Dissertation

Marcel Taubert (mt652)
20962335

School of Computing
MSc Advanced Computer Science
University of Kent

Supervisor: Dr. Michael Vollmer

August 7, 2023

1 Introduction

In the modern world we live in today, everything is controlled by code. Everyone of us is using technology controlled by code, if they want it or not. But not many people know how the code that programmers around the world write gets executed on their device.

1.1 What is an interpreter?

In general there are two ways to execute code. Both approaches include the process of translating the human readable code into something that the computer can understand.

During the first technique the code is compiled (translated) into machine code. The end product is a stand alone program that can be executed at any time.

The second approach is called 'interpreting'. In this case the compiler does not output machine code but produces a bytecode that will be executed by another program (the interpreter).

Interpreters are generally easier to implement and have some other advantages that makes them more approachable than compilers like portability or a fast edit-compile-run cycle as stated by the authors of vmgen [2].

1.2 What is a virtual machine interpreter?

A famous technique to implement interpreters is to build a virtual machine interpreter. A vm interpreter is generally divided into two systems. A frontend and a backend. [2]

The frontend consists of a compiler that takes the written code and produces a sequence of bytecode instructions. The backend is a virtual machine that gets the stream of bytecode instructions as input and executes them. [2]

The bytecode or intermediate representation used in a vm interpreter is usually designed to be very similar to a real machine. [2]

Some real world examples of virtual machine interpreters are for example the JVM (Java Virtual Machine) or the PVM (Python Virtual Machine). (TODO: link to them???)

1.3 Virtual machine

When we are talking about virtual machines we distinguish between stack based vm's and register based vm's.

Each of the version has it's advantages and disadvantages.

Stack based virtual machines are generally easier to implement than the register based approach. That shows for example in the complexity of the intermediate representation used in the virtual machine. The stack based bytecode has a tendency of being smaller and by that more efficient in comparison to the rather complex bytecode instructions of the register based approach.

```
typical stack based instruction:
Push 1    -- push value to the stack

typical register based instruction:
LD R1, 42 -- load value ‘42‘ into register R1
```

This simplicity is the reason we have to decided to rely on a stack based virtual machine in this paper.

1.4 Optimizations

When you are running your Java program the compiler does not just generate bytecode for the JVM from your written code. The compiler will try it's best to optimize as much as possible to ensure that the execution of the produced bytecode will be as fast as possible. This optimization part of the compiler is by far one of the most crucial tasks of a compiler.

During the development of this project we have looked at a number of different optimizations. Some of the optimizations that we have implemented for the project are threaded code and peephole optimizations (superinstructions) (TODO: SHOULD I EXPLAIN THEM HERE)

1.5 Automation

Writing an interpreter for a programming language can be a tedious and challenging task on it's own. Thinking about how to keep the code efficient and additionally implement optimizations makes it even harder.

One solution to this is automating the process of writing a virutal machine interpreter by providing a configuration file.

1.6 Objectives

This paper will use the Rust programming language to build each part of a virtual machine interpreter with the optimal goal of automating the generation of it self depending on a given configuration of a user.

We will explore techniques and approaches on how to build a vm interpreter in Rust and use benchmarking to visualize results depending on applied optimizations.

2 Description of the problem

2.1 Efficiency

In terms of efficiency at run time of a program nativ compiled machine code will always outperform the interpreted version. So why would we even care about writing efficient interpreters and not just use native code compilers.

One of the main reasons interpreters are preferred over compilers is that native code compilers are more complex to develop and difficult to maintain. [1]

Another big advantage of the interpreting approach is that compilers can only generate native code for one target system while the virtual machine interpreter stays the same on every system. By that the interpreter is portable and the generated code does not depend on the underlaying machine.

2.2 Automation

Many programmers will come to the idea of implementing their own programming language at some point in their career. Most of them will have noticed that buidling an interpreter is a challenging task and requires a lot of work and a clear structure. In addition to that it shows that many parts of an vm interpreter are similar and repetitive. For example the code for executing VM instructions will be similar for most of the instructions. [2]

But what happens when the interpreter does not give the expected outcome in terms of efficiency. It results in manual rewriting of a codebase just to change the implementation of some part of the vm interpreter to see if the performance increases. Rewriting the whole interpreter to test if the performance is better using a different development approach is not only time consuming but also error prone.

One solution to this problem is automation. The user should be able to provide a configuration file and based on that we will generate an efficient vm interpreter. It will already use efficient implementation techniques and come with built in optimizations. It also provides easy extensibility for the user without needing to change any source code.

3 Description of work

The goal of this project is to build an virtual machine interpreter and explore the implementation of such in the Rust programming language.

3.1 Tools used

During the development of this project we used three programming languages. We build the virtual machine and all parts of the interpreter in Rust and used Python to visualize any benchmarking results. The input language of the interpreter is the Imp programming language (see section 3.5). [3]

But why did we use Rust and not the C programming language like most other systems do? One reason for that is that as stated before most of the already existing solutions are written in C and we wanted to explore something new. Another reason why we decided to use Rust is that we were looking for a low level systems programming language that allows us to work as efficient as

possible while still offering high level features like match statements and iterators. An extension to that is Rust's expressive type system and strict memory management rules which helps avoiding many kinds of errors that appear in standard C programming.

3.2 VM

3.3 Interpreter

The interpreter consists of a scanner and a parser. We have decided to handwrite both parts instead of generating them. Even though the scanner and the parser are not the most performance critical parts of an interpreter we decided to handwrite them in Rust.

3.3.1 Scanner

The scanner was implemented very straight forward by using a match statement to iterate the source code and output a stream of tokens.

Tokens are defined as the token type and an optional literal.

```
pub struct Token {
    pub token_type: TokenType,
    pub literal: Option<Object>,
}
```

Token definition

```
pub enum Object {
    Num(f64),
    Bool(bool),
    Variable(String),
    DivByZeroError,
    ArithmeticError,
}
```

Object definition

```
pub enum TokenType {
    LeftParen,
    RightParen,

    Minus,
    Plus,
    ...
    Eof,
}
```

TokenType definition

3.3.2 Parser

The parser takes the stream of tokens as input and builds up an abstract syntax tree. In the Rust code it is represented as a ‘Vec¡Rc¡Stmt¿¿‘.

```
As an example the code ‘x := 10;‘ will get tokenized into
‘[Identifier,Assignment, NumberLiteral, Semicolon]‘
and the parser will output a tree structure like:

ExpressionStmt {
    expr: AssignExpr {
        name: "x",
        value: LiteralExpr {
            value: Num(10)
        }
    }
}
```

3.3.3 Interpreter (testing)

To be able to test the scanner and parser before writing the bytecode generator we build a small interpreter that just runs the code.

It uses the visitor pattern to traverse the abstract syntax tree generated by the parser and executes the code immediatly.

To use the visitor pattern approach we implemented a statement visitor and an expression visitor.

```
pub trait StmtVisitor<T> {
    fn visit_block_stmt(&self, stmt: &BlockStmt) -> Result<T, ()>;
    fn visit_if_stmt(&self, stmt: &IfStmt) -> Result<T, ()>;
    fn visit_expression_stmt(&self, stmt: &ExpressionStmt) -> Result<T, ()>;
    fn visit_print_stmt(&self, stmt: &PrintStmt) -> Result<T, ()>;
    fn visit_while_stmt(&self, stmt: &WhileStmt) -> Result<T, ()>;
}
```

Statement visitor example

For each statement we have setup a structure holding the necessary data.

```
#[derive(Debug)]
pub struct IfStmt {
    pub condition: Rc<Expr>,
    pub then_branch: Rc<Stmt>,
    pub else_branch: Option<Rc<Stmt>>,
}
```

Structure holding data for an if statement

3.4 Optimizations

3.5 Input language

References

[1] M. Ertl and David Gregg. The structure and performance of efficient interpreters. *J. Instruction-Level Parallelism*, 5, 11 2003.

[2] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: A generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, mar 2002.

[3] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *"Logical Foundations"*, volume "1" of *"Software Foundations"*. "Electronic textbook", "2023".