STATIC DETECTION OF SEQUENCE-POINT RELATED UNDEFINED BEHAVIOR IN C

Lukáš Hellebrandt
Bachelor Degree Programme (3), FIT BUT
E-mail: xhelle04@stud.fit.vutbr.cz

Supervised by: Petr Müller
E-mail: imuller@fit.vutbr.cz

Abstract: This paper describes a tool that automates the process of static detection of certain kinds of undefined behavior in the C language programs. We focus on undefined behavior related to sequence points and side effects. The tool's high-level architecture and implementation are described. The paper briefly shows what the C standard defines, what dangerous situations it can lead to, and demonstrates it on examples.

Keywords: Undefined behavior, C language, side effect, sequence point

1 INTRODUCTION

Many programmers, even experienced ones, are confused by the sequence points and side effects in the C language. It is, however, a common source of undefined behavior in C programs, which leads not only to unspecified result of the operation, but to the situation where the whole program makes no real sense. This is because there is no particular requirement for the compiler implementation in case of undefined behavior.

These malicious constructs are, however, sometimes hard to find even for a programmer who is aware of their possible impacts. Therefore, my goal is to create a tool for their automated detection using static analysis.

2 UNDEFINED BEHAVIOR

The C standard [1] differentiates multiple categories of behavior ("external appearance or action"). Above all, *unspecified* behavior, where the standard provides multiple possible behaviors. *Implementation-defined* behavior, similar to unspecified, but the implementation is required to document the choice. And most important, *undefined* behavior, described as "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements".

2.1 SEQUENCE POINT

The evaluation order of expressions may be specified, but in many cases it is not. If it is not, the implementation is *in certain cases* required to specify it and do it consistently and predictably.

The standard defines multiple sequence points, e.g. semicolon, before a function returns, or between the evaluations of the first operand of the ternary operator and the second or third one.

An expression A must be evaluated before an expression B when it is on the left side of a sequence point that is between A and B. Otherwise A and B may either be evaluated in an unspecified order or they can even *not be evaluated in any order*.

2.2 SIDE EFFECT

A side effect is usually a modification of memory during evaluation of an expression.

```
i = j++;
```
(1)

The code 1 is an example of side effects – the `j++` expression (post-increment) returns the original `j` value *and* increments the object `j`'s value by 1 (side effect). The assignment operator stores the returned value in `i` (another side effect).

3 CONSEQUENCES

The standard states that *undefined behavior* (i.e. behavior which may lead to virtually anything) occurs when "a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object". This usually means a variable can not be modified (or read and modified) more than once within one sequence point.

These constructs may be easy to find, for example the one in code 2, where there is no sequence point between the assignment and the post-increment operator.

```
i = i++;
```
(2)

They may, however, be harder to find because of use of pointers (code 3) or functions.

```
int i, *j; j = &i; i = (*j)++;
```
(3)

An automated tool for finding these constructs might significantly lower the risk of missing them and save a lot of programmers' time. It will also help preventing bugs in code that worked "correctly" for a long time and suddenly it stops working because of some change in the compiler implementation.

4 SEQUENCE POINT ANALYZER

Due to the problem with using pointers mentioned above, the architecture of the tool consists of two main parts. To be able to find undefined behavior in the C source file, we need access to the Abstract Syntax Tree (AST). Having this, the main part of the tool, a Clang [3] plugin, will be able to tag each node in the AST as either implying or not implying a side effect on a certain object. The output of this part will be a set of constraints telling under which circumstances the behavior is undefined, e.g. "there is undefined behavior at the row 42 if `i` aliases with `*j`". This set of constraints will be verified by the second part, the alias analysis [2]. Currently we know of no implementation of alias analysis capable of using the Clang's AST directly, but this output can be used by arguably any external alias analysis, e.g. the one in LLVM [4]. In this paper, the Clang part of this tool is further discussed.

There are two things we can decide for each node depending on its type:

- Whether it preserves the object's L-value (thus whether any of its predecessors can modify that object, implying a side effect on it)
- Whether it actually implies a side effect on an object

There is a special type of node, called DeclRefExpr, which is always a leaf of the AST and is a reference to the object declaration. Iterating over all these nodes and going to the root, we can tag the whole AST using the algorithm on figure 1.

Part of the tagged AST of the code 2 is shown in the figure 2. We began tagging from the left DeclRefExpr. It surely can be used as an L-value but the expression itself implies no side effect, hence the tag in italic ("not implying anything"). We went towards the root and as we passed the BinaryExpression (=), we mark it as an expression implying a side effect on `i` (tags in bold) as it modifies the memory contents of `i`. We continued to the top implicitly marking all the parent nodes as having a side effect on `i`. Then we can do the same for the right DeclRefExpr and as we see two bold tags of the same variable, we can say we found an undefined behavior as there is no sequence point between the post-increment operator and the assignment.

```
for declRefExpr in declRefExprs do
    node = declRefExpr
    while node.type != functionDecl do
        if nodeImpliesSideEffect(node) then
            tagNodeAndAllPredecessors(node, declRefExpr)
            break
        else if nodePreservesLvalue(node) then
            node = node.parent
        else
            break
```

Figure 1: Tagging algorithm

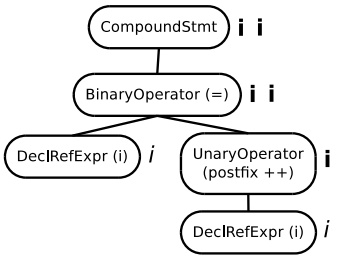

Figure 2: Tagged graphical representation of the AST of the code 2 (`i = i++;`)

Having this tagged AST, we can determine for each node whether it causes undefined behavior depending on its type and on whether its children have any side effects. As we are currently not aware of any implementation of cross-function alias analysis, we will assume that if a function *can* have a side effect on a certain object (it has access to its memory location), it *does*.

Declaring two variables as `restrict` in the same scope means they do not alias with each other. There will be no constraints generated where two different variables are declared as `restrict`. However, `restrict` variables can not be omitted completely because there can still be side effects implied on them and those side effects do not need to be sequenced.

5 CONCLUSION

A tool for static detection of certain kind of undefined behavior in C programs has been implemented. Its output is a set of constraints telling under what circumstances the behavior is undefined. The constraints can be checked by external alias analysis. Alias analysis in LLVM has been used as an example.

The tool's main part is implemented as a Clang plugin which makes it maintainable and possibly useful in upstream. Currently, the Sequence Point Analyzer is a separate project with its own repository.

REFERENCES

[1] WG14. N1570 – Information technology – Programming languages – C – Committee draft. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1570.pdf`, 2011. [Online; accessed 2014-02-27].

[2] Thomas Lenherr. Taxonomy and Applications of Alias Analysis. `http://e-collection.library.ethz.ch/eserv/eth:30904/eth-30904-01.pdf`. [Online; Accessed: 2014-02-28].

[3] clang: a C language family frontend for LLVM. `http://clang.llvm.org`. [Online; Accessed: 2014-02-28].

[4] The LLVM Compiler Infrastructure. `http://llvm.org/` [Online; Accessed: 2014-02-28]