

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.*

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.

[☆]This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. David Insa was partially supported by the Spanish Ministerio de Educación under FPU grant AP2010-4415.

*Corresponding Author. Phone Number: (+34) 96 387 7007 (Ext. 73530)

Email addresses: dinsa@dsic.upv.es (David Insa), jsilva@dsic.upv.es (Josep Silva)

URL: <http://www.dsic.upv.es/~dinsa/> (David Insa),

<http://www.dsic.upv.es/~jsilva/> (Josep Silva)

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.

Keywords: Program transformation, Iteration, Recursion

2010 MSC: 68N15, 68W40

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., [? ? ?]). 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 [?]. In particular, Gustavson and Elmroth [? ?] 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 [?] 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 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 [?]. 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 [? ?]. 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 [?] 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.

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

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: }
```

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`.

Algorithm 3 Sqrt (recursive version)

```
1: public double sqrt(double x) {
2:   if (x < 0)
3:     return Double.NaN;
4:   double b = x;
5:   if (Math.abs(b * b - x) > 1e-12)
6:     b = this.sqrt_loop(x, b);
7:   return b;
8: }
9: private double sqrt_loop(double x, double b) {
10:  b = ((x / b) + b) / 2;
11:  if (Math.abs(b * b - x) > 1e-12)
12:    return this.sqrt_loop(x, b);
13:  return b;
14: }
```

Essentially, the transformation performs two steps:

1. Substitute the original loop by new code (lines 5-6 in Algorithm 3).
2. Create a new recursive method (lines 9-14 in Algorithm 3).

In Algorithm 3, the new code in method `sqrt` includes a call (line 6) to the recursive method `sqrt_loop` that implements the loop (lines 9-14). This new recursive method contains the body of the original loop (line 10). Therefore, each time the method is invoked, an iteration of the loop is performed. The rest of the code added during the transformation (lines 5, 11-13) is the code needed to simulate the same effects of a *while*-loop. Therefore, this is the only code that we should change to adapt the transformation to the other kinds of loops (*do/for/foreach*).

The rest of the paper is organized as follows. Section 2 discusses some related approaches. In Section 3 we introduce our transformations for each particular kind of loop and provide detailed examples and explanations for *while* loops. In Section 4 we extend our algorithms with a special treatment for *break* and *continue* statements. Section 5 presents the transformation in presence of exceptions and errors (*try*, *catch*, *throw* and the hierarchy of objects that inherit from *Throwable*). Section 6 presents the transformation in presence of recursion and the statements *return* and *goto*. Section 7 presents the implementation of the technique, some optimizations that can be applied to the general transformation, and an empirical evaluation with a benchmarks suite. Finally, Section 8 concludes.

2. Related work

The theoretical relation between iteration and recursion has been studied for decades in different paradigms, with regard to types, with regard to procedures, from a mathematical point of view, from an algorithmic point of view, etc. There is a big amount of work that studies this theoretical relation (see, e.g., [? ?]).

On the practical side, there exist many different approaches to transform recursive functions into loops (see, e.g., [? ? ?]). This has been in fact a hot area of research related to compilers optimization and tuning. Contrarily, there are very few approaches devoted to transform loops into equivalent recursive functions. However, recursion provides important advantages in debugging [?], theorem proving [?], verification [?] and termination analysis [?].

For instance, algorithmic debugging is a debugging technique that asks the programmer about the validity of method executions, i.e., an algorithmic debugger can ask the question: `object.method(40,2)=42?`. If the answer is `yes`, then the debugger knows that this particular execution of the method was correct. If the answer is `no`, then the debugger knows that method `method` or one of the methods invoked during the execution of `method` is buggy (and it continues asking questions in that direction to find the bug). The final output of the debugger is always a buggy method. This is often very imprecise and forces the programmer to inspect the whole method to find the bug. If we are able to transform all loops into recursive methods, this means that algorithmic debuggers are able to detect bugs inside loops, and report a particular loop as buggy. For this reason, the debugger DDJ implements our transformation from loops to recursion since version 2.6.

In some articles, some transformations are proposed or mentioned for particular kinds of loops. For instance, in [?] a transformation for *while*-loops is used to detect the termination conditions of the transformed recursive functions. Unfortunately, the other kinds of loops are ignored, but they provide a treatment for the *break* statement (however they do not allow the use of labels, which is what introduces the complexity in the transformation).

In [?], authors argue that recursive functions are more efficient than loops in some architectures. In particular, they want to measure the performance benefits of recursive algorithms on multi-level memory hierarchies and on shared memory systems. For that, they define a transformation for *do*-loops (very similar to the Java's *for*-loop).¹ The use of *break* and *continue* is not accepted by the transformation.

Myreen and Gordon describe a theorem prover in [?]. In their examples, they include an ad-hoc transformation of a *while*-loop to an equivalent recursive function. This is done because the recursive function facilitates the verification condition generation and avoids the inclusion of assertions in the source code.

¹The syntax of their *do*-loops is “`do k = 1, N`” meaning that N iterations must be done with `k` taking values from 1 to N.

We have not been able to find a description of how to transform some specific loops such as *foreach*-loop. Moreover, the program transformations that appear in the literature are just mentioned or based on ad-hoc examples; and we are not aware of any transformation that accepts *break/continue* statements with labels. The use of exceptions has been also ignored in previous implementations. From the best of our knowledge no systematic transformation has been described in the literature yet, thus, researchers and programmers are condemned to reinvent the wheel once and again.

The lack of a systematic program transformation is a source of inefficient implementations. This happens for instance when transforming loops into functions that are not tail-recursive [?] or in implementations that only work for a reduced subset of the languages and produce buggy transformed code in presence of not treated commands such as *throws*, *try-catch*, *break* or *continue* [?].

In this work, we also describe our implementation that is the first to accept the whole Java language; and it allows us (in the particular case) to automatically transform a given loop into a set of equivalent recursive methods, or (in the general case) to input a program and output an equivalent program where all loops have been automatically transformed into recursion. This library has been already incorporated in the Declarative Debugger for Java (DDJ) [?].

Even though we present a transformation for Java, it could be easily adapted to many other languages. The only important restriction is that *goto* statements are not treated (we discuss their use in Section 6). Moreover, our transformation uses some features of Java:

- it uses blocks to define scopes. Those languages where blocks cannot be defined, or where the scope is not limited by the use of blocks should use fresh variable names in the generated code to avoid variable clashes between the variables of the transformation and the variables of the rest of the program.
- it assumes that, by default, arguments are passed by value. In those languages where the default argument passing mechanism is different, the transformation can be simplified. For instance, with call by value, exceptions need a special treatment described in Section 5. With call by reference, this special treatment can be omitted.

3. Transforming loops into recursive methods

Our program transformations are summarized in Table 1. This table has a different row for each kind of loop. For each loop, we have two columns. One for the iterative version of the loop, and one for the transformed recursive version. Observe that the code is presented in an abstract way, so that it is formed by a parameterized skeleton of the code that can be instantiated with any particular loop of each kind.

In the recursive version, the code inside the ellipses is code inserted by the programmer (it comes from the iterative version). The rest of the code

is automatically generated by the transformation. Here, `result` and `loop` are fresh names (not present in the iterative version) for a variable and a method respectively; `type` is a data type that corresponds to the data type declared by the user (it is associated to a variable already declared in the iterative version). The code inside the squares has the following meaning:

`1` contains the sequence formed by all variables declared in `Code1` (and in `ini` in *for*-loops) that are used in `Code2` and `cond` (and in `upd` in *for*-loops).

`1'` contains the previous sequence but including types (because it is used as the parameters of the method, and the previous sequence is used as the arguments of the call to the method).

`2` contains for each object in the array `result` (which contains the same variables as `1` and `1'`), a casting of the object to assign the corresponding type. For instance, if the array contains two variables `[x,y]` whose types are respectively `double` and `int`; then `2` contains:

```
x = (Double) result[0];
y = (Integer) result[1];
```

Observe that, even though these steps are based on Java, the same steps (with small modifications) can be used to transform loops in many other imperative or object-oriented languages. The code in Table 1 is generic. In some specific cases, this code can be optimized. For instance, observe that the recursive method always returns an array of objects (`return new Object[] { ... }`) with all variables that changed in the loop. This array is unnecessary and inefficient if the recursive method only needs to return one variable (or if it does not need to return any variable). Therefore, the creation of the array should be replaced by a single variable or null (i.e., `return null`). In the rest of the paper, we always apply optimizations when possible, so that the code does not perform any unnecessary operations. This allows us to present a generic transformation as the one in Table 1, and also to provide specific efficient transformations for each kind of loop. The optimizations are not needed to understand the transformation, but they should be considered when implementing it. Therefore, we will explain the optimizations in detail in the implementation section.

The four kinds of loops (*while/do/for/foreach*) present in the Java language behave nearly in the same way. Therefore, the modifications needed to transform each kind of loop into a recursive method are very similar. In the rest of this section we explain the transformation of *while*-loops with a detailed example. We refer the reader to [?] for an explanation of the variations needed to adapt the transformation for *do/for/foreach*-loops.

3.1. Transformation of while-loops

In Table 2 we show a general overview of the steps needed to transform a Java iterative *while*-loop into an equivalent recursive method. Each step is described in the following.

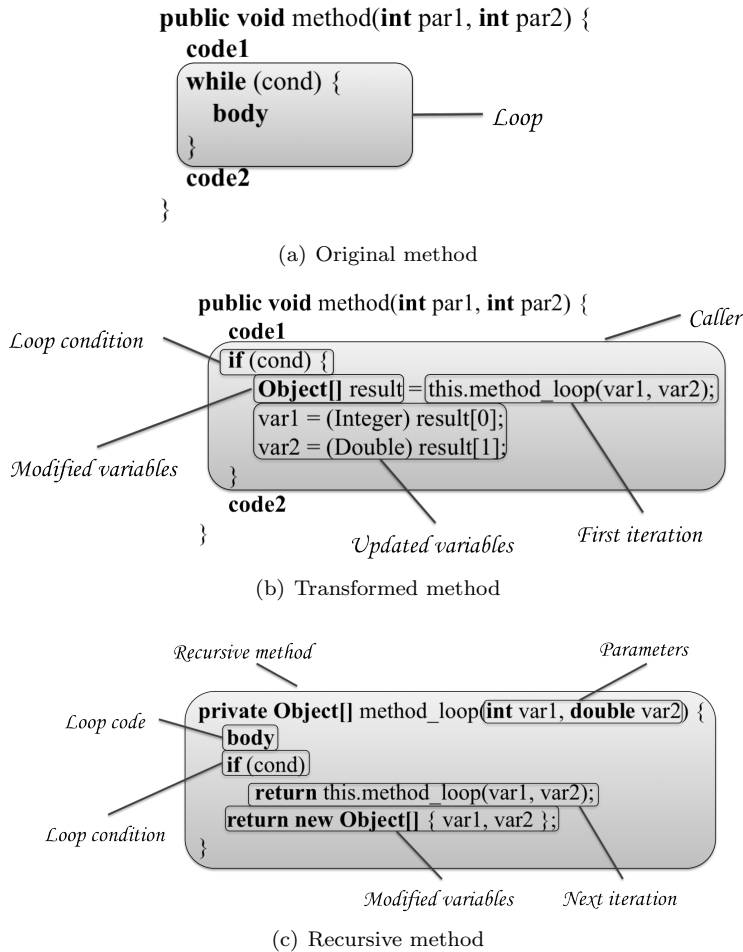


Figure 1: *while*-loop transformation

3.1.1. Substitute the loop by a call to the recursive method

The first step is to remove the original loop and substitute it with a call to the new recursive method. We can see this substitution in Figure 1(b). Observe that some parts of the transformation have been labeled to ease later references to the code. The tasks performed during the substitution are explained in the following:

- **Perform the first iteration**

In the *while*-loop, first of all we check whether the *loop condition* holds. If it does not hold, then the loop is not executed. Otherwise, the *first iteration* is performed by calling the recursive method with the variables used inside the loop as arguments of the method call. Hence, we need an

analysis to know what variables are used inside the loop. The recursive method is in charge of executing as many iterations of the loop as needed.

- **Catch the variables modified during the recursion**

The variables modified during the recursion cannot be automatically updated in Java because all parameters are passed by value. Therefore, if we modify an argument inside a method we are only modifying a copy of the original variable. This also happens with objects. Hence, in order to output those *modified variables* that are needed outside the loop, we use an array of objects. Because the *modified variables* can be of any data type², we use an array of objects of class **Object**.

In presence of call-by-reference, this step should be omitted.

- **Update the modified variables**

After the execution of the loop, the *modified variables* are returned inside an **Object** array. Each variable in this array must be cast to its respective type before being assigned to the corresponding variable declared before the loop.

In presence of call-by-reference, this step should be omitted.

3.1.2. Create the recursive method

Once we have substituted the loop, we create a new method that implements the loop in a recursive way. This *recursive method* is shown in Figure 1(c).

The code of the *recursive method* is explained in the following:

- **Define the method's parameters**

There are variables declared inside a method but declared outside the loop and used by this loop. When the loop is transformed into a *recursive method*, these variables are not accessible from inside the *recursive method*. Therefore, they must be passed as arguments in the calls to it. Hence, the parameters of the *recursive method* are the intersection between the variables declared before the loop and the variables used inside it.

- **Define the code of the recursive method**

Each iteration of the original iterative loop is emulated with a call to the new recursive method. Therefore in the code of the *recursive method* we have to execute the current iteration and control whether the *next iteration* must be executed or not.

- **Include the code of the original loop**

When the *recursive method* is invoked it means that we want to execute one iteration of the loop. Therefore, we place the *original code* of the loop at the beginning of the *recursive method*. This code

²In the case that the returned values are primitive types, then they are naturally encapsulated by the compiler in their associated primitive wrapper classes.

is supposed to update the variables that control the *loop condition*. Otherwise, the original loop is in fact an infinite loop and the recursive method created will be invoked infinitely.

– **Perform the next iteration**

Once the iteration is executed, we check the *loop condition* again to know whether another iteration must still be executed. In such a case, we perform the *next iteration* with the same arguments. Note that the values of the arguments can be modified during the execution of the iteration, therefore, each iteration has different arguments values, but the names and the number of arguments remain always the same.

– **Otherwise return the modified variables**

If the loop condition does not hold, the loop ends and thus we must finish the sequence of *recursive method* calls and return to the original method in order to continue executing the rest of the code. Because the arguments have been updated in each recursive call, at this point we have the last values of the variables involved in the loop. Hence these variables must be returned in order to update them in the original method. Observe that these variables are passed from iteration to iteration during the execution of the recursive method until it is finally returned to the recursive method caller.

In presence of call-by-reference, this step should be omitted.

Figure 2 shows an example of transformation of a *while*-loop.

4. Treatment of *break* and *continue* statements

The control statements *break* and *continue* can change the normal control flow of a loop. These commands can also be used in combination with a parameter that specifies the specific loop that should be continued or broken. We can illustrate their use with a Java program extracted from <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html> and shown in Algorithm 4. This program takes two strings and decides whether the second is a substring of the first.

In the example, the statement *continue test* (line 11) is used to continue the search in the next substring of the *searchMe* variable. Since it contains the label *test*, it affects the *for*-loop (labeled with “*test*”) instead of the *while*-loop (as it would happen if *continue* was unlabeled).

Table 3 presents a schema that summarizes the additional treatments that are necessary in presence of *break* and *continue* statements. We have not found any public report for any language that describes how to transform these statements. This table is general enough as to be able to work with any number of combined (possibly nested) loops and *break/continue* statements with or without labels. The table includes the case where *break/continue* are included inside a nested loop (*loop2*). Note however that there could be other loops in the different codes inside the ellipses. In such a case, the treatment would be

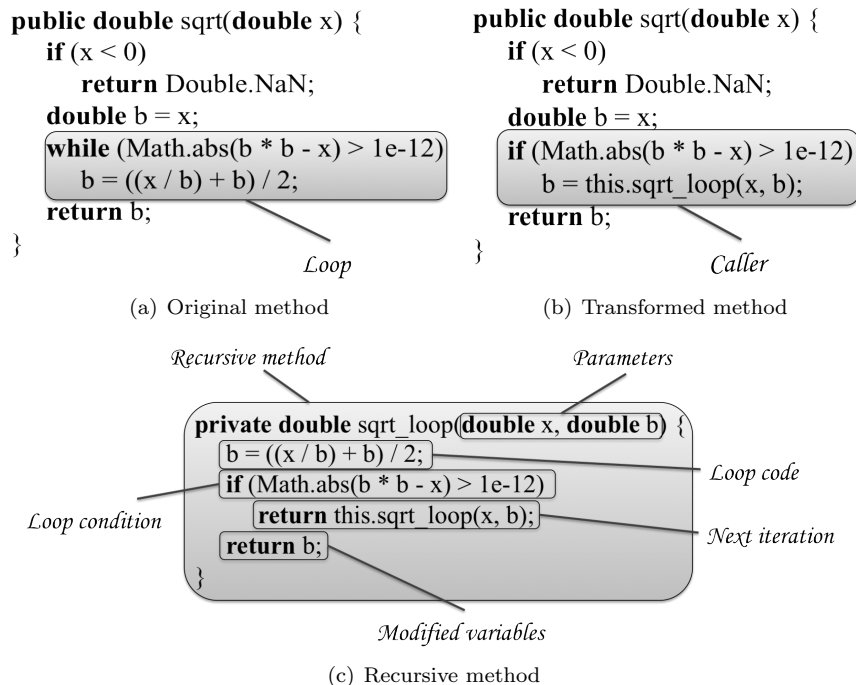


Figure 2: *while*-loop transformation

Algorithm 4 Use of the statement *continue* with a label

```

1: public void substr() {
2:   String searchMe = "Look for a substring in me";
3:   String substring = "sub";
4:   int max = searchMe.length() - substring.length();
5:   boolean foundIt = false;
6:   test:
7:   for (int i = 0; i <= max; i++) {
8:     int n = substring.length(), j = i, k = 0;
9:     while (n-- != 0)
10:      if (searchMe.charAt(j++) != substring.charAt(k++))
11:        continue test;
12:     foundIt = true;
13:     break test;
14:   }
15:   System.out.println(foundIt ? "Found it" : "Didn't find it");
16: }

```

extended exactly in the same way as it is described in the table for the case of two nested loops. Note also that *loop1* and *loop2* could be any kind of loop

(*while/do/for/foreach*), and thus the transformation should proceed in each kind as stated in Table 1. Essentially, right column in Table 3 performs the following steps:

- In the output of all recursive methods (i.e., in the fresh variable *result*), we always return in the first position (*result[0]*) an array of strings. This array of strings (referred to with the fresh variable *control*) contains two flags to indicate to the outer recursive methods (representing the outer loops) whether a *break* or *continue* statement that must be processed has been executed in the inner recursive methods (representing the inner loops). In the case that none of them were executed, or if they do not have a label associated, then *control* is *null*. The flags of *control* are the following:
 - *statement*: indicates whether a *break* or *continue* have been executed.
 - *label*: indicates the label associated with *statement*.
- After a call to a recursive method, the transformed code examines the output and depending on the values in the flags it behaves differently:
 - if *control* is *null*, then no *break/continue* statement has been executed or it has been already processed by the inner recursive methods. Hence, the execution proceeds normally. Otherwise, if the statement inside the control is *break* or *continue*, then
 - if the label of the current loop is different from the *label* returned, this means that a *break/continue* statement has been executed but it affects another outer loop. Therefore, we exit this method (without further executing anything else) and continue with the outer method. In this case, we have to output *control* to indicate the external methods that one of them should be broken or continued.
 - if the label of the current loop is equal to the *label* returned, and *statement* is equal to *break*, we exit this method and set *control* to *null* so that external methods proceed normally.
 - if the label of the current loop is equal to the *label* returned, and *statement* is equal to *continue*, this method ends its current iteration and continues with the next by performing a recursive call.

We can see the behavior of this table in a concrete example. Algorithm 5 implements a recursive method equivalent to Algorithm 4. This example is specially interesting because it combines the transformation of two nested loops (a *for*-loop and a *while*-loop) together with the use of a *break* and a *continue* statements that use a label of the same loop where they are declared (the *break* statement) and a label of an outer loop (the *continue* statement).

In this example, *loop1* and *loop2* correspond to *for*- and *while*-loops respectively. We can observe the analogy with the treatment described in Table 3. In particular, in Algorithm 4, inside the *for*-loop, the *break* statement (line 13) refers to this loop (due to the label *test*). Hence, in Algorithm 5 we substitute it by a return statement that just ends the execution of the loop (line 25).

Algorithm 5 Transformation of a *continue* with label

```
1: public void substr() {
2:   String searchMe = "Look for a substring in me";
3:   String substring = "sub";
4:   int max = searchMe.length() - substring.length();
5:   boolean foundIt = false;
6:   {
7:     int i = 0;
8:     if (i <= max)
9:       foundIt = substr_test(searchMe, substring, max, foundIt, i);
10:  }
11:  System.out.println(foundIt ? "Found it" : "Didn't find it");
12: }
13: private boolean substr_test(String searchMe, String substring, int max, boolean
    foundIt, int i) {
14:   int n = substring.length(), j = i, k = 0;
15:   if (n- != 0) {
16:     String[] control = substr_test_loop(searchMe, substring, n, j, k);
17:     if (control != null) {
18:       i++;
19:       if (i <= max)
20:         return substr_test(searchMe, substring, max, foundIt, i);
21:     }
22:   }
23: }
24: foundIt = true;
25: return foundIt;
26: }
27: private String[] substr_test_loop(String searchMe, String substring, int n, int j, int k) {
28:   if (searchMe.charAt(j++) != substring.charAt(k++))
29:     return new String[] { "continue", "test" };
30:   if (n- != 0)
31:     return substr_test_loop(searchMe, substring, n, j, k);
32:   return null;
33: }
```

Contrarily, inside the *while*-loop, the *continue* statement (line 11) refers to the outer loop (due again to the label *test*). Therefore, we substitute it by a return statement where we set the control flags to their corresponding values (line 29).

5. Handling exceptions

Exceptions interrupt the normal execution of a program. They can be thrown due to two reasons: explicitly thrown by the programmer (using the command *throw*) or implicitly thrown by executing an instruction (e.g., a division by zero, opening a file that does not exist, etc.). When an exception is thrown, there must exist a part of the code that catches this exception and handles it by executing some subroutines. Therefore, all methods in a program (and the methods generated by our transformation in particular) must decide what exceptions are caught by them, and thus throw the other exceptions to the caller of the method.

Table 4 presents a schema that summarizes the additional treatment necessary to handle exceptions. In the left column we have a method with two nested loops and four different exceptions that are thrown inside the inner loop. *Exception1* is caught inside the inner loop (*loop2*); *Exception2* is caught inside the outer loop (*loop1*); *Exception3* is caught outside both loops but inside the method; and *IOException*, which could be thrown due to the statement *file.read()*, is thrown out of the method.

The right column shows the transformed code. The transformation uses the same mechanism used for *break* and *continue*. In each method that represents a loop, it uses array *result* to output the exceptions that have occurred inside the method and must be caught by other method (i.e., the caller or any other previous one). Observe the transformed code for the inner loop (method *loop2*). This method catches all exceptions that were already caught by the original loop (i.e., *catch (Exception1 e)*). The other exceptions should be caught out of the method (i.e., *Exception2*, *Exception3* and *IOException*). Therefore, method *loop2* includes a *try-catch* that surrounds all statements. This *try-catch* catches any possible throwable exception because it catches a generic object *Throwable*. This is necessary to be able to catch all exceptions thrown by the programmer, and also those that can be thrown by the system. When an exception is caught, it is returned to the caller method inside an array called *control* with a flag “*throw*” to indicate that an exception that must be handled is being returned.

In the outer loop (method *loop1*), after the call to *loop2*, we check whether variable *control* is *null*. This is the way to know if the method should proceed normally, or it should handle the exception. Whenever *control != null*, it checks whether the exception is *Exception2*. In such a case, the exception is thrown, caught and handled by this method. Otherwise, it is thrown again to the caller of this method. This process is repeated inside each transformed method that represents a loop. Note in the example that the exception *IOException* is not handled by any method, and thus, it will be thrown by the initial method exactly as it happens in the original code.

There is a detail in the transformed *method* that is important to remark and explain. In Java there are two kinds of exceptions, named *checked* and *unchecked*. While *checked* exceptions must be always explicitly caught or thrown, *unchecked* exceptions are implicitly handled by the compiler (i.e., there is no need to include them in the *throws* statement). Moreover, all *unchecked* exceptions inherit from *RuntimeException* or *Error*. Therefore, in the case that (i) an exception reaches the transformed *method* (thus, it was not handled inside any loop), (ii) the exception is not handled by this method (it is not *Exception3*), and also, (iii) it does not appear in the *throws* section (it is not *IOException*), then this exception is necessarily an *unchecked* exception and must be thrown as such. To allow the compiler to catch an *unchecked* exception (e.g., *NullPointerException* in *file.read()* if *file* is *null*), the transformation must use a casting before throwing it (in the figure, *throw (RuntimeException) throwable* and *throw (Error) throwable*). This ensures that all checked and unchecked exceptions are caught.

6. Treatment of recursion and the statements *return* and *goto*

In this section we explain what happens when the loop contains recursive calls, and how to treat commands *return* and *goto*. We start explaining how to transform loops with recursive calls. Roughly, recursion does not need special treatment. Recursive calls are treated as any other statement in the loop, as it is shown in the programs in Table 5. In the iterative version of the loop we have three recursive calls (numbered 1, 2 and 3). Recursive calls 1 and 3 are outside the loop. They both are executed only once before and after the loop respectively in the original code. Analogously, they are executed before and after the transformed code associated with the loop. Therefore, trivially, they do not affect the transformation and remain unchanged. Contrarily, recursive call 2 is done inside the loop and, thus, it will be executed in every iteration of the loop. The transformed code places the recursive call inside the new recursive method generated (`loop`). Each activation of this method corresponds to one iteration of the original loop. Note that the recursive call to `function` is treated as any other statement. If it produces a change in the state of the loop (like, e.g., an assignment), it will affect the rest of the code in the same way in both the original and the transformed code.

The transformation of *return* statements is very similar to the one for *break* statements. Both statements force the control to stop the execution, leave the current loop, and continue the execution in another point of the code. We illustrate the transformation of *return* with Table 6. The code in Table 6 shows a *return* inside a nested loop. Therefore, it forces the execution to leave both loops and exit method `function`. Observe that the transformed code is completely analogous to the one for *break* statements. The transformation converts the *return* in rhombus 0 to the *return* in rhombus 3. Observe that we use a flag with the string "`return`" to indicate the outer loops that a *return* statement was executed. Then, all other methods check the flag and propagate the *return* statement (rhombus 2) until it reaches the original method (the caller). The caller removes the flag and returns the same value as in the original loop (rhombus 1).

Our transformation does not consider the use of *goto* statements. Note that, even though *goto* is a reserved word in Java, it is not used as a language construct, and thus, in practice, Java does not have *goto*.³ If we had a *goto* statement inside a transformed loop, three situations would be possible: the *goto* statement points to a label inside (1) the same loop; (2) an inner loop; or (3) an outer loop. In the first case the *goto* statement can remain unchanged in the transformed code, and the transformation will perfectly work. Cases 2 and 3 are not handled by our transformation. However, in the third case, there

³The *goto* command was included in the list of Java's reserved words because, if it was added to the language later on, existing code that used the word *goto* as an identifier (variable name, method name, etc.) would break. But because *goto* is a keyword, such code will not even compile in the present, and it remains possible to make it actually do something later on, without breaking existing code.

can exist an equivalence between a *goto* statement and a labeled *break/continue* statement. In such cases the transformation of the *goto* statement would correspond to the transformation of a labeled *break/continue* statement.

7. Implementation as a Java library, optimizations and empirical evaluation

All the transformations described in this paper have been implemented as a Java library called *loops2recursion*. This library consists of approximately 3,000 lines of Java code, and it is publicly available at: <http://www.dsic.upv.es/~jsilva/loops2recursion/>.

The library contains the implementation of all the individual algorithms needed to transform each kind of Java loop, and it also contains generic code able to parse a whole Java file or project. Roughly, it parses the source code and it automatically detects and transforms all loops in the program.

The transformation can be used, e.g., as follows:

```
import loops2recursion;
(...)
FileTransformer ft = new FileTransformer(sourceFile, targetFile);
ft.openFile();
ft.showCode();
ft.loops2recursion();
ft.showCode();
ft.saveFile();
```

Observe that an object of type `FileTransformer` is created to transform a file. Then, we can invoke several methods provided by the library to manipulate this object. In the example, we first open the source file (`openFile`), next we show in the screen the code in the source file (`showCode`), then we perform the transformation (`loops2recursion`) and show the transformed code (`showCode`). Finally, we save the transformed code in the target file (`saveFile`).

7.1. Optimizations

The code in Table 1 is generic. In some specific cases, this code can be optimized. The following optimizations are not needed to understand the transformation, but they should be considered when implementing it (all of them are implemented by *loops2recursion*):

1. **Do not update unmodified variables.**

If the loop does not modify a variable (e.g., it is only read), then do not return this variable in the recursive method. We can see an example in Algorithm 3 where variable x is not modified during the execution of method *sqrt_loop* and thus we do not return it (line 13).

2. **Variable uniformity.**

If all variables returned in the recursive method are of the same type, then the returned array should be of that type (e.g., `int[]`). We can see

an example in Algorithm 5 where the method *substr_test_loop* returns an array of strings (line 27).

3. **Avoid returning an array when possible.**

When the transformed method only returns a single variable, then the array is unnecessary and only the variable should be returned. We can see an example in Algorithm 3 where we only return variable *b* (line 13).

4. **Throw the exceptions (do not catch them) if there are no variables that should be updated.**

In the generated code, exceptions are handled to update the variables when returning from the recursive method. If there are no variables to update, then it is not necessary to catch the exceptions.

5. **Avoid the external block when possible.**

The transformation generates an external block to avoid that variables declared inside the generated code exist after it. If there are no declared variables, then the block is unnecessary. We can see an example in Algorithm 3 where variable *result* is not necessary because we can directly assign the return value to variable *b* (line 6).

6. **Remove unreachable code.**

The transformation can generate unreachable code when the last statement of the loop is a *break/continue/return/throw*. In that case, the code generated after this sentence is unreachable and can be removed. We can see an example in Algorithm 5 where method *sqr.test* is the transformation of a *for*-loop. In this example the transformation would normally insert the code to execute the *next iteration* at the end of the method, but since the last statement of the loop is a *break*, then this code would be unreachable and it has been removed (line 25).

7. **Remove the treatment for exceptions and *break/continue/return* when it is unnecessary.**

The code generated to treat exceptions should be generated only in the case that an exception can occur. The same happens with the code generated for *break/continue/return*. We can see an example in Algorithm 5 where method *substr_test* directly performs the *next iteration* after checking that *control != null* (lines 18-21). The transformation can take this decision by analyzing method *substr_test_loop* where *control != null* can only occur when the *continue* statement has been executed.

7.2. Empirical evaluation

We conducted several experiments to empirically evaluate both the transformation and the transformed code using the library *loops2recursion*. These experiments provide a precise and quantitative idea of the performance of the transformation.

For the evaluation, we selected a set of benchmarks from the Java Grande benchmarks suite [?]. A Java Grande application is one that uses large amounts of processing, I/O, network bandwidth, or memory. They include not only applications in science and engineering but also, for example, corporate databases

and financial simulations. Specific information about the benchmarks including the source code can be found at: http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

We designed the experiment from a set of 14 Java Grande benchmarks. Firstly, we automatically transformed the *loops2recursion* library by self-application (i.e., all loops were translated to recursive methods). Then, we transformed the 14 benchmarks with the two versions of the library (the original, and the transformed one) and executed all of them several times to compare the results. Effectively, all transformed benchmarks were equivalent to their original versions.

In a second phase of the experiment, we evaluated the performance. We strictly followed the methodology proposed in [?]. Each benchmark was repeatedly executed 1000 times in 100 different Java Virtual Machine (JVM) invocations (100.000 executions in total). To ensure real independence, the first iteration was always discarded (to avoid influence of dynamically loaded libraries persisting in physical memory, data persisting in the disk cache, etc.). From the 999 remaining iterations we retained 50 measurements per JVM invocation when steady-state performance was reached, i.e., once the coefficient of variation (CoV)⁴ of the 50 iterations falls below a preset threshold of 0.01. Because 0,01 was difficult to reach for some benchmarks, we took the minimum CoV of 50 iterations in a row somewhere in between the window of 999 iterations. Then, for each JVM invocation, we computed the sum of the 50 benchmark iterations under steady-state performance. This produced one statistical value. With 100 JVM invocations, we obtained 100 statistical values. Finally, we computed the 0.99 confidence interval across the computed values from the different 100 JVM invocations.

This process was repeated for the 30 benchmarks (14 iterative versions + 14 recursive versions + 2 versions of *loops2recursion*). Iterative and recursive versions were executed interlaced, so that the impact of any possible variability in the overall system performance was minimized. This produced the set of measures that is shown in Table 7. Here, we use the notation $[_a \ b \ c]$ that represents a symmetric 0.99 confidence interval between a and c with center in b . Each column in the tables has the following meaning: **Benchmark** is the name of the benchmark. **Transf. Time** is the total amount of milliseconds needed by *loops2recursion* to produce the recursive version from the iterative version. Here, **Open** is the time needed to open all files of the target project, **Process** is the time needed to transform all loops in these files to recursion, and **Save** is the time needed to save all transformed files. **Loops** is the number of different loops in the source code. **Loops Executed** is the number of loops executed (possibly with repetitions, specially in the case of nested loops). **Iterations Executed** is the number of loop iterations performed during the execution. The execution time (**ET**) is measured in milliseconds and represents the execution time of the whole benchmark. **LOT** stands for loop overhead time, it is measured

⁴CoV is defined as the standard deviation s divided by the mean \bar{x} .

in milliseconds, and it represents the time used to execute all loops without considering the time used by the body of the loops. That is, it just measures the time spent in a loop to control the own loop. This was calculated by inserting the command `System.nanoTime()`, which returns the current time in nanoseconds, immediately before and after the loop, immediately after entering the iteration, and immediately before leaving the iteration.

Example 7.1. Consider the following code that has been instrumented to measure its LOT:

```

t1=System.nanoTime();
while (cond){
    t2=System.nanoTime();
    body;
    t3=System.nanoTime();
}
t4=System.nanoTime();

```

If we assume that the loop performs 0 iterations, then LOT is $t4-t1$.

If we assume that the loop performs $n > 0$ iterations, then LOT is calculated as follows:

```

EntryTime = t2 - t1;  PrepareNextIterationTime = t2 - t3;  ExitTime
= t4 - t3;
LOT = EntryTime + ((n-1) * PrepareNextIterationTime) + ExitTime

```

Note that LOT is the appropriate measure to know what is the real speedup reached with the transformation. ET is not a good indicator because it depends on the body of the loops, but LOT is independent of the size and content of the loops.

For the recursive versions, LOT is calculated in a similar way. For instance, in the While-loop row of Table 1:

- `t1=System.nanoTime()`; is placed immediately after Code 1,
- `t2=System.nanoTime()`; is placed immediately before Code 2,
- `t3=System.nanoTime()`; is placed immediately after Code 2,
- `t4=System.nanoTime()`; is placed immediately before Code 3.

Rec/Iter measures the speedup/slowdown of the transformed code with respect to the original code. It is computed as $100 \times (\text{Recursive time} / \text{Iterative time})$.

The results for LOT have been normalized in a chart shown in Figure 3(a). Thanks to the confidence intervals computed for the iterative and recursive versions we can extract some statistical conclusions. In 3 out of 15 experiments, the confidence intervals overlap, thus no statistical conclusion can be extracted. In 12 out of 15 experiments confidence intervals are disjoint. In 11 experiments,

we can ensure with a confidence level of 99% that the recursive version is less efficient than the iterative version of the loops. Finally, in 1 experiment we can ensure with a confidence level of 99% that the recursive version is more efficient than the iterative version. Looking at Table 7, we see that the differences have been quantified and they are small (around 10%).

Initially, this was the end of our experiments. But then, we also looked at the results obtained for the ET. These results have been normalized in the chart shown in Figure 3(b). This figure reveals something unexpected: Sometimes the recursive version of the benchmarks is statistically (with 99% confidence level) more efficient than the iterative version. One could think that this does not make sense, because all the code except LOT is exactly the same in the iterative version and in the recursive version. But it is the same *before compilation*. After compilation it is different due to the optimizations done by Java, which are different in both versions due to the presence of the transformed code. In order to prove this idea, we repeated all the experiments in interpreted-only mode using the ‘-Xint’ JVM option. This flag disables the just-in-time (JIT) compilation in the JVM so that all bytecodes are executed by the interpreter. Disabling JIT compilation avoids many Java optimizations performed on code that significantly speedup recursion. The results obtained for ET with JIT compilation disabled are shown in Figure 3(c). The difference is clear. Without JIT compilation, iteration is more efficient. With JIT compilation iteration and recursion are similar, and sometimes recursion is even better (statistically). This confirms that in some cases, the library can increase code efficiency. However, in Java, our transformation has limited improvement due to the fact that the JVM does not support tail recursion optimization. This raises the point whether targeting another language the transformed code would produce better results. This part seems to be certainly interesting and it could open a line of research to decide in what cases the compiler should transform iterative loops to recursive methods to produce more efficient code. We left this research for future work.

8. Conclusions

Before we started the implementation of our library, we thought that compiling loops produces more efficient object code than compiling recursive methods. Therefore, we implemented our library not for efficiency reasons, but to gain other advantages of recursion that are useful in many situations such as, e.g., debugging, verification or memory hierarchies optimization.

The lack of an automatic transformation from loops to recursion is surprising, but it is even more surprising that no public report exists that describes how to implement this transformation. Hence, in this article we solve both problems. First, we describe and exemplify the program transformation step by step. Each kind of Java loop, namely *while/do/for/foreach*, needs a special treatment that is illustrated. Moreover, we provide a complete treatment of exceptions, *return*, and *break* and *continue* statements that can be parameterized with labels. Second, we present an implementation that is:

Complete. It accepts the whole Java language.

Efficient. Both the transformation and the generated code are scalable as demonstrated by our empirical evaluation. The generated code implements all the optimizations and it is tail-recursive. This property is missing in other implementations (e.g., [?]), and it is important not only for the sake of efficiency. Concretely, an infinite loop can only be translated to equivalent recursive methods if they are tail-recursive. Otherwise, the transformed code would produce a stack memory overflow not occurring in the original code.

Automatic. The transformation only needs to input a source program and it automatically searches for the loops and transforms them.

Public. The implementation and the source code is open and free.

The transformation has been described in such a way that it can be easily adapted to any other language where recursion and iteration exist.

After the empirical evaluation we discovered that the recursive version of a loop is sometimes more efficient than the original loop. This is something that must be further investigated to know the causes, because we could produce a transformation that systematically exploits opportunities to improve efficiency. For instance, if we determine that some kind of loops are more efficient if they are implemented as recursion, then a compiler could decide to transform loops to recursion or vice-versa before compiling them. We left this study for future work.

	Iterative version	Recursive version	
		Caller	Recursive method
While-loop	<pre> Code 1 while (cond) { Code 2 } Code 3 </pre>	<pre> Code 1 if (cond) { Object[] result = loop(1); } Code 3 </pre>	<pre> private Object[] loop(1') { Code 2 if (cond) return loop(1); return new Object[] { 1 }; } </pre>
Do-loop	<pre> Code 1 do { Code 2 } while (cond); Code 3 </pre>	<pre> Code 1 { Object[] result = loop(1); } Code 3 </pre>	<pre> private Object[] loop(1') { Code 2 if (cond) return loop(1); return new Object[] { 1 }; } </pre>
For-loop	<pre> Code 1 for (ini; cond; upd) { Code 2 } Code 3 </pre>	<pre> Code 1 { ini; if (cond) { Object[] result = loop(1); } } Code 3 </pre>	<pre> private Object[] loop(1') { Code 2 upd; if (cond) return loop(1); return new Object[] { 1 }; } </pre>
Foreach-loop (Array version)	<pre> Code 1 for (type elem : elems) { Code 2 } Code 3 </pre>	<pre> Code 1 if (0 < elems.length) { Object[] result = loop(1, elems, 0); } Code 3 </pre>	<pre> private Object[] loop(1', type[] elems, int index) { type elem = elems[index]; Code 2 index++; if (index < elems.length) return loop(1, elems, index); return new Object[] { 1 }; } </pre>
Foreach-loop (Iterable version)	<pre> Code 1 for (type elem : elems) { Code 2 } Code 3 </pre>	<pre> Code 1 { Iterator<type> iter = elems.iterator(); if (iter.hasNext()) { Object[] result = loop(1, iter); } } Code 3 </pre>	<pre> private Object[] loop(1', Iterator<type> iter) { type elem = iter.next(); Code 2 if (iter.hasNext()) return loop(1, iter); return new Object[] { 1 }; } </pre>

Step	Correspondence with Figure 1
	Figure 1(b)
1) Substitute the loop by a call to the recursive method	Caller
1.1) If the loop condition is satisfied	Loop condition
1.1.1) Perform the first iteration	First iteration
1.2) Catch the variables modified during the recursion	Modified variables
1.3) Update the modified variables	Updated variables
	Figure 1(c)
2) Create the recursive method	Recursive method
2.1) Define the method's parameters	Parameters
2.2) Define the code of the recursive method	
2.2.1) Include the code of the original loop	Loop code
2.2.2) If the loop condition is satisfied	Loop condition
2.2.2.1) Perform the next iteration	Next iteration
2.2.3) Otherwise return the modified variables	Modified variables

Table 2: Steps of the *while*-loop transformation

Iterative version	Recursive version
<pre> Code loop1: do { Code 1 loop2: do { Code A break loop1; Code B continue loop1; Code C } while (cond2); Code 2 } while (cond1); Code </pre>	<pre> private Object[] loop1([1]) { Code 1 { Object[] result = loop2([1]); String[] control = (String[]) result[0]; [2] if (control != null) { String statement = control[0]; if (statement.equals("break") statement.equals("continue")) { String label = control[1]; if (label.equals("loop1") == false) return new Object[] { control, [1] }; if (statement.equals("break")) return new Object[] { null, [1] }; if (cond1) return loop1([1]); return new Object[] { null, [1] }; } } } Code 2 if (cond1) return loop1([1]); return new Object[] { null, [1] }; } private Object[] loop2([1]) { Code A return new Object[] { new String[] { "break", "loop1" }, [1] }; Code B return new Object[] { new String[] { "continue", "loop1" }, [1] }; Code C if (cond2) return loop2([1]); return new Object[] { null, [1] }; } </pre>

Table 3: Schema showing the transformation of *break* and *continue* statements

Iterative version	Recursive version	
<pre> void method() throws IOException { try { loop1: do { try { loop2: do { try { file.read(); throw new Exception1(); throw new Exception2(); throw new Exception3(); } catch (Exception1 e) { Code 1 } } while (cond2); } catch (Exception2 e) { Code 2 } } while (cond1); } catch (Exception3 e) { Code 3 } } </pre>	<pre> void method() throws IOException { try { Object[] result = loop1(1); Object[] control = (Object[]) result[0]; if (control != null) { String statement = (String) control[0]; if (statement.equals("throw")) { Throwable throwable = (Throwable) control[1]; if (throwable instanceof Exception3) throw (Exception3) throwable; if (throwable instanceof IOException) throw (IOException) throwable; if (throwable instanceof RuntimeException) throw (RuntimeException) throwable; throw (Error) throwable; } } catch (Exception3 e) { Code 3 } } } private Object[] loop2(1) { try { try { file.read(); throw new Exception1(); throw new Exception2(); throw new Exception3(); } catch (Exception1 e) { Code 1 } } if (cond2) return loop2(1); return new Object[] { null, 1 }; } catch (Throwable throwable) { Object[] control = new Object[] { "throw", throwable }; return new Object[] { control, 1 }; } } </pre>	<pre> private Object[] loop1(1) { try { try { Object[] result = loop2(1); Object[] control = (Object[]) result[0]; if (control != null) { String statement = (String) control[0]; if (statement.equals("throw")) { Throwable throwable = (Throwable) control[1]; if (throwable instanceof Exception2) throw (Exception2) throwable; throw throwable; } } catch (Exception2 e) { Code 2 } } if (cond1) return loop1(1); return new Object[] { null, 1 }; } catch (Throwable throwable) { Object[] control = new Object[] { "throw", throwable }; return new Object[] { control, 1 }; } } </pre>

Table 4: Handling exceptions

Iterative version	Recursive version	
	Caller	Recursive method
<pre> public void function() { ... function(); while (cond) { ... function(); ... } function(); ... } </pre>	<pre> public void function() { ... ① function(); ... if (cond) { loop(); } ... ③ function(); ... } </pre>	<pre> private Object loop() { ... ② function(); ... if (cond) return loop(); return null; } </pre>

Table 5: Treatment of a recursive method

Iterative version	Recursive version		
	Caller	Recursive method (Outer loop)	Recursive method (inner loop)
<pre> public int function() { ... while (cond1) { ... while (cond2) { ... if (cond3) return expr; } ... } ... } </pre>	<pre> public int function() { ... if (cond1) { Object[] control = loop1(); if (control != null) { String statement = control[0]; if (statement.equals("return")) return (Integer) control[1]; } } ... } </pre>	<pre> private Object[] loop1() { ... if (cond2) { Object[] control = loop2(); if (control != null) { String statement = control[0]; if (statement.equals("return")) return control; } } ... if (cond1) return loop1(); return null; } </pre>	<pre> private Object[] loop2() { ... if (cond3) return new Object[] { "return", expr }; ... if (cond2) return loop2(); return null; } </pre>

Table 6: Treatment of *return* statements

Benchmark	Transf. Time (ms)			Loops	Loops Executed	Iterations Executed
	Open	Process	Save			
loops2recursion	954	206	213	117	52580	153729
JGFArithBench	348	26	13	24	144	12590
JGFAssignBench	337	22	21	20	120	10389
JGFCastBench	161	16	6	8	48	4200
JGFCreateBench	337	27	17	34	520	44860
JGFExceptionBench	129	20	6	6	36	3148
JGFLoopBench	135	16	3	6	36	3159
JGFMathBench	419	45	33	60	360	31498
JGFMethodBench	212	20	9	16	96	8320
JGFCryptBench	146	20	8	10	277	4447
JGFFFTBench	158	17	8	14	4122	24570
JGFHeapSortBench	135	14	7	5	1502	12939
JGFSeriesBench	141	18	24	4	2005	1996013
JGFSORBench	155	21	4	7	10101	990102
JGFSparseMatmultBench	115	18	4	5	204	203200
Average	258,8	33,73	25,07	22,4	4810,07	233544,27

(a) Transformation performance

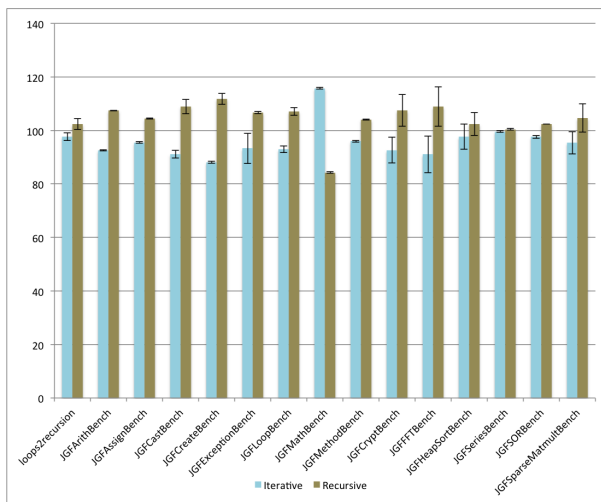
Benchmark	Iterative ET (ms)	Recursive ET (ms)	Rec/Iter ET (%)
loops2recursion	[7083,08 7195,94 7308,81]	[7023,41 7156,43 7289,46]	99,45
JGFArithBench	[144,72 144,96 145,20]	[167,49 167,59 167,69]	115,61
JGFAssignBench	[113,55 113,98 114,41]	[119,67 119,83 119,99]	105,14
JGFCastBench	[47,19 48,57 49,95]	[52,10 53,89 55,68]	110,94
JGFCreateBench	[13057,33 13155,91 13254,49]	[12422,11 12581,79 12741,48]	95,64
JGFExceptionBench	[819,71 824,62 829,52]	[9288,23 9292,87 9297,51]	1126,93
JGFLoopBench	[39,07 39,96 40,84]	[42,79 43,70 44,61]	109,34
JGFMathBench	[1616,55 1617,75 1618,95]	[985,08 986,07 987,06]	60,95
JGFMethodBench	[94,88 95,23 95,58]	[99,26 99,38 99,49]	104,36
JGFCryptBench	[281,63 285,11 288,59]	[287,46 291,65 295,83]	102,29
JGFFFTBench	[543,14 560,15 577,15]	[557,91 575,31 592,72]	102,71
JGFHeapSortBench	[405,34 414,09 422,84]	[406,91 414,92 422,93]	100,20
JGFSeriesBench	[44288,14 44363,17 44438,21]	[44033,69 44197,43 44361,16]	99,63
JGFSORBench	[7384,92 7426,96 7469,01]	[7466,40 7472,71 7479,01]	100,62
JGFSparseMatmultBench	[2149,15 2204,96 2260,77]	[2208,14 2273,53 2338,92]	103,11
Average	5232,76	5715,14	169,13

(b) Transformed code performance: Execution time results

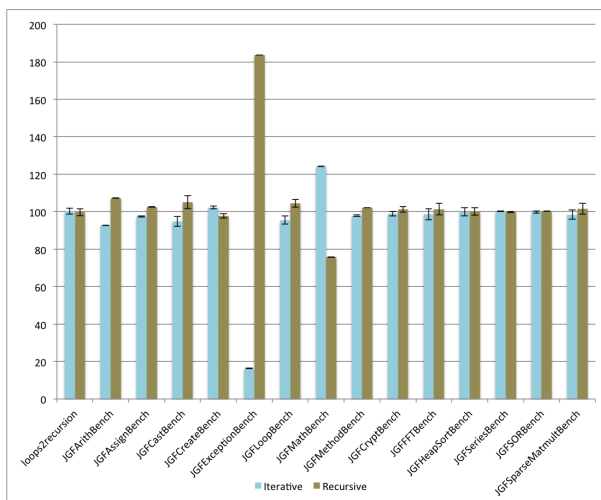
Benchmark	Iterative LOT (ms)	Recursive LOT (ms)	Rec/Iter LOT (%)
loops2recursion	[1728,64 1754,99 1781,35]	[1804,54 1840,76 1876,98]	104,89
JGFArithBench	[42,44 42,52 42,60]	[49,31 49,37 49,42]	116,10
JGFAssignBench	[35,49 35,60 35,71]	[38,87 38,94 39,02]	109,38
JGFCastBench	[14,45 14,67 14,90]	[17,11 17,55 17,98]	119,57
JGFCreateBench	[96,97 97,42 97,87]	[121,28 123,59 125,90]	126,87
JGFExceptionBench	[10,60 11,29 11,97]	[12,86 12,90 12,95]	114,32
JGFLoopBench	[11,28 11,44 11,59]	[13,00 13,17 13,35]	115,19
JGFMathBench	[171,27 171,73 172,20]	[124,68 125,04 125,41]	72,81
JGFMethodBench	[28,18 28,29 28,40]	[30,62 30,67 30,72]	108,42
JGFCryptBench	[14,66 15,46 16,26]	[16,94 17,94 18,94]	116,06
JGFFFTBench	[73,44 79,44 85,44]	[88,51 94,96 101,41]	119,54
JGFHeapSortBench	[45,78 48,08 50,38]	[48,33 50,41 52,49]	104,84
JGFSeriesBench	[3952,32 3965,49 3978,66]	[3985,82 4000,19 4014,57]	100,88
JGFSORBench	[1841,56 1852,23 1862,91]	[1943,41 1944,21 1945,01]	104,97
JGFSparseMatmultBench	[606,37 634,00 661,62]	[660,63 695,42 730,22]	109,69
Average	584,18	603,68	109,57

(c) Transformed code performance: LOT results

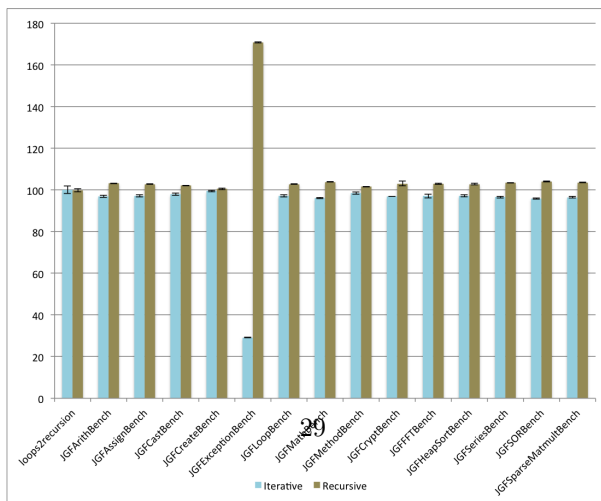
Table 7: Benchmark results



(a) LOT results



(b) ET results



(c) ET results with -Xint flag enabled

Figure 3: Normalized (scaled over 100) charts