A Facility for the Downward Extension of a High-Level Language

Thomas N. Turba
SPERRY UNIVAC
P.O. Box 43942
St. Paul, Minnesota 55164

Key Words and Phrases:
Assemblers, Code Generation, Compilers, Efficiency, Encapsulation, Extensibility, High-Level Languages, Inline Code, Machine Code

Abstract

This paper presents a method whereby a high-level language can be extended to provide access to all the capabilities of the underlying hardware and operating system of a machine. In essence, it is a facility that allows a user to make special purpose extensions to a language without requiring the compiler to be modified for each extension. Extensions are specified in an assembler-like language that is used at compile time to produce executable code to be combined with compiler-generated code. This facility has been implemented in a systems-programming language and was designed to provide access to facilities not directly available in the language. The way in which the facility was implemented calls for a minimum of user-visible language changes and is well suited for generating code sequences for any language. The facility provides the extension writer access to information available in the high-level language during compilation, permits the selective generation of user-defined code sequences depending on the context in which they are being used, provides for the integration of this code with compiler-generated code, and provides for the generation of user-understandable error messages when an extension is used incorrectly.

Introduction

Many high-level languages have facilities for upward extension by a user. Normally this extensibility is provided by specific language constructs to define and operate on new data types; although, sometimes it is provided by a macro facility that simulates these constructs. In either case, the extensibility consists of defining new data types and operations on them in terms of existing language primitives. Because of this, an extensible language does not provide any increase in capability over that provided by the basic constructs of the language. What is provided is the ability to work with application-specific data types in a way that is more natural to the application area. This is a valuable goal and a tremendous step forward in programming languages, and is not meant to be down-played. However, extensible languages do not provide any new capabilities to a language, only variations on expressing existing capabilities. Therefore, some operations either cannot be done in an extensible language, or they are not done as efficiently as they could be. In particular, programs that make extensive use of user-defined operations on new data types can have a problem because these operations are normally not as efficient as built-in operations.

Languages as old as ALGOL and as new as Ada have considered the inclusion of assembler or machine-dependent code in their programs important enough to have indicated the need for such a facility in their defining documents. Some of the reasons for providing such a facility are that it:

- Allows the use of operating system requests without having to build them into the language,

- Allows access to specialized hardware instructions not normally generated by a compiler, and

- Allows a new program written in a high-level language to interface with existing code written in another language (usually assembler) without needing interface routines.

Even with these reasons and the desire to provide such a facility, little has been defined regarding the actual capabilities and how such code should be integrated with compiler-generated code. This lack of definition is no doubt partly responsible for the fact that few compilers provide this type of facility and most of those that do provide a facility appear to have a very limited capability. (A complete determination is hard to make because most compilers that have such a capability are for proprietary languages of computer manufacturers and documentation is not generally available.)

This paper discusses the general considerations of a facility for the integrated inclusion of user-defined code sequences in compiler-generated code. The paper is based on a facility implemented on Univac Series 1100 computers in a high-level systems-programming language called PLUS (Programming Language for Univac Systems). Since its original implementation, the facility has evolved from a straightforward mechanism for accessing system functions and specifying machine instructions to a general facility where users can specify the insertion of sophisticated context-tailored code sequences that can be effectively integrated with compiler-generated code.

Interface Points

When designing a facility for the inclusion of assembler-like code, one of the first considerations is where such code should be allowed. Although it is conceivable to allow the insertion of such code anywhere in a program, this would have implementation problems. In addition, it would not provide a well-defined user interface, which is of equal, if not greater, importance.

Another consideration is whether a sequence of code will be used only once or in numerous places throughout a program. In the design of the system described here, it was felt that the normal case would be for a code sequence to be used repeatedly in many places. Therefore, the approach taken was to provide a facility where a code sequence would be defined once and then later used in many places.

The interface points where assembler-like code can be used should correspond to logical units in a language. Examples of such units include: procedures, functions, operators, and statements. The units chosen in the PLUS system were procedures and functions. Operators and statements were not included because the language does not provide user extensibility in these areas. There are, however, no logical differences if operator and statement interface points are included.

The declarations in the high-level language for the inclusion of assembler-like code are similar to procedure and function declarations. The forms differ in that user-defined code (called inline code) permits the definition of generic procedures and functions which can accept a different number of arguments as well as arguments of different types. Such generality need not be included in an *inline* facility, but if included implies some special considerations. For example, the number and type of parameters are not specified or checked in the high-level language. This checking is performed by the inline writer with special constructs in the lower-level code. The declarations in the high-level language have a form of:

INLINE *name*; and
INLINE *name* RESULT *type*;

As indicated by the second declaration, the result type of an inline function is indicated along with the name. In some respect, this could be considered an anomaly because although generic argument types can be accepted, a generic type cannot be returned. This restriction, however, is consistent with the rest of the high-level language which is strongly typed in the sense that a function must have a specific type associated with its result. The inline language, on the other hand, is not strongly typed and it makes sense not to do type checking in the high-level language when parameters are passed down to that level.

Passing Information

For the inclusion of assembler-like code to be generally useful, there must be a means to pass

information from the high-level language to the lower-level code. Such an information port could conceivably be totally general and permit interrogation of all information known to the compiler at the place where inline code is being used. In the design of the PLUS system, such generality was not felt necessary or desirable. The approach taken was to allow information to be obtained on the arguments supplied with each use of an inline. This is done with a set of built-in interrogating functions in the inline language. These functions include retrieval of information on:

- The number of arguments supplied with the use of an inline,
- The data type of an individual argument (e.g., integer, real, character),
- The class of an argument (e.g., constant, automatic variable, static variable),
- Access information for an argument (e.g., storage address, size, bit offset), and
- The availability or status of an argument (e.g., presently loaded in a register).

These functions constitute the information port into the assembler-level code facility. They provide information on the actual arguments used with an inline so that the inline writer can specify code that will change depending on the actual arguments used with it.

Code Generation

The specification of machine instructions is done in a language that looks much like assembler language. This language permits access to all the machine instructions, but is a simpler language that does not include many of the more complex features of the assembler such as macros. In addition, it does not provide some facilities normally expected in an assembler such as the definition of entry points and static variables. This simplicity is acceptable because inline code generally consists of small sequences of straightforward code that is to be integrated with compiler-generated code rather than complete programs. Those portions of the inline language that do the same thing as the assembler have the same syntax so that programmers need not learn another language. New features, such as the interrogating functions, are provided in a form that is consistent with similar assembler features. These functions can be used to supply information to instructions that are dependent on argument characteristics and to control statements that determine the actual code to be generated.

One control structure essential for selective code generation is a mechanism that includes or excludes code based on the result of a conditional expression. If the conditional expression is true, code will be generated; otherwise, it will not. This type of control structure is needed so that different code sequences can be generated from a single inline depending on the type, number, and class of arguments that are supplied with each use of it. The control structure used in the PLUS system is a simple ON-OFF that corresponds to a similar control structure in the assembler. It is the equivalent of an IF-THEN in a high-level language. The form is:

label ON *conditional-expression*
. } Statements to be included
. } if expression is true.
OFF

The selection of different code sequences is similar to the binding of different code skeletons in a compiler, i.e., different code sequences are generated depending on the values associated with each use. The user of the inline facility is in a sense augmenting the set of code skeletons that are available in a compiler.

In addition to the ON-OFF control structure for the conditional generation of code, there is also an ESCAPE statement that permits an exit from a code sequence (i.e., the remaining code for that sequence will not be generated if an escape is done). ON-OFF sequences can be nested and a multi-level escape performed by referencing a *label* attached to an outer ON.

Another control structure in the PLUS system is an ENTER statement which is the equivalent of a local procedure call. It consists of a reference to another inline within the same source text that is to supply code at the point it is used. After adding code from a referenced inline, code generation resumes in the original inline. Each inline is, in effect, a separate entity with local scope for compile-time variables, labels, and other names.

Additional control structures, such as a CASE statement, have been proposed and could be added at a later date.

Interface Directives

An interface directive is the counterpart of an interrogating function; much like an interrogating function provides information to an inline, an interface directive provides information from an inline back to the compiler. Together, they provide for a flow of information both into and out of an inline so that code can be effectively integrated with surrounding compiler-generated code.

An example of an interface directive is one that informs the compiler about the location of a value returned from an inline function. Normally, a function result is returned in a predefined register. However, if it is advantageous to return a result in another register, this directive can be used to indicate the register that will contain the result.

Another directive for passing information back to the compiler is one that informs the compiler that the value of a variable will exist in a register at the end of an inline sequence. The compiler can then take advantage of this information, if the value should need to be loaded for later code generation.

In addition to these directives for integrating code generation, the inline facility also has a directive that permits the sharing and integration of working storage. This directive enables inline code to request and be allocated working storage from the automatic stack associated with a procedure invocation. This storage can be used for holding temporary results and other transient information during the execution of inline code. Inlines can also access the storage for static global variables that have external names and use the storage associated with arguments if they are passed by reference.

Register Handling

As already has been implied, the inline facility described here was implemented on a computer with a register architecture where values must be loaded into registers to be operated on. Because compilers for such an architecture normally perform optimizations where values are kept in registers for later use, some method had to be found so that compiler optimization is not degraded when an inline is used and also assures that inline code has registers to use. Although it would have been possible to require inline code to save and restore the registers it uses, or to inhibit optimization across an inline use, both of these alternatives were rejected for efficiency reasons.

The solution chosen was to have a convention where a set of "volatile" registers are available for inline use which the compiler does not use for holding information across inline code. This set of registers consists of 7 of the 28 general-purpose arithmetic and index registers and 4 of the 16 repetition registers available in the machine (i.e., 1/4 of the registers are in the volatile set). This approach leaves sufficient register space for optimization. If an inline should need more registers than are in the volatile set, or a specific register that is outside it, a save and restore can be done in the inline code.

The use of a volatile register set was an acceptable solution in the PLUS system because of the large number of registers available and because an inline code sequence generally will be small. If very few registers were available, or inline code was to be large, a more sophisticated approach could have been taken. One such approach is to have an inline declare what registers it uses and have the compiler generate code around this register usage. Another approach is to have the inline code written using generic registers and let the compiler bind them depending on the surrounding code. Both of these approaches are acceptable and have been tried. They require only slightly more work for the compiler to integrate inline code with compiler-generated code. It should be noted, however, that if inline code is being used to interface with existing code that has established register conventions, the inline code must be able to dictate what registers are used. This is often the case when inline code is to access system functions or existing libraries.

Another case when inline code may need to dictate specific registers is if it establishes a run-time environment. Examples include the use of a register for stack maintenance or for communication purposes. If such a register is to contain information across inline uses, it may be necessary to dedicate it and inhibit its use by the compiler. This can be done by a directive to the compiler indicating that it cannot use a specific register.

Error Cases

So far, we have only been considering the aspects of generating and integrating inline code. This, however, in only part of the problem. Once an inline is written and tested, it will be used; and as usual, it will often be used incorrectly. For example, it may be passed the wrong number or type of arguments. Such errors must be considered and handled. Preferably, it should be possible to issue an error message of the same quality for an error in the use of an inline as for an error in the use of a built-in function. However, even if this is not possible, some form of an error message facility must be provided.

The way error messages are handled in the PLUS system is via a display statement in the inline language. A display statement is normally activated by a conditional statement designed to check for an error or is specified as the last alternative in a set of possibilities. When a display statement is encountered in generating inline code, the compiler will produce a message in the output listing just as if the compiler itself had detected the error. Such a message can be prefixed with the word *error, warning,* or *message* and contains the name of the inline where the error was detected, the line number where the inline was used in the high-level program, and descriptive text supplied by the display statement. This text can vary in content on each activation of a display statement and contain information associated with the incorrect use of the inline. For example, it could contain a number indicating what argument caused an error.

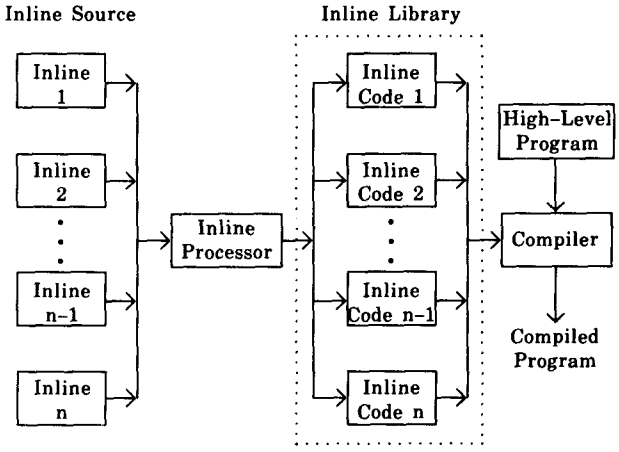
Implementation Overview

The way that inline code is handled in the PLUS system is different from other systems known to the author. Most other systems have assembler code appear directly in the high-level language. This is normally done by having a directive that indicates to the compiler that the following code is in assembler rather than the high-level language. The approach taken in the PLUS system is to have all inline code kept outside of the high-level language it is used with. The only statements that appear in the high-level language are declarations and references. A declaration defines the name and, if a function, the result type of an inline. A reference appears as a normal procedure or function call.

All inline code is specified in one or more source texts that are separate from that of the high-level language. In addition, inline code is placed in a separate library before it is used by the compiler. This library is kept in an internal format for easy use by the compiler, rather than as source text. This internal format is created by a special processor as a separate step.

The declaration of an inline name in the high-level language establishes a link to the inline library; code is brought in from the library as needed by the compiler. Each reference to an inline name in the high-level language causes the generation of an occurrence of the code sequence specified in the inline. As indicated earlier, the actual code sequence generated for each use of an inline may be different depending on the arguments that are supplied to it.

The generation and use of inline code is depicted in the following figure.



Use of the Inline Facility

The separation of definition and use was done for several reasons. These include:

- The desire to isolate the writing of inline code from the writing of high-level code. This was felt necessary so that the advantages of using a high-level language would not be lost.
- The desire to encourage the design and use of inlines on a project basis rather than on an individual basis. To further encourage this, each compilation is restricted to accessing a single inline library.

- The desire to provide a high degree of encapsulation so that a user does not need to see unnecessary information. All that a user needs to know, and all that is seen by the average user, is the inline interface declaration.

- The desire for an efficient implementation. Because inlines are kept in an internal format, it eliminates the need for the compiler to reprocess an inline definition each time it is used.

- The desire to permit an inline to be shared by modules that are separately compiled. This sharing can be within a project or across projects that have a common inline library.

There is one disadvantage to the separation of definition and use. This disadvantage is seen by the definer of an inline who must perform two steps to test an inline. First the inline must be processed into the internal format and placed in a library; and then, a compilation must be done to test it. However, because definition is considerably less frequent than use, this is not a major disadvantage and may even encourage a more complete testing of an inline because it is a separate step.

In addition to the reasons that promoted the separation of definition and use of inline code, some other equally valid reasons have been noted. These are:

Language Independence. The separation of definition and use permits the creation of a single facility that can be used with any high-level language. All that is needed are minor extensions to a language to indicate that an inline is being used and what its name is. These extensions can be made in a way that is compatible with the language so that there is no need for a different lexical analyzer or parser to handle inline code. These would otherwise be required for the compiler of each language that was to provide an inline facility if the facility was integrated with each compiler. The only important changes needed for a compiler are in its later stages where inline code must be integrated with compiler-generated code. This can often be handled in a common manner and, with a proper design, can even use the same code.

Machine Independence. The separation of definition from use also provides for machine independence because machine-specific code is not included as part of a program. Therefore, a program that uses inline code sequences can be transported to another machine as long as the new machine also has an inline processor. This was, in fact, done in the PLUS system which has compilers on more than one machine. However, even if an inline facility is not available on the new machine, the transportation of the code will be considerably easier due to the isolation that is enforced between definition and use. In the case of the PLUS system, inline uses can be replaced by procedure and function calls.

Environment Establishment. Although inline code cannot ignore the normal code generation conventions used by a compiler and its run-time system, the use of inline code can provide for an invariant run-time environment that is not bound to a particular compiler implementation. Therefore, a project can establish its own environment and isolate itself from possible or expected compiler changes.

Internal Representation

The internal format of an inline that is seen by the compiler can best be described as a set of interpretive instructions that corresponded to the instructions in the inline language. The inline processor is, in effect, a compiler for the inline language that produces interpretive instructions. These instructions are executed by an interpreter within the compiler. This execution, which brings together the information of a compiled program and an inline, determines the actual code to be generated for an inline.

There are two types of interpretive instructions. Those that generate code and those that do not. The instructions that generate code correspond in a one-for-one manner to the instructions in the inline source that specify actual code. The instructions that do not generate code are used for the evaluation of arguments, computation of values, generation of messages, selection of code sequences, conveyance of information to the compiler, and performing other related activities.

Possible Changes

The inline facility described here has evolved through use and is still changing as new features are deemed necessary. There are many enhancements and changes that could be made in a future or different system. Some of the areas where such changes could occur include:

Parameter Safeguards. The present system places the responsibility for parameter checking upon the inline writer. Some of this responsibility could be assumed by the inline interpreter and compiler. This could be done by having the set of possible argument types declared in the high-level language and have the compiler perform checks for invalid argument types. In addition, the inline processor could assure that all argument types have been accounted for by the inline writer.

Register Load Facility. Presently, the inline writer must supply all the instructions that are needed to load an argument into a register regardless of its size or storage alignment characteristics. Although this can be done because the inline writer has all the information required, it would be less error-prone and tedious if the system had generic load instructions that would generate any sequence of instructions needed to load an argument. A similar statement can be made for a store to an arbitrarily aligned record field.

Generic Output Types. The present facility requires that the result type of an inline function be declared, and is therefore fixed. Because inlines can accept generic arguments, it would be desirable if they could also return a generic type. This would permit the specification of generic functions such as MAX and MIN where the type of the result is dependent on the arguments passed. Such a type could be one of a set of types associated with an inline declaration, or it could be a type dynamically selected by an inline depending on the argument types passed to it without the set of possibilities being specified beforehand.

Generic Registers. As indicated earlier, inline code writers are given a set of volatile registers that they can use without the need to save and restore their contents. This was possible because the machine on which the system is used has a large number of registers. This approach may not be possible on machines that have a small register set. A good solution in such a case is to have generic registers and let the compiler determine which registers to use with the code. This approach could be combined with specific register handling where needed. If such a hybrid approach is used, the compiler would need to allocate registers around those instances that are bound by the inline writer.

Conclusion

Experience with the inline system described here has shown that it is both possible and practical to provide facilities for the downward extension of a high-level language, and that these facilities can be effectively and efficiently integrated with compiler-generated code. The essential characteristics of such a facility include:

- The choice of well-defined points where inline code is allowed,
- Interrogating functions for obtaining information about the arguments supplied with a use,
- Control structures for selective code generation,
- A method for allocating registers,
- Directives to pass information back to the compiler, and
- An error message capability.

It is not expected that all high-level language users, or even a majority, will have a need for such a facility. However, for the few that do, access to machine instructions and operating system primitives can be provided in an isolated way that does not have a major effect on the compiler and does not compromise the benefits of using a high-level language. It is believed that the generality and encapsulation features of the PLUS inline system fit well with other high-level languages, such as Ada, where there will occasionally be the need to access more primitive operations than those directly provided in the language.