Basic C Interpreter

Michael Covarrubias

March 13th 2020

Abstract

My project was initially to interact with the Amazon Alexa (software development kit) SDK, however, I made a change to implement a basic C interpreter due to its relevance to programming languages. Through my research I hoped to gain a better understanding of the underpinning of the C languages formal and operational semantics. Also, to understand how to appropriately implement a subset of the different expressions, declarations, and scope of the language. This is not a replacement for compilers such as GCC, however just an abstract interpreter implementation to understand the foundation of how the C language is interpreted.

1 Introduction

Motivation for this project was to submerge in a profound understanding of new concepts learned in the programming languages course. Concepts of the operational semantics of a language, and the formation of abstract syntax trees when parsing a language. Assignments in the course introduced the idea of interpreters, however, implementation was only applied to the WHILE language. A language not widely used in the realm of computer science. Therefore, I wanted to gather a deeper understanding of programming languages topics as to how they relate to a popular general-purpose programming language such as C. And in doing so I would add to my formal reasoning about C programs, a valuable tool in software development when using C in practice. There has been much development of the C language since its inception in the Bell Labs by Dennis Ritchie between 1972 and 1973.

From first being developed in order to make utilities running on Unix, to becoming a widely used programming language standardized by the ANSI since 1989, I believe there is much to learn on the progressive development of C. Related material for C has been closer to compiler development rather than interpreters. Even through interpreters and compilers are similar in structure, they have some key differences. Interpreters directly execute the instructions in the source programming language such as C, while a compiler translates the instructions into efficient machine code. Portability is highly kept in mind for the C language. Therefore there has been extensive work on compilers for C in order to be utilized in a wide variety of computer platforms and operating systems while only introducing minor changes to the source code.

2 Related Work

C is a general purpose procedural language that is capable of supporting paradigms such as lexical variable scope and recursion. During the 1980's C slowly gained popularity and became one of the most widely used programming language. It is due to this sole development that has made C a studied language. Resulting in informal semantics of ANSI C defined in standard [4]. A source of ambiguity in interpretation, but an excellent source in determining my subset of defined syntax. It is these ambiguities within the C language that has influenced the development of formal semantics of C. Ellison and Rosu's recent work describes an executable semantics of C that has been tested against the GCC torture test suite, passing 99.2% test programs. This is the most complete and thoroughly tested formal definition of C [2]. This formal semantics of C, shows capability of automatically finding program errors both statically and at runtime.

3 Abstract Syntax

Initially, I was to implement both the parsing and interpreting of the C language, with my development being in Python. Through trial I quickly started realizing the difficulty in creation of a Lexer and Parser for a basic subset of the C language. In hopes to finish the interpreter, I then turned to utilizing the Python library PyCParser.

Though I started utilizing PyCParser to parse C code into an abstract syntax tree (AST), I still spend extended periods of time on implementation and design of the syntax my interpreter would evaluate. I needed to troubleshoot the AST's formed from PyCParser in order to effectively design an abstract interpreter.

Figure 1. A sample C program

```
int addTwo(int x, int y) {

    int sum = x + y;

    return sum;
}

int main() {

    int one = 1;
    int two = 2;

    int val = addTwo(one, two);

    return val;
}
```

The abstract syntax tree relating to the C code above, can be seen below. AST was generated through the use of PyCParser.

Figure 2. AST generated via PyCParser

```
FileAST:
  FuncDef:
    Decl: addTwo, [], [], []
      FuncDecl:
        ParamList:
          Decl: x, [], [], []
            TypeDecl: x, []
              IdentifierType: ['int']
          Decl: y, [], [], []
            TypeDecl: y, []
              IdentifierType: ['int']
        TypeDecl: addTwo, []
          IdentifierType: ['int']
    Compound:
```

```
      Decl: sum, [], [], []
        TypeDecl: sum, []
          IdentifierType: ['int']
        BinaryOp: +
          ID: x
          ID: y
      Return:
        ID: sum
  FuncDef:
    Decl: main, [], [], []
      FuncDecl:
        TypeDecl: main, []
          IdentifierType: ['int']
    Compound:
      Decl: one, [], [], []
        TypeDecl: one, []
          IdentifierType: ['int']
        Constant: int, 1
      Decl: two, [], [], []
        TypeDecl: two, []
          IdentifierType: ['int']
        Constant: int, 2
      Decl: val, [], [], []
        TypeDecl: val, []
          IdentifierType: ['int']
        FuncCall:
          ID: addTwo
          ExprList:
            ID: one
            ID: two
      Return:
        ID: val
```

I needed to determine the different type of nodes that could be encapsulated in the an AST. I started focusing on the complexity in formation of the C language. This had me develop different structures for the syntax of the C language. Initially starting with declarations, I needed to define the different type of declaration for my basic C interpreter. As I progressed on what my interpreter would support, I resulted in capability of declaring functions and type declarations, where a type declaration is the declaration of a variable to a value or nothing (e.g., int x = 9;).

Figure 3. C interpreter declarations syntax

```
Declaration ::= TypeDeclaration | FunctionDeclaration
```

Next, was determining the different types to support. With C having a vast type set, (e.g., signed char, unsigned char, short int, long int, double, long double, etc) I chose to prune the amount of interpretable types chars, floats and integers.

Figure 4. C interpreter type list

```
type-specifier ::= char | int | float
```

The abstract syntax supported for the different type of statements are also a trimmed down set of standard C. Statements such as if, while for and return are included as evaluable. The abstract syntax for statements are:

Figure 5. C valid statements for interpretation

```
Statement ::= expression
    | if (expression) Statement
    | if (expression) Statement else Statement
    | while (expression) statement
    | for (expression; expression; expression) Statement
    | Var = expression
  | return expression;
  | [Statement]
```

3.1 Memory

The majority of difficulty in my implementation was through the introduction of functions and function calls. With functions being included in the semantics and syntax, this introduced the concept of scope. Scope is a region of the program, and the scope of variables refers to the area of the program where these variables can be accessed once they are declared. Therefore I needed to implement different levels of scope. The first was the global scope (e.g., File scope), this is the top level scope where all declarations of functions in a file (e.g., test.c) are visible.

Functions introduced Function Scope as well, where every declaration is valid only for that block. Where the containing block is within a function. Therefore this results in scope to be a part of a function. A scope that ends once a function returns. In the case of the C language, scope of the function is created when called, and destroyed at return. C does provide static local variables, where the lifetime of the variable lives on throughout the entire life of the program, however, I decided to not include this ability into the syntax of my simple C interpreter. For implementation of scope and memory, I needed to further my understandings of the C language stack and frames.

To keep track of different scopes of a C file being interpreted I created A memory class that within itself had an implementation of a stack. This call stack is then composed of stack frames, where each stack frame is a call to a subroutine which has not yet terminated with a return. For example, a C program with only a 'main' function would add a stack frame to the stack to keep the main function block in scope, while the code within the main function continues executing. This is so all variables declared within the main function are accessible, throughout main function execution. A code snippet of the stack implementation utilized in keeping function calls and scope in order can be seen in Figure 6 below.

Figure 6. tack utilized for function calls.

```
"""
    Stack implementation utilized to keep track of different
    function calls. usage allows to keep track of scope when
    interpreting C code
"""
from frame import Frame


class Stack(object):
    """
    Stack implementation
    """
    def __init__(self):
        self.current_frame = None
        self.frames = []

    def add_frame(self, name, global_scope):
        """
        Function used to add a new frame
        :param name:
        :param global_scope:
        :return None:
        """
```

```
        curr_frame = Frame(name, global_scope)
        self.current_frame = curr_frame
        self.frames.append(self.current_frame)

    def del_frame(self):
        """
        Function used to delete a frame
        :return None:
        """
        self.frames.pop(len(self.frames) - 1)

        if len(self.frames) > 0:
            self.current_frame = self.frames[len(self.frames)
    - 1]
        else:
            self.current_frame = None
```

4 Model Checking

Once the utility of scope and function calls was developed, I felt confident in testing different interpretations of C, within the scope of my defined syntax subset. I then created example C programs and validated their exit code's. This was done to validate the ability to execute statements (e.g. For loop, While loop, If statement, assignment), declarations (e.g. type declarations, function declarations), and function calls. usage of printing was not included in the checking of my interpreter exit status versus the gcc compiler since I did not include printf as part of the syntax and semantics of my basic C interpreter.

To automate the checking of my interpreter with a varying configurations of declaration and statements, I created a test python script. This test script was used to collect all .c files in my examples directory. The tests were composed similar to the assignment test code developed by our teaching assistant Sohum. This was the crux of my validation of my implementation, bringing rise to different bugs in interpretation of statements such as for loops.

5 Conclusion

Course CSE 210A: Programming Languages was a very difficult course. We may not of indulged in implementation of compilers, but the concepts such as interpreters and abstract syntax tree's added a new level of complexity of my understanding in programming languages. In short, interpreters are hard. We swiftly take it for granted the years of development on interpreters and compilers created for widely used programming languages such as Python, Java, JavaScript, C and C++. This was my first attempt in creation of a module written in python, to be considered anything close to an interpreter of the C language. Even with reducing my scope in implementation, by including only a subset of the C language syntax, there was still a vast number of complexities throughout development. Important concepts such as the memory management, and scope of functions. While writing software in C, we take it for granted of the underpinnings and idiosyncrasy's in effectively and correctly maintaining scope. So now, instead of programming languages paradigms being a black box, I have new insight and much respect for the novel developers and researcher in the field. I hope to continue working on my abstract C interpreter, this project has given me the opportunity to submerge in the complex semantics and syntax of the C language. A core language in its impact to computer science as a whole. As for my project I hope to continue implementing more difficult syntax and semantics of C, such as arrays and the ability to interpret the including of external libraries. I would like to continue with the ability to interpret structures, and how to manage struct's in different scopes. Lastly, I am very content with my decision in creation of an interpreter. Yes, it was about the most difficult project I have worked on. Partly due to the concepts and way of thinking that go into the semantics of a programming language which was new to me. However, there was a vast amount of knowledge absorbed on my part. Underlining understandings that will benefit my future developments in C and C++.

There are on going issues that still need to be correctly implemented. There are issues with there being more than one return statement in a function. For example, having an if-else statement that has a return value for both the true or false statement block. There is also a lack in ability to print any useful information via my interpreter. The future addition of being able to print in .c script would allow far more complex tests to be generated in order to really cut down bugs and test corner cases.

6 References

6.1 Literature

[1] Blazy S., Dargaye Z., Leroy X. (2006) Formal Verification of a C Compiler Front-End. In: Misra J., Nipkow T., Sekerinski E. (eds) FM 2006: Formal Methods. FM 2006. Lecture Notes in Computer Science, vol 4085. Springer, Berlin, Heidelberg

[2] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. SIGPLAN Not. 47, 1 (January 2012), 533–544. DOI:https://doi.org/10.1145/2103621.2103719

[3] Nikolaos S. Papaspyrou, A Formal Semantics for the C Programming Language, Doctoral Dissertation, National Technical University of Athens, Department of Electrical and Computer Engineering, Division of Computer Science, February 1998.

[4] Nikolaos S. Papaspyrou, "Denotational Semantics of ANSI C", Computer Standards and Interfaces, vol. 23, no. 3, pp. 169–185, July 2001

[5] International Organization for Standardization, New York,NY. ISOrIEC 9899-1999, Programming Languages: C, 1999.

6.2 Code and Online References

https://ruslanspivak.com/lsbasi-part1/
https://github.com/knowknowledge/Python-C-Parser
https://github.com/eliben/pycparser
http://hdl.handle.net/2142/34297
https://www.craftinginterpreters.com/resolving-and-binding.html

7 Appendix

7.1 Source Code

https://github.com/mdcovarr/interpreter