A Catalog and Classification of Fortran Refactorings

Mariano Méndez[1], Jeffrey Overbey[2], Alejandra Garrido[1,⋆], Fernando G. Tinetti[1,⋆⋆], and Ralph Johnson[2]

[1] Fac. de Informática, Universidad Nacional de La Plata, 50 y 120, La Plata, Buenos Aires, Argentina
[2] Dept. of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, Illinois, USA

Abstract. This paper presents a catalog of code refactorings that are intended to improve different quality attributes of Fortran programs. We classify the refactorings according to their purpose, that is, the internal or external quality attribute(s) that each refactoring targets to improve. We sketch the implementation of one refactoring in Photran, a refactoring tool for Fortran.

Key words: Refactoring. Fortran Programming

1 Introduction

In 1956, the first draft of The IBM Mathematical Formula Translating System was finished [2]. This first version of Fortran (called FORTRAN I) was the start of a complex evolutionary process. This process led to many different versions: FORTRAN 66, FORTRAN 77, Fortran 90/95, Fortran 2003, and Fortran 2008. For evolution to be practical, backward compatibility with older versions of the language was essential [9].

Over so many years of evolution, program maintenance becomes challenging. The magnitude of maintenance tasks is increased not only by the evolution of versions, but the large amount of Fortran code in production and the importance of Fortran as a programming language in several disciplines such as meteorology, physics and mathematics.

Refactoring is a technique used to improve internal qualities of the code like readability, flexibility, understandability and maintainability [5]. It is applied interactively on code with "bad smells" like duplication and lack of parameters [5], and after a series of small transformations, it beautifies the code preserving its behavior. In the case of Fortran, refactoring can make substantial improvements to readability and maintainability, and it can also modernize the code by replacing obsolete constructs with newer alternatives. [9].

⋆ also LIFIA and CONICET Argentina
⋆⋆ also LIDI and Comisión de Investigaciones Científicas de la Prov. de Bs. As.

The real value of refactoring comes from its automation and its integration in development environments. The first refactoring tool was built for the Smalltalk language, but it was not until it was integrated into the Smalltalk browser that it became successful [10]. Nowadays there are several refactoring tools for Java like the one integrated in Eclipse JDT [4] and Idea [6].

2 Characteristics of Fortran Programs

As a programming language, Fortran is one of the most ancient, yet it is still being used. Its evolution has resulted in a wide range of equivalent syntactical constructions. From those equivalent constructions, the older ones (coming from old language version/s) have many disadvantages/drawbacks.

However, not all Fortran code is legacy code. Fortran has gained a leading role in the High Performance Computing world throughout the years. Currently, old Fortran programs need to be made more efficient in multiprocessing systems with multi-core architectures [12]. Furthermore, multi-core processors are making single threaded (or, directly, sequential) software obsolete, such as most of the legacy Fortran programs.

3 A Catalog of Fortran Refactorings

This section presents a catalog of refactorings for Fortran code. We have found two categories of Fortran refactorings: *Refactorings to Improve Maintainability* and *Refactorings to Improve Performance*. Each one of these classes may be divided into subclasses. This categorization is not the only possible one. Many classical refactorings have been intentionally omitted from this list since they are widely described in the literature [7, 5], although they fit into this categorization as well.

3.1 Refactorings to Improve Maintainability

The refactorings in this category are intended to improve internal quality attributes of the code such as: readability, understandability and extensibility (attributes that refactoring has been recognized to improve) and also refactorings that allow upgrading the code to newer versions of Fortran, removing obsolete features.

- **Refactorings to Improve Presentation/Readability :**
  - ***Rename:*** change the name of a variable, subprogram, etc.
  - ***Change Keyword Case:*** change the case of keywords in the source code.
  - ***Extract Local Variable:*** remove a subexpression from a larger expression and assign it to a local variable.
  - ***Extract Internal Procedure:*** remove a sequence of statements from a procedure, place them into a new subroutine, and replace the original statements with a call to that subroutine.
  - ***Canonicalize Keyword Capitalization:*** make all applicable keywords the same case throughout the selected Fortran program files.
- **Refactorings to Facilitate Design/Interface Changes :**
  - ***Encapsulate Variable:*** create getter and setter methods for the selected variable.

- ***Make* Private *Entity* Public*:*** switch a module variable or subprogram from Private to Public visibility.
- ***Change Subprogram Signature:*** allow the user to add, remove, reorder, rename, or change the types of the parameters of a function or subroutine, updating call sites accordingly.
- ***Add* Only *Clause To* Use *Statements:*** create a list of the symbols that are being used from a module, and adds it to the Use statement.
- ***Move Entity Between Modules:*** move a module variable or procedure from one module to another and adjust Use statements accordingly.

– **Refactorings to Avoid Poor Fortran Coding Practices:**
  - ***Remove Unreferenced Labels:*** delete a label if it is never referenced.
  - ***Remove Real Type Iteration Index:*** change non-integer Do parameters or control variables.
  - ***Remove Reserved Words As Variables:*** rename variables named equal to Fortran reserved keywords.
  - ***Introduce* Implicit None*:*** add Implicit None statements to a file and add explicit declarations for all variables that were previously declared implicitly.
  - ***Introduce Intent* In/Out*:*** introduce intent In or Out in each variable declaration within functions and subroutines.
  - ***Remove Unused Local Variables:*** remove declarations of local variables that are never used.
  - ***Minimize* Only *List:*** delete symbols that are not being used from the Only list in a Use statement.
  - ***Make* Common *Variable Names Consistent:*** give variables the same names in all definitions of the Common block.
  - ***Delete Unused* Common *Block Variable:*** remove unused variables declared in a Common Block.
  - ***Add* Dimension *Statement:*** add the Dimension statement to declare an array.
  - ***Remove* Format *Statement Labels:*** replace the format code in the read/write statement directly, instead of specifying the format code in a separate format statement.

– **Refactorings to Remove Outdated, Obsolete and Non-Standard Constructs:**
  - ***Replace Obsolete Operators:*** replace all uses of old-style comparison operators (such as .LT. and .EQ.) with their newer equivalents (symbols such as ¡ and ==).
  - ***Change Fixed Form To Free Form:*** change Fortran fixed format files to Fortran free format files.
  - ***Transform* Character* *to* Character(Len =) *declaration:*** replace Character* with the equivalent Character(Len =) for string declaration.
  - ***Remove Computed* Go To *statement:*** replace a computed Go To statement with an equivalent Select-Case construct containing Go To or if possible remove the Go Tos statement entirely.
  - ***Remove Arithmetic* If *Statement:*** replace an old arithmetic If statement, being analogous to removing computed Go To.
  - ***Remove Assigned* Go To*s:*** remove assigned Go To statements.
  - ***Replace Old Styles DO loops:*** replace old styles Do Loop Continue with the equivalent Do Loop with End Do statement.
  - ***Replace Shared* Do *Loop Termination:*** replace all shared Do Loop termination construct with the equivalent Do Loop with End Do statement.
  - ***Transform To* While *Sentence:*** remove simulated While made by If and Go To statement.
  - ***Move* Common *Block to Module:*** remove all declarations of a particular Common block, moving its variable declarations into a module and introducing Use statements as necessary.
  - ***Move* Saved *Variables To* Common *Block:*** create a Common block for all saved variables of a subprogram.
  - ***Convert* Data *To* Parameter*:*** change a Data declaration to Parameter declaration making more clear which variables are constant and which ones are not.

3.2 Performance Refactorings

This category currently has two examples of how refactoring can be used to improve performance while preserving not only the behavior of the program but also the readability and maintainability of the code. This is one of the factors that sets refactoring apart from optimization.

– **Refactorings For Performance**
  - ***Change To Vector Form:*** rewrite a Do Loop into an equivalent Fortran vectorial notation, which allows the compiler to make better optimizations [12].
  - ***Interchange Loops:*** swap inner and outer loops of the selected nested do-loop, in the case that doing so optimizes memory access pattern and allows to take advantage of data prefetching techniques.

4 Photran: A Refactoring Tool for Fortran

Photran is an advanced, multiplatform integrated development environment (IDE) for Fortran based on Eclipse. From the beginning, Photran was designed to support refactoring, and much of its development effort has focused on providing a robust refactoring infrastructure. Version 6.0 (released June, 2010) contains 16 refactorings, and many more are under development. The development version of Photran provides name binding, control flow, and basic data flow information to support precondition checking.

4.1 Building New Refactorings

Photran divides refactorings into two categories: An *editor-based refactoring*, which requires the user to select part of a Fortran program in a text editor in order to initiate the refactoring, and a *resource refactoring* which applies to entire files.

A refactoring generally consists of three parts. First, it expects certain *inputs* from the user. Often, this means the user must select a particular Fortran construct in the source code in order to activate the refactoring, although some refactorings will pop up a dialog box asking for additional information. Second, it checks a set of *preconditions.* These validate the user input and perform any static analyses necessary to guarantee that the refactoring will preserve the program's behavior. If all of the preconditions pass, the refactoring *transforms* the source code.

4.2 Example: Replace Old-Style Do-Loops

One refactoring we implemented is called Replace Old-Style Do-Loops [9, 12]. There are many different ways to write a do-loop in Fortran, depending on what version of Fortran is being used. "Old-style" do-loops contain a numeric statement label in the loop header; the statement with that label constitutes the end of the loop (see Table 1). In contrast, "new-style" do-loops consist of matched DO/END DO pairs, which are generally preferred (see Table 1).

It was implemented as follows. *Preconditions*: The source code must have at

Table 1. Fortran DO LOOPs

```
    ....              ....              ....                  ....
    DO 100 I=1,30     DO 100 I=1,30     DO I=1,30             DO I=1,30
    V(I)=0            100 V(I)=0           V(I)=0                 100  V(I)=0
100 CONTINUE          ....                 100 CONTINUE       END DO
           ...        ....              END DO                ....
```

least one do-statement. The terminating statement label for each old-style do-loop must be unique. The terminating statement must be at the same level of the nesting as the do-statement. For example, the terminating statement cannot be inside an if-construct in the loop.
Transformation: This refactoring transforms all old-style do-loops in the selected files. An END DO statement is inserted immediately following the terminating statement for each old-style do-loop. The statement label is removed from the loop header, and the loop body is re-indented. Figure 1 shows the diff-like preview of an old-style do loop refactoring as implemented in Photran.

The most difficult part of implementing a new refactoring is designing a correct set of preconditions. We believe that Replace Old-Style Do-Loops is a straightforward, syntactic transformation, whereas many other refactorings require much more complicated analyses.

5 Related Work

The concept of refactoring as an interactive process performed by an expert programmer while carefully examining the code, in small and safe steps, was defined in Opdyke's thesis many years ago [7]. Since that time, Ralph Johnson's research group at the University of Illinois has promoted refactoring and the development of automated refactoring tools, although it was not until the advent of agile methodologies that refactoring received widespread attention. Specifically for Fortran, Vaishali De's master's thesis [3] enumerates a set of possible Fortran 90 refactorings. Later on, Overbey et al. [8] bring to light the need of refactoring tools integrated with IDEs for Fortran programs and in the High Performance world. Photran is introduced as an integrated development environment that provides the necessary infrastructure for implementing Fortran refactoring [1].In a subsequent work [9], a study founded on the Fortran evolution enumerates outdated language constructs that a refactoring tool could help remove from Fortran code and proposes, more generally, a role that refactoring tools could play in language evolution. As an example, Photran was used to eliminate global variables. Tinetti et al. [11] base their work improving Fortran legacy source for performance optimization on a weather climate model implemented about two decades ago. This work is close to some refactorings proposed in this paper.

6 Conclusions and Future Work

There are some automatic tools for upgrading or migrating Fortran programs, but they have not been successful in removing legacy features of code. We believe that refactoring tools can have a profound impact in this respect. For this reason, we are working on both: the definition of a catalog of Fortran refactorings, classified with the intention of guiding developers to use the right refactoring for their needs, and on the construction of a powerful tool for development and refactoring.

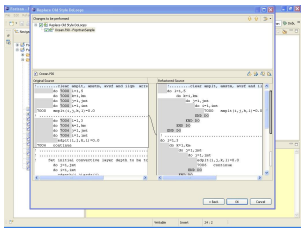

Fig. 1. Photran diff view of Replace old-style Do-Loop refactoring.

Future work includes implementing more refactorings on Photran and implying it on some case studies to measure the overall improvement. Another important factor is to encourage the scientific world to use Photran, and that will require not only successful stories of the use of Photran in large applications but also providing a formal foundation that ensures behavior preservation.

References

1. Photran, an Integrated Development Environment and Refactoring Tool for Fortran. http://www.eclipse.org/photran/.
2. J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.
3. Vaishali De. A Foundation for Refactoring Fortran 90 in Eclipse. Master's thesis, University of Illinois, 2004.
4. The Eclipse Foundation. Eclipse.org home. http://www.eclipse.org/, 2010.
5. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
6. JetBrains. IntelliJ IDEA 9. http://www.jetbrains.com/idea/, 2010.
7. W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, 1992.
8. J. L. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and High-Performance Computing. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39, New York, NY, USA, 2005. ACM.
9. J.L. Overbey, S. Negara, and R.E. Johnson. Refactoring and the Evolution of Fortran. In *2nd International Workshop on Software Engineering for Computational Science and Engineering (SECSE'09)*, 2009.
10. D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
11. F. G. Tinetti, P. G. Cajaraville, J. C. Labraga, M. A. López, and M. G. Olguín. Reverse Engineering Applied to Numerical Software: Climate Models (in Spanish). *X Workshop de Investigadores en Ciencias de la Computación*, pages 434–438, 2008. http://hpclinalg.webs.com/hpclinalg_en.html.
12. F. G. Tinetti, M. A. López, and P. G. Cajaraville. Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP. *World Congress on Computer Science and Information Engineering*, pages 471–475, 2009.