Part II project proposal

Name: Atharv Nema

Title: An actor-based programming language for safe concurrent programming

1. Introduction

Writing safe concurrent code is difficult. Major problems in this area are data races and deadlocks. Data races occur when multiple threads access shared memory simultaneously without proper synchronisation, resulting in undefined behaviour. Deadlocks occur when threads wait indefinitely for resources locked by each other, causing the threads involved to halt indefinitely. Bugs caused by these issues are notoriously subtle and difficult to reproduce. Mainstream programming languages provide little direct support to help programmers avoid them.

One programming model that circumvents these problems is the actor model. In this model, concurrency is structured around lightweight entities called actors. An actor encapsulates its own private state and communicates with other actors by sending and receiving messages. There is no shared memory or locks in this model, and hence it avoids deadlock and data race issues.

However, pure message passing has real costs. Copying data between actors can be expensive, and coordination patterns that are naturally expressed with locks become awkward or incur extra latency. In short, forbidding shared memory and locks entirely simplifies correctness but can hinder performance and expressiveness.

This project aims to implement a compiler for an actor based programming language, provisionally named '*Coherence*', that statically rules out data races and deadlocks. *Coherence* aims to keep the benefits of the actor model while reintroducing shared memory and locks safely. It does this by using reference capabilities, inspired by those used in the Pony programming language [1], which restrict how references can be used and whether they may be sent between actors. To support mutable shared state coordinated via a lock, *Coherence* will provide a *locked* reference capability, similar in concept to (and sharing its name with) a reference capability used in Kappa [2]. Data under this capability may only be accessed inside atomic sections. Following Gudka's work on deadlock-free programming with atomic sections [3], the compiler computes for each atomic section the set of locks that may be touched on any path, acquires them up front in a compiler-chosen consistent order, and releases them on exit. This guarantees data race and deadlock freedom.

2. Starting point

- Solid grounding at the Part IB level in Compilers, C++, Concurrent & Distributed Systems, and Computer Architecture.
- Some familiarity with existing literature on type systems for safe concurrent programming (e.g. Pony, Kappa, Rust).
- Some prior hands-on experience with lexing and parsing.
- No code currently exists. Implementation will begin from scratch.

3. Substance of the project

Key concepts:

- Reference capabilities: Type system feature used to control how references to shared data can be used.
- Actor-based model: Concurrency is structured around lightweight entities called actors. Each actor encapsulates its own private state and defines behaviours that are triggered when messages are received. Communication occurs through asynchronous message passing: when an actor sends a message, it continues immediately without waiting for the message to be processed. Actors handle one message at a time to completion, potentially creating new actors or sending further messages in response. *Coherence's* actor model also guarantees causal message delivery to make reasoning about program behaviour easier, drawing inspiration from Pony's actor model.
- Atomic sections and lock-set inference: Atomic sections are sections of code in a thread/actor that must run to completion without interruption. The compiler statically analyses the code to determine a conservative set of locks that the atomic section may touch on any execution path, and emits code to acquire them up front in a consistent order and release them on exit.

Major work items:

- Language design: Define the syntax and operational semantics of *Coherence*.
- Compiler implementation: Lexing, parsing, type checker, lock-set inference and LLVM IR generation.
- Evaluation: Verify correctness with rigorous unit and integration tests. Use a representative set of concurrent programs to demonstrate both the expressive power of the language and that well-typed programs are free from data races and deadlocks.

4. Goals

Core goals:

- Implement a type checker that enforces that all accepted programs are data race and deadlock free.
- Implement a working end-to-end compiler that takes in *Coherence* programs and translates them to LLVM IR.
- Verify that the compiler can successfully compile and run a representative suite of example programs.

Possible extensions:

- To evaluate *Coherence* realistically, it is important to take memory usage into account. Without a garbage collector, *Coherence* programs would accumulate unreachable data, unfairly disadvantaging it in comparison with other languages. As an extension, I will implement a simple garbage collector to ensure that performance evaluations are not distorted by the accumulation of unreachable data and accurately reflect the language design.
- In core *Coherence*, programmers can only access references with the *locked* capability inside atomic sections, where the compiler automatically determines and acquires the necessary locks. While this ensures deadlock freedom, it can be overly restrictive or inefficient in some cases. As a possible extension, I will explore mechanisms for giving programmers more explicit control through manual lock acquisition, while still maintaining freedom from deadlocks.

- Explore compiler optimizations.

5. Evaluation

Core evaluation:

- Testing the compiler: Unit tests for individual parts of the compiler (lexer, parser, type checker, and IR generation) and integration tests for end-to-end compilation and execution to demonstrate correctness.
- Qualitative evaluation of language features: Implement representative concurrent algorithms to demonstrate expressive power and ergonomics of *Coherence*.

Optional evaluation:

- Compare performance of programs in *Coherence* against equivalent implementations in other similar languages.
- Write formal proofs on *Coherence's* type system, demonstrating that the typing rules enforce the intended safety properties (freedom from data races and deadlocks) with respect to the operational semantics.

6. Work plan:

Work package 1 (13/10/2025 – 26/10/2025):

- Review literature on different reference capabilities and formalizations of the actor model.
- Design the syntax and operational semantics of *Coherence*.

Milestone: Draft document describing syntax and semantics, noting down rationale for design choices.

Work package 2 (27/10/2025 – 09/11/2025)

- Study compiler construction in C++ and acquire working knowledge of LLVM.
- Design a preliminary AST.
- Implement a basic lexer and parser using a generator that can parse desugared programs of *Coherence*.

Milestone: Lexer and a parser that can parse simple programs into an AST.

Work package 3 (10/11/2025 – 23/11/2025)

- Make further modifications to the lexer/parser to represent the full core language syntax.
- Implement a type checker.
- Write unit tests for the type checker to evaluate its correctness.

Milestone: Functional compiler front end.

Michaelmas term contingency (24/11/2025 – 07/12/2025)

Reserved for module work and backlogs from earlier work packages.

Work package 4 (08/12/2025 – 21/12/2025)

- Translate AST to LLVM IR for an initial subset of *Coherence*.
- Implement a simple actor scheduler to support execution of test programs.

- Begin writing end-to-end tests and building the program corpus to be used in qualitative evaluation.

Milestone: Compiler generates correct LLVM IR for trivial well-typed programs.

Christmas break / contingency (22/12/2025 – 04/01/2026)

Reserved for rest, exam revision preparation, or backlogs from earlier work packages.

Work package 5 (05/01/2026 – 18/01/2026)

- Extend LLVM IR generation to cover full language features.
- Complete end-to-end tests and the program corpus.
- Begin work on progress report.

Milestone: Compiler passes success criterion: correct end-to-end compilation of representative actor programs.

Work package 6 (19/01/2026 – 01/02/2026)

- Write the progress report and start working on the presentation.
- Improve error messages, code quality and documentation.
- *Extension*: Preliminary reading and planning for potential extensions.

Milestone: A completed compiler core.

Work package 7 (02/02/2026 – 15/02/2026)

- Submit progress report and deliver progress report presentation.
- Start working on the Introduction and Preparation chapters of the dissertation.
- *Extension*: Implement a simple mark and sweep garbage collector.

Milestone: Progress report and presentation completed.

Work package 8 (16/02/2026 – 01/03/2026)

- Start working on the implementation chapter of the dissertation.
- *Extension*: Start writing benchmarks for programs and looking into compiler optimizations.

Work package 9 (02/03/2026 – 15/03/2026)

- Start working on the evaluation and conclusion chapters of the dissertation.
- *Extensions*: Implement a few optimizations and finish comparisons with programs in other languages.

Work package 10 (16/03/2026 – 29/03/2026)

- Complete dissertation draft.
- *Extensions:* Extend the syntax and operational semantics to allow the programmer to manually acquire locks while still statically maintaining deadlock and data-race freedom.

Milestone: Provide full draft dissertation to supervisors and DoS.

Easter break / contingency (30/03/2026 – 12/04/2026)

Reserved for rest, exam revision preparation, or backlogs from earlier work packages.

Work package 11 (13/04/2026 – 26/04/2026)

- Address reviews on dissertation draft.
- *Extensions:* Extend the lexer, parser, AST, type checker and compiler to work with this additional syntax.

Work package 12 (27/04/2026 – 15/05/2026)

- Finish the dissertation and any remaining code and prepare them for submission.

Milestone: Submit the completed dissertation and source code.

Resources declaration:

I will be using my own personal computer for this project. The specifications are:

Processor: Intel(R) Core(TM) Ultra 7 155H (3.80 GHz)

Installed RAM: 32.0 GB

System type: x64-based processor, running Windows dual-booted with Ubuntu Linux

I will use Git as my version control system, with all source code and any other files relevant to my dissertation maintained in a private GitHub repository.

In the event that my personal computer fails, I plan to use a computer belonging to the MCS. I will be able to clone the repository from GitHub and continue work without major disruptions.

Declaration: *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.*

Bibliography:

[1] S. W. Clebsch, "Pony: Co-designing a type system and a runtime," Ph.D. dissertation, Dept. of Computing, Imperial College London, Oct. 2017. [Online]. Available: https://www.ponylang.io/media/papers/codesigning.pdf

[2] E. Castegren, *Capability-based type systems for concurrency control*, Ph.D. dissertation, Acta Universitatis Upsaliensis, 2018. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-336021

[3] K. Gudka, *Lock inference for Java*, Ph.D. dissertation, Dept. of Computing, Imperial College London, London, U.K., Dec. 2012. [Online]. Available: https://www.khilan.com/pubs/khilan-phd-thesis.pdf