A Review on working of Cflow

Madhuri Bhalekar
Computer Dept
MIT, Pune

Arjun Jain
Computer Dept
MIT, Pune

Debajyoti Majumdar
Computer Dept
MIT, Pune

Nigrah Bamb
Computer Dept
MIT, Pune

Ajinkya Shendre
Computer Dept
MIT, Pune

Abstract— In the field of program analysis, call graphs provide a succinct and human readable visual form of function flows in a program. Typically, call graphs are directed graphs, that determine the sequence of invocation of subroutines depicting the caller callee dependencies. This is used to tap the flow a program takes during execution, laying a foundation for further needful analysis. In this context, Call graph generators, taking a program as input, are typically used to generate call graphs. GNU Cflow, is one such tool. It accepts a C program or a number of C programs as input and generates a procedure flow, with clear caller-callee sequence distinguished by level indentation, with callee functions indented inside caller functions. This output can be altered by supplying different available flags and output-formatting options to suit the requirement. There is a lot of scope to revamp the Cflow source code and utilize the dispensed output. In this paper, we discuss the nature of cflow, its expected output, its limitations and scope for future research in it.

Keywords— *Cflow, call graph, function flow, call graph generators, caller callee*

*****

I. Introduction

In order to effectively analyze programs, one must be aware of the function-invocation sequence in the program. Analyzing the function invocations does not limit itself to trivial order in which they are called, rather it involves an elaborate approach to jot the caller functions and the functions called by them in a graphical view using suitable and distinct notations for user's aid. Its vital applications include determining the flow of values between the subroutines and finding the unreachable functions to name a few. Call graph generators are equipped with the necessary gear to generate these graphs, in accordance with the programmers or users requirement. However, there is a need to identify the path a program takes through the control structures like if-else, switch. Runtime analysis reveals the utilization of only the selected path based on the satisfied conditions, whereas static analysis typically ignores the control structures to display the function invocation within them sequentially. Open source tools are available which provide static or dynamic call graph generation. Cflow is a GNU Cflow is a static call graph generator which produces a lucid indented output depicting caller-callee dependencies.

II. GNU Cflow- An Overview

GNU Cflow is a utility which is useful for analyzing a collection of functionally interdependent C files and producing an indented output, which exhibits functional dependency between procedures in the listed files. The generated output comprises of clear notations used to show the line number and the source file a function was encountered in. Moreover, function recursion also has a definite notation, which charts the line number and file where its original definition exists. Typically, cflow scans the listed source files, extracts the function definitions and invocations throughout, and generates a neatly indented output, with calling function as a parent and called functions slightly indented to the right of the caller. This scheme of indentation is recursively forwarded from the main() function till the end of the last function invoked by main. observe the typical output below . A typical cflow invocation syntax looks like follows [1]:

cflow [<file> | -<options>]

where option can be expanded as:

<options>: I | m <func> | o <file> | u

Options may also appear between the filenames, with no attention to order of appearance. –i option lets cflow open and read #include files referred to in the source code. A function name following the –m option instructs cFlow to start the dependency tree from that function instead of main. The –o option redirects the output to the given destination file. –u option enables the inclusion of undefined functions in the dependency tree. For a given C program (Fig. 1) , the Cflow output looks like follows.

```
#include <stdio.h>
void add( int num1, int num2){
int result = num1 + num2;
display(result);
}
void display(int result){
printf("\n sum is: %d", result);
display(result);
}
int main(){
int num1, num2;
printf("\n enter two integers: \n");
scanf("%d", &num1);
scanf("%d", &num2);
add(num1, num2);
return 0;
}
```

Fig. 1 The example C code

The typical cFlow output consists of a distinct notation, (recursive: see 3), for displaying recursive functions, which indicates the line number in the listed files where the original function is defined (Fig. 2). (R) indicates the invocation of the parent function.

```
    main() <int main () at add.c:15>:
    printf()
    scanf()
    add() <void add (int num1, int num2) at
    add.c:3>:
    display() <void display (int result) at
    add.c:10> (R):
    printf()
display() <void display (int result) at
add.c:10> (recursive: see 5)
```

Fig. 2 The typical Cflow output

The above output can be further understood pictorially with the help of following figure:

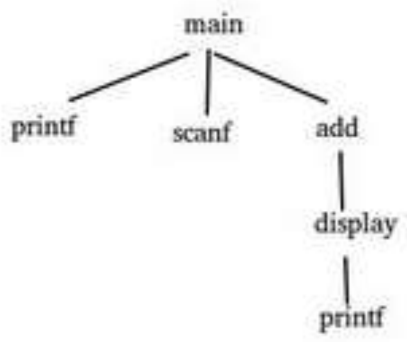

Fig. 3 Pictorial representation of Cflow output

III. The Cflow Source Files

The Cflow source files provide the core functioning skeleton of the tool. The files can be segregated ino Header files (the declaration and initialization part) and Core files, which contain the definitions of all the function needed for Cflow operation.

A. Header Files

Cflow operation is realized by the numerous definitions inside the header files. Each header file, has useful declarations and pre-specifications carrying a significant global usage throughout the functioning of this utility.

1) Cflow.h: This header file is responsible for the declaration of environment variables and structures used by functions across different files in the source folder. A prominent structure defined inside cflow.h is Symbol. This structure is an assortment of pointers and varibles used to assign name of the encountered symbol/token, line number, token type, storage type, recursive check, and most importantly it consists of pointers to two major structures linked_list_entry and linked_list. There exist two pointers “caller” and “callee” inside this structure which point to the linked_list structure and help in formulating the dependency matrix. Another prominent structure definition inside cflow.h is linked_list_entry. Each encountered function is added to a list specified in the structure with the next and previous pointers to traverse the same. The function details like name,line number, arguments are handled by the “data” pointer.

2) Parser.h: each token has been assigned a particular ID based on their syntactical classes for instance IDENTIFIERS, MODIFIERS, QUALIFIERS, WORD, LBRACE, RBRACE to name an important few.

On carefully examining the working of this utility using the mentioned techniques, it is observed that cflow like all C programs kickstarts from the main.c file in the source folder. There onwards, the interworking of lexer-parser is initiated, with the tokens being extracted in the c.l file and the syntax checking being performed in the parser.c file. Parser.c implements a sequence of functions to filter only the desired tokens from the listed file, according to the specified goal of cFlow. Lastly, each function-token from the listed files is appended to a linked list. The linked list operations are looked upon in the linked-list.c file. symbol.c file compares symbols obtained from the lexer, and calculates its hash value. gnu.c generates a level-indented tree based on its type (direct or inverted) which is displayed using output.c. A brief description of each file is a follows:

B. Core Files

1) *main.c*: The main.c is responsible for initiating the lexer-parser co-routine by calling yyparse function. Apart from this, it also checks for the existence of the listed file. main.c also takes care of initialization of the environment, by invoking the init() function. Init takes care of pre-appending spaces and initializing the cflow canvas. Two major functions invoked by init are init_lex() and init_token(). init_lex further calls init_tokens() (different from init token()) function in c.l that handles the keyword, types, and qualifiers initialization.

2) *c.l:* c.l comprises of essential declarations, structures and initializations required by the scanner and parser. The life-granting yylex() function of a lexer is invoked by the c.l through the get_token() function. As mentioned above, an array of keywords, types and qualifiers

is declared and initialized. Moreover, the source() function looks after the listed file by making it available for scanning ,that is, it opens the file in read mode.

3) *parser.c:* Parser.c is encapsulates the core functionality of filtering the tokens in the desired manner, that is, deal with functions, variables, arguments in a manner, suitable for the required output of cFlow. Major syntax checking is done here so as to segregate variables, functions, handle arguments, and match patterns to decide if the encountered token is a function or not. A number of vital functions influence the operation of parser.c . Broadly speaking, cflow follows a nomenclature for a function definition (the function definitions that exist before main()) , by calling the encountered function a “caller” and assigning it to the caller variable. For an indented output, it also keeps track of the nesting level for a particular function. This is handled by the 'level' variable. As specified, yyparse() is responsible for initiating parsing, and initializes the caller and level variables. To check if the encountered token is a function or a variable, it invokes parse_declaration ().

parse_declaration () verifies if the encountered token is a function and thereby calls is_function () to perform the check.

is_function() comes into action as soon as a return-type is encountered in the input file/s. Based on the return value of is_function() it evokes parse_function_declaration() if true, else parse_variable_declaration().

parse_function_declaration() procedure looks after the assignment of the encountered function definition to the caller variable and then hands over the subsequent functionality to func_body().

func_body() is a significant function that influences the nesting level of the encountered function in the listed file. It checks the function declaration syntax of the encountered procedure using expression() . Consequently, It works as a scanner that implements a while loop to recursively scan the body of the function and the same treatment is meted out to the functions and variables as described above, by invoking the above functions with current parameters.

The expression() function does the notable task of checking the function declaration by examining the current token. If it is an IDENTIFIER and followed by a ‘ (‘ , it should be appended in a linked list, with the help of call() function. The existence of parameters, for the current function, is examined in call() using 'arity'. Consequentially, the function is appended to the linked list using linked_list_append () in linked_list.c .

To create (using linked_list_create()) and allocate memory to the linked list, add_reference() is invoked. Another function called nexttoken() is the core function which obtains the tokens from the lexer using the get_token() function, as described above. As soon as the token is obtained, it is pushed on a local stack, along with its line number and type using the tokpush () operation which is instrumental in assigning the memory location to the current token, by pushing the type, line along with the token on the stack.

4) l*inked-list.c:* cFlow uses linked list to store and retrieve the functions for all purposes. Thus, linked-list.c file comprises of functions directing the linked list operations. Two major functions in the file are linked_list_create() and linked_list_append(). linked_list_create(): This function is invoked by add_reference() in parser.c to allocate memory dynamically, according to size of the list. linked_list_append(): The task of appending a function to the linked list is handled by this function. Memory equal to the linked_list_entry structure(mentioned previously) is allocated , which encapsulates the the current symbol name, its previous and next symbol in the list. If the current token is a first , it is made the head, else appended to the tail of the list.

5) *output.c:* output.c is responsible for shaping the output format, achieved with the help of direct_tree () function. This is realized by traversing the formed linked lists and calling the print_symbol() function defined in gnu.c file. Likewise, another function inverted_tree () is used to print an inverted tree, by performing the appropriate traversal.

6) *gnu.c:* The output display format and notations are printed on the standard output by gnu.c. A function print_symbol() is invoked by direct_tree() in output.c, this function is responsible for printing the encountered function names with the desired indentation. This is majorly achieved by invoking two functions, print_level() and print_function_name(). print_level () decides upon the indentation to be supplied on the basis of current level with respect to the calling function, while print_function_name() takes care of printing the function names with the suitable notations. print_function_name() prints the final cflow output in the desired format. The angular brackets in the output notify the presence of the function declaration, the (R) notation representation of a recursive function, the source file name etc. are all printed using this function.

C. Tools to analyze Cflow

A number of tools can be used to analyze and determine the activity taking place between Cflow functions. There needs to be a mechanism to visually depict the functional dpendency between various procedures in cflow, as well as, to manually enter the files to make alterations and view the corresponding output. Following are the two tools which can be used for achieving the same.

1) pycflow2dot: Aforementioned description and complex inter-functional dependencies can be difficult to realize without the help of visual depiction of the same. To aid in the same, pycflow2dot is a utility used to draw the call graph of the c source code using cflow and dot . Output to LateX, .dot, .PDF, .SVG, .PNG and from dot to all formats supported from it [2]. Syntax to use pycflow2dot is:

cflow2dot –I file_name.c –f png Following figure exhibits a typical output from this utility for the main () function inside main.c file.

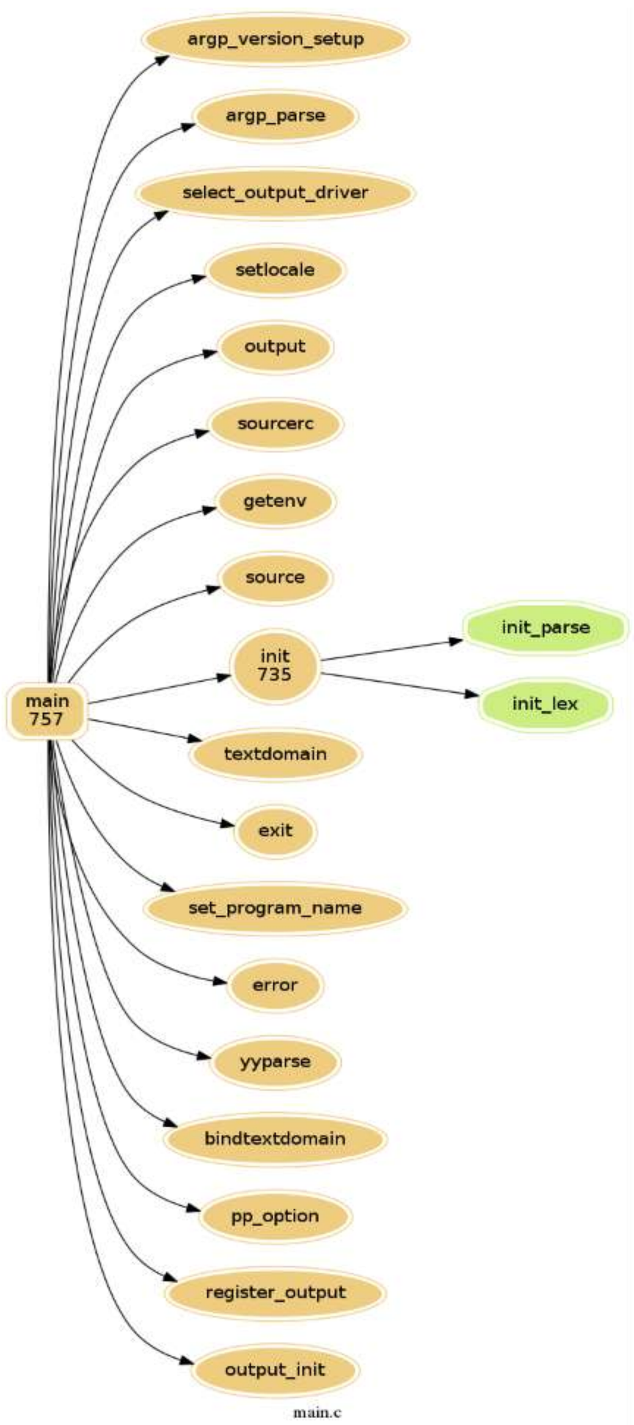

Fig. 3 Sample pycflow2dot output for main() function in main.c file

2) cscope: Ideally, visual depiction is insufficient to determine the way a software like cFlow works. There is a need for a mechanism to identify the structure of the software, search and lookup for symbols and functions. To navigate C-like code files, the C-scope interface is used. Using Cscope, you can search for identifier or function definitions, their uses, or even any regular expression [3].

Thus Cflow is a useful tool that helps a programmer to determine the sequence of function invocation as well as the the dependency between functions in a C program. It is equipped with the essential flags that provide flexibility in the nature of the output. Even though Cflow presents a succint output format , there is scope for a lot of tweaks in the source code contained in the assortment of files, to achieve the desired output.

IV. CONCLUSIONS

This paper, provides a comprehensive overview of the cflow skeleton. Cflow internally scans the listed files and implements the common lexer parser scheme leading to token handling and token-related operations. Cflow is logically segregated into a number of vital header files and core files that help in building its functionality. The header files handle the major declarations and initializations that are required throughout the core files. The core files consist major functions which contain and direct the cflow operations. Going through a number of stages, a final indented output is displayed on screen, with proper notations.

REFERENCES

[1] Asaf Arkin,"Cflow Operating Instructions", Internet: http://ftp.stratus.com/vos/tools/cflow.txt, Oct. 20 , 1989 [20 January 2016].

[2] Ioannis Filippidis, "Layout cflow output using GraphViz dot ", Internet: https://github.com/johnyf/pycflow2dot, Feb. 28, 2015 [25 January 2016].

[3] Tonymec, "Cscope", Internet: http://vim.wikia.com/wiki/Cscope, Nov. 25, 2009 [25 January 2016].