Nust: A Statically-Typed, Reference-Safe Programming Language

David Tran

April 22, 2025

Outline

Overview

- Nust is a statically-typed, reference-safe programming language
- Four main components:
  - Parser
  - Type Checker
  - Compile
  - Virtual Machine
- Each component has specific responsibilities and implementation details

Design Goals & Scope

- **Parsing**: Implement a recursive descent parser to transform source code into an Abstract Syntax Tree (AST).
- **Type Checking**: Statically check the AST for type correctness, enforcing mutability and borrowing rules (annotate AST nodes with types or report errors).
- **Compilation**: Design and implement a compiler backend that translates the typed AST into bytecode for the stack-based Nust VM (NVM).
- **Virtual Machine**: Develop a stack-based virtual machine to execute the generated bytecode (manage stack frames, instruction execution).

Language Grammar (1/4)

```
<program>       ::= <item_list>

<item_list>     ::= <item> <item_list>
                  |

<item>          ::= <function_decl>
                  | <stmt>

<function_decl> ::= "fn" <ident> "(" <param_list_opt> ")" "{" <stmt_list> "}"

<param_list_opt> ::= <param_list>
                   |

<param_list>    ::= <param> "," <param_list>
                  | <param>

<param>         ::= <mut_opt> <ident> ":" <type>
```

Language Grammar (2/4)

```
<stmt_list>     ::= <stmt> <stmt_list>
                  |

<stmt>          ::= <let_stmt> ";"
                  | <expr_stmt> ";"
                  | <if_stmt>
                  | <while_stmt>
                  | <block>
                  | <return_stmt> ";"

<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>
                    |

<while_stmt>    ::= "while" "(" <expr> ")" <block>

<return_stmt>   ::= "return" <expr_opt>

<expr_opt>      ::= <expr>
                  |
```

Language Grammar (3/4)

```
<expr>          ::= <literal>
                  | <ident>
                  | <expr> <binop> <expr>
                  | <unop> <expr>
                  | <expr> "(" <arg_list_opt> ")"          // function call
                  | "&" <expr>                             // immutable borrow
                  | "&mut" <expr>                          // mutable borrow
                  | "(" <expr> ")"                         // grouping
                  | <ident> "=" <expr>                     // assignment

<arg_list_opt> ::= <arg_list>
                  |

<arg_list>      ::= <expr> "," <arg_list>
                  | <expr>

<binop>         ::= "+" | "-" | "*" | "/"
                  | "==" | "!=" | "<" | ">" | "<=" | ">="
                  | "&&" | "||"   // logical AND and OR

<unop>          ::= "-" | "!"

<mut_opt>       ::= "mut"
                  |
```

Language Grammar (4/4)

```
<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>              // immutable reference
                 | "&mut" <type>           // mutable reference

<ident>        ::= [a-zA-Z_][a-zA-Z0-9_]*
```

Parser

- Converts source code into Abstract Syntax Tree (AST)
- Handwritten recursive descent parser
- Key features:
  - Position tracking for error reporting
  - Scope management

Parser Implementation

```
class Scope {
public:
    std::weak_ptr<Scope> parent;
    std::vector<std::string> declarations;  // Variables declared in this scope

    explicit Scope(std::weak_ptr<Scope> parent = {}) : parent(parent) {}
};

class Parser {
    // Scope management
    std::shared_ptr<Scope> current_scope;
    std::shared_ptr<Scope> enter_scope();
    void exit_scope();

    // Parsing functions
    std::unique_ptr<FunctionDecl> parse_function();
    std::unique_ptr<Stmt> parse_statement();
    std::unique_ptr<Expr> parse_expr();
    // ...
};
```

Type Checker

- Ensures type safety and performs semantic analysis
- Key features:
  - Static type checking
  - Reference safety checks
  - Mutability tracking
  - Function signature validation
- Uses symbol table for variable tracking

Type Checker Implementation

```
class TypeChecker {
    // Scope management
    struct VariableInfo {
        std::unique_ptr<Type> type;
        bool is_mut;
    };

    std::vector<std::unordered_map<std::string, VariableInfo>> scopes_;

    // Borrow checking
    bool is_assignable(const Type& target, const Type& source) {
        if (target.kind == source.kind) {
            if (target.kind == Type::Kind::Ref ||
                target.kind == Type::Kind::MutRef) {
                return is_assignable(*target.base_type, *source.base_type);
            }
            return true;
        }
        // Allow implicit conversion from &mut T to &T
        if (target.kind == Type::Kind::Ref &&
            source.kind == Type::Kind::MutRef) {
            return is_assignable(*target.base_type, *source.base_type);
        }
        return false;
    }
};
```

Compiler

- Transforms AST into bytecode instructions
- Three-pass compilation:
  - Find main function
  - Build function table
  - Generate bytecode
- Handles local variables and control flow

Compiler Implementation

```
class Compiler {
    std::vector<Instruction> instructions;
    FunctionTable function_table;
    std::unordered_map<std::string, size_t> local_vars;

    void compile_function(const FunctionDecl* func);
    void compile_statement(const Stmt* stmt);
    void compile_expression(const Expr* expr);
    // ...
};
```

Virtual Machine

- Executes compiled bytecode
- Stack-based execution model
- Key features:
  - Function call stack
  - Reference management
  - Memory safety

VM Implementation

```
class VirtualMachine {
    std::vector<Value> stack_;
    std::vector<Value> memory_;
    size_t pc_ = 0;
    size_t fp_ = 0;

    void execute_instruction(const Instruction& instr);
    void handle_call(size_t operand);
    void handle_ret();
    // ...
};
```

Bytecode Format

- Stack-based instruction set
- Value Types:
  - `i32`: 32-bit signed integer
  - `bool`: Boolean value
  - `str`: String reference
  - `ref`: Reference to a value
  - `mut_ref`: Mutable reference
  - `fn`: Function reference
- Stack Frame Layout:
  - Return Address
  - Frame Pointer
  - Local Variables
  - Arguments

Instruction Categories

- Stack Operations
  - `PUSH_I32`, `PUSH_BOOL`, `PUSH_STR`
  - `POP`
- Variable Operations
  - `LOAD`, `STORE`
  - `LOAD_REF`, `STORE_REF`
- Control Flow
  - `JMP`, `JMP_IF`, `JMP_IF_NOT`
  - `CALL`, `RET`, `RET_VAL`

Instruction Categories

- Reference Management
  - `BORROW`: Create immutable reference
  - `BORROW_MUT`: Create mutable reference
  - `DEREF`: Access referenced value
  - `DEREF_MUT`: Access mutable reference

Function Calls

- Call Process:
  - Arguments pushed in reverse order
  - Return address and frame pointer saved
  - New stack frame created
  - Control transferred to callee
- Return Process:
  - Return value (if any) saved
  - Stack frame restored
  - Control returns to caller

Demo

- Test Suite:
  - Parser tests
  - Type checker tests
  - Compiler tests
  - VM tests
  - Integration tests
- Example Programs:
  - Function calls and returns
  - Reference safety
  - Control flow
  - Arithmetic operations

Outcome

- Successfully implemented a statically-typed, reference-safe programming language
- Some bugs still present
- Future work possibilities
  - Dynamically allocated functor objects
  - Collection types
  - Metaprogramming