CS4215 Rust Project

Team members: Elbert Benedict (A0245772X), Mario Alvaro(A0255846R)

Repository URL: https://github.com/Elben85/CS4215-Rust-Project

This report can also be viewed online at: https://co-note.vercel.app/docs/-OOHYrq40NzCHt9QX5Rc

Table of Contents

Running / Building The Project

To run the project

1. Go to https://sourceacademy.org
2. Set the conductor to https://elben85.github.io/CS4215-Rust-Project/index.js
3. Run the program

If you would like to build the project on your own, you can clone / fork the repository and do the following steps:

1. Run `yarn install` to first download the dependencies
2. Run `yarn build` to build the `dist/index.js` file
3. Host the `index.js` file on some url
4. Go to https://sourceacademy.org
5. Set the conductor to be the url where you hosted the `index.js` file
6. Run the program

Project Objectives and Completion

We implemented a virtual machine for the following subset of the Rust programming language with the following objectives:

- Variable declaration and assignments
- Block Expression
- If expression
- Predicate loop expression (while loops) with break and continue
- Functions (as compile-time entities) and closures
- Common operator Expression (e.g. `+` , `-`)
- Ownership handling
- Borrow and dereference
- Borrow checker

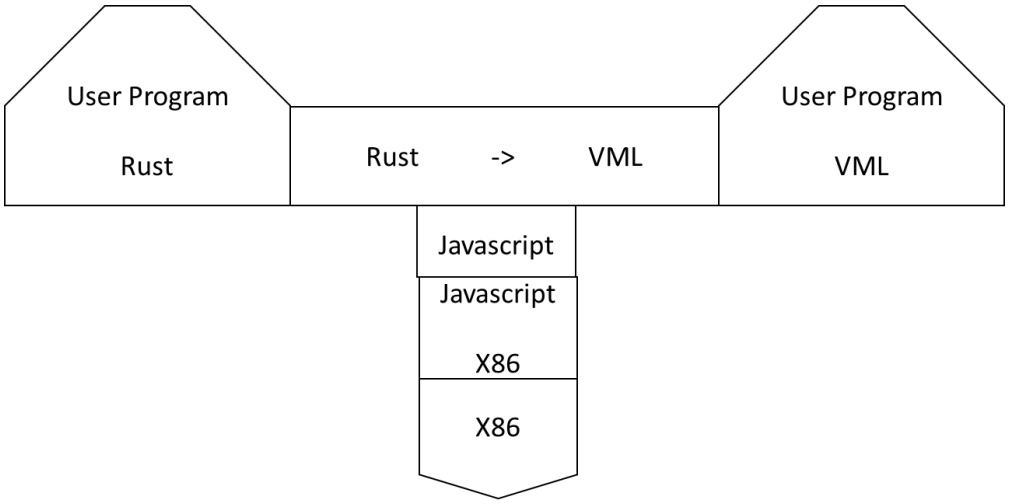
Moreover, we implemented a low-level memory management system using a buddy allocator, copy, move, and drop semantics.

In the final version of the program, all the following objectives are implemented with some restrictions:

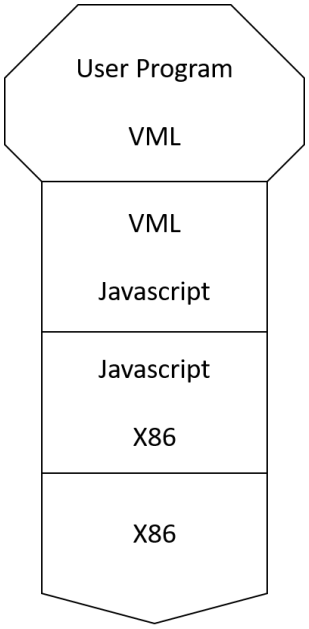
- No `move` closures, that is, the implemented closures won't take ownership of the captured variable from the enclosing environments.
- A partial borrow checker, which checks drop, ownership, and borrow checking, excluding functions, closures, and while loops. The ownership handling (move and copy semantics) of objects is still decided at compile time and executed deterministically by the instruction set.

Language Processing Steps

As a high level overview, the program is both a compiler than compiles the user program writtern in rust to the virtual machine language (VML) and also acts as an interpreter to run the User program written in VML using javascript.



During the compilation steps shown above, first the program will be parsed into the corresponding syntax tree using the parser generated by `antlr4g`. The compiler will then traverse the tree to do some type checking to ensure the program is type safe. The resulting types are cached for later use to determine whether a value should be copied or moved by the compiler. The compiler will then do some (partial) drop and borrow checking, making sure that there aren't issues like "user-after-free" and "accessing moved values". After doing some borrow checks, the compiler will then compile the program to VML.



After being compiled, the user program written in VML will be executed similarly using the concept in the 'More Realistic Virtual Machine' module.

Sequential Construct

Variables and Expressions

Data Type

Currently supported primitive data types are shown below:

Data Type	Syntax Example	Description
bool	let x: bool;	Boolean type (true or false) used for logical operations and control flow.
f64	let y: f64;	64-bit floating-point type used for decimal numbers. Corresponds to Rust's f64.

Note that primitive types are copied by default.

Non primitive data type includes:

- function type
- pointers
- strings.

Function type

function types are in the form of

```
fn(...argumentsType) -> returnType
```

For example, function `foo` :

```
fn foo(x: f64, b: bool) -> bool {
  return x > 0 && b;
}
```

would have to be type:

`fn(f64, bool) -> bool` function types are copied by default.

Pointer type

Currently, mutable and immutable references (borrows) are supported. Pointer types are in the form of `& (mut)? type`

For example consider the following variable `b`

```
let mut a: f64 = 1.0;
let mut b = &mut a;
```

Then variable `b` would have type `&mut f64`

Immutable references are copied by default while mutable references are moved.

String type

Currently the `String` type only supports the simple operator `+` and `==`. Strings types can be annotated using the `String` keyword. String types can't be copied and moved by default.

Variable Declaration and Assignment

Variables are declared using the `let` keyword, followed by the identifier and optionally by typing and assignment. The syntax follows such example:

```
let x = 10;
let s: f64;
```

Variables can be mutable or not, which are indicated by the keyword `mut` after the `let` keyword:

```
let mut y = 9;
```

Variables that are not tagged with the `mut` keyword are by default considered as immutable variables, which as a result will not be able for its value to be reassigned. The syntax for assignment is done using the `=` operator:

```
let mut z = 0;
z = 10;
```

If the variable type isn't specified, the type will be inferred the first time it is assigned. If the variable isn't assigned a value throughout the whole program an error will be thrown.

Operators

These operators are supported by the program:

Operator	Description	Example	Result	Associativity
`*`	Multiplication (f64)	`2 * 3`	`6`	Left-to-right
`/`	Division (f64)	`6 / 3`	`2`	Left-to-right
`+`	Addition (f64 / String)	`8 + 2`	`10`	Left-to-right
`-`	Subtraction (f64)	`92 - 52`	`40`	Left-to-right
`||`	Logical OR (bool)	`true || false`	`true`	Left-to-right
`&&`	Logical AND (bool)	`false && true`	`false`	Left-to-right
`>`	Greater than (f64)	`8 > 2`	`true`	Left-to-right
`>=`	Greater than or equal (f64)	`8 >= 9`	`false`	Left-to-right

Operator	Description	Example	Result	Associativity
`<`	Less than (f64)	`8 < 2`	`false`	Left-to-right
`<=`	Less than or equal (f64)	`8 <= 9`	`true`	Left-to-right
`==`	Equality (f64 / bool / String)	`8 == 8`	`true`	Left-to-right
`!=`	Inequality (f64 / bool)	`8 != 8`	`false`	Left-to-right
`%`	Remainder (f64)	`7 % 3`	`1`	Left-to-right
`-` (unary)	Negation (f64)	`-2 + (-3)`	`-5`	Right-to-left
`!`	Logical NOT (bool)	`!(1 == 7)`	`true`	Right-to-left

Dereference and Borrows

Currently, restricted borrowing and dereferencing are implemented using the following grammar:

```
dereferenceExpression: '*' (dereferenceExpression | variableName)
borrowExpression: '&' ('mut')? variableName
```

Note that borrows do not own the value they are referencing. Some checks are done to prevent invalid borrows. The lifetime of shared and mutable borrows is from the time it is first defined until the time it is last used. Hence, the following is a completely valid program.

```
let mut i = 1;
let b = &mut i;
let c = &mut i;
```

Block Expression

Block expressions are in the form of

```
{
  statements* expressionWithoutBlock
}
```

Block expressions will evaluate to the value of `expressionWithoutBlock` if provided and will evaluate to `()` otherwise. Note that the result of the block expression value will be copied/moved depending on the type.

Control Flows

If-Else

Conditional branching using `if`, `else if`, and `else`.

```
let x = 10;
if x > 5 {
    // code
} else if x == 5 {
    // code
} else {
    // code
}
```

Each block expression on all the branches must evaluate to the same type.

While Loop

Repeats a block of code while a condition holds true.

```
let mut i = 0;
while i < 5 {
    // code
    i = i + 1;
}
```

In the case of while loop, the block expression must evaluate to the unit type `()`

Break and Continue are also implemented in the program with the keyword `break` and `continue`

```
let mut i = 0;
while i < 5 {
    // code
    i = i + 1;
    if i%2 == 0 {
        continue;
    } else {
        break;
    }
}
```

Functions

Function Declaration

Declare functions using the fn keyword. Functions can take parameters and return values.

```
fn add(a: f64, b: f64) -> f64 {
    return a + b;
}

let result = add(2, 3); // result = 5
```

Each parameters must be specified with a type. Moreover, For a value returning function, a returnType must be specified with the function as well. Unspecified returnType is only allowed on non returning function, for example:

```
fn do(a: f64, b: f64) {
  a + b;

  // more code...
}
```

Which in this case, the returnType will be the Unit type `()`.

Functions are compile time entities. That is, it will not be assigned to any variable during runtime. All occurence of the function will directly be replaced by the corresponding `LDF` instruction. As a side effect of being compile time entities, function will not be able to capture dynamic environments and will always extend from the global environment.

Closure Statement

Closures in Rust are anonymous functions that can capture variables from their surrounding environment. They are often used when you need a short function-like construct, especially for callbacks or functional programming patterns.

Closures can be written in *expression-based or block form.

These are concise closures that consist of a single expression. The return value is implicitly the result of the expression and the return type is inferred in this case

```
let add = |a: i32, b: i32| a + b;
let result = add(2, 3); // result = 5


let result2 = (|a: i32, b: i32| a * b)(2, 3); // result2 = 6
```

Closures can also use braces and include multiple statements. You must explicitly use the `return` keyword if you're returning a value early.

```
let complex = |x: f64| -> f64 {
    if x > 10 {
```

```
        return x * 2;
    }
    x + 5
};

let result = complex(8); // result = 13
```

Note that since we didn't implement `move` closures, they will not capture ownership of the surrounding environment. Hence, the following program is not a valid program since `x` does not live long enough.

```
let foo;
{
    let x = 1;
    foo = | | x;
};
foo();
```

Copy, Move, and Drop semantics

Currently, these are the following cases where a value will be dropped

- when the value is popped out of the stack and it has no owner
- when the variable it is assigned to goes out of scope
- when the variable it is assigned to gets reassigned

Whether a value should be copied or moved is determined at compile time using the result of the type checker. Copy and move semantics are used for the following values:

- values to be assigned to a variable
- function and closures return values
- the expression result of a block expression

Instruction Set

Instruction	Description	Parameters	Operating Stack
`LDC`	Allocate literal on the heap and push the allocation address into stack	`value`	None
`BINOP`	Apply `operator` to `arg1` and `arg2`. Both args addresses are expected to be on the stack	`operator`	`arg1` and `arg2`

Instruction	Description	Parameters	Operating Stack
POP	Pop the topmost value on the stack and drop it if there is no owner.	None	value
ASSIGN	Assign topmost stack value to specified address (which should be a pointer). The instructions expects both the pointer address and the value on the stack.Note that the value assigned previously will be dropped and value will be moved to the specified address.	None	assignee adress and value
LDA	Load the address of the pointer at frameIndex and valueIndex to the stack	[frameIndex, valueIndex]	None
LD	Load the value at frameIndex and valueIndex to the stack	[frameIndex, valueIndex]	None
ENTER_SCOPE	Extend the env with a frame of given length. Push the old env to the runtime stack	frameSize	None
EXIT_SCOPE	Restore the address of the env to one on top of the runtime stack. Move the value on top of the stack such that it has no owner. The old environment and the last frame are dropped.	None	value
UNOP	Execute unary operator. Gets argument from stack	operator	arg
JOF	jump to address if the top of the stack evaluates to false	PC address	predicate
GOTO	jump to address	PC address	None
DEREF	dereference a pointer to the location it points to. expects a pointer on top of the stack	None	pointer
BORROW	Produce a pointer referencing to a variable. Expects the address of the borrowed variable on top of the stack	None	address
LDF	Creates a closure on the heap and pushes its address onto the operand stack. Takes the function's arity and address, and captures the current environment. If the useGlobal is set to true (e.g. for functions), it will capture the global environment instead.	arity , PC address , useGlobal	None

Instruction	Description	Parameters	Operating Stack
`RESET`	Restores the previous execution context by popping the top frame from the return stack. If the frame is a call frame, it restores the program counter and environment from it. Drops the top frame in the process of each restoration. Move the value on top of the stack such that it has no owner.	None	`value`
`CALL`	Executes a function call. Takes the arity of the function. Expects the function address and arguments on the operand stack. Creates a call frame to save the current execution context, extends the environment with arguments, and jumps to the function's address.	`arity`	`arguments` and `function`
`COPY`	Pop the topmost value from the heap and copy the value. Push the copied value onto the heap.	None	`value`
`DONE`	The last instruction indicating execution has finished.	None	None

Testing

Unit tests are located on the `src/tests` directory, with each unit test file testing a specific language construct based on the file name. For example, `BorrowChecker.test.ts` will mainly contain test cases regarding the borrow checker.

On all our test cases that run the program, we additionally assert that at the end of evaluation:

- The length of the Operating Stack is 0
- All dynamically allocated memory in the heap has been deallocated

To run the test scripts you can simply run the following command

```
yarn test
```

or optionally, if you only want to run a specific test file you can run

```
yarn test [PATH_TO_FILE]
```