Critical Review of `vmgen`

Marcel Taubert (mt652)
20962335

School of Computing
MSc Advanced Computer Science
University of Kent

Supervisor: Dr. Michael Vollmer

August 6, 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 [1].

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. [1]

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. [1]

The bytecode or intermediate representation used in a vm interpreter is usually designed to be very similar to a real machine. [1]

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

A final objective of this paper is to automatically generate the code for a vm interpreter based on a config file that the user of the program has to provide. But more on this in

2 Critique

3 Synthesis

References

[1] 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.