National Institute of Technology Calicut

Principles of Programming Languages

TERM PAPER

Study on the Formal Semantics and Type System of C Language

Name:
Bhukya Vasanth Kumar

Details:
B180441CS SEM 7 : CSE

Date:
28th October 2021

Abstract: Formal semantics is a method of studying meaning that has roots in logic, linguistics, for a language. It is normally based on the inference rules and some mathematical logical systems in-order to overcome ambiguities. A type system defines the structure of a program. It is a mechanism for defining, detecting, and preventing illegal program states. This mechanism works by defining and applying some specific constraints or conditions. The constraint definitions are types, and the conditional applications are usages of types. Example: identifier initialisation. In this paper, C Programming language has been studied to explain the formal specification of semantics and typing rules for various constructs.

Key-words: Syntax, Semantics, Type System

1. Introduction

This Term Paper is a brief report on the study of Semantics and Type System of the C Programming Language. This term paper will explain about the programming language and explain semantics and typed systems. Later on, a detailed study about few constructs in the C Language is made which is further explained with the help of semantics, types and typing rules.

1.1. The C Programming Language

C Programming language is one of the powerful and widely used languages in the world. It was developed in the early 1970s at the AT&T Bell Labs. Initially, it was developed as a system implementation language for the Unix operating system. C gives you direct access to memory and many "low level" computer operations. Also, the source code is compiled into stand-alone executable programs. By maintaining all the fundamental functionalities, it forms the base for the modern programming languages. Therefore, C can be accepted as a common basis for other languages and is a point of reference by the developers, other tools and implementations.

1.2. Semantics

The study of programming languages lies between two fundamental features:
SYNTAX AND SEMANTICS

- **Syntax** is mainly concerned with the structure of the well formed sentences of the language. The syntax of a programming language is usually specified.
- **Semantics** refers to the meaning of these sentences which should be preserved by some compilers or interpreters of other languages. It is concerned with specifying the perfect meaning or behaviour of programs or some pieces of hardware. We need the perfect meaning because semantics can reveal ambiguities in documents like manuals. Semantics can form the basis for implementation, analysis and verification. Proof of correctness mainly depends on semantics.
- Formal syntax is well known and there are various standard formalisms for this purpose. CFG (Context Free Grammar) is used a lot which has the form, BNF (Backus Naur Form). Grammar supports connection between syntax and parser implementation, thus it has been easier to develop the formal syntax specification. Semantics for programming languages are specified in an informal way due to the complexity of tasks. Such semantics are normally ambiguous and may depend on the reader's knowledge.

1.2.1 Types Of Semantics

- **Operational semantics:** The meaning of a particular construct is given by the computation it induces when it is executed on a machine. Structural operational semantics are a more systematic variation.
- **Denotational semantics:** Here, meanings are modelled by mathematical objects by mapping inputs - outputs.
- **Axiomatic semantics:** Specific properties of the effect of executing the constructs are expressed as assertions. Here, meanings are given indirectly in terms of some propositions by explaining properties of the programs.

1.3. Examples

1.3.1 Operational Semantics

1. int a=2;
2. int b=3;
3. int temp=a;
4. a=b;
5. b=temp;

Semantics: In the above set of statements, we are swapping two numbers. In the step1, We are assigning the value 2 to id 'a'. In the step2, we are assigning the value 3 to id 'b'. In the step3, we are assigning the value of id 'a' to id 'temp'. Therefore id 'temp' now holds the value 2. In the step4 , we are assigning the value of id 'b' to id 'a'. Therefore id 'a' now holds the value 3. In the step5 , we are assigning the value of id 'temp' to id 'b'. Therefore id 'b' now holds the value 2.

1.3.2 Axiomatic Semantics

We explain the semantics of a program by including statements that are always true when the program's control reaches the assertions' points. Accuracy of a command is stated in particular by inserting one assertion, referred to as a precondition, before the command and another assertion, referred to as a postcondition, after it.

{ PRE } C { POST }

1. char ch='e'
2. ch++;
3. printf("%c",ch);

Precondition: char ch='e', Here we are assigning character 'e' to identifier ch
Command : ch++, ch becomes ch+1
Postcondition: ch will be displayed which is 'f' will be displayed

1.4. Type System

Type system deals with the types in languages and rules to assign the types in language constructs. We need a type system to statistically check the codes in order to avoid certain run-time errors. In the formal specification of programming languages, typing semantics is used to define aspects that cannot be represented by a context-free grammar. The association of phrases participating in syntactically well-formed programs with phrase types is the primary goal of a language's typing semantics. In this method, useful information about the meaning of program terms is provided. Typing semantics can often totally replace the language's abstract syntax in simple programming languages.

2. Literature Survey

[1] explains about semantics and types of semantics. Various examples can be seen in a step by step manner for each semantics. A part of [2] explains about Type system and it's advantages along with the language safety. It explains briefly about detecting errors as well. Various resources mentioned about the history of C language and the uses of it. [3] explains the syntax that has been fixed for C. The working of Axiomatic semantics is also studied from [1]. I've tested various constructs from [3] to know the edge cases and to know more about the expression type for few operations like assignment, addition and for selection statements like if else, etc. [6] briefly explains how a semantics looks like.

3. Language Definition

The syntax of C in Backus-Naur Form [3]

<declaration-specifier>:: <storage-class-specifier> | <type-specifier>

<selection-statement>:: if (<expression>) <statement>
| if (<expression>) <statement> else <statement>
| switch (<expression>) <statement>

<iteration-statement>:: while (<expression>) <statement>
| do <statement> while (<expression>) ;
| for (<expression>? ; <expression>? ; <expression>?) <statement>

<labeled-statement>:: <identifier> : <statement>
| case <constant-expression> : <statement>
| default : <statement>

<jump-statement>:: goto <identifier> ;
| continue ;
| break ;
| return <expression>? ;

4. Semantics

4.1. If-Then-Else Construct

```
IF ( expression ) THEN {
        Statements // a statement or a block of statements
}
ELSE {
        Statements // a statement or a block of statements
}
```

The IF construct calculates the logical values of the provided expression

1. If the expression evaluates to 0 or the null string, it is false;
2. if it evaluates to anything else, it is true.
3. If the expression is true, the statements after THEN will start running;
4. if the expression is false, the statements after the ELSE will be executed, or if
5. there is no ELSE component, program execution will continue with the next executable statement.
6. The THEN clause and the ELSE clause are both optional, but one of them must be used.

4.2. Switch-Case-Default Construct

```
Switch (Expression){
        Case (Expression) : Statements ( break Statement is optional)
        Case (Expression) : Statements
        Default :       Statements // optional
}
```

1. **SWITCH:** The expression at Switch must evaluate to integer type to return true.
2. **CASE:** The CASE construct evaluates a series of conditions until one is true and executes a set of statements accordingly. The expressions in the CASE statements are evaluated sequentially for their logical value until a value of true is encountered. When an expression evaluates to true, the statements between the CASE statement and the next CASE statement are executed, and all subsequent CASE statements are skipped.
3. **DEFAULT:** This is optional. No action is taken if none of the expressions evaluate to true, and program execution continues with the statement after the switch closes.

4.3. Do-While Construct

Do {
Statement1
}
WHILE(expression)

The WHILE construct calculates the logical values of the provided expression.

1. The DO makes sure the statement(s) after DO are executed atleast once. Thus, initially it is executed.
2. If the expression at WHILE evaluates to 0 or the null string, it is false;
3. if it evaluates to anything else, it is true.
4. If the expression is true, the statements after DO will start running; And it will again go to 2 after the statements are executed.
5. if the expression is false, program execution will continue with the next executable statement. i.e he statements after DO will no longer execute, and the control is sent to whatever statement follows after the expression
6. The WHILE clause may/ may not have the DO Clause. But DO Clause needs WHILE.

4.4. Return Construct

4.4.1 Return

Return transfers control from a subprogram to the calling program unit. Whenever a RETURN statement is executed in a sub program, the control is given to the referencing statement in the calling program. it does not specify the returned value, thus can be used in any function, provided that the returned value is not used.

4.4.2 Return (Expression)

Return transfers control from a subprogram to the calling program unit. Whenever a RETURN statement is executed in a sub program, the control is given to the referencing statement in the calling program. This form of return statement determines types of all statements. It can be attributed to different types based on the subprogram it is in.

4.5. Properties

1. Stuck term cannot be evaluated.
2. Termination of a C program depends on the logic of the program. Given the program is not syntactically or semantically valid, the program terminates with error. A valid program would end without errors. There can be infinite sequence of steps where the function just evaluates to itself and evaluation will never terminate.
3. Uniqueness of Normal Forms holds, you can perform the reductions in any order, but you'll always get the same result.

5. Type System

5.1. Set of Types

C is a statically typed language. All variables must be declared as specific data types. The main disadvantage to this is that due to the declarations, a source code is a bit longer and it takes more time to write it. The main advantage is that before running, our compiler checks whether all of the data types match.

The following are Type Specifiers [4]:

1. void : means nothing or no value i.e. empty type
2. char : Stores single character, signed and unsigned version
3. short : Stores integer with size 2 bytes
4. int : Stores integer with size 4 bytes
5. long : Stores integer with size 4 bytes
6. float : Stores floating value i.e. with decimals with size 4 bytes
7. double : same as float with size 8 bytes

8. signed : can hold both negative and positive values
9. unsigned : can hold only positive values
10. array : stores data type in continuous memory allocation
11. function : Describes functions with return type
12. pointer : Points/ refers to objects
13. <struct-or-union-specifier> : stores various data types
14. <enum-specifier> : Assigns integral constants
15. <typedef-name> : Alternative names to existing types

Functions related to types [5] :

1. *isIntergal:* It is used to identify integer types, character types and also enumeration types. It returns true if expression is Integral or else it returns false.
2. *intPromote:* It works on integral promotions. Char, Signed Char, short-int, unsigned short-int, enum etc are promoted to integer.
3. *isScalar:* It is used to identify arithmetic and pointer types. It returns true if expression is Scalar or else it returns false
4. γ: It can be used to make assumptions in order to make the typed system. It is a set that contains all the terms of free variables.

5.2. If-Then-Else Construct

$$\frac{\gamma 1 \vdash expression : exp[T1] \quad isScalar(T1) \;\; \gamma 1 \vdash statement1 : stmt[T2] \quad \gamma 2 \vdash statement2 : stmt[T2]}{\gamma 3 \vdash if(expression) \;\; statement1 \;\; else \;\; statement2 : stmt[T2]} \tag{1}$$

Here, we have 3 different gammas to represent three different sets of assumptions needed for various statements. The expression (Type T1) must evaluate to **isScalar** and the type of statement1 and statement2 must be some type T2. Then, the construct would have the type T2.

5.3. Switch-Case-Default Construct

$$\frac{\gamma 1 \vdash expression : exp[T1] \quad isIntegral(T1) \quad \tau" \;\; := \;\; intPromote(T1) \;\; \gamma 2 \vdash statement : stmt[T2]}{\gamma 3 \vdash \textbf{switch}(expression) \;\; statement : stmt[T2]} \tag{2}$$

$$\frac{\gamma 1 \vdash constant - expression : val[T1] \quad isIntegral(T1) \quad \gamma 2 \vdash statement : stmt[T2]}{\gamma 3 \vdash \textbf{case:} \;\; constant - expression : \;\; statement : stmt[T2]} \tag{3}$$

$$\frac{\gamma 1 \vdash statement : stmt[T1]}{\gamma 2 \vdash \textbf{default:} \;\; statement : stmt[T1]} \tag{4}$$

1. **SWITCH:** The expression of type T1, inside the switch must evaluate to **isIntegral** and **intPromote** and given that the statement is of type T2, the construct would also be of type T2.
2. **CASE:** The expression of type T1 should evaluate to **IsIntegral** and the expression must be constant. Then the construct would be of type T2, given the statement if of type T2. We can have many cases and in each case, break statement is optional.
3. **DEFAULT:** We use γ for set of free variables. The default statement is optional. The type of construct would be T1 if the statement has same type.

5.4. Do-While Construct

$$\frac{\gamma 1 \vdash statement : stmt[T2] \;\; \gamma 2 \vdash expression : exp[T1] \quad isScalar(T1)}{\gamma 3 \vdash \textbf{do} \;\; statement \;\; \textbf{While} \;\; (expression); : stmt[T2]} \tag{5}$$

Here, we have 3 different gammas to represent three different sets of assumptions needed for various statements. The expression must evaluate to **isScalar**. Consider that the type of expression is T1, which is being evaluated in *isScalar*. Also, considering that the Type of Statement is T2, the type of construct would be type T2.

5.5. Return Construct

5.5.1 Return

$$\frac{1}{\gamma 1 \vdash \textbf{return ;} : stmt[T1]} \tag{6}$$

5.5.2 Return (Expression)

$$\frac{1}{\gamma 1 \vdash \textbf{return } (expression); : stmt[T1]} \tag{7}$$

5.6. Properties

1. Type Safety normally implies the range that the programming language discourages or prevents type errors. The standard type system in C does not rule out programs that are considered nonsensical by normal practice. C can have many constructs as undefined behavior. (TYPE-SAFETY=PROGRESS + PRESERVATION)
2. Even if the syntax in C goes statically valid, it might still throw a run time error after a series of execution. Thus, Progress will not hold. (PROGRESS)
3. C Language holds preservation property. If a well-typed term takes a step of evaluation, then the resulting term is also well typed. (PRESERVATION)

6. Conclusions

In this term paper, the work deals with understanding the history of C Programming language followed by understanding the means of syntax and semantics along with few examples. A brief information about Type System is also provided. There has been a thorough study on various topics related to the term paper, which has been included in the Literature Servery. The basic language definition is provided Later on. Few constructs are taken from the C Programming language and their semantics and properties are explained. The typing system and required functions are given. For the same semantics that has been mentioned earlier, typing rules are given and explained. The references are provided at the end.

References

1. https://faculty.sist.shanghaitech.edu.cn/faculty/songfu/course/spring2018/CS131/sa.pdf
2. https://theswissbay.ch/pdf/Gentoomen%20Library/Maths/Comp%20Sci%20Math/Benjamin_C._Pierce-Types_and_Programming_Languages-The_MIT_Press%282002%29.pdf
3. https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm
4. https://www.javatpoint.com/data-types-in-c
5. https://www.researchgate.net/publication/2843680_A_Formal_Semantics_for_the_C_Programming_Language
6. https://chortle.ccsu.edu/java5/Notes/chap15/ch15_6.html