Goto Hell: the Need for Better Control Structures and Compilers

Craig Kelly

December 5, 2016

1 Introduction

The unconditional branch instruction (called the "goto" statement) has been a source of controversy for more than 30 years. The necessity of the statement for efficiency and its harm to programs that must be maintained by more than one person have been discussed at length. Although the most famous pro and con opinions were written by Dijkstra and Knuth respectively [7, 11], many more articles, papers, and discussion have considered the issue at length. Many have considered the argument settled for quite some time, but there is still some dissent. In 1987, more than 20 years after the arguments began, F. Rubin claimed that the belief that goto should be eliminated lacked an adequate rational foundation [18]. That letter sparked yet another round of lively debates [9] with no universally recognized winner.

Although it is interesting to review the particulars of such a debate, the issue of the goto statement still holds relevance for the computing profession today. *Ongoing issues with goto should be viewed as a need to improve control structures and compiler optimization techniques.*

2 Justification

Before continuing the story of goto, it is instructive to consider the relative importance of the goto statement and in what scope it will be examined.

There is little doubt that the issue of goto is important to the computing community. Some of the most influential scholars in the computing field have published papers and articles concerned with nothing but the evaluation of a single small instruction common to most procedural languages. When Dijkstra published his now famous "Go to statement considered harmful" [7] it was meant to be a short article in a journal. However, Knuth reports that it was published as a letter to the editor in an effort to speed its publication [11] (ostensibly to skip the peer-review process).

As Leavenworth put it, the concept of the goto statement "does not appear in most formal systems of computability theory, but does appear in programming language extensions of those systems" [13]. One might argue then that goto appears in programming languages because it is a conceit that has existed in some form for most of the history of the theoretical computer; the Turing machine itself has the concept of a goto statement. However, Rojas has shown that there are only four basic instructions needed for a von Neumann computer: load, store, increment, and goto [17]. Since most modern computer systems are based on the von Neumann architecture, the goto statement is here to stay in some form (even if only as a microcode at the heart of a CPU's hardware).

3 Scope of Inquiry

Insomuch as programming languages can be viewed on a continuum from low to high level, the scope of any consideration of goto much be constrained to a subset of that spectrum. At the very lowest level is hardware machine code and assembly language. As mentioned above, goto is intrinsic to von Neumann computers and so much be viewed as necessary to the hardware instructions. As a result, it is not reasonable to consider the fate of the goto until the language under consideration is at some minimum higher level.

At the other end of the spectrum are languages that are far enough removed from procedural operation that even the consideration of a branching instruction makes little sense. For instance, most declarative languages (such as SQL) and functional languages (such as Erlang) don't contain a concept related to a procedural branch and so are outside any scope of consideration. [1]

[1] Of course, there are declarative and functional languages implemented via a procedural language. While these procedural programs may be considered part of the scope under consideration, their output and runtime concerns are not.

4 Overview of Debate

As a matter of history, Knuth claims [11] that Peter Naur was the first to publish about "harmful go to's":

> If you look carefully you will find that surprisingly often a go to statement which looks back really is a concealed for statement. And you will be pleased to find how the clarity of the algorithm improves when you insert the for clause where it belongs... [16]

The most influential publications in the historical goto debate are Dijkstra's "Go to statement considered harmful" [7] and Knuth's "Structured Programming with go to Statements" [11]. [2] These two positions and the commentary surrounding them sum up much of the debate surrounding the goto statement.

Dijkstra's claimed that "[t]he goto statement as it stands is just too primitive; it is too much an invitation to make a mess of ones program" [7]. However, much of the article is dedicated to the conceptual handicap imposed by a program containing goto's:

> ...our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible. [7]

Dijkstra goes on to develop a simple way of keeping enough data to restart a program at any given point. Leavenworth pointed out that Dijkstra's argument can be considered in a stronger light; unrestricted goto's increase the difficulty of the halting problem. If goto's are removed, "there are only two ways a program may fail: either by infinite recursion, or by the repetition clause." [13]

Knuth's argument is often seen as a counterpoint to Dijkstra, and a pro goto stance. Although he does argue for the use of goto in some situations, he also argues "for the elimination of go to's in certain cases" [11]. Additionally, Knuth quotes personal correspondence from Dijkstra [11] that reveals Dijkstra's fairly measured stance on goto usage:

> Please don't fall into the trap of believing that I am terribly dogmatical about [the go to statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline! [6]

Knuth's main arguments for using goto's are a series of patterns and examples given a number of situations in which goto usage is clearer, more efficient, or more desirable. Leavenworth amalgamates these reasons and situations into three justifications for the use of goto: synthesis of missing control structures, efficiency, and abnormal exits [13].

The main example given for the case of missing control structures is the simulation of a *case* statement in a language (like PL/I) that has no such structure but does have goto. This particular argument may be given less weight because goto removal and adding a case statement are both languages changes. Adding a language feature provides backward compatibility, while removing a feature from a programming language breaks existing programs. Thus, any argument for goto involving the synthesis of a missing control structure in a specific language must be viewed as a weaker version of an argument to add that specific control structure.

Abnormal exits and efficiency are two concerns that are often related. The use of a goto in the case of an abnormal exit from a loop or subprogram is often replaced with a structure involving the evaluation of a boolean variable. On many platforms, the goto is a single machine instruction, while the boolean test is often reduced to multiple instructions.

5 Goto Removal

As mentioned in section 2 (*Justification*), the goto construct is necessary at a fundamental level in the von Neumann architecture; however, at a sufficiently abstract level it becomes unnecessary (such as in functional languages). Thus it is only natural to ask if we can do away with the goto entirely in procedural programming languages. Dijkstra himself pointed this out [7]. Only two years earlier, Jacopini had shown that goto's could be removed via a technique involving the transformation of flow diagrams [5].

[2] In an odd twist of fate, Knuth's paper is often better known for it's use of the phrase "premature optimization is the root of all evil". Although often quoted, most don't realize it comes from a paper defending goto on the grounds of optimization!

However, Dijkstra also pointed out that the resulting program was actually less clear than the original and so was actually worse than a program with a goto.

Knuth covered the same issue with Jacopini's work. However, work continued on automated goto removal. As Knuth also points out, Ashcroft and others created a technique that in many cases provided a more readable goto-less transformation [3]. Unfortunately the technique may cause an exponential code growth. This particular path of inquiry continued with Kosaraju's work on replacing goto with features that do not have the same drawbacks (usually known by the keywords break, continue, exit, etc) [12]. The work was especially important in that it proved that goto's could be replaced by the specified commands with no extra computation provided that the structures allow exiting an arbitrary number of nested loops.

Kosaraju's work may appear to apply to only those languages that have "exit control" structures that allow escape from multiple nested loops. However, this is achieved in most procedural languages by exiting the current function, implying that function refactoring can be using as part of the transformation technique. Unfortunately, this removes the original guarantee of no increased computation. The guarantee may be returned given a compiler that could compile the function called as if it were a labeled block and not a subprogram. Reliance on this compiler feature (known as "inlining") is seen a great deal in modern *C* programs, which have a conspicuously low number of goto statements. It could be argued that this is a goto-removal technique that has gone undocumented but is fully practiced by a the programming community.

Perhaps the largest boost to the goto-replacement argument came three years later from McCabe's landmark paper describing cyclomatic complexity [14]. Although cyclomatic complexity is a helpful software metric, the important contribution of the paper from the "goto perspective" is the attempt to give a formal definition of a non-structured program. McCabe notes that Kosaraju [12] provided a proof concerning which flow graphs are reducible to structured programs. McCabe also points out that while this is a nice result, practitioners need a way to formally identify a non-structured program. He demonstrates that "a structured program can be written by not 'branching out of or into a loop, or out of or into a decision'." [14]

This result provides the other side of goto replacement; having a formally unstructured program is the cost of not removing goto's that violate loop and decision boundaries. Of course, McCabe also points out that "there a few very specific conditions when an unstructured construct works best" [14], pointing again to the necessity of efficiency over program correctness. Wulf continues this questioning in a published argument against goto [19]. As he explains, "...it is possible to eliminate the goto" but it must be determined "...whether it is practical to remove the goto" [19].

However, Wulf demonstrates that it can be argued that goto's may have already been removed without anyone noticing. That, in fact, if "any rational set of restrictions" is placed on goto, it is "equivalent to eliminating the [goto] construct if an adequate set of other control primitives is provided" [19]. Given that none of the presented efficiency arguments (from Knuth or anyone else) involves drastic uses of goto that preclude "rational" restrictions, in might be argued that goto has already been conceptually removed from any language with sufficient control primitives.

Wulf continues his argument by relating a programming language named Bliss that he co-designed and used as a system language. In addition to user applications, Bliss was used to write an operating system and multiple compilers. This situation appears to be a multi-year experiment in the expunction of goto. The reported Bliss experience would appear to support this argument. However, some may argue that the systems could have been more efficient with some judicious goto usage. This argument ignores the fact that goto's were used in the Bliss system, but were used exclusively by the compiler. Wulf acknowledges the importance of a well-written compiler as a necessary component in goto-less programming. He states that "the appropriate mechanism for achieving this efficiency is a highly optimizing compiler, not incomprehensible source code" [19]. [3]

[3] Wulf also wrote that "[m]ore computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity." [19] This is not as relevant, but it is certainly more amusing.

6 Improved Control Structures and Compiler Optimization Techniques

Wulf's reasoning makes explicit what has been implicit in most of the discussion of the goto statement. The attempt to remove goto from use can also be interpreted as a request for better control structures and better compilers. As often happens, this position is a return to the origins of the debate. Knuth himself pointed out that "new types of syntax are being developed that provide good substitutes for these... go to's..." [11]. However, Knuth set aside the possibility of an optimizing compiler handling the efficiency question because it "would have to be so complicated (much more so than anything we have now)" [11]. Although his proposed alternative ("Program Manipulation Systems") never came to fruition, optimizing compilers have come quite a long way.

There have been a few notable attempts at better control structures. One very large attempt was described by Jonsson as a replacement for code patterns used to replace goto statements (which he called "goto-patches") [9]. This group of structures (known as "pancode" [10]), includes several constructs that target situations usually thought to require a goto for maximum efficiency. These include an *also* statement for an explicitly short-circuited logical conjunction and a *repeat* statement which loops back to the nearest if statement.

Other constructs have also been proposed. Wulf's Bliss language included a "$leave\langle label\rangle$" construct where the label specifies the name of the program section to be left (or exited) [21]. Conceptually this construct has been implemented in languages as a labeled break statement; perhaps the most popular language with this feature is Java.

Another construct that has found eventual use is Bochman's statements for multiple loop exits [4]. This structure is an unlabeled break (envisioned by Bochman as the keyword "exitloop") with special labels for post-loop code execution. There is an "ended" label for code to be executed when the loop terminates normally, and an "exited" label for code to be executed when the loop is terminated by a break statement. This concept has been implemented as an optional else clause for loops in Python.

Circumstantial evidence seems to imply that these constructs have indeed been helpful in removing goto. Neither Java nor Python provide a goto statement [4]. Of course, a cynic may respond that these languages still have efficiency problems since Python is interpreted and Java executes on a virtual machine. As a result, compiler technology must also be considered.

Certainly, Wulf's Bliss compiler mentioned in section 5 (*Goto Removal*) is an answer to the complaint. However, relatively recent work (in 1994) by Erosa and Hendren [8] shows that optimizing programs after automatically eliminating goto's is not only possible but a requirement for some compiler optimizations. The process described in the paper is an automated removal of goto statements as described multiple times above. The elimination of goto's allows for multiple optimizations that would otherwise be impossible. In most situations (with one notable exception), performance was equivalent.

Other compiling techniques that require the elimination of goto's have been described as well. Allen et al described a process for eliminating goto's for their vectorizing compiler project [1]. Ammarguellat described a process based on simultaneous equations to remove goto's for a parallelizing compiler [2]. These two efforts demonstrate the need to remove goto's for optimal vectorization or parallelization.

These efforts are complementary because vectorization allows for parallel instructions within a single flow of control, while parallelization allows multiple flows of control to proceed simultaneously. Gains from these types of enhancements could far outweigh any performance penalties. Consider a program which contains a goto and can be executed in time T. Further imagine that it could be fully parallelized by one of the above techniques. Assuming that the compiler increases the execution time by a factor of α when the goto removal operation is applied, the final execution time would be $\alpha T\frac{1}{N} = \frac{\alpha T}{N}$ where N is the degree of parallelization (perhaps the number of processors). As long as $N > \alpha$, it is more efficient to remove goto's (in this idealized case). As an example, assume that $N = 4$ based on the current multicore chips available on the market for home computers. Unless the goto removal process quadruples the program run time, it will always be more efficient to remove the goto statements.

[4] As a matter of interest, *goto* is a reserved word in Java [15]. However, Java does not implement goto, which formally forbids the use of goto for any reason in Java.

7 Conclusion

In some sense, goto's must always be with us if only in machine code. Of course, this implies that at the very least someone must write a compiler that emits goto instructions. This special case aside, the general agreement has always been that goto instructions are not good for most human-readable programs. Those arguing for the use of goto do so almost universally on the basis of efficiency.

The proper response to this state of affairs would obviously be to create alternatives to goto that are just as efficient. Thus, any situation that appears to require a goto should be considered a requirement for a control structure that can be used instead. This creates an ongoing requirement to improve compiler optimizations so that these structures are at least as efficient as goto statements and maximally as efficient as possible.

Ongoing work in this area has demonstrated that some forms of optimization and parallelization require removal of goto's. As a result, the combination of automated parallelization and ubiquitous multicore hardware will mean that goto removal is required for optimal program execution. As a result, the humble goto might become as rare and esoteric as interrupt suspension – used only by programmers delving into assembly language to deal directly with the physical CPU.

References

[1] J. R. Allen et al. "Conversion of control dependence to data dependence". In: *Conf. Rec. of the POPL-10*. Austin, Texas, 1983, pp. 177–189. Removing goto's as part of a vectorizing compiler.

[2] Z. Ammarguellat. "A control-flow normalization algorithm and its complexity". In: *IEEE Trans. on Software Eng.* 18.3 (1992), pp. 237–250. Eliminating goto's for a parallelizing compiler.

[3] E. Ashcroft and Z. Manna. "The translation of 'go to' programs to 'while' programs". In: *Proc. IFIP Congress 1971*. Vol. 1. Amsterdam, The Netherlands, 1972, pp. 250–255. Attempt at improved automated replacement of goto. Seen by Knuth as an improvement over Jacopini.

[4] G. V Bochmann. "Multiple exits from a loop without the GOTO". In: *Commun. ACM* 16.7 (1973), pp. 443–444. URL: `http://doi.acm.org/10.1145/362280.362300`. Introduces new construct in context of replacing goto's.

[5] C. Bohm and G. Jacopini. "Flow diagrams, Turing machines and languages with only two formation rules". In: *Commun. ACM* 9 (1966), pp. 366–371. This letter started the debate.

[6] E. W. Dijkstra. personal communication with Donald Knuth. 1973. Reported in Knuth's famous coverage of goto; Knuth reports quite a bit of private correspondence when preparing the paper.

[7] E. W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Commun. ACM* 11.3 (1968), pp. 147–148. URL: `http://doi.acm.org/10.1145/362929.362947`. This letter started the debate.

[8] A M. Erosa and L. J. Hendren. "Taming Control Flow: A Structured Approach to Eliminating Goto Statements". In: *Proceedings of the 1994 International Conference on Computer Languages*. Toulouse, France, 1994. Eliminating goto's for an optimizing and parallelizing compiler.

[9] D. Jonsson. "Next: the elimination of Goto-patches?" In: *SIGPLAN Not.* 24.3 (1989), pp. 85–92. URL: `http://doi.acm.org/10.1145/66083.66091`. Discussion of elimination of goto's with new constructs (pancode and boxcharts).

[10] D. Jonsson. "Pancode and Boxcharts: Structured Programming Revisited". In: *SIGPLAN Not.* 22.8 (1987), pp. 89–98. Initial introduction of pancode and boxcharts.

[11] D. E. Knuth. "Structured Programming with go to Statements." In: *ACM Comput. Surv.* 6.4 (1974), pp. 261–301. URL: `http://doi.acm.org/10.1145/356635.356640`. Landmark and lengthy survey of goto usage.

[12] S. R. Kosaraju. "Analysis of structured programs". In: *Proc. Fifth Annual ACM Symp. Theory of Computing*. 1973, pp. 240–252. Proof that goto can be replaced with other constructs like break. Also published in one year later in Journal of Computer and System Sciences.

[13] B. M. Leavenworth. "Programming with(out) the GOTO". In: *Proceedings of the ACM Annual Conference - Volume 2*. Ed. by R. Shields. 1972, pp. 782–786. URL: http://doi.acm.org/10.1145/800194.805859. Good summation of the goto debate, but published prior to Knuth.

[14] T. J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. Introduction of cyclomatic complexity.

[15] Sun Microsystems. *Java Language Keywords*. 2009. URL: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html. From Sun's Java reference. Goto is included as a keyword but not implemented.

[16] P. Naur. "Go to statements and good Algol style". In: *BIT* 3.3 (1963), pp. 204–208. According to Knuth, first published remarks about harmful goto's.

[17] R. Rojas. "Conditional Branching is not Necessary for Universal Computation in von Neumann Computers". In: *Journal of Universal Computer Science* 2.11 (1996), pp. 756–768. URL: http://www.jucs.org/jucs_2_11/conditional_branching_is_not. Demonstrates that only four instructions required for von Neumann computers: load, store, inc, and goto.

[18] F. Rubin. ""GOTO Considered Harmful" Considered Harmful". In: *Commun. ACM* 30 (1987), pp. 195–196. Claim that harmful goto position was dogma and not logically based.

[19] W. A. Wulf. "A case against the GOTO". In: *Proceedings of the ACM Annual Conference - Volume 2*. Ed. by R. Shields. 1972, pp. 791–797. URL: http://doi.acm.org/10.1145/800194.805861. Includes discussion of efficiency and goto drawing on his experience with the Bliss language.

[20] W. A. Wulf. "Bliss: a language for systems programming". In: *CACM* (1971). Uncited and provided for user convenience. Wulf's article detailing the Bliss language.

[21] W. A. Wulf. "Programming without the go to". In: *Proceedings 1971 IFIP Congress*. Ljubljana, Yugoslavia, 1971. Introduces the *leave* construct.