Early Deliverable of `Generating Efficient Interpreters`

Marcel Taubert (mt652)
20962335

School of Computing
MSc Advanced Computer Science
University of Kent

Supervisor: Dr. Michael Vollmer

July 6, 2023

1 Introduction

Generating efficient interpreters is a project aiming to help programming language developers to be able to quickly build interpreters for their language in a few simple steps.

At the time the program consists of a compiler that takes the source code of the programming language described in the book (TODO) Imp. and outputs bytecode. The structure of the bytecode and the syntax of the language it self will be described in a later section of this paper (TODO ADD REFERENCE).

If you have ever written multiple compilers or interpreters yourself you might have realized that many parts of the program are very repetitive and not too depended on the syntax of the language that you're building the interpreter for. Many parts like the code generation (TODO) are very similar every time and can be generated quite easily. In addtion to that the user can provide a configuration file that describes possible missing parts in detail and modify the generation without writing any code.

Another part that can be automated is optimizations. We can provide multiple optimizations and let the user define their own in the configuration file. The user then can choose which of the optimizations they want to include in their generation. It is easy to see what optimizations work the best or worst by just generating multiple versions and running benchmarks between them. All of this can be archived without needing to write a lot of code.

2 Technical description

Currently the compiler is written to work with source code of the programming language "Imp". This is a simple imperative programming language which uses just a few keywords and a clear syntax which makes it perfect to start with in this project. The future work section of this paper will go into detail about how we can extend this project to work with a real programming language.

This is an example of Imp.

```
Z := X;
Y := 1;
while Z != 0 do
    Y := Y  Z;
    Z := Z - 1;
end
```

The output of the compiler is a custom bytecode. It is designed to work with a stack-based virtual machine.

The compiler itself is implemented in the Rust programming language. The following is the representation of a single bytecode instruction as a Rust enum.

```
pub enum ByteCode {
    Push(usize),
```

```
    Pop,
    Add,
    Sub,
    Mul,
    Var(String),
    Eq,
    NEq,
    Lt,
    Gt,
    Lte,
    Gte,
    And,
    Or,
}
```

At the moment the complier is held pretty simple and not all the features that are possible in "Imp" are implemented.

Currently none of the control flow featrues are implemented since we're missing instructions for conditional jumps.

For debugging convenience we've also added a new keyword ('print') to the language which prints the content of the given variable.

Since "Imp" does not have scope management we store all variables in a global table held in the bytecode generator structure in the rust code.

3 Future Work

Since this project is still in the early development phase there is a lot still missing until we have a working version that can be benchmarked.

The next step is to implement conditional jumps and by that being able to compile programs that use if-statements or loops.

Then we have think about different optimizations that we want to implement and build them.

At this point we have a fully working compiler for the "Imp" programming language and we can start benchmarking the performance of it.

One step to getting closer to work with a real programming language is to implement scope management.

Then we can think about modifying the compiler to work with Lua for example.

References