CS 502 – Fall 2012

Project 2: Interprocedural Analysis on Uninitialized Variables

Deadline: Friday, November 30, 2012, 11:59pm

1. Project Overview

Your goal in this project is to implement an interprocedural analysis on uninitialized variables. Such an analysis can help discover programming errors. The project is to be implemented in gcc-4.7.0, based on the high level IR GENRIC (or AST). This gives the students an opportunity to practice what they learned about dataflow analysis in a well-known compiler framework.

When a program takes a variable as an operand in an operation, the variable is said to be used at the program point where the operation appears. A correct program must never use a variable as an operand before the variable is assigned a value, i.e. before it has a *definition*.

If a variable is used before any definition, it is said to have an uninitialized use. Based on dataflow analysis, the compiler can recognize a set of variables which are guaranteed to be defined before any use. If any variable cannot be guaranteed by the compiler, then we conservatively assume it may be uninitialized. In this project, we implement dataflow analysis to identify program points where uninitialized uses may potentially take place. The analysis is conservative because it does not analyze under what circumstances an IF condition must be true.

2. Project Requirements

2.1 The scope

We assume the same subset of C language as in Project 1. We only require to recognize **uninitialized uses of scalar variables**.

The project requires a very limited form of alias analysis to deal with Condition_1 below:

Condition_1: the address of a scalar variable may be passed to a function. For example, in Fig. 1, the local variable **a** in main() function is initialized by the function init().

```
void init(int *b) {
*b = 1;
}
void main(){
```

```
    int a;
    init(&a);
    int *point_to_a ;
    point_to_a = (int *) malloc (sizeof(int));
    (*point_to_a) = 1;
    printf("%d\n", a);
}
```

Fig.1 Example 1

Other than Condition_1, we assume that no pointer dereferences (such as "*p" and "p->q") will result in accessing the same memory location that is accessed by a scalar-variable reference. Therefore, if a pointer is neither a formal parameter of a function parameter nor the address of a scalar variable, your analyzer can assume that memory writes and reads due to such a pointer's dereference will not have any impact on the analysis. For example, in Fig.1, you can ignore the memory write by the operation" (*point_to_a) = 1;" because point_to_a is a local variable rather than a function parameter.

Moreover, we also assume that：

(i) Different pointer variables never point to the same memory addresses.

(ii) A pointer never points to any local variable declared in the same function, nor does it point to any global variable.

You are required to recognize both uninitialized local variables and uninitialized global variables, as discussed below.

(1) Uninitialized local variable.

In the following example, variable "a" is uninitialized when accessed in the printf statement, because "a" is defined in one execution path leading to the printf but not in the other. Notice that importing a function parameter is also a way to define a variable's value, such as variable "i" in the example.

```
void foo(int i){
        int a;
        if (i) a=1;
        printf ("%d", a);
}
```

Fig.2 Example 2

(2) Uninitialized global variable.

In the following example, global variable "a" is assigned in function init(). As a result, there is no more uninitialized use of a within function foo.

```
int a;
void init() {a =1;}
int foo(int i){
        init();
        if (i) a = 1;
        printf("%d", a);
}
```

Fig.3 Example 3

2.2 Output of your program

When any use of a variable is recognized as an uninitialized variable in a given program, your analyzer must report all uninitialized variables in an output file "output.txt". The output should be formatted strictly as described below. Otherwise you will lose points for not following the format.

Output format:

The 1st function's name: the name of the 1st uninitialized local variable, the name of the 2nd uninitialized local variable, …

The 2nd function's name: the name of the 1st uninitialized local variable, the name of the 2nd uninitialized local variable, …

And so on.

If a function contains no uninitialized local variables, you do no not need to print that function's name in the list.

In the above, we follow the order of functions and variables when you traverse the GENERIC in the normal **depth-first** search.

If there exist any uninitialized global variables, then add one line at the end of your output file as follows.

Global : the name of the 1st uninitialized global variable, the name of the 2nd uninitialized global variable, …

For example, for Fig.2, the output of your analyzer should contain exactly one line:

foo: a

3 How to start

To start the project, students need to follow the steps similar to those taken in Project 1:

1. To set up the environment of Project2, copy and unzip the file “cs502_fall12_p2.tar.gz" from the directory “/u/data/u3/cs502/Fall12/CS502” to **your scratch directory**. A directory “cs502_fall12_p2" will be created, in which a soft link “gcc-4.7.0_src” and a directory “install” are included. The soft link “gcc-4.7.0_src” leads to gcc-4.7.0 source code directory on xinu, which can be read only. The “install” directory is students’ working directory.
2. Go to the directory “cs502_fall12_p2/install/". In this directory, students should write a source file named “csproj2.c" to implement the analyzer. Now a template with an empty function cs502_proj2 and a sample Makefile is provided. After the analyzer is written, run the command “make" to create your own version of program “cc1" which has the unparsing feature built in.
3. After a successful make, please run “./cc1 some-file-name.c" to analyze some-file-name.c. The program cc1 reads the source code, analyzes the variables and output uninitialized variables if there is any.
4. Three sample test files are provided under “/cs502_fall12_p2/install” directory. They will serve as the base of test cases.

4 Hints and Suggestions

Before you start the project, you need to design the method for analyzing the GENERIC in gcc 4.7.0. As in Project 1, you are not allowed to modify any existing files in gcc-4.7 source code. Instead, we provide a “csproj2.c” file which contains a function “void cs502_proj2()” from which you should start the analysis.

Where is Project 2 called?

Similar to Project1, cs502_proj2 is called at the end of GCC frontend, after the GENERIC and Call Graph is built.

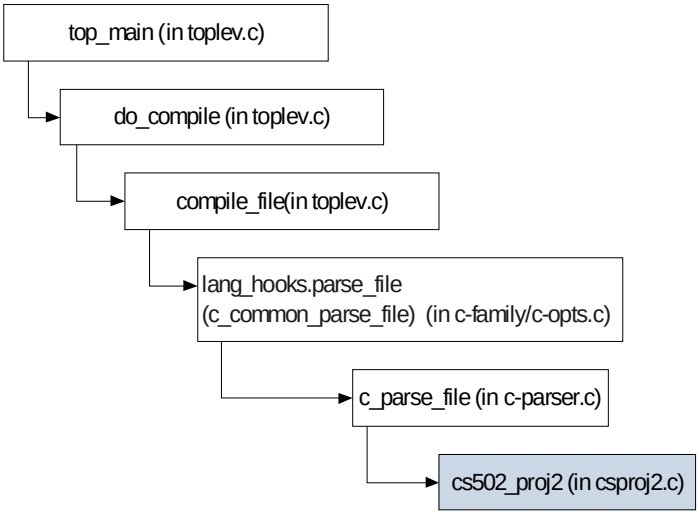

Fig 4. Where project 2 is invoked

One Suggested Algorithm

This analysis is quite similar to live variable analysis which is performed in a backward order. For each variable, you keep track of all defs and uses so that you can determine whether it is possible for that variable to be used before ever being defined.

For the interprocedural analysis, a call graph must be constructed, with each node representing a function and each edge representing a call site. No recursive functions are considered in this project. You can still use "struct cgraph_node cgraph_nodes" to traverse each function as what we did in Project 1. **Note***:* Although cgraph_nodes is used to represent call graph in GCC, however, the caller/callee edges haven't been added up at the end of the front-end. Therefore, you have to build your own data structure to represent a call graph.

For global variable analysis, you may want to create a data structure to keep the information of each function about which global variables are defined and referenced by the function.

5. How to submit

Use the following commands to submit your files on xinu machines,

turnin –c cs502 –p proj2 [your working directory]

The submission should include:

1) All new source files.

2) The new Makefile.

3) A README file explaining anything you want the TA to know in order to re-produce your directory and generate your analyzer.

4) A project report in plain text (please see 6.Grading Criteria – Documentation)

NOTE1: Please be sure to run "**make clean**" before submitting your working directory.

NOTE2: Please do **NOT** include any other files which are not listed above, especially the softlink to gcc source code.

6. Grading Criteria

The implementation for local variables and global variables will be graded separately. For each part, the grading will take in account the correctness and the clarity of documentation. The grade breakdown is as follows:

a. Analysis for local variables (70%)
   - Implementation (%60)
     - Correctness
     - Output format
   - Documentation (%10)
     - Describe the main data structures and code outline
     - List all the files you have created and explain those new major functions and data structures you implemented.
     - Describe the status of your implementation, such as cases you can handle and you cannot.

b. Analysis for global variables (30%)
   - Implementation (20%) (Same requirements as local variables)
   - Documentation (10%)(Same requirements as local variables)