Formal Semantics of Linearly Typed Stackful Coroutines: Final Proposal

Aryan Wadhwani
Purdue University
wadhwani@purdue.edu

1. Introduction

Performance is crucial for low-level systems languages, and supporting various concurrency constructs in these languages is in turn necessary to provide programmers with better abstractions while implementing concurrent algorithms. One such abstraction is coroutines, which allows for cooperative task management on a single thread. In other words, functions are able to coordinate with each other to voluntary yield control to allow other functions to execute, before being resumed at the point at which it was halted.

Asynchronous functions, or functions that do not return a value immediately, are a language construct that can be built on top of coroutines. For example, a function that reads from a file would not return a value until the file has been read. This is a problem because the function would block the rest of the program from executing. Coroutines allow for the function to be suspended and resumed at a later time, allowing the rest of the program to continue executing.

Additionally, we can consider finite state machines, which are programs that hold a finite number of *states*, each being able to *transition* to a certain set of other states, based on the result of computations on their current state. Although these can also be implemented using *goto* statements, or mutually recursive functions, These are most naturally represented by symmetric coroutines, where each state is represented by a coroutine, and the transitions are represented by yielding control to another coroutine.

Languages like C and C++, with very few memory safety features built-in, expect programmers to explicitly handle memory allocation and deallocation, which if done incorrectly can lead to memory leaks, use-after-free and several other memory errors.

Hence, in modern languages, there has also been a push towards memory safety. Most languages, like Java, Go, and Javascript have opted to use Garbage Collection to handle memory management and provide some improvements in memory safety of their programs.

However, this requires the program to be periodically paused to perform a garbage collection pass, which can be an expensive operation, and unusable for some workloads. Additionally, garbage collection may still lead to memory leaks, since the garbage collector may not be able to identify all the memory that is no longer being used by a program. These languages still also fail to provide solutions to various other related memory errors, such as null pointer dereferences, race conditions, and resource leaks.

A new approach, taken by languages like Rust is to instead use affine/linear types to statically check and guarantee memory safety in programs. By introducing the notion of *ownership*, there is strictly a single variable that is bound to a value, and the value can be destroyed/unallocated when the variable goes out of scope. The ownership of a value can be transfered, but the previous owner can no longer access the value.

To have pointers to values, there is also a notion of *references*, where a value can either have multiple read-only references, or a single mutable reference. Additionally, lifetimes of the references are statically checked to ensure that they do not *outlive* the value behind the reference. These rules provide Rust with much stronger memory safety, as well as thread safety, guarantees, without having to rely on garbage collection or runtime checks.

By introducing affine type semantics to coroutines, we can provide a more robust and safe way to use coroutines in a memory-safe manner. We can provide guarantees about the lifetimes of the values that are being passed between coroutines, and ensure that the values are not used after they have been destroyed.

Hence, to explore this further, this paper proposes defining the semantics for a subset of the Rust language, particularly covering the concepts of ownership, mutable and immutable references and lifetimes. Then, this paper defines the semantics for stackful coroutines in this language.

Additionally, by representing these definitions using the $\mathbb{K}$ Framework, we are able to also provide an executable semantics for our language.

2. Background

2.1 Essence of Rust

The foundations of linear types, used by Rust, began with the work of Wadler (1990), which 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.

The paper also discussed a relaxed constraint from pure linearity, where there is strictly a single reference to a value, to allowing multiple references to a value while reading, as long as there is only a single reference to the linear value while writing. The latter constraint is similar to the borrowing semantics introduced by Rust.

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.

There have been several previous papers that provide semantics for portions of Rust's language. One such paper, Reed (2015), particularly focuses on creating formal semantics for the unique pointers and borrowed references, by creating a model of the Rust language, Patina.

In Patina however, the program requires explicit frees to be added, as the frees define the end of a variable's lifetime, as scopes are not present in this language, and to also avoid nested deallocations. This is to say, the Rust program:

```
{
  let mut x: i32 = 5;
  {
    let y: &i32 = &10;
  }
  x = 10;
}
```

can be represented in a Patina program as:

```
  let x: mut ~int = ~5;
  let y: ~~int = ~~10;
  free(*y);
  free(y);
  x = 10;
  free(x);
```

The core idea behind proving memory safety, is the idea of a **shadow heap**, which represents what each variable is bound to in memory, which may include partially deallocated values. For example, if we had $x = [1, 'a']$, initially the shadow heap for x would hold $[\ int,\ char]$. If we later freed $x.1$, we would hold $[\ int, uninit]$. We can use this to assert later that $x.1$ cannot be dereferenced, unless a new value is assigned to it.

Using this idea, this paper can statically identify null pointer dereferences, use-after-free errors, and missing frees. The paper goes on to show that any well-typed program with an empty shadow heap will be memory safe.

Another paper, Pearce (2021), goes further in also providing formal semantics for deciding between copy- and move-operations, lifetimes, and drops for simple store and load operations on variables, and then further demonstrates how this simple model can be extended to support conditional statements and tuples.

2.2 Coroutines

Coroutines, as a language construct, has had different definitions and expressive power in different languages. Moura and Ierusalimschy (2009) provides a clear classification of coroutines, based on three main properties.

First, coroutines can be **symmetric** or **asymmetric**. Symmetric coroutines provide a single operation to yield control to another coroutine, and can be resumed by any other coroutine. On the other hand, asymmetric coroutines provide two operations, one to invoke a coroutine, and another to yield control to the caller of this coroutine. Hence, there is a clear hierachy of control, where the caller of the coroutine can only resume the coroutine, and the coroutine can only yield control to the caller.

Second, coroutines can be **first class** or **constrained**. First class coroutines can be passed as arguments to other functions, and can be returned from functions. On the other hand, constrained coroutines cannot be directly manipulated by the programmer. For example, the *iterator* construct in many modern languages can be considered a constrained coroutine, as the programmer cannot directly manipulate the state of the iterator.

Third, coroutines can be **stackful** or **stackless**. Stackful coroutines have access to their own stack for calling functions, and hence can suspend their execution from within nested functions. On the other hand, stackless coroutines do not have access to their own stack, and hence cannot suspend their execution from within nested functions.

The paper then argues in favor of first-class stackful asymmetric coroutines, also defined as **Full asymmetric coroutines** as they provide a simple interface for the programmer, while retaining the expressive power of its symmetric counterpart, as well as one-shot continuations.

Another paper, Anton and Thiemann (2011), introduces a static type system for first-class stackful coroutines, which can either be symmetric or asymmetric. The language from this paper distinctly separates the initialization of a coroutine from its invokation, and also separates the values that come from an invokation into *yielded* values and the final *returned* value.

Expressions in the language are associated with three types: the type it may *yield*, the type it expects when it resumes, and the type it may *return*. These types are necessary to ensure that coroutines are correctly type-checked through nested function or coroutine calls.

3. Motivating Example

In this example, we wish to implement a join operation on two sorted tables, where the join is performed on the *id* column. The tables are stored in two separate files, and we wish to read the tables in parallel, and yield the joined rows as they are read.

```
coro read_rows(s: &str)() -> Row {
  let mut file = open(s);
  while !eof(file) {
    let buf: [Row] = read_next(file, 100);
    for i in 0..100 {
      yield buf[i];
    }
  }
}

coro join(s1: str, s2: str)() -> Row {
  let coro1 = read_rows(&s1);
  let coro2 = read_rows(&s2);
  let mut r1 = coro1()?;
  let mut r2 = coro2()?;
  loop {
    if row1.id == row2.id {
      yield &row1;
      r1 = coro1()?;
      r2 = coro2()?;
    } else if row1.id < row2.id {
      r1 = coro1()?;
    } else {
      r2 = coro2()?;
    }
  }
}

fn search(s: str) {
  let coro = join("table1", "table2");
  for row in coro() {
    if row.id == &s {
      print(row);
    }
  }
}
```

By having coroutines to implement the `read_row` operation, we can easily have a buffer of rows in memory, and yield them as they are read. This allows us to read the tables in parallel, and yield the joined rows as they are read. Although this could still be implemented using structs and iterators, the coroutines provide a simple interface for the programmer.

The yielded row from the `read_rows` is performing a move operation, as the row is taken by **value**, and not reference. This move operation can be shown to be sound, as the row is not used after it is yielded. Moreover, attempting to yield rows by reference would not be allowed, as the row is not guaranteed to be valid after the coroutine yields control, since the buffer, in which it is held, may be overwritten by the next read operation.

On the other hand, the `join` coroutine yields rows by reference, as the rows are guaranteed to be valid after the coroutine yields control. This is because the rows are not overwritten by the next read operation, as the `coro` call is performed after `row` exits its scope.

Another feature from Rust's type system that can be brought in for coroutines is propagating returns, in a similar manner to how Errors and None values are propagated from a function. Applying the ? allows us to unwrap and either early return the returned value, or continue execution with the yield value.

Hence, in this example we're able to show why linear types work well with coroutines, as we can guarantee that values and references that are yielded from a callee are not used after they yield control. Without this constraint from linear types, callees would be able to mutate values that are yielded by them, which would in turn cause the caller to observe changed values as well. We also guarantee that if a reference is yielded from a callee, it is guaranteed to be valid after the callee yields control, which would prevent runtime errors such as use-after-free.

4. Approach

4.1 Methodology

Implementing coroutines can either be done in the language level, or as a library.There currently do exist libraries that support coroutines in Rust, such as `coroutine-rs` and `corosensei`. However, these libraries both have their own drawbacks.

`coroutine-rs` is a library that implements asymmetric first-class coroutines, and is implemented using the `setjmp` and `longjmp` functions. This library however, relies on unsafe code to copy over the stack of the caller to the callee, and hence is not compatible with the Rust borrow checker. Additionally, this library also does not support stackful coroutines, and hence cannot suspend their execution from within nested functions. Finally, the library only allows a primitive integer type to be used as the input to the coroutine.

Implementing coroutines in the language level would be more desirable, as we can type-check coroutines with more information available to us. We would also need to introduce additional type information to the language, such as the type of the yielded value, and the type of the returned value. This would allow us to type-check coroutines with nested function calls, and also allow us to propagate returns from coroutines.

Hence, we wish to define our own subset of the Rust language, and implement coroutines in this subset. The main features that need to be implemented are the ownership, moving, borrowing, and lifetime semantics of the language. We would also want to implement a few data types like integers, strings, and arrays, as well as a few control flow constructs like loops and conditionals. Finally, we would also want to implement a few functions that would allow us to test our coroutines.

The $\mathbb{K}$ framework allows us to define a language through its syntax and operational semantics, and accordingly implements the parsing, compilation, and test-case generation. This would allow us to easily define our language, and test our implementation against programs written in our language. Wang et al. (2018) has implemented a subset of Rust in $\mathbb{K}$, for an earlier version of the framework. Many of the semantic rules can be derived from this implementation.

4.2 Approach

When a coroutine is instantiated, it creates a new stack frame, and copies over the initial value to the coroutine. The coroutine can then be called with an input value, and the coroutine will execute until it yields a value, or returns a value. The coroutine can be called again with a new input value, and will continue execution from where it left off. The coroutine can be called multiple times, and will continue execution from where it left off.

We define coroutines to have a type $Coro(\tau_1\tau_2\tau_3\tau_4)$, where τ_1 is the type for the initial value for to the coroutine, τ_2 is the type of the input to the coroutine, τ_3 is the type of the yielded value, and τ_4 is the type of the returned value.

The $Coro$ type is only permitted to have a single operation defined on it, which is the `co_init` call, which is used to instantiate the coroutine. When the coroutine is first instantiated with τ_1, it then creates a $Frame(\tau_2\tau_3\tau_4)$ object. We can then `call` the coroutine, i.e, a $Frame$ type, with an input value of type τ_2, and the coroutine will copy over the input value to the $Frame$'s stack frame, and then resume execution from where it left off. The coroutine will then either `yield` a value of type τ_3, or `return` a value of type τ_4, and the coroutine will then return this value to the caller for them to unwrap.

The $Frame$ type is allowed to have two asymmetric operations defined on it, which are the `co_yield` and `co_return` operations. `co_yield` needs to have a value of type τ_3 as its input, and `co_return` needs to have a value of type τ_4 as its input. `co_yield` also will return a value of type τ_2 after it resumes execution from where it left off.

The $Frame$ type also has a single symmetric operation defined on it, which is the `co_resume` operation, which takes a $Frame(\zeta_2, \zeta_3, \zeta_4)$ as its input, as well as an argument of type ζ_2 that will be applied to this frame. After the current frame resumes execution, it will return a value of type τ_3.

4.2.1 Stackful Coroutines

To support stackful coroutines, we need to embed more information in our functions, for them to be able to suspend their execution.

Similar to the approach taken by (Anton and Thiemann 2011), we would need to introduce new information to the types held by functions, namely the type of the yielded value η_1, along with it's return type η_2. If a function with type $f(\eta_1, \eta_2)$ is executing inside a coroutine of type $Coro(\tau_1\tau_2\tau_3\tau_4)$, η_1 needs to be a subtype of τ_3, as the function needs to be able to yield a value of type τ_3 to the coroutine. However, η_2 does not need to be a subtype of τ_4.

This would be implemented as syntax within the defined function, like: `fn foo() -> i32 yield char { ... }`, where the `yield` can be removed if the function does not yield a value.

Considering the implementation of these coroutines, we first have a function-stack, which holds the current function being executed, along with the environment and remaining computations. This function may have several other functions behind it, which are the functions that led to the current function being executed.

Now, the idea behind coroutines, is that we can traverse across these different function-stacks, and resume execution from where we left off in that stack. Hence, each coroutine that is being executed has its own function-stack.

4.2.2 $\mathbb{K}$ Implementation

The approach taken by Wang et al. (2018) is to define a configuration for the various states of Rust programs, or cells. For example, there is a Map from memory locations in the heap to their corresponding types, as well as a map from locations to the number of references held to them. The cells of the configuration are then used to define the operational semantics of the language.

Of particular interest, in first implementing the functionality of coroutines in this language, are the following cells:

$$\left\langle \langle K\rangle_k \langle Map\rangle_{env} \langle Map\rangle_{store} \langle\langle List\rangle_{fstack}\rangle_{control} \right\rangle$$

The k cell holds all the pending computations of a program, which is initially set to the computations present inside the `main` function. The env cell holds a mapping from variables V to a location L, which is used to lookup the type, mutability, references held and move semantics of the value at that location. The $store$ cell holds a mapping from locations L to a value V, which is used to lookup the value at that location.

The $fstack$ cell holds a list of functions that are currently being executed, which is used to implement the call stack. Each value in the list is a tuple of the env of that function, the k of that function (its remaining computations), and the return type for the function.

To introduce coroutines, the first idea is to add a new cell to the configuration, which will hold all the frames that have been created. Upon a coroutine being instantiated, a new frame will be created, and the initial value will be copied over to the new frame.

$$\left\langle \langle\langle List\rangle_{cframe}\rangle_{control}\langle\langle List\rangle_{cstack}\rangle_{control} \right\rangle$$

The $cframe$ cell holds a list of frames that have been created, while $cstack$ holds the order in which coroutines were called, similar to a call stack. Each value in $cframe$ holds an $fstack$ value, and each value in $cstack$ is an index in the $cframe$.

When a coroutine is first initialized, a new $fstack$ is created, which is pushed onto the $cframe$.

When a coroutine is called, the current $fstack$ is copied into it's corresponding $cstack$ location. The $cframe$ index of that frame is pushed onto the $cstack$, and the $fstack$ is copied over, and in turn the env and k for the last item for that $fstack$ are copied over as well. We can continue execution then.

When a coroutine yields, the current $fstack$ is copied into it's corresponding $cstack$ location. The $cframe$ index of that frame is popped onto the $cstack$, and the popped frame's $fstack$ is copied over, and in turn the env and k for the last item for that $fstack$ are copied over as well. We can continue execution then.

When a coroutine returns, the $fstack$ is removed from the $cframe$, and the $cstack$ is popped. We take the previous $fstack$ from the $cframe$, and copy over the env and k for the last item for that $fstack$, and continue execution.

5. Milestones

There was first an attempt to implement the semantics of the language in Rust itself. Parsing of programs, using recursive descent, was implemented successfully. Additionally, a type checker was implemented, which was able to type check programs with complex types, such as nested tuples, and functions with multiple arguments, references, and mutable variables. There's also an interpreter, which is able to execute programs for this AST representation successfully.

However, the implementation of the semantics of the language is a bit of a daunting task, and it would be better to implement the semantics in a language that is more suited for the task, such as $\mathbb{K}$ framework. Despite some time being lost in the initial implementation of the language, switching to $\mathbb{K}$ will allow for a more efficient and straightforward implementation of the semantics.

There are still some questions to be answered and researched further, like with how to better express lifetimes for this language, particularly in the context of coroutines.

Another interesting topic, to further look into, is how to approach the two possible values that can come from calling a coroutine, which is either a yielded value or a return value. Anton and Thiemann (2011) opted to have different continuations to call if the coroutine yields or returns, but this paradigm is not present in Rust, and hence should be avoided. As discussed in the motivating examples, Rust's enumeration types can be used to represent this.

References

K. Anton and P. Thiemann. Typing coroutines. In R. Page, Z. Horváth, and V. Zsók, editors, *Trends in Functional Programming*, pages 16–30, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22941-1.

A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), feb 2009. ISSN 0164-0925. doi: 10.1145/1462166.1462167. URL `https://doi.org/10.1145/1462166.1462167`.

D. J. Pearce. A lightweight formalism for reference lifetimes and borrowing in rust. *ACM Trans. Program. Lang. Syst.*, 43(1), apr 2021. ISSN 0164-0925. doi: 10.1145/3443420. URL `https://doi.org/10.1145/3443420`.

E. C. Reed. Patina : A formalization of the rust programming language. 2015.

P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. Krust: A formal executable semantics of rust. *CoRR*, abs/1804.10806, 2018. URL `http://arxiv.org/abs/1804.10806`.