Title

An inherently concurrent and multi-threaded language with adaptive scheduling runtime and queueing theories applications.

Abstract

This proposal contains an attempt to compile many design ideas of a general-purpose programming language. If some sections may be a bit terse and not logically connected to each other, this is because making a new language or thinking about what a good programming language could look like isn't so easy.

The main purpose of the research is to create a language that simply have inherently concurrent and multi-threading mechanisms. That is a basic idea, little bit weird because all (many) programming languages already provide multi-threading or concurrent systems, and creating a programming language isn't so daunting if you know some of those books: *The dragon book*, *Crafting Interpreters*, *Introduction to Compilers and Language Design*. Nevertheless, more we dig into a programming language specificity, and more we try to make it performance, more it become challenging.

This research primarily design Eniem (name subject to change) based on a runtime with a concurrent and optionally multi-threaded approach. In Eniem classics expressions are fundamentally unordered. It includes lazy evaluation of everything and still go forward in the execution.

That new language aims to be flexible, easy to learn, and contains powerful features, providing opportunities to model and analyze complex systems. That multi-threaded paradigm enable runtime adaptation of execution, job class specifications, that can be used both in a real-world environment for any developments requiring parallelism/concurrency, and in a study environment to experiment some networks in queueing theory.

With that innovating paradigm, Eniem could enable runtime adaptation of the execution. It includes job (expression) class specifications, dynamic reclassification. In addition, Eniem toolchain provides a powerful type checking and security features thanks to an abstract interpreter that can be enabled which makes the code safe and correct.

Project on GitHub Right Now

A first attempt of the prototype is already in development and accessible to public. The GitHub project presentation (README, contributing, documentation) are missing at this time. But some example of code demonstrate a first idea of the syntax of the language.

Pre introduction

Developing a programming language with a classical paradigm of scheduling is already a challenge if you enable some features like recursions, dynamic arrays, type checking, static analysis, memory safety and so on. Making all the program concurrent insert another layer of difficulty: how is it concurrent, what the runtime looks like, is it really efficient and how to implement the memory management, the mutability, et cetera. There is a lot of subject of research in one project to present, so lets dump here, before the real introduction, a simple list of references that inspired this adventure.

- A dependently Typed Assembly Language
- You should make a new terrible programming language
- An Experimental Integration of io_uring and Tokio: An Asynchronous Runtime for Rust
- Rust Atomics and Locks: Low-Level Concurrency in Practice, by Mara Bos
- A History of Haskell: being lazy with class
- Lazy Functional State Threads
- Compiling with Abstract Interpretation

Introduction

In traditional programming language, the more recurrent paradigm is a strict ordering of expressions evaluation. For instance, his is what the front-end of C and C++ propose, and the output generated by such programs reflects what the developer require. In other functional programming languages, like in Rust and Haskell, there is much to tell because they introduce both a parallelism with lazy evaluations. But none primarily target multi-threading or environmental adaptation as the proposed language here.

It has been shown that making evaluation unordered in a program can be very powerful. In many context in high level, the development of a system tends to use threads, coroutines, futures, promises, etc. At low levels, compilers, like GCC aims to reorder some expressions to make generated machine code faster based on the build environment. At a lower level, an out-of-order processor is able to evaluate expressions of the same program if no fences are detected. The implementation of a language where parallel evaluation is the inherent paradigm makes sens, but it can be difficult for the developer to understand what his code actually do.

The language design has to be easy to use to make development of complex system possible. Generally a code is interpreted sequentially, they are several reason for that but one of them is that developers thinks sequentially. If Haskell is known as a hard to learn language because of his features of concurrency and lazy evaluation and so on, in a very wildly used language like JavaScript, concurrency is used in any context by any programmers, beginners to masters. This is because in JavaScript, the concurrency is mono-threaded and managed by a virtual machine which makes accesses to the memory safe. In that kind of context, a developer can built a very complex systems of states machines with a large scale of jobs.

> - Comparative Study of Refactoring Haskell and Erlang Programs
> - Kyle Simpson- You Don't Know JS, Async and Performance

There is a growing need for programming paradigms to have one that inherently implement concurrency and adaptable modeling. Rust and C++ both provides library that are limited by what these language propose to create a program. It's known that bigger a system is, harder it became to change how it works. Many projects suffer of big refactoring because of the dependency of a library. Rust proposal of defining async/await behaviour is not enough and is today controversial.

> - The Rust I wanted had no future
> - Experimental async / await support for Tokio

While multi-threading programming is becoming a wildly used approach to design a system, due to the observed performance and the global ideology spreed by developer's influencers. There is still open issues, partially resolved for fitting with specific cases. The opportunity to use previous mistakes as lesson is very interesting in this research. In real-world applications, promises implementation shown to be quite difficult, and category theory demonstrate another controversial way which finally is considered better than before. Tokio crate faced at stuttering and big latency issues, which were patched but still very rigid.

> - CLR thread pool injection, stuttering problems
> - Jon Gjengset - Decrusting the tokio crate
> - Category Theory Promise/A+
> - Reducing tail latencies with automatic cooperative task yielding

This research aims to design a new language with a complete toolchain (static analyser, interpreter and transpiler into C89) where every expression are fundamentally unordered jobs, written like if it were sequential, with lazy evaluation and inherent parallelism.

Research Questions

1. How to design a language to treat all expressions in any order without sacrificing correctness and linearizability conditions?

The language should be able to recognize pure and impure function to minimize the use of mutexes. Early in the code analysis, the toolchain must recognize static, immutable and mutable variables to mitigate the access overhead to the memory shared between threads.

> - Implementation Strategies for Mutable Value Semantics
> - Pure vs Impure functions
> - Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
> - Rayon data parallelism

2. What mechanisms can allow the runtime system to reorder expression evaluation based on the execution context?

Said that the language in this research aims to be used in real-world and study environment, the runtime has to dynamically reorder expression evaluation. The development of a program must be handful to tell that a job has a higher/less priority than another. Most of the needs here have to be defined after studying the state of art in queueing theory.

> - Making the Tokio scheduler 10x faster

3. How can the language's runtime serve as a realistic environment for modeling and testing queueing systems?

In first part of the design, multi-threading and concurrent systems require such mechanisms like `wait_all` , `then` , `timeout` to be implemented. In the other hand, the toolchain must contains handful tools to compare performances by slightly modifying job classes, runtime configurations and jobs implementations. The code have to not be broken at each refactoring of the global behavior of the system. Finally, insights about program's performances are often hard to get when a project scale and inherent functionalities are big requirements in real-world applications and in study cases.

> - A Product-form Network for Systems with Job Stealing Policies
> - The Gittins Policy is Nearly Optimal in the M/G/k under Extremely General Conditions

Objectives

The goals are:

1. To design a complete prototype of a language and his toolchain with some libraries as network communication, input and outputs, parsing, etc. Directly binded with libc.

2. Develop benchmark's programs with the designed language and other traditional approach (C, C++, Haskell, Rust)

3. After an evaluation of the adaptability/flexibility of the language. Apply different models of queueing theory, analyse and compare scenarios.

> - No-op compiler benchmarking

Methodology

1. Language specification. Define the syntax and the semantics of the language.
2. Create a scalable interpreter capable to read and execute a program.
3. Design some cases studies and benchmarks involving computation tasks and queueing models.
4. Evaluation metrics, parallel execution efficiency, context switching overhead, runtime latency, linearizability and correctness guarantees.

Contributions to computer science

- A new language that primarily use multi-threading, lazy evaluation and unordered computation.
- New techniques for experimental platforms in queueing theory.
- New techniques for real-world scalable applications development.
- Complete toolchain with type checking, abstract interpretation, interpreter and compiler.
- Implementation of benchmarks of queueing networks with job stealing (multiple jobs queues), single queue with/out jobs reordering.
- Insights to the interplay between program evaluation order and system performance.

Timeline

Phases	Duration (Months)
Literature Review & Design	0–6

Phases	Duration (Months)
Prototype Development	7–18
Experimental Design & Testing	19–30
Analysis	31–36
Dissertation Writing	37–42
Defense Preparation	43–48