Refactoring and the Evolution of Fortran

Jeffrey L. Overbey   Stas Negara   Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave. MC 258
Urbana, IL 61801
{overbey2,snegara2,rjohnson}@illinois.edu

Abstract

Successful languages like Fortran keep changing and tend to become more complex, often containing older features that are rarely used. Complexity makes languages harder to use and makes it harder to build tools for them. A refactoring tool can eliminate use of these features from programs; this makes programs easier to understand and maintain, and it can simplify building certain programming tools. This is illustrated by using Photran, a refactoring tool for Fortran, to eliminate global variables from Fortran programs so that they can be used with Adaptive MPI, a version of MPI that performs load balancing.

1. Introduction

Just as software designs evolve, so do programming languages. The abstractions and models in a software system evolve as its requirements are better understood; similarly, the abstractions and constructs in a programming language evolve as the expressivity demands of its applications are better understood.

Fortran is a stellar example. FORTRAN I [5] emerged as the first successful high-level language during the late 1950s. Predating nearly every major development in computer science (and, in fact, predating the term "computer science" [15]), FORTRAN I now appears quaint, differing substantially from its modern incarnation, Fortran 2008, which reflects the benefits of half a century of accumulated experience. Since its inception, the Fortran language has been adapted to incorporate subprograms (FORTRAN 66), structured programming constructs (FORTRAN 77), modules and dynamic memory allocation (Fortran 90), object orientation and C language interoperability (Fortran 2003), and co-arrays (Fortran 2008).

In the Fortran world, backward compatibility is paramount, and this is evident in the way the language has evolved. Fortran 2008 is, for the most part, a superset of its predecessors, dating back to at least FORTRAN 77. The choice of language features to delete in any given revision of the ISO Fortran standard [11] is extremely conservative. Indeed, it is legal in Fortran 2003 to write an object-oriented program in a source format designed for 80-column punch cards. Not recommended, but legal.

Language evolution always comes at a cost. Introducing new features adds complexity to the language. Deleting features can obsolete existing programs. In the case of Fortran, the costs of evolution are largely attributable to its emphasis on backward compatibility. Continuously adding new features while retaining anachronistic alternatives has resulted in a complex language that is effectively splintered into several dialects. Any single piece of code will only use a subset of the Fortran language, depending on when the code was written and what experience the author has. Fortran applications written when FORTRAN 66 was current will use fixed source form and `goto` statements, while more recent applications will use free source form and `if` statements. FORTRAN 77 programmers use common blocks, while Fortran 90 programmers will use module variables. Among the latter, programmers with more training in software engineering will be more inclined to encapsulate these variables, making them `private`. It remains to be seen how the object-oriented facilities of Fortran 2003 will—or will not—be used.

This means that a Fortran program written in 1970 will use a different subset of the language than the same program written in 2010. This will only be exacerbated as the language continues to evolve. One can only imagine comparing Fortran programs from 1970 and 2070.[1]

[1] Arguably, Fortran should not exist in 2070. Perhaps it should not exist in 2009. Another language should be used instead. Nevertheless, we are operating under the assumption that the language will continue to exist, people will continue to use it, and they will insist on its evolution. Inertia, and substantial intellectual and economic investments, have a tremendous impact on a language's practical viability.

We believe the problem is *not* that the Fortran language is evolving, or that different users will use different subsets of the language, but rather that there is no strategy for deleting old language constructs. This results in a language that is increasingly large and complex not by design but by default, since old features must be retained and every new feature must co-exist with every old feature. The benefits of retaining outdated constructs are fairly obvious (all relating to backward compatibility), so let us consider the costs.

Arguably, the most important consideration is what impact this has on the day-to-day Fortran programmer. Many programmers can safely ignore most of the language, concentrating on a particular dialect that suits their purpose. For example, students in a numerical analysis course can program in free-format Fortran 90, using statically-allocated arrays, subprograms, and the simplest control flow constructs. But the story is quite different for programmers forced to maintain others' code: These programmers cannot confine themselves to their preferred subset of Fortran, but rather they must be fluent in whatever dialect the code's original author used. In many cases, the programmer needs to be aware of "old" and "new" ways to accomplish the same task. Ideally, he should understand *why* the new way is preferred. And, although it is not the job of the language or compiler to force good programming style, failing to make outdated constructs obsolete can leave a programmer blissfully unaware that he is using a construct for which a preferred, modern alternative exists.

A second consideration impacts Fortran programmers indirectly: Retaining old language features (and increasing the complexity of the language) makes Fortran programming tools—compilers, IDEs, static analysis tools, refactoring tools, performance analysis tools, debuggers, etc.—increasingly expensive to build. In turn, this limits the number of tools that will be made available to Fortran programmers. No company will build a tool unless it reasonably expects that it can recover its costs and eventually make a profit from it. Increasing the complexity of the language increases the time and cost of building tools, which lengthens the payback period—the time it takes to recoup the initial cost of creating the tool—and thus gives the tool a lower return on investment compared to other projects. The lower ROI, combined with the fact that Fortran is already a niche market, renders the tool a less desirable investment. So while a company that already produces a tool for Fortran 90 may be able to justify upgrading it to Fortran 2003 (since much of the complexity has been mitigated), it is far more difficult for a company to justify building a Fortran 2003 tool from scratch.

It appears that language evolution is a problem with no good solution. Failing to add new features will make the language stagnate. Adding new features without deleting old ones results in the complexity-related problems just discussed. And deleting old features will break backward compatibility.

But there is one strategy that has the potential to allow the language to evolve with fewer consequences. . .

2. Enter Refactoring

Refactoring [6] is the process of making substantive changes to source code that do not have a net effect on the program's observed behavior. For example, one might rename a variable or function, split a long subprogram into several smaller subprograms, or convert an array of structures to a structure of arrays.

The real benefit of refactoring comes from the fact that many common refactorings can be automated. Automated refactorings are included in many major IDEs, including Eclipse JDT, IntelliJ IDEA, Microsoft Visual Studio, and Apple Xcode, among others. Photran [14] provides the same for Fortran. In an automated refactoring tool, the user provides some input, the tool verifies that the refactoring can be applied, and finally the tool changes the user's source code. For example, to rename a function, the user would select the function to rename and provide a new name for the function; the tool would verify that the new name is legal and that a function does not already exist with that name, and finally it would change the user's source code to reflect the new name in the function declaration, `interface` declarations, and all call sites. Like most refactorings, this is a simple but tedious change to make manually.

Refactoring has traditionally been discussed in the context of object-oriented design, where it allows a system's design to be changed retroactively so that unforeseen changes can be incorporated without compromising the integrity of the design. According to common usage, however, refactoring is not limited to object-oriented languages, and it need not be limited to design-level concerns. In fact, we believe refactoring tools can serve a very different purpose.

Automated refactoring tools can replace many outdated language constructs with their modern equivalents. The widespread availability of such a tool could allow programming languages to deprecate features much more aggressively. If such a tool were robust, reliable, readily available, and reasonably fast, there would be much less impetus to retain outdated language features, assuming such a tool would allow older programs to be updated almost "for free."

3. From FORTRAN to Fortran

A refactoring tool has an internal program representation much like that of a compiler, except that it contains additional machinery for making changes to source code while preserving formatting, comments, and preprocessing directives (like `include` lines). This means that a refactoring tool can perform simple textual changes, but it can also perform much more complex changes requiring whole-program name binding analysis, control flow analysis, and so forth.

The following list is not exhaustive but certainly representative of the types of features that a refactoring tool could help eliminate from Fortran programs. This list of features is intentionally ambitious; our intent is to illustrate what is possible, not to argue that all of these changes *should* necessarily be made.

- *Eliminate fixed source form.* A Fortran refactoring tool contains all of the machinery needed to build a fixed-to-free form converter that maintains comments and formatting. Fixed form source is already an obsolescent feature in Fortran 2003 [11, §B.2.6], indicating the standardization committee's intent to delete it in a future revision of the standard. A tool to reliably convert fixed to free form will be essential for easing this transition on large code bases.

- *Reserve keywords.* Using keywords like `if` and `while` as variable names is generally considered poor practice, and failing to make these reserved words makes implementing Fortran parsers difficult. Renaming identifiers is a canonical example of refactoring; building a tool to identify such names and change their names to non-keywords could allow keywords to be reserved in a future revision of the standard.

- *Replace common blocks and block data subprograms with module variables.* In the simplest version of this transformation, each (named or unnamed) common block is replaced with a module containing a list of the same variables. The `common` statement (or `include` line) is removed and replaced with a `use` statement for the new module. `Block data` subprograms can be replaced with specification statements and initializers in the new module. Subsequent refactorings could be used to encapsulate these variables, if desired.

- *Require explicit `interface` blocks; eliminate `external` statements.* Fortran allows external subprograms to be declared using `interface` blocks, which specify the parameters and return type of the subprogram, or they may simply be listed by name in `external` declarations, in which case the parameters and return type, if any, are unknown. In the latter case, the compiler cannot verify anything about the subprogram call—not even that the number of parameters is correct—so this is also considered poor practice, as it can lead to cryptic runtime errors. If the external subprograms are written in Fortran, it is straightforward for a tool to generate `interface` blocks for them and replace `external` statements with these; if they are written in another language (e.g., C), the refactoring tool would either need to (1) parse the C code and attempt to generate equivalent `interface` blocks, (2) infer `interface` blocks from the call sites in the Fortran program, or (3) punt and require the user to manually code `interface` blocks. (Of these, we believe option (2) has the most potential.)

- *Require explicit variable declarations; eliminate `implicit` statements.* This is also straightforward and is a refactoring already available in Photran. An `implicit none` statement is added (potentially replacing an existing `implicit` statement), followed by explicit type declaration statements for all variables that were previously declared implicitly.

- *Remove other specification statements.* Fortran allows most variable attributes—`public`, `private`, `pointer`, `target`, `allocatable`, `intent`, `optional`, `save`, `dimension`, `parameter`, and many C language-binding attributes—to be included in a variable's type declaration statement, or they may be given in separate statements. Arguably, spreading a variable's declaration across several statements is poor practice, since a programmer must read the entire list of specification statements to determine all of the attributes assigned to a variable. Metcalf and Reid [13, p. 243] note this in the particular case of the `dimension` attribute: Omitting array dimension information from the type declaration statement makes it "look like a declaration of a scalar." Replacing these specification statements with equivalent clauses in a variable's type declaration statement (assuming `implicit none`) is straightforward.

- *Remove `entry` statements.* The `entry` statement allows several entrypoints to be declared within a single subprogram. The intent it to allow several procedures to share variables and/or code. A better practice is to create a module and make each entrypoint into a module procedure [13, p. 240]. We will not attempt to specify this refactoring in detail, although we note that it is a somewhat more complicated variant of *Extract Subprogram,* requiring an analysis of what variables and what code is shared among entrypoints, and potentially replacing `goto` statements with subprogram invocations.

- *Remove computed `goto`.* The computed `goto` is equivalent to a `case` construct [13, p. 288]. A refactoring tool can always substitute a `case` construct containing `goto` statements for a computed `goto`. However, it can also use a control flow analysis to determine if the statements branched to can be moved into the `case` construct, eliminating the `goto` statements entirely. More empirical work would be necessary to determine other idiomatic uses.

- *Remove arithmetic* `if`, largely similar to removing computed `goto`.
- *Remove* `character*`*n*. This is equivalent to `character(len=`*n*`)` and can be removed through a simple syntactic substitution.
- *Replace statement functions with internal functions.* This is a much simpler variant of the *Extract Method* refactoring.
- *Remove old-style* `do` *loops.* These are entirely equivalent to `do` constructs. If the loop is terminated with a `continue` statement, this statement can be replaced with `end do`; if it is terminated with another executable statement, the `end do` must be inserted after that statement. The statement label may be removed if it is not referenced elsewhere.

4. From Dialect to Dialect, Library to Library

The role of refactoring tools in language evolution is not necessarily limited to standardized, general-purpose programming languages.

Even if all obsolete constructs are eliminated, the Fortran language is still large; each application and each programmer will use a subset of the language, since there are often several ways to accomplish the same thing, and which way is preferable depends on the application and the programmer. For example, two matrices can be added using array notation or nested loops, arrays may be statically or dynamically allocated, subprograms may be internal or external, and modules are often interchangeable with singleton [7] objects.

The choice of which constructs to use (and which to omit) constitutes a dialect. Like the Fortran language itself, the dialect used for a particular application (or by a particular programmer) may evolve over time. Refactoring tools can often be used to translate between equivalent constructs, and thus can facilitate the evolution of an individual's dialect of Fortran.

Similarly, the objects, procedures, and constants provided by a framework or library comprise a different kind of "language." For example, `MPI_COMM_WORLD` and `MPI_SEND` are part of the "language" of MPI.[2] These languages tend to evolve, too; for example, Dig et al. [4] analyzed the evolution of several Java APIs, observing that 80% of the API changes could be expressed as (automated) refactorings.

[2]An API, like that of MPI, provides a vocabulary and has both a syntax (inherited from Fortran) and semantics, and therefore meets most common definitions of "language."

5. Case Study: AMPI

We have not attempted to design or implement a comprehensive refactoring tool for Fortran language evolution. However, we have built a tool with some of this functionality which is intended for the specific purpose of converting programs to run on Adaptive MPI (AMPI). In doing so, we have dealt with several specific instances of the aforementioned problems: Our tool automatically converts Fortran programs to a dialect absent of common blocks, `save` variables, and other global data. In this section, we will describe the unique requirements of AMPI, how our tool transforms Fortran programs to satisfy its requirements, and how these transformations fit into the more general subject of Fortran program evolution.

5.1. Overview of AMPI

Adaptive MPI [9] is an implementation of the Message Passing Interface (MPI) standard [12]; it currently implements all of MPI 1.1 and some of the MPI 2 standard. AMPI runs on the Charm++ Runtime System [2] and provides MPI programs with dynamic load balancing, virtualization, and checkpointing, among other features.

MPI programs usually assume that each MPI process will be a distinct operating system process—usually each will run on a separate processor of a multi-processor system—and thus there is no shared address space. However, to achieve dynamic load balancing, AMPI may map several MPI processes to different *threads* on a single processor, giving them a shared address space. This can lead to unexpected behavior, since different processes could be accessing the same copies of global variables, breaking the programmer's original intent.

The authors of AMPI propose two solutions to avoid problems associated with this sharing of global data [1]: *automatic globals swapping* and *manual global variables privatization.* Automatic globals swapping uses the operating system's facilities to swap one set of global variables for another on each thread context switch; it does not require any changes to the original MPI program but is available only on x86 and x86_64 platforms that fully support Executable and Linking Format (ELF). Manual global variables privatization is a more universal solution, but it requires changing the original MPI program. As our tool implements this procedure for Fortran programs, we will describe it in more detail.

5.2. Fortran Global Variables Privatization

Global variables are those variables that can be accessed by more than one subprogram (including several calls of the same subprogram) and are not passed as arguments of these subprograms. In Fortran 90 global variables are module variables, variables that appear in common blocks, and local variables that are declared with the `save` attribute—these retain their values between subprogram calls (like `static` variables in C).

Privatizing global variables means giving every process its own copy of these global variables. Again, this happens automatically in most MPI implementations since each MPI process is a separate operating system process, but AMPI requires that it be done manually. One way to do this is, essentially, to put all of the global variables into a large object (a derived type in Fortran, or `struct` in C), and then to pass this object around between subprograms. Each process can be given a different copy of this object. Figure 1 presents an example of privatizing the global variable `counter`, which is the only global variable in the original program. (According to the Fortran standard, the local variable `counter` is implicitly a `save` variable because its declaration includes an initializer.)

A more detailed description of the global variables privatization procedure implemented by our tool is as follows. First, a new derived type is declared in a new module. This derived type contains a component for every global variable in the program. A statement to dynamically allocate an object of this type is inserted after the call to `MPI_Init`, giving each process its own instance of the object. A pointer to this type is passed as an argument to every subprogram. Throughout the program, every access to a global variable is replaced with an access to the corresponding field of the derived type. Finally, the declarations of global variables are removed from the program.

We implemented global variables privatization for Fortran using the refactoring infrastructure in Photran, an Eclipse-based IDE for Fortran [14]. Although the tool is intended to be used as a preprocessor immediately before compilation (so the programmer never sees the privatized version of the program), it is also accessible as a refactoring within the IDE.[3] The privatization procedure proceeds in four passes:

1. Stubs are generated for the derived type and the module that contains this type. Their names should not conflict or shadow names of other entities in the program.

2. Subprograms are processed. An extra parameter is added to each subprogram and each call site within its body. Components for `save` variables are inserted into the derived type, accessed to these variables are replaced with accesses to the derived type component, and finally the `save` variables are deleted from the subprogram.

3. Module variables are eliminated in a manner similar to `save` local variables.

4. Finally, common blocks are eliminated similarly.

These code transformations are composed from many smaller pieces, which we have implemented as refactorings in Photran. Among these refactorings are eliminating common blocks, moving "saved" local variables out of a subprogram, transforming public variables of a module into the corresponding fields of a derived type, and adding a new parameter to a subprogram.

5.3. Returning to Language Evolution

The global variables privatization procedure corrects the behavior of programs that would otherwise execute "incorrectly" on AMPI by converting them to a dialect of Fortran that does execute correctly: One which forgoes common blocks, `save` variables, and module variables in favor of components in a derived type.

As we noted earlier, eliminating common blocks is one example of a refactoring that eliminates uses of an obsolete language construct. Similarly, replacing module variables with derived type components is one step in replacing modules with objects. Modifying procedures to receive and pass an additional parameter of a derived type is one step in converting procedural Fortran to object-oriented Fortran; in the latter case, the extra parameter is the *self* object.

6. Reflection

Our experiences implementing refactorings in Photran, including those for AMPI, support our belief that refactoring is a viable technical strategy for language evolution in many cases. Nevertheless, there are several potential obstacles.

First, some language features simply cannot be eliminated automatically, or automatic elimination gives a less than optimal result. For example, as noted in §3, computed `goto`s can be replaced with `case` constructs. Although this eliminates the construct, the resulting code is more verbose and no more readable than the original.

[3] On a technical note, it might seem that global variables privatization is not a refactoring, because the original program (before privatizaton) and the transformed program (after privatization) behave differently on AMPI. However, this is because the AMPI runtime changes the semantics of global variables. The original and transformed programs have the same semantics relative to the Fortran language and relative to any ordinary implementation of MPI, so we believe the transformation is a refactoring.

```
PROGRAM MyProgram
  include 'mpif.h'
  INTEGER :: ierr
  CALL MPI_Init(ierr)
  CALL count_calls
  CALL count_calls
  CALL MPI_Finalize(ierr)
END PROGRAM MyProgram

SUBROUTINE count_calls
  INTEGER :: counter = 0
  counter = counter + 1
  print *, 'I was called ', counter, ' times.'
END SUBROUTINE count_calls
```

```
MODULE GeneratedModule
  TYPE GeneratedType
    INTEGER :: counter = 0
  END TYPE GeneratedType
END MODULE GeneratedModule

SUBROUTINE MPI_Main
  USE GeneratedModule
  include 'mpif.h'
  INTEGER :: ierr
  TYPE(GeneratedType), POINTER :: var
  CALL MPI_Init(ierr)
  ALLOCATE(var)
  CALL count_calls(var)
  CALL count_calls(var)
  CALL MPI_Finalize(ierr)
END SUBROUTINE MPI_Main

SUBROUTINE count_calls (var)
  USE GeneratedModule
  TYPE(GeneratedType) :: var
  var%counter = var%counter + 1
  print *, 'I was called ', var%counter, ' times.'
END SUBROUTINE count_calls
```

Figure 1. Example of the code transformation that privatizes the "saved" local variable `counter` of the subroutine `count_calls`. The original code of an MPI program is on the left; the refactored code, which can be executed on AMPI, is shown on the right.

Turning it into readable code would require a more advanced analysis, and it would likely require heuristics on common usage patterns. Similarly, converting procedural programs to object-oriented programs requires creativity and domain knowledge, so one should not expect a "make program object-oriented" refactoring.[4] In both of these cases, the old and new constructs represent different *styles* of programming—unstructured vs. structured, procedural vs. object-oriented—and the inadequacy of tools is due to the lack of a 1–1 correspondence between old and new constructs.

Second, in the specific case of Fortran, widespread use of language extensions and preprocessors may preclude automated source code transformation. Vendor-specific language extensions are problematic because a tool cannot reliably transform a programs containing constructs it does not "understand." Refactoring a program containing C preprocessor directives is complicated but tractable [8]; unfortunately, Fortran programmers do not just use the C preprocessor. Some use M4. Some use M5. IBEAM [10] uses a custom preprocessor written in Python to concatenate modules together, a makeshift attempt at inheritance. Plenty of makefiles use *sed*. Again, a refactoring tool cannot "understand" the parts of the program that are preprocessed into something else before the compiler sees them; there is no guarantee that it will analyze and transform them correctly unless it is specifically programmed to do so.

Although some empirical study is necessary, we believe that these are surmountable problems. The biggest obstacle to language evolution through refactoring is cultural. It has not been attempted before, and the Fortran community has often been resistant to ideas not already proven in other circles. Only recently have refactoring tools been adopted widely in the Java and C# communities; many programmers still have a difficult time trusting a tool to rewrite parts of their source code. And perhaps they should; these tools are often quite buggy [3]. This is not a problem if the program contains an extensive test suite (as many Java and C# programs do); unfortunately, this is not the case for many Fortran programs. Finally, many Fortran programs are in government labs, running highly classified simulations in strict security environments under heavy bureaucracies.

These social and cultural obstacles are also surmountable, but they will require the momentum of a community. Gaining community interest will require a robust, capable tool. Building this tool will require a significant effort, but we believe it has great promise. We hope that others in the Fortran community will feel the same.

This work was supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and by the National Science Foundation under award #0725070. The authors would like to thank Nicholas Chen, Isaac Dooley, Brian Foote, and Daniel Overbey for discussions that contributed to this paper.

[4]However, many of the *steps* in making a program object-oriented are algorithmic and are excellent candidates for implementation as refactorings.

References

[1] Adaptive MPI Manual. http://charm.cs.uiuc.edu/manuals/html/ampi/manual.html.

[2] The Charm++ Runtime System. http://charm.cs.uiuc.edu/tutorial/CharmRuntimeSystem.htm.

[3] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[4] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[5] Fortran Automatic Coding System for the IBM 704: Programmer's Reference Manual. http://www.fortran.com/FortranForTheIBM704.pdf.

[6] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, January 1995.

[8] A. Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, Champaign, IL, USA, 2005. Advisor: Ralph Johnson.

[9] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[10] IBEAM. http://www.ibeam.org.

[11] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 1539-1:2004: International standard: information technology, programming languages, Fortran*. Fourth edition, 2004.

[12] Message Passing Interface Forum. http://www.mpi-forum.org/.

[13] M. Metcalf and J. Reid. *Fortran 90/95 Explained*. 1999.

[14] Photran - An Integrated Development Environment for Fortran. http://www.eclipse.org/photran/.

[15] Professor Forsythe. http://infolab.stanford.edu/pub/voy/museum/pictures/display/floor1.htm.