Regrowing a Language

Refactoring Tools Allow Programming Languages to Evolve

Jeffrey L. Overbey Ralph E. Johnson

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave. MC 258
Urbana, IL 61801
{overbey2,johnson}@cs.uiuc.edu

regrow *(v. tr.): to grow (as a missing part) anew; (v. int.): to continue growth after interruption or injury*

—Merriam-Webster's Online Dictionary, 11/e

Abstract

Successful programming languages change as they age. They tend to become more complex, and eventually some features become outdated or are rarely used. Programming tools for these languages become more complex as well, since they have to support archaic features. Old programs are hard to maintain, since these archaic features are unfamiliar to modern programmers. These problems can be solved by refactoring tools that can transform programs to use the modern form. We show that refactoring tools can ease the cost of program evolution by examining the evolution of two languages, Fortran and Java, and showing that each change corresponds to an automatable refactoring.

Categories and Subject Descriptors D.3.m [*Programming Languages*]: Miscellaneous

General Terms Languages

Keywords Fortran, Java, language design, language evolution, refactoring, restructuring

1. Introduction

Software designs evolve, and 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.

Over time, features are added to programming languages but rarely removed, for fear of breaking backward compatibility. This makes languages become increasingly complex over time. Complexity makes languages more difficult for programmers to learn, and it makes programming tools more expensive to build.

Eventually, some language features become outdated, and they should be removed from the language. But language designers have no strategy for doing this besides warning developers many years ahead of time.

We propose that automated refactoring tools be used to eliminate old constructs and idioms from source code. This would benefit language designers by allowing them to remove archaic constructs from languages more quickly than they do now. It would also benefit maintenance programmers, who could avoid learning old versions of a language.

Imagine a future in which refactorings are an essential part of language design and refactoring tools are an essential part of language implementation, on par with compilers. In such a world, languages could be designed with the expectation that end users would use refactoring tools to migrate their codebases from one version of the language to the next. While this power would need to be used judiciously, it would help languages maintain an elegant, coherent design over time. And it would not have to place an unreasonable burden on end users. Several basic, almost fully-automatic refactorings could make minimal changes to appease the compiler, removing outdated constructs and making other mandatory changes. These would be appropriate for generated code and other "black box" components. A second set of refactorings would provide the capabilities necessary to fully upgrade code to the newer version of a language. In some cases, these refactorings might involve design-level changes, in which case more interactivity would be both necessary and desirable.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-734-9/09/10. . . $10.00

This paper will describe the problems caused by language evolution, using Fortran and Java as examples (§2). It will then describe automated refactoring (§3) and discuss how, by forming a symbiotic relationship with language design, it can help solve these problems (§§4–5). The feasibility of this approach will be supported by tracing the changes that have been made to Fortran and Java and enumerating the refactorings that could have accompanied these changes (§6). This will show that refactoring tools could have made the evolution of both of these languages much less problematic.

2. On Language Evolution

2.1 How Languages Evolve

FORTRAN I [Backus et al. 1956] 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" [Janss 1999]), 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).

Even the much-younger Java language has evolved from its initial release in 1996. Although most of the changes have been in the library, the core language has been expanded as well, adding inner classes (JDK 1.1), `strictfp` (J2SE 1.2), `assert` statements (J2SE 1.4), annotations, autoboxing, enumerations, a new `for` loop, generics, `import static`, and varargs methods (J2SE 5.0).

Language evolution always comes at a cost. Introducing new features adds complexity to the language. Deleting features can make existing programs obsolete.

In the case of Fortran, the costs of evolution are largely due 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.

The costs of evolution are different in Java. Being significantly younger, it has not had to endure philosophical shifts (like the disbarring of `goto` statements) that would render part of the language obsolete. But it has still increased significantly in complexity. Java's original design attempted to keep the number of language features small, using idioms or libraries to express concepts when possible. Unfortunately, several things were awkward, inconvenient, or difficult to express in this way. For example, assertions could not easily be disabled in a production release, prompting the introduction of the `assert` statement. Collections required clients to repeatedly downcast to the contained type, and iterating through them used a fairly verbose idiom; these lead to the introduction of generics and the new-style `for` loop, respectively. As with Fortran, the addition of new features gave rise to different dialects of Java being used, depending on what version of the language was available at the time. For example, code written prior to the widespread availability of Java 5 will be littered with downcasts and `Iterator` instantiations, whereas code written later would use generics and new-style `for` loops instead.

The rise of such temporal dialects is a necessary consequence of language evolution. No one should be surprised that a Fortran program written in 1970 will use a different subset of the language than the same program written in 2010. But these differences will only be exacerbated as the language continues to evolve. One can only imagine comparing Fortran programs from 1970 and 2070.

But this is just a symptom of a larger problem, namely, that *code does not evolve with the language.* This has been noted, for example, by J.M. Favre [Favre 2005], who observes that "language evolution and language/program co-evolution ... are still neglected by the software engineering research community" and presents an extensive argument for their study.

2.2 Languages Evolve, Code Stays Behind

Programming language revisions turn up-to-date programs into legacy code. This is a problem. Arguably, the most important consideration is what impact it has on the day-to-day programmer. Many programmers can safely ignore outdated parts of the language, concentrating on the current dialect, or even a subset of that dialect. But the story is quite different for programmers forced to maintain others' code: 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 (and idioms) obsolete can leave a programmer blissfully unaware that he is using a construct for which a better alternative exists.

When code does not evolve with its language, maintaining backward compatibility means a language can be expanded, but nothing can be removed. For both Fortran and Java, backward compatibility is paramount, and this is evident in the way the languages evolved. Any Java 1.0 program will run on a Java 6 virtual machine and (excepting minor changes like the addition of the `assert` keyword) will compile using a Java 6 compiler. The constraints caused by backward compatibility are more evident in Fortran 2008, which 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 [Fortran 2003] 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.

2.3 The Resulting Impasse

The inability to delete old language constructs (as in the case of Fortran) 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. This makes the learning curve steeper and maintenance more difficult, but it also impacts programmers in a more subtle way: Retaining old language features (and increasing the complexity of the language) makes 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 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 older languages (like Fortran) tend to become niche markets, 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.

The most obvious non-solution to these problems is to "freeze" programming languages at some particular version and not allow them to evolve beyond that point. Fortran has had more than enough opportunities to do this (its death has been imminent for several decades now), and some people would argue that the same should be done for Java before it can be polluted with, say, parallel programming constructs. Unfortunately, no programming language is eternally, universally perfect. Not even Lisp, although it may be close. Some defects in a language's design will not be found until it is widely used, and opinions about good programming language design will continually be influenced by advances in computer architecture, compilers, programming language theory, and software engineering. Successful software systems live lives much longer than their creators anticipated, and they are made to perform tasks that their creators never intended; suggesting that a programming language should be frozen at a particular point in time tacitly denies that systems written in those languages will continue to evolve, adapting to new hardware, new programmers, and new requirements, and this evolution could be ameliorated through changes to the language. Programmers will find (often kludgy) ways to make the frozen language accommodate their needs. The counterargument is that these systems should be rewritten in a newer language; while this is perhaps justifiable on technical grounds, it ignores the tremendous intellectual and economic investments that companies have in existing software and languages (and the role of inertia in their selection).

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 an increasingly complex and dialect-ridden language. Deleting old features can break backward compatibility.

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

3. Enter Refactoring

Refactoring [Opdyke 1992, Fowler 1999] 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. Often, the intent is to allow the system's design to be changed retroactively so that unforeseen changes can be incorporated without compromising the integrity of the design.

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 [Photran, Overbey et al. 2005], ROSE [ROSE], and SPAG [SPAG] provide 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 it will not conflict with an existing 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 Libraries

Although refactoring has traditionally been applied to applications, recent research has suggested ways that it can also be applied to libraries, frameworks, and components.

Dig and Johnson [Dig and Johnson 2006] analyzed the evolution of several Java APIs, observing that 80% of the API changes could be expressed as common, automated refactorings. They further suggest that the refactorings used to change the library's API could be "recorded" as a script; the next version of the library could ship with that script, which could then be executed on client code to upgrade it to the newer version of the library.

For the 20% of changes that do not correspond to existing refactorings, it is sometimes possible to develop library-specific refactorings. For example, Dig et al. [Dig et al. 2009] implemented refactorings that migrate Java applications to use thread-safe collection classes that will be provided in Java 7, while Tansey and Tilevich [Tansey and Tilevich 2008] provide refactorings that migrate codebases from the JUnit 3 unit testing framework to JUnit 4.

4. Refactoring Languages

Refactorings are used frequently to allow applications to change. They can be used to allow libraries to change. But this idea should be taken a step further: Refactorings should allow languages themselves to change. *Automated refactoring tools can replace many outdated language constructs and idioms with their modern equivalents.*

In other words, we want to make language evolution painless. When a language standard changes, there should be tools to convert from the old to the new version of the language. Old programs will change to meet the new standard. Tool venders will not need to maintain versions of their tools for every version of the language.

In a sense, refactoring languages is similar to the problem of refactoring libraries. Libraries provide a vocabulary, and they have both a syntax (inherited from the host language) and a semantics, so they provide a "language" according to most definitions. And languages like Lisp and Smalltalk blur the distinction between library and language, so what would be considered a language change in Fortran would be considered a library change in Smalltalk. Thus, much of what we propose applies equally to libraries. But as we will see, refactoring the core of a language—the constructs handled directly by the compiler—is a different, and often more challenging, problem.

4.1 History

The idea that tools could help source code adapt to language changes first gained interest during the 1970s, motivated by the desire to convert programs using `goto` statements into structured programs (an application which led to the coining of the term *restructuring)* [Chikofsky and Cross II 1990]. Several systems provide this capability for Fortran (e.g., [SPAG, VAST]).

This idea also appeared in PLATO [Jones 1973, Woolley], a time-sharing system developed at the University of Illinois in the early 1960s. PLATO was designed for computer-assisted instruction; PLATO IV (1972–1992) could support over a thousand simultaneous users and had over ten thousand hours of courseware available. Most of the courseware was developed in a language called TUTOR. TUTOR evolved as new features were added to PLATO. Part of the PLATO folklore is that changes to the TUTOR language were often transparent to the programmers. PLATO did not have removable storage; all TUTOR programs were stored in a single file system. Thus, when a new version of TUTOR was installed, it was possible to scan the file system and change every TUTOR problem to fit the new standard. All the TUTOR manuals were online, so they could be changed, too. Many programmers would not even notice the change, because they did most of their programming by copying old programs, and when they would look at an old program to see how to use a particular feature, they would read the new version of that feature.

4.2 Interactivity: An Asset, Not a Liability – Part I

PLATO's upgrades, and the `goto` restructurers of the 1970s, were batch systems; they attempted to automate the entire process. What makes *refactoring* tools particularly promising is that they are interactive. Clearly, it is beneficial to automate transformations as much as possible, and good defaults must be chosen when the user is asked to intervene. We will return to these ideas in §5.1. But ultimately the ability to involve the user can make the process much more satisfactory. There are three reasons why this is the case.

First, as we will see in §§6.1–6.2, some transformations are necessarily heuristic, or ambiguous, or a completely conservative program analysis is not feasible. By making such a tool interactive, the analyses do not *have* to be completely conservative, and often it is acceptable for the tool to make a "guess" at the appropriate transformation and ask the user to visually inspect the result or provide some input to further assist the tool.

Similarly, there are some cases where a language change may result in design changes in the user's program. (Design, by definition, requires creativity and domain knowledge, and thus cannot be automated.) One example would be the introduction of modules in Fortran 90. While a tool could automatically eliminate older-style (`common`) global variables by turning them into module variables, the user would likely want to move some subprograms into modules as well. A tool could guess which procedures these would be, but its decision could not rival that of a domain expert.

Finally, refactoring tools modify the user's source code, and *the user maintains the refactored code,* so there are times when a user may want to intervene in order to influence the formatting, spacing, or comments in the refactored code—issues that are "uninteresting" to tools but extremely important to programmers.

5. Language/Refactoring Co-design

Based on our experience implementing several refactoring tools [Opdyke and Johnson 1990, Roberts et al. 1997, Garrido and Johnson 2003, Overbey et al. 2005] and previous successes in the area (e.g., [Kieżun et al. 2007]), we believe that refactoring tools can eliminate many outdated language constructs and idioms from programs. But our proposal is more ambitious than that. *We believe that such refactorings should play a key role in the language design process.*

Our vision is that refactoring tools will become an integral part of a language implementation, just as compilers are now. Every developer will have access to a refactoring tool, and it will not be a luxury but a necessity. When the language evolves and a new construct or idiom is introduced, it should be accompanied by a refactoring that replaces instances of the outdated construct or idiom. In other words, such refactorings would be developed proactively, alongside changes to the language, rather than as a reactive, best-effort attempt to upgrade codebases.

The implementations of such refactorings would need to be robust, reliable, and reasonably fast in practice—exactly what one expects of a compiler. Their availability would reduce the impetus to retain outdated language features, because such a tool would allow older programs to be updated almost "for free." This, in turn, would allow programming languages to eliminate outdated features and idioms much more aggressively.

Of course, this ability would need to be used with care. Becoming fluent in a new programming language is a substantial intellectual endeavor, and a succession of rapid, ill-conceived changes to a language could easily leave its end users frustrated. Similarly, language changes affect programming tools as well—compilers, IDEs, static analysis tools, etc.—and so language changes must still be carefully considered. (Practically, languages are not as malleable as software; this is one reason why programming language design usually follows something resembling the Waterfall model.) Nevertheless, when changes are warranted, the ability to propagate them through all of the code implemented using that language can be extremely powerful.

Accepting automated refactoring as a natural and essential part of upgrading from one version of the language to the next can benefit the language, its implementation, and its end users. When the refactorings are sufficiently general, "old" constructs may be eliminated from the language altogether, and they can be eliminated immediately upon the next release of the language. (Contrast this with current languages, where features are deprecated and then obsoleted over the course of many years, and most compilers retain them anyway, making them a *de facto* part of the standard language, even if they are not part of any official, published standard.) This means those features can also be eliminated from the compiler, static analysis tools, IDEs, etc. This can simplify the implementations of these tools, but it can also help the language design remain concise and cohesive. This also benefits the end users of the language, whose codebases will be upgraded to the current version of the language, eliminating the need for new programmers to learn outdated dialects (and possibly helping experienced programmers to un-learn them).

5.1 Interactivity: An Asset, Not a Liability – Part II

Of course, for upgrading a codebase to benefit the end users, the perceived cost cannot be unreasonably high. As discussed in §4.2, interactivity is essential. But at the same time, an obsequious refactoring tool is more annoying than helpful on a large system. There must be a way to control the amount of user input.

Refactoring, by definition, is a technique for restructuring software which uses small-scale, behavior-preserving changes to achieve larger, behavior-preserving changes in a system [Fowler 2004]. Refactoring does not imply that the system changes all at once. Rather, it implies that the system moves gradually, step-wise, from an initial design to a final design. This observation can be used to control the amount of interactivity in an evolutionary refactoring tool.

Some parts of a system never change and are treated as "black boxes." These include mature components and generated code. Developers do not want to devote effort to redesigning these; they simply want them to work. Other parts of a system are actively maintained, and developers want very fine-grained control over the code's appearance and the component's design.

A successful evolutionary refactoring tool must have two types of refactorings available.

The first make very minimal changes, just enough to make the code compliant with the new language specification. These address backwards-incompatible changes, like the removal of language constructs and changes to type systems. Ideally, these should require little or no interactivity, allowing them to be applied painlessly to black box code, although more interactivity may be optional for actively-maintained code.

After these have been applied, the remaining refactorings can be used to fully upgrade actively-maintained code. Often, these can be used to adjust a system's design to compensate for changes in the language. One example would be the elimination of old-style (`common`) global variables in Fortran. These can be converted into module variables by creating a new module from each `common` block. However, the resulting design is less than optimal. In actively-maintained code, the programmer would probably want to move some subprograms into the module as well, or he might want to merge that module with another one with similar functionality. Another example is the conversion of procedural programs to object-oriented programs. This requires creativity and domain knowledge, so one should not expect a "make program object-oriented" refactoring. 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.

That said, refactoring tools can still *ease the process* of eliminating these features from programs. For example, many of the *steps* in making a program object-oriented are algorithmic and are excellent candidates for implementation as refactorings, even though the entire process cannot be automated.

5.2 Open World, Open Problems

The language in PLATO could be upgraded at will because programs were moderately sized, the changes were minor, and the system was "closed" in the sense that every PLATO program existed on the (one) PLATO system. In fact, the documentation and tutorials existed on that system and could be changed as well.

Refactoring modern programming languages is more difficult because they exist in an ecosystem which includes compilers, build tools, preprocessors, and code generators, among other things. This is where many research challenges in refactoring can be found.

In the specific case of Fortran, widespread use of language extensions and preprocessors make automated source code transformation more difficult. 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 [Garrido 2005]; unfortunately, Fortran programmers do not just use the C preprocessor. Some use M4. Some use M5. Some go even further: IBEAM [IBEAM] 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.

Is there a uniform way to handle a variety of preprocessors? Can a minimal set of refactorings be applied if the preprocessor is treated as a black box? Can refactoring tools "learn" to refactor language extensions? Can generated code be refactored with user assistance once, and then the same refactoring be repeated, fully automatically, when the code is regenerated? Can a refactoring tool infer useful configuration information from build scripts? All of these are open problems whose solutions range from useful to essential in making refactoring-based language evolution a possibility.

6. Feasibility

If you look only at the language itself, and not the programming environment in which it is embedded, it is feasible to upgrade programs from an old version of a language to a newer version. We are going to show this by looking at two languages, Fortran and Java, examining how they *have* evolved and showing that the changes to these languages could have been accompanied by automated refactorings.

6.1 From FORTRAN to Fortran

We will begin by considering refactorings that could have expedited the removal of outdated features from the Fortran language. This features considered include those that the ISO standardization committee has officially deprecated (e.g., fixed source form) as well as some that are officially supported but whose use is discouraged in practice, either by culture (e.g., the use of keywords as identifiers) or because a newer alternative exists (e.g., `common` blocks).

- *Eliminate fixed source form.* Fixed source form was originally designed for 80-column punch cards. The first six characters on a line have special meanings, columns 73 and beyond are effectively a comment, actual code lies between columns 7 and 72, and whitespace can appear anywhere, even in the middle of a token. This is in contrast to the newer free source form, which is more lexically similar to C or Java. Fixed-to-free form converters already exist (e.g., [SPAG] and [VAST]), although it should be noted that this conversion meets the definition of a refactoring, and Fortran refactoring tools contain all of the machinery needed to build a fixed-to-free form converter that maintains comments and formatting. Fixed form is already an obsolescent feature in Fortran 95 and 2003 [Fortran 2003, § B.2.6], indicating the standardization committee's intent to delete it in a future revision of the standard.
- *Reserve keywords.* Using keywords like `if` and `while` as variable names is generally considered poor practice, and failing to reserve these 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. However, while a tool could auto-generate names (e.g., by adding a prefix or suffix), it could also benefit from the interactivity of refactoring by allowing the user to choose an entirely different name based on the application domain.
- *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. Again, user input is needed to generate meaningful names when a valid name cannot be inferred from the `common` block or `include` line.

- *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.
- *Require explicit variable declarations; eliminate implicit typing and `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—including `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 [Metcalf and Reid 1999, 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. While user input is not strictly necessary, many users would want to maintain some control over the order and formatting of the revised declarations.
- *Remove `entry` statements.* The `entry` statement allows several entrypoints to be declared within a single subprogram. The intent is 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 [Metcalf and Reid 1999, p. 240]. This refactoring is a somewhat more complicated variant of *Extract Method* [Fowler 1999, p. 110], 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 [Metcalf and Reid 1999, 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 form of a declaration for `character` (string) variables 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.

6.2 From Java 1 to Java 6

Next, we will consider the language changes between Java 1 and Java 6. Java is different from Fortran because no language constructs have been outdated. Rather, some *idioms* were outdated through the addition of new language constructs. This eases demands on the refactoring tool, since no language construct is obsoleted and therefore the original code is still valid. The objective of refactoring is more stylistic. At the same time, this can also make designing an effective refactoring difficult, since the exact patterns to match are necessarily heuristic and may vary from system to system. It can also increase the demands on the user. In some cases (e.g., introducing assertions), the user may need to indicate what patterns to match. In others (e.g., introducing generics), the refactoring cannot be blindly applied across the entire codebase, and so the user must specifically decide where it is and is not warranted.

- *Introduce generics (parametric polymorphism).* Of the new features introduced in J2SE 5.0, generics are by far the most complex, and it is not at all obvious that they can be introduced by a refactoring. Fortunately, the requisite refactorings (Introduce Type Parameter and Infer Generic Type Arguments) have already been studied in detail [Kieżun et al. 2007, Tip 2007]. Some user input is still required; most importantly, the user must decide which variables were intended to have a generic type.
- *Introduce new-style* for *loop.* The new-style *for* loop

  ```
  for (Something value : iterable) {
     ...
  }
  ```

  is syntactic sugar for the idiom

  ```
  Iterator<Something> it = iterable.iterator();
  while (it.hasNext()) {
     Something value = it.next();
     ...
  }
  ```

  Replacing this idiom with the corresponding new-style *for* loop involves both matching the idiom syntactically and verifying that the iterator (`it`) is not accessed except as shown. Unfortunately, not everyone uses this exact idiom; the most difficult part of defining this refactoring is ensuring that it handles a sufficient number of common variations. For example, the `it` declaration and *while* loop are often combined into an (old-style) `for` loop. It is also possible that `it` and/or `value` may be declared earlier; `it` may even be reused several times. The user may also insert other statements between those shown (e.g., between the `iterator()` invocation and the *while* loop), in which case a dependence analysis is necessary to verify that the intervening statements do not influence the iterator's operation. The new-style *for* loop can also be used to iterate through arrays; a refactoring has already been implemented in Eclipse which converts C-style array iterations into new-style *for* loops. The preconditions here are similar but somewhat more complicated (e.g., the elements must be accessed from index 0 through *length* $- 1$, each iteration i must read only the element at index i, and the numeric index i must not be used for any other purpose, such as updating the elements in the array).
- *Convert to enumeration.* Like the new-style *for* loop, `enum` declarations are syntactic sugar for a particular idiom ("type-safe enumerations") which can be mechanically replaced. However, programmers also tend to use integer constants to mimic a C `enum`, and so a useful refactoring would also need to allow the user to convert these to Java 5 enumerations. In practice, this is often used in tandem with refactorings like *Replace Conditional with Polymorphism* [Fowler 1999, p. 255], which a tool might also provide. Again, the choice between classes, constants, and enumerations is a design decision.
- *Introduce autoboxing.* §§5.1.7–5.1.8 of the Java Language Specification, 3/e [Gosling et al. 2005] define the contexts in which primitive values are boxed and unboxed automatically. Since there are only two ways to box each primitive (for `int` values, these are the `Integer` class constructor and `Integer#valueOf(int)`) and one to unbox (`Integer#intValue()`), the refactoring involves matching these patterns and determining whether or not they occur in a context where the primitive would be auto-boxed/unboxed. It may be necessary to parenthesize the replacement expression to maintain the correct precedence and associativity. While this refactoring is straightforward, the user must be aware that it *may not preserve semantics* if the code makes any assumptions about the pointer-equality of boxed objects (cf. [Gosling et al. 2005, p. 87]). Although a tool could attempt to determine this (e.g., using a pointer analysis and checking for `==` and `!=` comparisons), the analysis would be expensive and very conservative, so the decision should ultimately be left to a user familiar with the code.
- *Introduce static import.* Java 5's `import static` allows `static` members of another class to be accessed without qualifiers. A canonical use is to reference constants like `Math.PI` or methods like `Math.cos()` as simply `PI` and `cos()`. Sun recommends that this feature be used "very sparingly," when the programmer would otherwise "be tempted to declare local copies of constants, or to abuse inheritance" [Static Import]. As such, an automated refactoring to remove static qualifiers would similarly need to be applied judiciously. A tool could detect some candidates (e.g., if a class uses several features from Java's `Math` class, or one feature repeatedly), but ultimately the user would need to decide when a static import is warranted. The actual transformation is simple: ensure that the unqualified name will neither shadow nor be shadowed in the target namespace, add an `import static` statement, and then remove the static qualifiers from uses of the name.
- *Convert to varargs methods.* Varargs methods are also syntactic sugar for another idiom: passing an array as the last argument to a method in order to simulate a variable number of arguments. Like static imports, they should be used sparingly. Blindly replacing every possibile occurrence would be overzealous, since there are times when an array represents (for example) a vector or matrix, and a tool cannot tell the difference between these cases and those where a varargs transformation would be appropriate. Again, user input and domain knowledge are required.

- *Force strict floating point computations (`strictfp`).* J2SE 1.2 changed the semantics of floating point computations in the Java virtual machine, and the `strictfp` keyword was introduced to force the JVM to use the older floating point behavior. Determining when `strictfp` is necessary is a decision best left to the end user. Completely preserving the semantics of a Java 1.1 program on a J2SE 1.2 JVM would require an abundance of `strictfp` modifiers in the code; however, given how infrequently `strictfp` actually appears in Java code, this is clearly one case where preserving semantics is often *not* desirable, since the "new" floating point behavior is acceptable for most applications. Thus, a refactoring to introduce the `strictfp` keyword would be useful only to the extent that it could provide a means for the user to quickly add the keyword to a large number of classes (or methods) at his discretion.
- *Assertions.* Assertions are different because there is no single idiom they were intended to replace. Prior to their introduction, each system had a proprietary idiom for making assertions, often something like

  `Assert.isTrue(`*expression*`);`

  This is one case where a general, syntactic find-and-replace is probably the best option, since both the syntax to replace and its replacement are system-dependent.
- *Annotations.* Annotations were a fundamentally new addition to the Java language, providing a facility that was not previously available, and so there is no widespread idiom that annotations were intended to replace. However, J2SE 5.0 also introduced three specific annotations (`@Override`, `@Deprecated`, and `@SuppressWarnings`) that could easily be inferred and introduced by a refactoring tool. `@Deprecated` could be inferred from the `@deprecated` JavaDoc tag. Suppressing compiler warnings would, of course, need to be done at the discretion of the user.

7. Conclusions

Our experiences implementing refactoring tools, and our observations about the evolution of Fortran and Java, support our belief that refactoring is a viable technical strategy for language evolution.

Why hasn't this happened already? There are many technical challenges, but perhaps the biggest obstacle is cultural. Only recently have refactoring tools been adopted widely in the Java and C# communities, and 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 [Daniel et al. 2007, Schäfer et al. 2008]. Many Java programs are running business-critical applications. Many Fortran programs are in government labs, running highly classified simulations in strict security environments under heavy bureaucracies. For refactoring to become a vehicle for language evolution, refactoring tools will need to enjoy the same robustness and trustworthiness that compilers currently do.

These obstacles are also surmountable, but they will require the momentum of a community. Language designers should consider what can be removed using refactoring and what is possible when the constraints of backward compatibility are removed. Programmers should learn to use refactorings, making them an essential part of their toolbox. Researchers and refactoring tool builders should focus on improving the reliability, scalability, and practical applicability of their tools. This change will require a significant effort, but it has great promise. We hope that others in the research community will agree, and the future will see a great number of languages evolving through refactoring.

Onward!

Envision the year 2020. Java 11 has just been released. Mixins and parallel programming constructs have been added. The `instanceof` operator is gone. The broken parts of the type system have been fixed. And the community decided that `import static` wasn't such a great idea after all.

The decision was made to upgrade your latest project to Java 11, and you just downloaded the Java 11 Developer's Kit. But that consists of more than a compiler and runtime: There is a tutorial on all the new features in Java 11, and there is a Java 11 Upgrade Kit. When you open your IDE, the Upgrade Kit has plugged in a variety of Java 11 refactorings.

You run a basic, fully-automatic refactoring to make your system Java 11-compliant, which makes some changes but also marks some portions of the changed code as potentially needing human assistance. Many of the changes are to a library that you purchased from another company. You figure that you'll get a Java 11 version of the library before too long, so there is no reason for you to clean it up now. So you focus on the parts of your own code that need to be improved. In some cases, the changed code is good enough. In others, there are design improvements needed. Through a series of several more refactorings, you are able to upgrade your code to a proper Java 11 design. In other places, you are able to improve your code by taking advantage of Java 11's new features.

At the end of the day, you run the regression test suite and check the upgraded code into version control. There are still parts of the code that ought to be cleaned up someday, but the parts of the program that you maintain from day to day are all in good shape. When you come into work the next day, you will be working exclusively on a Java 11 system. The language evolved, and so did your system.

Acknowledgments

This work was supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation. The authors would like to thank Paul Adamczyk, Bariş Aktemur, John Brant, Nicholas Chen, Danny Dig, Isaac Dooley, Brian Foote, Munawar Hafiz, Binh Le, Darko Marinov, Daniel Overbey, and Maurice Rabb for discussions that contributed to this paper, and Stas Negara who moderated our writers' workshop and contributed to an earlier version of this paper submitted to the 2nd International Workshop on Software Engineering for Computational Science and Engineering [Overbey et al. 2009].

References

J. Backus, R. Beeber, S. Best, R. Goldberg, H. Herrick, R. Hughes, L. Mitchell, R. Nelson, R. Nutt, D. Sayre, P. Sheridan, H. Stern, and I. Ziller. *Fortran Automatic Coding System for the IBM 704: Programmer's Reference Manual*. International Business Machines Corporation, New York, NY, 1956.

E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. ESEC-FSE '07*, pages 185–194, New York, NY, 2007. ACM.

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

D. Dig, J. Marrero, and M. Ernst. Refactoring sequential code for concurrency via concurrent libraries. In *Proc. ICSE '09*, Washington, DC, USA, 2009. IEEE Computer Society.

J. Favre. Languages evolve too! Changing the software time scale. In *Proc. IWPSE '05*, pages 33–44, Washington, DC, USA, 2005. IEEE Computer Society.

M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, Boston, MA, 1999.

M. Fowler. MF Bliki: RefactoringMalapropism. http://martinfowler.com/bliki/RefactoringMalapropism.html, January 2004.

A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, 2005.

A. Garrido and R. Johnson. Refactoring C with conditional compilation. *Proc. ASE '03*.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Boston, MA, 3/e, 2005.

IBEAM. IBEAM. http://www.ibeam.org.

Fortran 2003. *ISO/IEC 1539-1:2004: International standard: information technology, programming languages, Fortran*. International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland, 4/e, 2004.

N. Janss. Professor Forsythe. http://infolab.stanford.edu/pub/voy/museum/pictures/display/floor1.htm, 1999.

D. Jones. Run time support for the TUTOR language on a small computer system. Master's thesis, University of Illinois at Urbana-Champaign, Champaign, IL, 1973.

A. Kieżun, M. Ernst, F. Tip, and R. Fuhrer. Refactoring for parameterizing Java classes. In *Proc. ICSE '07*, pages 437–446, Washington, DC, USA, 2007. IEEE Computer Society.

M. Metcalf and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1999.

W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

W. Opdyke and R. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. SOOPPA*. ACM, September 1990.

J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and high-performance computing. In *Proc. SE-HPCS '05*, pages 37–39, New York, NY, USA, 2005. ACM.

J. Overbey, S. Negara, and R. Johnson. Refactoring and the evolution of Fortran. In *Proc. SECSE '09*, pages 28–34, Washington, DC, USA, 2009. IEEE Computer Society.

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

D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

ROSE. ROSE. http://www.rosecompiler.org/.

M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *Proc. OOPSLA '08*, pages 277–294, New York, NY, USA, 2008. ACM.

SPAG. SPAG. http://www.polyhedron.co.uk/spag0html.

Static Import. Static Import. http://http://java.sun.com/j2se/1.5.0/docs/guide/language/static-import.html.

W. Tansey and E. Tilevich. Annotation refactoring: Inferring upgrade transformations for legacy applications. *SIGPLAN Notices*, 43(10):295–312, 2008.

F. Tip. Refactoring using type constraints. In *Proc. SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag Berlin Heidelberg, 2007.

VAST. VAST/77to90. http://www.crescentbaysoftware.com/vast_77to90.html.

D. Woolley. Plato: The emergence of online community. http://thinkofit.com/plato/dwplato.htm.