Using assertions to improve language translators

by ARTHUR PYSTER
University of California
Santa Barbara, California

ABSTRACT

New enhancement techniques for language translators based on work in program verification are developed. Assertions are normally added to a program in order to verify the program is correct. Once verified, the assertions are usually ignored. This paper shows that verified assertions often contain information which can improve certain object code characteristics when the program is translated: execution time, storage requirements, and program style. The latter quality is especially important if the object code is itself in a “high-level” language. The techniques developed fall into three categories: early binding, using complementary constructs, and noting restricted cases.

INTRODUCTION

Enhancing translator output is an area of enormous practical concern. Optimizing object code is perhaps the most obvious enhancement a translator can make. Nearly all translators available try to be somewhat clever in reducing the run time and storage requirements of the code they generate. Even with increasing processor speeds and cheapening memory costs, time and space optimization will continue to play an important role in efficient computer utilization. Making full use of the more abstract features the target language offers is another form of translator enhancement. This is particularly true when the target language is itself high-level, therefore having many complex features to apply. For example, in translating into FORTRAN it would improve the style of the object code if DO-loops were generated rather than more primitive “IF,INCREASEMENT, GOTO” loops. Of course, the ability to translate between high-level languages increases portability when moving from one computer system to another where the languages or language dialects available differ. This paper develops new enhancement techniques based on work in program verification.

The increasing concern over program verification has created new opportunities for translator enhancement, especially from the specification and verification of program assertions. Assertions typically state properties of programs such as the range of variables or the relationship between the values of two or more variables. Once verified, these assertions may be treated as an integral part of the program body and hence information drawn from them may be used to enhance translator output. Of course, it is absolutely imperative to the correctness of the enhancement that the assertions made are, in fact, true. Otherwise the enhanced code could exhibit different input/output behavior than the source program! With automated verification systems, there is little likelihood of error except in specifying the input assertions. But an error here quite possibly indicates the programmer misunderstands the problem specifications and hence the program would likely fail independent of the optimization.

The paper itself has six sections. The second section discusses assertions in general. The third, fourth and fifth sections detail three enhancement strategies assertions make possible: (1) early binding, (2) complementary constructs, and (3) restrictive cases. Finally, the last section summarizes the paper’s contents and indicates future lines of work.

ASSERTIONS

The notion of program assertions is credited largely to Floyd\(^1\) in a classic paper on proving programs correct (although he called them “verification conditions”). Assertions are conditions on the commands of a program such that each time a command with an assertion is reached, the condition should be true at that point. This is shown in Figure 1. The particular formats used to state assertions in this paper are self-explanatory and do not require detailed introduction.

Verified assertions can add a new level of abstraction to a program: Assertions (a) and (b) might be

\[
\begin{align*}
(a) & \ x \text{ IS A stack;} \\
(b) & \ y \text{ IS A tree;}
\end{align*}
\]

found in a PL/I list processing program. Since stack and tree are not primitive PL/I data-types, the PL/I code itself would not contain this information directly about lists \(x\) and \(y\). \(x\) and \(y\) would be described in a far more primitive

---

manner within the code. A translator might use such “cheaply” obtained information in the assertions to optimize storage allocation or structure processing. Examples of this are given in the fifth section of this paper. The cost of determining that x and y are stacks as part of an optimization step would probably be prohibitive. Moreover, without the “machine-understandable” statement in the text that x and y are special types of lists, the compiler would not even know which type of list structure x and y might be!

In a sense, assertion statements are language extensions. Currently there are no standards established for their syntax in the way ANSI has established standards for COBOL and FORTRAN. Within the next few years, a movement for standardization will probably emerge. At the point when assertion formats stabilize, translator writers will have a firm basis for treating assertion statements as an integral part of the host language and will then be able to consider assertion properties in their enhancement strategies.

EARLY BINDING

Binding in programming languages is establishing the mappings between names and data objects, and their descriptions. For example, the PL/I declaration

DECLARE X FIXED DECIMAL (3,1) AUTOMATIC;

binds the name X to a data object which holds three-digit decimal numbers whose values range from -99.9 to 99.9, inclusive. Furthermore, the memory location for storing the value of X is by virtue of the term "AUTOMATIC" in the declaration allocated when the block containing this declaration is entered.

Different languages have radically different policies on binding time, the time in the history of program execution when bindings are established. In general, FORTRAN binds as early as possible, while APL binds as late as possible. The common rule of thumb is that early binding is cheaper to implement but less flexible than later binding. If a translator tries to convert code in a language with a late binding time policy to code in a language with early binding time policy, serious difficulties can arise. For example, ALGOL permits a program to determine array dimensions dynamically upon block entry. FORTRAN binds array dimensions at compile time. Therefore, an ALGOL program which relies on dynamic dimensioning cannot be converted in a straightforward manner to FORTRAN. However, if the translator knew the maximum value the ALGOL program will, in fact, use to set array bounds, it could allocate a static array in the FORTRAN program with this maximum value for its bounds. Of course storage would sometimes be wasted by the FORTRAN program, but avoiding such waste is one of the main reasons ALGOL uses dynamic memory management. Unfortunately, there is no vehicle in ALGOL for specifying the range of a variable. Hence, no translator can, in general, perform this straightforward translation by relying on the ALGOL text alone. However, it is quite common for an assertion about the range of a variable to appear in a program:

ASSERT 1<=N<=100;
INTEGER ARRAY B(N);

This assertion, which is not part of the ALGOL language, makes it possible to intelligently bind the array size earlier in the FORTRAN program than in the ALGOL original. Without the assertion, there is no clear strategy for handling the binding time difference problem.

The problem with different binding times crops up again in translating from a typeless to a typed language, such as APL to PL/I. An APL variable can freely hold a character string at one instant and an integer or real number the next. PL/I requires a programmer to declare the data-type of each variable at compilation time and to maintain that data-type for the entire program run. Therefore, knowing nothing about the range of values an APL variable will assume, a translator cannot easily substitute appearances of APL variables with PL/I counterparts. However, through the use of assertions about the source program, it is conceivable that the range of many APL variables could be determined. In many cases these variables may have values of only one data-type assigned to them. For such variables the translation would proceed quickly. Having the proper assertions present in this case would then greatly simplify the object code.

Assertions can also be used to optimize compiled code with respect to binding time. Consider again an ALGOL array with dynamically computed storage bounds:

INTEGER ARRAY B(e);

where 'e' is a positive integer-valued expression. Every time the containing block is entered, compiled ALGOL code would probably recompute the value of e. There are many circumstances under which the value of e would be
constant over a long period once it had been computed at run time. In these cases, it would be more efficient to save the value of e and re-use it, rather than repeatedly recompute it. An assertion to the effect that e remains constant over a specified time period would make this possible. An alternative way to avoid recomputing e would be to keep track of whether or not the values of the component variables of e change between block entries. This is an expensive bookkeeping operation. Furthermore, it will not handle situations in which the values of expression components change, but the overall expression value does not, such as in

\[ \text{INTEGER } B(I-J); \]

where the difference between I and J could be constant even if the values of I and J individually change.

**COMPLEMENTARY CONSTRUCTS**

This section is largely founded on the premise that it is better to use "high-level" constructs in the object code whenever possible. This practice not only enhances program readability, which could be important if the object code is itself in a high-level language, it can also lead to improved time and space bounds for the high-level translator output when it is itself compiled into machine code. The second advantage arises from the fact that special optimization techniques can often be developed to deal with complex but well-structured constructs. For example, FORTRAN DO-loops are often set up using a machine-language looping statement such as "Branch and Count" on IBM 360 hardware. It is more difficult to detect that this same fast construct is applicable if the more primitive "IF,increment,GOTO" form of loop is used instead.

Sometimes there are features in both source and target languages which seem analogous in purpose and often form. One such pair is the ALGOL FOR-loop and the familiar FORTRAN DO-loop. In translating from ALGOL to FORTRAN, however, it is not always possible to substitute a DO-loop for each FOR-loop occurrence in the source program. There are several important differences between DO- and FOR-loops even though they both have basically the same function. FORTRAN DO-loop parameters are restricted to be all integer variables or constants which have positive values. ALGOL FOR-loop parameters can be any arithmetic expressions. Furthermore, DO-loops are executed once even if the loop predicate is initially false. FOR-loops are skipped completely unless the loop predicate is initially true.

As a consequence of the differences between DO-loops and FOR-loops, it is not possible in general to substitute a DO-loop for a FOR-loop in the object code. Substitution is possible only when the translator knows that the FOR-loop parameters all have positive integer values, and that the loop predicate is always initially true. Without these guarantees, the object code must be gerry-rigged to accommodate the behavioral differences. This latter act slows the program, increases program size and hinders readability. With proper assertions inserted into the ALGOL source program, for those cases where the FOR-loop does behave in a manner equivalent to a DO-loop, the translator can generate the simple object code. In some cases it should be possible to translate the assertions automatically as well. Figure 2 illustrates the differences between translation with assertions and translation without.

A second example of complementary constructs is a built-in square-root function in the source and target languages. Suppose the source language has a complex number primitive data-type and the target language does not. It would be wrong to translate square-root function to square-root function unless the function argument is never negative. Computing the range of the argument may be prohibitively expensive or even impossible to compute using the source text alone. However, the proper assertions could make this determination feasible, if not trivial. The unappealing alternative is for the translator to construct its own square-root function whose range includes complex numbers. Since complex numbers are not a primitive data-type of the target language, they would have to be simulated using a 1x2 matrix or some similar vehicle. Such efforts would horribly muddy the object code without reason if, in fact, the function argument were never negative.

The two examples just cited are instances of a general phenomenon which is pictured in Figure 3. The parameter or argument space of a language construct is the domain of its input parameters or arguments. Figure 3a shows the union \( U \) of the parameter spaces of constructs A and B. The intersection of their parameter spaces, \( A \cap B \), is the area with diagonal hatches. The circle \( A \cap B \) within \( A \cap B \) encompasses all common data points for which A and B behave identically. The larger \( A \cap B \) is with respect to \( A \cap B \), the greater likelihood that the straightforward translation is possible. Assertions are helpful in determining if a particular data point is in \( A \cap B \) or \( (A \cap B) - (A \cap B) \).

Figure 3b shows a situation related to, but distinct from that of Figure 3a. It is possible for constructs A and B to
behave identically for a significant number of distinct data points. For example, construct B may behave the same for integer input 3 as construct A does for real input 3.0, but B may be undefined on real input. In that case, a conversion $f$ to the correct data-type in the object code would make it possible to emit the complementary construct B as the generated code. Assertions can identify whether such conversion is always possible; e.g., that construct A never has argument 3.5. Of course, the limiting factor here is the ease with which the conversion can be accomplished. Converting real to integer is fairly trivial, but converting sequential arrays to linked arrays can be costly.

**RESTRICTED CASES**

Often an operation or data item is restricted in some way which is difficult if not impossible to detect from the program text itself. A translator could often employ knowledge of such a restriction to advantage. Assertions can provide that information at low cost. For example, a program could assert that array X is diagonal or sparse. Storage could be conserved by allocating only half of the indicated space for the diagonal array and by using linked allocation rather than sequential for the sparse array. Having restricted cases of operations and data is common. Several other places where optimization is possible are:

1. In ALGOL, SNOBOL or any language which treats all subroutines as potentially recursive, assert a subroutine is not recursive and forgo expensive run-time set-up.

2. A high-level list processing language could have a "search for node $X$" operator. If it is asserted that the graph is acyclic and/or connected, the search is simplified.

3. For the FORTRAN computed GOTO, skip the test in the compiled code for the index variable not being between 1 and the number of alternative branches if it is asserted it always will be.

4. In a list-processing language skip the test for stack underflow if it is asserted the stack is never empty.

5. If the asserted range of a data item is small, then storage allocation could be less than would otherwise be efficiently possible. This is especially important in translating between dialects of a language which are implemented on machines with different word sizes; e.g., going from CDC-6600 FORTRAN with 60-bit words to IBM 360 FORTRAN with 32-bit words.

6. If it is asserted that string B is always a substring of string C, then when searching for the position in C where B begins, a recovery for a failing pattern-match can be eliminated.

**CONCLUSIONS**

This paper has demonstrated that assertions can be profitably applied outside their original context of program verification. In particular, several strategies for enhancing the object code generated by language translators, including but not restricted to compilers, have been developed.

Assertions can be viewed as language extensions. As such, they allow the translator to compensate for deficiencies of the host language whose programs are being translated. Thus, possible enhancement in style, speed and storage requirements of the object code are simply a beneficial side-effect of efforts in verifying program correctness. Consequently, the programmer need not concern himself with which assertions would be most profitable. There are many enhancement strategies made possible through effective use of assertions. Only a handful have been mentioned here.

There is a strong similarity between assertion usage here and the notion of a language preprocessor. The key difference is that a preprocessor is written to avoid modifying a compiler, while this work urges compiler changes for the sake of efficiency and style. Typical preprocessor extensions add new commands to a language. Assertions add new descriptors. There is usually no vehicle for expressing these descriptions in the host language; e.g., variable range in FORTRAN. Hence normal preprocessing techniques are not applicable.

Some of the inflexibilities of standard languages can be circumvented by the language augmentations assertions offer. It should be interesting to see how such features which enter a language through the back-door are applied.

**REFERENCES**


