SWE 212 Analysis of Programming Languages

Week 2 Summary of “Global Variables Considered Harmful” by W.Wulf, Mary Shaw

Samyak Jhaveri

Introduced as if it is a sequel to Dijkstra’s paper to abolish goto statements from high-level programming languages, this paper claims and gives supporting evidence to abolish global variables.

The authors intentionally refer to not only global variables, but to “non-local variables” in a much broader sense so that their argument is applicable to any variables which are referenced in a segment of the program, S, such that not all uses of that variable are contained in S. They begin their arguments against the use of non-local variables by briefly delineating the arguments against the goto statement like how they make the mapping between textual relations and execution relations ambiguous and difficult to understand. However, well-structured forms of control transfer like if-the-else, for, case, etc. are not condemned as they make it easy to locate their precedents and antecedents in the code, unlike in the case of goto which contributes to the spaghetti-fication of the code. By extension, the authors mention that not all programs are harmed by the use of goto and that some might actually benefit from its use - typically, small programs in which the targets are close to the goto’s jump points. Similarly, their argument concerning non-local variables asserts that non-local variables may be misused and not every use obscures the program structure.

The paper establishes the need of abstraction in “understanding” a program. Given the fact that the knowledge of the correspondence between the text of a program and the executions it produces embodies its “understanding”, and the fact that the class of all the computations a given program may execute is large, understanding a program may be enigmatic. Therefore, abstraction helps in straining out irrelevant information and considering only the effects which can be considered to be contributing towards obtaining the results of a given program. Formalizing the notion of abstraction, the authors suggest a segment of code S has logical propositions P and P’ such that P is true before the execution of S and P’ is true after the execution of S. An abstraction of this would be S:P →P’. This means that P’ should be able to describe the effect of all the non-local variables modified during the execution of S. This increases the complexity of the abstraction in proportion to the number of non-local variables since all of them would have to be traced backwards into S. How does this relate to non-local or global variables? To check the correctness of the proposition P, it is necessary to check whether the variables in P actually have the values as claimed for them. This is simple to do in a straight line program as it can be scanned backward until the proposition involving all the variables is found. Unfortunately, straight line programs are rare, which means that it would be necessary to scan backwards on all the possible control paths until an assertion about each variable is found on every path. The complexity of such an activity increases as the number of non-local variables increases in a given program text. Therefore, just the way well-structured control operators help identify the assumption(s) in effect at any point in the program, appropriate scope rules will aid in locating relevant values of the variables. To address such complexity, one may consider reducing the size of the segment S of the program text by using blocks. However, this may not be a silver bullet since the combinatorial complexity of keeping track of the remaining non-local variables along various control paths leading to S may again increase.

The paper further discusses the perils of a block structure that is typically used (at that time) to implement non-local variables and also the disadvantages of non-locality.

1. **Side Effects:** Procedures and function modifying variables other than their own local variables can cause confusion in both writing and understanding the code
2. **Indiscriminate Access:** Since a block structure is only a programming construct and not a real object or element of a programming language, is it possible to access and modify variables that have been declared elsewhere in the code, possible even outside the said block at hand. This means that the blocks are not watertight to the non-local variables. Changes made to these variables can obscure the understanding of its state.
3. **Vulnerability:** In the case of nested program blocks, the programmer can interpose new definitions of the non-local variables in the inner nested block. In such a situation, the programmer needs to be aware and alert about the bindings of all the names used at the intervening levels when using a variable at a deeply nested inner level of the program block.
4. **Non-overlapping definitions:** Sometimes, when a program segment depends on the output values of its immediate predecessor, the variables storing the values may be large arrays of various shapes. In such a situation, only the program segments concerned with these values should have access to these variables and be inaccessible to all the other program segments. Unfortunately, the block structure discipline does not have such a functionality , and global approaches make the variable vulnerable.

Finally, the paper proposes solutions to the problems with non-local variables they stated above. To summarize and reiterate their main arguments, the authors draw a parallel between the problems of goto and those of non-local variables. In the case of goto statements, the notion of “transfer of control” is not a problem as much as the syntax of the block structure itself is - the fact that these transfers must enforce locality of the transfers. Similarly with scopes, the notion of controlled access isn’t the problem but rather the lack of explicit and localized representation of such control is. The authors’ contention is not even to lobby for better syntax in programming languages to enable them to promote good program structure. Rather, their concern is with the semantics of languages since, unlike goto, the scope problem does not have :
i) a single offending construct whose elimination will correct the problem and
ii) a collection of familiar constructs which are just as adequate to capture the essential use of the underlying primitive concept.
Nevertheless, some alternative they suggest are:

- Extending the scope of a name into inner blocks should not be the default
- The right to access a name should be a mutual agreement between the creator and accessor
- Access rights to a structure and its sub-structures should be decoupled
- It should be possible to distinguish between different types of access
- Declaration of definition, name access and allocation should be decouples