# Coroutines for a Linearly-Typed Imperative Language

Aryan Wadhwani
Purdue University
wadhwani@purdue.edu

## INTRODUCTION

Performance is crucial for systems programming languages, and bringing support for concurrency design patterns to these languages has been crucial to take advantage of today's multicore processors.
Coroutines in particular are useful for systems languages, as they allow programs to switch between tasks without requiring the overhead of creating and running threads, while also providing better control over how tasks are executed.
Additionally, in the field of Systems languages, there has recently been a push towards guaranteeing safety properties of programs, most notably memory safety. Some languages, like Go, have adopted a garbage collector to guarantee memory safety, but this comes at the cost of performance and runtime predictability. However, there are other languages, like Rust and Austral, that have use a linear type system to guarantee memory safety.
Linear type systems guarantee that a variable is used exactly once, which prevents usage of a variable after it has been freed. Multiple functions may use a reference to the same value, but moving the value to a new variable will invalidate the old variable. This approach allows the language to know when a value is no longer needed, and can therefore free the memory associated with it.
There would be interesting questions about how coroutines could be supported in a linearly-typed language. For example, if a coroutine is suspended, and the value it is holding is moved to a new variable, should the coroutine be able to resume using the new variable? Or should the coroutine be suspended until the new variable is freed?
Hence, this project aims to implement a linearly-typed imperative language with support for stackful coroutines, and to evaluate the performance of the resulting language. To further expand this project, support for multithreading could be introduced, followed by implementations of concurrent algorithms and data structures.

## APPROACH

This language will be implemented in Rust. The language will be a simple imperative language, with basic language features such as variables, functions, and control flow. This language can be represented intermediately in continuation-passing style. In continuation passing style, all functions take an additional argument, which is a continuation to call when the function returns. This allows the language to represent functions as closures, which can be passed around as values.
The language will then introduce stackless coroutines, which can be intermediately represented as a closure, holding a reference to the continuation to resume execution from. This is a similar approach to the one taken by Rust, which desugars coroutines into a state-machine.
The language will then introduce linear types, and the implementation of coroutines will be modified to support these. To implement linear types, the language will use a borrow checker, similar to the one used by Rust. This will allow the language to track the usage of variables, and to determine when a variable is no longer needed.
Next, coroutines can be extended to have their own stack, providing support for recursive coroutines and for coroutines that are called from multiple places. This will be implemented by having the coroutine hold a reference to its own stack, and to the continuation to resume execution from.

Finally, we can relax the linear type system to allow copying of "simple" values, such as integers and booleans. Additional features, such as a structures and methods could also be added to the language, followed by support for multithreading.

## RELATED WORK

Phillip Wadler [3] introduced the idea of linear types for functional languages, with the motivation of creating values that cannot be duplicated or discarded. This would allow functional languages to treat resources (like File Input/Output) as linear values, while allowing the rest of the values to be non-linear. The non-linear values would be safely garbage collected, while the linear values would be deallocated when they are no longer needed.
As discussed in the paper, pure linearity, where there is strictly one reference to a value, is a stronger constraint than is required. While reading, multiple references to a value would still be safe, as long as their is only a single reference to the linear value the value while writing. This is the approach taken by Rust, which allows multiple references to a value while reading, but only a single reference when a write is performed.
Aleksandar Prokopec and Fengyun Liu [2],[1] also discuss their implementation first-class, type-safe, stackful coroutines using metaprogramming in Scala, rather than through call-stack manipulation or program transformation. However, as discussed in the paper, the approach taken would also be applicable to implementing it inside a compiler.
To support stackful coroutines, coroutines are initialized by *starting* an instance. Each call to the coroutine instance is then referred to as a *resume*. The coroutine instance has its own stack available to it, which is allocated when the coroutine is created.
Coroutines can also be duplicated, using a *snapshot* operation that creates a new instance of the coroutine, with its own stack, and hence both instances can be resumed independently. The paper also does discuss some of the formal semantics to type this extension to Lambda Calculus.

## REFERENCES

[1] Prokopec, Aleksandar, and Fengyun Liu. 2018. "Theory and Soundness of Coroutines with Snapshots". DOI:10.4230/LIPIcs.ECOOP.2018.3

[2] Prokopec, Aleksandar, and Fengyun Liu. 2018. "On The Soundness Of Coroutines With Snapshots". https://arxiv.org/abs/1806.01405.

[3] Wadler, Philip. 1990. "Linear types can change the world!". https://cs.ioc.ee/ewscs/2010/mycroft/linear-2up.pdf