Design and implementation of a programming language with LLVM

Josep Maria Domingo Catafal

Abstract– This thesis presents the development of a new programming language using Rust and LLVM. The goal of the project is to gain a deeper understanding of how compilers work by creating one from scratch using these tools. The language is designed to be simple and easy to understand, but at the same time, it aims to be fast and efficient, so some sacrifices have to be made. The thesis covers the design and implementation of the language, including the lexer, parser, and code generation. The project also includes a discussion of the challenges encountered during development and suggestions for future work. Overall, the project serves as a valuable learning experience for understanding the inner workings of compilers and the capabilities of Rust and LLVM.

Keywords– Programming Language, LLVM, SSA, Strongly Typed, Compiled

Resum– Resum del projecte, màxim 10 línies.

Paraules clau– Llenguatge de programació, LLVM, SSA, Fortament tipat, Compilat

1 Introduction

HISTORICALLY, there has always been a dilemma between the speed of execution, and speed of development. Some languages are easy to program: they allow the programmer to not worry about low-level concepts such as memory management, and create abstractions that streamline the development. The problem is that these abstractions limit language efficiency, and create slower programs. Another reason that allows speeding up development is dynamic typing, as it frees the user from the mental overhead that comes with deciding the type that should be used. But this also has its own disadvantages, since it is very likely that you will encounter errors in timing of execution These languages tend to be interpreted in order to save money it waits for the programmer, but it has an impact on the execution performance of the program if we compare it to compiled languages. Python, for example, would be one of the largest representatives of this group of languages. On the other side of the currency and we have languages like C, which have almost no abstraction and the programmer must be aware of what he is doing in every line of code he writes. they are languages that provide very good performance, but slow down the development, as the programmer must take into account many concepts of low level An additional problem is that when managing shape memory manual, it opens the door to a lot of runtime errors in the form of memory leaks. Currently, there are languages like Rust that solve this these memory management issues without losing performance, but the development is still slow and the compilation time long. these languages tend to be strongly typed, which reduces errors in runtime, but slows down development.

• E-mail de contacte: jdomingocatafal@gmail.com
• Menció realitzada: Computació
• Treball tutoritzat per: Javier Sanchez Pujadas (Ciències de la Computació)
• Curs 2022/23

2 Goals

3 State of the art

3.1 Programming Languages

3.2 Compilers

4 Methodology

5 Development

6 Architecture

Most compilers are divided into two parts: the front-end and the back-end. The front-end is the part of the compilers that takes the source code and transforms it into an intermediate representation than will later be transformed into the actual machine code by the back-end. In our case, since we are using LLVM, we don't need to worry too much about the back-end, since LLVM will be in charge of generating the machine code. Out job will be to go from the source code to the LLVM intermediate representation (we will call it ***IR*** from now on). The front-end of the compiler is typically composed of three main steps.

6.1 Lexing

This step consists of breaking the source code into a sequence of tokens. A token is a basic building block of the languages, such as a keyword or an identifier.

A lexer can be implemented rather easily, by using a state machine. An approach to do it programmatically could be the following:

1. Start by reading the source code character by character until we reach the end.

2. If the character by itself forms a valid sequence (e.g. a parenthesis) we create a token from it. If it doesn't, we continue reading characters until we find a valid sequence.

Note that sometimes we may find a valid sequence, but that's not enough to create a token, since it may be the start of another longer and valid sequence. We may need to check the following character/s to check weather it continues. An example of such case would be the '>' operator, since, by itself is a valid sequence, but it may be the start of the '>=' operator. So we need to check if the next character is an '=' or something else.

The lexer requires a bit of work to set up, but after that, expanding it is trivial, since we only need to add a new word to the list of reserved words, in the case we want to add a reserved word, or add a new rule that detects a new symbol for example.

6.2 Parsing

The lexer allowed us to identify the simbols of the program, but it does not allow us to determine if their order is correct, or if they follow the rules of the language (i.e. it's grammatically correct). That is the job of the parser.

The parser takes the sequence of tokens obtained from the lexer and transforms it into an Abstract Syntax Tree (AST). This tree represents the structure of the program and determines its syntactic structure. It will tell us the order in which we need to execute the instruction.

To generate the AST compilers use a context free grammar. A grammar is a set of rules that tells us how to form valid strings of tokens in a specific language. They are formed of a set of symbols, which can be divided into terminal and non-terminal symbols, and a set of production rules that specify how the non-terminal symbols can be replaced by sequences of terminal and non-terminal symbols. A context free grammar is a type of grammar that, the rules of the grammar do not depend on the context in which the symbols appear.

The goal of the parser is to make the program obey the rules of the grammar.

```
<proto>   ::= fn <id> "(" <params> ")"
<id>      ::= letter {letter | digit | "_"}
<params>  ::= "("{ <param> {, <param> } }")"
<param>   ::= <id>: <type>
```

```
fn parse_prototype() -> (Prototype, Err) {
    // we expect to find the fn keyword,
    // else it's an error
    match current_token().kind {
        // The advance function moves to the next token
        TokenKind::Fn => advance(),
        _ => return Err("Expected fn keyword"),
    };

    // we expect to find the function name,
    // else it's an error
    let name = match current_token().kind {
        TokenKind::Identifier => current_token().lexeme,
        _ => return Err("Expected an identifier"),
    };

    advance();

    // we call the params rule
    let params = parse_params();

    // we are done, we return a struct
    // with the info of the prototype
    return Prototype {
        name,
        params,
    };
}
```

6.3 Semantic Analysis and [IR] Code Generation

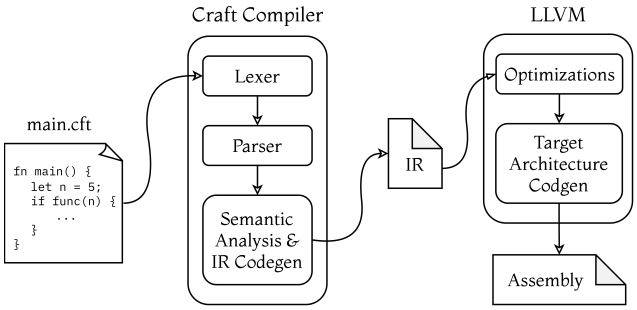

Fig. 1: Diagram showing the compilation workflow of a Craft program

7 Results

8 Conclusions

References

[1] http://en.wikibooks.org/wiki/LaTeX

[2] Referència 2

[3] Etc.

Apèndix

A.1 Secció d'Apèndix

....

A.2 Secció d'Apèndix

....