Formal Semantics of Linearly Typed Stackful Coroutines

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.

ISBN [to be supplied]. . . $15.00

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. Although this project is also implemeneted using K, the semantics used are different from the ones introduced in this paper.

4.2 Approach for Coroutines

First, we define what behavior we expect from our coroutines. A coroutine is expected to consistently yield values of the same type until they terminate. Additionally, the coroutine can take in some initial arguments of one type, and receive an argument of another type each time it is resumed.

When a coroutine is instantiated, it creates a new stack frame, and copies over the initial values to the coroutine. The coroutine can then be called with an input value, and the coroutine will execute until it yields a value. The coroutine can be called again with a new input value, and will always continue execution from where it left off. After the program ends, any calls to the coroutine will block.

We define coroutines to have a type $Coro(\tau_1\tau_2\tau_3)$, 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, and τ_3 is the type of the yielded value.

The $Coro$ type is only permitted to have a single operation defined on it, which is the `start` call, which is used to instantiate the coroutine. When the coroutine is first instantiated with τ_1, it then creates a $Frame(\tau_2\tau_3)$ 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 continue execution until it reaches a `yield` a value of type τ_3, where the calling frame can continue execution, with the value that was yielded now available.

Stackful Coroutines

To support stackful coroutines, we need to embed more information in our functions, for them to be able to suspend their execution. However, we also need to ensure, at every call to resume a coroutine's execution, we're able to provide the arguments from resume to the caller. To do this, there are two approaches.

4.2.1 Ensuring the argument names remain in scope

If a function is called from within a coroutine, it should be able to yield values too, as long as values are of the same type. Finally, since a coroutine could be resumed from within a nested function, we need to be able to resume execution from within a nested function. Hence, we expect the function to also have one of its arguments be the same type as the resume type for the coroutine.

In other terms, when we have a coroutine defined with type $Coro(\tau_1, \tau_2, \tau_3)$, and the coroutine at some point calls a function with type $\tau_4 \rightarrow \tau_5$, τ_4 must not only be the same type as τ_2, but must also have the same identifier.

This way, we always ensure that the resume arguments remain in scope, and in turn, at every resumption of the coroutine, we can provide the arguments from the caller to the coroutine.

However, this seems confusing to the developer, as they must account for each function called by a coroutine that also yields values to have a specific argument, always "fitting" the coroutine's resume type, which may change value after each yield call.

4.2.2 Passing the resume arguments to yield

In this method, we instead expect the yield command to return the value passed in as an argument, which ensures that regardless of the function we're in, we're able to resume execution from the caller.

In other terms, when we have a coroutine defined with type $Coro(\tau_1, \tau_2, \tau_3)$, when we call `yield` η_1, with η_1 being of type τ_3, we expect the yield expression to provide a value of type τ_2 upon resumption of the coroutine.

This seems more intuitive, as we can occasionally ignore the yielded value's types, and the times we expect it, we use our regular typing rules to just ensure that the type of the yield value is same as the type of the variable assigned to. For simplicity, we'll assume only one yield value is returned, but this can be easily extended to multiple yield values if we add tuples to our language.

4.3 Linear Typing

In our model, every value that's created has a memory location. If we had `let x: int = 5`, we would create a memory location that holds the value 5, and the state that the variable `x` currently holds that location.

To define executable semantics for linear types, we wish to attach specific information to each memory location. By attaching the linear type information to memory locations, over the variables that reference them directly, we're able to better understand different issues, like aliasing, when memory's moved, and when memory's dropped. With this in

mind, we define the following metadata for each memory location.

4.3.1 Liveness

When a location is created, it is added to the set of **alive** variables. As long as a location's alive, it is safe to read the value at that location.

However, when a location exits the scope it was created in, it is said to no longer be alive. The set of alive locations after exiting a scope is restored back to the set of alive locations before entering that scope.

```
let x: int = 10;
let q: &int;
if x < 20 {
  let y = 10;
  q = &y;
}
x + *q;
```

Here, after the `if` block ends, the memory location declared inside that `if` is no longer alive. Hence, when we attempt to do `x + *q`, we can see that we're no longer allowed to read that memory location, which is where we'll halt execution.

Additionally, when a value is moved, the old location is no longer alive, as "ownership" of the value is now with the new location.

```
{
  let x: int = 10;
  let y: int = x;
  x + y;
}
```

After the operation `y = x`, the memory location `x` references is removed from the **alive** set. Hence, when we attempt to do `x+y`, we can see that we're no longer allowed to read `x`, which is where we'll halt execution.

4.3.2 Mutability

This defines the mutability of the variable associated to a location. If a location is marked as immutable, this implies that the variable associated to that location cannot be reassigned to a new value, and the inverse is true for mutable locations.

4.3.3 Borrowing

This indicates if a specific location is borrowed, and if so, whether it is borrowed mutably or immutably. If a location is borrowed mutably, it cannot be borrowed again. However, if it is is borrowed immutably, it can be borrowed by other immutably as well.

Like liveness, the borrowing information is also restored when exiting a scope to its values before entering the same scope.

```
let mut x: int = 10;
{
  let z: &mut int = &mut x;
  *z = 15;
}
let q: &int = &x;
*q + x;
```

Also, to remain consistent with a single mutable refernce, observe that while there is a mutable reference to a location, we cannot use `x` to modify the value.

Here, we see that the location `x` is borrowed mutably by `z`, during which time, we cannot borrow `x` again. However, when we exit the scope under which `z` was declared, we can see that `x` is no longer borrowed, and hence, we can borrow it immutably by `q`.

4.3.4 Limitations

The borrow-checker, used by Rust to guarantee linear typing, is a much more complex algorithm, and can hence accept several cases which this approach would fail to proceed with. Cases where multiple mutable references under the same scope to the same variable, where it's clear that one mutable reference's usage ends before another's begins.

Additionally, our approach requires us to always have an intermediate variable that's declared before we use the memory location. That is to say, we're unable to do `func(&10)`, and would instead need to do: `let x: int = 10; func(&x)`.

However, this does give us a good baseline, to cover most of the simple examples of linear typing.

4.4 $\mathbb{K}$ 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. We keep a similar approach in our implementation, while changing the semantic rules that define the language.

We additionally strip down the syntax, so as to focus on the core features of the language, and to make the implementation simpler. Particularly, we discard arrays and structs, and we also simplify the types available to just integers, booleans, strings, references, functions and coroutines. We additionally simplify all the various types of integers, which are either unsigned or signed with different byte sizes to just a single `int`.

In $\mathbb{K}$, we first need to define a syntax for our language, and then create rewriting rules, which define how we can "rewrite" the program. A rewrite rule is fairly analagous to operational small-step semantics. And rewriting is analogous to evaluating the program; it defines under what conditions we can evaluate a certain expression, and what the result of that evaluation is.

4.4.1 Syntax

We'll begin by defining the syntax this language uses. Our first grammar relates to the types that can be defined in this language.

⟨*type*⟩ ::= `int`
| `bool`
| `string`
| `&` ⟨*type*⟩
| `&mut` ⟨*type*⟩
| `fn (` ⟨*types*⟩ `) ->` ⟨*type*⟩
| `coro (` ⟨*types*⟩ `)(` ⟨*types*⟩ `) ->` ⟨*type*⟩
| `corun (` ⟨*types*⟩ `) ->` ⟨*type*⟩

⟨*types*⟩ ::= ⟨*type*⟩ | ⟨*type*⟩ ',' ⟨*types*⟩

Now, we can define the declarations that can be made in this language.

⟨*stmt*⟩ ::= `let` ⟨*id*⟩ `:` ⟨*type*⟩ `=` ⟨*expr*⟩ `;`
| `let` ⟨*id*⟩ `:` ⟨*type*⟩ `;`
| `let mut` ⟨*id*⟩ `:` ⟨*type*⟩ `=` ⟨*expr*⟩ `;`
| `let mut` ⟨*id*⟩ `:` ⟨*type*⟩ `;`

Note that the declarations with a value are desugared into a declaration without a value, followed by an assignment.

Next, we have expressions.

⟨*expr*⟩ ::= `Int` | `Bool` | `String`
| `&` ⟨*expr*⟩
| `&mut` ⟨*expr*⟩
| `*` ⟨*expr*⟩
| `start` ⟨*expr*⟩ `(` ⟨*exprs*⟩ `)`
| `call` ⟨*expr*⟩ `(` ⟨*exprs*⟩ `)`
| `yield` ⟨*expr*⟩
| ⟨*expr*⟩ `=` ⟨*expr*⟩

⟨*exprs*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ',' ⟨*exprs*⟩

For brevity, we'll exclude the normal binary and unary operations that operate over integers, booleans and strings.

Finally, to bring this all together, we'll define some control flow statements, along with function and coroutine declarations.

⟨*stmt*⟩ ::= ⟨*expr*⟩ `;`
| `if` ⟨*expr*⟩ ⟨*stmt*⟩ `else` ⟨*stmt*⟩ `;`
| `while` ⟨*expr*⟩ ⟨*stmt*⟩ `;`
| `return` ⟨*expr*⟩ `;`
| `fn` ⟨*id*⟩ `(` ⟨*params*⟩ `) ->` ⟨*type*⟩ ⟨*block*⟩
| `cr` ⟨*id*⟩ `(` ⟨*params*⟩ `) (` ⟨*params*⟩ `) ->` ⟨*type*⟩ ⟨*block*⟩

⟨*block*⟩ ::= `{` ⟨*stmts*⟩ `}`

⟨*params*⟩ ::= ⟨*id*⟩ `:` ⟨*type*⟩ | ⟨*id*⟩ `:` ⟨*type*⟩ `,` ⟨*params*⟩

4.4.2 Configuration

In $\mathbb{K}$, we represent all the information that's needed to evaluate a program as a nested multiset of items, where each item could be a value, map, set or list, which we call the configuration of the program.

For our semantics, we'll define the following cells:

1. `<stacks>`: This is a map of all the function stacks in the program. Each item here represents a single function stack. We begin with `main()` running on its own stack. Whenever a coroutine is created, a new stack is created and added to this map. Inside a stack, we have the following information:
   (a) `<k>`: This holds the actual computations of a program that will be executed next, in the order they need to be processed. This initially begins with all the declarations, after which `main()` is called.
   (b) `<control>`: This holds the function stack, which is a list of `K` items. Everytime a function is called, the remaining computations in `<k>` and other information from the stack are pushed into this list. When a function returns, the top item of this list is popped, and the remaining computations are pushed back into `<k>`, with the return value of the function taking the place of the function call in `<k>`.
   (c) `<env`: This holds a mapping from variable names to memory locations in the `<store>`. This is used to lookup the memory location of a variable when it's used in an expression, and ensure that the variable we reference is in scope.
   (d) `<id>`: This is a unique identifier for this specific stack, which allows us to understand which coroutine is being referenced when we wish to call and resume it.
2. `<typeEnv>` This is a map from memory locations to their corresponding types. This is used to lookup the type of a variable when it's used in an expression, and ensure that the type of the variable we reference is correct.
3. `<genv>` This is a version of the `<env>` that is constructed before `main()` is called. As a result, it contains all the function and coroutines that are declared in a program.
4. `<store>` This holds the actual memory of the program. It is a map from memory locations to values. This is used to lookup and modify the value of a variable when its used in an expression.
5. `<nextLoc>` This is just a number, allows us to know which is the next free memory location we can allocate
6. `<input>` and `<output>`: These are just list of values, which act as abstractions for the input and output of a program. When a read is done, we pop the first item from `<input>`, and when a write is done, we push the value to `<output>`. The actual input and output operations are then done by the interpreter.

7. To ensure linear typing, we also introduced the concepts of liveness, mutability, and borrowing, which we can define as `<alive>`, `<mutable>` and `<borrow>`, all holding mappings from memory locations to their respective information.

Using this configuration, we're now able to define the rules that make up the semantics of this language.

4.4.3 Semantics

We'll now begin going over the semantics of this language. Some of the basic arithmetic rules will be omitted for brevity. Also, to make it simpler to understand, the heating/cooling rules, which help specify the order in which subexpressions of an expression are evaluated, are omitted.

Variable Declarations and Lookup

$$\left\langle \frac{\text{let X: T;}}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle X \Rightarrow L \rangle_{env} \langle L \Rightarrow \bot \rangle_{store} \langle \langle L \rangle \rangle_{alive} \\ \langle L \Rightarrow T \rangle_{typeEnv} \langle L \Rightarrow \text{false} \rangle_{mutable} \\ \langle L \Rightarrow \bot \rangle_{borrow} \langle L + 1 \rangle_{nextLoc} \end{array} \right\}$$

$$\left\langle \frac{\text{let mut X: T;}}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle X \Rightarrow L \rangle_{env} \langle L \Rightarrow \bot \rangle_{store} \langle \langle L \rangle \rangle_{alive} \\ \langle L \Rightarrow T \rangle_{typeEnv} \langle L \Rightarrow \text{true} \rangle_{mutable} \\ \langle L \Rightarrow \bot \rangle_{borrow} \langle L + 1 \rangle_{nextLoc} \end{array} \right\}$$

$$\left\langle \frac{\text{X}}{\text{V}} \right\rangle_k \left\{ \begin{array}{l} \langle X \rightarrow L \rangle_{env} \\ \langle L \rightarrow V \rangle_{store} \\ \langle L \rangle_{alive} \end{array} \right\}$$

Here, we have the two basic operations available to any variable, declaration and lookup. For declaration, we take the variable `X` and assign it a memory location `L` in the env, and accordingly intialize all the relevant cells. This includes adding the type of the variable, and whether the location is alive or out of scope. Hence, in the end, the declaration is rewritten with `.`, or nothing, as the result.

For lookup, we take the variable `X` and lookup its memory location `L` in the env, and then lookup the value `V` in the store. We also ensure that the location is alive, and that the type of the variable is correct. Hence, in the end, the lookup to variable `X` is rewritten with the value `V`.

Function and Coroutine definitions

$$\left\langle \frac{\text{fn F(Xs)} \rightarrow \text{T B}}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle F \Rightarrow L \rangle_{env} \langle \langle L \rangle \rangle_{alive} \\ \langle L \Rightarrow \text{func}(T, Xs, B) \rangle_{store} \\ \langle L + 1 \rangle_{nextLoc} \end{array} \right\}$$

$$\left\langle \frac{\text{cr C(Xs)(Ys)} \rightarrow \text{T B}}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle C \Rightarrow L \rangle_{env} \langle \langle L \rangle \rangle_{alive} \\ \langle L \Rightarrow \text{corodef}(T, Xs, Ys, B) \rangle_{store} \\ \langle L + 1 \rangle_{nextLoc} \end{array} \right\}$$

$$\left\langle \frac{\text{execute} \rightsquigarrow .}{\text{main()}} \right\rangle_k \left\{ \quad \right\}$$

These definitions are fairly straightforward. For a function *F* with parameters *Xs* and return type *T*, we consume the definition and store it in the Env. These are the steps that are done for a coroutine as well, and after all declarations are consumed, the only thing remaining is a reserved `execute` statement, which is rewritten to `main()`.

Function Application and Returns

$$\left\langle \begin{array}{c} \text{call func}(T, \text{Xs}, B)(\text{Vs}) \rightsquigarrow K \\ \hline \text{mkDecls(Xs, Vs)} \rightsquigarrow B \rightsquigarrow \text{return;} \end{array} \right\rangle_k \left\{ \begin{array}{l} \langle \langle \text{Env, Brw, Alv}, K, T \rangle \rangle_{fstack} \\ \langle Env \leftarrow GEnv \rangle_{env} \\ \langle Alv \rangle_{alive} \langle Borrow \rangle_{borrow} \end{array} \right\}$$

$$\left\langle \frac{\text{return } V; \rightsquigarrow -}{V \rightsquigarrow K} \right\rangle_k \left\{ \begin{array}{l} \langle \langle \text{Env, Brw, Alv}, K, T \rangle \rightarrow \ldots \rangle_{fstack} \\ \langle \_ \leftarrow \text{Env} \rangle_{env} \langle \_ \leftarrow \text{Brw} \rangle_{env} \\ \langle \_ \leftarrow Alv \rangle_{alive} \end{array} \right\}$$

Here, when we reduce a `call` expression to a function value, we first take the function parameters and the values passed in, and use a helper function `mkDecls` to create the declarations for each of the parameters with their values. After this, we insert `B`, which is the body of the function, into the `k` cell. We then push the current environment, borrow, alive, continuation, and return type onto the function stack, and reset the environment back to the global environment. Note that we do not reset the alive and borrow cells, as references passed into a function are still valid.

Next, when we're ready to return from a function, we pop the top of the function stack, and restore the environment, borrow, and alive cells. The value that's returned is then placed before the continuation, and execution resumes.

Coroutine Start, Call and Yields

Here, we now will need to work with multiple `k` cells, since each function frame has its own stack, environment and return type. First, we define how to start a coroutine.

$$\left\langle \frac{\text{start corodef}(T, \text{Xs}, \text{Ys}, B)(\text{Vs})}{\text{coro}(T, \text{Ys}, L + 1)} \right\rangle_{k_1}$$

$$\left\langle \frac{\cdot}{\text{mkDecls(Xs, Vs)} \rightsquigarrow \text{mkDecls(Ys, } \textbf{wait}\text{)} \rightsquigarrow B} \right\rangle_{k_{L+1}}$$

$$\left\{ \begin{array}{l} \langle L + 1 \rangle_{id_{L+1}} \\ \langle Env \leftarrow GEnv \rangle_{env_{L+1}} \\ \langle L \rangle_{loc} \end{array} \right\}$$

Here, we create a new stack, with a new environment. We rewrite the `start` operation with a `coro` object, which specifies the next arguments to execute with.

We then allocate a new stack, which is used by our new coroutine. We specify the ID of this coroutine object to be the next location, and we set the environment to be the global environment. We then have a specific **wait** operation, which is used to indicate that the coroutine is waiting for a value to be passed in. Hence, the `mkDecls` function blocks until the coroutine is resumed, and the value for wait is passed in. Finally, we have the body of the coroutine, which can only be executed after the coroutine is resumed, due to the **wait**.

Now, we can observe how a coroutine is called.

$$\left\langle \frac{\text{call coro}(T, \text{Ys}, C)(\text{Vs})}{\text{call coro}(T, \text{Ys}, C)(\text{Vs})} \right\rangle_{k_1} \left\langle \frac{wait}{\text{Vs}} \right\rangle_{k_C} \left\{ \quad \right\}$$

This, in comparison is much simpler. We simply take the values that a *running* coroutine is called with, and then

replace the **wait** with those values. We use the id **C** to recognize which stack we're interested in.

We do not make any rewrites in the caller, which implies that the caller is blocked until the coroutine yields. Additionally, observe this call would only occur if the next expression on the coroutine's stack is the **wait**.

Finally, we can look at how a coroutine yields.

$$\left\langle \frac{\text{call coro}(T, \text{Ys}, C)(\text{Vs})}{V} \right\rangle_{k_1} \left\langle \frac{\text{yield } V}{\text{wait}} \right\rangle_{k_C} \text{when type}(V) = T$$

This is also quite elegant. We know that k_1 cannot continue until the coroutine call is rewritten, and hence when we observe a yield, we simply replace the **call** with the value that's yielded. The callee is then just made to halt and wait for their next call, by replacing the `yield` expression with a **wait** again. Of course, we also need to ensure the return type of the coroutine is the same as the type of the value that's yielded.

Linear Typing

First, we'll define the rules under which we can take a mutable or immutable reference. Before this, we define a specific evaluation context while taking references.

So far, we always attempt to reduce an expression down to its value. However, for the expressions `&<Exp>` and `&mut<Exp>`, along with assignments as we'll observe later, we actually wish to reduce an expression down to a location. That is to say for `&x` we wish to find the variable `x`'s location, and not its value.

We describe this as: `&(HOLE => loc(HOLE))`, which states that the for the expression inside the reference operation, we wish to find its expression.

$$\left\langle \frac{\&L}{\text{ref}(T, L)} \right\rangle_k \left\{ \begin{array}{l} \langle L \to (Q \Rightarrow \text{immut}) \rangle_{borrow} \\ \langle L \to T \rangle_{typeEnv} \\ \langle L \rangle_{alive}, \text{when } Q \neq \text{mut} \end{array} \right\}$$

$$\left\langle \frac{\&\text{mut } L}{\text{mref}(T, L)} \right\rangle_k \left\{ \begin{array}{l} \langle L \to (Q \Rightarrow \text{mut}) \rangle_{borrow} \\ \langle L \to T \rangle_{typeEnv} \langle L \to \text{true} \rangle_{typeEnv} \\ \langle L \rangle_{alive}, \text{when } Q = \bot \end{array} \right\}$$

While borrowing a location, we first need to make sure that location is still alive. For immutable references, we wish to ensure it's not currently mutably borrowed, and for mutable references, it cannot be borrowed at all. If these cases hold, we then accordingly update it to its new value. In the end, we rewrite the expression to its respective values.

Now, we can look at the rules for dereferencing a location.

$$\left\langle \frac{*\text{ref}(T, L)}{V} \right\rangle_k \left\{ \begin{array}{l} \langle L \to V \rangle_{env} \\ \langle L \rangle_{alive} \\ \text{when } T = \text{type}(V) \end{array} \right\}$$

$$\left\langle \frac{*\text{mref}(T, L)}{V} \right\rangle_k \left\{ \begin{array}{l} \langle L \to V \rangle_{env} \\ \langle L \rangle_{alive} \\ \text{when } T = \text{type}(V) \end{array} \right\}$$

We just need to ensure that the location is alive, and then we can rewrite the expression to its value.

Assignment

We can finally look at the rules for assignment. What's important to note, is that similar to our rules for taking references, we wish to reduce the left hand side of the assignment to a location, and not its value. Hence, again, we have the same context: `(HOLE => lvalue(HOLE)) = _`.

First, we have a specific rule, when we're able to also reduce the right hand side of an assignment to a location. This is the case when we're moving ownership of a value from one location to another.

$$\left\langle \frac{L_1 = L_2}{L_1 = V} \right\rangle_k \left\{ \begin{array}{l} \langle L_2 \to V \rangle_{store} \\ \langle L_2 \to \bot \rangle_{alive} \end{array} \right\}$$

Here, we just need to ensure that the location we're moving from is still alive, and then we can move the value from one location to another. We then let the following rewrite rules actually verify that the typing and L_1's state is valid.

$$\left\langle \frac{L = V}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle L \to (\bot \Rightarrow V) \rangle_{store} \langle L \rangle_{alive} \\ \langle L \to T \rangle_{typeEnv} \langle L \to \bot \rangle_{borrow} \\ \text{when } T = \text{type}(V) \text{ and } Q \neq \text{immut} \end{array} \right\}$$

When a memory location has a $\bot$ value in the store (it has just been declared and not assigned a value), it is fine to assign a value to it, as long as it has not been immutably borrowed.

$$\left\langle \frac{L = V}{\cdot} \right\rangle_k \left\{ \begin{array}{l} \langle L \to (\_ \Rightarrow V) \rangle_{store} \langle L \rangle_{alive} \\ \langle L \to T \rangle_{typeEnv} \langle L \to Q \rangle_{borrow} \\ \langle L \to \text{true} \rangle_{mutable} \\ \text{when } T = \text{type}(V) \text{ and } Q \neq \text{immut} \end{array} \right\}$$

When a memory location is already assigned a value, we need to make sure that it is mutable, and that again it has not been immutably borrowed.

Miscellaneous Rules

We do have specific rules around *blocks*, as the scoping is important to ensure that the lifetimes of variables is correctly represented.

$$\left\langle \frac{\{S\}}{S \rightsquigarrow \text{reset}(\text{Env}, \text{Alive}, \text{Borrow})} \right\rangle_k \left\{ \begin{array}{l} \langle Env \rangle_{env} \\ \langle Alive \rangle_{alive} \\ \langle Borrow \rangle_{borrow} \end{array} \right\}$$

Here, S refers to the statements inside a scope. We ensure that any variables, that are declared inside the scope, are removed from the environment. Additionally, as lifetimes end, the we can discard the `alive` and `borrow` contexts from that scope.

This completes all the core rules for our language. We choose to skip the rules for control flow statements and arithmetic operations, as they're all straightforward: rewrite all

the expressions until they become values, and then perform the expected operation.

By implementing these rules in $\mathbb{K}$, we're able to generate an interpreter for our language. We can then use this interpreter to execute programs written in our language and see if they're correct.

5. Results

The rewriting rules defined here were written in $\mathbb{K}$, and example test cases were created with this language to evaluate the correctness of the rules. The written test cases were then executed with the interpreter generated by $\mathbb{K}$, and the results were compared to the expected results.

During the development of the semantics, running the test cases with the interpreter generated was helpful to identify issues with the semantics, and to ensure that the semantics were correct.

In the end, the defined semantics were able to correctly identify the errors in the test cases, and also correctly execute the programs that were written in the language. The defined language supported all the features that were defined in the language specification, above, as well as control-flow through `if` statements and `while` loops. The language also supported arithmetic operations.

However, while trying to extend the language to support tuples, arrays, and other structures, there were several roadblocks. Particularly, the current linear typing rules work well for single variables, but are unable to understand **partial references**.

6. Conclusion

Using K framework

We found that $\mathbb{K}$ alleviates a lot of the issues that arise while writing a language, and also allow us to form a strong set of semantics for the language. However, the learning curve for $\mathbb{K}$ itself is steep, given its weak toolchain compared to most modern languages. It also does have sometimes outdated documentation, which can make it difficult to find the correct way to implement certain features.

After overcoming that however, it certainly is a powerful tool, and reduces the work needed to write a full-fledged parser and interpreter from several thousand lines of code to a few hundred. Although the interpreter itself is slow, it is still a useful tool to have, as it lets programmers quickly test and modify their language, before writing an actual interpreter or compiler in a more performant language.

The executable semantics allow us to ensure that the observed path of execution at runtime is safe according to our linear typing rules, but it does not allow us to identify and ensure that a program as whole, regardless of its execution path, will be safe. To do this, we will need to implement static type-checking, which is much more difficult to accomplish with linear types by itself.

Coroutines and Linear Types

When coroutines are introduced to a linear type system, the problem even becomes more difficult. Currently, with functions, we know that any references that are passed into a function will be valid for the duration of the function's execution. For coroutines, it's difficult to statically show that, as after a yield, the coroutine's next call may be from a different scope, and the references may no longer be valid.

Additionally, if we wish for a function to return references to its caller, we just need to check whether the reference's lifetime is larger than the function's. That is to say, the value was created before the function was called, and hence will be valid for the duration of the function's execution and after it returns. For coroutines, this is no longer the case, as yielded references from local variables in a coroutine could be plausbibly be returned to the caller.

What all of this concludes, is that the interactions between linear types and coroutines is very difficult to statically understand. While it might be possible to have a static linear type system in a language with coroutines, the resulting type system might be too conservative, preventing most uses of references inside coroutines.

Additionally, while not explored in this project, it's unclear what the semantics for *symmetric* coroutines would look like in a linear type system. To construct three symmetric coroutines, each coroutine would need to have a reference to the other two. However, these references **should** be mutable, as upon every call to the coroutine, the coroutine would change in some manner, and this would hence be against the linear type system.

Coroutines vs other Concurrency Patterns

In general, it seems that Coroutines are a rarely adopted design pattern for concurrency in a language. Most languages that adopt coroutines, either go for asynchronous programming, or generators. Both of these are severely limited in comparison to coroutines, as they do not allow for the same level of control over the execution of the program. However, they are much simpler for programmers to understand, and can provide similar functionality in most cases.

Some of the ideas behind coroutines, like cooperative task scheduling, have been adopted by languages. For example in Go, there are Goroutines, which are lighterweight threads, and by using message passing as the main construct for cooperative concurrency, they can be used to write concurrent programs that are easier to understand than coroutines.

Future Work

In conclusion, we've succesfully defined executable formal semantics for a linearly-typed language with coroutines using $\mathbb{K}$. This allows us to identify errors in programs at runtime, and also to execute programs written in the language. The language includes basic types, control flow, functions, and coroutines, but lacks support for complex data types.

In the future, we would like to extend the language to support tuples, arrays, and other complex data types. We would also like to look into whether static type-checking for such a language would be possible, which would allow us to identify errors in programs before they are executed.

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`.