University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

A Concurrent Runtime for the Pat Programming Language

Kai Wang
March 10, 2024

Abstract

The complexity of concurrent programming and the challenges of distributed systems are becoming increasingly emphasized in modern computing, especially with major problems such as thread deadlocks and communication mismatches. The Pat language tightly constrains concurrent tasks and communication flows by introducing a mailbox type system and concurrent programming primitives. While the Pat language has implemented a type checker, a corresponding runtime interpreter has not yet been developed. We will implement it using a task scheduler based on the CEK machine execution model and a Round-Robin algorithm to improve the efficiency and accuracy of concurrent programming. And support for the unique "Free" guard is provided using a garbage collection strategy.

This approach is the first time that a mailbox type system has been used at the programming language level, providing a new programming model for concurrent programming. Through a series of correctness tests and micro-benchmarks, we find that the Pat language is able to efficiently handle a variety of concurrent programming tasks and exhibits good performance under multiple concurrency models. These results not only demonstrate the potential of the Pat language in the field of concurrent programming, but also provide a new direction for future research and development of concurrent programming models.

Acknowledgements

I would like to start by giving special thanks to my supervisor, Simon Fowler, whose expertise and passion for research has deeply influenced me throughout this research process, and whose patient guidance and invaluable advice at every meeting has been crucial in enabling me to successfully complete this work. He always encouraged me while teaching me how to cope with challenges and difficulties. For this, I express my deepest gratitude.

I also wish to express my sincere appreciation to my girlfriend, whose selfless support and encouragement was an indispensable strength to me during my research. I would also like to thank myself. During this academic journey, I have faced many challenges and achieved a lot. I am proud of my growth and progress and look forward to the future.

Finally, I would like to thank all those who have given me help and support. Your encouragement and trust is what keeps me going.

Education Use Consent

Contents

1 Introduction

1.1 Concurrency and Distributed Systems

Concurrency and distributed systems are becoming increasingly important in modern computational science, especially in the context of cloud computing and big data. Sutter and Larus (2005) emphasize that with the proliferation of multicore processors, the software industry needs to adopt new tools and ways of thinking in order to fully utilize the potential of multicore processors. Concurrency refers to the ability of a system to handle multiple tasks at the same time, which allows efficient use of computational resources. For example, by processing hundreds or even thousands of client requests in parallel, modern web servers are able to dramatically improve response times and system throughput. This shift in demand requires mainstream software development to stop ignoring concurrency, a challenge that requires developers to master the design and implementation of concurrent programs (Sutter and Larus (2005)).

The design of Multiplayer Online Role-Playing Game (MMORPG) servers, such as exemplified by the World of Warcraft (2019) (Figure 1.1) demonstrates the value of concurrent programming in real-world applications, where the use of a client-server model and event-driven architecture enables asynchronous processing of player requests. Kim and Kim (2018) study, by utilizing Input/Output Completion Port (IOCP) and multithreading techniques, has demonstrated methods to effectively improve server performance, handle concurrent connections and employ fine-grained locking mechanisms for access control of shared resources to reduce the risk of deadlock and effectively improve system responsiveness and throughput.

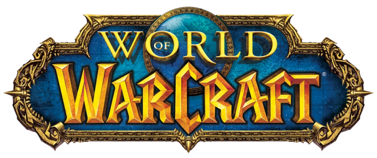

Figure 1.1: *World of Warcraft*

Figure 1.2: *Pokémon GO*

On the other hand, distributed systems achieve workload decentralization by assigning tasks across multiple compute nodes, which is especially critical for processing large-scale data sets. For example, the distributed architecture of Pokemon GO (2019) demonstrates the efficient ability of distributed systems to manage and synchronize millions of players globally, through a network of servers deployed in multiple locations around the world, and the use of geo-distributed database technology to decentralize the storage of data based on the geographic location of players, showing the importance of this technology in dealing with large-scale, real-time interactive applications.

The combination of concurrency and distributed systems not only greatly improves the efficiency of task processing, but also enhances the reliability and scalability of the system. In the case of distributed databases, for example, data is replicated across multiple nodes, ensuring that the entire system can maintain normal operation even if some nodes fail. In addition, this technology supports the dynamic adjustment of computing resources according to actual demand, a feature that is widely used in cloud service platforms such as Amazon Web Services (2023) and Microsoft Azure (2023). This exposition not only demonstrates the key role of concurrent programming in ensuring high availability and optimizing user experience, but also provides an important reference for designing efficient concurrent and distributed systems.

1.2 Challenges

While these systems are capable of handling unprecedented amounts of data and computational tasks, they also introduce a new set of challenges.

1.2.1 Thread Deadlock

Thread deadlocks are a common problem in concurrent programming, which in short means that no thread is able to continue execution due to circular locking or communication dependencies. For example, if two threads each occupy a portion of a resource and are waiting for the other to release its resource in order to continue execution, this creates a deadlock. This situation is particularly common in database transactions and operating system resource allocation, and it can seriously affect the stability and efficiency of a program.

1.2.2 Communication Mismatch

Communication mismatch is another critical issue in distributed systems. Nodes in a distributed environment need to exchange information frequently, and any inconsistency in communication protocols or data formats can lead to major failures. Taking microservice architecture as an example, if the communication interfaces between services are not clearly defined or fail to maintain compatibility during version updates, it may lead to the functional failure of the whole system. This is particularly noticeable in fast iterative and multi-team projects and requires special attention.

1.3 Case Study: The Mailbox Mechanism

When exploring the field of concurrent programming in depth, actor modelling languages, such as Erlang, provide an effective concurrent programming paradigm. By introducing mailboxes as a communication mechanism, these languages allow processes to collaborate with each other in a message-passing fashion, effectively avoiding the locking and shared state management problems of traditional concurrent programming. However, while mailboxes serve as a bridge between actors, they do not explicitly constrain the types of messages and can only process messages in the order they are received, which is not sufficient to deal with an inherently unorganized message flow.

Building on this, the Pat language, an emerging language in the research field, introduces the notion of mailbox types for first-class mailboxes(Fowler et al. (2023)). Compared with traditional actor modelling languages, the Pat language is designed to emphasize the importance of mailbox types, and further enhance the security and maintainability of concurrent programs by constraining the types and processing order of messages in the mailboxes through a type system.This innovative point of the Pat language not only simplifies the complexity of concurrent programming, but also provides new perspectives and tools for research in the field of concurrent and distributed programming.

In further exploring the application of the Pat language in the field of concurrent programming, this paper aims to show how mailbox types in the Pat language can effectively manage concurrent tasks by introducing a concrete code example. As we see in Listing 1.1, By defining the `interface Tasks` and its implementation, the message type in the mailbox is forced to be constrained to String. The sample code clearly shows the process of producing and consuming tasks, where the `produceTasks` function is responsible for sending tasks, encapsulated as String payloads, to instances of the Tasks interface, and the `consumeTasks` function is responsible for receiving and executing these tasks.The Pat language utilizes the concurrency primitive **spawn** to implement the parallel execution of functions. Thus, tasks are produced and consumed simultaneously in different execution streams.

```
interface Tasks {
  Task(String)
}

def produceTasks(mb : Tasks!) : Unit {
  mb ! Task("task1");
  mb ! Task("task2")
}

def consumeTasks(self: Tasks?) : Unit {
  guard self: Task {
    free -> ()
    receive Task(i) from self ->
        print(i);
        consumeTasks(self)
  }
}

def main(): Unit {
  let task = new[Tasks] in
  spawn { produceTasks(task) };
  consumeTasks(task)
}

main()
```

Listing 1.1: Pat language example which enforces Task message payloads to be of type String

1.4 Research Goals

The aim of this research is to develop a runtime interpreter to run a new programming language, Pat language. The Pat language addresses the programming challenges of concurrency and distributed systems, and by introducing advanced concurrent programming primitives and a unique mailbox type system, it aims to provide developers with a more concise way to handle concurrent tasks and communication flows. While a type checker for the Pat language has been implemented, an efficient execution environment has yet to be developed. The main goal of this project is to build an execution environment, so developers can easily write, test and run programmes written in the Pat language.

This interpreter should not only satisfy the runtime requirements of a general programming language, such as memory management, but should also focus on the specific needs of concurrent and distributed computing. By providing such a tool, the Pat language will become a powerful support for the research and development of concurrent and distributed systems, which not only promotes the wide application and continuous development of the Pat language, but also provides new perspectives and methods for exploring problems in the field of concurrent programming, and establishes a solid foundation for future technological innovations.

1.5 Outline

This dissertation organized into the following structure:

- **Chapter 2** introduces the need for concurrent programming, and discusses the definition, differences, and relative advantages and disadvantages of channels and actors. It also examines the mailbox type system, its significance in concurrent programming, and outlines scheduling algorithms and their implementation in OCaml.
- **Chapter 3** identifies the essential functionalities required for the project, desirable features, and potential enhancements. It delineates the scope of the current phase by highlighting the aspects of research that are not addressed.
- **Chapter 4** details the application of the CEK mechanism within the Pat language. It elucidates the rationale behind choosing CEK over CPS and describes the design of Pat's scheduling and garbage collection mechanisms.
- **Chapter 5** covers the implementation of the CEK machine, the communication and scheduling mechanisms within Pat's mailbox system, the current state of garbage collection, and the development of both the evaluation step printer and the error-handling mechanism.
- **Chapter 6** assesses the system's performance using correctness and runtime tests, includes a critical evaluation, and discusses the outcomes and potential areas for improvement.
- **Chapter 7** reviews Pat language's principal contributions, summarizes the findings of the thesis, and proposes future work directions and potential research application areas.

2 Background

2.1 Concurrency

Modern concurrent programming uses two main paradigms: Message-Passing Concurrency and Shared-Memory Concurrency, which use different strategies and mechanisms to handle concurrent tasks.

2.1.1 Shared Memory

Shared-memory concurrency model allows multiple processes or threads to share the same memory space and communicate through this shared space. While this approach improves efficiency in certain scenarios, it also introduces data consistency and synchronisation issues. In the shared memory model, developers must carefully handle locking mechanisms and synchronisation issues to prevent deadlocks and race conditions from occurring. Java and C++ are examples of languages that support this concurrency model.

Listing 2.1 shows an example of a deadlock in Java. If the first thread successfully acquires `lock1`, and the second thread acquires `lock2` at the same time, then each thread will be blocked waiting for the other thread's lock. Both threads will wait indefinitely because the locks they each hold will not be released.

```java
......

new Thread(() -> {
   synchronized (lock1) {
      System.out.println("Thread 1: Holding lock 1...");
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      System.out.println("Thread 1: Waiting for lock 2...");
      synchronized (lock2) {
         System.out.println("Thread 1: Holding lock 1 and 2...");
      }
   }
}).start();

new Thread(() -> {
   synchronized (lock2) {
      System.out.println("Thread 2: Holding lock 2...");
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      System.out.println("Thread 2: Waiting for lock 1...");
      synchronized (lock1) {
         System.out.println("Thread 2: Holding lock 1 and 2...");
      }
   }
}).start();

......
```

Listing 2.1: Java example demonstrating a potential deadlock

2.1.2 Message Passing

In contrast, Message-passing Concurrency is a programming paradigm that enables communication and coordination by sending and receiving messages between processes or threads. The main advantage of this approach is that it provides a clear mechanism to avoid data contention and conflict, as data is passed through message exchange rather than sharing. In this model, each process or thread has a separate memory space and all communication takes place through well-defined message interfaces, but additional overhead is introduced.The Erlang language and the Akka framework are excellent representatives of this model.

Although message-passing concurrency and shared-memory concurrency each have their own strengths, message-passing concurrency provides a clearer and more controlled approach to dealing with the problems of distributed systems and reducing complex synchronisation issues. Therefore, the message-passing concurrency paradigm was chosen in the design of the Pat language in order to provide a safer and more intuitive approach to concurrent programming.

2.2 Communication Mechanisms

With its clear communication mechanism, the message-passing concurrency model provides the Pat language with an effective means to avoid data conflicts and competition, which significantly enhances the advantages of concurrent program design. On this basis, by introducing two high-level abstractions, Channel and Actors, not only the expressive ability of message-passing concurrency is further enhanced, but also the manageability and extensibility of concurrent programs are significantly improved (Figure 2.1). Nevertheless, both models have revealed some problems in practice.

2.2.1 Channel

The channel model, especially widely used in languages such as Go, facilitates message passing between processes or threads by naming channels. As shown in the Figure 2.1.a, processes or threads communicate and coordinate with each other by passing messages through these channels. These named channels allow messages to be exchanged between different execution units and are a key mechanism in concurrent programming. However, the use of channels is not without challenges, especially when it comes to synchronization issues. These problems stem mainly from communication mismatches and channel-based deadlocks, for example, when two processes are both waiting to receive data on the same channel, or when they try to send data on a channel blocked by a receive operation. These types of deadlocks and communication mismatches are common challenges in concurrent programming in the channel model.

Implementing channels in a distributed environment is more complex, as discussed by Chaudhuri (2009) in his paper "Concurrent ML Library in Concurrent Haskell", which is mainly attributed to the complexity of the distributed abstract state machine required for synchronization. Channels, points, and synchronizers interact according to rules and states defined for each entity to facilitate operations and transitions. In turn, state changes of entities lead to the complexity of implementing channels. In addition, the implementation of channels in a distributed environment involves not only communication structures and types, but also how to handle complex interaction patterns such as iteration, branching, and delegation(Hu et al. (2008)). These researches show that while channel messaging between processes or threads explicitly specifies the type of data transferred by the channel, ensuring deadlock-free and efficient communication in distributed systems is a challenge.

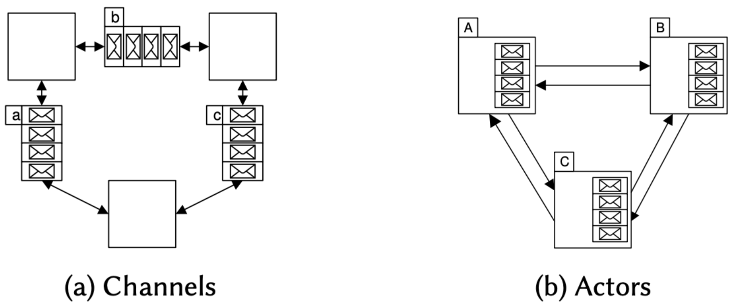

Figure 2.1: *(a) illustrates the Channels model with concurrent units communicating via message-passing channels. (b) shows the Actors model, where each Actor independently processes messages from its queue. (Fowler et al. (2017))*

2.2.2 Actor

In contrast, the actor model, first proposed by Hewitt et al. (1973) and implemented in languages such as Erlang, demonstrates the efficiency and generalizability of the actor model for AI languages dealing with the need for high parallelism. The model focuses on enabling each individual actor entity to interact via asynchronous message passing. According to Figure 2.1.b, where each Actor is an addressable process with a message queue or mailbox for receiving messages. Each Actor is a separate entity that interacts via asynchronous message passing. This model is particularly well suited for building large-scale, scalable distributed applications, providing natural support for distributed computing, especially in terms of fault tolerance and system reliability. Agha (1990) points out that the Actor model has addressed key challenges in distributed computing, such as deadlocks and divergence. However, the actor model faces challenges in implementing type safety, and in the case of Akka, a framework that provides a model based on the actor model, it relies on dynamically typed messages, which can lead to type errors that are not detected until runtime, making debugging and maintenance more difficult. He et al. (2014) in "Typecasting Actors: from Akka to TAkka" in which they first attempted to address this problem and improve type safety by introducing statically typed messages to TAkka. Compared to the simplicity of specifying message types directly in the channel model, such an improvement requires additional design and implementation on top of the actor model, which is more complex to implement.

2.2.3 Pat's Solution

To cope with these challenges, the introduction of Actor type systems has become an important advancement. In this context, the design of the Pat language introduces a new solution based on the Actor model along with a mailbox behavioural type system, which is the first time in a behavioural type system at the programming language level.

2.3 Mailbox

The mailbox communication mechanism is a core concept of concurrent programming widely used in a variety of programming environments, introduced by Ford and Hamacher (1976). It allows program components to exchange messages asynchronously, thus enabling efficient communication between processes or threads. And Erlang, as a language designed for concurrent and distributed programming, deeply integrates the mailbox communication mechanism as the cornerstone of its inter-process communication.

Listing 2.2 is a concrete example of the application of the mailbox communication mechanism in

Erlang: an asynchronous task processing model. In this model, the `empty_task` function is a standby process that performs an asynchronous computation by receiving a message containing a Result. When the computation is complete, the Result is passed to the `queryable_task` function. The function is in a waiting state, ready to respond to the client's query request and by sending the result of the computation back to the requester. In addition, it handles error status and returns an error if the computation request is received again. The client starts the asynchronous task through the client function, first creating a new process to run `empty_task` through spawn, then sending computation commands and query commands to the process, and eventually receiving and printing out the results of the computation.

However, it also reveals some potential problems, such as the risk of self-deadlock. Self-deadlock occurs when a process waits for an event that will never occur, causing the process to hang permanently. In this example, if the AsyncTask process fails to receive any kind of query request after completing a computation task, possibly due to a client failure or logic error that did not send a query message, the `queryable_task` will wait indefinitely for the query request, resulting in wasted resources and potential system performance degradation. Additionally, if the client mistakenly sends the computation command again instead of the query command, although the program is designed to return an error, this does not elegantly address the problem of receiving the computation request again in the asynchronous task completion state, and may result in unnecessary error handling and resource usage.

```erlang
empty_task() ->
   receive
      { compute, Result} -> queryable_task(Result)
   end.

queryable_task(Result) ->
   receive
      { query, Pid } ->
         Pid ! { reply, Result },
         queryable_task(Result);
      { compute, _ } -> erlang:error("Task already computed")
   end.

client() ->
   AsyncTask = spawn(async_task, empty_task, []),
   AsyncTask ! { compute, 2 },
   AsyncTask ! { query, self() },
   receive
      { reply, Result } ->
         io:fwrite("Task result: ~w~n", [Result])
   end.
```

Listing 2.2: *Asynchronous task handling and querying using Erlang adapted from Fowler et al. (2023)*

2.3.1 Behavioural Type System

As described in the book "Behavioural Typing: from Theory to Tools" written by Gay and Ravara (2017), the Behavioural Type System provides an innovative solution to the type problem. The system facilitates efficient cooperation patterns in distributed applications by regulating inter-component interactions through behavioural contracts as service-level protocols. By defining precise communication protocols (i.e., timing session types), the behavioural type system is able to carefully control and manage the timing of inter-component interactions, effectively deal with concurrency and synchronization issues, and ensure the consistency and predictability of system behaviour. In addition, the runtime monitoring mechanism of the behavioural type system can detect and respond to abnormal behaviours in the system in real time, which significantly improves the reliability and security of the system.

Further, Ancona et al. (2016) researchers point out that recent developments in behavioural type theory are applied to guarantee diverse correctness properties of large-scale communication-intensive systems. They explored ways to use behavioural types in conjunction with object-oriented programming. In summary, behavioural type systems extend the scope of checking to the verification of behavioural attributes, such as communication matching problems, in addition to helping to detect the usual data type errors. And through their application in a wide range of programming languages and computing environments, they provide empirical support for ensuring the correctness, security, and maintainability of distributed systems. This emphasizes the importance of behavioural type systems in modern software development, especially when building complex distributed applications.

2.3.2 Mailbox Type

Mailbox types are an essential part of behavioural type systems, originally used in process calculus like the π-calculus(Milner (1999)) to capture mailbox contents mainly in the form of exchanged regular expressions. Mailbox types were introduced by de'Liguoro and Padovani (2018) to enhance the message-passing mechanisms of programming languages based on the role model. In the Pat language, each actor's mailbox is defined in detail through type annotations that specify not only the types of messages that the mailbox can receive, but also contain expected behavioural patterns such as the order in which messages are sent and received. This fine-grained type annotation allows developers to perform strict type checking of message passing patterns to achieve precise control of complex communication behaviours, which significantly improves the type safety of concurrent programming and effectively avoids common concurrency problems such as violation of communication protocols and deadlocks.

The following code snippet(Listing 2.3) from Pat's program demonstrates the application of mailbox types in real-world programming.This code simulates the processing of asynchronous compute and query tasks by defining `emptyTask` and `queryableTask` functions, as well as a client function. Where `emptyTask` is a mailbox for an empty task that can receive one `Compute` message and multiple `Query` messages. Once a `Compute` message is received, the mailbox will call `queryableTask`. and `queryableTask` represents a mailbox that has already received a `Compute` message, and this mailbox can only receive multiple `Query` messages. In these definitions, `guard` expressions are used to monitor the mailbox self and perform the appropriate actions based on the type of message received. In addition, we are able to capture a range of potential errors. For example, if we ignored sending the `Compute` message, the empty mailbox would not enter the queryable state, resulting in the task not being able to be queried; if we did not send a reply according to the protocol, the consumer would be in a waiting state; and sending two `Compute` messages would violate the mailbox's rules since each task should be in only one queryable state after the completion of the computation should be in only one query-only state. This pattern not only shows how mailbox types can be used in Pat to control the sending and receiving of messages, but also how Pat can use the type system to statically check the behaviour of concurrent programs to ensure communication protocol compliance and program robustness.

```
def emptyTask(self:EmptyTask):1 {
   guard self: Compute ·*Query {
      receive Compute[result] from self ->
         queryableTask(self, result)
   }
}

def queryableTask(self:FullTask, Result:Int):1 {
   guard self: *Query{
      free -> ()
      receive Query[user] from self ->
         user ! Reply[Result];
         queryableTask(self,Result)
   }
}

def client():1 {
   let asyncTask = new in spawn emptyTask(asyncTask);
   let self = new in
   asyncTask ! Compute[2];
   asyncTask ! Query[self];
   guard self: Reply {
      receive Reply[Result] from self ->
         free self;
         print(intToString(result))
   }
}
```

Listing 2.3: *Implementing concurrent program for asynchronous task processing and result querying adapted from Fowler et al. (2023)*

2.4 λ-Calculus

Before delving into the implementation of Control Flow Management techniques such as CPS and CEK, it is crucial to understand the underlying concepts and principles of λ-Calculus, which forms the basis of all functional programming languages, including Pat. It is a formal system for representing the logic of a computational process or mathematical function, introduced by mathematician Alonzo Church in the 1930s, which is one of the core foundations of modern computer science. The system focuses on the process of binding and substitution of variables and serves as a generalized model of computation, equivalent to a Turing machine, that exhibits the fundamentals of the theory of computation.

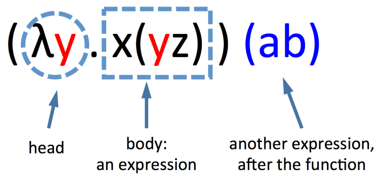

Figure 2.2: *Illustration of λ-Calculus expression, showcasing the head of the expression marked in red, the body of the expression in dashed blue, and another expression following the function in solid blue (Lambda Calculus (2015))*

In the λ-Calculus system, all functions are defined anonymously, and their definition and application are accomplished uniquely through λ-abstraction. λ-expressions essentially define a function, which is realized by binding variables and expressing the expression inside the function body. The application of the function then simply places the actual arguments after the function name, without the need to separate them with parentheses or commas (Figure 2.2). For example, consider the λ-expression for a function that adds two numbers:

$$\lambda x.\lambda y.x + y$$

This expression is made up of two lambda abstractions:

- λx which creates a function that takes an argument x
- λy which creates a function that takes an argument y

When this function is applied to two numerical arguments, such as 3 and 2, the operation proceeds as follows:

$$\begin{aligned}\text{Apply the first number: } & (\lambda x.\lambda y.x + y)3 \rightarrow \lambda y.3 + y \\ \text{Apply the second number: } & (\lambda y.3 + y)2 \rightarrow 3 + 2 \\ \text{Evaluate the addition: } & 3 + 2 \rightarrow 5\end{aligned}$$

Hence, the application of the lambda function $\lambda x.\lambda y.x + y$ to the arguments 3 and 2 yields the result 5, which demonstrates basic function creation and application in lambda Calculus.

2.5 Implementation Strategies

While it has been explored in detail in this dissertation, how strict type-checking can be implemented through the Pat language to ensure the correctness and robustness of concurrent programs. In practice, however, the core goal of this project is to actually run these concurrency constructs to ensure that they are not just theoretically correct, but also efficiently executed in real-world environments, demonstrating the utility and effectiveness of the Pat language in dealing with real-world concurrency problems

2.5.1 Scheduling

Scheduling in computing systems refers to the strategic scheduling and management of the order in which jobs are executed. This process is a high-level strategic decision-making aimed at optimising processes and computational tasks in terms of time and resource allocation. In a concurrent environment, it is the responsibility of the scheduler to determine the timing of execution of each task to ensure that processor resources are used efficiently while minimising the task completion time. In distributed systems, the scheduler's responsibility extends to task allocation across different compute nodes, aiming to maximise the utilisation efficiency of each node in the network. Regardless of the type of system, the fundamental purpose of scheduling is to improve the overall throughput capacity of the system. In this framework, control flow management techniques, such as CPS and CEK, precisely the techniques necessary to interrupt and resume process execution when implementing scheduling algorithms. This ability to 'control' the remainder of the process is a core feature of such techniques, as they allow the system to implement precise control and scheduling in small-step reductions, which is essential for the management of complex concurrent computational tasks.

2.5.2 CPS and CEK

Continuation Passing Style (CPS) is a high-level control flow management technique that operates by explicitly passing the unexecuted portion of a program, called a "continuation". In the functional programming paradigm, this approach allows a function not to return a result directly, but to continue execution by accepting another function (i.e., a "continue") and passing it control.

This style is inherently suited to implementing non-blocking and asynchronous operations, and provides a great deal of flexibility in modern programming. Although CPS elegantly handles asynchronicity and non-blocking operations, it introduces the need for concurrent multiprocess control and complex communication patterns in concurrent and distributed system applications. However, CPS applications may make the code structure appear more complex and rigid, which not only challenges the readability of the programme but also increases the maintenance cost.

The Pat language, however, is dedicated to the problem of communication in concurrent and distributed systems, employing mailbox types as its core communication mechanism. In such systems, the communication process is often asynchronous, the order in which messages are delivered is uncertain, and multiple concurrent senders and receivers may be involved. In contrast to CPS, CEK-style abstract machines provide a more detailed division of programme state - Continuation, Environment, and Control - through more detailed Explicit management provides a more structured and flexible framework for dealing with such complex communications. In the context of Pat language applications, where nested evaluation contexts and aliasing issues need to be considered, the CEK machine demonstrates excellent adaptability and handling capabilities, making it ideal for supporting the features of the Pat language.

2.5.3 Implementing Pat in OCaml

OCaml was chosen as the implementation language for the Pat language primarily because of its excellent capabilities in functional programming. OCaml's strong type system and pattern matching capabilities provide an intuitive and flexible approach to data processing and control flow, greatly simplifying the implementation of language features. While languages like Rust also place a high value on efficiency and safety, OCaml's unique functional programming paradigm relies much less on mutable state, and this preference for immutability effectively reduces side effects, making concurrent programming safer and easier to understand. As a result, OCaml is particularly well suited to the development of systems such as compilers and language processors that require a high degree of reliability and clear logic.

3 Requirements

The purpose of this chapter is to provide a comprehensive requirements analysis of the project, including explicit functional and non-functional requirements. The detailed description of the project's core components provides clear guidance for the design and implementation of the project, ensuring that the final product will meet the intended quality standards and performance metrics.

3.1 Functional Requirements

In terms of functional requirements, we focus on the core functions that must be implemented by the interpreter, including syntax parsing, program state control, task scheduling, and memory management, to ensure the basic operation and execution efficiency of the interpreter.

3.1.1 Must Have

Keyword Implementation: Must be able to implement the evaluation of all keywords of the Pat language, enabling the correct interpretation and functionality of these keywords within the language's runtime environment, including but not limited to variable declarations, control flow statements (e.g., if-else, switch), as well as support for basic data types (integers, strings).

Mailbox processing: Realize the creation of mailboxes, message insertion, storage, retrieval, and release functions.

Program state control: Adopts CEK machine to achieve precise control of the execution state of the Pat program to ensure the correct execution of program logic.

Task Scheduling: Implements a round-robin-based task scheduling mechanism to achieve fair task scheduling and support parallel task creation and execution.

Memory management: Although OCaml provides automatic garbage collection, explicit memory management is used in order to implement the Pat language's unique 'Free' guarding feature.

3.1.2 Should Have

Execution flow printing: In debug mode to provide the execution of step-by-step printing to assist developers to debug and understand the program execution process.

Error Handling Mechanism: Design and implement a well-defined error handling mechanism to effectively manage exceptions and runtime errors.

3.1.3 Could Have

Garbage collection optimization: Research and implement optimized garbage collection mechanism to reduce the impact of execution time and improve execution efficiency.

3.1.4 Won't Have

Scheduling Algorithm Research: Does not delve into other scheduling algorithms that might replace the polled scheduling approach, such as EIO that has the capability of parallel IO operations. in order to focus on optimization of the current implementation.

Distributed communication:Does not include the implementation of a distributed communication model, due to the current implementation of mailbox typing not accounting for failure scenarios.

3.2 Non-Functional Requirements

Non-functional requirements relate to the usability, and cross-platform compatibility of the interpreter to ensure that the interpreter not only meets the requirements in terms of functionality, but also provides a good user experience during use.

3.2.1 Must Have

Cross-platform compatibility: The interpreter should be able to run seamlessly on major operating systems including Windows, Linux, macOS to ensure wide availability.

3.2.2 Should Have

Ease of use: Design the interpreter to be friendly to novice users so that users can get started quickly.

3.2.3 Won't Have

IDE Integration: Plug-in support for popular IDEs (e.g. VS Code, IntelliJ) including syntax highlighting, code auto-completion, and quick jumps to definitions is not provided at this stage. This feature is considered a separate project dedicated to enhancing the Pat language development environment.

Debugging tools integration: Not integrated debugging tools , including the lack of support for breakpoint settings, single-step execution and variable status view function. This feature is considered a separate project as well.

4 Design

4.1 Principles of CEK-style Machine

4.1.1 Conception of CEK Machine

The CEK machine, developed by Felleisen and Friedman (1987), is a machine for λ-Calculus that is particularly well suited for describing and enforcing the semantics of functional programming languages. This machine, with its realism and abstraction, provides a framework to show how computers can efficiently execute programs. The basic CEK machine is based on a ternary configuration of the form $\langle\, M \mid \gamma \mid \Sigma\, \rangle$. where

- **M** for Control, the expression currently being evaluated;
- $\boldsymbol{\gamma}$ for Environment, binding free variables;
- Σ for Continuation, which guides the action taken by the machine after completing the evaluation of the current term C.

The concept of small-step reduction is indispensable for understanding the nature of the operation of CEK machines. This approach underlies the λ-Calculus and the theory of computation, and helps to decompose complex expressions into a series of simpler incremental steps. Each step consists of a basic operation or transformation that progressively guides the computation, towards the final result.The CEK machine embodies this approach through its ternary configuration, which meticulously performs small-step reductions, thus providing a clear, traceable pathway through the evaluation process.

Let us consider an expression "let x = 3 in x + 2". On this occasion, we will perform small-step semantics using the CEK machine(Figure 4.1):

$$
\begin{aligned}
&\langle\, \text{let } x = 3 \text{ in } x + 2, \{\}, \text{empty}\, \rangle \\
\to &\langle\, 3, \{\}, \text{let } x = [\,] \text{ in } x + 2\, \rangle \\
\to &\langle\, x + 2, \{x \mapsto 3\}, \text{empty}\, \rangle \\
\to &\langle\, 5, \{x \mapsto 3\}, \text{empty}\, \rangle
\end{aligned}
$$

Figure 4.1: *Example of the CEK evaluation*

In the initial phase, the CEK machine sets the control section to the full expression let $x = 3$ in $x + 2$. At this point, the environment is empty, indicating that no variable binding has been evaluated yet, and the continue section is also empty, indicating that no further computations are waiting to be evaluated. Next, the CEK machine recognizes the variable binding action in the let statement and begins to add the variable x to the environment in preparation for its association with the value 3. During this process, the control section is updated to 3, the environment is updated accordingly to contain the mapping $\{x \mapsto 3\}$, and the expression $x + 2$ is placed in the continue section, waiting for a subsequent computation. When evaluating the computation of $x + 2$, the CEK machine looks up and replaces the variable x with 3 in the environment, and then evaluates an addition operation to obtain the final result 5.

The reason why the Pat language is implemented using the CEK machine is that its syntax adopts a style called 'fine-grain call-by-value'(Levy et al. (2003)). In this style, each subexpression of a computation is designed to have a definite value, e.g., in the expression V + W, both V and W are the result of an explicit call-by-value, rather than an expression that may require a further call-by-value as in the case of M + N, and all subexpressions are explicitly ordered by let bindings. This explicit structure simplifies the implementation of CEK machines, allows us to precisely control the evaluation process of the expression. This is demonstrated by Hillerström and Lindley (2016), whose base machine for the Links language was implemented based on this idea. For example, if we reanalyse the code snippet of Figure 4.1, we first evaluate x, then $x + 2$. This serialization of let bindings allows each step of the evaluation to be explicitly tracked and executed by the CEK machine, thus providing a robust and efficient execution model for computation in functional programming.

4.1.2 Syntax Definitions and Rules

Before proceeding to the operational semantics, a brief introduction to the syntactic structure of the Pat language is in order. Figure 4.2 provides syntactic definitions and rules for Pat, that summarize the structure of the language and the characteristics of the communication mechanism. The first is J, K, representing mailbox types, which describes the form of communication in a mailbox, which can be either sending ($!E$) or receiving ($?E$) messages. E, F describes the types of messages that a mailbox can match, $\mathbb{0}$ for error mailboxes, $\mathbb{1}$ for empty mailboxes, messages with labels **m**, and different composite types.

In the second part is the core syntax, which includes basic types and mailbox types. The type annotation indicates the state, which will be applied to the mailbox J's usage state, and the variables and names f are the identifiers of Pat. The values V, W include variables and constants. Term is the basic building block of the language, including the values V, Let bindings, process creation, mailbox creation, etc. will be elaborated in the next section. Guard(G) defines the states of the mailboxes that can receive messages, including failures(**fail**), releasing mailboxes(**free**), and receiving ($\vec{x}$) messages of message type M. The type environment provides a context for variables and types.

Mailbox types and patterns

Mailbox types	J, K	$::=$	$!E \mid ?E$
Mailbox patterns	E, F	$::=$	$\mathbb{0} \mid \mathbb{1} \mid \mathbf{m} \mid E \oplus F \mid E \odot F \mid \star E$

Types and Values

Base types	C	$::=$	$1 \mid \text{Int} \mid \text{String} \mid \cdots$
Types	T, U	$::=$	$C \mid J$
Usage annotations	η	$::=$	$\circ \mid \bullet$
Usage-annotated types	A, B	$::=$	$C \mid J^{\eta}$
Variables	x, y, z		
Definition names	f		
Definitions	D	$::=$	$\textbf{def}\ f(\overrightarrow{x : A}) : B\{M\}$
Values	V, W	$::=$	$x \mid c$
Terms	L, M, N	$::=$	$V \mid \textbf{let}\ x : T = M\ \textbf{in}\ N \mid f(\vec{V})$
		$\mid$	$\textbf{spawn}\ M \mid \textbf{new} \mid V\,!\mathbf{m}[\vec{W}] \mid \textbf{guard}\ V : E\{\vec{G}\}$
Guards	G	$::=$	$\textbf{fail} \mid \textbf{free} \mapsto M \mid \textbf{receive}\ \mathbf{m}[\vec{x}]\ \textbf{from}\ y \mapsto M$

Type Environments

Type environments	Γ	::=	$\cdot \mid \Gamma, x : A$

Figure 4.2: *Pat Syntax definitions(Fowler et al. (2023))*

4.1.3 Operational Semantics

The introduction of Environment is an essential improvement when exploring the operational semantics of the Pat language. In previous operational semantics design by Fowler et al. (2023), the substitution of variables was usually accomplished through substitution operations, which required traversing the entire expression to replace all instances of the variable whenever a variable was used. This process is not only inefficient because it may involve multiple traversals of the entire expression tree, but it may also lead to the creation of multiple unnecessary copies. The introduction of environments optimizes this computational process and brings significant improvements at both theoretical and implementation levels.

As can be seen in Figure 4.3, the CEK machine makes important changes to the runtime syntax of the previous version, with environments forming part of the core composition, in a model in which environments are not just static structures storing variable bindings, but rather act as dynamic, queryable and updatable records, in stark contrast to the traditional substitution operation. contrasting with traditional substitution operations. Specifically, an environment (γ) is defined as a possibly empty sequence in which each entry is associated with a mapping from a variable to its value. Where "$\cdot$" denotes an empty environment, "$\gamma, x \mapsto V$" denotes an extended environment that maps the variable x to the value V on top of the existing environment γ.

Moreover, Σ is consisted of a series of continuation frames $[\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_n]$, each frame $\boldsymbol{\delta} = (\boldsymbol{x}, \boldsymbol{\gamma}, \boldsymbol{M})$ representing a pure continuation within a particular processor closure. Here $\boldsymbol{x}$ denotes the processor closure (handler closure), $\boldsymbol{\gamma}$ maintains its role as an environment for mapping variables to their values, and $\boldsymbol{M}$ represents the current computational term or expression being evaluated. This design was inspired by Hillerström and Lindley (2016), even though **Yield** functionality is not usually provided directly in functional programming paradigms, the model effectively implements yield-like behaviour. This is particularly important for thread scheduling in Pat. In a concurrent environment, **Yield** allows a thread to actively release processor resources, thus allowing the scheduler to allocate resources to other threads or tasks. The continuation part of the CEK machine Σ provides the basis for implementing this operation. Each continue frame can be thought of as a pause point that contains all the information needed to resume execution, similar to the pause and resume mechanism of the yield operation.

This Runtime syntax also involves other concepts to these to precisely describe the behaviour of the program. Runtime names are dynamically generated during program execution to uniquely identify mailboxes, which are an integral part of concurrent communication. Names, on the other hand, such as variables or runtime names, are identifiers used in program code to refer to specific values or mailbox. Guard contexts are structures used in particular to locate the parts of the computation to be governed, which is a key aspect of the application of rules in concurrent computation. Configurations represent the running state of a program, including the items being evaluated, the Frame Stack, and the Environment. Finally, Runtime type environments record the mapping of variables to types, providing support for the enforcement of type checking and type safety.

In addition to the syntactic elements mentioned above, the operational semantics defines how expressions are reductive within the constraints of specific rules. where E-Let first evaluates M, binds the result to x, and adds this new binding to the environment γ. It then evaluates N in the updated environment γ. E-Return, on the other hand, is used when a value needs to be returned from the current frame to the parent frame (x, γ, M), where x will point to the return value V and continue evaluating M in the environment γ. E-App stands for the rule of function application. When a function f is applied to a set of arguments $\overrightarrow{V}$, it replaces the formal parameter with a real parameter and puts it into the environment γ and evaluates in that environment.

In the handling of concurrent operations, the E-New rule generates a unique runtime mailbox address a. The E-Send rule will send a message to mailbox a consisting of the label m (type restriction) and a set of values $\overrightarrow{V}$ creating the configuration for sending the message. E-Spawn will generate a new concurrent computation process that performs the computation M and inherits the parent process's environment. And E-Free allows a mailbox named 'a' to be garbage collected. This will remove references to that mailbox from all processes. For E-Recv will handle the operation of receiving messages from the mailbox. Match the corresponding to receive statement to map the received value to the environment.

Furthermore, the management rules provide the basic structure of concurrent computation. E-Nu allows reductions under name restrictions, while E-Par ensures that each parallel branch can be independently reductions. E-Struct, on the other hand, uses structural equivalence to rewrite configurations before and after reductions to conform to the needs of the reductions rules. These management rules, while not directly related to the core computation of the language, are the basis for describing the state transitions of the system, ensuring that the language behaves consistently and predictably in concurrent environments.

Runtime syntax

Runtime names		a
Names	$u, v, w ::=$	$x \mid a$
Environments	$\gamma ::=$	$\cdot \mid \gamma, x \mapsto V$
Frames	$\sigma ::=$	$\langle x, \gamma, M \rangle$
Frame stacks	$\Sigma ::=$	$\epsilon \mid \sigma \cdot \Sigma$
Guard contexts	$\mathscr{G} ::=$	$\overrightarrow{G_1} \cdot [\,] \cdot \overrightarrow{G_2}$
Configurations	$\mathscr{C}, \mathscr{D} ::=$	$⦇ M, \gamma, \Sigma ⦈ \mid a \leftarrow \mathbf{m}[\overrightarrow{V}] \mid \mathscr{C} \parallel \mathscr{D} \mid (\nu a)\, \mathscr{C}$
Runtime type environments	$\Delta ::=$	$\cdot \mid \Delta, u : T$

$\boxed{\mathscr{C} \longrightarrow_{p} \mathscr{D}}$

Reduction rules

E-Let	$⦇ \mathbf{let}\ x : T = M\ \mathbf{in}\ N, \gamma, \Sigma ⦈$	$\longrightarrow$	$⦇ M, \gamma, \langle x, \gamma, N \rangle \cdot \Sigma ⦈$
E-Return	$⦇ V, \gamma', \langle x, \gamma, M \rangle \cdot \Sigma ⦈$	$\longrightarrow$	$⦇ M, \gamma[x \mapsto V], \Sigma ⦈$
E-App	$⦇ f(\overrightarrow{V}), \gamma, \Sigma ⦈$	$\longrightarrow$	$⦇ M, \gamma[\overrightarrow{x} \mapsto \overrightarrow{V}], \Sigma ⦈$
			(if $\mathscr{P}(f) = \mathbf{def}\ f(\overrightarrow{x : A}) : B\{M\}$)
E-New	$⦇ \mathbf{new}, \gamma, \Sigma ⦈$	$\longrightarrow$	$(\nu a)(⦇ a, \gamma, \Sigma ⦈)$ (a is fresh)
E-Send	$⦇ a\, !\, \mathbf{m}[\overrightarrow{V}], \gamma, \Sigma ⦈$	$\longrightarrow$	$⦇ (), \gamma, \Sigma ⦈ \parallel a \leftarrow \mathbf{m}[\overrightarrow{V}]$
E-Spawn	$⦇ \mathbf{spawn}\ M, \gamma, \Sigma ⦈$	$\longrightarrow$	$⦇ (), \gamma, \Sigma ⦈ \parallel ⦇ M, \gamma, \varepsilon ⦈$
E-Free	$(\nu a)(⦇ \mathbf{guard}\ a : E\{\mathscr{G}[\mathbf{free}\ \mapsto M]\}, \gamma, \Sigma ⦈)$	$\longrightarrow$	$⦇ M, \gamma, \Sigma ⦈$

E-Recv $⦇\ \mathbf{guard}\ a : E\ \{\ \mathscr{G}\ [\mathbf{receive}\ \ \mathbf{m}[\overrightarrow{x}]\ \ \mathbf{from}\ \ \mathrm{y} \mapsto \mathrm{M}]\}, \gamma, \Sigma\ ⦈ \parallel \mathrm{a} \leftarrow \mathbf{m}\ [\overrightarrow{V}]$

$\longrightarrow\ ⦇\ \mathrm{M}, \gamma[\overrightarrow{x} \mapsto \overrightarrow{V}, \mathrm{y} \mapsto \mathrm{a}\ ], \Sigma\ ⦈$

$$\text{E-Nu}\ \frac{\mathscr{C} \longrightarrow \mathscr{D}}{(\nu a)\mathscr{C} \longrightarrow (\nu a)\mathscr{D}} \qquad \text{E-Par}\ \frac{\mathscr{C} \longrightarrow \mathscr{C}'}{\mathscr{C} \parallel \mathscr{D} \longrightarrow \mathscr{C}' \parallel \mathscr{D}} \qquad \text{E-Struct}\ \frac{\mathscr{C} \equiv \mathscr{C}' \quad \mathscr{C}' \longrightarrow \mathscr{D}' \quad \mathscr{D}' \equiv \mathscr{D}}{\mathscr{C} \rightarrow \mathscr{D}}$$

Figure 4.3: *Pat operational semantics*

4.2 Pat Scheduling

The CEK machine provides an execution-level support and implementation framework for the scheduling mechanism of the Pat language. We select the round-robin scheduling mechanism as Pat's fair scheduling strategy, which centres on ensuring that all threads are given an equal amount of processor time, thus preventing any thread from occupying CPU resources for a long period of time. In Pat, this mechanism is improved by changing the time to steps to ensure that all threads get the same number of evaluation steps. This strategy effectively prevents threads from waiting indefinitely after a message is delivered. Once the evaluation steps of a thread reaches the preset upper limit, the system will automatically switch to the next thread in the queue, which ensures the timeliness of message processing between threads and avoids the problem of message backlog.

Figure 4.4 shows a simplified example of the Pat scheduling mechanism, where the active steps of each process are represented by the length of the colour slice, and different coloured slice are used to distinguish between processes. In this model, the maximum value of the step slice is set to 2, which means that after any process executes two steps, the scheduler will force it to move to the next process. The diagram clearly shows the state of the four processes (P1 to P4) during the execution steps. Process P1 reaches the maximum step slice limit after two steps of execution. Process P2 still has one step left to complete after the second step, at which point the scheduling mechanism interrupts P2 and switches to the execution of P3. After P3 completes, the system returns to complete the remaining steps of P2. Finally, P4 is executed similarly to P1. This allocation model ensures that the processes are executed sequentially in a fixed number of steps, reflecting the fairness of the rotation scheduling strategy and ensuring that no process will monopolize the processor resources for a long time.

Process\Step	1	2	3	4	5	6	7	8	9
P1	🟦	🟦							
P2			🟥	🟥			🟥		
P3					🟩	🟩			
P4								🟨	🟨

Figure 4.4: *Pat's Round-Robin scheduling*

4.3 Memory Management

4.3.1 The Imperative of Garbage Collection

Garbage collection is a core feature of automatic memory management in programming languages, designed to identify and free memory that is no longer being used by a program. This mechanism is essential for preventing memory leaks, improving software performance, and avoiding problems such as memory exhaustion. Especially in managed languages such as OCaml, the garbage collection process of objects is mostly transparent, however, the unique semantic properties of the Pat programming language, in particular its unique '**Free**' guard condition, make it necessary to adopt a special garbage collection strategy. This strategy ensures that the release of a mailbox is triggered only when no process references the particular mailbox.

Listing 4.1 shows a specific example in the Pat language, where the `main` function serves as an entry point for the function, first instances a `coin` and passes it as a parameter to the function `flip` in the newly spawned process, and later on also passes it as a parameter to the `output` function. In that function, the mailbox is guarded until a message is received. At this point, the `flip` function in the other process will recursively send a message('`flip`') of type `M`, to the `coin` mailbox based on the value of a random number, until `random()` is greater than or equal to 5, then the recursion will end. When the `output` function receives the message, it matches whether it is of type `M` or not, and then prints the message '`flip`'. And it also calls itself recursively. In this case, if there is no explicit garbage collection, the `coin` mailbox will wait forever for new messages until the end of the program. So in Pat if `self(coin)` has been freed (i.e., there are no more references to it), the `free → ()` branch is executed, freeing the `coin` mailbox. This means that once the free guard is triggered, garbage collection will reclaim `self(coin)` and ensure that `coin`'s lifecycle is over. In the next section, we will discuss the currently used garbage collection mechanism in detail.

```
interface Coin {
   M(String)
}

def random(): Int {
   rand(10)
}

def flip(x: Coin!) : Unit {
   if (random()<5) {
      x ! M("flip");
      flip(x)
   } else {
      ()
   }
}

def output(self: Coin?) : Unit {
   guard self: *(M + 1) {
      free -> ()
      receive M(i) from self ->
         print(i);
         output(self)
   }
}

def main(): Unit {
   let coin = new[Coin] in
   spawn { flip(coin) };
   output(coin)
}

main()
```

Listing 4.1: Example of the necessity of garbage collection in the Pat

4.3.2 Garbage Collection Strategies in Pat

The garbage collection mechanism in the Pat language(Fig 4.5), inspired by the Mark-and-Sweep and Reference Counting methods, has been improved to manage memory resources efficiently. In the context of the Pat language, garbage collection of mailboxes requires traversing the state to ensure that all potential references have been considered. When the system detects that a process needs to block - usually because it is guarding a mailbox and waiting for a message, the garbage collector initiates a state traversal. This traversal consists of examining the environment of all processes to determine if a reference to the mailbox exists. If it exists, the reference counting method adds a reference count to that mailbox in a dictionary structure. The maintenance of this reference count ensures that the mailbox will not be released while at least one process may access it in the future.

When subsequent checks require the same process to be blocked again, the garbage collector will act based on the reference count of the mailbox in the dictionary. If the reference count is greater than zero, the garbage collector will block the process without having to do a full traversal, which is a significant efficiency improvement. The garbage collector will only perform a further full traversal if the reference count goes to zero, i.e., if the mailbox is not referenced in any process environment. If the reference count of the mailbox is **still zero**, meaning that it is no longer being guarded or used by any process, the garbage collector releases (frees) the memory occupied by the mailbox.

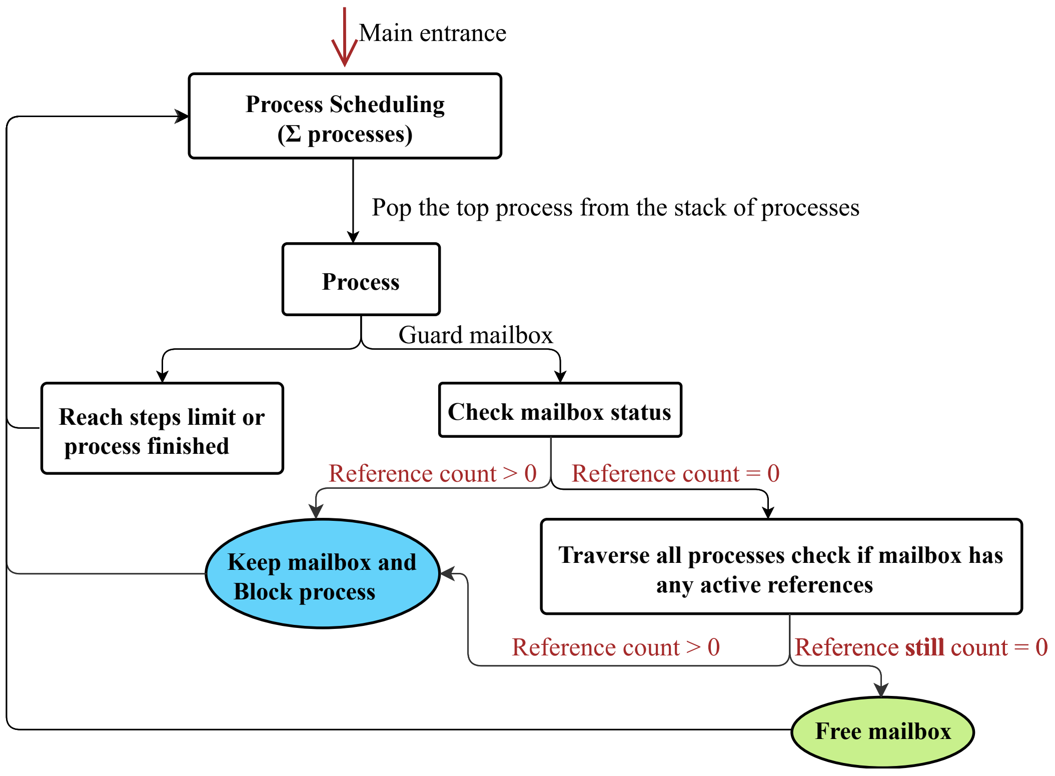
Main entrance
Process Scheduling (Σ processes)
Pop the top process from the stack of processes
Process
Guard mailbox
Reach steps limit or process finished
Check mailbox status
Reference count > 0
Reference count = 0
Keep mailbox and Block process
Traverse all processes check if mailbox has any active references
Reference count > 0
Reference still count = 0
Free mailbox

Figure 4.5: Garbage collection workflow in process Scheduling

In summary, the Pat language effectively manages memory resources by combining a garbage collection strategy with a similar idea of Mark-Sweep and Reference Counting methods, while avoiding the performance loading that frequent traversal may bring. The adoption of this strategy provides an efficient and safe solution for memory management in concurrent programming scenarios, guaranteeing the reasonable utilization of resources and stable execution of programs.

5 Implementation

5.1 CEK in Pat

5.1.1 Definition

When implementing the Pat language interpreter based on the CEK machine, the main concern was to simulate the control flow in the execution of the program, the binding of environment variables, and the management of the continuation of the execution. Based on the source code provided by Listing 5.1, we can see that processes are represented by tuples consisting of several components, including the entire `program`, `process ID`, the number of `steps`, the `computation expression`, `environment`, and the `frame stack`.

The **Control** in the CEK corresponds to the `comp` section of the code, which represents the currently pending computational expression. the **Environment** corresponds to the `(Ir.Binder.t * Ir.value)` list in the code, which is an environment list where each element is a binding that places a variable name. **Continuation** corresponds to the code's `frame list`, which is a frame stack in which each frame is a three-part tuple containing a variable binding `(Binder.t)`, an environment (also a list of variable-to-value mappings), and a computational expression to be executed.

This represents the framework for the concrete implementation of the CEK machine in the Pat language interpreter. According to this framework, all subsequent code of the interpreter will follow this structure for the evaluation and execution of the program, ensuring that each step of the computation will be in the correct context and that the program flow can be properly controlled according to the execution path.

```
type process = program * pid * steps * comp *
                                          environment * frame_stack
and pid = int
and steps = int
and environment = (Ir.Binder.t * Ir.value) list
and frame = Frame of Binder.t * environment * comp
and frame_stack = frame list
```

Listing 5.1: *Source code of CEK framework in Pat*

5.1.2 Implementation of Keywords

Specifically, depicted in Listing 5.2 is an Abstract Syntax Tree (AST) for the Pat language, a structure that provides the underlying framework for the interpreter execution discussed in the current dissertation. In this syntax tree, the evaluation of all keywords is realized through the CEK mechanism. Such as the `Return` keyword that marks the end of a computation, while `Let` and `LetPair` are concerned with variable binding and scope management, keeping the environment consistent. Composite structures such as `Seq` and `Case` show how the interpreter handles sequential and conditional branching, while `App` demonstrates the handling of function applications. More advanced control structures, such as `Spawn` and `Send`, deal with concurrency and message passing, showing how the interpreter operates within a broader model of computation, while `Guard` expressions allow for pattern matching as messages arrive, and are key to synchronization and communication in concurrent programming. As for `value`, it includes various data types such as `Constants`,`Variables`, functions (via the `Lam` keyword), and `Mailbox` specialized for messaging, which are the elements that form the basis of program execution. This implementation ensures that the execution of the program not only follows the semantics, but also maintains the precision and consistency of the execution. Building on the above, the later parts of this chapter details the inner workings of some key mechanisms, including how the `Spawn` keyword supports process spawning, how the `Mailbox` enables efficient communication between processes, and how garbage collection is performed to optimize memory usage. The implementation of these mechanisms is critical to understanding the concurrent programming model and how the Pat language manages resources and synchronizes processes.

```
type program = {
   prog_interfaces: Interface.t list;
   prog_decls: decl list; prog_body: comp option}
and decl = {
   decl_name: Binder.t; decl_parameters: (Binder.t * Type.t) list;
   decl_return_type: Type.t; decl_body: comp}
and comp =
   | Annotate of comp * Type.t
   | Let of {binder: Binder.t; term: comp; cont: comp}
   | Seq of (comp * comp)
   | Return of value
   | App of {func: value; args: value list}
   | If of { test: value; then_expr: comp; else_expr: comp}
   | LetPair of {binders: (Binder.t * Binder.t);
                 pair: value; cont: comp}
   | Case of {term: value; branch1: (Binder.t * Type.t) * comp;
                           branch2: (Binder.t * Type.t) * comp}
   | New of string
   | Spawn of comp
   | Send of {target: value; message: message;
              iname: string option}
   | Guard of {target: value; pattern: Type.Pattern.t;
               guards: guard list; iname: string option}
and value =
   | VAnnotate of value * Type.t
   | Constant of constant
   | Primitive of string
   | Variable of Var.t * Pretype.t option
   | Pair of value * value
   | Inl of value
   | Inr of value
   | Lam of {linear: bool; parameters: (Binder.t * Type.t) list;
             result_type: Type.t; body: comp}
   | Mailbox of RuntimeName.t
and message = (string * value list)
and guard =
   | Receive of {tag: string; payload_binders: Binder.t list;
                 mailbox_binder: Binder.t; cont: comp}
   | Free of comp
   | Fail
```

Listing 5.2: *AST in Pat(Fowler and Attard (2023))*

5.2 Scheduler interaction

In the implementation of Pat language, the core function of scheduler is introduced with the aim of providing an efficient and reliable concurrency mechanism for program running. Unlike the traditional round-robin scheduling mechanism based on time slices, we adopt a rotation scheduling strategy based on evaluation steps in the Pat language. Instead of scheduling tasks in terms of time, this strategy rotates tasks based on the number of evaluation steps they have completed. And we limit the maximum number of evaluation steps to 30. This is mainly for preliminary testing purposes, and aims to balance the execution efficiency and computational complexity of the language. It should be clear that this limit is not fixed, but is an adjustable parameter. As the language is further developed and application scenarios are expanded, this limit will be adjusted according to specific needs.

In the code shown in Listing 5.3, the `process_scheduling` function takes two arguments: `processes` is a list containing all the process to be executed, and `max_steps` is the maximum number of steps each process is allowed to execute before being suspended. The logic of the function starts with a pattern match against the processes list. If the list is empty, all processes have been completed, which means the function is over. Otherwise, it takes the first process from the list and checks to see if the process has reached or exceeded its maximum number of allowed steps. If so, the process will be placed back at the end of the list and the number of steps reset to 0 so that it can be scheduled again later. If the process has not reached its maximum number of steps, it will be executed via the execute function. Execute returns a tuple containing the new `status` of the process and an `updated process`. Based on the status of the process after execution, the scheduler decides how to process next:

- `Finished`: If a process has completed, it is removed from the process list, and the scheduler proceeds to handle the remaining processes.
- `Unfinished`: If a process is not finished(Reaching the maximum number of steps or other reasons), its updated state is appended to the end of the process list for future execution.
- `Spawned new_process`: If the execution spawns a new process (new_process), this new process is added to the beginning of the process list, while the updated state of the current process is appended to the end.
- `MessageToSend (target, message)`: If the current process needs to send a message to another process, the scheduler handles the message sending through the function.
- `Blocked need_free_check`: If a process is blocked due to waiting for resources, some functions are responsible for managing the blocked state.
- `FreeMailbox mailbox`: If a process releases a mailbox, the function takes care of the logic after freeing the mailbox.

Through this approach, the scheduler can effectively manage the execution of processes, concurrent spawning of processes, message passing, process blocking, and resource release, ensuring efficient system operation and rational resource allocation. The implementation of this scheduling strategy not only enhances the performance of concurrent programs but also achieves flexible adaptation to the varying computational demands of different tasks by dynamically adjusting the order and frequency of process execution.

```
let rec process_scheduling processes max_steps =
 match processes with
 | [] -> ()
 | (prog, pid, steps, comp, env, stack) :: rest ->
   if steps >= max_steps then process_scheduling (rest @
                 [(prog, pid, 0, comp, env, stack)]) max_steps
   else
     let (status, updated_process) =
       execute (prog, pid, 0, comp, env, stack)
     in
       match status with
       | Finished ->
         process_scheduling rest max_steps
       | Unfinished ->
         process_scheduling (rest @ [updated_process]) max_steps
       | Spawned new_process -> (*......*)
       | MessageToSend (target, message) -> (*......*)
       | Blocked need_free_check -> (*......*)
       | FreeMailbox mailbox -> (*......*)
```

Listing 5.3: *Implementation of a round-robin scheduler based on evaluation steps in the Pat language*

5.3 Implementing Concurrency and Communication

5.3.1 Process Spawn

Code snippet 1 in Listing 5.4 shows the specific code section that handles the generation of a new process by a process. When a `Spawn` operation is encountered, the system first generates a new process identifier(PID) through the `generate_new_pid()` function, which ensures that each generated PID is unique, incremented sequentially starting from 1, and ensures that the IDs of processes that have been marked as `Finished` are not reused. Subsequently, the newly created process is initialized with a new `PID`, a `step` reset to 0, and `comp`, `env`, and an `empty stack` inherited from the current process, as a way to ensure that the new process can continue to perform the computational tasks split from the current process. For the current process, its computation expression is updated to `Return (Constant Unit)`, which is designed to allow the current process to simply terminate its recursive execution early. This design ensures that the current process can quickly return to the scheduler, which in turn allows the scheduler to continue executing the newly spawned process.

In the process_scheduling function depicted in Listing 5.3, the returned Spawn state is precisely captured. At this point, the second code snippet in Listing 5.4 detailed this status, the scheduler places the newly spawned `new_process` at the top of the process list by matching the state of the current process, while the current process, which failed to complete its execution, is relocated to the end of the process list. This operation ensures that the newly spawned process gets immediate scheduling so that it can start executing as soon as possible, and also ensures that the original process can continue executing in subsequent scheduling rounds if it has not completed its computational tasks. This mechanism ensures that every process gets scheduled at the right time, enabling efficient and fair resource allocation.

```
let rec execute (program,pid,steps,comp,env,stack) =
   (*......*)
   | Spawn comp, env, stack ->
      let new_pid = generate_new_pid () in
      let new_process = (program,new_pid, 0, comp, env, []) in
       Spawned new_process, (program, pid,steps+1,
                             Return (Constant Unit), env, stack)

let rec process_scheduling processes max_steps =
   (*......*)
   | Spawned new_process ->
      process_scheduling (new_process :: rest @
                           [updated_process]) max_steps
```

Listing 5.4: *Process spawning mechanism in Pat*

5.3.2 Mailbox Communication

The mailbox communication feature is a key feature in the Pat language, providing an elaborate mechanism for inter-process message passing. This section describes the process of sending and receiving messages separately.

Sending Messages

Listing 5.5 looks at how the send message operation is handled in the process. `Send` in the code snippet first constructs a `MessageToSend` status that includes the target mailbox variable (`target`), the message variable to be sent (`message`), and the actual value in its environment (`env`). It then updates the state of the current process by increasing its step count by one and updating the computational expression to `Return (Constant Unit)`, which indicates that the current process is expected to end its execution path.

Further, in the scheduler, the `MessageToSend` status is processed with the appropriate substitution and preparation of the destination address and message content via the `substitute_in_message` function to ensure that the message can be correctly sent to the specified mailbox. Afterwards, the `add_message_to_mailbox` function is called to place the prepared message into the appropriate mailbox, and this operation may unblock the process waiting for the message on that mailbox. The scheduling process then proceeds with the remainder of the execution by rescheduling unblocked processes as a result of message reception into the execution queue, followed by placing the current process at the head of the list.

Receiving Messages

In this receive message section(Listing 5.6), when a process guards a mailbox, it first determines if the target is a Mailbox by using the `eval_of_var` function, and if it is, then it looks up the message list corresponding to the mailbox name. If a message exists in the corresponding message list, it iterates through the receive conditions defined in the guards. For each receive condition, check if there is a message in the message list that matches the condition. If it exists, the message is extracted, the message content is bound to an environment variable, and the corresponding to continue expression (`cont`) is executed. If no matching message is found in the message list, the current process is marked as `Blocked`, indicating that it is waiting for a specific message to arrive.

```
let rec execute (program,pid,steps,comp,env,stack) =
   (*......*)
   | Send {target; message; _}, env, stack ->
      (MessageToSend (target, message, env), (program, pid,
                      steps+1, Return (Constant Unit), env, stack))

let rec process_scheduling processes max_steps =
   (*......*)
   | MessageToSend (target, ((tag, _) as message),env'') ->
      let _,messages = message in
      let (substituted_target, substituted_values) =
                  substitute_in_message env'' target messages
      in let unblocked_process = add_message_to_mailbox
                substituted_target (tag,substituted_values) pid
      in process_scheduling ([updated_process] @ rest @
                                  unblocked_process) max_steps
```

Listing 5.5: Message sending in Pat

```
let rec execute (program,pid,steps,comp,env,stack) =
   (*......*)
   | Guard {target; pattern; guards; _}, env, stack ->
     (match pattern with
       | Type.Pattern.One -> (* Free Mailbox *)
       | _ ->
        let need_free_check = (*......*) in
        let m = eval_of_var env target in
        match m with
          | Mailbox mailbox_name ->
             match Hashtbl.find_opt mailbox_map mailbox_name with
             | Some msg_list ->
              let rec match_guards = function
                | Receive {tag; payload_binders; mailbox_binder;
                   cont}
                                                        :: rest ->
                   if (* tag = (tag in msg_list) *) then
                    let message_to_process =
                       extract_message tag mailbox_name msg_list
                            in
                    let new_env = bind_env message_to_process
                       payload_binders env target mailbox_binder in
                    execute(program,pid,steps+1,cont,new_env,stack)
                   else
                    match_guards rest
                | [] ->
                 (Blocked (need_free_check, mailbox_name),
                        (program, pid, steps, comp, env, stack))
                | _ :: rest -> match_guards rest
              in
              match_guards guards
```

Listing 5.6: Message reception and guard evaluation in Pat

5.4 Execution of Garbage Collection Strategies

In the Pat language, to implement its unique "Free" guarding feature, we use a garbage collection mechanism. This feature has three cases:

- When guard matches the pattern 𝟙 or `Free V` appearing in the receive statement (which is syntactic sugar for the former), it means that the mailbox is empty, and therefore the mailbox can be freed directly. Specifically, according to Listing 5.6 when a match is made to `Type.Pattern.One`, the system returns `freeMailbox` (the message of the empty mailbox) to the scheduler. The first part of Listing 5.7 shows the garbage collection process, in which all processes and references in their environments related to this mailbox are removed and the updated process is returned to the scheduler.
- If the guard does not contain `Free V` or does not contain the pattern 𝟙, the system will not perform a garbage collection operation (`need_free_check` is false in listing 5.6), but instead, the process will be added directly in the scheduler to the `blocked_list` and wait for further processing.
- In addition to the above two cases, the system will perform the most important garbage collection check. When `need_free_check` is true, all processes will be traversed in `Blocked` status in the scheduler (second part of Listing 5.7) by `mailbox_counting_update`. This is a simplified version of the evaluation iterator, which does a quick traversal of all processes and evaluates each keyword. There are four cases where mailbox counting is increased, first when the mailbox is returned as a value, next when it is passed as a parameter to a function, and also when there is a message ready to be sent to the mailbox, or when the message already exists in the mailbox. If a reference is found after iteration, it will be directly blocked. If the reference is still not found, the same process is performed as in the first case.

```
let rec process_scheduling processes max_steps =
   (*......*)
   | FreeMailbox mailbox ->
      let new_processes = update_processes_after_free
                                   (updated_process::rest) mailbox
      in
      process_scheduling new_processes max_steps
   | Blocked (need_free_check,mailbox)->
      let should_block =
        if need_free_check then
          let all_processes = rest @
            (Hashtbl.fold (fun _ process acc -> process :: acc)
                                          blocked_processes []) in
            if mailbox_counting_update mailbox all_processes then
                true
            else (* Still no reference, free mailbox *)
        else true
      in
      if should_block then
         add_process_to_blocked_list mailbox updated_process;
         process_scheduling rest max_steps
      else ()
```

Listing 5.7: Mailbox free and process blocking

5.5 Evaluation Step Printer

In order to achieve transparent observation during the evaluation process of the Pat language and to easily be debugged, this research implements a step printer whose core function is to capture and print the current execution state at each evaluation step. Specifically, by calling the `print_config` function defined in Listing 5.8, the current number of evaluation steps is first recorded via counter to allow for tracking the progress of the execution process. Next, the function constructs and splices the status strings of different components separately, including the identifier of the current process, the number of steps, the description of the computation expression, the environment variables, the contents of the execution stack, and the current status of the mailbox mapping and blocking processes. This mechanism not only provides developers with real-time execution feedback, but also greatly simplifies the debugging process and performance analysis by accurately displaying the state changes at each step. In addition, the listing shows that the printer will specifically label the states such as the end of the process and blocking, further increasing the visualization and readability of state changes. With these detailed outputs, it is easier to locate problems and optimize concurrency strategies and resource management.

```
let print_config (comp, env, stack, steps, pid, mailbox_map,
                  blocked_processes) =
  counter := !counter + 1;
  let step_str = Printf.sprintf "\n Total step %d\n" !counter in
  let mailbox_map = print_mailbox_map mailbox_map in
  let blocked_processes =
        print_blocked_processes blocked_processes in
  let steps_str = Printf.sprintf "\n\nCurrent PID:
                      %s Steps: %d\n\n" string_of_int pid steps in
  let comp_str = Printf.sprintf "Comp: %s\n\n" (show_comp comp) in
  let env_str = Printf.sprintf "Env: %s\n\n" (show_env env) in
  let frame_stack_str = Printf.sprintf "Frame Stack: %s\n"
                                       (show_frame_stack stack) in
  step_str ^ mailbox_map^ blocked_processes ^ steps_str ^
             comp_str ^ env_str ^ frame_stack_str
(*......*)
Buffer.add_string steps_buffer
       (Printf.sprintf "\n Process %d Finished \u{2705} \n" pid);
(*......*)
Buffer.add_string steps_buffer
       (Printf.sprintf "\n Process %d Blocked \u{1F6AB} \n" pid);
```

Listing 5.8: Evaluation step printer

Listing 5.9 shows a concrete example of the output of the step printer during the execution of the Pat language, where the output first shows the mailbox count status. This is followed by a detailed listing of the state of all mailboxes currently in the system, including mailbox identifiers (e.g.,`alice0, bob1, self2`) and a list of the messages each contains. This section provides an intuitive view for understanding inter-process communication and is especially important when debugging message passing and synchronization issues. A list of currently blocked processes is also shown, including the mailboxes waiting for messages and their corresponding PIDs. this information helps developers to quickly identify potential deadlocks or resource contention problems. Finally, a full picture of the current execution context is provided, including inter-process function calls, variable bindings, and stack frame changes.

```
Mailbox_counting:
   Mailbox: self2, Value: 1
------------------ Total step 22 --------------------
Mailbox: alice0, Messages: [(Debit, [(Ir.Constant
          (Common_types.Constant.Int 20)),(Ir.Mailbox future3)]);]
Mailbox: bob1, Messages: []
Mailbox: self2, Messages: []
Blocked process:
  Mailbox: self2 -> PID: 1
  Mailbox: alice0 -> PID: 2
Current PID: 3 Steps: 2
Comp: Ir.App {func = (Ir.Variable (await4, (Some (Int,
                     AccountMb!(Debit) [U]) -> FutureMb)));
              args = [(Ir.Variable (amount29, (Some Int)));
                (Ir.Variable (recipient28, (Some AccountMb)))]}
Env: [self30 -> Mailbox bob1; amount29 -> 20; recipient28 ->
      Mailbox alice0; sender27 -> Mailbox self2;
      self22 -> Mailbox bob1; balance21 -> 20]
Frame Stack:
  [Frame(future,[self30->Mailbox bob1;amount29->20;
         recipient28 -> Mailbox alice0; sender27 -> Mailbox self2]
```

Listing 5.9: *Example of evaluation step printer output*

6 Evaluation

In this research, we comprehensively evaluated the effectiveness of the Pat language runtime interpreter, employing methods such as correctness testing and micro-benchmarking, aiming to accurately quantify its performance metrics. Correctness testing ensured that the Pat language executed as expected when processing different example files, verifying the correctness of the language design and runtime. The efficiency of the interpreter in performing computationally intensive tasks was evaluated through careful measurements of the runtime of specific algorithmic tasks (e.g., computing Fibonacci series), and these tests focused on evaluating the performance of the Pat language under real-world running conditions. The tests were performed on a computer equipped with an M2 Pro chip and 16 GB of RAM, with the operating system macOS 14.3, using the OCaml version 5.1.0 environment, and the time calculations, which included all parts of the type checking and the interpreter, were repeated 100 times, thus ensuring the accuracy and reliability of the results.

6.1 Concurrent Reduction

6.1.1 Functional Reduction

In performing a series of fundamental tests on the Pat language, we aim to verify whether the Pat language is able to accurately execute basic programming constructs and how well it performs in different programming scenarios. These tests help to evaluate whether the runtime environment of the Pat language can meet the needs of concurrent programming, and whether it can correctly handle various programming tasks according to its design specifications.

Specific tests performed include basic arithmetic tests, operator precedence, logical operation tests, anonymous functions, local variables and function bindings, empty interfaces and linear functions, type annotations, and pattern matching. The results of the tests show (Table 6.1) that all tests were successfully completed as expected and no runtime errors were found. This result proves that the underlying functionality of the Pat language matches the design specifications and is capable of handling a wide range of programming tasks.

6.1.2 Concurrency Reduction

After the functional correct tests were completed, we further tested the correctness of six advanced concurrency models for the Pat language, based on the original mailbox algorithmic model with ideas taken from the research of de'Liguoro and Padovani (2018). The specific test implementations refer to the research of Fowler et al. (2023). These test cases cover everything from concurrent lock, future variable handling, and account concurrent transactions, account concurrent transactions via future variables, master-worker parallel networks, and session-type communication actor models using a single arbiter.

First, we simulate concurrent locks to verify the efficiency and reliability of the Pat language in handling mutually exclusive operations. In addition, we evaluated the performance of the Pat language in handling one-time write and multiple read operations by testing the "Future"

Name	Description	Exp.	Act.
Arithmetic I	Basic arithmetic check	True	True
Arithmetic II	Operator precedence checks arithmetic order	True	True
Arithmetic III	Logic & arithmetic test combines operations	False	False
Anonymous Function	Function returns input integer value	5	5
Local Binding	Local binding and function definition	10	10
Nested Binding	Nested function returns double input value	10	10
Memory Management	Interface declaration and memory deallocation	( )	( )
Type annotations	Handles union type and printing	5	5
Pattern Matching	Pattern matching with union types	5	5

Table 6.1: *Functional correct test results*

variable. Then, by simulating debit/credit instruction exchanges between concurrent accounts, we further validate the accuracy and efficiency of the Pat language in complex concurrency scenarios. In order to explore the application of Pat in asynchronous programming model, we also test the debit/credit commands of accounts implemented by "Future". In addition, we evaluate the performance of the Pat language by constructing a parallel network of master-worker jobs. Finally, we explore the ability of the Pat language to implement complex communication protocols through a session-type communication actor model, which involves the use of a single arbiter to coordinate communication behaviour.

All tests show that the Pat language is able to successfully handle these high-level concurrent and programming tasks, fully indicating alignment between expected and actual output(Table 6.2). The test results not only validate the performance of the Pat language in different programming scenarios, but also demonstrate its power in handling concurrent system programming requirements. The average response times ranged from 45.8 ms for the concurrent model to 102.9 ms for the session type model, further proving that the Pat language is able to provide a high degree of functionality and reliability while ensuring performance.

Name	Description	Exp.	Act.
Lock	Concurrent lock modelling mutual exclusion	12	12
Future	Future variable that is written to once and read multiple times	55	55
Account	Concurrent accounts exchanging debit and credit instructions	( )	( )
Account Future	Concurrent accounts where debit instructions are effected via futures	( )	( )
Master-Worker	Master-worker parallel network	55385	55385
Session Types	Session-typed communicating actors using one arbiter	6	6

Table 6.2: *Advanced concurrency model correct test results*

6.2 Micro-Benchmark Test

Based on the correctness tests, we designed and implemented a series of benchmark tests aimed at evaluating the performance of the Pat language's runtime interpreter under different concurrent programming patterns. These tests cover a wide range of concurrent programming scenarios including master-worker (e.g., KFork, Fibonacci, Log Map), client-server (e.g., Ping Pong, Counter), and peer-to-peer (e.g., Big), and involve common network topologies such as Star (e.g. Philosopher, Smokers, Transaction) and Ring (e.g. Thread Ring), as referenced from part of the savina benchmarks written by Imam and Sarkar (2014). Each test is designed to evaluate the execution efficiency of the Pat language under a particular concurrency model.

The purpose of these benchmarks is to establish an initial performance baseline for the Pat language and to demonstrate that it can effectively handle multiple concurrency models. Given that the Pat language is the first programming language to integrate mailbox typing, we do not use the results of these tests for direct comparisons with existing functional languages on the market. Instead, our goal is to demonstrate the controlled operation of the Pat language by showing how it can stably execute multiple concurrency patterns based on a design and implementation that incorporates the concept of mailbox types.

Through this series of benchmark tests, we observe the performance of the Pat language under different concurrency models (table 6.3). For example, the Ping Pong test evaluates the message processing speed by simulating the bidirectional delivery of messages; the Thread Ring test measures the message delivery latency by passing tokens in a ring network; and the Fibonacci test demonstrates the efficiency of the Pat language in handling recursive concurrency tasks. Note that in the Smokers test, this design caused a significant increase in test execution time due to the introduction of a randomized sleep function.

In tests performing the K-Fork model, the range of random numbers generated by the rand function in the initial code was too large, thus making the computation process excessively time-consuming. In light of this, we adjusted the upper limit of random numbers from 100,000 ms to 10 ms. This adjustment makes the computation process more efficient and more closely matches the lightweight computation needs of actors processing messages in real application scenarios. In addition, the initial code's multiple message flooding of a single mailbox (100, 1000, and 10000 messages, respectively) actually caused the mailbox to be overloaded. To more realistically simulate mailbox performance under regular load, we reduced the number of floods to a two and fixed the total number of messages to 20. These optimizations improve the accuracy and reliability of the benchmark in evaluating concurrent processing performance.

Name	**Description**	**Avg Time (ms)**
Ping Pong	Process pair exchanging $k = 5$ ping and pong messages	54.5
Thread Ring	Ring network where actors cyclically relay one token with counter $k = 1000$	541.5
Counter	One actor sending messages to a second that sums the count	45.5
K-Fork	Fork-join pattern where a central actor delegates $k = 20$ requests to workers	76.2
Fibonacci	Fibonacci server delegating parallel actors with $k = 5$	56.7
Fib_pairs	Fibonacci actors recursively resolving and returning terms independently	55.1
Big	Peer-to-peer network where actors exchange $k = 100$ messages	265.8
Philosopher	Dining philosophers problem	111.9
Smokers	Centralised network where one arbiter allocates $k = 10$ messages to actors	5578.4
Log Map	Computes the term $xx + 1 = r \cdot xx \cdot (1 - xx)$ by delegating to parallel actors	97.9

Table 6.3: *Part list of savina benchmarks*

In the field of concurrent computation, efficiently managing the overhead of parallel execution is one of the key factors in achieving high performance. By comparing the performance of these two strategies in computing Fibonacci series, we aim to evaluate the efficiency and feasibility of concurrent execution in the Pat language. We have chosen selected a range of K values (from 1 to 10) as input parameters to observe the variation of execution time for different input sizes. As shown in the figure 6.1, the graph on the left side demonstrates the comparative results of Fibonacci computational performance. The curves for sequential execution and concurrent execution closely follow each other, showing the similarity in time complexity between the two execution modes. Notably, Concurrent execution introduces no additional significant overhead, suggesting that the concurrency model of the Pat language has a potential advantage

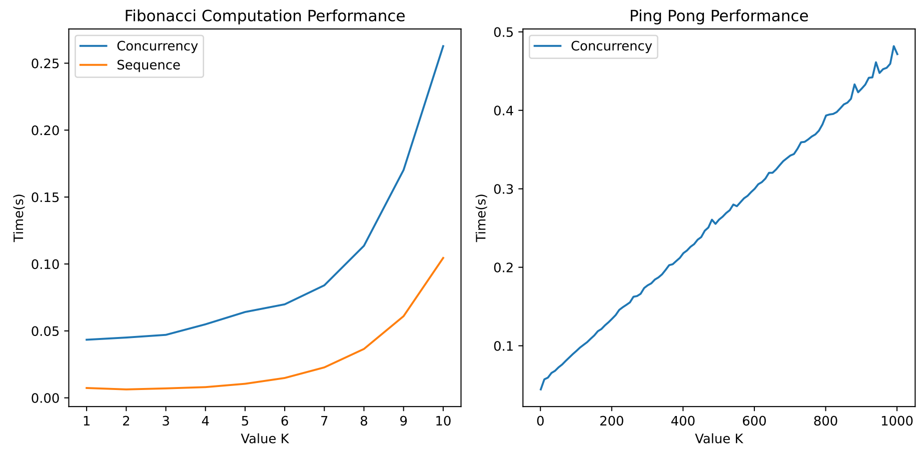

Figure 6.1: *Comparative execution times of sequential and concurrent code snippets in Pat language at various K values*

in maintaining low overhead.

In addition, we consider a Ping Pong performance test as an example of concurrent communication overhead. In this test, we only considered concurrent execution (without the sequential execution model) to demonstrate the linear relationship between execution time and input size. The graph on the right clearly depicts this relationship, a finding that suggests that in the Pat language, even in concurrent tasks that require frequent communication, the execution time remains linear, unaffected by significant concurrency overhead.

In summary, these benchmark tests provide us with a comprehensive evaluation framework to scrutinize the performance of the Pat language in the area of concurrent programming. The unique advantages of the Pat language in handling complex concurrent tasks are demonstrated. This research provides important references and insights for the future development of functional language design and concurrent programming patterns for mailbox types.

6.3 Reflection

In the reflection section, the discussion of the research on garbage collection optimization and scheduling algorithms reveals two key areas in our research and development process, as well as pointing to potential directions for future work.

6.3.1 Garbage Collection Optimization

For garbage collection optimization section, we currently employ a mechanism that similarly combines a simple Mark-Sweep algorithm and a Reference Counting method. This approach provides our system with basic memory management capabilities. Although this combined strategy works effectively in many cases, we also recognize its limitations when dealing with branching statements such as if/switch, and complex programming constructs such as recursion. Especially in recursive calls and complex control flow structures, the simple reference counting method may not be able to accurately identify all garbage objects, resulting in memory not being reclaimed in a timely manner, thus affecting the execution efficiency and stability of the system.

Therefore, given enough time, we plan to explore and implement a deeper reference counting algorithm: Perceus, a precise reference counting method that combines reuse and specialization developed by Reinking et al. (2021). This approach is mainly implemented by delaying the execution of the dup operation and executing the drop operation as early as possible. A dup operation here means increasing the reference count of an object, while a drop operation decreases the reference count. When the reference count drops to zero, the object is released.

In a traditional reference counting system, whenever a variable is copied, the dup operation is executed immediately to increase the reference count of the object. However, this immediate increase in the reference count may not always be necessary, especially if some variables are no longer needed in the short term.The Perceus algorithm increases the reference count by delaying the execution of the dup operation until the reference actually needs to be shared by multiple owners. This strategy reduces unnecessary reference count adjustments. In contrast to delaying execution of the dup operation, the Perceus algorithm performs the drop operation as early as possible. This means that as soon as it is determined that an object is no longer needed, its reference count is immediately reduced, and the object is also freed immediately if the reference count drops to zero. This approach optimizes memory utilization because it ensures that objects that are no longer needed are reclaimed in a timely manner.

In summary, Perceus provides an efficient reference counting mechanism that supports complex control flow and programming constructs through precise memory management, while reducing memory leaks and improving system performance. We therefore believe that Pat's memory management and stability can be significantly improved by introducing this innovative algorithm.

6.3.2 Advanced Scheduling Methods

Regarding the research on scheduling algorithms, our current implementation employs the Round-Robin scheduling approach, which is a simple and straightforward scheduling strategy. However, we also recognize that other framework, such as the EIO (2024) scheduling, which is capable of supporting parallel IO operations, may provide better performance and efficiency(Figure 6.1).The introduction of EIO not only takes advantage of the high-performance IO operation features provided by modern operating systems, such as Linux's io_uring, but also allows concurrent code to be written in a much more concise and efficient manner through a direct style of IO stacking. This approach does a lot more than just reduce heap allocation and increase speed, it also makes concurrent code written in the same style as non-concurrent code, which greatly improves the readability and maintainability of the code.

Within the time constraints of the project, we did not explore these alternatives in depth. Had we had more time, we would have compared the performance of different algorithms under multiple loads and scenarios, and we could have determined the scheduling strategy that best suited our system. In addition, research on how to combine multiple scheduling strategies in order to dynamically select the scheduling algorithm that best suits the current execution environment and task characteristics will also be a focus of our attention.

```
let main _env =
  Fiber.both
    (fun () -> for x = 1 to 3 do traceln "x = %d" x; Fiber.yield
        () done)
    (fun () -> for y = 1 to 3 do traceln "y = %d" y; Fiber.yield
        () done);;
```

Listing 6.1: *Example of running two threads of execution concurrently using Eio.Fiber*

7 Conclusion

In this research, we explore the importance of concurrent and distributed systems in modern computational science. We not only compare concepts, challenges, and case studies related to concurrent programming and distributed systems, but also implement and comprehensively evaluate a concurrent runtime for the Pat language. The Pat language, by introducing advanced concurrent programming primitives and a unique mailbox type system, aims to provide developers with a more intuitive and concise way of handling concurrent tasks and communication processes. The goal of this research is to develop a runtime interpreter that specifically supports the Pat language, which not only needs to fulfil the runtime requirements of a general programming language, such as memory management, but also focuses on the specific needs of concurrent computation.

Starting from the foundations of concurrent and distributed systems, our research work provides a deep analysis of the problems faced by concurrent programming in modern computing environments, such as thread deadlocks, communication mismatches, and data consistency problems. These reveal the technical challenges in designing efficient and reliable concurrent systems.

To address these challenges, we have successfully implemented a runtime interpreter for the Pat language, whose main innovation is the introduction of a mailbox type system that statically constrains the order of message passing and processing through a type-checking mechanism, thus reducing the complexity of concurrent program design and improving program security and reliability. In addition, the Pat language provides a series of high-level concurrency primitives, such as Spawn, Send and Guard, which provide developers with flexible tools to build complex concurrency logic.

Our runtime interpreter for the Pat language employs an execution model based on a CEK mechanism that accurately manages the control flow, environment bindings, and persistent execution state of a program. By implementing a task scheduler based on a rotation algorithm, the interpreter is able to efficiently manage the execution of concurrent tasks and the utilization of system resources, thus guaranteeing high-performance execution of concurrent programs. In addition, through a garbage collection mechanism similar to the one that combines the mark-and-scan algorithm with reference counting, we implement the semantics of 'free'

In order to improve runtime transparency and ease of use, we also implemented detailed step printing and error handling mechanisms. The step-printing feature allows tracking each step of program execution for a better understanding of the concurrent logic's running process, facilitating debugging and performance analysis. The error handling strategy simplifies the problem localization and repair process by catching runtime exceptions and providing clear error messages and failure reasons.

Through a series of tests and evaluations, we verified the effectiveness and efficiency of the Pat language and its runtime interpreter in handling concurrent tasks and communication flows. The test cases we designed cover a wide range of performance benchmarks from basic functionality tests to complex concurrency models, and the results show that the Pat language is able to accurately perform a variety of concurrent programming tasks and exhibits excellent performance in a wide range of test scenarios. These tests not only prove the rationality and usefulness of the Pat language design, but also demonstrate its potential for development in concurrent programming models.

7.1 Future Work

Despite the results of our research, there are still many potential challenges and opportunities for application in practice and further optimization.

First, in future work, the focus will be on further optimization of the runtime interpreter. In particular, in terms of task scheduling and memory management, we plan to explore more efficient methods to improve the execution efficiency of the Pat language. In particular, we will consider introducing the Perceus algorithm(Reinking et al. (2021)), a garbage collection strategy that combines precise reference counting and deferred processing to address memory management under complex control flow and recursive structures.

In addition, we can research how to utilize the high-performance IO operation features provided by modern operating systems, such as EIO (2024), which can support parallel IO operations and may provide better performance and efficiency.

In summary, this research provides a new tool and methodology for the research and development of concurrent programming models by proposing the Pat language and the corresponding runtime interpreter. We believe that with the further development and improvement of the Pat language, it will provide effective solutions to complex problems in the field of concurrent programming and contribute to future technological innovations.

7 Bibliography

G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990. ISBN 978-0-262-01092-4.

I. Amazon Web Services. Amazon web services (aws) - cloud computing services, 2023. URL `https://aws.amazon.com/`. Accessed on March 7, 2024.

D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi: 10.1561/2500000031. URL `https://doi.org/10.1561/2500000031`.

A. Chaudhuri. A concurrent ML library in concurrent haskell. In G. Hutton and A. P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 269–280. ACM, 2009. doi: 10.1145/1596550.1596589. URL `https://doi.org/10.1145/1596550.1596589`.

U. de'Liguoro and L. Padovani. Mailbox types for unordered interactions. In T. D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPICS.ECOOP.2018.15. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2018.15`.

EIO. ocaml-multicore/eio, 2024. URL `https://github.com/ocaml-multicore/eio`. Accessed on March 7, 2024.

M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 193–222. North-Holland, 1987.

W. S. Ford and V. C. Hamacher. Hardware support for inter-process communication and processor sharing. In M. J. Flynn, O. N. Garcia, and D. P. Siewiorek, editors, *Proceedings of the 3rd Annual Symposium on Computer Architecture, Clearwater, FL, USA, January 1976*, pages 113–118. ACM, 1976. doi: 10.1145/800110.803559. URL `https://doi.org/10.1145/800110.803559`.

S. Fowler and D. P. Attard. Type check in pat, 05 2023. URL `https://github.com/SimonJF/mbcheck`. Accessed on March 7, 2024.

S. Fowler, S. Lindley, and P. Wadler. Mixing metaphors: Actors as channels and channels as actors. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 11:1–11:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPICS.ECOOP.2017.11. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2017.11`.

S. Fowler, D. P. Attard, F. Sowul, S. J. Gay, and P. Trinder. Special delivery: Programming with mailbox types. *Proc. ACM Program. Lang.*, 7(ICFP):78–107, 2023. doi: 10.1145/3607832. URL `https://doi.org/10.1145/3607832`.

S. Gay and A. Ravara. *Behavioural Types: from Theory to Tools*. Taylor & Francis, 2017.

J. He, P. Wadler, and P. W. Trinder. Typecasting actors: from akka to takka. In P. Haller and H. Miller, editors, *Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28-29, 2014*, pages 23–33. ACM, 2014. doi: 10.1145/2637647.2637651. URL `https://doi.org/10.1145/2637647.2637651`.

C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973. URL `http://ijcai.org/Proceedings/73/Papers/027B.pdf`.

D. Hillerström and S. Lindley. Liberating effects with rows and handlers. In J. Chapman and W. Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016. doi: 10.1145/2976022.2976033. URL `https://doi.org/10.1145/2976022.2976033`.

R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In J. Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi: 10.1007/978-3-540-70592-5_22. URL `https://doi.org/10.1007/978-3-540-70592-5_22`.

S. M. Imam and V. Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In E. G. Boix, P. Haller, A. Ricci, and C. A. Varela, editors, *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 67–80. ACM, 2014. doi: 10.1145/2687357.2687368. URL `https://doi.org/10.1145/2687357.2687368`.

Y. Kim and K. Kim. Design and evaluation of a MMO game server. In R. Lee, editor, *Computational Science/Intelligence & Applied Informatics, CSII 2018, Yonago, Japan, July 10-12, 2018, selected papers*, volume 787 of *Studies in Computational Intelligence*, pages 69–82. Springer, 2018. doi: 10.1007/978-3-319-96806-3_6. URL `https://doi.org/10.1007/978-3-319-96806-3_6`.

Lambda Calculus. The lambda calculus for absolute dummies (like myself), 05 2015. URL `http://bach.ai/lambda-calculus-for-absolute-dummies/`. Accessed on March 7, 2024.

P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi: 10.1016/S0890-5401(03)00088-9. URL `https://doi.org/10.1016/S0890-5401(03)00088-9`.

I. Microsoft Azure. Cloud computing services | microsoft azure, 2023. URL `https://azure.microsoft.com/en-us`. Accessed on March 7, 2024.

R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

Pokemon GO. Pokémon go, 09 2019. URL `https://en.wikipedia.org/wiki/Pok%C3%A9mon_GO`. Accessed on March 7, 2024.

A. Reinking, N. Xie, L. de Moura, and D. Leijen. Perceus: garbage free reference counting with reuse. In S. N. Freund and E. Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 96–111. ACM, 2021. doi: 10.1145/3453483.3454032. URL `https://doi.org/10.1145/3453483.3454032`.

H. Sutter and J. R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005. doi: 10.1145/1095408.1095421. URL `https://doi.org/10.1145/1095408.1095421`.

World of Warcraft. World of warcraft, 2019. URL `https://wowpedia.fandom.com/wiki/World_of_Warcraft`. Accessed on March 7, 2024.