the use of a reader-writer lock.
- **encapsulated** holds if all of the modes in a capability are *subordinate*, that is no sub-capability can be accessed from outside the encapsulating object.

Key Takeaway. *Kappa uses traits and capability composition ($\kappa_1 \oplus \kappa_2$, $\kappa_1 \otimes \kappa_2$, $\kappa_1\langle\kappa_2\rangle$) to encode additional information in the type system about how capabilities are used. We use* `hasmode()` *to ensure a composite capability satisfies all constraints imposed by each of the sub-capabilities' modes.*

2.4.2 Using Capabilities Concurrently

Consider a reference to an object of type `BST`. This has the composite capability (`Left`⊗`Right`)⊕`Search`. Rather than tracking when the reference uses each of the capabilities and type-checking that the `Search` capability is indeed never used concurrently with the other two, *Kappa* makes a simplifying assumption. A reference holding a composite capability must be *decomposed* into separate references for each of its sub-capabilities. Since these decomposed references have only *one* capability they can use, it is straightforward to type-check their behaviour. *Kappa* only checks that the decomposition is sound by providing two language primitives: `unpack` and `jail`. The `unpack` primitive destructively reads the original reference, and returns two references, which each have a sub-capability and can be used concurrently. To prevent both sub-capabilities κ_1, κ_2 in a disjunction $\kappa_1 \oplus \kappa_2$ being used concurrently, *Kappa* requires that programmers specify which sub-capability will not be used by "jailing" one of them. (The `jail` operator prevents the capability from being used). Below we decompose the reference to access its subcapabilities `Left` and `Right` concurrently, jailing the `Search` capability in the process. We recover the original reference by composing subcapabilities with the `pack` operator:

```
class BST = (linear Left * linear Right) + read Search {
  ...
}
main(){
  let tree = new BST();
  let children, jail(search) = unpack tree;
  // children has (linear Left * linear Right) capability, search has none.
  let leftTree, rightTree = unpack children;
   // leftTree has Left capability, rightTree has Right capability.
   finish{
     async{
      leftTree.setLeftTreeVal(20);  // can access Left trait's methods
      }
      rightTree.setRightTreeVal(6); // likewise with Right trait's methods
   }
  let updatedSubTrees = pack(leftTree,rightTree);
   // updatedSubTrees has capability: (linear Left * linear Right)
  let recoveredTree = pack(updatedSubTrees,search);
  // recover original capability
}
```

Sub-capabilities κ_1, κ_2 in a disjunction $\kappa_1 \oplus \kappa_2$ have an additional constraint. If we extract κ_1, we cannot *forget* the mode constraints of the jailed sub-capability κ_2 when using κ_1. We cannot

use these sub-capabilities independently of one another like in a conjunction $\kappa_1 \otimes \kappa_2$, since they both access overlapping state that is not thread-safe. Here we extract a sub-capability from each of `queue` and `queueReplica`, but forget the constraints of the other jailed sub-capability. Had we enforced those constraints, we wouldn't have been able to access `viewQueue.head` in another thread, nor would we have been able to mutate `updateQueueReplica.head`:

```
class Queue = read View + local Update {
  var head:int; // can either be viewed or updated thread-locally
  ...
} ...
  let queue = new Queue();
  let queueReplica = queue ; // queueReplica is an alias of queue.
  let viewQueue, jail(updateQueue) = unpack queue ;
  // viewQueue, updateQueue now alias queueReplica
  finish{
    async{
      viewQueue.head // read using View, forget Update's constraint
    }
    let jail(viewQueueReplica), updateQueueReplica = unpack queueReplica ;
    // viewQueueReplica,updateQueueReplica alias viewQueue, updateQueue.
    updateQueueReplica.head := newHead; // use Update, forget View's constraint
  }
```

Key Takeaway. *To simplify type-checking, Kappa requires the programmer to decompose capabilities using the* `unpack` *and* `jail` *logical operations so each reference has precisely one sub-capability.*

2.4.3 Function Borrowing

As section 2.2 demonstrates, one source of programmer overhead with linear values is the need to destructively read and reinstate them when passing them to functions. *Kappa* instead allows a linear value to be *borrowed* for a function scope *(e-borrow-var)*. Within the function scope, the *borrowed* reference is linear as the original reference can't be accessed, and once the scope ends, the original reference can continue to be used. We can likewise return a *borrowed* value from a function (for example, if accessing a linear field). A common use-case is when we're passing the result of one function as an argument to another function (`f(g())`). *Kappa* prevents borrowed values from being assigned to ensure they do not outlive their scope. This contrasts with the use of borrowing in Rust, where *borrowing* occurs in the *same* scope, so aliases are made read-only.

```
function printInOrderTraversal(borrowed BST t){
  ... // t is linear as tree not visible in this scope
}
let tree = new BST();
printInOrderTraversal(tree); // we don't need to destructively read tree
// tree can still be used linearly since t bound to function scope.
```

$$\frac{\Gamma(x) = \tau, \kappa}{\Gamma \vdash \textbf{consume}\ x : \tau, \kappa}(\textit{e-consume-var})$$

$$\frac{\Gamma(x) = \tau, \kappa \quad \neg\ \texttt{hasmode}(\kappa, \textbf{linear})}{\Gamma \vdash x : \tau, \kappa}(\textit{e-var})$$

$$\frac{\Gamma(x) = \tau, \kappa_1\langle\kappa_2\rangle \qquad \neg\ \texttt{hasmode}(\kappa_1, \textbf{read})}{\Gamma \vdash x.f := e : \tau, \kappa}(\textit{e-assign-field})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau', \kappa' \quad \Gamma(x) = \tau_1, \kappa_1 \\ (\texttt{hasmode}(\kappa, \textbf{subordinate}) \vee \\ \texttt{hasmode}(\kappa', \textbf{subordinate})) \\ \implies \texttt{hasmode}(\kappa_1, \textbf{encapsulated})\end{array}}{\Gamma \vdash x.m(e) : \tau, \kappa}(\textit{e-subord-encaps})$$

$$\frac{}{\Gamma \vdash \mathit{this}.m(e) : \tau, \kappa}(\textit{e-subord-this})$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma_2 \quad \textbf{dom}(\Gamma_1) \cap \textbf{dom}(\Gamma_2) = \emptyset \\ \Gamma_1 \vdash e_1 : \tau_1, \kappa_1 \quad \Gamma_2 \vdash e_2 : \tau_2, \kappa_2 \\ \nexists x : \tau', \kappa' \in \Gamma_1.\ (\texttt{hasmode}(\kappa', \textbf{thread-local}) \vee \texttt{hasmode}(\kappa', \textbf{subordinate}))\end{array}}{\Gamma \vdash \textbf{finish}\{\textbf{async}\{e_1\}\ e_2\} : \tau, \kappa}(\textit{e-finish-async})$$

$$\frac{}{\vdash \kappa_1\langle\kappa_2\rangle \oplus \kappa_1'\langle\kappa_2'\rangle}(\textit{cap-disj})$$

$$\frac{\kappa_2 = \kappa_2' \implies \texttt{hasmode}(\kappa_2, \textbf{thread-safe})}{\vdash \kappa_1\langle\kappa_2\rangle \otimes \kappa_1'\langle\kappa_2'\rangle}(\textit{cap-conj})$$

$$\frac{\Gamma(x) = \tau, \kappa \quad \texttt{hasmode}(\kappa, \textbf{linear})}{\Gamma \vdash x : \tau, \textbf{borrowed}\langle\kappa\rangle}(\textit{e-borrow-var})$$

Figure 2.1: A subset of the typing judgements in *Kappa*.

2.5 Requirements Analysis

The requirements of the core project are as stated in the Success Criteria of the Project Proposal (Appendix C). Given the considerable complexity of *Kappa*'s type system compared to even Rust's type system, my core deliverable focused on implementing the **core subset of Kappa** (where classes contain a single capability). Java is a large programming language with many language constructs that are orthogonal to data-race type-checking, so following common practice in the field, I set out to implement a **minimal concurrent object-oriented language** that would highlight the type system. The final part of my core deliverable was a **Bolt bytecode interpreter** which nondeterministically interleaved instructions to simulate concurrency – I would use this to run a corpus of Bolt programs to evaluate the type system.

I set out to implement the remainder of the *Kappa* type system as a stretch requirement. I also performed a MosCoW analysis (table 2.1) of language features that I would incorporate if my project ran ahead of schedule. This analysis ranked language features by relevance (methods were more relevant than garbage collection) and complexity of implementation (control flow is easier to implement than parametric polymorphism). These language features were not critical to the success of the project, however broadened the scope of Bolt programs I could evaluate the type system on. Finally, I listed the following nice-to-have extensions:

Target LLVM IR: Generating native code and implementing concurrency with hardware threads would allow me to evaluate Bolt against Java in a more comparable way.

Extend Kappa: *Kappa* required traits, which were not compatible with Java. I wished to extend *Kappa* to support class-based inheritance and subtype polymorphism.

Capability Inference: This would reduce the annotation overhead of *Kappa*, making Bolt a more practical language to program in.

2.5.1 Spiral Model of Software Development

The project can be split into milestones, as language features and typing judgements can be added iteratively to the core deliverable. This lends itself to the *spiral model* [10] of software development. The milestones were structured to minimise risk: the first milestones achieved each of the core deliverable's requirements, and the subsequent extension milestones focused on implementing the "should-have" language features in the MoSCoW table and targeting native code. I then prioritised implementing the rest of *Kappa* and implementing the capability inference algorithm as these were a substantial deliverables that would greatly enrich the evaluation.

I evaluated the risk of new language features by compiling the equivalent C++ code to LLVM IR to understand the complexity of implementing them. I identified modifying *Kappa* to support inheritance and subtype polymorphism as high-risk milestones due to the additional reading it would require, so budgeted them as *reach* extensions that I would complete if I had time. I filed **126 subtasks** on GitHub, using a Kanban board [4] (a card-based project-management tool) to structure each milestone's development plan.

MoSCoW Category	Language Features
Must-have	Classes, structured fork-join parallelism, assignment to object fields, first-order functions
Should-have	Methods, control flow (if-else, while, for loops)
Could-have	Inheritance, parametric, subtype and ad-hoc polymorphism
Won't-have	Garbage collection

Table 2.1: MoSCoW [12] analysis of language features

2.5.2 Licensing

I have chosen to license my project under the MIT license [15] as I have made my code publicly available on GitHub. The MIT license is short, simple and permissive, and places limited restriction on reuse of the code: any person has permission to use, copy, modify and distribute copies of the software, subject to the condition that the license is included in all copies.

2.5.3 Version Control and Testing

I used Git version control (see fig. 2.3), and performed daily backups to Google Drive. I also backed the repository up to the MCS servers with a README containing instructions to install the project dependencies on a Linux machine or within a Docker container, should my main computer fail. I created 14 branches, one for each new milestone and merged back into master branch once fully tested.

I used Alcotest [1] for the OCaml code, writing more than **50 unit tests** for the lexer. I used Jane Street's **expect tests** library to test the remainder of the frontend (explained further in section 4.1). I chose this library as it reduced the *incidental complexity* of writing regression tests. Instead of manually writing the expected output, I ran the test case whose output I wished to capture for regression, and the test library would automatically update the expected output of the test with that output. This was especially beneficial when comparing against large expected abstract syntax tree (AST) outputs and meant I could write many more tests (I wrote **over 200 tests**) and thus cover more edge cases. I used Google Test [5] to write unit tests for the C++ backend. I ran a custom bash script comparing against the expected AST output of the compiler for the **integration tests**, and the expected LLVM IR output and program execution output for the **end-to-end tests**. In total, I wrote in excess of **300 tests**.

I set up a Continuous Integration workflow that ran regression tests and a linter on each commit. I tracked test coverage per commit using `Coveralls.io` [3] and achieved **94% test coverage** on the OCaml source code.

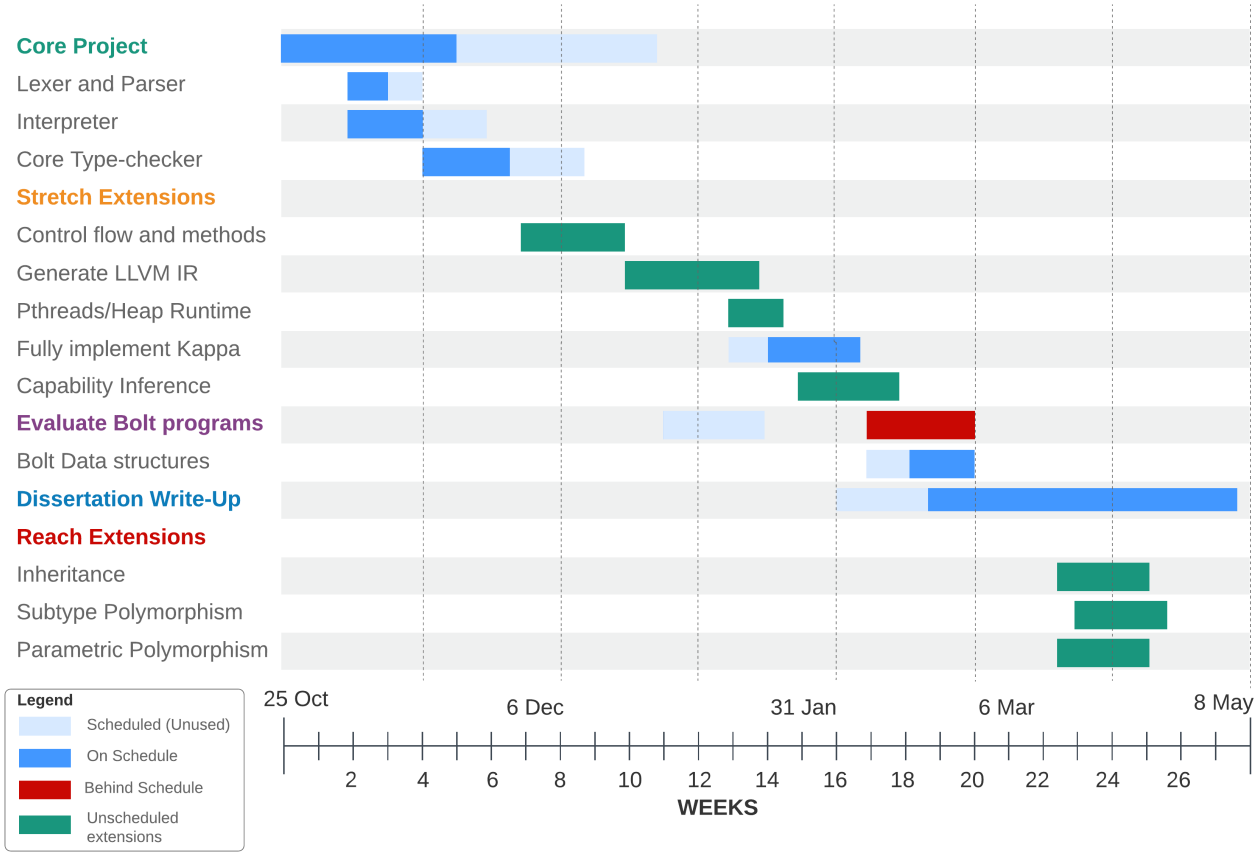


Figure 2.2: A Gantt chart detailing project progress. I chose to delay evaluation as the extensions led to a richer evaluation. I implemented the native code extensions in the slack time between the core project and scheduled extensions.

2.6 Starting Point

I had no previous experience with OCaml, C++ or compilers beyond the relevant courses[1] studied in the Tripos. The concurrency patterns discussed in *Part IB Concurrent and Distributed Systems* and *Part IB Concepts in Programming Languages* also proved useful when evaluating my language.

2.7 Summary

Rust's type system is centered around *ownership*: each value is owned by one reference. This provides two modes of operation: the owning reference can mutate its value when unaliased, and read-only access when *borrowed*. Rust uses *liveness analysis* to approximate when references are borrowed (a reference is borrowed when it has *live* aliases). *Kappa*'s type system is far more expressive and nuanced: objects can use *multiple* capabilities, each associated with an individual trait. However, this requires the programmer to *explicitly* compose and decompose the capabilities.

[1]Part IB Semantics of Programming Languages, Part IB Compiler Construction, Part II Types, Part II Optimising Compilers

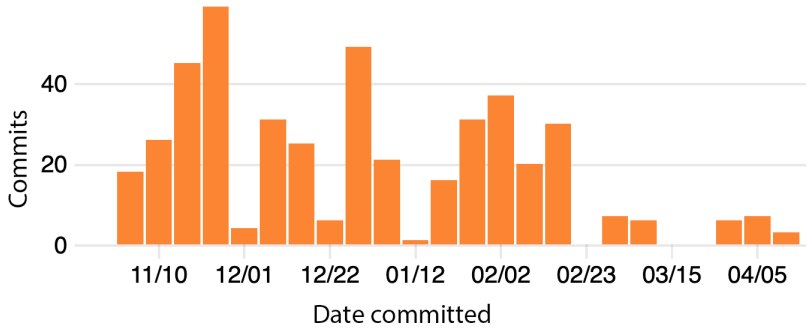


Figure 2.3: I pushed over 500 commits to the project repository.

Kappa's rich type system meant it was prudent to initially implement a minimal core language and core subset of the type system and then iteratively add language features and typing judgements using the spiral model of development. I extensively used version control and ran a large test suite on each commit to ensure each milestone did not introduce regressions.

Chapter 3

Implementation

The implementation of *Bolt* can be split into two halves. Firstly, the theoretical implementation: section 3.1 details the language design decisions made in Bolt and section 3.2 details how I extended *Kappa* to work in an inheritance-based language, with extensions to support capability inference and subtype polymorphism. The second of the chapter (section 3.3) details the practical implementation of the Bolt compiler, explaining how Bolt programs were compiled to native code.

3.1 Language Design

Bolt is a concurrent object-oriented language that models Java to demonstrate how the ideas presented in *Kappa* can be applied to inheritance-based languages. One of the biggest changes is to eschew traits in favour of class-based inheritance, since Java does not support traits. Bolt supports method overriding, a form of *subtype polymorphism*. Bolt also implements other forms of polymorphism present in Java: *generics* and method overloading. We show in section 3.3.3 that this does not require modifications to *Kappa*, demonstrating how *Kappa* can integrate with a rich programming language.

Bolt introduces a `consume` operation that destructively reads from a variable. However, Bolt doesn't contain the `pack`/`unpack`/`jail` operations; **programmers do not have to manage capability composition and decomposition**. Section 3.2.3 explains Bolt's capability inference algorithm, which makes this possible.

Unlike Java, Bolt uses *structured* fork-join parallelism via the `finish-async` construct, where threads can only be forked directly inside a `finish` block using the `async` keyword and the program waits for all forked threads to complete at the end of that `finish` block. This approach makes it easier for programmers to reason about the lifetime of a threads than in Java (listing 3), where threads' lifetimes depend on whether or not they are *daemon* threads, or if programmers have explicitly joined the threads.

```
function int concurrentSearch(){
  ...
  finish{
    async{ // thread spawned here
      ...
    }
    ... // executed concurrently here
  } // joins here
  ... // no longer executing here
}
```

(a) Bolt

```
int concurrentSearch(){
 ...
 Thread t = new Thread(searchTask);
 t.start() // thread spawned
 ...
 logExecution(t); // was t joined when
       // logExecution was executed?
 ...
 // does thread t outlive its function?
}
```

(b) Java

Listing 3: Bolt is more explicit about thread lifetimes than Java. The latter approach requires *global* knowledge of the codebase (such as the side-effects of `logExecution(t)` on `t`).

3.2 Modifications to Kappa

3.2.1 Capability Annotations Replacing Traits

Kappa used traits as a sub-unit of a class, to limit the fields of an object a capability could access. We can instead *directly* annotate the fields with the capabilities used (see listing 4). Rather than grouping methods by traits, methods too are annotated with the capabilities they can access. In *Kappa*, methods could only use one capability (the capability associated with the trait it was part of); now methods can be annotated with *multiple* capabilities. Bolt's annotations introduces a small annotation overhead, however making these annotations *declaration-site* (in class definitions rather than for each object) minimises the burden.

The programmer can also optionally annotate function/method parameters with the capabilities used. In the absence of annotations, both the type-checker and the programmer must assume *all* capabilities of the parameter are used, since we cannot locally inspect the actual function implementation. By annotating the parameter with the capabilities *actually* used, it allows more *fine-grained concurrency* as listing 5 demonstrates. These annotations allow programmers to *locally* reason about the capabilities used by a function/method call by inspecting its type signature, especially useful in a large codebase.

3.2.2 Type-checking Linear Capabilities

Kappa does not allow a reference with a *linear* capability to be directly accessed, only destructively read (fig. 2.1). As discussed in section 2.2, this conservative approach to preventing aliasing is onerous for the programmer. Bolt addresses this by allowing a reference to be temporarily aliased, as in Rust. We can use the *liveness analysis* algorithm used in the Rust borrow checker to determine whether any of a reference's aliases are *live* at a program point. If so, we drop its *linear* capability.

```
trait Left {
  require var BST leftChild;
  def setLeftVal(int val) : void{
    ...
  }
}
trait Right { ... }
trait Search {
  def search(int key) : int{
    ...
  }
}
class BST = (linear Left *
linear Right) + read Search {
  var leftChild:Tree;
  var key: int;
  var value: int;
  var rightChild:Tree;
}
```

(a) *Kappa*

```
class BST{
  capability linear Left, linear Right,
  read Search;
  var BST leftChild: Left, Search;
  var int key : Search;
  var int value: Search;
  var BST rightChild: Right, Search;
  int search(int key) : Search {
    ...
  }
  void setLeftVal(int val) : Left {
     ...
  }
 ...
}
```

(b) Bolt

Listing 4: Note that in Bolt we do not use the $\otimes, \oplus$ operators, instead *inferring* which capabilities can be used concurrently. We directly annotate fields and methods rather than grouping by trait.

```
function void printLeft(BST x){
  // we *could* be using x's Left,
  // Right & Search capabilities
  ...
   }
function void printRight(BST x){}
finish{
  // not allowed: functions could
  // use same capabilities
  async{
    printLeft(x)
  }
  printRight(x)
}
```

(a) No function parameter annotations

```
function void printLeft(BST{Left} x){
  // we know we *aren't* using x's
  // Right & Search capabilities
  ...
   }
function void printRight(BST{Right} x){}
finish{
  // fine as Left and Right can be
  // accessed concurrently
  async{
    printLeft(x)
  }
  printRight(x)
}
```

(b) Function parameters annotated

Listing 5: Capability annotations provide more information about how parameters are used.

We use *abstract interpretation* to track aliasing. Abstract interpretation is a static analysis technique where we track a program property whilst simulating its execution using the *operational semantics* of the language. Suppose we have an expression `let x = e`. If `e` could reduce to some identifier `y` then this expression would reduce to `let x = y` so we conclude that `x` potentially aliases `y`. Unlike *Kappa*, where functions/methods have only one argument, Bolt supports functions with multiple arguments, which can also implicitly introduce aliasing (consider `f(x,x)`). For programs with multiple execution paths (such as if-else statements), we end up with a set of possible aliases. We also want to track *transitive* aliasing: if `x` aliases `y` and later in the program `z` aliases `x`, `z` would alias `y`. To generate an upper bound on the set of *all* possible aliases, we iteratively run this algorithm starting from each of the immediate aliases `x` and add the aliases of `x` to our set, repeating until we have a *fixed-point* (no more aliases found). This is an upper bound as we don't execute the program, so in general we have no way of knowing if a branch is taken. We must *overapproximate* for soundness, so below we output that `newTree` potentially aliases `freshTree` and `existingTree`, even though the `else` branch will never be executed.

```
void maybeResetTree(){
  let shouldResetTree = true;
  let freshTree = new BST(); // "new BST()" does not reduce to an identifier
                             // so freshTree does not alias anything
  let existingTree = ... ;
  let newTree = (
    if (shouldResetTree) {
      ...
      freshTree // if branch reduces to freshTree
    } else {
      existingTree // else branch reduces to existingTree
    }
  ); // so expression could reduce to either freshTree or existingTree.
  ...
}
```

Finally, since *Kappa*'s typing judgements are *flow-insensitive* (don't take into account the order of execution), they do not prevent *dangling reference* bugs (where a programmer attempts to access a reference that has been destructively read). We can use *abstract interpretation* to track the set of destructively read references as the program executes, raising a compiler warning if later in the execution the programmer tries to access a destructively read reference that hasn't subsequently been reassigned.

3.2.3 Capability Inference

```
let tree = new BST();
let children, jail(search) = unpack tree;
let leftTree, rightTree = unpack children;
finish{
  async{
  leftTree.setLeftTreeVal(20);
  }
  rightTree.setRightTreeVal(6);
}
let updatedSubTrees = pack(leftTree,
                      rightTree);
tree = pack(updatedSubTrees,search);
```

(a) *Kappa*

```
let tree = new BST();
finish{
  async{
  tree.setLeftTreeVal(20);
  }
  tree.setRightTreeVal(6);
}
```

(b) Bolt

Listing 6: *Kappa* requires explicit capability decomposition – a significant overhead. Bolt's inference algorithm eliminates this.

Recall that *Kappa* (section 2.4) requires programmers to explicitly specify how capabilities are composed in a class and to use `unpack`/`jail` operations to decompose references into their sub-capabilities. The type-checker then *checks* whether the capability composition and `unpack`/`jail` operations are valid. However, this approach is purely to simplify type-checking. In Bolt, programmers need only declare the set of capabilities an object can use. The type-checker *infers* the capabilities being used by references and which capabilities can be used concurrently (Listing 6).

To infer capabilities used, we use *constraint-based* analysis. We begin by initialising each occurrence of a reference with the set of possible capabilities it can access. Rather than tracking which capabilities a reference can use, we instead *remove* capabilities that cannot be used. We check each of the modes according to the typing judgements in fig. 2.1, removing capabilities whose modes' typing judgements have been violated from that reference's set of capabilities. We are now left with the set of capabilities that are safe for the reference to use.

Next, we check that the reference only uses capabilities from this safe set, by checking the capabilities used when the reference accesses a field or call a methods. We also inspect the type signatures of functions/methods that take this reference as a parameter, checking they do not use other capabilities. In the case where an reference can potentially use multiple capabilities to access a field, we choose the capability whose mode is least restrictive. For example, we would favour *locked* over *linear*, as we can concurrently reuse the same *locked* capability but cannot use a *linear* capability in multiple threads.

Thus far we have inferred the sets of capabilities that are safe for references to use, and *individually* checked which capabilities each field access, method and function call uses. However we need to check that *concurrent* capability accesses are safe. We first determine which references are shared by multiple threads. For each of the shared references, we aggregate the capabilities used in each

thread. We then enforce *Kappa*'s typing judgements (fig. 2.1) by comparing capabilities across threads pairwise to check they can be used concurrently.

3.2.4 Inheritance and Subtype Polymorphism

In Java, classes can inherit from other classes to *reuse* existing functionality, for example if `Car` inherits from `Vehicle`, then a `Car` object can reuse the `Vehicle` methods. Java treats `Car` as a subtype of `Vehicle` (we can use `Car` anywhere we expect an `Vehicle`), which we might intuitively consider to be true: a car *is* an vehicle. However, Java allows us to add an `eat()` method to the `Car` class, despite this not being a behaviour associated with a `Vehicle`. To prevent this, we require a stronger notion of subtyping than what Java provides: *behavioural subtyping* [16]. A class C is a behavioural subtype of class D if we can replace objects of type D with objects of type C *without altering any desirable properties of the program.* Here, it is that all subtypes of `Vehicle` should behave like a vehicle; in the case of Bolt, the property we desire is data-race freedom. If we pass a `Car` to a function that expects an `Vehicle` we should not have data races.

This is complicated by method overriding, a form of *subtype polymorphism* where the subclass provides a different implementation of a method in its superclass, and the method implementation executed is *dynamically* determined by the object type (not reference). The capabilities actually used by a `Car` may differ from those statically assumed by the `Vehicle` reference (it uses `UpdateModel`, `UpdateMake` when expected to use just `UpdateModel`):

```
class Vehicle{
  capability linear UpdateModel, linear UpdateDriveTrain;
  var String model : UpdateModel;
  var String engine : UpdateDriveTrain;
  void setModel(String model) : UpdateModel {
    this.model := model
  }
}
class Car extends Vehicle {
  capability linear UpdateMake;
  var String carMake : UpdateMake;
  void setModel(String model) : UpdateModel, UpdateMake {
      ...
  }
}
```

Subtypes should *preserve* the external behaviour of the method: if an overridden method can be used safely in conjunction with another field access or method invocation in the superclass, then it should be able to in the subclass. Bolt guarantees this by only allowing subclasses to add or extend capabilities to cover any *additional* fields, not the fields present in the superclass. A function treating the `Car` as an `Vehicle` will not use the additional fields (`carMake` in the example above) so the behaviour will appear the same. However, below the `Truck` is not a subtype of `Vehicle`:

there are two changes in external behaviour. Firstly, `UpdateModel` and `UpdateDriveTrain` can no longer be safely used concurrently as the `wheels` field is accessed by both `UpdateModel` and `UpdateDriveTrain` – the capability no longer access disjoint state. Secondly, `setModel` now also uses `Vehicle`'s `UpdateDriveTrain` capability, so we can no longer access `setModel()` and the `engine` field concurrently:

```
class Truck extends Vehicle{
   // trucks can have 6 or 8 wheels depending on model and drivetrain
  var int wheels : UpdateModel, UpdateDriveTrain;
  void setModel(String model) : UpdateModel , UpdateDriveTrain {
    ...
  }
}
```

We must also consider *covariance* in function arguments. A function may call an `Vehicle`'s overridden method assuming its capability requirements have been satisfied. When we pass a `Car` as argument, the `Car`'s overridden method may actually also use additional capabilities present only in the `Car` (which the function is unaware of). To preserve data-race freedom, we conservatively assume that if a function *could* call an overridden method, then it *does*, and therefore assume it must use those additional capabilities. Therefore, we reject the program below:

```
function void updateVehicle(Vehicle{UpdateModel} v){
  // the function assumes v is of type Vehicle, and it has the
  // UpdateModel capability so it could call v.setModel()
  ... // according to Vehicle class definition
}
void main(){
  let ferrari = new Car();
  finish{
    async{
      ferrari.make := "F50" // this write uses UpdateMake
    }
    updateVehicle(ferrari) // if this calls ferrari.setModel() it would use
    // UpdateModel and UpdateMake - the latter could cause a data race
  }
}
```

3.3 Compiler Implementation

A typical compiler takes a program, type-checks it and compiles it to a simpler intermediate representation (IR). LLVM IR is a common target for industrial compilers as it can target many assembly backends and also provides in-built IR optimisations, making the generated native code more performant. Bolt's compiler frontend compiles programs into an IR designed close to LLVM IR. The compiler backend converts this Bolt IR into LLVM IR, and links in runtime libraries when compiling to native code. Bolt's type system consists of both traditional types (for example `int`) and capabilities, so we split type-checking into **two stages**: type-checking the traditional types, followed by the more complex capability type-checking. We desugar the typed abstract syntax tree (AST) between the stages, so the capability checking operates on a simpler representation.

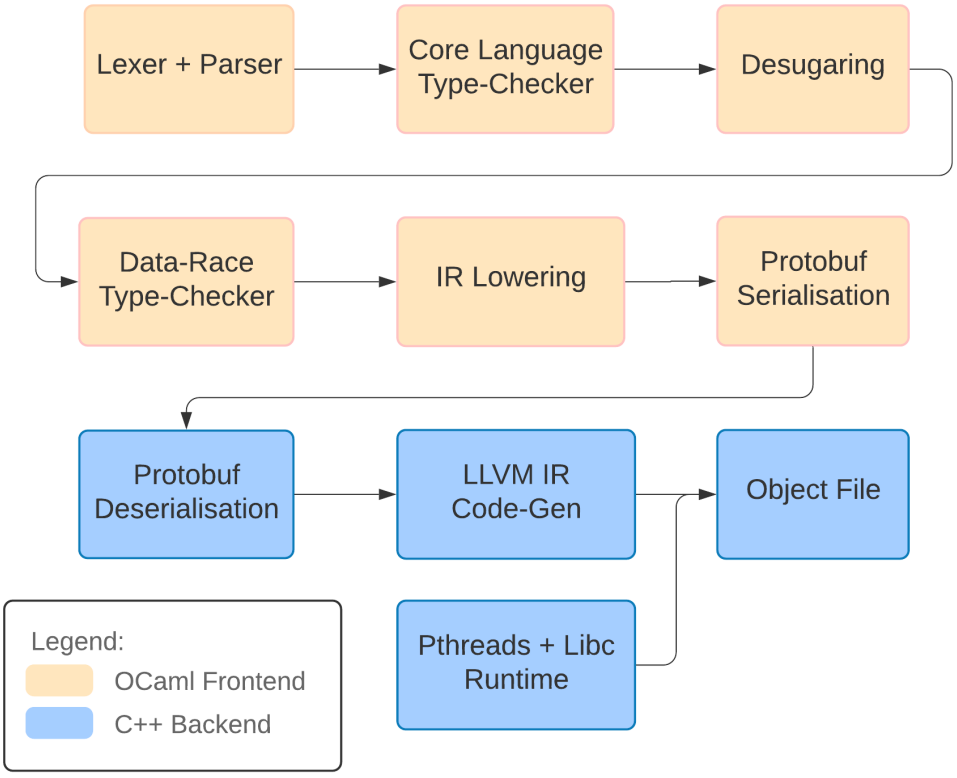


Figure 3.1: An overview of the compiler pipeline.

3.3.1 Repository Overview

The `src/` folder is split into the frontend and the LLVM backend, grouped into libraries corresponding to each stage of the compiler (table 3.1). The OCaml frontend is built using the Dune build system [6] and the LLVM backend is built the Bazel [2] build system. Within each frontend stage's library, I group the functions for each sub-task into *modules*: each `.ml` file corresponds to a module, with a corresponding `.mli` interface file. There are **58 such modules** in the repository, for example `type_alias_liveness.ml` contains the implementation of liveness analysis.

Folder	Description	Lines of Code
`scripts/`(and `Makefile`)	Automates testing, linting and autoformatting of code and generates HTML documentation from the docstrings.	206
`src/frontend/ast`	Contains type definitions and pretty-printing utilities common to all ASTs.	211
`src/frontend/parsing`	Contains lexing and parsing code.	599
`src/frontend/typing`	Core language type-checker stage	1429
`src/frontend/desugaring`	Desugaring stage between type-checking phases	1445
`src/frontend/data_race_checker`	Implements judgements in *Kappa*	2255
`src/frontend/ir_gen`	Lowers the type-checked AST to Bolt IR.	593
`src/llvm-backend/deserialise_ir`	Converts the Bolt IR into C++ class definitions.	689
`src/llvm-backend/llvm_ir_codegen`	Converts the sanitised Bolt IR into LLVM IR.	865
`tests/`	Grouped tests by testing library used: Alcotest, Expect and Google Test, and separately grouped integration and end-to-end tests.	3763

Table 3.1: Repository overview: I wrote all code present in the repository (the line count reflects this – it does not consider autogenerated test output).

3.3.2 Choice of Languages

I chose OCaml to implement the frontend of the compiler since it has good tooling for lexing and parsing and a powerful type system that ensures pattern-matching is exhaustive. This is crucial when adding new language features to Bolt: adding a new type to Bolt requires 82 other files in the frontend to be updated – with such a large compiler pipeline this cannot be tracked manually. As the OCaml LLVM API lacked support for memory fences and other concurrent memory operations, I wrote the compiler backend in C++ using the native LLVM C++ API.

3.3.3 Compiler Pipeline

Lexer and Parser Rather than handwriting my lexer and parser, I decided to use lexer and parser generators `ocamllex` [21] and `menhir` [18]. These take in a specification – regular expressions for tokens in `ocamllex`, context-free grammar rules for `menhir` – from which they autogenerate the

lexer and parser. The generators reduced the incidental complexity of adding language features in later milestones.

The ASTs in the compiler are represented as OCaml algebraic datatypes, corresponding to the expressions in the grammar. The type `loc` refers to the line number and position of the expression: I use this when printing compiler error messages.

```
type expr =
| Integer     of loc * int
| Boolean     of loc * bool
| BinOp       of loc * bin_op * expr * expr (* e.g. + operator *)
...
| Constructor of loc * Class_name.t * constructor_arg list
| Assign      of loc * identifier * expr
```

Core Language Type-Checker The core language type-checker locally infers the types of expressions (fig. 3.2), and annotates the AST with their types. Bolt expressions have either a primitive type (`bool`, `int` or `void`) or are objects belonging to some class. The type-checker checks type-parameterised classes in two stages. First it checks that the uses of the abstract type parameter within the class definition are consistent. It then type-checks each of the concrete instantiations of the type parameter.

$$\frac{\Gamma \vdash e_1 : \textbf{int}, \emptyset \quad \Gamma \vdash e_2 : \textbf{int}, \emptyset}{\Gamma \vdash e_1 + e_2 : \textbf{int}, \emptyset}(e\text{-}add) \qquad \frac{\Gamma \vdash e_1 : \textbf{bool}, \emptyset \quad \Gamma \vdash e_2 : \tau, \kappa \quad \Gamma \vdash e_3 : \tau, \kappa}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau, \kappa}(e\text{-}if)$$

Figure 3.2: Sample typing judgements for core language type-checker.

Desugaring Desugaring is the process of simplifying the representation of the AST for subsequent compiler stages. In this stage, I desugar parametric and ad-hoc polymorphism to simpler representations so they do not have to be considered in the exposition of *Kappa*.

Bolt supports two forms of ad-hoc polymorphism: variable shadowing and function/method overloading. Variable shadowing in inner scopes complicates tracking aliases of objects in the data-race type-checking stage. Likewise, when we analyse an overloaded function call, we need to first check which function definition it refers to before proceeding with the data-race type-checking – an unnecessary overhead.

I therefore traverse the AST and perform *name mangling* on the function, method and variable names. This ensures that functions are globally unique, methods are unique within their class, and variables are no longer shadowed. Desugaring generics is more challenging. I first aggregate the concrete types each generic class was instantiated with. Just like C++, I then convert the parameterised class definition into multiple class definitions, one for each concrete type it is instantiated

with. I then replace each generic type in the AST with the concrete type that object was instantiated with, and replace all references to the generic class with the corresponding type-instantiated class.

In preparation for data-race type-checking stage, I also annotate each AST identifier node with the set of possible capabilities it can use. This is used in the constraint-based analysis on this set detailed in section 3.2.3. To type-check concurrent capability accesses, we need to determine which variables are accessed in multiple threads. To do this we need only consider the free variables in each of the thread sub-expressions, since all variables bound in that thread are not visible to other threads. We annotate the `finish-async` AST node with these free variables.

```
let x  = ...; // x is visible to all threads in finish-async block
finish{
  async{
      x; // x is free in this thread sub-expression
      let y = ...; // the y is bound within this thread
    }
  ... // y is not visible in another thread.
}
```

Data-Race Typechecking This stage implements the typing judgements in *Kappa* (fig. 2.1) and the extensions to *Kappa* presented in section 3.2.

IR Lowering Having performed the core language and data-race type-checking stages, we can drop capability and type annotations for expressions since we do not need them at runtime. We insert `lock` and `unlock` instructions around any field accesses and method calls that use the *locked* capability. I generate a further desugared representation: *Bolt IR*. Bolt IR is conceptually similar to LLVM IR: to determine how to represent the richer Bolt language features in Bolt IR, I wrote a C++ program using that language feature and compiled it with `clang` to LLVM IR and analysed the output. This makes the translation to LLVM IR straightforward.

LLVM IR has no notion of classes and objects, only structs. Thus in Bolt IR, I desugar objects into structs of fields, representing each field access as an index into the struct. Methods are desugared into regular functions which pass in an additional parameter: `this` (the object calling the method). I *name mangle* each class's methods so that they are *globally* unique. Implementing *method overriding* (where we decide at runtime which method to execute using the type of the *object* not the *reference*) was particularly challenging. For each class, I constructed a *virtual table* (*vtable*) of pointers to the methods in that class. Instead of a static method call, objects contain a pointer to their class' *vtable*, and an index to the method being called. At runtime, we query this vtable to determine which method to execute.

Despite Bolt IR and LLVM IR having similar representations, the LLVM API was in C++, whilst the Bolt IR was represented as an OCaml type definition. In order to pass the Bolt IR to the backend, I needed to serialise it into a *language-independent* format. I chose the *protocol buffers* [22] (abbreviated *protobuf*) format to serialise the IR, since it was a faster and more compact data

representation than JSON or XML. Protobuf encodes structured data (like the IR) as a series of binary messages based on a message schema. I used the OCaml *deriving protobuf* library [7] both to generate the message schema (listing 7) from the IR type definitions and to serialise the IR. The Bazel build system automatically generates the C++ deserialisation methods from the schema. Bolt IR deserialises to a placeholder data representation that we sanitise into C++ classes that correspond to the IR type definitions in OCaml.

```
message expr {
  enum _tag {
    ...
    Constructor_tag = 4;
    ...
    BinOp_tag = 13;
    ...
  }
  message _Constructor {
  required string _0 = 1;
  repeated constructor_arg _1 = 2;
}
message _BinOp {
  required bin_op _0 = 1;
  required expr _1 = 2;
  required expr _2 = 3;
}
```

Listing 7: Part of the Protobuf schema autogenerated by the *deriving protobuf* OCaml library.

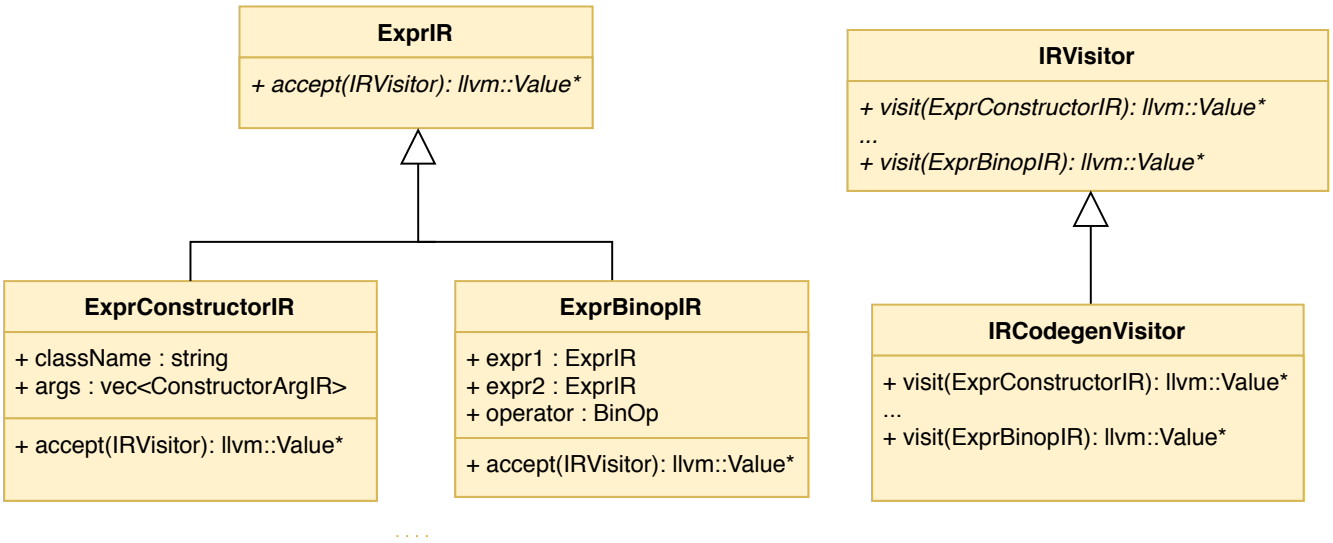


Figure 3.3: The Visitor design pattern used for LLVM IR Code generation.

LLVM IR Code-Generation I used the *Visitor design pattern* [14] when generating the LLVM IR from our sanitised frontend IR representation. This pattern (fig. 3.3) separates the code generation algorithm `IRCodegenVisitor` from the frontend IR classes it operates on (which subclass `ExprIR`).

I implemented a custom implementation of a *re-entrant* reader-writer lock. This modifies the standard reader-writer lock to allow the thread holding the writer lock to re-enter as a reader or writer. This modification is necessary as we might have nested lock/unlock instructions in our IR generated by nested invocations of an object's locked capability. With a standard reader-writer

lock implementation, a thread attempting to reacquire a writer lock will *deadlock*. When creating the object structs, we reserve fields to store the ID of the thread that currently owns the write lock and the reader/writer lock counters.

Language Runtime I bootstrap the Bolt language runtime with C libraries, declaring their function signatures in the LLVM IR and linking in the C libraries. I used the `pthreads` [17] C library to fork and join threads. One challenge is that the library's pthread type is *platform-specific*, so I represented this as an *opaque* type in the LLVM IR and dynamically linked in the correct type.

When a thread is forked from the current thread, it is allocated a fresh stack. I therefore had to copy across references to all the free variables in that forked thread from the current thread's stack when forking that thread. Since the LLVM API only supports allocation to the stack, I used the `libc`'s `malloc` function to allocate objects to a global heap. Variables point to references on the stack, which stores primitives and pointers to objects on the heap.

3.4 Summary

Kappa is presented in the context of a theoretical calculus that uses *traits*. Therefore, implementing it in the context of a practical inheritance-based language required 3 major changes. Firstly, I recasted *Kappa* to no longer require traits. Secondly, since programmer overhead is an important consideration of a practical language, I implemented a capability *inference* algorithm and incorporated liveness analysis (used in the Rust borrow checker). Finally, Bolt supports method overriding, a form of subtype polymorphism present in inheritance-based languages. Java's definition of subtyping was not sufficient to prevent data races, so I introduced a stronger notion: *behavioural subtyping* and provided additional typing constraints.

Having discussed the theoretical implementation of *Kappa* and the language design decisions for Bolt, we discussed the practical implementation of the Bolt compiler. The Bolt compiler contained two stages of type-checking, and the desugaring stage between these stages transformed ad-hoc and parametric polymorphism into a simpler representation for the data-race type-checker to operate on. This stage let us add support for additional language features without modifying *Kappa*. After type-checking, the compiler represents programs in Bolt IR, which we translate into LLVM IR and compile to native code, linking in C runtime libraries to support hardware threading.

Chapter 4

Evaluation

The programming language Bolt far exceeds all success criteria set out in the project proposal (section 4.1). In the introduction, I set out that Bolt would demonstrate how data-race freedom could interoperate with inheritance-based languages. I demonstrate this in section 4.2 where I transliterate Java programs that are hundreds of lines long into Bolt, including implementations of data structures like Binary Search Trees, only requiring a single annotation per field or method. Section 4.3 compares the concurrency paradigms representable in Bolt against Java and Rust. Finally, in section 4.4 I show that Bolt outperforms Java on the benchmarks.

4.1 Review of Project Requirements

Reviewing the requirements in the preparation chapter (section 2.5), I have achieved not only the core deliverable and stretch requirements but also the nice-to-have extensions. Evaluating against the original success criteria detailed in the proposal (appendix C):

Design a minimal concurrent object-oriented language Bolt far exceeds this, with support for control flow, inheritance, method overloading and overriding and generics. Bolt achieves all of the *Must-have*, *Should-have* and *Could-have* language features detailed in the MosCoW analysis (table 2.1).

Implement the core subset of Kappa not only have I implemented the entire *Kappa* type system, but I have also recast the ideas from it to work in inheritance-based languages like Java. I removed *Kappa*'s requirement for traits and extended it to support inheritance and subtype polymorphism. I also reduced the programmer overhead of *Kappa* with a capability inference algorithm.

OCaml bytecode interpreter simulating concurrency Having achieved this early in the project development, as an extension to target performance, I targeted LLVM IR to compile Bolt to native code with hardware thread support.

Corpus of Bolt programs for evaluation I used the Jane Street Expect Test library to evaluate Bolt's type-checker using Bolt programs provided as input, writing **over 200 tests**. The

subset of the corpus with data races all fail to type-check, providing empirical evidence for Bolt's assurance of data-race freedom.

4.2 Transliteration Between Java and Bolt

```
public class BSTNode<K extends
  Comparable<K>, T> {


    private K key;
    private T value;
    private BSTNode left;
    private BSTNode right;
    private int size;

    T search(K k){
      ...
    }
  }
public class BST<K extends
  Comparable<K>, T> {
  private BSTNode<K,T> root;
}
```

(a) Java

```
class BSTNode<T>{
  capability linear Left, linear Right,
       linear Elem, read Search;

  var int key : Elem, Search;
  var T value: Elem, Search;
  var BSTNode<T> left: Left, Search;
  var BSTNode<T> right: Right, Search;
  var int size: Elem, Search;
  var bool isNull: Elem, Search;
  T search(int key) : Search {
    ... // no annotations in body
  }
}
class BST<T>{
  capability read View, linear Update;
  var BSTNode<T> root: View, Update;
}
```

(b) Bolt

Listing 8: An example Binary Search Tree class definition I transliterated from Java to Bolt.

One of the claims made in the preparation chapter is that Bolt reduces the programmer overhead present in *Kappa*. However this comparison is somewhat artificial as Castegren and Wrigstad present a *theoretical* calculus. The `pack`, `unpack` and `jail` instructions are present purely to simplify the presentation of *Kappa*, with simpler-to-use programming constructs detailed in the paper as a direction for future work. It is more apt to evaluate Bolt's annotation overhead against Java, the language it set out to model.

Annotations can either be *declaration-site* (annotating field/method definitions) or *use-site* (when accessing an object's fields or methods). To minimise the annotation overhead, Bolt uses *declaration-site* annotations: annotations are only needed for fields and in function/method type signatures, with none present in the function/method bodies or the main expression. One practical benefit over *use-site* annotations is that users of a library that implements capabilities do not need to annotate their code.

Listing 8 shows how a Binary Search Tree written in Java maps to Bolt. We choose capabilities based on how we expect to use the object: here I demonstrate a lock-free implementation. The most common lock-free concurrency pattern is the *Multiple Reader Single Writer* pattern, which

we can represent with *linear* and *read* capabilities. Since we would like to mutate the `BSTNode`'s value and its left and right children concurrently, we assign them *disjoint* linear capabilities. Most of the transliteration overhead actually came from the difference in the languages' feature sets. As is to be expected with a compiler project of this size, Bolt lacked some of the features present in Java such as bounded generics `<K extends ...>`, *null* comparisons or exceptions. This meant error-checking code present in Java had to be rewritten in the style of C, returning *error codes* in place of exceptions. Thus Bolt's BST implementation uses an `isNull` flag and an `int` key for comparison.

It is also possible to transliterate from Bolt to Java. We can use a *marker interface* `DataRaceFreedom` to indicate that a class uses capabilities and should be type-checked using *Kappa*. Marker interfaces contain no fields or methods and are used to indicate to the JVM that a class has special behaviour (`Serializable` indicates that objects of that class can be serialised). We could throw an exception if an object is used in a concurrent context without implementing `DataRaceFreedom`, similar to how a `NotSerializableException` is thrown if we attempt to serialise an object that doesn't implement `Serializable`. Within each class, we can use Java *annotations* to annotate capabilities: for example annotating a *linear* `Age` capability to a `Person` class. These annotations could be checked at compile-time and then compiled away, a form of *type erasure* akin to generics (where `ArrayList<String>` is compiled to `ArrayList`). *Kappa* would thus have **no runtime overhead** and be **backwards-compatible** with existing Java code:

```
public interface DataRaceFreedom { }

class Person implements DataRaceFreedom {
  @capability(name=Age mode=linear)
  @fieldCapability(name=Age)
  private int age;

  @methodCapability(name=Age)
  int getAge(){
    return age;
  }
}
```

4.3 Expressivity of Bolt

Proving an absence of data races is an undecidable problem, so Bolt conservatively *overapproximates* the set of programs containing data races and rejects safe programs. As with other type systems, this analysis is *syntactic* rather than *semantic*. Bolt rejects programs where a branch contains data races even if that branch would never be taken. A more interesting discussion is analysing which concurrency patterns Bolt supports in comparison to Java and Rust (table 4.1).

Bolt is as expressive as Java when considering single-threaded programs. To convert a Java program to Bolt, we can annotate all classes with a single *thread-local* capability (which allows unrestricted

Concurrency Pattern	**Bolt**	**Java**	**Rust**
Single-threaded:			
Mutating linear references	Yes (`linear`)	Yes	Yes
Aliasing and mutation	Yes (`local`)	Yes	No
Multi-threaded:			
Independent computations	Yes	Yes	Yes
Concurrently mutate disjoint parts of object	Yes	Yes*	No
Concurrently mutate object, locking overlapping state	Yes	Yes*	No
Concurrent read-only access	Yes (`read`)	Yes	Yes (`Arc`)
Reader-writer lock	Yes (`read`/`locked`)	Yes (`ReadWriteLock`)	Yes (`RwLock`)
Automatic lock insertion	Yes (`locked`)	Yes (`synchronised`)	No
Programmer-specified:			
Locking instructions	No	Yes (`Lock` interface)	Yes (`Mutex`)
Atomic access instructions	No	Yes (`volatile`)	Yes (Atomics)
Inter-thread Communication:			
Producer-Consumer	Partially (`locked`)	Yes (Monitors)	Yes (channels)
Message Passing	No	No	Yes (channels)
Actors	No	No	No (channels)

Table 4.1: A comparison of concurrency patterns supported by Bolt, Java and Rust. (*no protection against data races)

aliasing and mutation within a single thread). However, Rust's type system prevents mutation in the presence of aliasing, so rejects some safe single-threaded programs.

Bolt, Java and Rust all support *embarrassingly parallel* independent computations with no shared state. Java accepts all programs with shared state concurrency, however provides no data race protection. Bolt supports more **fine-grained concurrency** than Rust. Rust rejects programs where objects are safely being concurrently mutated and any overlapping state is thread-safe. This belies the difference in information encoded in the type systems. Rust's ownership system is all-or-nothing: references are either able to mutate *all* fields of an object (if *owning reference*) or none (if *aliased*). The type system cannot restrict which parts of the object the reference accesses in each thread, so conservatively prevents it being used in multiple threads. However, Bolt's capabilities operate on the level of *individual field annotations*, rather than the *entire object*, so Bolt can guarantee the capabilities access disjoint or thread-safe state (as in listing 8). Bolt can emulate Rust's core ownership type system using capabilities:

```
class RustEmulator {
  capability linear ownerRef, read borrowedRef;
  var int f : ownerRef, borrowedRef;
}
void main(){
  let z = new RustEmulator();
  z.f := 2; // can write to x when owned
  let x = consume z  // can transfer ownership with consume
  let y = x; // x aliased (no longer linear)
  ... // so x only has read capability as "borrowed"
}
```

However, unlike Java and Rust, Bolt does not support any programmer-managed locking and memory accesses, managing all accesses via the type system. Whilst programmer-managed concurrency can be error-prone, there *are* intricate concurrency algorithms over shared data structures that require this level of precision. One example is *hand-over-hand locking* when traversing a tree, where programmers acquire the lock on the root node, acquire the lock on a branch, release the root node, and repeat, iteratively traversing the tree. This is impossible to represent in Bolt since, like Java's `synchronised`, locks can only be released in the opposite order they were acquired. Rust strikes a balance, not allowing this functionality in the core type system, but providing escape hatches through libraries. These libraries can use the type system to reduce bugs: `Mutex<T>` automatically releases a lock once the reference using it is no longer live.

Shared-state concurrency is the primary paradigm supported by Bolt and Java however it is not sufficient to represent all concurrent programs. One common concurrency pattern that Bolt only partially supports is the *producer-consumer* paradigm. Bolt can manage the shared buffer via locks but there is no mechanism for a thread to wait on another thread's computation. Java supports *monitors* which lets threads wake up other threads sleeping on a condition but does not support other inter-thread communication. However, Rust's type system supports an alternative concurrency paradigm introduced by the language Go that allows this: *message passing* between threads via shared data-race-free *channels*. Rust adapts its ownership type system to work with channels: a thread can relinquish ownership of a value by pushing it to a shared channel, and another thread then takes ownership of this value by consuming it from the channel. Finally, a concurrency paradigm that none of the languages support is the *actor* model, where lightweight threads (actors) request other actors to do work *on their behalf*.

4.4 Benchmarks

The primary focus of this project is to demonstrate data-race freedom in a practical programming language. Here we show that Bolt also outperforms Java. I run the benchmarks on a Macbook Pro (2016) with the following specification:

Processor Intel Core i7-6700HQ CPU @ 2.60GHz

Memory 16 GB 2133 MHz LPDDR3 RAM

OS macOS Mojave Version 10.14.6

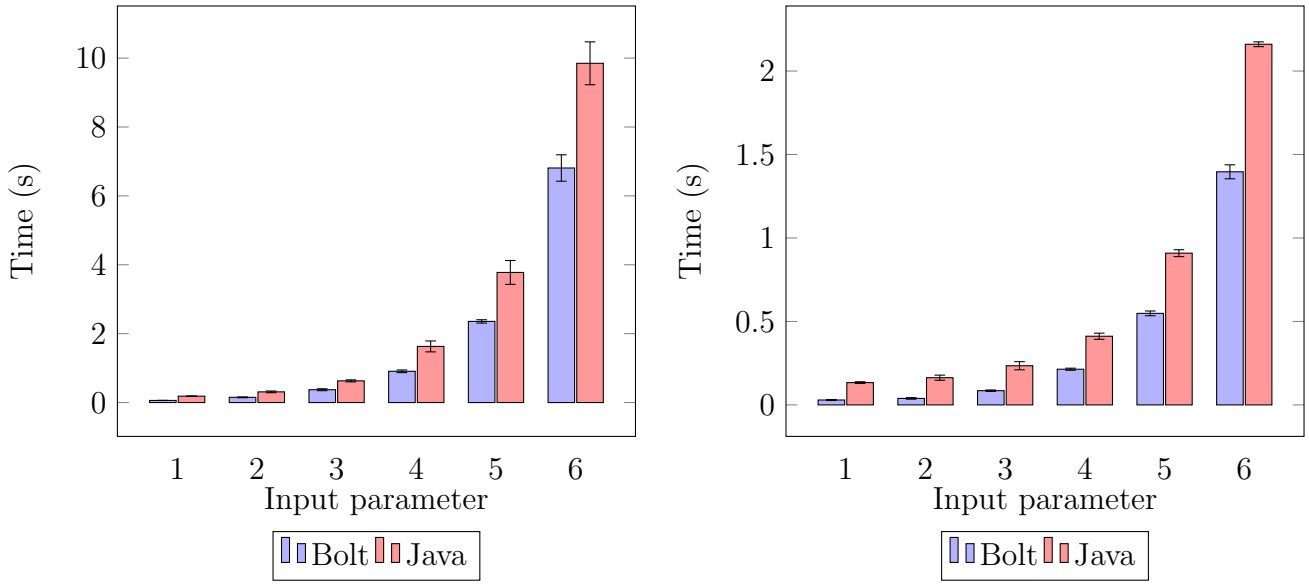
Java Version openjdk 13.0.2, OpenJDK Runtime Environment (build 13.0.2+8)

JVM OpenJDK 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)

LLVM IR Optimisation Passes Mem2Reg, Instruction Combining, Reassociate Expressions, GVN Elimination, CFG Simplication Pass

Clang `-O3` optimisation flag

Bolt programs compile to LLVM IR. I then compile the IR file using `clang` to an object file that is run natively. Java programs run on the JVM. I ensure the machine is unloaded, and I clear file system caches and swap: this minimises the variance between the earlier and later timings. Java lazily loads in classes at runtime using the class loaders – this overhead would be included in the earlier timings if I took the initial runs. To allow the JVM to warm up I benchmark 13 runs, and discard the timings of the first 3 runs. To minimise the effect of garbage collection on the Java benchmarks, I initially provide the JVM 1GB of RAM (the default is 256MB), with the max heap size set to 4GB (so garbage collection will not need to be run). The benchmarks are automated by a Bash script, which uses the Unix `time` command. For small timings, we are limited by the precision of the `time` command, so we execute the benchmarks multiple times in a single run, averaging our timings over a longer duration to get a more precise timing.



(a) Single-threaded benchmark

(b) Multi-threaded benchmark

Figure 4.1: Benchmarks (appendix B) comparing Bolt's single-threaded and multi-threaded performance against Java. Error bars represent $\pm 2\sigma$.

In both the single-threaded and multi-threaded benchmarks (fig. 4.1), Bolt outperforms Java by a factor of 1.5, most pronoucedly on the smaller input parameters. This can be explained by 2 main

factors: Bolt compiles ahead-of-time so benchmarks run native code, whereas Java's just-in-time (JIT) compiler performs bytecode to native code translation at runtime, which adds a runtime overhead. Secondly, Bolt takes advantage of the highly optimised Clang-LLVM toolchain (used by C/C++), which optimises code far better than Java's JIT compiler. Upon inspection, the assembly code produced by Bolt contains far fewer instructions than the equivalent Java bytecode. For both Bolt and Java, the primary overhead in the multi-threaded benchmark is the cost of system calls to spawn and clean up hardware threads; a thread pool implementation could amortise these costs.

4.5 Summary

Bolt was a success, achieving all the core success criteria and extensions listed in section 2.5. Bolt outperformed Java on the benchmarks, validating my decision to target LLVM IR for performance. Java programs could be transliterated into Bolt with only declaration-site annotations. Bolt programs could be transliterated into Java using marker interfaces and annotations, and these capability annotations had no runtime overhead. Crucially, I showed that this approach was backwards-compatible with existing Java code, and that users using a library with capabilities would not need to annotate their code.

Comparing Bolt, Java and Rust, it was clear that Bolt's type system offered the most expressivity of any *single* approach to lock-free shared-state concurrency, and that the Rust's core type system could be emulated in Bolt. The *hand-over-hand locking* example illustrated that letting programmers manage concurrency allows programmers to implement more intricate concurrency algorithms. Here Rust's approach towards concurrency is best: the type system shoulders most of the burden for ensuring data-race freedom but offers escape hatches via libraries for programmers to take over when the type system is unable to express safe programs.

In the same vein, it became apparent that a single concurrency paradigm (shared-state) implemented by all three languages was insufficient to represent all concurrent programs. This reflects a common theme across programming languages: one paradigm cannot capture all programming styles. Rust was the only language of the three to offer a hybrid approach, combining shared-state with message-passing. The best approach would therefore be to integrate these alternative concurrency paradigms into Bolt, and provide an escape hatch for programmers to manage concurrency.

Chapter 5

Conclusions

The project was a success: I exceeded all of the core success criteria, including extensions (section 2.5) and demonstrated how the ideas from *Kappa* could be applied to inheritance-based languages like Java. This achieved the aim set out in the introduction: *to demonstrate how data-race freedom could be retrofitted to the type systems of the traditional Java-style languages.*

The programming language Bolt was richer than the core deliverable or even the scheduled extensions: supporting inheritance and ad-hoc, subtype and parametric polymorphism. I was able to transliterate Java programs that were hundreds of lines long into Bolt, with minimal annotation overhead thanks to the capability inference algorithm. These included common data structures such as trees and linked lists. Moreover, I also demonstrated how other language features like ad-hoc and parametric polymorphism could interoperate with the data-race type-checker by desugaring them to a simpler intermediate representation.

I also exceeded the initial objective of an OCaml interpreter: the compiler generates LLVM IR and thus targets native code, and the language runtime supports hardware threads and a global heap. The resulting compiler outperforms Java.

5.1 Lessons Learnt

Surprisingly, one of the simplest milestones (the LLVM IR generation) was hardest to implement. Whilst Bolt IR and LLVM IR were conceptually similar, translating between them took longer than expected due to the enormity of the LLVM API and the comparative lack of safety provided by C++ compared to OCaml. Fortunately, the slack time budgeted in the proposal time plan enabled me to avoid project delays.

There is a balance in a project of this form between learning more theory and exploring alternative approaches versus implementing the current type system. The understanding gained ultimately led to a richer evaluation of the approach presented in *Kappa*, however many approaches could not be integrated into the type system presented.

The length of the compiler pipeline substantially increased implementation time – adding new language features took progressively longer as the changes had to be propagated through each stage, touching over 100 files in the process in the latter stages of the project. I had not anticipated such a slowdown when starting the project, and would have budgeted more time accordingly. Method overriding and generics were examples of language features that proved to be more challenging undertakings. The complexity of the generics implementation was especially surprising as it did not require any modifications to *Kappa*, the conceptually most difficult aspect of the compiler.

Upon reflection, I found it was easy to be over-optimistic about time required to implement conceptually straightforward modules. For example, I had not expected that a simple `count_instantiations` function used when desugaring generics would end up being 110 lines of code once all cases were considered. This project was substantially larger than any other software engineering project I had undertaken (14,000 lines of code versus a couple of thousand lines) so I had not factored these practical implementation risks in my spiral model. These practical considerations are especially important for industrial compilers, which are hundreds of thousands of lines in size. Going forward this is something I will weight more when performing risk analysis for future engineering projects.

5.2 Future Work

Having demonstrated an equivalence between Java and Bolt programs, a natural extension would be to integrate *Kappa* into the Java type-checker, using the syntax outlined. Åkerblom, Castegren and Wrigstad's follow-up paper [8] extends *Kappa* to support concurrent array algorithms, increasing the subset of Java that could be represented.

In this dissertation, I explained how the modifications I made to *Kappa* preserved data-race freedom and substantiated these claims against a corpus of Bolt programs. A written proof of a type system does not however guarantee soundness if there is a bug in the type-checker, so increasingly soundness is proven using a proof assistant like Coq or Isabel. Providing a mechanised proof of data-race freedom is a substantial endeavour and beyond the scope of this dissertation, however is an interesting direction for future work.

Bibliography

[1] Alcotest testing framework. `https://github.com/mirage/alcotest`. Accessed: 2020-04-10.

[2] Bazel Build System. `https://bazel.build/`. Accessed: 2020-04-10.

[3] Coveralls.io. `https://coveralls.io/`. Accessed: 2020-04-10.

[4] Github Project Board. `https://help.github.com/en/github/managing-your-work-on-github/about-project-boards`. Accessed: 2020-04-10.

[5] Google Test framework. `https://github.com/google/googletest`. Accessed: 2020-04-10.

[6] OCaml Dune build system. `https://dune.build/`. Accessed: 2020-04-10.

[7] OCaml PPX Deriving Protobuf. `https://github.com/ocaml-ppx/ppx_deriving_protobuf`. Accessed: 2020-04-10.

[8] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. Reference Capabilities for Safe Parallel Array Programming. *arXiv preprint arXiv:1905.13716*, 2019.

[9] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[10] Barry W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[11] Elias Castegren and Tobias Wrigstad. Reference capabilities for trait based reuse and concurrency control, 2016.

[12] Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[13] C++ Standards Committee et al. ISO/IEC DIS 14882, Standard for Programming Language C++(C++ 17). Technical report, Technical Report. ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee . . . , 2017.

[14] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[15] Open Source Initiative et al. The MIT license. *2015b.[Online]. Available: https://opensource.org/licenses/MIT.[Accessed 27 March 2017]*, 2006.

[16] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[17] Frank Mueller. Pthreads library interface. *Florida State University*, 1993.

[18] François Pottier and Yann Régis-Gianas. Menhir reference manual. *Inria, Aug*, 2016.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[20] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[21] Joshua B Smith. Ocamllex and Ocamlyacc. *Practical OCaml*, pages 193–211, 2007.

[22] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72, 2008.

[23] Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. Oxide: The Essence of Rust. *arXiv preprint arXiv:1903.00982*, 2019.

Appendix A

Grammar

Bolt is a concurrent object-oriented language defined by the following grammar:

Cl	class name
c	capability name
f	field name
m	method name
fn	function name
x	variable name
n	integer
b	boolean
T	type parameter
j, k, l	index variable

program	::=		
	|	$Cd_1 \ldots Cd_j\ Fnd_1, ..., Fnd_k$ **void main** $()\{e_1; ...; e_l\}$	
Cd	::=		Class definition
	|	**class** $Cl\{Capd\ Fd_1, ..., Fd_j\ Md_1, ..., Md_k\}$	
	|	**class** $Cl < T > \{Capd\ Fd_1, ..., Fd_j\ Md_1, ..., Md_k\}$	Generic Class
mode	::=		Modes
	|	**linear**	
	|	**subordinate**	
	|	**thread**	
	|	**read**	
	|	**locked**	
Capd	::=		Capability definition
	|	**capability** $mode_1\ c_1, ..., mode_k\ c_k$	
modifier	::=		
	|	**const**	
	|	**var**	
Fd	::=		Field definition
	|	*modifier type* $f : c_1, .., c_k;$	
param	::=		Method/Function Parameter

	$\mid$ $type\ x$	
	$\mid$ $type\{c_1, ..., c_k\}x$	
Md	::=	Method definition
	$\mid$ $type\ m(param_1, ..., param_j) : c_1, ..., c_k\{e_1; ...; e_l\}$	
Fnd	::=	Function definition
	$\mid$ **function** $type\ fn(param_1, ..., param_j)\{e_1; ...; e_k\}$	
type	::=	Types of expressions
	$\mid$ Cl	
	$\mid$ $Cl < type >$	Generic class
	$\mid$ **borrowed** Cl	
	$\mid$ **int**	
	$\mid$ **bool**	
	$\mid$ **void**	
	$\mid$ T	Type parameter
id	::=	Identifier
	$\mid$ x	Variable
	$\mid$ $x.f$	Field access
e	::=	Expression
	$\mid$ $unop\ e$	
	$\mid$ $e_1\ binop\ e_2$	
	$\mid$ (e)	
	$\mid$ n	Integer
	$\mid$ b	Boolean
	$\mid$ id	Identifier
	$\mid$ **new** $Cl(f_1 : e_1, ..., f_k : e_k)$	Constructor
	$\mid$ **new** $Cl < T > (f_1 : e_1, ..., f_k : e_k)$	
	$\mid$ **let** $x = e$	Define identifier
	$\mid$ **let** $x : type = e$	
	$\mid$ $id := e$	Assign to identifier
	$\mid$ **consume** id	Consume identifier
	$\mid$ $x.m(e_1, ..., e_k)$	Object method
	$\mid$ $fn(e_1, ..., e_k)$	Application
	$\mid$ **if** $e\{e_1; ...; e_k\}$ **else** $\{e_1; ...; e_l\}$	If-else statement
	$\mid$ **while** $e\{e_1; ...; e_k\}$	While loop

	$\mid$ **for** $(e_1; e_2; e_3)\{e_1; \ldots; e_k\}$	For loop
	$\mid$ **finish** $\{$ **async** $\{e_1; \ldots; e_j\}; \ldots;$ **async** $\{e_1; \ldots; e_k\}; e_1; \ldots; e_l\}$	Finish async
unop	::=	
	$\mid$!	Not
	$\mid$ $-$	Negation
binop	::=	
	$\mid$ +	Addition
	$\mid$ *	Multiplication
	$\mid$ /	Integer division
	$\mid$ %	Remainder
	$\mid$ <	Less than
	$\mid$ <=	Less than equal
	$\mid$ >	Greater than
	$\mid$ >=	Greater than equal
	$\mid$ ==	Equal
	$\mid$! =	Not equal
	$\mid$ &&	Logical and
	$\mid$ $\mid\mid$	Logical or

Appendix B

Benchmarks

For the single-threaded benchmark, we offset our input to `fib` in order to get timings with suitable precision (small values passed to `fib` complete near-instantaneously and are hard to measure accurately.

```
function int fib(int n){
  if ((n==0) || (n==1)){
    1
  } else{
    (fib(n - 1)) + (fib(n - 2))
    }
}
function void benchmark1(int n){
  fib((2*n)+34)
}
```

(a) Bolt

```
static int fib(int n){
  if (n ==0 || n==1){
  return 1;
  }
  else return fib(n-1)+fib(n-2);
}

void benchmark1(int n){
  fib((2*n)+34);
}
```

(b) Java

Listing 9: Single-threaded benchmark.

In the multi-threaded benchmark, we use the single-threaded version of `fib` for smaller values passed to `asyncFib` to prevent too many threads from being spawned. Otherwise for small cases to `asyncFib` the benchmark for both *Bolt* and Java would be dominated by system calls spawning threads.

```
    class FibHelper{
      capability linear left,
      linear right, read view;
      var int left : left;
      var int right : right;
      const int n : view;
    }
    function int asyncFib(int n){
      if(n <35) {
          fib(n)
      }
      else{
        let x = new FibHelper(n : n);
        finish{
          async{
            x.left :=  asyncFib(x.n - 1)
          }
          x.right :=  asyncFib(x.n - 2)
        };
        x.left + x.right
      }
  }
function void benchmark2(int n){
  asyncFib((2*n)+34)
}
```

```
public class AsyncFibonacci{
        int left;
        int right;
        int n;
        AsyncFibonacci(int n){
                this.n = n;
  }
  static int asyncFib(int n){
    if (n < 35){
      return fib(n);
    }
    AsyncFibonacci x = new
        AsyncFibonacci(n);
    Thread t = new Thread() {
      public void run() {
        x.left = asyncFib(x.n - 1)
      }
    };
    t.start();
    x.right = asyncFib(x.n-2);
    try{
    t.join();
    } catch(InterruptedException e){}
    return  x.left + x.right;
  }
}
void benchmark2(int n){
  asyncFib((2*n)+34);
}
```

Listing 10: Multithreaded benchmark for *Bolt* and Java respectively.

Appendix C

Project Proposal

Introduction and Description of the Work

Data races occur when two threads are trying to access the same data address concurrently, leading to inconsistent state. For example, a data race may occur when trying to credit a bank account whilst calculating interest on the current amount. Data races are notoriously difficult to detect and reproduce, since it is difficult to predict the order of execution of large multi-threaded systems, and some orders of execution are so seldom traversed that bugs only manifest years later. These bugs are so prevalent in modern concurrent programs that they've been dubbed *Heisenbugs*.

Traditional mechanisms of preventing data races such as locking place the onus on programmers to correctly enforce the guarantees, which leads to other issues like deadlock.

The solution therefore is for the type system to statically guarantee data-race freedom, a notable example being Rust's type system's *ownership* feature - which only allows one thread to own a reference to the data at any given time. However Rust's *ownership* feature does not support the Multi-Reader Single Writer concurrency design pattern, where multiple threads can read the data concurrently (since reading doesn't alter the data). The concurrent object-oriented programming languages Encore and Pony have a richer type system, which associates a capability with each reference based on the type of data access (read/write) and number of threads accessing the reference. This affords a greater degree of concurrency whilst still providing the same static guarantees.

In this project, I will create an interpreter for a simplified object calculus henceforth provisionally dubbed *Bolt* and adapt the static type system Kappa [1] used in Encore to implement a static type checker *Pauli* for the calculus. I will demonstrate that *Pauli* enforces the property of data-race freedom in *Bolt* and thus excludes this class of *Heisenbugs* from the programs that type-check.

Starting Point

I have no previous experience in implementing type checkers or interpreters, however I studied Semantics of Programming Languages and Compiler Construction. The Concurrent and Distributed Systems course I studied last year will be relevant when identifying programs that have data races.

Success Criteria

For the project to be deemed a success, the following items must be successfully completed:

1. Design the simplified object calculus *Bolt* in which data races can occur. This should support classes with directly-accessible fields (no methods), first-order functions, and structured concurrency in the form of explicit forking and synchronisation points.
2. Implement an interpreter for *Bolt*. This will use a random scheduling algorithm to interleave the thread computations in order to simulate potential data races.

3. Implement the type checker *Pauli* for the calculus *Bolt*. This should distinguish between data references that are exclusive to a thread and those that are shared.
4. Create a corpus of *Bolt* programs that can be used to evaluate the static type checker *Pauli*, with a subset exhibiting potential data race conditions.

Evaluation

The structure of the project does not lend itself to quantitative evaluation, so I will instead evaluate *Pauli* using the test suite of *Bolt* programs created as part of the success criteria.

The corpus of *Bolt* programs will be handcrafted, taking inspiration from the Concurrent and Distributed Systems course, which gave pseudocode of programs with concurrency bugs. In addition, I will look at simple concurrent programs from other languages such as Java, and map these to the *Bolt* calculus.

I will evaluate the program's output when run using the interpreter (noting any data races that may have occurred) and compare this with whether the program type checks using *Pauli*.

In addition to checking the soundness of *Pauli* using the test suite (programs with data races do not type check), I will also evaluate the expressiveness of *Pauli* by analysing which data-race-free programs do not type check.

Possible Extensions

The following is a list of possible extensions to the project:

1. Augment interpreter to maintain state about each thread's data accesses to determine if *any* concurrent execution order has a data race.
2. Add ability to compose reference type capabilities in *Pauli*, and introduce a hierarchy of reference types for nested data references.
3. Implement data-race-free data structures using *Pauli*, such as Binary Search Trees.
4. Extend *Pauli* based on follow-up papers related to Encore - again evaluating with a corpus of *Bolt* programs to determine the increase in expressiveness.
5. Add type inference to *Pauli*.

Timetable and Milestones

Weeks 1 to 2 (25 Oct 19 - 7 Nov 19)

Proposal Submitted

Read up on OCaml syntax and the resources available (e.g. lexers and parsers) for writing an interpreter and type-checker in OCaml.

Read ahead in the Optimising Compilers course about liveness analysis in preparation for type-checking the references.

Weeks 3 to 4 (8 Nov 19 - 21 Nov 19)

Design the language syntax for *Bolt*.

Implement the lexer and parser for the language syntax designed in the previous block, and write tests to verify the output abstract syntax trees are correct.

Milestone: Given *Bolt* programs, generate the abstract syntax trees.

Weeks 5 to 6 (22 Nov 19 - 5 Dec 19)

Implement the interpreter for *Bolt*. Write some *Bolt* programs and verify the interpreter executes them correctly.

Milestone: Run *Bolt* programs using the interpreter.

Weeks 7 to 9 (6 Dec 19 - 26 Dec 19)

End of Michaelmas Term - start of Christmas holidays.

Implement *Pauli* and continue to write further test *Bolt* programs and verify *Pauli* type checks them correctly.

Take time off for Christmas.

Milestone: Given a *Bolt* program, determine whether it type-checks using *Pauli*.

Weeks 10 to 12 (27 Dec 19 - 9 Jan 20)

Slack time to finish off any implementation.

Milestone: Finish core project implementation.

Weeks 13 to 14 (10 Jan 20 - 23 Jan 20)

End of Christmas holidays - start of Lent Term.

Prepare further test cases for the core project evaluation. Draft the Progress Report and discuss this with supervisor well ahead of deadline.

Milestone: Finish core project evaluation.

Weeks 15 to 16 (24 Jan 20 - 6 Feb 20)

Finalise progress report and presentation.

Start work on some of the possible project extensions if time allows - adjust timetable based on current progress. Primarily focus on Extension (2) - adding the ability to compose reference type capabilities.

Deadline (31 Jan): Submit progress report.

Weeks 17 to 18 (7 Feb 20 - 20 Feb 20)

Continue to work on Extension (2) and implement more complex test *Bolt* programs that use these new composed reference capabilities.

Start writing drafts for Introduction and Preparation chapter.

Milestone: Finish implementation of Extension (2).

Milestone: Complete Draft Introduction and Preparation chapters.

Weeks 19 to 20 (21 Feb 20 - 5 Mar 20)

If time permits, implement Extension (3) - i.e. create a set of *Bolt* data-race-free data structures using the primitives in the language.

Wrap up work on extensions and focus on writing draft Implementation chapter.

Milestone: Implement data-race-free data structures in *Bolt*.

Milestone: Complete Draft Implementation chapter.

Weeks 21 to 23 (6 Mar 20 - 26 Mar 20)

End of Lent Term - start of Easter holidays.

Write up draft Evaluation chapter, and discuss evaluation with supervisor.

Based on feedback, write up more tests and evaluation metrics if needs be (slack time to continue extended evaluation of project).

Finish Conclusions chapter. Aim to send first draft to supervisor for feedback.

Milestone: Finish first draft of dissertation.

Weeks 24 to 25 (27 Mar 20 - 9 Apr 20)

Revise for exams whilst waiting on supervisor feedback.

Week 26 (10 Apr 20 - 16 Apr 20)

Incorporate feedback from supervisor and submit draft to Director of Studies. Spend time going through code repository and checking code style and layout.

Week 27-30 (17 Apr 20 - 1 May 20)

Revise for exams. Slack time in case additional evaluation and tests need to be written. **Milestone (1 May):** Submit Dissertation!

Resource Declaration

I will be using my personal laptop (MacBook Pro 2016 - 2.6GHz i7, 16GB RAM) as my primary machine for software development, and will use the Computing Service's MCS as backup, along with periodic backups to Google Drive and Git version control.

Bibliography

[1] Elias Castegren and Tobias Wrigstad. Reference capabilities for trait based reuse and concurrency control, 2016.