Changing the Face of High-Performance Fortran Code

A White Paper

Ralph Johnson, Brian Foote, Jeffrey Overbey and Spiros Xanthos
Department of Computer Science
University of Illinois
Urbana, IL 61801
{johnson, foote, overbey2, xanthos2}@cs.uiuc.edu

Abstract

There is a huge amount of Fortran code that is too valuable to throw away, but very expensive to maintain. Refactoring tools have had a great impact on the productivity of object-oriented developers and the quality of code. However, high-performance languages such as C and Fortran do not have these tools. Moreover, refactorings specific to high-performance and parallel computing have not yet been adequately examined. This paper describes Photran, an integrated development environment for Fortran that will support automated refactoring and how a tool like this can be used to reduce the cost of maintenance and development of systems implemented in Fortran.

1 Introduction

Fortran is over fifty years old. Though there probably are not any Fortran programs in use that are fifty years old, there are some that are older than the programmers who maintain them. Fortran remains very popular for scientific programming, and is the backbone of the HPC community. In the weapons simulation community, typical applications undergo 10-35 [12] years of continuous development.

Fortran has changed radically over the years, and programming practices have changed, too. So, many of these older programs seem poorly designed to modern programmers.

As programs are changed to meet changing requirements, even well-designed systems are subject to structural erosion. Programs are changed to port them to new computers, to use new algorithms, to use new libraries, and to make them be components in larger systems. The cumulative effect of these changes is to damage the system's architecture [2]. They increase the complexity of the system, cause code duplication and global information sharing [5]. Maintenance and expansion can become tedious, costly, and

time-consuming [1]. One way to combat this structural erosion and to improve the design of a system is to refactor it. This paper will describe how tools can help to refactor a system.

A refactoring is a *behavior preserving program transformation*[9, 4]. *Behavior preserving* means that it is a change in the internal structure of the program that does not affect the way the program works. Refactorings are important during maintenance because they are used to improve modularity, to reduce duplication, and to reduce coupling and information sharing.

The key to successfully refactoring a system is to take small steps, to separate refactorings from changes to the behavior, and to have and continually improve an automated test suite. Big steps usually cause defects, but small steps can usually be taken safely. Since refactorings do not change the behavior of the system, a test that doesn't pass shows that a refactoring was not implemented correctly. Most refactoring errors can be caught by an automated test suite.

The main reason that programmers do not improve the structure of their system is that it takes too long and it is likely to introduce errors. Refactoring tools not only reduce the time to change a system, they reduce the chance of introducing errors. A big advantage of using a refactoring tool over a regular text editor is that the refactoring tool can guarantee that the refactorings do not break the code. It is possible to prove that the refactorings do what they are supposed to do. and do not break anything else in the process. This takes a tremendous burden off the programmer.

Refactoring tools are popular for object-oriented languages like Java and Smalltalk because they allow programmers to improve the structure of a system and keep it from eroding. We believe that refactoring tools would have the same effect on HPC languages.

Refactoring tools differ from traditional reengineering tools in that refactorings are fine-grained, incremental modifications that are made under programmer control. Traditional program tranformation systems, by contrast, attempt to make large, wholesale changes to programs automatically, in batch mode. Incremental refactoring leaves design decisions in the hands of the programmer. These decisions are difficult to automate. Refactoring tools eliminate tedious clerical work and let the programmer concentrate on creative work such as discovering abstractions and ensuring that modules hide design decisions.

2 Reasons To Refactor

One reason to refactor code is to make it easier to read, and easier to comprehend, easier to understand. For example, one way to help code to more accurately reflect its underlying design is to change the names of variables, subroutines, functions and modules. Other ways are to break a large subroutine into smaller ones, and to break a

large module into smaller ones. Putting related elements together can make the system easier to read in one pass. Eliminating duplication makes the system smaller.

Another reason to refactor is to make code more reusable. For example, a refactoring tool can replace manifest constants with symbolic names or variables. It can allow a programmer to create a new module and to move subroutines into it, changing the modules that call the subroutines to make sure they call the right version. It can substitute statically allocated arrays with dynamically allocated variables.

A third reason to refactor is to modernize code. Fortran, has experienced several major reincarnations. In the seventies and eighties, the GOTO-laden spaghetti code of the Fortran IV era gave way to more disciplined control structures and longer variables names. During the nineties, Fortran added derived types, dynamically allocated data structure, a more modern, type-safe module facility, generic procedure and operators, and support for parallel programming. The latest Fortran standard has added type-bound procedure support for object-oriented polymorphism, and mixed language support. Although some older codes have been updated, many have not, because updating them is costly and error-prone. Thus, some codes are still mired in the sixties, others in the seventies or eighties. A Fortran shop is often a museum of obsolete programming techniques. There are conversion tools for obsolete language features, but they usually don't fix the bad style caused by using these features. A refactoring tool can change the design of the code.

A fourth reason to refactor is to improve the code's structure and quality, thereby setting the stage for additional enhancements and improvements, such as adding new functionality, or integrating new components. The reason that a feature is hard to add is usually because it requires changes to many parts of the code. If the code is first refactored to put all the parts that need to be changed in one module then the change will become much easier. Refactoring can eliminate duplication and can divide the code into layers. This will help ensure that each design decision is encapsulated.

A fifth reason to refactor is to extract useful functionality from moribund or obsolete code. Rewriting existing applications can lead to disaster, since existing code is often the sole repository of hard-won institutional wisdom and irreplaceable domain knowledge. [14] Refactoring tools can make it much easier to extract the useful pieces and reuse them instead of starting over. This can aid in the extraction of reusable components, and their subsequent incorporation into component frameworks.

For instance, when refactoring our *IBEAM* code, we needed to extract code for the *PARAMESH* adaptive mesh package that we'd been maintaining along with our code so that it could be linked as a stand-alone library. This required creating a derived type for a lengthy list of new runtime parameters, and then required that dozens of array declarations be changed from static fixed size form to runtime dynamic allocations. These manipulations were exactly the sort of dispersed, intricate, cumbersome, and error prone manipulations that refactoring tools automate.

Another motivation for refactoring is to make the code more portable, or to move it to new hardware. Since successful high performance applications will survive several generations of hardware, they will eventually need to migrate to newer platforms. One way to handle this is to first refactor the code so that the parts which must change are encapsulated in a new layer, or facade. One can then build a new implementation of this facade to support the new, target platform, while retaining the support for the current platform in the existing implementation. This technique does not work as well for high-performance software as for other kinds of software, because a large part of porting a high-performance program to another platform is to optimize it for that platform, and optimizations can't be put in its own layer. Optimizations will be covered later.

3 Fortran Refactoring

The classical use of refactoring as a form of retroactive engineering is particularly applicable to Fortran. The amount of Fortran 77 code that remains in production is a clear indication that software maintenance issues apply equally to HPC.

In her M.S. thesis [3], Vaishali De identifies many possible Fortran refactorings. Many of the standard refactorings for fields and methods (described in [4]) apply equally to variables and subroutines in Fortran, such as Extract Method (i.e., removing a section of code into its own subroutine), Decompose Conditional (replacing a complex boolean expression with a more descriptive function call), Rename Variable, and Reorder Procedure Arguments. Some refactorings typically applied to classes in an object-oriented system can be applied equally well to Fortran modules, such as Encapsulate Field (where accesses of and assignments to a variable are replaced with function calls) and Move Method (moving a subroutine between modules).

Fortran also requires several unique refactorings. For example, migrating code from Fortran 77 to Fortran 90/95 requires transforming code from fixed format to free format and classic DO loops to modern DO/While. It is often a good idea to also convert COMMON to modules, to make IMPLICITELY declared variables explicit, and to migrate parallel arrays to derived types. Other changes to Fortran arrays are to change an assumed size array to an assumed shape array and to reorder array dimension. Converting a fixed offset array reference to a derived type field is a possible refactoring in other languages but seems especially useful in Fortran. Similarly, once Fortran 2003 compilers materialize, existing Fortran code will need to be migrated to that.

4 Performance Refactoring

Most HPC codes are optimized for a particular architecture. Despite the best efforts of compiler vendors, code intended to run on a specific supercomputer needs to be hand-

optimized to achieve the best performance. Thus, programmers perform *performance refactorings* such as manual unrolling of loops and optimizing data structures based on the machine's cache size. Unlike the other refactorings, performance refactorings are rarely performed on non-HPC code because they usually decrease readability. Applying performance refactorings by hand is tedious and error-prone, and it would useful to automate them, as well.

Performance refactoring is important when moving a code to a new machine. When a code is moved from one machine to another, the optimizations for the first machine must be undone and then the code optimized for the second machine.

Performance refactorings are machine specific and usually make code less readable, so they ought to be treated differently from other refactorings. Rather than transforming the code in-place, a performance refactoring should be a deferred transformations. Programmers will tag a section of code to indicate that a given transformation (e.g., *unroll this loop five times*) should be applied immediately before the code is compiled. This would allow programmers to maintain the more readable version of their code while compiling a performance-optimized version.

The performance refactorings will be labeled with the machine that they are for. This will make it easier to maintain a single code to run on multiple machines. The portable version of the code will be the version without any performance refactorings, and the version for a particular machine can be obtained by applying the performance refactorings for that machine.

5 Current Status of Refactoring Tools

Research on refactoring started at UIUC in the late 80s[10, 9], and at first focused on understanding how C++ programs evolved. Then we focused on building tools to automate refactoring. The Smalltalk Refactoring Browser was first released in 1994, with many iterations to follow[13]. Then we looked at the problem of refactoring C programs and the problems caused by the C preprocessor, resulting in a tool (CRefactory) that can refactor C programs regardless of how the CPP is used[6, 7, 8]. The Smalltalk Refactoring Browser has become a standard tool that is currently supported by several of the Smalltalk vendors, while the CRefactory is a research prototype in the process of being rewritten to be more reliable.

The publication of Fowler's *Refactoring* book[4] created interest in refactoring. Refactoring has come to be considered a key part of agile methodologies. There are now a number of refactoring tools for Java. Martin Fowler's list of refactoring tools (http://refactoring.com/tools.html) includes eleven Java tools, five for C#, three each for Visual Basic and C and C++, and one for Python, Self, Delphi, and, of course, Smalltalk. Some of these are only research projects, while others are reliable and fully

supported tools.

The Java tool that we use is Eclipse, which was created by IBM. Eclipse is not just a programming environment for Java, it is an open source framework for integrated development environments that is implemented in Java. It is based on a "plugin" architecture that makes it easily adapted to new uses. One of the plugins that comes with Eclipse is the CDT, which is a programming environment for C and C++. However, unlike the Java environment, the CDT does not provide much refactoring support. Refactoring support is not a basic part of Eclipse but must be provided by each programming environment that is built upon it, because refactoring is language specific.

There are currently no refactoring tools for Fortran. So, we created the Photran project. Photran is an Eclipse plugin that provides a Fortran IDE. It currently is similar to the CDT, i.e., it provides minimal refactorings. Unlike the CDT, it was designed to support refactoring, and we are in the process of adding refactoring support to it.

6 Photran

Photran [11] is an Eclipse-based Fortran IDE being developed at the University of Illinois. Photran is an official Eclipse Tools Project, currently provides a basic IDE for Fortran, and has an active user community. As of July 2005, it has been downloaded ten thousand times, though there are probably only a few hundred people who use it regularly. It is the only open-source IDE for Fortran 90, and it also supports Fortran 77. We plan to turn Photran into a refactoring tool for Fortran.

There are many aspects of Photran that can be improved. For example, users want better debugging support and better support for Fortran 77. However, most of the improvements we plan to make are related to refactoring. We are in the process of implementing the kinds of refactorings that other refactoring tools implement, and which we have implemented for other languages. Then we plan to implement refactorings that will be useful in converting from Fortran 77 to Fortran 90, and from Fortran 90 to Fortran 2003. Finally, we plan to implement performance refactorings. This will include the ability to convert a code that has been optimized to a particular machine into a portable code with machine-specific performance refactorings, as well as the ability to introduce performance refactorings into a portable code.

Although refactoring is always performed under the direction of a programmer, we plan to create *code critics* that analyze a code and give advice on ways to improve it. There can be critics that try to make code more portable, and others that try to make it more efficient on a particular machine. Critics can try to make code more modular, to reduce duplication, or make it meet a coding standard. Different critics will give different advice, and their advice will often conflict. But they will help programmers spot ways of improving their code and will be especially helpful to novice programmers.

One of the reasons we first implemented a basic IDE was to obtain a user community. We plan to listen carefully to this community to discover which refactorings are most important to them. We know which ones are most important to Java or Smalltalk programmers, but they will not necessarily be the most important for Fortran. When we built the Smalltalk Refactoring Browser, we got a lot of good ideas from our users, and we expect that to happen for Photran, as well.

Photran is an open source project, and we expect that our users will contribute to it. In fact, our fixed-format parser was developed by one of the Photran users. We would like to make this community as open as possible, which means that we have to be active on mailing lists, go to conferences where our users go, run workshops where we can meet with them, listen to their concerns, and take their code. This is an effort that does not immediately contribute towards publishing papers, but in the long run it will make our tool fit the needs of those who use it, which will make our research be adapted.

One of the big issues in making Photran a full-featured refactoring system is creating a program model that can handle all the Fortran language variants. Some of these variants are standard (Fortran 4X, Fortran 77, Fortran 90, fortran 2003) and some are vender-specific extensions. Some are even project-specific extensions, because it is common for projects to use a preprocessor to extend Fortran. This is often the C preprocessor, but is often custom-built. We know how to build refactoring tools for any language variant, and one solution is to bulld a tool for each variant, but there are dozens of well-known variants and so it is probably better to make a refactoring tool that can be easily extended to handle extensions in the language.

Photran is both a research and a development project. We know how to implement basic refactorings and how to grow a user community, but it takes time to do it. We are not sure which refactorings are most important to Fortran programmers, how an environment for performance refactoring should work, or a good program model to support a large number of language variations. These are the research parts of Photran. The research and development are intimately connected, however, because we need feedback from users to learn which refactorings are most important and to make sure that the refactoring environment is easy to use and works on a wide range of projects. Good development practices are needed to get and keep those users.

Photran was seeded with support from IBM several years ago, and is currently being supported through IBM under the PERCS/HPCS project [11]. We are looking for larger and longer-term support so that we can make faster progress.

7 Conclusion

Refactoring tools might be even more important for Fortran than they are for Java. One reasons is that there is more legacy code in Fortran than in Java, and much of it cries out

for modernization. Another reason it that systems are often built by scientists who then bring in a programmer to help maintain the system. The programmers often see things that ought to be done better, but it is difficult to make those changes without refactoring tools. Moreover, nobody has yet looked at refactorings specific to high-performance, parallel computing.

Scientific codes can be exceptionally long lived and are often the subjects of decades of incremental development and refinement. Even the most skilled programmers often find that the structure of these codes erodes under the impact of wave after wave of changing requirements. As these codes become more intricate and complex, and as programmers leave the project and are replaced by new ones, the codes become increasingly difficult to change. Programmers tend to avoid even conceptually simple changes, such as name changes, since any changes carry the risk of introducing defects.

Refactoring technology is ideally suited to fixing these problems, and allowing developers to once again be confident in addressing changing requirements.

We believe that refactoring tools have the potential to make a dramatic difference in how scientific code is able to evolve and keep pace with change. We base this belief in over a decade of experience with these tools, on the importance of refactoring in setting the stage for agile development in the Java world, and on the belief that the greater complexity of languages such as Fortran will lead to even larger productivity gains through refactoring than seen in Smalltalk or Java.

Our aim is to provide Fortran programmers with a state-of-the-art tool that can increase productivity and allow them to adapt their code to changing requirements and modern software engineering practices. We want to make Photran and refactoring a key part of the life of a Fortran programmer. It should help them make their codes more portable, more reusable, easier to maintain, and easier to change. This will decrease the cost of HPC and increase the productivity of HPC programmers.

References

[1] Barry Boehm and Ellis Horowitz (editors). *The High Cost of Software: Practical Strategies for Developing Large Software Systems.* Addison-Wesley, 1975.

[2] Fred Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[3] Vaishali De. A foundation for refactoring fortran 90 in eclipse, ms thesis. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[4] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[5] Brian Foote and Joseph W. Yoder. Big ball of mud. In *Pattern Languages of Program Design 4, Neil Harrison, Hans Rohnert, and Brian Foote, editors*. Morgan Kaufmann, 2000.

[6] A. Garrido and R. Johnson. Challenges of refactoring c programs. In *Proceedings of IWPSE 2002: International Workshop on Principles of Software Evolution*, Orlando, Florida, May 19-20 2002. Morgan Kaufmann.

[7] A. Garrido and R. Johnson. Analyzing multiple configurations of a c program. In *Proceedings of the 21st International Conference on Software Maintenance*, September 2005.

[8] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[9] William Opdyke. *Refactoring Object-Oriented Frameworks, PhD Thesis*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[10] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *"Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[11] Jeff Overbey, Spiros Xanthos, Ralph Johnson, and Brian Foote. Photran: An eclipse plug-in for fortran development, http://www.eclipse.org/photran, 2005.

[12] D. E. Post and R. P. Kendall. Software project manafement and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from asci. *The International Journal of High Performance Computing Applications*, 18, 2004.

[13] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems 3(4)*, 1997.

[14] Joel Spolsky. Things you should never do. *http://www.joelonsoftware.com*, 2000.