Statically Preventing Data-Races in Concurrent Programming Languages.

Oliver Salman 13881750

Supervised by Dr. Luke Mathieson

December 1, 2023

Abstract

The undecidability of proving the absence of data-races in concurrent programming presents a challenge in ensuring memory safety. Most contemporary type systems largely rely on the programmer to guarantee data-race freedom through the use of locks, semaphores, or other synchronising mechanisms. This approach is error-prone and does not scale to larger code bases. Addressing this gap, this dissertation introduces Poppy. Poppy is a novel programming language designed for safe, concurrent, object-oriented computations. It is designed to handle concurrent computations in the type system, eliminating the possibility of human error when manually managing threads. Poppy explores the adoption of reference capabilities to guarantee the absence of data races. It uses a combination of tools and techniques from other notable type systems, namely the Rusts type system and the Kappa type system, to statically guarantee data race freedom. This dissertation will discuss the implications Poppy has in safe, concurrent programming and the theory behind its type system including modifications to the aforementioned type systems. Further, the implementation of Poppy will be discussed, including various examples to prove its non-triviality and its practical application in the ever-growing field of concurrent programming.

Topic: Data Race Prevention

Keywords: Data Race, Type Theory, Compiler Design, Concurrency.

Thesis Statement: Creating an expressive type system can statically prevent race conditions in concurrent programming.

Research Question: Derived from the thesis statement poses the research question:

- How race conditions in concurrent programming be prevented using features from existing static type systems.

Aims: Derived from the research question is a list of project aims:

- **Aim 1:** Design an minimally viable object oriented programming language which implements fork-join parallelism.
- **Aim 2:** Design a static type system, implementing the core subset of the Kappa type system.

Objectives: Derived from the list of project aims is a list of objectives. Each objective is correspondent to the project aims and are measurable. They are:

- **Aim 1:**
  - Gather necessary requirements to carry out this project within the specified time frame.
  - Determine Poppy's concurrency and memory model
  - Define "minimally viable" in the context of Poppy's language features.
  - Develop a concise yet expressive syntax which is able to support all proposed language features.
  - Develop Poppy's semantics, which will dictate the behaviour of a Poppy program.
- **Aim 2:**
  - Understand the Kappa type system.
  - Analyse Rust's ownership and borrowing concepts.
  - Develop an explainable, detailed, and efficient typing environment.
  - Type check Poppy programs.
  - Verify the type system against test programs.

Possible extensions: The following list are possible extensions to the project:

- Introduce generics
- Introduce polymorphism
- Package import capabilities

Contents

Chapter 1

Introduction

Thread scheduling and execution in concurrent programming languages are inherently non-deterministic and as a result, race conditions can arise in which the outcomes of the program depend on the scheduling of thread interleaving. Data races are a category of bug that falls under these race conditions and can be next-to-impossible to debug. Data races arise in the presence of mutable shared resources, when multiple threads simultaneously access the same memory address, with at least one of the accesses being a write operation. Resulting of this is unpredictable program behaviour that never guarantees a program's true correctness. Despite the severity of data races, there is still little support from older sequential programming languages to help ensure correctness in concurrent codebases. The onus is on the programmer to correctly guard any concurrent accesses. Of course, this method is error-prone and does not scale to large code bases. There have been a plethora of advancements where compilers are now robust enough to aid the programmer in writing type-safe concurrent programs. This is most notable in the concept of Ownership that Rust employs, which allows at most one thread to own a reference to a piece of data at any given time, effectively preventing data-races. Rust also supports the many-readers-one-writer concurrency pattern, where multiple threads can read from the same data concurrently, but only one thread can write at a time. There has been a development of more fine-grained type systems, most notably the Kappa type system used in the programming language Encore [19]. Kappa associates capabilities with their reference based on the type of data access, and the number of threads accessing the reference, giving programmers more flexibility in what their code can safely execute.

This work intends to **create an expressive type system that can statically prevent race conditions in concurrent programming.** This will be achieved through the creation of the programming language Poppy, a minimally viable, object-oriented language with an expressive type system with features [1], and mechanisms adopted from the Kappa type system. Poppy's type system will implement the entire Kappa type system and subsets of Rust's type system. The aim of poppy is to statically prevent data-races in a way that offers more flexibility to the programmer and offers higher degrees concurrency support than the aforementioned type systems.

[1] Features derived from the Bolt language, an implementation and enhancement of Kappa, discussed in Section 3.2

Chapter 2

Preparation

Robust type systems, while often overlooked in purely sequential programming languages, are not only valuable for making a program more secure and reliable, but they help the compiler in generating more efficient, optimised code. Formally, type systems consist of a set of typing judgements that govern the undesirable behaviour a program should *not* exhibit. For example, a `string` type cannot be multiplied and an `int` type cannot be concatenated, and in the case of this project, safe programs cannot contain a potential data race. Three different type systems are presented in Section 2.2-2.4, all of which guarantee the absence of data races: linear type systems, Rust's type system [15], and Kappa's type system. Kappa will be explored further as it forms the basis of Poppy's type system (Section 3.2), drawing specific attention to traits and capability composition, and Rust's influence in Kappa's adaptation of borrowing. Moreover, the aims and objectives of this project will be explored, exploring further how each aim will be fulfilled.

2.1 Type Systems

The fundamental purpose of a type system is to prevent the occurrence of type errors during the compilation of a program. Formally, type systems have been studied in the context of mathematics, particularly the λ-calculus, as a means to reason about types.

Typing judgments in a type system are formal assertions that an expression e conforms to a type τ within a given typing environment Γ. The environment Γ is a context that provides types for free variables within e. By using types, we abstract from actual computation and focus on how object references are utilized. This abstraction is crucial for ensuring that object accesses within a program do not lead to data races.

Inference rules in type systems build upon the typing judgments for sub-expressions to form conclusions about larger expressions.

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma_k \vdash e_k : \tau_k}{\Gamma \vdash e : \tau}$$

When interpreting these rules, if each judgment $\Gamma_i \vdash e_i : \tau_i$ is valid for all i from 1 to k, then the judgment $\Gamma \vdash e : \tau$ is also valid. This structure allows the type system to infer the type of an entire expression from its components, ensuring that the expression can be evaluated without type errors or access violations in a concurrent setting.

We may reason about the attributes of the expressions by abstracting away the expressions' actual values. The outcome of the calculation is not important to us; what matters is how references to objects are employed.

2.2 Linear Type Systems

The core idea of linear types is that for a typing judgement $\Gamma \vdash e : \tau$ to hold, the set of variables in Γ must be exactly the same as the set of variables in e. Linear type systems are seen as a natural progression from linear logic and the λ-calculus in its capability to prohibit or control the management of resources with its 'propositions-as-types' principle. For example, a linear type system can be used to keep track of memory ownership to support region-based memory allocation. This guarantees memory safety without the overhead of garbage collection [4].

Moreover, the use of linear types in programming language design has applications in a cadre of programming languages, most notably and famously, in Rust's type system. Rust eliminates the possibility of data races by eliminating aliasing; which occurs when several references point to the same memory location. Implementing a linear type system is one strategy to achieve this, as it ensures that at any given moment, there's only one reference to any given value. This design means that objects are accessed exclusively by one thread at a time, effectively preventing concurrent access issues.

When values are passed between references in a linear type system, it's done through a destructive read; this means the original reference is invalidated or consumed in the process, no longer pointing to the value. Many routine operations, such as function calls, naturally involve aliasing. The requirement for destructive reads—indicated by a consume operation—in such cases can be cumbersome for programmers. After the consume operation, the value must be reassigned back to the original reference if there's a need to continue using that reference

Listing 2.1: Application of linear types in Poppy

```
fn 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 and Borrowing

The foundation for Rust's notion of ownership is a long history of work, starting with the contributions of linear logic and focusing in particular on attempts to eliminate the overhead of garbage collection in programming languages by Lafont [4], Walder [6], and Baker [7]. Thinking of Rust's ownership model as building on Baker's research on Linear Lisp, where linearity allowed for efficient reuse of objects in memory, is the simplest way to comprehend it [17]. Instead, all memory management computations are handled by the compiler's "borrow checking" feature. Rust's borrow checker handles memory allocation and releases during compilation, in contrast to garbage collection, where it is the programmer's duty, and the program won't compile if any rules are broken [15]. In general, there are three main ownership rules, being:

- **Each value has an owner.** Each value in Rust has a single owner at any time, and the owner is responsible for managing its value's resources.
- **There can only be one owner at a time.** This rule prevents multiple parts of the code from modifying the same data concurrently, which could lead to race conditions or other synchronization issues. When ownership is transferred from one variable to another, the original owner is no longer allowed to access the value. This process is called "moving" the value, and it enforces the exclusive ownership principle.
- **When the owner goes out of scope, the value will be dropped.** This means that the memory and resources associated with the value are automatically cleaned up when the owner is no longer accessible. This automatic resource cleanup eliminates the need for manual memory management or garbage collection, and it ensures that resources are efficiently managed throughout the program's lifetime.

To complement the ownership mechanism, Rust employs a *borrowing* feature, temporarily aliasing a variable. When a reference of a value is borrowed, we can have multiple immutable references to that value concurrently, enforcing the *multiple readers, one writer* concurrency pattern. It distinguishes between aliased and unaliased values through a conservative over approximation of it; a reference would be marked as borrowed if an alias exists in the same lexical scope.

Key takeaways: *Rust's ownership system is linear, references own values, and transferring ownership between references is achieved by a destructive read. Rust reduces the programmer's burden by introducing borrowing, temporarily aliasing a value. Rust's determination of when a reference is borrowed is an overapproximation, and can reject some safe programs.*

2.4 Kappa - Capabilities for Access Control

Kappa [16] employs a rich type system whose uniqueness arises from its use of reference capabilities to prevent data races. Reference capabilities can be thought of as access modifiers in Java or C++ (private, public, protected) that facilitate encapsulation. In Kappa, reference capabilities are introduced via traits, which can be thought of as Java-style interfaces that can name fields and provide implementations of methods. Each trait requires a field, meaning that if the trait is implemented, its field definitions from the trait are carried. Kappa introduces capabilities as traits with modes, where each mode controls how the capability gains exclusive access to the underlying object.

- Linear: the capability must be treated linearly
- Thread-local: the capability can be aliased freely, but aliases are restricted to a single thread
- Lock: interactions with the capability will be wrapped in acquiring and releasing a lock
- Read: the capability only provides reading operations
- Subordinate: the capability is strongly encapsulated inside some object and inherits protection from data-races from it

Kappa augments its type system to track both types as well as capabilities, as a pair of the form (τ, κ), where τ corresponds to a traditional type (int, float, bool, etc) and κ corresponds to the capability held by the reference.

$$\frac{\Gamma_1 \vdash e_1 : (\tau_1, \kappa_1) \qquad \ldots \qquad \Gamma_k \vdash e_k : (\tau_k, \kappa_k)}{\Gamma \vdash e : (\tau, \kappa)}$$

Moreover, all these capabilities prevent the occurrence of data races. The **linear** and **thread-local** modes are exclusive to only a single thread at a time, so concurrent access is impossible. Both **locked** and **read** modes are *thread-safe*; the locked mode locks accesses and the **read** mode will only perform reads on the object. The **subordinate** capability doesn't itself have any protection mechanisms, but it is unique by it being encapsulated in some other capability, called a *dominating capability*. It is used to enforce the object-oriented principle of *encapsulation* in a concurrent context. If the dominating capability is data race free, then so is the encapsulated object.

Key takeaways: *Kappa type system tracks types and capabilities, allowing additional flexibility to programmers in how references should be used.*

2.4.1 Traits and Capability Composition

Kappa enhances access control granularity in comparison to Rust by associating capabilities with *traits* rather than classes. Traits group methods that implement a given behaviour, where classes are formed by multiple traits. Consequently, a reference possessing a certain capability is restricted to interacting only with the corresponding segment of an object – specifically, the fields and methods linked to the trait associated with that capability. This trait-based system presents a novel paradigm for code reuse in object-oriented programming, enabling classes to share functionalities through trait reuse instead of traditional inheritance.

Further, Kappa constructs classes by composing multiple traits using conjunction ⊗ and disjunction ⊕ operators, denoted as `*` and `+` in the example below.

Listing 2.2: Trait composition in Kappa

```
trait LeftSubtree
    ...
    require var leftChild: Tree;
    ...

trait RightSubtree
    ...
    require var rightChild: Tree;
    ...

trait Search
    require var leftChild: Tree;
    require var rightChild: Tree;
    require var val: int;
    def search() {
        // search implementation
    }

class BST = (linear LeftSubtree * linear RightSubtree) + read Search {
    ...
}
```

The operators encode the parallel or sequential access use of capabilities, disjunction meaning sequential use, conjunction meaning parallel use. It is the programmer's duty to determine how capabilities should be used by a class. If A and B are capabilities, the conjunction of A ⊗ B only holds if A and B do not share a mutable state which is not protected by a concurrency control (i.e. Linear). In listing 2.1, capabilities `LeftSubtree` and `RightSubtree` share no mutable state as they access disjoint subtrees, and thus are well formed and can be used concurrently (⊗). On the other hand, the `Search` capability accesses all fields present in `LeftSubtree` and `RightSubtree`, so it cannot be used concurrently (⊕).

Listing 2.3: Unpacking in Kappa

```
class BST = (linear LeftSubtree * linear RightSubtree) + read Search {
    ...
}

main(){
    let tree = new BST();
    let children, jail(search) = unpack tree
    let leftST, rightST = unpack children
    finish{
        async{leftST.insert(4)}
        async{rightST.insert(5)}
    }
    let updatedChildren = pack(leftST, rightST)
    let updatedTree = pack(updatedChildren, search)
}
```

Conjunctions describe well-formed objects constructed in parallel that can be manipulated without any race conditions, and can be unpacked into their constituent parts. Consider listing 2.2, a reference to an object of type BST which has a composite capability of (`LeftSubtree` ⊗ `RightSubtree`) ⊕ `Search`). To use capabilities in parallel, Kappa ensures that a reference holding a composite capability must be decomposed into its constituent capabilities. Kappa ensures the correct decomposition by introducing the `unpack`, `pack`, and `jail`, keywords. The `unpack` keyword destructively reads the original reference and returns its sub-capabilities. To prevent some disjunction capabilities κ_1, κ_2 from being used in parallel, the `jail` keyword indicates which sub-capability should not be used. Listing 2.2 decomposes the BST object to concurrently access its capabilities. It unpacks the object by jailing the search capability (since Search is a disjunction capability) and then unpacks the children to access `leftST` and `rightST`, which are conjunction capabilities. They are now aliases of the same object, and mutating them in parallel is safe. The mutated objects are then packed back into their original reference.

Key takeaways: *Kappa uses capability composition through disjunction and conjunction operators to reason about how capabilities can be used. It introduces the pack, unpack, and jail operators to reason about how capabilities can be decomposed and recomposed.*

2.4.2 Function Borrowing

As discussed in Section 2.2, handling linear values typically involves the overhead of destructively reading and reinstating them for function calls. Kappa introduces a more efficient approach by allowing linear values to be temporarily borrowed within the scope of a function. The borrowed reference is treated as linear, and the original reference becomes unusable, acting as a destructive

read. Once the function execution is complete, the original reference becomes usable again, reinstating it. Kappa also ensures the safety of this approach by not allowing borrowed values to be assigned, preventing them from outlasting their intended scope. This approach differs from Rust's borrowing system, outlined in Section 2.3, whereby Rust restricts borrowing to the same scope, resulting in aliases being read-only.

Further, Kappa introduces a more fine-grained control of resource borrowing by introducing two methods of borrowing, forward borrowing and reverse borrowing. *Forward borrowing* occurs when a linear value is temporarily passed to another function or method, allowing it to be used within a different context without transferring its ownership permanently. Consider Listing 2.3, `foo` takes a pair `p`, and unpacks it into a linear `Fst` and an aliasable `Snd` without destructively consuming `p`. The value of `p` is temporarily used in `foo` but remains intact in its original context. *Reverse borrowing* is a bit more complex and is used when the path of linear values is longer than a singleton variable. It allows for non-destructive reads of linear fields into stack-bound values. This enables the safe extraction of a linear value's component without consuming the entire value. Consider Listing 2.4, the `get` method in the `Accessor` trait performs reverse borrowing on `elem`. It returns `elem` without destroying the containing object, thus allowing access to `elem` while preserving the integrity of the whole structure.

Listing 2.4: Forward Borrowing

```
1 def foo(p: S(linear Fst * thread Snd)):
      void
2     ...
3 foo(p)
```

Listing 2.5: Reverse Borrowing

```
1 read trait Accessor<linear T>
2 require elem: T
3 def get(): S(T)
4     return this.elem
```

Key takeaways: *Kappa introduces forward and reverse borrowing to reduce programmer overhead by removing the need to destructively read and reinstate linear values, avoiding unnecessary memory writes.*

2.5 Aims and Objectives

The aims and objectives are designed to guide the implementation of this project. It is split into three sections, language design, type system design, and compiler design. Each section is comprised of aims, and a set of objectives to achieve the aim. The aims and objectives (derived from the project summary) will then be evaluated in Chapter 4.

2.5.1 Language Design

The design of Poppy forms the basis of this project. The aim here is to craft a syntactic and semantic design that is inspired by effective principles from Rust, but still unique in its own right. Given that Poppy will be designed without classes, a great emphasis will be put on designing how structs, traits, and methods will work in an object-oriented context. The goal is to design a language that is not only safe, concurrent, and free from data races, but also intuitive to use. This entails defining a clear set of grammar rules, intuitive syntax, and semantic constructs that map well to the problem space that Poppy aims to address. The language design phase will involve creating a language specification that comprehensively describes Poppy's syntax and semantics. The document will serve as a crucial reference point for both the compiler design and the language's users (Appendix A).
The following objectives are in place to achieve this defined aim:

- Gather the necessary requirements to carry out this project within the specified time frame. This will involve understanding the time required and technical complexities of each component in the project and planning accordingly.
- Analyse and determine Poppy's concurrency model. This can be done by analysing existing programming languages that adopt these different models, and determine which one best fits the needs of this aim.
- Determining the extent of Poppy's object-oriented features. This will involve understanding the benefits and drawbacks of a fully object-oriented language, which will help in determining the extent of Poppy's feature set.
- Develop a concise yet expressive syntax that can support all proposed language features. Design an EBNF grammar that can support all features in Poppy.
- Develop Poppy's semantics, which will dictate the behaviour of a Poppy program. The semantics should emphasise memory safety and concurrency.

2.5.2 Type system design

The design of Poppy's type system is integral to the prevention of data races and ensuring the safety of concurrent programming. This will involve a thorough exploration and understanding of the Kappa type system, which is known for its ability to prevent data races. Moreover, key concepts from Rust such as ownership and borrowing will be deeply analyzed and incorporated into Poppy's type system. Attention will also be given to the creation of clear and concise type rules that make the language predictable and easy to use. Careful consideration will be given to how structs, traits, and methods interact with the type system in an object-oriented context, and any language-specific rules will be enforced to guarantee the language's safety.
The following objectives are in place to achieve this defined aim:

- Understand the Kappa type system. The Kappa type system will be the main source of inspiration for the creation of Poppy's type system, so a deep understanding of its motivations, features, and guarantees will be essential.
- Analyse Rust's ownership and borrowing concepts. These concepts will be core in Poppy's memory model, removing the need for garbage collection and providing memory safety. Understanding how ownership and borrowing work at the fundamental and the practical level is necessary for implementing it in Poppy.
- Develop an explainable, detailed, and efficient typing environment. The typing environment (or symbol table) is the central data structure in the type system and is where user declarations like type definitions, and function definitions are stored.
- Type check Poppy programs. This stage will involve type checking every facet of what defines a Poppy program such as struct, interface, and function definitions, expressions, statements, calls, control flow, etc.
- Verify the type system. Make extensive use of unit tests and end-to-end tests. These two types of tests can reason about different facets of the type system. The former will verify the mechanical functionality of the code, and the latter will verify if the type system is checking the input program correctly.

2.5.3 Compiler Design

The compiler design phase will involve the creation of a compiler for the Poppy language, ensuring that programs written in Poppy can be correctly and efficiently converted into an intermediate representation that can be used by LLVM. This involves designing the architecture and stages of the compiler, including the lexer, parser, type checker, and optimisation processes. The compiler frontend will be developed using OCaml and will be responsible for parsing the source code, conducting semantic analysis, and generating an intermediate representation (IR). This IR will then be optimized and translated to machine code with the help of LLVM as the middle-end and back-end of the compiler. This modular design enables each part of the compiler to be developed and tested separately, thus improving the maintainability and extensibility of the compiler. Emphasis will be placed on efficiency and accuracy during the compiler design and development process.
The following objectives are in place to achieve this defined aim:

- Develop a compiler architecture. Understand the intricacies of compiler development and how the frontend can be implemented efficiently using OCaml, and understand how the compiler frontend is linked to the LLVM toolchain.
- Develop the lexer. With the help of OCamlLex, develop a lexer that correctly implements the EBNF grammar defined in the language design objective.
- Design and implement the parser. Using the Menhir parser generator, develop a parser that implements Poppy's semantics defined in the language design objective.
- Design error handling and definitive error messages. Develop a system that decides how errors should be handled, and when an error is encountered, output verbose error messages.
- Implement various semantic analyses such as type checking and data-race checking. The core implementation will involve implementing the type system defined in the previous aim to the compiler. Part of the type checker will include the data-race checker, a unique feature of Poppy that will handle concurrency-based type checking.
- Generate LLVM intermediate representation. Understand the OCaml bindings available with the LLVM toolchain, and how they can be utilised to generate LLVM IR.
- Integrate LLVM IR to the LLVM middle-end and backend. This involves how to lower the LLVM IR to the LLVM middle-end for it to be optimised and then compiled to target code. A solid understanding of the LLVM toolchain is necessary to implement this.
- Verify the compiler. Similar to verification in the previous aim's objective, employ unit tests and end-to-end tests to check mechanical functionality of the compiler, and to test the language as a whole.

2.6 MoSCoW Analysis

The MoSCoW analysis (or prioritisation) method has been used to manage the objectives defined in Section 2.5. The analysis serves to prioritise tasks that have an immediate benefit to the project, creating a clearer scope of language features.

MoSCoW Category	**Language Features**
Must-have	Fork-join parallelism, capabilities, functions, variable assignment
Should-have	Structs, traits, methods, control flow
Could-have	Inheritance, polymorphism, generics
Won't-have	Garbage collection, global state, macros

Table 2.1: MoSCoW Prioritisation

Chapter 3

Methods and Implementation

The implementation of Poppy can be split into two sections; the first is the theoretical implementation, and the second is the practical implementation. The theoretical implementation will detail the language design decisions and changes made to Kappa type system. It will detail the implementation of the capability inference algorithm, and the liveness algorithm presented in Bolt's type system [21]. The practical implementation will detail how Poppy was built, detailing the separate phases in the compiler pipeline.

3.1 Language Design

Poppy is a concurrent object-oriented language that demonstrates how ideas present in Kappa can be applied to a language not bound by a theoretical calculus. Poppy moves away from Kappa by removing the `pack`, `unpack`, and `jail` operators presented in 2.4.1 which are in place merely to simplify Kappa's theoretical calculus, and to move the onus to Poppy's type system to handle capability composition and decomposition.

Poppy's syntax deviates from the syntax presented in Kappa by adopting Rust's effective structural principles. The use of structs, traits, and implementations make capability annotations simple, intuitive, and at declaration-site.

Listing 3.1: Poppy's structural syntax

```
struct Foo {
  capability read A, local B, linear C;
  ...
}
trait Baz {
    methodA : A () -> Foo
    methodB : B () -> Foo
    methodC : C () -> Foo

impl Baz for Foo {
    A : A () -> Foo { ... }
    B : B () -> Foo { ... }
    C : C () -> Foo { ... }
}

void main() { ... }
```

Additionally, Poppy introduces a consume operation that destructively reads from a linear variable, making it possible for programmers not to have to manually manage capability composition and decomposition. This `consume` operation makes it easy to transfer ownership and control of linear variables without the need for explicit memory management. Listing 3.2 shows when `consume x` is called, it transfers ownership of x to y via a destructive read. This design simplifies the handling of linear resources without the risk of resource leaks or unintended side-effects from a shared mutable state.

Listing 3.2: Consuming linear variables

```
struct Foo {
    capability linear Bar, read Baz;
    const int f : Bar, Baz;
    const int g : Bar ;
    const int h : Bar;
}

void main(){
    new x = Foo{f:4, g:5, h:6};
    let z = x;
    z.f; // z's liveness ends here
    let y = consume x // Consume linear variable
}
```

Finally, Poppy introduces a structured fork-join parallelism via the finish-async construct. In Poppy's fork-join construct, 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. Unlike shared state concurrency (mutexes) and message passing present in Rust, Poppy's finish-async construct makes it easier to reason about the lifetimes of threads.

Listing 3.3: Java

```
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?
}
```

Listing 3.4: Poppy

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

3.2 Modifications to Kappa

This section will detail the modifications made to Kappa which make up Poppy's data-race checker. It will also explain the *liveness analysis* and *capability inference* algorithms present in Bolt's type system [21] that have been adapted to construct Poppy's data-race checking system.

3.2.1 Capability annotations

In Kappa, object fields are accessed through capabilities restricted by traits. On the other hand, Poppy allows for a more direct approach, where fields are annotated with the capabilities that can be accessed. This means Poppy programs can use multiple capabilities for a single implementation, offering more access control flexibility.

Additionally, Poppy enables programmers to annotate function or method parameters with specific capabilities. When there's a lack of annotations, it's assumed that all capabilities of a parameter are utilized, which can be restrictive if the function's implementation isn't locally available. By specifying only the necessary capabilities in annotations, Poppy allows for more *fine-grained control* over concurrency. These annotations are particularly useful in large codebases, as they provide clear insights into the capabilities employed by a function or method, simply by looking at its type signature.

Listing 3.5: No function parameter annotations

```
fn printLeft(x:BST) -> void {
    // We could be using x's Left,
    // Right & Search capabilities
    ...
}
fn printRight(x:BST) -> void {}
finish{
    // functions could be using the
        same capabilities, which is not
        allowed
    async{
        printLeft(x)
    }
    printRight(x)
}
```

Listing 3.6: Function parameter annotations

```
fn printLeft(x:BST{left}) -> void {
    // We know we aren't using x's
    // Right & Search capabilities
    ...
}
fn printRight(x:BST{Right}) -> void {}
finish{
    // Left and Right can be used
        concurrently since explicitly
        defined.
    async{
        printLeft(x)
    }
    printRight(x)
}
```

Key takeaway: *Poppy moves away from requiring capabilities to be bound by the trait it is declared in, towards being able to directly annotate methods' fields with capabilities that are being used. This, in contrast with Kappa, allows for multiple capabilities to be annotated for every method. Although it invites a small annotation overhead, programmers can locally reason about the capability of a method by inspecting its type signature.*

3.2.2 Type checking linear capabilities

Kappa restricts a reference with a linear capability to only be destructively read and not directly accessed. The Bolt type system introduces a **liveness analysis** algorithm, similar to Rust's borrowing model, assessing the activity of any aliases of a reference at a given point in the program. Should there be any active aliases, the reference's linear capability is relinquished.

To do so, Bolt uses abstract interpretation to monitor aliasing, a static analysis technique that involves simulating a program's execution to track certain properties using the language's semantics. Consider an expression like `let x = e`. if `e` could be simplified to some identifier `y`, the the expression would effectively become `x = y`, leading to the possibility of `x` aliasing `y`. For programs

with branching paths, such as those containing if-else statements, this analysis could result in a collection of possible aliases. It's also important to consider *transitive aliasing:* if x aliases y and z later aliases x, then z also aliases y. To establish an upper limit on all potential aliases, the algorithm is executed iteratively from each immediate alias such as x, incorporating x's aliases into the overall set, and this process continues until no new aliases are identified, reaching a fixed-point. This approach yields an upper bound because the actual execution paths of the program are not determined—since we don't run the program, we can't be certain which branches will be taken. For the sake of ensuring soundness, we must make an overapproximation.

Since Kappa's typing judgments do not consider the sequence of operations (they are flow-insensitive), they cannot guard against bugs related to dangling references, which occur when a programmer tries to use a reference after it has been destructively read. To address this, abstract interpretation can be employed to keep track of the references that have been destructively read as the program runs. If a programmer attempts to access a destructively read reference later in the program's execution without it being reassigned, the compiler can issue a warning. This method provides a dynamic check that supplements the static typing system, enhancing the reliability of the code by preventing the use of invalid references.

Key takeaway: *Poppy allows for references to linear objects to be temporarily aliased instead of destructively read. Akin to Rust's borrow checker, Poppy uses a liveness analysis algorithm to determine the lifetimes of references and manage their capabilities.*

3.2.3 Capability Inference

Kappa, as described in section 2.4, mandates that programmers explicitly define the combination of capabilities within a class and utilize `unpack`, `pack`, and `jail` operations to break down references into their constituent sub-capabilities. This requirement facilitates the type checker's validation of both the capability composition and the correctness of `unpack`, `pack`, and `jail` operations. Yet, this approach primarily serves to streamline the type checking process. In contrast, Poppy simplifies this process by adopting Bolt's **capability inference** algorithm, requiring programmers to only declare an object's *potential* capabilities. The algorithm then deduces which capabilities are actively employed by references and identifies those that may be utilized concurrently (as shown in Listing 3.8).

Listing 3.7: Kappa

```
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);
```

Listing 3.8: Poppy

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

The method for inferring used capabilities involves a constraint-based analysis. Initially, each reference instance is associated with a set of all capabilities it might access. The process then involves eliminating inapplicable capabilities, rather than tracking usable ones. This elimination is based on the typing judgments. Any capability whose mode's typing judgment is violated gets removed from the reference's potential capability set, leaving behind a set of safely usable capabilities.

The next step involves ensuring that the reference utilizes only capabilities from this vetted set. This verification is done when the reference is used to access fields or invoke methods. The analysis also extends to functions or methods receiving this reference as a parameter, ensuring they do not employ disallowed capabilities. In scenarios where a reference might utilize multiple capabilities for a single field access, the system opts for the least restrictive capability mode. For instance, a *locked* capability would be preferred over a *linear* one due to its concurrent usability, as opposed to the single-thread limitation of a linear capability.

So far, the process has involved deducing safe capability sets for references and scrutinizing the capabilities each field access, method, and function call employs. However, there remains a need to validate the safety of concurrent capability uses. This involves first identifying references shared across multiple threads. For each shared reference, we accumulate the capabilities employed in each thread. Subsequently, Kappa's typing judgments are enforced by comparing capabilities used in different threads to ensure they can safely be used concurrently.

Key takeaway: *Poppy adopts a capability inference algorithm to infer the capabilities of a reference. This redirects the manual composition and decomposition from the programmer to Poppy's type system.*

3.3 Compiler Implementation

A standard compiler workflow involves taking the source code of a program, conducting a type-checking process to ensure that the code adheres to the language's type rules, and then transforming the code into an intermediate representation (IR), see Figure 3.1.

The implementation of Poppy can be represented as a 2-dimensional grid where the columns represent language features such as structs, traits, function and method definitions, and expressions, and the rows represent the compiler phases such as lexical analysis, parsing, type checking, and intermediate representation. Implementing the complete compiler means each record in the grid must be completed. The intended implementation of Poppy was done using a **waterfall** approach [18], where each stage of the compiler is built one after the other: the complete lexer, the complete parser, the complete type checker, the complete intermediate representation, etc. The main advantage of this method is finding bugs in the code – since the compiler pipeline is shallower, it limits the scope of where bugs can occur and can potentially make debugging easier. However, the downside of this method is testing specific language features and getting to the first "hello world" as this can only occur once code generation and LLVM integration have been complete. This method will succeed in unit testing mechanical components of compiler components but lacks in utilising integration and end-to-end tests early on. However, the reality of development pressures and time restrictions meant that the Poppy compiler was written with a more informal waterfall-ish strategy, omitting some harder-to-implement features and prioritising others.

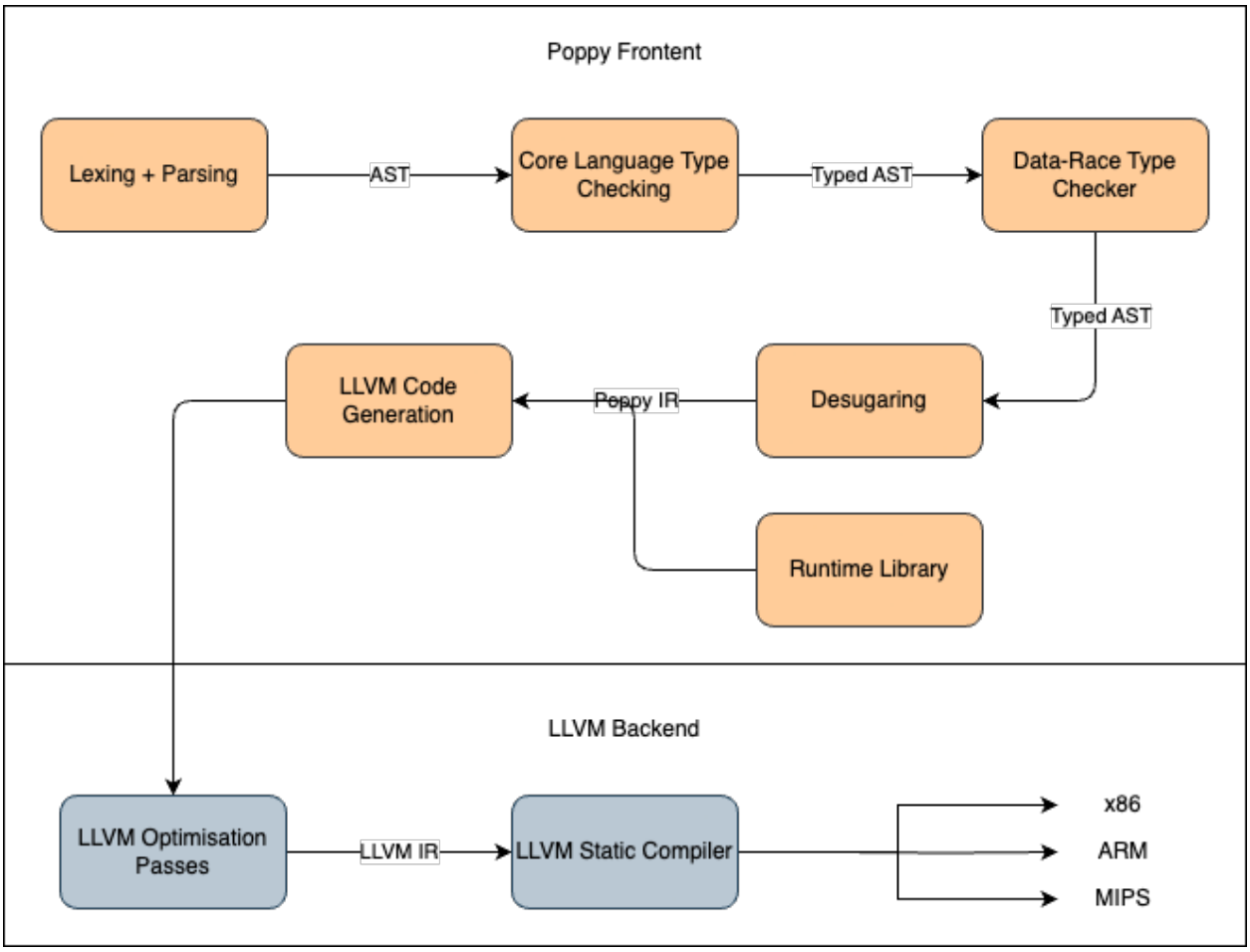
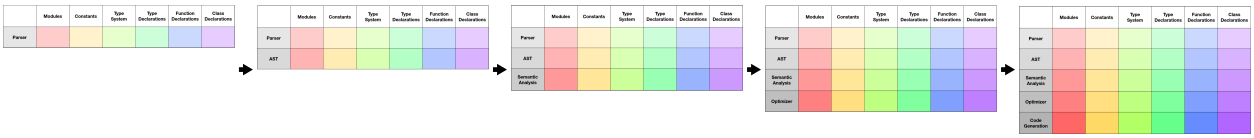


Figure 3.1: Poppy Pipeline



Poppy's compiler is designed with a front end that translates programs into a simplified representation (Poppy IR) that is closely aligned with LLVM IR. LLVM IR is widely used in the industry due to its versatility; it is compatible with multiple assembly languages and offers built-in optimizations to enhance the performance of the final, compiled native code. Additionally, the back end is responsible for integrating necessary runtime libraries during the compilation to native code.

The type system in Poppy incorporates both conventional types (`int`, `bool`, `etc`), and more sophisticated constructs like *capabilities*. To manage this complexity, the type-checking process is divided into two distinct phases. The first phase involves type-checking the conventional types. The second phase involves type-checking capabilities. The approach of dividing the type checking into two distinct phases allows for a more manageable and modular type-checking process, where the intricacies of capability-based type-checking are handled separately.

3.3.1 Repository overview

Poppy repository: `https://github.com/os12345678/Poppy`
The repository is structured in a modular manner, where Poppy's core logic is in the `lib/` folder, grouped into libraries, corresponding to each stage of the compiler pipeline. Poppy's runtime library is located in the `core_lib/` folder, which contains a collection of external `C` functions that have been converted to `.ll` files (via Clang) that will then be used in the code generation phase to build the LLVM module.

Folder	**Description**	**Lines of Code** [1]
lib/parsing	Contains lexing and parsing code + type information relevant to all AST's	732
lib/type_checking	Contains core language type checker + type augmented AST	1196
lib/data_race_checker	Contains data-race checking module	2190
lib/desugar	Contains desugaring code + desugared AST	356
lib/llvm_codegen	Contains code generation logic + LLVM specific symbol table + linking runtime functions to LLVM module	816
core_lib	Contains core library functions, written in C, and their LLVM IR implementation.	123
tests	Contains a range of tests written with ppx_expect for all frontend modules of the compiler.	299

Table 3.1: Description of folders and their contents

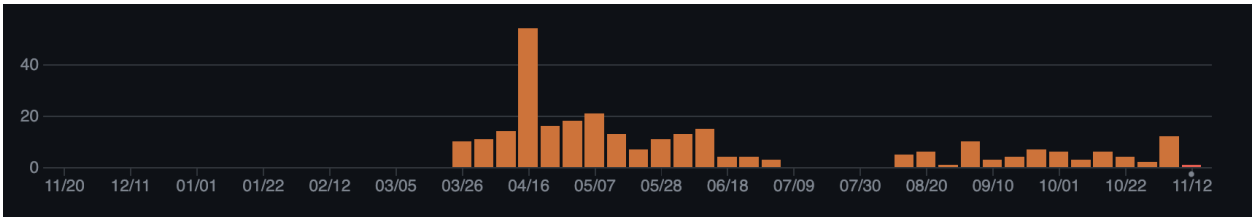
3.3.2 Language and Build System

Poppy has been developed using OCaml, an industrial-strength programming language supporting functional, imperative, and object-oriented styles. Its rich type system offers strong type checking whilst minimizing verbosity through its exhaustive pattern-matching capabilities. One of its main uses is in compiler design, specifically through its great tooling for lexing and parsing, and its native support for the LLVM OCaml API [24]. The choice to utilise LLVM for Poppy's middle-end and backend has its benefits. LLVM is a set of modular and reusable compiler frameworks which is platform independent. Its intermediate representation is high-level, allowing for easy analysis and optimisation. OCaml's Dune build system for the entirety of this project.

3.3.3 Version Control and Project Commits

Git version control is used to keep track of project changes. Each branch made was a phase in the compiler pipeline where it was merged back into the main branch once tested. The project consisted of over 340 commits.

[1] Lines of Code was calculated using the `cloc` library.



3.3.4 Compiler Pipeline

Parsing – Parsing is usually broken down into two two parts, lexical analysis and parsing. Lexical analysis was implemented using `ocamllex` [23], where a stream of characters (source code) is converted into a stream of logical tokens, which are derived from regular expressions. On the other hand, parsing was implemented using `Menhir` [22], which constructs parsing rules based on the context free grammar, which is represented in the form of an *abstract syntax tree* (AST). See Appendix A for Poppy's full grammar definition.

Listing 3.9: lib/parser/lexer.mll

```
(* Helper Regexes *)
let digit = ['0'-'9']
let alpha = ['a'-'z' 'A'-'Z']

let int = digit+
let id = (alpha) (alpha|digit|'_')*

let whitespace = [' ' '\t']+
let newline = '\r' | '\n' | "\r\n"

(* Rules *)
rule read_tok =
  parse
  | "(" { LPAREN }
  | ")" { RPAREN }
  | "{" { LBRACE }
  ...
```

Listing 3.10: lib/parser/parser.mly

```
program:
    | struct_defns=list(struct_defn);
    trait_defns=list(trait_defn);
    impl_defns=list(impl_defn);
    function_defns=list(function_defn);
    main=main_expr; EOF
    {Prog(struct_defns, trait_defns,
        impl_defns, function_defns,
        main)}

// Productions related to struct and
    interface definitions
struct_defn:
    | STRUCT; name=ID;
    LBRACE;
    capability=capability_defn;
    field_defns=nonempty_list(field_defn);
    ...
```

The AST's in Poppy are represented as OCaml algebraic data types, corresponding to the constitute parts that make up a Poppy program (i.e. structs, traits, methods, functions, expressions etc). Expressions in Poppy are built as a record type, which stores additional information such as location, node information, and in later AST's, type information.

Listing 3.11: lib/parser/AST.ml

```
  type expr = {
  loc : loc;
  node: expr_node
}
[@@deriving sexp]

and expr_node =
| TInt              of int
| TBoolean          of bool
```

```
10 | TIdentifier        of identifier
11 | TLet               of type_expr option * Var_name.t * expr
```

The expression record not only carries the syntactic information about the node, but also metadata for error messages, and later on, types for type checking. `loc` refers to the line number and position of the expression, crucial information when printing error messages and `node` refers to the actual node information. One main advantage of storing expressions as a record type is that it can be easily augmented to fit the needs of different AST's. For example, the process of augmenting this AST to a typed AST requires us to just add a "type" field in the `expr` type, augmenting all nodes in the AST with type information.

Key takeaway: *Lexing and parsing have been simplified using OCaml's parser generators Menhir and OCamlLex. These tools simplify grammar production, as well as generating parsing rules. OCaml's algebraic data types stand as the foundation of its abstract syntax tree, making it possible to build complex types that correspond to Poppy's language.*

Core Type Checker The core type checker infers conventional types of expressions and populates that node's type with it. Poppy expressions have either a primitive type (`int`, `bool`, or `void`) or are objects belonging to some struct i.e. `Struct Foo`.

$$\frac{\Gamma \vdash e_1 : \text{int}, \varnothing \qquad \Gamma \vdash e_2 : \text{int}, \varnothing}{\Gamma \vdash e_1 + e_2 : \text{int}, \varnothing}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}, \varnothing \qquad \Gamma \vdash e_2 : \tau, \kappa \qquad \Gamma \vdash e_3 : \tau, \kappa}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau, \kappa}$$

Additionally, in preparation for data-race type-checking stage, each identifier node is annotated with the set of possible capabilities it can use. This is used in the constraint-based analysis of 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.

Listing 3.12: lib/type-checking/typed-ast.ml

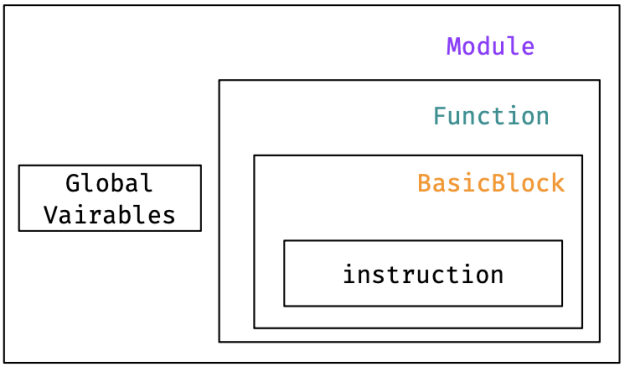
```
1  and expr_node =
2  | TInt               of int
3  | TBoolean           of bool
4  | TIdentifier        of typed_identifier
5  | TLet               of type_expr option * Var_name.t * expr
6  | TBlockExpr         of block_expr (* used to interconvert with block expr *)
7  | TAssign            of typed_identifier * expr
8  | TConstructor       of Struct_name.t * constructor_arg list
9  | TConsume           of typed_identifier
10 | TMethodApp         of Var_name.t * Struct_name.t * Trait_name.t * Method_name.t *
       capability list * expr list
11 | TFunctionApp       of Function_name.t * expr list
12 | TIf                of expr * block_expr * block_expr
13 | TWhile             of expr * block_expr
14 | TPrintf            of string * expr list
15 | TBinOp             of bin_op * expr * expr
```

```
| TUnOp              of un_op * expr
| TFinishAsync       of async_expr list * obj_var_and_capabilities list * block_expr
```

Key takeaway: *Poppys core type checker governs the semantics of primitive type rules. The language representation is augmented with additional type information, that will be used by the data-race type checker.*

Data-Race Type Checker This stage implements the modifications to Kappa, presented in Section 3.2.

Desugaring Having performed the core language and data-race type checking, we can begin to desugar our language into a simpler interpretation, which we can call Poppy IR. We drop all information that is not necessary at runtime. The aim of this is to make Poppy IR as conceptually similar to LLVM IR as possible, to make the translation between the two easy.



In LLVM IR, modules contain functions and global variables. Functions contain basic blocks and arguments, which correspond to functions. Basic blocks contain a list of instructions where each block is a contiguous chunk of instructions. Poppy IR is built similarly, where Rust-like method implementations are desugared into mangled functions, and expressions are desugared into more atomic instructions. For example, method and function applications are desugared into mangled call instructions, and for loops are desugared into while loops. The `finish-async` construct is also just syntactic sugar for spawning threads, waiting, and joining them. As a result, the `finish-async` blocks are desugared into two instructions, `threadCreate` and `threadJoin` respectively, derived from the C POSIX thread (pthread) library.

Listing 3.13: lib/desugaring/desugared-ast.ml

```
    and dexpr_node =
| DIntLit            of int
| DBoolLit           of bool
| DStringLit         of string
| DBlockExpr         of dexpr_node list
| DVar               of string
| DAssign            of string * dexpr_node
| DBinOp             of T.bin_op * dexpr_node * dexpr_node
| DUnOp              of T.un_op * dexpr_node
| DCall              of string * dexpr_node list
| DIf                of dexpr_node * dblock * dblock
| DWhile             of dexpr_node * dblock
| DCreateThread      of string * dexpr_node list (* thread id + argument *)
```

```
| DJoinThread        of dexpr_node (* thread identifier *)
```

Key takeaway: *Poppy has been desugared into a simpler representation of information (Poppy IR), whereby objects have been desugared into structs, methods into functions, applications into calls, identifiers into variables, and fork-join parallelism into atomic thread operations. This language lowering makes it much easier to translate into our target, LLVM IR.*

LLVM Code Generation Having lowered Poppy into Poppy IR, it can be translated into LLVM IR. LLVM IR is a more granular representation of code that is closer to machine code, making use of various efficient optimisations. One main use of translating to LLVM IR is that it is platform independent, meaning it can be translated to any target specific machine architecture.

Listing 3.14: Poppy Fibonacci

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

void main() {
    fib(10)
}
```

Listing 3.15: LLVM IR Fibonacci

```
define i32 @fib(i32 %0) {
entry:
  %eqtmp = icmp eq i32 %0, 0
  %eqtmp1 = icmp eq i32 %0, 1
  %ortmp = or i1 %eqtmp, %eqtmp1
  br i1 %ortmp, label %then, label %else

then:

    ; preds = %entry
  br label %ifcont

else:

    ; preds = %entry
  %subtmp = sub i32 %0, -1
  %fib = call i32 @fib(i32 %subtmp)
  %subtmp2 = sub i32 %0, 2
  %fib3 = call i32 @fib(i32 %subtmp2)
  %addtmp = add i32 %fib, %fib3
  br label %ifcont

ifcont:

    ; preds = %else, %then
  ret i32 0
}

define void @main() {
entry:
  %fib = call i32 @fib(i32 10)
  ret void
}
```

One of the key advantages of this phase is the utilisation of OCaml's LLVM API [24]. The API provides bindings to most native LLVM functions, enabling direct manipulation of LLVM constructs within the OCaml environment. The process involves several key steps:

- **Module and Function Setup**: Mentioned in the desugaring section, the structure of LLVM

IR is organized into modules. Each module in LLVM corresponds to a source file in Poppy IR. Inside these modules are functions. The module contains declarations for Poppy's runtime library, function definitions, and struct definitions.

```
let context = global_context ()
let the_module = create_module context "Poppy JIT"
let builder = builder context
...
declare_externals the_module;
(* Adds llvm ir runtime library to the module *)
...
lookup_function "create_thread" the_module
(* Function retrieval in the module *)
```

- **Basic Block Creation**: Each function in LLVM IR is composed of basic blocks. A basic block in LLVM is a linear sequence of instructions that has one entry point and one exit point. The desugared expressions from Poppy IR, such as control flow constructs and operations, are mapped to these basic blocks. Listing 3.15 shows the basic blocks present inside the recursive `fib` function.

- **Instruction Translation**: The individual expressions in Poppy IR are translated into corresponding LLVM instructions. Each instruction in LLVM IR is a low-level operation, closely resembling machine instructions. The process of translating expressions into instructions requires a deep understanding of low-level control.

Key takeaway: *Poppy IR has been translated to LLVM IR, a lower-level representation that is closer to machine code. This transition is crucial for leveraging LLVM's optimizations and achieving efficient executable code.*

Language Runtime A runtime library has been created to extend functionality not present in the LLVM OCaml API. This runtime library is written in C and compiled into LLVM IR, which are then declared as functions in the LLVM module. The runtime library primarily consists of lower-level functionality such as memory management and multi-threading.

Firstly, LLVM OCaml API only natively supports stack-based memory allocation, suitable for local, short-lived variables. However, it doesn't natively support dynamic, heap-based memory allocation, where the lifespan of an object extends further than its local scope. To facilitate this, the `malloc` function was added to Poppy's language runtime, being able to dynamically allocate memory to the heap, especially useful in creating struct-like objects. Another area that LLVM doesn't natively support are threads and thread operations since they involve a complex routine of instructions that are platform specific. Instead, Poppy incorporates POSIX threads (pthreads) that are a standardised API for thread system calls. The two main thread functions incorporated have been `pthread_create` and `pthread_join`. These two functions enable the creation and management of multiple threads. Below is a more complete view and description of each function created as part of Poppy's runtime library.

Table 3.2: Runtime Library function names and definitions

Function	**Description***
GC_Malloc	The function GC_malloc is a wrapper for the malloc function in C. The size parameter is the number of bytes to allocate for the memory block. @return a pointer to the memory block that has been allocated using the malloc function.
thread_create	The function 'thread_create' creates a new thread using the pthread library in C. A pointer to a pthread_t variable where the thread ID will be stored after the thread is created. The start_routine is a pointer to the function that will be executed by the new thread. It takes a single void pointer argument and returns a void pointer. This function will be called when the new thread is created. The 'arg' parameter is a void pointer that can be used to pass any additional arguments to the thread's start routine. The start routine is the function that will be executed by the thread when it starts.
thread_join	The function 'thread_join' is a wrapper for the 'pthread_join' function in C. The 'thread' parameter is of type pthread_t and represents the thread to be joined. It is the identifier of the thread that you want to wait for. The value_ptr parameter is a pointer to a location where the exit status of the joined thread will be stored.
pthread_equal_wrapper	The function 'pthread_equal_wrapper' is a wrapper function that calls the 'pthread_equal' function and returns its result. The first parameter, t1, is of type pthread_t. It represents the first thread identifier that we want to compare. The parameter 't2' is of type 'pthread_t', which is a data type used to represent a thread identifier in the POSIX threads library. @return the result of the pthread_equal function, which is an integer value indicating whether the two pthread_t variables t1 and t2 are equal.
pthread_self_wrapper	The function 'pthread_self_wrapper' returns the thread ID of the calling thread. @return The function 'pthread_self_wrapper' returns the thread ID of the calling thread.
print	The function 'print' takes a format string and variable arguments, and prints the formatted output to the console. The format parameter is a string that specifies the format of the output. It can contain format specifiers that are replaced by the values specified in the subsequent arguments. @return the value of 'retval', which is the return value of the 'vprintf' function.

* The descriptions have been generated by Doxygen Documentation Generator.

Key takeaway: *Poppy's runtime library effectively extends the functionalities not natively supported by the LLVM OCaml API, particularly in memory management and multi-threading.*

3.4 Verification

Verification is a crucial objective in this project to ensure that Poppy performs as expected. It not only provides confidence in the language's robustness but also validates the design choices made during its creation. Verification will be done through a well-designed suite of test programs. This test suite will contain programs with and without data races, allowing for a comprehensive evaluation of Poppy's ability to prevent data races. A rigorous testing process will be in place, which includes unit tests for individual components, integration tests for the entire system, and end-to-end tests for the real-world usability of the language. The success of these tests will be a major determining factor in assessing the correctness of Poppy and its ability to fulfill the project's aims. Testing specifically for data races can include testing the compiler with programs that contain data races, and ensuring the compiler correctly handles these cases. Jane Street's `ppx_expect` library [20] was used for verifying expected output vs actual output. The testing folder contains regression tests for each pipeline component (i.e. parsing, type checking, desugaring, etc) to ensure the compiler doesn't regress from its expected outputs after each pipeline component. This falls nicely into place with the *waterfall implementation* strategy defined in Section 3.3. These tests are shown in Appendix C, showing Poppy's ability to statically prevent data races.

3.5 Summary

Overall, the implementation of Poppy has been an overwhelming success. The adoption and modification of Kappa in Poppy's type system showed how it can statically prevent data races. Using key features from the Bolt language, there have been three major changes to Kappa's type system. Firstly, it was shown how to reduce programmers' overhead by implementing the capability inference algorithm to infer the available capabilities an object can hold. This makes capability composition and decomposition much easier than presented in the Kappa's theoretical calculus. Next, Poppy allows for references to linear objects be temporarily aliased (borrowed) instead of destructively read. Like Rust, allowing for efficient reuse of objects in memory. Lastly, Poppy allows for multiple capabilities for every method through annotations, diverging from Kappa whereby a method is bound to its trait and its capabilities.

The practical implementation detailed the steps involved in creating the Poppy compiler, including implementation methodology, a repository overview, and the compiler pipeline. The implementation of Poppy was discussed from lexing and parsing, through to LLVM IR code generation. The section gives an insight into how the project was structured and the methodology that was used to create it.

Link to source code: `https://github.com/os12345678/Poppy/tree/main`

Chapter 4

Evaluation

This project aims to implement a minimally viable, object-oriented language that incorporates a smart, flexible static type system to ensure the prevention of data-races. Evaluating a research project is pivotal in assessing success and to identify areas for improvement.

Firstly, the research and development of Poppy has achieved most success criteria defined in the aims and objectives, which will be explored in Section 4.1. Secondly, the design of the language is syntactically similar to Java and structurally similar to Rust, so in Section 4.2, Poppy will be transliterated against Rust and Java programs to show its design effectiveness and shortcomings. Finally, Poppy will also be evaluated against its concurrency expressivity against Rust and Java, illustrating how its concurrency is more fine-grained than both others.

4.1 Review of aims and objectives

In retrospect, the aims and objectives defined in Section 2.5, and the MoSCoW analysis in 2.7 have been mostly a success. All core deliverables from the project proposal in Section 1.1 have been achieved with the exception of some nice-to-have extensions. Below is a discussion of how each aim has been fulfilled:

Design of a minimally viable object-oriented programming language which implements fork-join parallelism. The core of this aim has been fulfilled. Poppy is a minimally viable, object oriented language that supports fork-join parallelism. It has implemented the *must have* and *should have* features presented in the MoSCoW analysis in Section 2.7 with the exception of linking LLVM's JIT compiler. On reflection, Poppy's overall design choices have been deemed to be not ideal. Section 2.5.1 aims to design Poppy using effective structural principles from Rust (such as Rust-style structs, traits, and implementations), and syntactic principles from Java. This design choice doesn't syntactically improve or build on anything from either Rust or Java. A more productive method of design would be to mirror either Rust or Java and build Poppy that is backward-compatible with existing Rust or Java code. Section 4.2 evaluates the language design differences between Poppy, Java, and Rust, referring to an implementation of a Binary Search Tree. Albeit, Poppy succeeds in this aim.

Design a static type system, implementing the core subset of the Kappa type system. This aim has exceeded expectations. Poppy's type system mirrors that of Kappa's type system, using capabilities in the data-race type checker to prevent the occurrence of data races

in a concurrent setting. It uses mechanisms and algorithms from the Bolt and Rust type system; the capability inference algorithm to reduce programmer overhead from Bolt, and the concept of ownership and borrowing from Rust. Evaluating Poppy's type system will be explored in Section 4.3 where Poppy's concurrency expressivity will be analysed against Java and Rust. Additionally, appendix C explores example programs that show Poppy's ability to statically prevent data-races.

4.2 Poppy's Language Design

One of the claims made in Section 2.5.1 is that Poppy will be designed with effective structural principles from Rust and syntactic principles from Java. By doing so, Poppy aims to reduce programmer overhead introduced in Kappa by removing the `pack`, `unpack`, and `jail` keywords used for capability composition and decomposition. Though, these keywords are only used to simply present the theoretical calculus, they are not present in the practical application of its programming language. So the goal of reducing programmer overhead present in Kappa by modifying the structure of language is redundant. With this in mind, it is more appropriate to evaluate Poppy's language design against Rust and Java. It will show the similarities and differences of Poppy, Rust, and Java, making reference to a Binary Search Tree implementation.

A Binary Search Tree (Appendix B) has been implemented to show the differences and similarities between the language design between Poppy, Rust, and Java. The implementation inherits a *multiple-reader-one-writer* pattern, represented by annotating the structs with **linear** and **read** capabilities. These capabilities are included at the beginning of every struct (declaration site), whereby every trait implementation of that struct inherits these capabilities. The implementation should be able to mutate both the left and right subtrees concurrently so, as described in Section 2.4.1, we can assign `leftChild` and `rightChild` disjoint linear capabilities.

Listing 4.1: Binary Search Tree implementation in Poppy

```
struct TreeNode {
  capability linear Left, linear Right, linear Elem, read Search;
  var int key : Elem, Search;
  var int value: Elem, Search;
  var TreeNode leftChild: Left, Search;
  var TreeNode rightChild: Right, Search;
  var int size: Elem, Search;
  var bool isNull: Elem, Search; // for null comparisons
}
struct BinarySearchTree {
  capability read View, linear Update, read ErrorCode;
  var TreeNode root: View, Update;
  var int errorCode : ErrorCode; // for exception handling
}

// ... trait implementation definitions

void main() {
    let lc = new TreeNode(isNull:true);
    let rc = new TreeNode(isNull:true);
    let root = new TreeNode (
        key: 1, value: 42, isNull:false, leftChild: lc, rightChild: rc
    );
```

```
    let tree = new BinarySearchTree (root: root, errorCode: -1);
    // tree has disjoint linear capabilities, so can be mutated safely in parallel
    finish{
        async{
            tree.search_left(root)
        }
        async{
            tree.search_right(root)
        }
        // ...
    }
    // ...
}
```

With Poppy's immaturity, most of the time with this implementation came from transcribing features that are not present its language. Poppy had been designed to be a minimally viable programming language, building only necessary features that will achieve the defined aims. Poppy doesn't allow generic types, generic implementations of structs, polymorphism, exception handling, and non-primitive comparisons. These cases had to be dealt with by improvising functionality. For example, Poppy improvises exception handling by implementing C-style error codes, as seen by the `errorCode` integer key in the struct definitions. Similarly, to handle non-primitive comparisons, a boolean `isNull` flag has been included in the struct definition.

Listing 4.2: Poppy's improvisations

```
if(this.isEmpty()) {
    this.errorCode // equivalent of raising an exception
}

if (currentNode.isNull){ // equivalent of null comparisons
    currentNode
}
```

4.3 Concurrency Expressivity

The problem of proving the absence of data-races in a program is inherently undecidable, which is why Poppy over-approximates the set of programs that may contain data races, which leads to the exclusion of some safe programs. This analysis, akin to other type systems, is based on syntax rather than semantics. For instance, Poppy will fail to type check programs with potential data races in conditional branches, regardless of whether these branches are ever executed. Bolt offers a comprehensive view of expressivity against Java and Rust in table 4.1.

Table 4.1: A comparison of concurrency patterns supported by Kappa/Poppy, Java, and Rust.

Concurrency Pattern	Bolt/Poppy	Java	Rust
Single-threaded:			

Continued on next page

Concurrency Pattern	Bolt	Java	Rust
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

* No protection against data races.

Single-threaded Design

With regard to single-threaded programs, Poppy's expressiveness is on par with Java. To adapt a Java program to Poppy (assuming classes have been replaced by Rust-style structs, traits, and implementations), a `local` annotation can be included in all struct declarations. This allows unrestricted aliasing and mutation within the bounds of a single thread. Rust on the other hand restricts mutation in the presence of aliasing. This example highlights differences in the approach these languages have toward handling concurrency, mutation, and aliasing.

Multi-threaded Design

Poppy, Java, and Rust each provide their own mechanisms for handling parallel, independent computations. However, their approaches to concurrency and the safeguards they offer against data races vary, reflecting differences in their type systems and concurrency models.

Java allows concurrency with shared states but does not inherently protect against data races. This means while Java supports concurrent programming, it places the responsibility of ensuring thread safety on the developer. Developers typically use synchronization mechanisms to prevent

data races, but this is not enforced by the language's type system. Rust, on the other hand, employs a more rigid ownership system for concurrency control. In Rust's model, references to an object are either entirely mutable (if they are unique owning references) or entirely immutable (if they are shared, non-owning references). This binary distinction in the Rust type system doesn't allow for selective mutability of an object's fields based on the thread context. As a result, Rust's type system often conservatively disallows the use of an object across multiple threads unless it can guarantee thread safety, such as through the use of atomic types or safe concurrency abstractions. Poppy, in contrast, introduces a more granular approach to concurrency. Its capability system operates at the level of individual field annotations within an object. This allows Poppy to provide guarantees about concurrent access to different parts of an object, ensuring that either the accesses are to disjoint fields or that the accessed fields are thread-safe. This capability-based approach enables Poppy to effectively manage more complex concurrency patterns than Rust, while still maintaining thread safety. Poppy is even capable of emulating Rust's ownership model using its capability system. By applying capabilities to the entire object rather than to individual fields, Poppy can mimic the all-or-nothing mutability model seen in Rust. However, Poppy's advantage lies in its ability to offer more nuanced control over concurrent object access, which can be particularly beneficial in complex multi-threaded applications where different parts of an object need to be accessed independently by different threads.

Programmer-specific

One of Poppy's anti-features is disallowing any mechanisms for programmer-managed locking or direct memory access. Instead, it solely relies on its type system to manage all access to memory. While this approach promotes safety by reducing the chances of error-prone manual concurrency management, it does have limitations, especially when dealing with complex concurrency algorithms that require precise control over shared data structures. For example, the *hand-over-hand* (or lock coupling) concurrency control algorithm is used mainly in linked data structures like trees or linked lists. When traversing trees, programmers lock certain nodes, allowing for concurrent computation. Given the data structure, the technique of requiring locks to be acquired and released may be done in a non-linear order. Similar to Java's `syncronized` keyword, Poppy's concurrency model doesn't allow for locks to be released in a non-linear order. Poppy's fork-join concurrency structure mandates that locks may be released in the reverse order of its acquisition. This limits Poppy's ability for certain concurrency patterns that require programmer-specific concurrency control. On the other hand, Rust allows for programmer-managed locks through its `Mutex<T>` library. The lock is released when the reference to the `Mutex<T>` falls out of scope. This method allows for more complex concurrency patterns by leveraging the safety benefits of its type system with the need for intricate concurrency control.

Inter-thread Communication

Poppy, while adept at managing shared buffers through locks, falls short in supporting the producer-consumer model fully, lacking a mechanism for threads to await the outcomes of others' computations. Java, offering monitor support, allows threads to signal others waiting on specific conditions, yet it doesn't extend to broader inter-thread communication mechanisms. In contrast, Rust embraces a different concurrency style, akin to the one popularized by Go: inter-thread communication via channels that are safe from data races. Rust's approach involves channels for message passing, where threads can transfer ownership of values safely. A thread sends a value to a channel, relinquishing its ownership, and another thread retrieves this value from the channel, assuming ownership. This system aligns with Rust's strict ownership rules within its type system, ensuring safe and controlled access across threads. However, it's noteworthy that all three languages –

Poppy, Java, and Rust – do not inherently support the actor model of concurrency. This model, characterized by lightweight threads (actors) delegating tasks to each other, represents a different paradigm of managing concurrent operations, distinct from shared-state or message-passing approaches.

4.4 Summary

Poppy was a success, achieving all the core criteria listed in Section 2.5 in creating a minimally viable, concurrent programming language. The effectiveness of Poppy's language design has been evaluated in Section 4.2, explaining changes made from Kappa's language, and the adoption of effective principles from both Rust and Java. Poppy's syntactic effectiveness is explored via the implementation of a Binary Search Tree. The implementation is compared against Rust and Java showing Poppy's similarity between the two, as well as its adaptation of its reference capabilities and capability inference features.

In the same vein, Poppy's expressivity has been compared against the concurrent expressivity of Rust and Java. Poppys's type system offered the greatest expressivity of any single approach to lock-free shared-state concurrency. This evaluation draws great attention to different concurrency models that are present in Rust and Java. By drawing attention to the hand-over-hand locking example, it showed the advantages of allowing progammer's to handle concurrency, allowing for the ability to create more intricate concurrency algorithms. This is telling that one concurrency paradigm could not represent every style of concurrent programming. Rust was the only language capable of offering a blended approach to concurrency, incorporating shared-state with message-passing. In an ideal strategy, incorporating Rust's concurrency model into Poppy would give programmers the ability to implement a broader range of concurrent programs and algorithms.

Chapter 5

Conclusions

Overall, Poppy was a success, fulfilling the aims presented in 2.5 with the exception of some nice-to-have features. This paper presented how the ideas from the Kappa type system could be translated into a practical programming language, postulated in the introduction.

The implementation of Poppy detailed the theoretical underpinnings of the type system, specifically how the Kappa type system has been modified to achieve the defined. Its type system surpassed has shown to be richer and more robust than type systems from other popular languages. Poppy's non-triviality was proven by its ability to type check programs that are hundreds of lines long. This included a data-race free implementation of a Binary Search Tree that used a multiple-reader one-writer concurrency pattern. Its capability inference algorithm allows for minimal annotation overhead, otherwise presented in the Kappa type system. We explored in depth the language design and highlighted key differences and similarities in languages Java and Rust, highlighting its effective syntactic design. Finally, its concurrency model is compared against that of Java and Rust, proving it **supports more fine-grained concurrency than Rust** in certain concurrency patterns.

5.1 Lessons Learnt

Building Poppy offered a wealth of valuable lessons in both programming language theory and practical software engineering.

OCaml Coming from a purely an imperative and object oriented programming background, writing a compiler in a functional language was a challenging component. It necessitated learning the functional programming paradigm before starting the project, along with becoming familiar with the libraries OCaml offers. The learning process consisted of utilising content and resources such as Real World OCaml [25], Jane Street's 'Learn OCaml Workshop' [26], and CS3110 OCaml Programming [27]. This lesson emphasised the ability to quickly and extensively learn a new programming paradigm, deemed critical to the nature of this project.

LLVM Without any formal education in compiler design, another great lesson learned was about LLVM, the target backend architecture of Poppy. LLVM was chosen to be Poppy's backend due to its high performance and ability to target multiple machine architectures. One of the more challenging components for this was becoming familiar with lower-level concepts such as memory allocation, registers, and instructions. This is necessary in the LLVM IR code generation phase,

whereby translating Poppy IR to LLVM IR. The process emphasised the importance of low-level translations in compiler design.

The importance of debugging and testing Debugging and testing are crucial components of any software project, particularly for one as large and complex as Poppy. It plays a vital role in ensuring the quality of the compiler, and the correctness of the type system. Though, OCaml's debugging tools have a steep learning curve, there is no reliance of a single debugging method. As a result, the most effective method turned out to be using debug print statements. On the other hand, OCaml has a range of testing libraries that are used for different purposes. Poppy was tested using Jane Street's `ppx_expect` library, primarily for its auto-generated test outputs. This streamlined the testing process and taught me the importance tests play for each stage in the compiler pipeline.

Time optimisation Perhaps the most significant lesson learned was optimising time management. As previously mentioned, the LLVM static compiler was not implemented for the final version of this project as a result of time management challenges. While this feature was a desirable extension that would have further proven the non-triviality of Poppy and allowed for performance benchmarking, more effective time management strategies could have made a difference. Firstly, utilising a Gantt chart would have been extremely helpful in creating estimates for task duration. A MoSCoW analysis, if conducted during the project's preparation stage (Technology Research Preparation), would have been beneficial. Finally, providing a more rigorous analysis of risks (especially for a project of this size) would provide a more well-defined project scope. These experiences have imparted a crucial lesson about optimizing time management for large projects, taking into account both internal and external risks. In future engineering projects, greater emphasis will be placed on efficient time management.

5.2 Future Direction

As discussed in the evaluation chapter about the successes and shortcomings of this project, the future direction of Poppy is logical. The natural direction of Poppy is to link the LLVM static compiler to be able to execute Poppy files. As discussed in the evaluation chapter, this was the only shortcoming of Poppy due to time constraints. Once demonstrated that Poppy can be used as a non-trivial language, one possible extension is to build syntactic equivalence with Rust. This would involve refactoring the grammar and including some concurrency patterns (such as programmer specified locking instructions) that are in Rust's concurrency model. By doing this, Poppy would then be backward compatible with existing Rust code.

Another possible extension is to verify Poppy's type system using formal methods. Formal verification is the process of mathematically proving the correctness of Poppy's type system through logical proofs and deductions. Proof assistants like Coq and Agda can be employed to prove properties about the system, such as type safety, soundness, and completeness. Though this goes beyond the scope of this project, it serves as an interesting extension for future work.

Bibliography

[1] Church, Alonzo (1932). "A set of postulates for the foundation of logic". Annals of Mathematics. Series 2. 33 (2): 346–366. doi:10.2307/1968337. JSTOR 1968337.

[2] Church, Alonzo (1936). "An unsolvable problem of elementary number theory". American Journal of Mathematics. 58 (2): 345–363. doi:10.2307/2371045. JSTOR 2371045.

[3] Church, Alonzo (1940). "A Formulation of the Simple Theory of Types". Journal of Symbolic Logic. 5 (2): 56–68. doi:10.2307/2266170. JSTOR 2266170. S2CID 15889861.

[4] Lafont, Y., 1988. The linear abstract machine. Theoretical Computer Science, 62(3), pp.327-328.

[5] Howard, W.A. (1969). The formulae-as-types notion of construction.

[6] Wadler, P., 1991. Is there a use for linear logic?. ACM SIGPLAN Notices, 26(9), pp.255-273.

[7] Baker, H., 1992. Lively linear Lisp. ACM SIGPLAN Notices, 27(8), pp.89-98.

[8] Moore, G., 2006. Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. IEEE Solid-State Circuits Society Newsletter, 11(3), pp.36-37.

[9] Jung, R., Dang, H., Kang, J. and Dreyer, D., 2020. Stacked borrows: an aliasing model for Rust. Proceedings of the ACM on Programming Languages, 4(POPL), pp.1-32.

[10] Wright, A. and Felleisen, M., 1994. A Syntactic Approach to Type Soundness. Information and Computation, 115(1), pp.38-94.

[11] Igarashi, A., Pierce, B. and Wadler, P., 1999. Featherweight Java. ACM SIGPLAN Notices, 34(10), pp.132-146.

[12] Tutorial.ponylang.io. 2022. Overview - Pony Tutorial. [online] Available at: `https://tutorial.ponylang.io/reference-capabilities/index.html`.

[13] Lu, S., Park, S., Seo, E. and Zhou, Y., 2008. Learning from mistakes. ACM SIGOPS Operating Systems Review, 42(2), pp.329-339.

[14] PFENNING, F. and DAVIES, R., 2001. A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science, 11(04).

[15] Weiss, A., GIERCZAK, O., PATTERSON, D. and AHMED, A., 2018. Oxide: The Essence of Rust. Arxiv,.

[16] CASTEGREN, E., 2018. Capability-Based Type Systems for Concurrency Control. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology,.

[17] Rust-lang.org. 2022. [online] Available at: https://www.rust-lang.org/.

[18] Borretti, F 2022, Lessons from Writing a Compiler, Fernando Borretti, viewed 27 May 2023, https://borretti.me/article/lessons-writing-compiler.

[19] Castegren, E & Wrigstad, T n.d., Extended abstract presented at IWACO'16, viewed 27 May 2023, http://palez.github.io/IWACO2016/castegren-iwaco2016.pdf.

[20] Jane Street. 2022. Janestreet/PPX_EXPECT: Cram like framework for ocaml. GitHub. https://github.com/janestreet/ppx_expect

[21] Rathi, M. 2020. Types for Data-Race Freedom. https://github.com/mukul-rathi/bolt-dissertation/blob/master/dissertation.pdf

[22] Pottier, F & Régis-Gianas, Y. 2018. Menhir Reference Manual https://www.semanticscholar.org/paper/Menhir-Reference-Manual-(-version-20181113-)-Pottier-R%C3%A9gis-Gianas/67f14e9e0f5a0763c3d86961eef22a3007f6039f.

[23] Smith, S. 2007. Ocamllex and Ocamlyacc. Practical OCaml, pages 193–211.

[24] LLVM OCaml API. 2021. Llvm. https://llvm.moe/ocaml/Llvm.html

[25] Real world ocaml. Real World OCaml. (n.d.). urlhttps://dev.realworldocaml.org/

[26] Janestreet. (n.d.-a). Janestreet/Learn-Ocaml-workshop: Exercises and projects for jane street's ocaml workshop. GitHub. https://github.com/janestreet/learn-ocaml-workshop

[27] OCaml programming: Correct + efficient + beautiful. OCaml Programming: Correct + Efficient + Beautiful - OCaml Programming: Correct + Efficient + Beautiful. (n.d.). https://cs3110.github.io/textbook/cover.html

Appendix A

Grammar

Listing A.1: Poppy is defined by the following grammar

```
program: // Poppy program
    | struct_defns, trait_defns, impl_defns, function_defns, main

struct_defns:
    | STRUCT StructName {mode, field_defns}

trait_defns:
    | TRAIT TraitName{method_signatures};

impl_defn:
    | IMPL TraitName FOR StructName {method_defns}

function_defn:
    | FUNCTION function_defns

capability:
    | CAPABILITY mode CapabilityName

mode:
    | LINEAR
    | LOCAL
    | READ
    | SUBORDINATE
    | LOCKED

field_defns:
    | modifier type_expr FieldName : CapabilityName

modifier:
    | CONST
    | VAR

type_expr: // types of expressions
    | StructName
    | int
    | bool
    | void
```

```
38
39  method_defns:
40      | method_signature {expr}
41
42  method_signatures:
43      |  MethodName BorrowedFlag CapabilityName (params) -> type_expr
44
45  function_defns:
46      | function_signature {expr}
47
48  function_signatures:
49      | FunctionName BorrowedFlag (params) -> type_expr
50
51  id:
52      | VarName // variable
53      | VarName.FieldAccess // field access
54
55  param:
56      | maybeBorrowed type_expr {capability_guards} ParamName;
57
58  constructor_args:
59      | FieldName:expr
60
61  async_expr:
62      | ASYNC {expr}
63
64  expr:
65      | (e)
66      | int
67      | true
68      | false
69      | id
70      | e1 binop e2
71      | unop e
72      | CONSUME id // consume identifier
73      | NEW StructName {constructor_args} // constructor
74      | LET VarName = e // define identifier
75      | LET VarName : type_expr = e
76      | id := e // assign to identifier
77      | VarName.MethodName // method application
78      | FunctionName (args) // function application
79      | IF (expr) {expr} ELSE {expr} // if statement
80      | WHILE (expr) {expr} // while loop
81      | FOR (expr, expr, expr) {expr} // for loop
82      | FINISH {async_expr} // finish async
83
84  binop:
85      | +
86      | -
87      | *
88      | /
89      | %
90      | <
91      | <=
```

```
92      | >
93      | >=
94      | &&
95      | ||
96      | ==
97      | !=
98
99  unop:
100     | !
101     | -
```

Appendix B

Binary Search Tree

Location: `https://github.com/os12345678/Poppy/blob/main/examples`

Appendix C

Poppy's Test Suite

Listing C.1: Trying to access linear capability in multiple locals

```
struct Foo {
    capability linear Bar;
    var int f : Bar;
}

trait Baz {
    id : Bar (int x) -> int
}

impl Baz for Foo {
    id : Bar (int x) -> int {
        x
    }
}

void main(){
let x = new Foo();
    finish{
        async{
            x.f // error - as accessing linear capability in multiple locals
        }
    x.f
    }
}

// OUTPUT: Compilation error: Potential data race: 1:18, c:5 Can't access capabilities
    Bar and Bar of object x concurrently
```

Listing C.2: Trying to access local variable from multiple locals

```
struct Foo {
capability local Bar;
var int f : Bar;
}

void main(){
    let x = new Foo();
```

```
        finish {
            async{
                x.f := 1
            }
        let y = x;
        y.f
    }
}

// OUTPUT: Compilation error: 1:10, c:17 Potential data race: no allowed capabilities
    for Objfield: (Struct: Foo) x.f
```

Listing C.3: Trying to access capabilities concurrently that don't share safe state

```
struct Foo {
  capability linear Bar, linear Baz;
  var int f : Bar, Baz; // since Baz and Bar aren't both safe(), we can't access this
      field concurrently
}

trait Baz {
    get : Baz () -> int
}

impl Baz for Foo {
    get : Baz () -> int {
        this.f
    }
}

void main(){
    let x = new Foo();
        finish{
            async{
                x.f
            }
        x.get()
    }
}

// OUTPUT: Compilation error: Potential data race: 1:18, c:9 Can't access capabilities
    Baz and Bar of object x concurrently
```

Listing C.4: Trying to assign to a read capability

```
struct Foo {
capability read Bar;
var int f : Bar;
}

void main(){
    let x = new Foo();
    x.f := 1
```

```
}

// OUTPUT: Compilation error: l:8, c:5 Potential data race: no allowed capabilities for
    Objfield: (Struct: Foo) x.f
```

Listing C.5: Good capability annotations

```
struct Foo {
capability linear Bar, read Baz;
var int f : Bar;
const int g : Bar, Baz;
const int h : Baz;
}

fn f (Foo{Bar,Baz} y) -> int {
    y.f + y.g
}

void main(){5}
```

Listing C.6: Good immutable references in multiple threads

```
struct Foo {
capability read Bar, linear Baz;
const int f : Bar, Baz;
}

fn test() -> int {
    5
}

void main() {
    let x = new Foo(f:5);
    let y = 5;
    finish{
        async{
            x;
            test();
            y
        }
        x;
        y
    };
    x.f
}
```