CS4022D:

PRINCIPLES OF PROGRAMMING LANGUAGES

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

तमसो मा ज्योतिर्गमय

TERM PAPER

STUDY ON THE FORMAL SEMANTICS AND TYPE SYSTEM OF C LANGUAGE

Bhukya Vasanth Kumar
B180441CS
(A Batch)

TABLE OF CONTENTS

1. ABSTRACT & INTRODUCTION

- 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. A type system defines the structure of a program. It is a mechanism for defining, detecting, and preventing illegal program states.
- In the type system, 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.
- C Programming language is one of the powerful and widely used languages in the world. C gives you direct access to memory and many "low level" computer operations. C can be accepted as a common basis for other languages and is a point of reference by the developers, other tools and implementations

- *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.
- 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). 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.

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. 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.

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 }

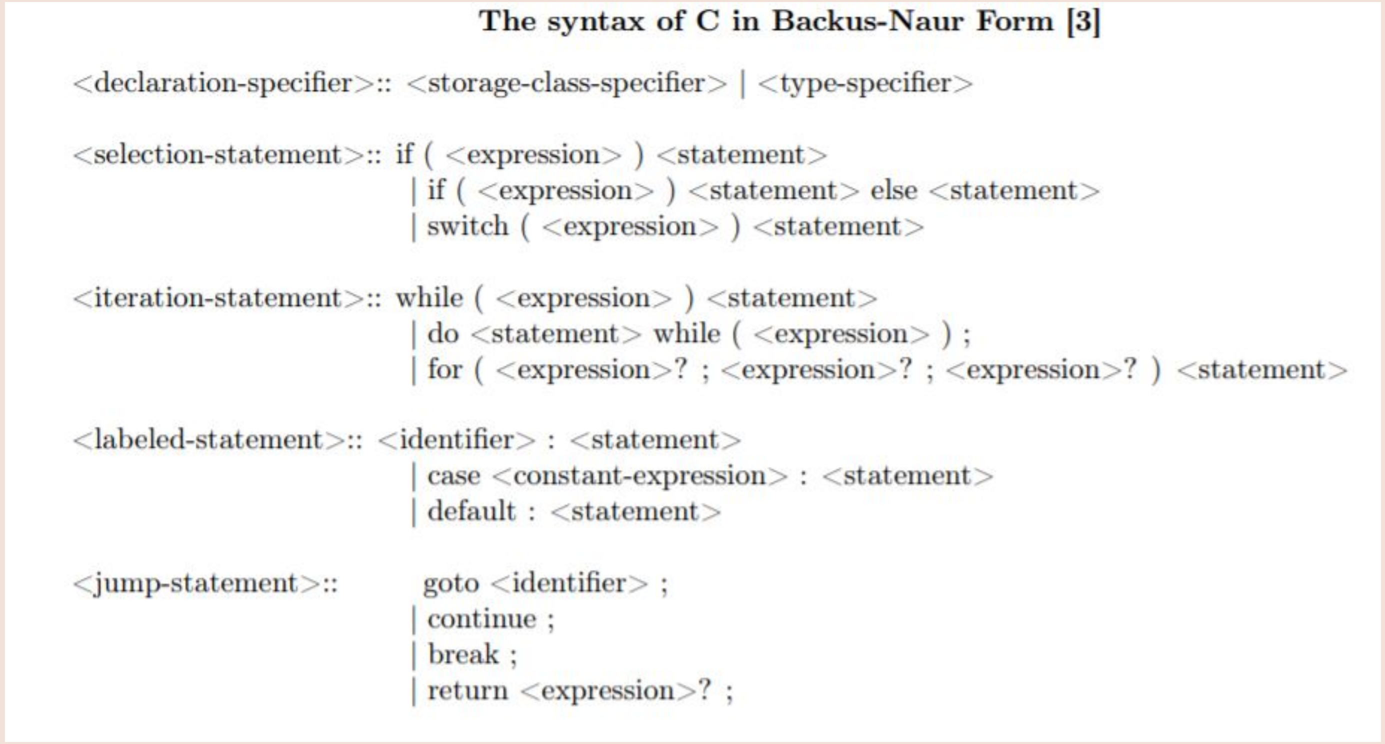
```
1. char ch='e'
2. ch++;
3. printf("%c",ch);
```

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

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 its 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 and the approach of semantics is studied.

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>? ;
```

3. SEMANTICS

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
5. If 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.

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.

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.

Properties

- Stuck term cannot be evaluated.
- 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.
- A normal form need not be a value always in C programming language. There are stuck terms.
- Determinacy of one-step evaluation: This property is satisfied by this language since there is at-most one rule for evaluating a term.

4. TYPE SYSTEM

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.

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 4
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> : stores various data types
14. <enum-specifier> : Assigns integral constants
15. <typedef-name>: Alternative names to existing types

If-Then-Else Construct

$$\frac{\gamma_1 \vdash \text{expression} : \text{exp}[T1] \quad \text{isScalar}(T1) \quad \gamma_1 \vdash \text{statement1} : \text{stmt}[T2] \quad \gamma_2 \vdash \text{statement2} : \text{stmt}[T2]}{\gamma_3 \vdash \text{if(expression) statement1 else statement2} : \text{stmt}[T2]}$$

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.

Do-While Construct

$$\frac{\gamma_1 \vdash \text{statement} : \text{stmt}[T2] \quad \gamma_2 \vdash \text{expression} : \text{exp}[T1] \quad \text{isScalar}(T1)}{\gamma_3 \vdash \text{do statement While (expression);} : \text{stmt}[T2]}$$

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.

Switch-Case-Default Construct

(1)
$$\frac{\gamma 1 \vdash \text{expression} : \text{exp}[T1] \ \text{isIntegral}(T1) \ \tau'' := \text{intPromote}(T1) \ \gamma 2 \vdash \text{statement} : \text{stmt}[T2]}{\gamma 3 \vdash \text{switch(expression) statement} : \text{stmt}[T2]}$$

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

(3)
$$\frac{\gamma 1 \vdash \text{statement} : \text{stmt}[T1]}{\gamma 2 \vdash \text{default: statement} : \text{stmt}[T1]}$$

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.

Properties

- 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)
- Even if the syntax in C goes statically valid, it might still throw a runtime error after a series of execution. Thus, Progress will not hold. (PROGRESS)
- 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. CONCLUSION & REFERENCES

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 Survey. 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. **C does not provide type safety.**

REFERENCES

[1]https://faculty.sist.shanghaitech.edu.cn/faculty/songfu/course/spring2018/CS131/sa.pdf *(Semantics with Applications by Hanne Riis Nielson, Flemming Nielson)*

[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 *(By Nikolas Papaspyrou, National Techincal University of Athens)*

[6]https://chortle.ccsu.edu/java5/Notes/chap15/ch15_6.html

[7] http://homepage.divms.uiowa.edu/~slonnegr/plf/Book/Chapter11.pdf *(By University of Lowa)*

thanks!