Information and Software Technology 58 (2015) 95–109

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automatic transformation of iterative loops into recursive methods

David Insa, Josep Silva *

Departament de Sistemes Informàtics i Computació, Universitat Politècnica de València, Camino de Vera s/n, E-46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 30 May 2013
Received in revised form 3 September 2014
Accepted 1 October 2014
Available online 29 October 2014

Keywords:
Program transformation
Iteration
Recursion

ABSTRACT

Context: In software engineering, taking a good election between recursion and iteration is essential because their efficiency and maintenance are different. In fact, developers often need to transform iteration into recursion (e.g., in debugging, to decompose the call graph into iterations); thus, it is quite surprising that there does not exist a public transformation from loops to recursion that can be used in industrial projects (i.e., it is automatic, it handles all kinds of loops, it considers exceptions, etc.).
Objective: This article describes an industrial algorithm implemented as a Java library able to automatically transform iterative loops into equivalent recursive methods. The transformation is described for the programming language Java, but it is general enough as to be adapted to many other languages that allow iteration and recursion.
Method: We describe the changes needed to transform loops of types *while/do/for/foreach* into recursion. We provide a transformation schema for each kind of loop.
Results: Our algorithm is the first public transformation that can be used in industrial projects and faces the whole Java language (i.e., it is fully automatic, it handles all kinds of loops, it considers exceptions, it treats the control statements *break* and *continue*, it handles loop labels, it is able to transform any number of nested loops, etc.). This is particularly interesting because some of these features are missing in all previous work, probably, due to the complexity that their mixture introduce in the transformation.
Conclusion: Developers should use a methodology when transforming code, specifically when transforming loops into recursion. This article provides guidelines and algorithms that allow them to face different problems such as exception handling. The implementation has been made publicly available as open source.

1. Introduction

Iteration and recursion are two different ways to reach the same objective. In some paradigms, such as the functional or logic, iteration does not even exist. In other paradigms, e.g., the imperative or the object-oriented paradigm, the programmer can decide which of them to use. However, they are not totally equivalent, and sometimes it is desirable to use recursion, while other times iteration is preferable. In particular, one of the most important differences is the performance achieved by both of them. In general, compilers have produced more efficient code for iteration, and this is the reason why several transformations from recursion to iteration exist (see, e.g., [12,16,18]). Recursion in contrast is known to be more intuitive, reusable and debuggable. Another advantage of recursion shows up in presence of hierarchized memories. In fact, other researchers have obtained both theoretical and experimental results showing significant performance benefits of recursive algorithms on both uniprocessor hierarchies and on shared-memory systems [20]. In particular, Gustavson and Elmroth [4,10] have demonstrated significant performance benefits from recursive versions of Cholesky and QR factorization, and Gaussian elimination with pivoting.

Recently, a new technique for algorithmic debugging [15] revealed that transforming all iterative loops into recursive methods before starting the debugging session can improve the interaction between the debugger and the programmer, and it can also reduce the granularity of the errors found. In particular, algorithmic debuggers only report buggy methods. Thus, a bug inside a loop is reported as a bug in the whole method that contains the loop, which is sometimes too imprecise. Transforming a

* Corresponding author. Tel.: +34 96 387 7007x73530.
E-mail addresses: dinsa@dsic.upv.es (D. Insa), jsilva@dsic.upv.es (J. Silva).
URLs: http://www.dsic.upv.es/~dinsa/ (D. Insa), http://www.dsic.upv.es/~jsilva/ (J. Silva).

http://dx.doi.org/10.1016/j.infsof.2014.10.001
0950-5849/

loop into a recursive method allows the debugger to identify the recursive method (and thus the loop) as buggy. Hence, we wanted to implement this transformation and integrate it in the *Declarative Debugger for Java* (DDJ), but, surprisingly, we did not find any available transformation from iterative loops into recursive methods for Java (neither for any other object-oriented language). Therefore, we had to implement it by ourselves and decided to automatize and generalize the transformation to make it publicly available. From the best of our knowledge this is the first transformation for all kinds of iterative loops. Moreover, our transformation handles exceptions and accepts the use of any number of *break* and *continue* statements (with or without labels).

One important property of our transformation is that it always produces tail recursive methods [3]. This means that they can be compiled to efficient code because the compiler only needs to keep two activation records in the stack to execute the whole loop [1,11]. Another important property is that each iteration is always represented with one recursive call. This means that a loop that performs 100 iterations is transformed into a recursive method that performs 100 recursive calls. This equivalence between iterations and recursive calls is very important for some applications such as debugging, and it produces code that is more maintainable.

The objective of this article is twofold. On the one hand, it is a description of a transformation explained in such a way that one can study the transformation of a specific construct (e.g., exceptions) without the need to see how other constructs such as the statement *return* are transformed. This decomposition of the transformation into independent parts can be very useful for academic purposes. In particular, the paper describes the transformation step by step using different sections to explain the treatment of advanced features such as exception handling and the use of labels. Because we are not aware of any other publicly available description, some parts can help students and beginner programmers to completely understand and exercise the relation between iteration and recursion, while other more advanced parts can be useful for the implementors of the transformation. On the other hand, the proposed transformation has been implemented as a publicly available library. From the best of our knowledge, this is the first automatic transformation for an object-oriented language that is complete (i.e., it accepts the whole language).

Example 1.1. Transforming loops to recursion is necessary in many situations (e.g., compilation to functional or logic languages, algorithmic debugging, program understanding, memory hierarchies optimization, etc.). However, the transformation of a loop into an equivalent recursive method is not trivial at all in the general case. For this reason, there exist previous ad-hoc implementations that cannot accept the whole language, or that are even buggy. For instance, the transformation proposed in [7] does not accept exceptions and it crashes in situations like the following:

```
for (int i = 0; i < 10; i++)
  for (int j = 0; j < 10; j++)
    break;
```

due to a bug in the implementation. Consider the Java code in Algorithm 1 that is not particularly complicated, but shows some of the difficulties that can appear during a transformation.

Algorithm 1. Iterative loop with exceptions

```
 1: public int example(int x) throws IOException {
 2:   loop1:
 3:   while (x < 10) {
 4:     try {
 5:       x = 42 / x;
 6:     } catch (Exception e) {break loop1;}
 7:     loop2:
 8:     for (int i = 1; i < x; i++)
 9:       if (x % i > 0);
10:         throw new Exception1();
11:       else continue loop1;
12:   }
13:   return x;
14:}
```

This algorithm contains two nested loops (*while* and *for*). Therefore, it would be normally translated to recursion using three methods, one for the original method *example*, one for the outer loop *loop1*, and one for the inner loop *loop2*. However, the use of exceptions and statements such as *break* and *continue* poses restrictions on the implementation of these methods. For instance, observe in line 11 that the control can pass from one loop to the other due to the use of the label *loop1*. This forces the programmer to implement some mechanism to record the values of all variables shared by both loops and pass the control from one loop to the other when this point is reached. Note also that this change in the control could affect several levels (e.g., if a *break* is used in a deeper loop). In addition, the use of exceptions imposes additional difficulties. Observe for instance that the inner loop throws an exception *Exception1* in line 10. This exception could inherit from *IOException* and thus it should be captured in method *loop2* and passed in some way to method *loop1* that in turn should decide if it catches the exception or passes it to method *example* that would throw it. From the best of our knowledge, this example cannot be translated to recursion by any of the already existing transformations.

In the rest of the paper we present our transformation for all kinds of loops in Java (i.e., *while/do/for/foreach*), and we describe in detail the transformation for *while* loops. We start with an illustrative example that provides the reader with a general view of how the transformation works.

Example 1.2. Consider the Java code in Algorithm 2 that computes the square root of the input argument.

Algorithm 2. Sqrt (iterative version)

```
1: public double sqrt(double x) {
2:   if (x < 0)
3:     return Double.NaN;
4:   double b = x;
5:   while (Math.abs(b * b - x) > 1e-12)
6:     b = ((x / b) + b) / 2;
7:   return b;
8: }
```

This algorithm implements a *while*-loop where each iteration obtains a more accurate approximation of the square root of variable *x*. The transformed code is depicted in Algorithm 3 that implements the same functionality but replacing the *while*-loop with a new recursive method *sqrt_loop*.