Fortran 95 support in GCC

Paul Brook
paul@nowt.org

Abstract

This paper details the current status of Fortran 95 language support in GCC, with reference to the future targets and goals of the g95 project. Some of the problems encountered and design decisions made in the process of interfacing with the GCC backend code generator will also be discussed.

1 The Evolution of Fortran

Fortran is a programming language primarily designed for performing computationaly intensive mathematical tasks. Indeed the name itself is derived from the words FORmula TRANslation.

Common uses include Finite Element and Computational Fluid Dynamics codes. Authors of Fortran programs are often not professional software developers. It is commonly used in academic research situations where the primary goal is the analysis and solution of the problem, rather than the development of the software itself.

Fortran was originally implemented by IBM as an alternative to assembly language for programming its 704 systems. The development of the language started in 1954, with a manual published in 1956 (there are rumors that the first customer got a preview compiler without manual in December 1955). The first ISO Fortran Standard was released in 1966. Since then, the standard has undergone four major revisions. These are typically named by the year they were released.

Possibly the most significant changes were introduced in the Fortran 90 standard. Many new features were introduced, with the aim of ensuring the language remained viable for use on modern computing systems.

Fortran 90 introduces powerfull array handling facilities. It allows operations to be performed on whole arrays or sections of arrays in a single expression. From the compiler writer's view this is the most complex feature of the language from, as these must be converted into a collection of scalar operations. It also provides opportunities for the compiler to apply more advanced optimization strategies.

The concept of derived types (analagous to C struct types) was also introduced. While many Fortran vendors had previously provided ways to access and manage dynamically allocated storage areas these were only standardized in the Fortran 90 standard.

As well as these additions to the functional capabilities of language, several other syntactical additions were made. These include modules to aid code modularity and reuse, explicit procedure prototypes, block based flow control constructs and the removal of restrictions on

the source form imposed by the use of punch paper cards (so-called Hollerith cards).

Fortran 95 contains mostly minor changes relative to Fortran 90, and removes some of the features that were deprecated with the advent of Fortran 90. However the majority of Fortran 77 code is still legal under Fortran 95 rules.

2 The g95 project

The existing GNU Fortran compiler is widely respected, and a very competent compiler. However this is limited to Fortran 77 code. Even the author of g77 didn't believe that one could make a full Fortran 95 compiler based on the existing g77 code. Writing a new frontend from scratch means g95 is not restricted by design decisions made in g77, and is more easily able to take advantage of new technologies introduced into the common GCC middle- and back-ends.

Thus Andy Vaught created the GNU Fortan 95 project. Initial work concentrated on parsing and correctly resolving Fortran 95 source code.

Only in June 2002, when the parser and resolver were mostly complete, did work begin on the code generation pass and interfacing to the rest of GCC. For this reason g95 is able to correctly parse and verify almost all Fortran code, however it is only able to generate executable code for some of it.

Work is currently concentrated on implementing the few remaining constructs, and completion of the IO and runtime libraries.

Steven Bosscher and I created a fork from the original g95 code in January 2003. This is done in an attempt to achieve closer integration between GCC and g95, and to promote a more open development environment.

3 The Parser and Resolver

Fortran grammar predates most modern parsing techniques. It does not distinguish between keywords and identifiers, and in some cases the meaning of an identifier can only be determined from the way it is used. In other cases the same line of code can have different meanings depending on the context in which it is encountered. It is possibly to write automatically generated parsers for fortran. However these are qute complicated as there is not a clean seperation between lexical, syntactic and semantics analysis. G95 uses a hand crafted pattern matching parser which often operated in a recursive manner.

The majority of error checking and name resolution is done in this first pass. During this process a tree structure is contructed to represent the code. Each statement is represented by a node. These are linked together in lists to form code blocks. These are referenced by flow control statements. For example an IF statement node contains pointers to an expression node for the condition, and expression nodes for the true and ELSE blocks.

Constant folding and simplification of intrinsic functions is also performed while building this tree.

This tree is then traversed in a second pass to perform type checking, insert implicit type conversions where necessary, and to resolve overloaded functions. We also resolve calls to intrinsic function calls to the corresponding runtime library function.

After these two passes, the code tree is fully resolved, and any errors will already have been rejected. The completed tree is passed to the code generation interface one program unit at a time. A program unit is a module, top level subroutine or function, or PROGRAM block.

The first two passes are now almost complete, with legal code being parsed correctly. Most illegal code is detected and rejected, however there are still some constraints which are not enforced.

4 Interfacing to GCC

G95 uses the GCC middle end and back ends to perform code generation and optimization. It is currently targeted at the tree-ssa branch of GCC. This uses a language independant, tree based intermediate representation of the code. This is very similar to the tree produced by the parser, except it can only represents scalar operations.

The GCC tree-ssa branch also provides a cleaner seperation between the language specific fontends and the common backend. Previous versions were still quite closely tied to the C frontend.

The translation of scalar code is mostly straighforward. After some initial setup this is simply a matter of transcribing the tree from one data format to the other. This is done by recursively walking the code tree, building the equivalent GCC tree as this is done.

The main complication is that some expressions require additional code to be associated with them. The solution is to use a state structure when translating expressions. This state structure contains the expression itself, and two code blocks. The pre block contains setup code which must be executed before the expression is evaluated. The post block contains code to clean up after the value is no longer needed.

For the majority of scalar operations both the pre and post blocks will be empty. However Fortran allows more complex operations which may require additional code. One example of this is passing the concatenation of two strings as the actual argument of a function. The pre block will contain code to allocate temporary string storage and perform the concatenation. The expression itself will consist of the function call with the temporary as the actual argument. The post block will contain code to free the temporary storage.

The same state structure is also used to hold information needed for the scalarization of array expressions.

5 Arrays

Modern computer systems employ a one dimensionsal memory space. Higher dimensioned arrays are transformed into this space by multiplying the index by the stride, or spacing, between consecutive elements of the corresponding dimension. These values are summed to obtain the offset of the element relative to the origin of the array. In g95 two pointers are used to manipulate array data. A pointer to the first element of data is required for memory management when allocating and freeing the array data. To access the array a biased base pointer is used. This pointer points to the location of element zero of the array. In this way the array can be accessed without needing to involve the lower bound of the array. It may be the case that element zero of the array does not exist. This does not matter, as it is only used as a base point for the offsets; no non-existing element of the array is ever referenced.

For fully contiguous arrays, where elements of the array are stored in consecutive memory locations, the stride of a dimension is equal to the size of all lower dimensions. This often speeds up access to the array as these values may be known at compile time.

The array descriptors used to pass actual arguments (what C calls "parameters") consist of a pointer to the first element of the array, the upper and lower bounds and the stride of each dimension. Array pointer variables are handled using the same structure. Array sections are accomodated by calculating the origin and strides to match the section, avoiding the need to make temporary copies of the data.

6 Scalarization

Array expressions introduce significantly complications. The first problem is that of scalarization. The Fortran language allows expressions involving operations on sections of arrays or whole arrays. In practical terms an operation on a whole array is simply a special case of an array section where the bounds of the section are the bounds of the array.

In order to evaluate array expressions it is neccessary to break them down into a set of scalar operations. This is done by generating loops, and using the implicit loop variables as indices into the array sections. The evaluation of array expressions involves several stages and two passes of the expression tree.

First the expression tree is traversed to identify which terms are scalar, and which are arrays. During this pass a list of subexpressions is constructed. Operators whose operands are all scalar result in a single scalar value. These subexpressions will be evaluated outside the scalarization loop, so the operands do not require individual processing. If an operator involves has an array valued result, its operands must be considered by the scalarizer.

The next task is to evaluate the bounds of the implicit loops. The array terms in the expression are examined, and one of these is used to determine the bounds of the scalarization loop. Constant bounds are picked by preference as this gives most potential possibilities for optimization. All the terms in an array expression must have the same shape, so the number of elements in each dimension can be determined from a single term.

For each array term an offset and stride relative to the implicit loop are evaluated. It is not neccessary to evaluate the upper bound of all the array sections, except for runtime error checking purposes.

The main body of the scalarization loop is generated using the same routines as are used for scalar expressions. The translation of the expression is performed in the same order as the initial walking, so only the next term in the list needs to be examined during the translation pass.

Operators which have not been marked as specific subexpressions are translated in the normal way after their operands have been processed. When a scalar subexpression is reached, the precalculated value is substituted.

When array expressions are reached, the implicit loop variables are used to index into the array to get a single scalar value. The offset and scaling factor calculated earlier are used to translate from the loop indices to individual array indices.

A naive implementation of this algoritm would require calculation of the offsets for all array indices on every access. However we traverse higher dimension array sections one dimension at a time. Within the inner scalarization loop the offset due to outer dimensions will be constant. We take advantage of this by calculating this offset before entering the inner scalarization loops.

7 Data Dependencies

The Fortran 95 standard specifies that all values on the right hand side of an assignment statement must be evaluated before any assignments take place. This is known as the "load-before-store" principle. In many cases this restriction has no impact as the source terms of the expression and the target variable are not related. However more care must be taken where both the source and target contain the same elements.

Where the source and target elements are not identically matched, the order in which the assignments are performed may effect the result. In some cases these data dependencies may be resolved by ensuring the assignments are performed in the correct order. In other cases an array temporary is required.

The behaviour of g95 in this area is currently quite simplistic. If any unmatched data dependencies are detected, or the expression is too complex to determine the exact dependencies, an array temporary will be used for the whole assignment. In this case two sets of scalarization loops are generated. The first evaluates the source expressions, and stores the result in a temporary array. The second copies the contents of the temporary array to the target array.

There are many optimization techniques that can be applied in order to reduce the size of the temporary required, and to improve memory access patterns within scalarized assignments. G95 currently only contains a partial implementation of the simpler of these.

8 Intrinsic Functions

Fortran includes many intrinsic functions for performing common mathematical and array operations, as well as operations on data which are impossible to implement using the Fortran language itself. Intrinsic functions and subroutines are implemented with a combination of inline code and runtime library calls.

Where inline code is required the expression state structure is used to hold the code to be execured in order to evaluate the expression.

Most of the required library functions have been implemented. However only the generic versions of there have been written. There is still significant scope for optimized versions to take advantage of simpler cases, processor specific features and more advanced algorithms.

9 IO Library

The IO library is currently one of the least complete parts of g95. Most of the infrastructure for the IO library is in place, as is parsing of format strings. However there is still a significant quantity of work required before this is completed. Formatted IO of integers is possible, however IO of real values is still limited.

10 Incomplete Features

The WHERE and FORALL constructs only work for simple cases where no data dependencies exist.

The WHERE construct performs masked array assignments. These are similar to normal array assignments except a third array expression is used as a mask. Only the assignments where the coresponding element of the mask array is true are preformed.

The FORALL construct allows assignments to be performed for all permutations of a set of

loop variables. This is equivalent to enclosing the assignment in multiple DO loops except that "load-before-store" semantics apply to the entire set of assignments. An array expression may be used to mask these assignments. The situation is further complicated by the ability to nest additional FORALL and WHERE constucts inside a FORALL block.

Arrays of character strings are not implemented. Some combinations of derived types and character strings are also incomplete.

Large array constructors used as variable initializers are not implemented. These typically contain large implicit DO loops. The simplest solution is to expand these loops at compile time as we do will small constructors. However this process would consume an unreasonably large amount of CPU time and memory. The solution is to initialize these variables at runtime.

11 Extensions

There are several extensions to the Fortran 95 standard which we would like to see included in g95. The first seven of these will included in the upcoming Fortran 200x standard.

1. Floating point exception handling
2. Allocatable arrays as structure components, dummy arguments, and function results.
3. Interoperability with the C programming language.
4. Parametrized data types.
5. Derived type I/O.
6. Asynchronous I/O.
7. Procedure variables.
8. OpenMP—provides multi-platform shared-memory parallel programming.
9. Cray pointers—provides functionality similar to C pointers.

12 Calling Conventions

The default behavior of g95 is to pass all actual arguments by reference. In many cases this is neccessary as procedures may be called via implicit interfaces. In this case the worst case calling convention must be assumed.

In some cases, eg. elemental procedures or procedures with assumed shape arguments, an explicit intarface must always be used. For these procedures optimizations such as passing INTENT(IN) parameters by value are possible. Although these optimizations are not currently preformed to simplify debugging, they are likley to be implemented in future revisions.

By default all array arguments are passed using an array descriptor. The advantage of this is that it allows discontiguous array section to be passed without requiring an array temporary. The disadvantage of is that such code will not be binary compatible with Fortran 77 code compiled by g77 or other Fortran compilers. To accomodate this, a compile time option is available to force g95 to use a g77 compatible calling convention. Procedures which use features which were not available in Fortran 77 (eg. POINTER arguments or assumed shape arrays) are still passed using the default calling convention.

While passing discontiguous arrays may reduce the overhead of a procedure call, it introduces a penalty every time the parameter is accessed. This is acceptable if only a small pro-

portion of the passed data is accessed. However if the passed array is heavily used it is beneficial to copy the array data into a contiguous array temporary and access it from there. If the array is INTENT(OUT) or INTENT(INOUT) it may also be neccessary to copy the modified data back to the original array.

The default behavior is to automatically add code to the start of a procedure to test for discontiguous arrays and repack them, as this matches the behaviour of most other Fortran compilers. Users are able to inhibit this behaviour when the cost of repacking the array is likley to exceed the increased cost of accessing the array. For cases where the shape of the array is not known at compile time the data is not repacked when the first dimension is contiguous, as this is unlikley to provide any performance gain.

13 Release dates

The tree-ssa branch of GCC is currently slated for mainline integration in GCC 3.5. The current release date for this, and hence the earliest realistic release date for g95, is late 2004.

G95 only generated its first piece of executable code in June 2002, and significant progress has been made since then. It is hoped that by Q4 2003 g95 will be functionaly complete and standards compliant.

We believe that all the major obstacles to inclusion in the GCC source tree have now been overcome. Inclusion in a non-release branch of GCC is expected in the very near future. It is expected that a seperate parallel development tree will still be maintained for the convenience of developers.

14 Acknowledgments

The g95 project was founded by Andy Vaught, without whom g95 would not exist. He also wrote a large portion of the code, braving the more esoteric aspects of fortran grammar and semantics.

Thanks should also be given to Steven Bosscher, Arnaud Desitter and everyone else who has contributed code, patches, ideas or even just support to the project. Also thanks to g77 maintainer Toon Moene for his assistance and support.