Nust: A Rust-Like Language for Stack-based VMs

David Tran (e1506839)

April 2025

Nust is a programming language based on a subset of the Rust language (with some minor modifications). The following report details the process of implementing an implementation of this language, specifically as a compiled language for a stack-based VM, the Nust Virtual Machine (NVM).

Scope

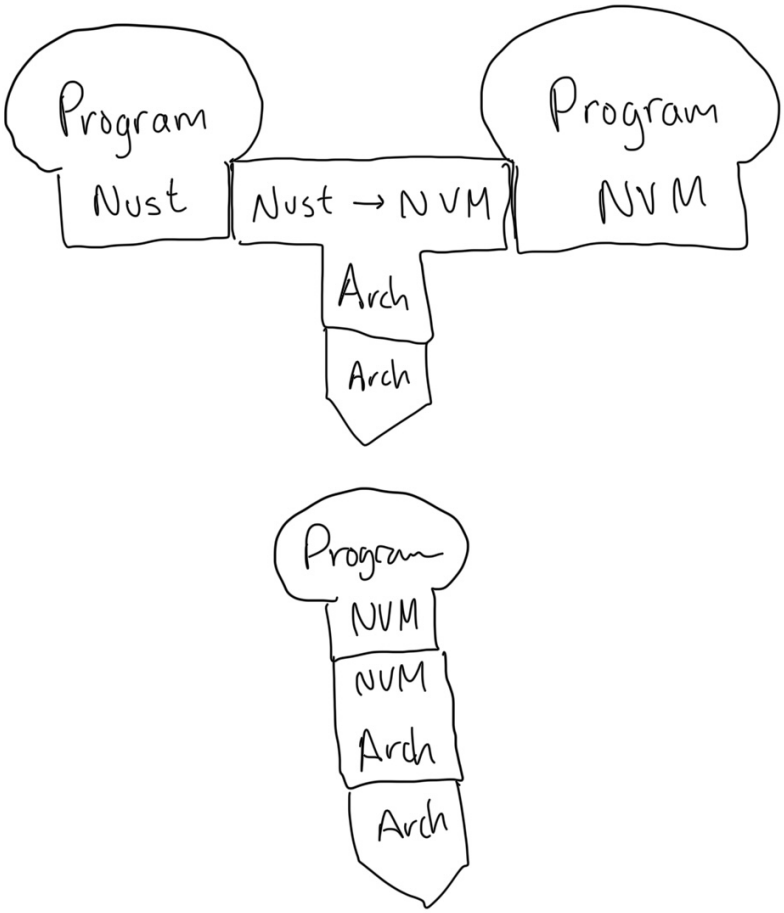
The scope and objectives of this project are the following:

1. Implement a recursive descent parser for the language, transforming the source code into an abstract syntax tree (AST)
2. Implement a type-checker for the language, operating on the AST and augmenting each node with its associated type, throwing errors otherwise.
3. Design a stack-based VM, the Nust Virtual Machine (NVM), and implement a compiler to compile the AST with its type information into machine code for the NVM.
4. Realize a concrete implementation of the NVM.

The formal specification of the grammar of Nust in Backaus-Naur form is shown below. Note the modification from Rust that we don't require the de-reference operator '∗' when de-referencing borrowed references. We shall see when we discuss the semantics of this language that we opt to auto-deref by default, as Rust does in some contexts (for example, calling a method on a reference).

```
<program> ::= <item_list>
<item_list> ::= <item> <item_list>
              | ε
<item> ::= <function_decl>
         | <stmt>
<function_decl> ::= "fn" <ident> "(" <param_list_opt> ")" "{ <stmt_list> "}"
<param_list_opt> ::= <param_list>
                   | ε
<param_list> ::= <param> "," <param_list>
               | <param>
<param> ::= <mut_opt> <ident> ":" <type>
<stmt_list> ::= <stmt> <stmt_list>
              | ε
<stmt> ::= <let_stmt> ";"
         | <expr_stmt> ";"
         | <if_stmt>
         | <while_stmt>
         | <block>
<block> ::= "{" <stmt_list> "}"
<let_stmt> ::= "let" <mut_opt> <ident> ":" <type> "=" <expr>
<expr_stmt> ::= <expr>
<if_stmt> ::= "if" "(" <expr> ")" <block> <else_clause_opt>
<else_clause_opt> ::= "else" <block>
                    | ε
<while_stmt> ::= "while" "(" <expr> ")" <block>
<expr> ::= <literal>
         | <ident>
         | <expr> <binop> <expr>
         | <unop> <expr>
         | <expr> "(" <arg_list_opt> ")"
         | "&" <expr>
         | "&mut" <expr>
         | "(" <expr> ")"
         | <ident> "=" <expr>
<arg_list_opt> ::= <arg_list>
                 | ε
<arg_list> ::= <expr> "," <arg_list>
             | <expr>
<binop> ::= "+" | "-" | "*" | "/" | "==" | "!="
          | "<" | ">" | "<=" | ">=" | "&&" | "||"
<unop> ::= "-" | "!"
<mut_opt> ::= "mut" | ε
<literal> ::= <int_lit> | <bool_lit> | <string_lit>
<int_lit> ::= [0-9]+
<bool_lit> ::= "true" | "false"
<string_lit> ::= "\"" <string_char>* "\""
<string_char> ::= any character except " or \ | \ <escaped_char>
<escaped_char> ::= "\"" | \ | "n" | "t" | "r"
<type> ::= "i32" | "bool" | "str" | "&" <type> | "&mut" <type>
<ident> ::= [a-zA-Z_][a-zA-Z0-9_]*
```

On completion, source code in the Nust language will be able to be compiled into byte-code for the NVM, which itself may be executed on a host machine. This process can be summarized with a tombstone diagram:



Specifications

In this section we describe in detail the type semantics of the language, the design of the parser, the type-checker, the design of the NVM, and the NVM machine code language.

Type Semantics

The language is statically typed with a focus on memory safety through a reference system. It supports three primitive types: 32-bit integers (i32), booleans (bool), and strings (str). The type system includes both value types and refer-

ence types, with references being either immutable (`&T`) or mutable (`&mut T`). Variables can be declared as either mutable or immutable using a let binding, and the type checker enforces mutability rules - immutable variables cannot be modified, and mutable references cannot be created from immutable variables.

The type system includes simple type inference for identifiers initialized with the `let` keyword while requiring explicit type annotations for function parameters and return types. Function calls are type-checked to ensure argument types match parameter types, and the return type of a function must match the type of its last expression.

The type checker also enforces reference safety rules, ensuring that references cannot outlive their referents and that mutable references have exclusive access. Type compatibility is checked for assignments and operations, with implicit conversions being limited to prevent unintended behavior. The type system is designed to catch type errors at compile time while providing clear error messages with source locations.

Parsing and Type-Checking

The parser is a hand-written LL(1) recursive-descent parser. Each node in the resultant AST contains all the information necessary to verify the type-correctness of the program from which the compiled bytecode may be generated. The type-checker performs a tree-walk of the AST to ensure that all expressions and statements conform to the language's static type rules. During this traversal, it annotates each node with inferred or validated types, reporting any mismatches or violations.

The Nust Virtual Machine

The Nust Virtual Machine (NVM) is a lightweight, stack-based virtual machine designed to execute bytecode generated from Nust programs. It supports a minimal set of value types and instructions sufficient to evaluate statically-typed code.

The NVM operates on a set of typed values including 32-bit signed integers (`i32`), booleans (`bool`), strings (`str`), references (`ref` and `mut_ref`), and function references (`fn`).

In addition to the byte-code, the compiler also outputs a *function table*: a static table mapping function identifiers to their location in the machine code. When a function is called, its location is found in this table. Each function call creates a new stack frame which holds the return address, frame pointer, local variables, and arguments. Arguments are passed in reverse order to ensure correct positioning in the callee's frame.

The VM uses a simple, linear sequence of bytecode instructions. Key categories include:

- *Stack Operations:* `PUSH_I32`, `PUSH_BOOL`, `PUSH_STR`, `POP`
- *Variable Operations:* `LOAD`, `STORE`, `LOAD_REF`, `STORE_REF`

- *Arithmetic:* `ADD_I32`, `SUB_I32`, `MUL_I32`, `DIV_I32`, `NEG_I32`
- *Comparison:* `EQ_I32`, `NE_I32`, `LT_I32`, `LE_I32`, etc.
- *Logic:* `AND`, `OR`, `NOT`
- *Control Flow:* `JMP`, `JMP_IF`, `CALL`, `RET`, `RET_VAL`
- *Reference Handling:* `BORROW`, `BORROW_MUT`, `DEREF`, `DEREF_MUT`

Function invocation is managed through the stack. Arguments are pushed, then `CALL` sets up a new frame. Return values are handled via `RET_VAL`. Since the size of each frame is known at compile-time, `LOAD` instructions are able to refer to local variables by their location in the stack.

Borrowing is implemented through the Reference Handling instructions. Borrow instructions push a pointer (as a stack-relative offset) to its operand, while the dereference instructions follow the reference at the top of the stack and push the referred-to value.

The below example shows an example compilation of Nust source:

```
; Function add(x: i32, y: i32) -> i32
LOAD 0
LOAD 1
ADD_I32
RET

; Function main
PUSH_I32 2
PUSH_I32 1
CALL 0
STORE 0
```

Objective Status

The repository can be found at `https://github.com/davidtranhq/nust-lang`. Instructions for building the system and tests are included in the README.md Over 30 tests across the parser, compiler, and type checker components are included. A feature implemented not mentioned in the specification is the recording of source locations in the AST for debugging and error message purposes. The first 3 objectives listed in *Scope* are functioning as specified. The last objective is incomplete.

Tests

Building and running the tests are described in the repository. We highlight some of the test cases of the parser, type checker, and compiler.

This test illustrates the different syntactic constructs of Nust supported by the compiler, including the optional mutable keyword, type declarations, references, mutable references, if blocks, function definitions, and assignment.

```
TEST(ParserTest, ComplexParseTree) {
    std::string source = R"(
        fn test() {
            let mut x: i32 = 42;
            let y: &mut i32 = &mut x;
            if (x > 0) {
                y = x + 10;
            }
        }
    )";

    // ...
}
```

We also include tests for borrow checking by the type checker:

```
TEST(TypeCheckerTest, InvalidImmutableToMutableBorrow) {
    std::string source = R"(
        fn main() {
            let x: i32 = 42;
            let a: &mut i32 = &mut x;  // Can't borrow immutable as mutable
        }
    )";
    // ...
}

TEST(TypeCheckerTest, MultipleMutableBorrows) {
    std::string source = R"(
        fn main() {
            let mut z: i32 = 10;
            let b: &mut i32 = &mut z;
            let c: &mut i32 = &mut z;  // Multiple mutable borrows
        }
    )";
    // ...
}

TEST(TypeCheckerTest, UseWhileBorrowed) {
    std::string source = R"(
        fn main() {
            let mut z: i32 = 10;
            let w: &mut i32 = &mut z;
            z = 20;  // Can't use z while mutably borrowed
```

```
        }
    )";
    // ...
}
```

and some basic test cases for compilation:

```
TEST_F(CompilerTest, FunctionCalls) {
    std::string source = R"(
        fn add(x: i32, y: i32) -> i32 {
            x + y
        }

        fn main() {
            let result: i32 = add(1, 2);
        }
    )";

    auto instructions = compile_source(source);

    // Expected bytecode for add:
    // LOAD 0
    // LOAD 1
    // ADD_I32
    // POP
    // RET

    // Expected bytecode for main:
    // PUSH_I32 2
    // PUSH_I32 1
    // CALL 0
    // STORE 0
    // RET
    // ...
}
```