Automatic Transformation of Iterative Loops into Recursive Methods☆

(extended version)

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.

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., [12, 15, 17]). 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 [14] 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 [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

☆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)

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.
*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*

Algorithm 1 Iterative loop with exceptions

```
public int example(int x) throws IOException {
    loop1:
    while (x < 10) {
        try {
            x = 42 / x;
        } catch (Exception e) { break loop1; }
        loop2:
        for (int i = 1; i < x; i++)
            if (x % i > 0);
                throw new Exception1();
            else continue loop1;
    }
    return x;
}
```

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 describe our transformation for all kinds of loops in Java (i.e., *while/do/for/foreach*). The transformation of each particular kind of loop is explained with an example. 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)

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

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. 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. Section 8 concludes. Finally, Appendix A provides a proof of correctness of the transformation.

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., [6, 9]).

On the practical side, there exist many different approaches to transform recursive functions into loops (see, e.g., [12, 15, 17]). 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 [14], theorem proving [16], verification [18] and termination analysis [5].

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 [5] 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 [20], 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 [18]. 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.

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.

[1]The 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`.

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 [20] 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* [7].

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) [13].

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:

$\boxed{1}$ 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).

$\boxed{1'}$ 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).

	Iterative version	Recursive version	
		Caller	Recursive method

While-loop	Code 1
	while (cond) {
	    Code 2
	}
	Code 3
		Code 1
		if (cond) {
		    Object[] result = loop($\boxed{1}$);
		    $\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			    Code 2
			    if (cond)
			        return loop($\boxed{1}$);
			    return new Object[] { $\boxed{1}$ };
			}

Do-loop	Code 1
	do {
	    Code 2
	} while (cond);
	Code 3
		Code 1
		{
		    Object[] result = loop($\boxed{1}$);
		    $\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			    Code 2
			    if (cond)
			        return loop($\boxed{1}$);
			    return new Object[] { $\boxed{1}$ };
			}

For-loop	Code 1
	for (ini; cond; upd) {
	    Code 2
	}
	Code 3
		Code 1
		{
		    ini;
		    if (cond) {
		        Object[] result = loop($\boxed{1}$);
		        $\boxed{2}$
		    }
		}
		Code 3
			private Object[] loop($\boxed{1^*}$) {
			    Code 2
			    upd;
			    if (cond)
			        return loop($\boxed{1}$);
			    return new Object[] { $\boxed{1}$ };
			}

Foreach-loop (Array version)	Code 1
	for (type elem : elems) {
	    Code 2
	}
	Code 3
		Code 1
		if (0 < elems.length) {
		    Object[] result = loop($\boxed{1}$, elems, 0);
		    $\boxed{2}$
		}
		Code 3
			private Object[] loop($\boxed{1^*}$,
			            type[] elems, int index) {
			    type elem = elems[index];
			    Code 2
			    index++;
			    if (index < elems.length)
			        return loop($\boxed{1}$, elems, index);
			    return new Object[] { $\boxed{1}$ };
			}

Foreach-loop (Iterable version)	Code 1
	for (type elem : elems) {
	    Code 2
	}
	Code 3
		Code 1
		{
		    Iterator<type> iter = elems.iterator();
		    if (iter.hasNext()) {
		        Object[] result = loop($\boxed{1}$, iter);
		        $\boxed{2}$
		    }
		}
		Code 3
			private Object[] loop($\boxed{1^*}$,
			            Iterator<type> iter) {
			    type elem = iter.next();
			    Code 2
			    if (iter.hasNext())
			        return loop($\boxed{1}$, iter);
			    return new Object[] { $\boxed{1}$ };
			}

Table 1: Loops transformation taxonomy

$\boxed{2}$ contains for each object in the array `result` (which contains the same variables as $\boxed{1}$ and $\boxed{1'}$), 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 $\boxed{2}$ 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. In the rest of this section we explain the transformation of all kinds of loop. 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. We start by describing the transformation for *while*-loops, and then we describe the variations needed to adapt the transformation for *do/for/foreach*-loops.

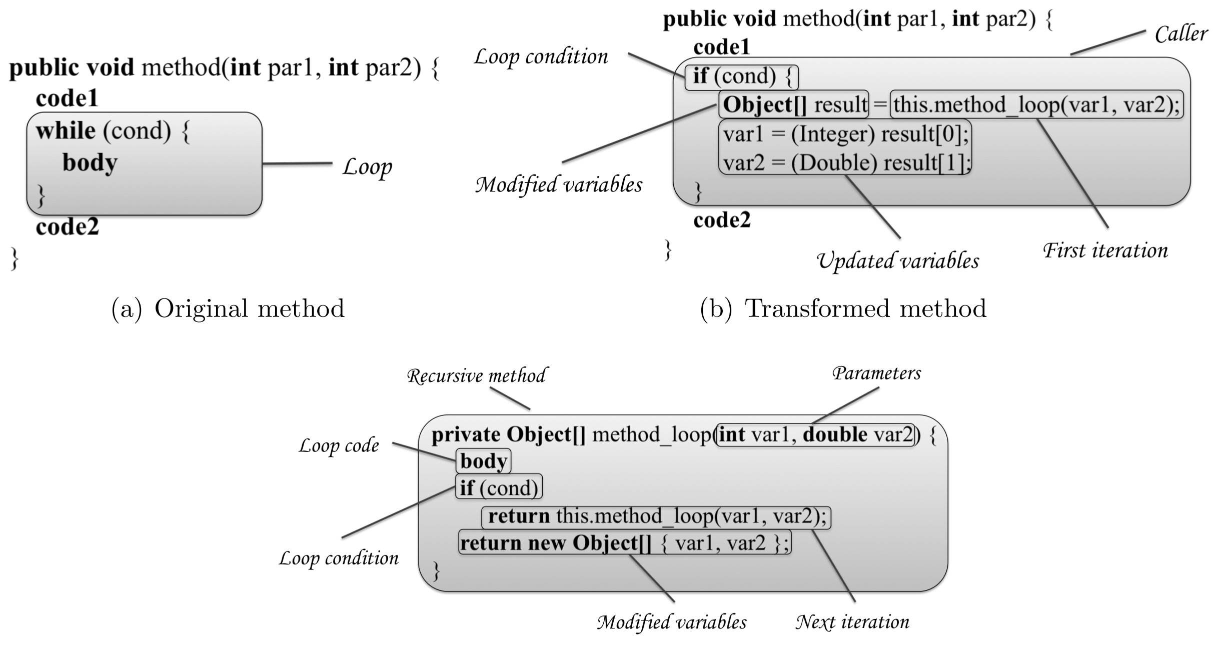
3.1. Transformation of while*-loops*

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.

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.



```
public void method(int par1, int par2) {
  code1
  while (cond) {
    body
  }
  code2
}
```

(a) Original method

```
public void method(int par1, int par2) {
  code1
  if (cond) {
    Object[] result = this.method_loop(var1, var2);
    var1 = (Integer) result[0];
    var2 = (Double) result[1];
  }
  code2
}
```

(b) Transformed method

```
private Object[] method_loop(int var1, double var2) {
  body
  if (cond)
    return this.method_loop(var1, var2);
  return new Object[] { var1, var2 };
}
```

(c) Recursive method

Figure 1: *while*-loop transformation

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

- **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 type[2], 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 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

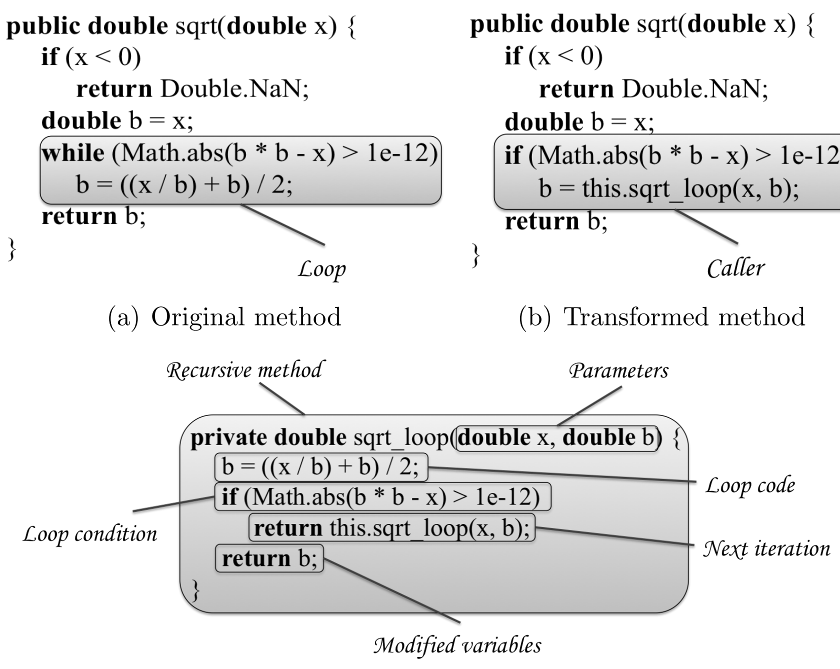
[2]In the case that the returned values are primitive types, then they are naturally encapsulated by the compiler in their associated primitive wrapper classes.

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.



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

(a) Original method

```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    if (Math.abs(b * b - x) > 1e-12)
        b = this.sqrt_loop(x, b);
    return b;
}
```

(b) Transformed method

```
private double sqrt_loop(double x, double b) {
    b = ((x / b) + b) / 2;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b);
    return b;
}
```

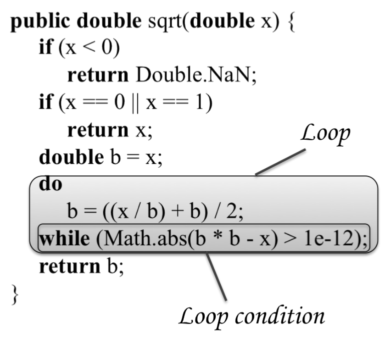
(c) Recursive method

Figure 2: *while*-loop transformation

3.2. Transformation of do*-loops*

do-loops behave exactly in the same way as *while*-loops except in one detail: The first iteration of the *do*-loop is always performed. In Figure 3 we can see an example of a *do*-loop.

This code obtains the square root value of variable x as the code in Algorithm 2. The difference is that, if variable x is either 0 or 1, then the method directly returns variable x, otherwise the loop is performed in order to calculate the square root. In order to transform

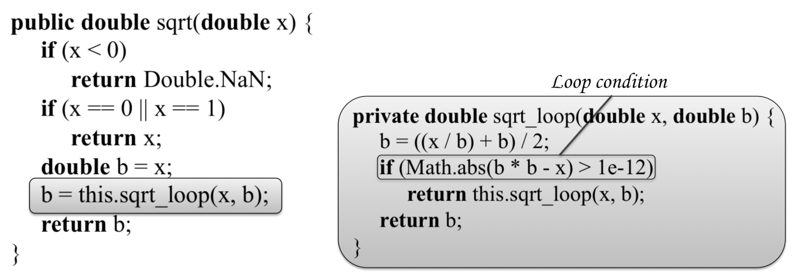


```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    if (x == 0 || x == 1)
        return x;
    double b = x;
    do
        b = ((x / b) + b) / 2;
    while (Math.abs(b * b - x) > 1e-12);
    return b;
}
```

Figure 3: do-loop

the *do*-loop into a recursive method, we can follow the same steps used in Table 2 with only one change: in step 1.1 the loop condition is not evaluated; instead, we only need to add a new code block to ensure that those variables created during the transformation are not available outside the transformed code.

Figure 4 illustrates the only change needed to transform the *do*-loop into a recursive method. Observe that in this example there is no need to introduce a new block, because the transformed code does not create new variables, but in the general case the block could be needed.



```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    if (x == 0 || x == 1)
        return x;
    double b = x;
    b = this.sqrt_loop(x, b);
    return b;
}
```

```
private double sqrt_loop(double x, double b) {
    b = ((x / b) + b) / 2;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b);
    return b;
}
```

(a) Recursive method caller (b) Recursive method

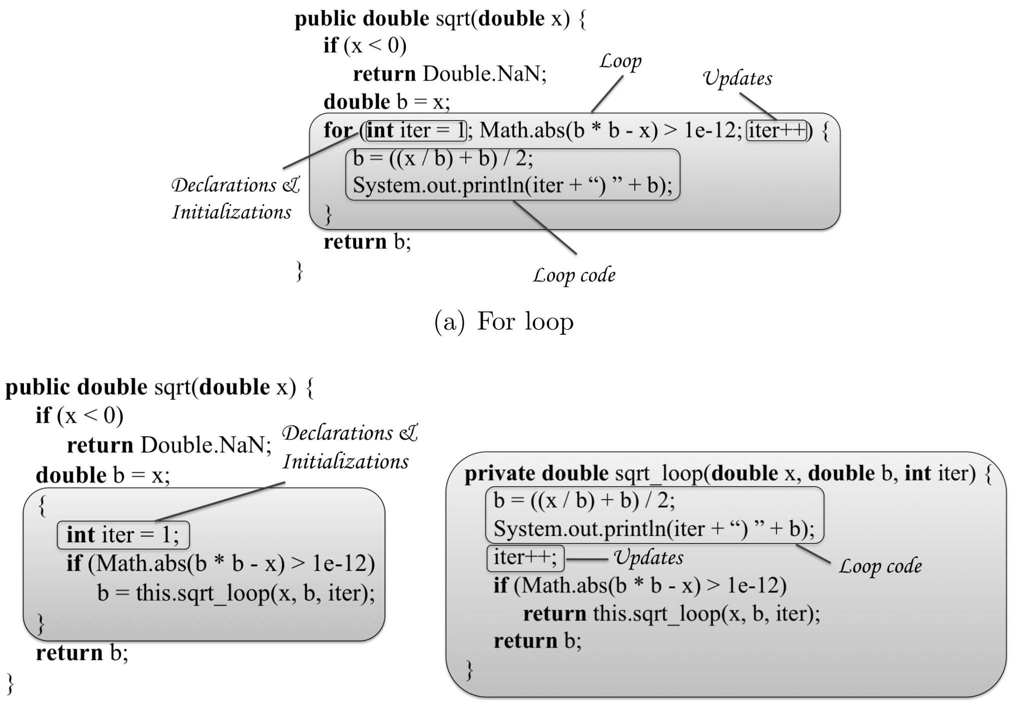
Figure 4: do-loop transformation

- **Add a new code block**
  Observe in Table 1, in column `Caller`, that, contrarily to *while*-loops, *do*-loops need to introduce a new *block* (i.e., a new scope). The reason is that there could exist variables with the same name as the variables created during the transformation (e.g., *result*). Hence, the new block avoids variable clashes and limits the scope of the variables created by the transformation.

3.3. Transformation of for*-loops*

One of the most frequently used loops in Java is the *for*-loop. This loop behaves exactly in the same way as the *while*-loop except in one detail: *for*-loops provide the programmer with a mechanism to declare, initialize and update variables that will be accessible inside the loop.

In Figure 5(a) we can see an example of a *for*-loop. This code obtains the square root value of variable x exactly as the code in Algorithm 2, but it also prints the approximation obtained in every iteration. We can see in Figure 5(b) and 5(c) the additional changes needed to transform the *for*-loop into a recursive method.



```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    for (int iter = 1; Math.abs(b * b - x) > 1e-12; iter++) {
        b = ((x / b) + b) / 2;
        System.out.println(iter + ") " + b);
    }
    return b;
}
```

Loop
Updates
Declarations & Initializations
Loop code

(a) For loop

```
public double sqrt(double x) {
    if (x < 0)
        return Double.NaN;
    double b = x;
    {
        int iter = 1;
        if (Math.abs(b * b - x) > 1e-12)
            b = this.sqrt_loop(x, b, iter);
    }
    return b;
}
```

Declarations & Initializations

```
private double sqrt_loop(double x, double b, int iter) {
    b = ((x / b) + b) / 2;
    System.out.println(iter + ") " + b);
    iter++;
    if (Math.abs(b * b - x) > 1e-12)
        return this.sqrt_loop(x, b, iter);
    return b;
}
```

(b) Recursive method caller

(c) Recursive method

Figure 5: for-loop transformation

As shown in Figure 5, in order to transform the *for*-loop into a recursive method, we can follow the same steps used in Table 2, but we have to make three changes:

- **Add a new code block**
  Exactly in the same way and with the same purpose as in *do*-loops.

- **Add the declarations/initializations at the beginning of the block**
  In the original method, those variables created during the *declaration* and *initialization* of the loop are only available inside it (and not in the code that follows the loop). We must ensure that these variables keep the same scope in the transformed code. This can be easily achieved with the new *block*. In the transformed code, those variables are declared and initialized at the beginning of the new *block*, and they are passed as arguments to the recursive method in every iteration to make them accessible inside it.

- **Add the updates between the loop code and the loop condition**
  In *for*-loops there exists the possibility of executing code between iterations. This code is usually a collection of *updates* of the variables declared at the beginning of the loop (e.g., in Figure 5(a) this code is `iter++`). However, this code could be formed by a series of expressions separated by commas that could include method invocations, assignments, etc. Because this *update* code is always executed before the condition of the loop, it must be placed in the recursive method between the *loop code* and the *loop condition*.

3.4. Transformation of foreach*-loops*

foreach-loops are specially useful to traverse collections of elements. In particular, this kind of loops traverses a given collection and it executes a block of code for each element. The transformation of a *foreach*-loop into a recursive method is different depending on the kind of collection that is traversed. In Java we can use *foreach*-loops either with *arrays* or *iterable* objects. We explain each transformation separately.

3.4.1. foreach*-loop used to traverse arrays*

An array is a composite data structure where elements have been sequentialized, and thus, they can be traversed linearly. We can see an example of a *foreach*-loop that traverses an array in Algorithm 4.

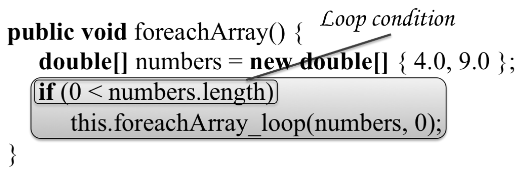
Algorithm 4 *foreach*-loop that traverses an array (iterative version)

```
public void foreachArray() {
    double[] numbers = new double[] { 4.0, 9.0 };
    for (double number : numbers) {
        double sqrt = this.sqrt(number);
        System.out.println("sqrt(" + number + ") = " + sqrt);
    }
}
```

This code computes and prints the square root of all elements in the array [4.0, 9.0]. Each individual square root is computed with Algorithm 2. The *foreach*-loop traverses the array sequentially starting in position *numbers[0]* until the last element in the array. The transformation of this loop into an equivalent recursive method is very similar to the transformation of a *for*-loop. However there are differences. For instance, *foreach*-loops lack of a counter. This can be observed in Figure 6 that implements a recursive method equivalent to the loop in Algorithm 4.

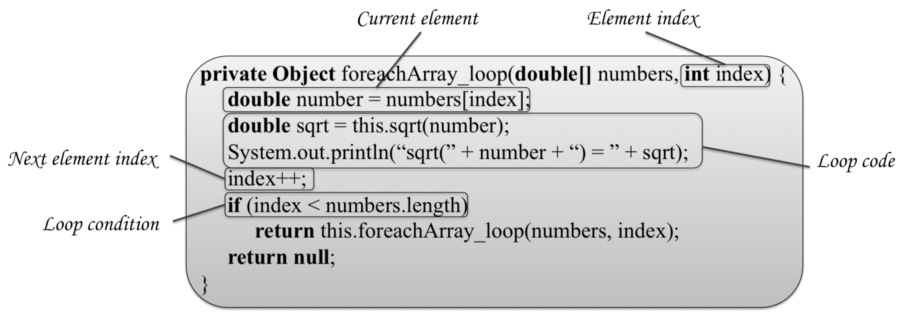
In Figure 6 we can see the symmetry with respect to the *for*-loop transformation. The only difference is the creation of a fresh variable that is passed as argument in the recursive method calls (in the example this variable is called *index*). This variable is used for:

- **Controlling whether there are more elements to be treated**
  A *foreach*-loop is only executed if the array contains elements. Therefore we need a



```
public void foreachArray() {
    double[] numbers = new double[] { 4.0, 9.0 };
    if (0 < numbers.length)
        this.foreachArray_loop(numbers, 0);
}
```

(a) Recursive method caller



```
private Object foreachArray_loop(double[] numbers, int index) {
    double number = numbers[index];
    double sqrt = this.sqrt(number);
    System.out.println("sqrt(" + number + ") = " + sqrt);
    index++;
    if (index < numbers.length)
        return this.foreachArray_loop(numbers, index);
    return null;
}
```

(b) Recursive method

Figure 6: *foreach*-loop transformation (Array version)

loop condition in the recursive method caller and another in the recursive method to know when there are no more elements in the array and thus finish the traversal. The later is controlled with a variable (*index* in the example) acting as a counter.

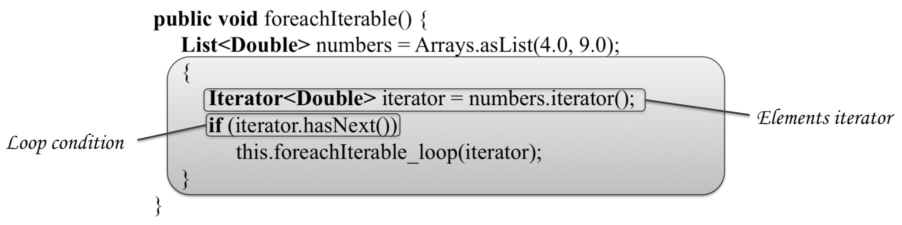
- **Obtaining the next element to be treated**
  During each iteration of the *foreach*-loop a variable called *number* is instantiated with one element of the array (line 3 of Algorithm 4). In the transformation this behavior is emulated by declaring and initializing this variable at the beginning of the recursive method. It is initialized to the corresponding element of the array by using variable *index*.

3.4.2. foreach*-loop used to traverse* iterable *objects*

A *foreach*-loop can be used to traverse objects that implement the interface *Iterable*. Algorithm 5 shows an example of a *foreach*-loop using one of these objects.

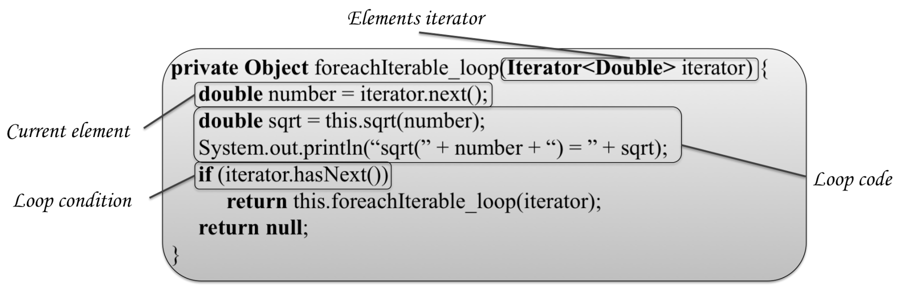
Algorithm 5 *foreach*-loop used to traverse an iterable object (iterative version)

```
public void foreachIterable() {
    List<Double> numbers = Arrays.asList(4.0, 9.0);
    for (double number : numbers) {
        double sqrt = this.sqrt(number);
        System.out.println("sqrt(" + number + ") = " + sqrt);
    }
}
```



```
public void foreachIterable() {
  List<Double> numbers = Arrays.asList(4.0, 9.0);
  {
    Iterator<Double> iterator = numbers.iterator();
    if (iterator.hasNext())
      this.foreachIterable_loop(iterator);
  }
}
```

(a) Recursive method caller



```
private Object foreachIterable_loop(Iterator<Double> iterator) {
  double number = iterator.next();
  double sqrt = this.sqrt(number);
  System.out.println("sqrt(" + number + ") = " + sqrt);
  if (iterator.hasNext())
    return this.foreachIterable_loop(iterator);
  return null;
}
```

(b) Recursive method

Figure 7: foreach-loop transformation (Iterable version)

This code behaves exactly in the same way as Algorithm 4 but using an *iterable* object instead of an array (*numbers* is an iterable object because it is an instance of class *List* that in turn implements the interface *Iterable*). The interface *Iterable* only has one method, called *iterator*, that returns an object that implements the *Iterator* interface. With regard to the interface *Iterator*, it forces the programmer to implement the *next*, *hasNext* and *remove* methods; and these methods allow the programmer to freely implement how the collection is traversed (e.g., the order, whether repetitions are taken into account or not, etc.). Therefore, the transformed code should use these methods to traverse the collection. We can see in Figure 7 a recursive method equivalent to Algorithm 5.

Observe that the transformed code in Figure 7 is very similar to the one in Figure 6. The only difference is the use of an *iterator* variable (instead of an integer variable) that controls the element of the collection to be treated. Note that method *next* of variable *iterator* allows us to know what is the next element to be treated, and method *hasNext* tell us whether there exist more elements to be processed yet.

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 6. This program takes two strings and decides whether the second is a substring of the first.

Algorithm 6 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: }
```

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 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*.

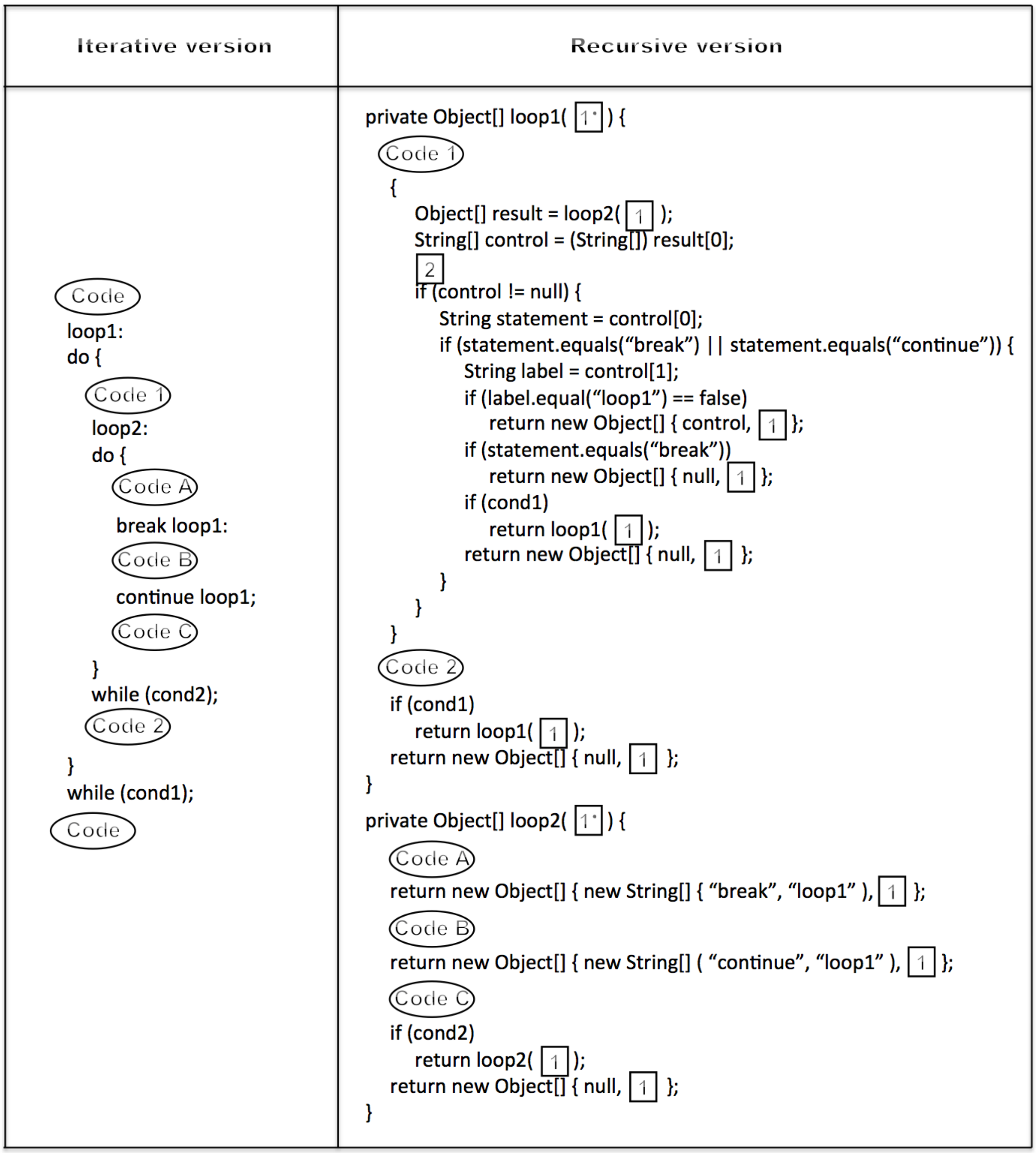
- After a call to a recursive method, the transformed code examines the output and depending on the values in the flags it behaves differently:
  - if *control* is *null*, then no *break/continue* statement has been executed or it has been already processed by the inner recursive methods. Hence, the 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 7 implements a recursive method equivalent to Algorithm 6. 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 6, inside the *for*-loop, the *break* statement (line 13) refers to this loop (due to the label *test*). Hence, in Algorithm 7 we substitute it by a return statement that just ends the execution of the loop (line 25). 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



Iterative version

```
Code
loop1:
do {
    Code 1
    loop2:
    do {
        Code A
        break loop1;
        Code B
        continue loop1;
        Code C
    }
    while (cond2);
    Code 2
}
while (cond1);
Code
```

Recursive version

```
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.equal("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 };
}
```

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

exceptions are caught by them, and thus throw the other exceptions to the caller of the method.

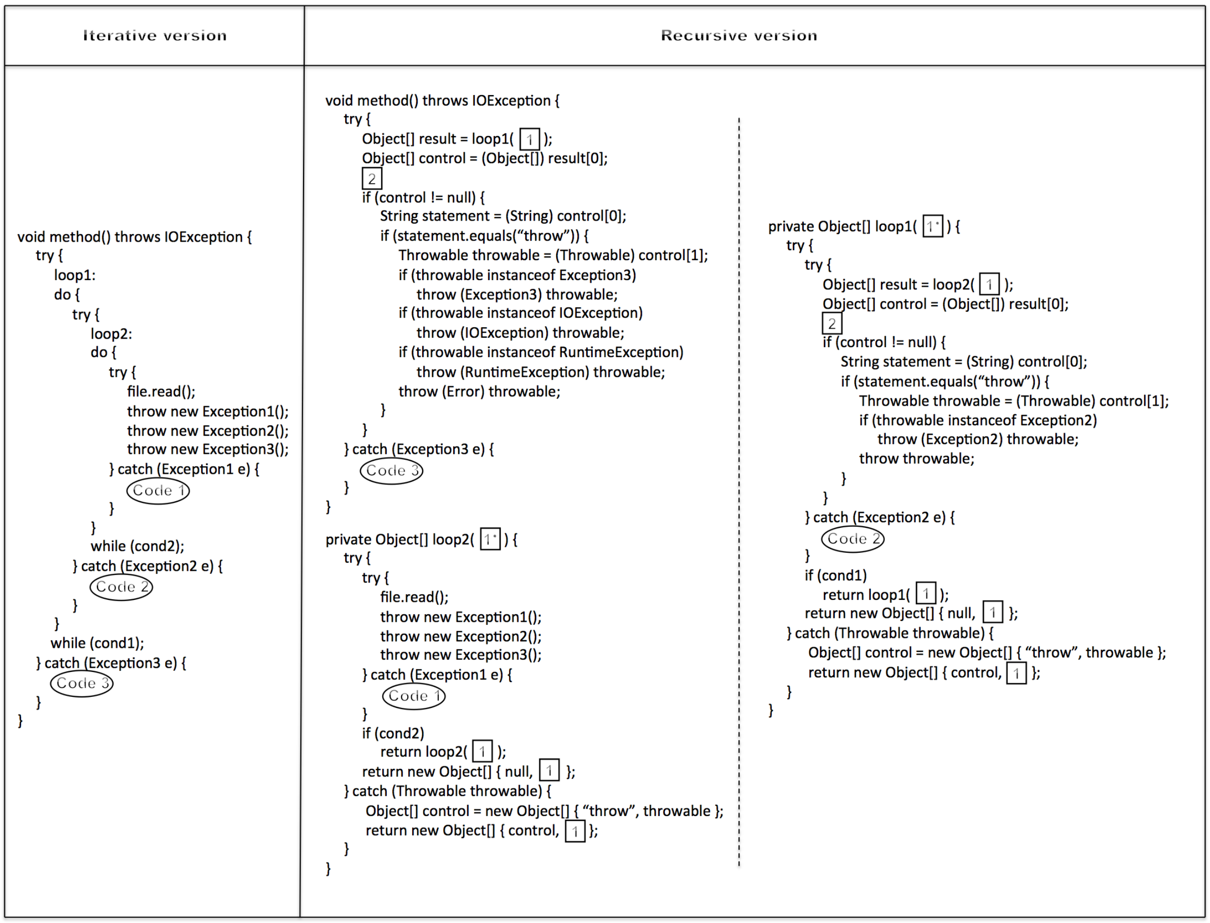
Table 4 presents a schema that summarizes the additional treatment necessary to handle exceptions. In the left column we have a method with two nested loops and four different exceptions that are thrown inside the inner loop. *Exception1* is caught inside the inner loop

Algorithm 7 Transformation of a *continue* with label

```
public void substr() {
    String searchMe = "Look for a substring in me";
    String substring = "sub";
    int max = searchMe.length() - substring.length();
    boolean foundIt = false;
    {
        int i = 0;
        if (i <= max)
            foundIt = substr_test(searchMe, substring, max, foundIt, i);
    }
    System.out.println(foundIt ? "Found it" : "Didn't find it");
}
private boolean substr_test(String searchMe, String substring, int max, boolean foundIt, int i) {
    int n = substring.length(), j = i, k = 0;
    if (n-- != 0) {
        String[] control = substr_test_loop(searchMe, substring, n, j, k);
        if (control != null) {
            i++;
            if (i <= max)
                return substr_test(searchMe, substring, max, foundIt, i);
            return foundIt;
        }
    }
    foundIt = true;
    return foundIt;
}
private String[] substr_test_loop(String searchMe, String substring, int n, int j, int k) {
    if (searchMe.charAt(j++) != substring.charAt(k++))
        return new String[] { "continue", "test" };
    if (n-- != 0)
        return substr_test_loop(searchMe, substring, n, j, k);
    return null;
}
```

(*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



Iterative version

```
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)
   }
}
```

Recursive version

```
void method() throws IOException {
   try {
      Object[] result = loop1( [1] );
      Object[] control = (Object[]) result[0];
      [2]
      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] };
   }
}
```

```
private Object[] loop1( [1'] ) {
   try {
      try {
         Object[] result = loop2( [1] );
         Object[] control = (Object[]) result[0];
         [2]
         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] };
   }
}
```

Table 4: Handling exceptions

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.

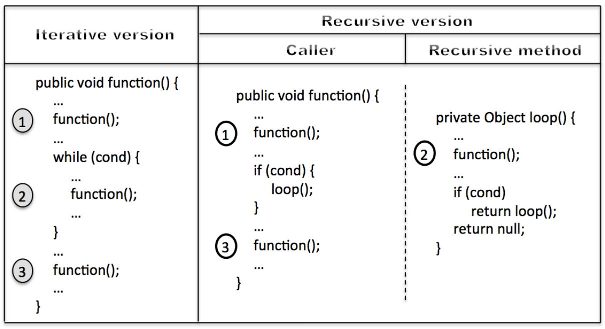
In the outer loop (method *loop1*), after the call to *loop2*, we check whether variable *control* is *null*. This is the way to know if the method should proceed normally, or it should handle the exception. Whenever *control != null*, it checks whether the exception is *Exception2*. In such a case, 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	Recursive version	
	Caller	Recursive method

Iterative version:

```
public void function() {
    ...
①   function();
    ...
    while (cond) {
        ...
②       function();
        ...
    }
    ...
③   function();
    ...
}
```

Recursive version — Caller:

```
public void function() {
    ...
①   function();
    ...
    if (cond) {
        loop();
    }
    ...
③   function();
    ...
}
```

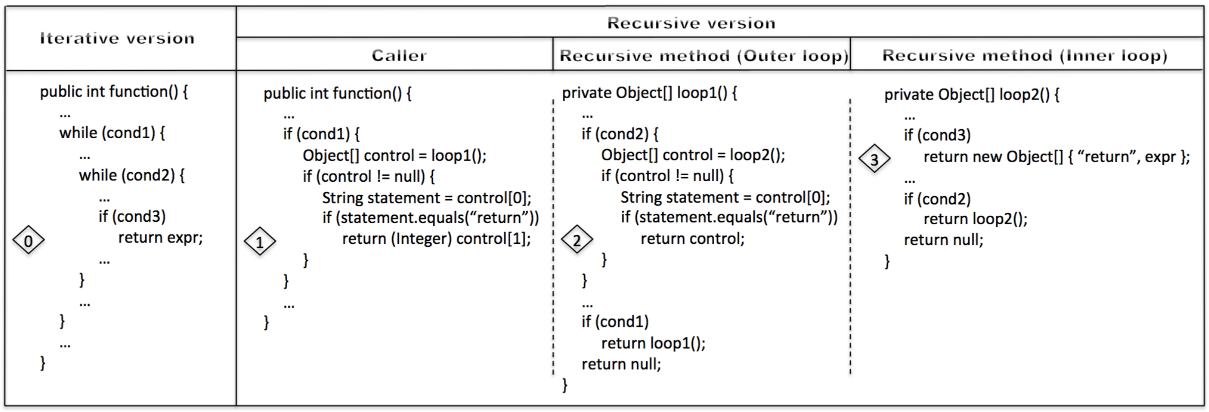
Recursive version — Recursive method:

```
private Object loop() {
    ...
②   function();
    ...
    if (cond)
        return loop();
    return null;
}
```

Table 5: Treatment of a recursive method

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



Iterative version	Recursive version		
	Caller	Recursive method (Outer loop)	Recursive method (Inner loop)

public int function() {
…
while (cond1) {
…
while (cond2) {
…
if (cond3)
◇0 return expr;
…
}
…
}
…
}
	public int function() {
	…
	if (cond1) {
	Object[] control = loop1();
	if (control != null) {
	String statement = control[0];
	if (statement.equals("return"))
	◇1 return (Integer) control[1];
	}
	}
	…
	}
		private Object[] loop1() {
		…
		if (cond2) {
		Object[] control = loop2();
		if (control != null) {
		String statement = control[0];
		if (statement.equals("return"))
		◇2 return control;
		}
		}
		…
		if (cond1)
		return loop1();
		return null;
		}
			private Object[] loop2() {
			…
			if (cond3)
			◇3 return new Object[] { "return", expr };
			…
			if (cond2)
			return loop2();
			return null;
			}

Table 6: Treatment of *return* statements

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 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/`

[3]The *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.

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 7 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 generare 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 7 where method *sqrt_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 7 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 [2]. 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 [8]. 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,

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

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 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;
```

[4]CoV is defined as the standard deviation s divided by the mean $\overline{x}$.

```
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.

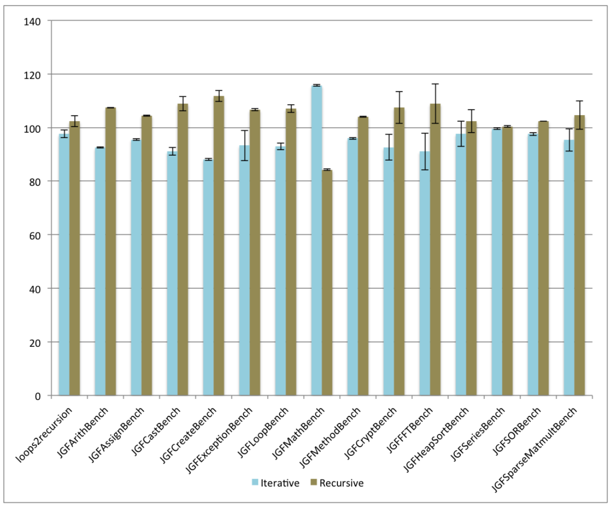
For the recursive versions, LOT is calculated in a similar way. For instance, in the `While-loop` row of Table 1:

- `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`.

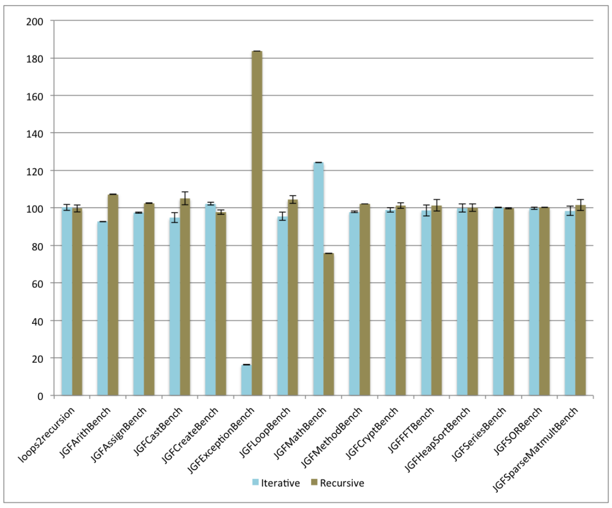
`Rec/Iter` measures the speedup/slowdown of the transformed code with respect to the original code. It is computed as 100×(Recursive time / Iterative time).

The results for LOT have been normalized in a chart shown in Figure 8(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%).

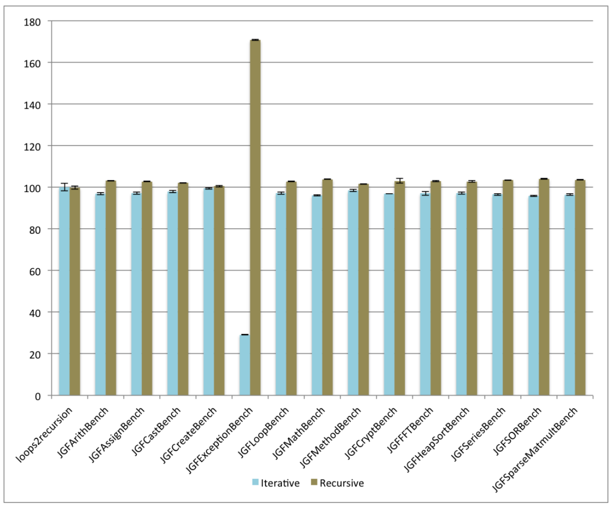
Initially, this was the end of our experiments. But then, we also looked at the results obtained for the ET. These results have been normalized in the chart shown in Figure 8(b). This figure reveals something unexpected: Sometimes the recursive version of the benchmarks is statistically (with 99% confidence 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 8(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



(a) LOT results



(b) ET results



(c) ET results with -Xint flag enabled

Figure 8: Normalized (scaled over 100) charts

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*, has been described independently with a specific treatment for it that is illustrated with an example of use. 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., [20]), 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 transforms loops to recursion or vice-versa before compiling them. We left this study for future work.

References

[1] Baker, H. G., dec 1996. Garbage collection, tail recursion and first-class continuations in stack-oriented languages. Patent, uS 5590332.
URL `http://www.patentlens.net/patentlens/patent/US_5590332/en/`

[2] Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., Davey, R. A., 2000. A benchmark suite for high performance java. Concurrency: Practice and Experience 12 (6), 375–388.
URL `http://dx.doi.org/10.1002/1096-9128(200005)12:6<375::AID-CPE480>3.0.CO;2-M`

[3] Clinger, W., may 1998. Proper tail recursion and space efficiency. ACM SIGPLAN Notices 33 (5), 174–185.
URL `http://doi.acm.org/10.1145/277652.277719`

[4] Elmroth, E., Gustavson, F. G., 2000. Applying recursion to serial and parallel QR factorization leads to better performance. IBM Journal of Research and Development 44 (4), 605–624.
URL `http://dx.doi.org/10.1147/rd.444.0605`

[5] Erascu, M., Jebelean, T., 2010. A Purely Logical Approach to the Termination of Imperative Loops. 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing 0, 142–149.

[6] Filinski, A., 1994. Recursion from iteration. LISP and Symbolic Computation 7, 11–37.
URL `http://dx.doi.org/10.1007/BF01019943`

[7] Fonseca, A., 2012. AEminium/loops2recursion Java Library. Available from URL: `https://github.com/AEminium/loops2recursion/`.

[8] Georges, A., Buytaert, D., Eeckhout, L., oct 2007. Statistically Rigorous Java Performance Evaluation. SIGPLAN Not. 42 (10), 57–76.
URL `http://doi.acm.org/10.1145/1297105.1297033`

[9] Geuvers, H., 1992. Inductive and Coinductive Types with Iteration and Recursion. In: Proceedings of the Workshop on Types for Proofs and Programs. pp. 193–217.

[10] Gustavson, F. G., 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development 41 (6), 737–756.
URL `http://dx.doi.org/10.1147/rd.416.0737`

[11] Hanson, C., 1990. Efficient stack allocation for tail-recursive languages. In: Proceedings of the 1990 ACM conference on LISP and Functional Programming (LFP'90). ACM, New York, NY, USA, pp. 106–118.
URL `http://doi.acm.org/10.1145/91556.91603`

[12] Harrison, P. G., Khoshnevisan, H., feb 1992. A new approach to recursion removal. Electronic Notes in Theoretical Computer Science 93 (1), 91–113.
URL `http://dx.doi.org/10.1016/0304-3975(92)90213-Y`

[13] Insa, D., Silva, J., 2010. An Algorithmic Debugger for Java. In: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010). pp. 1–6.

[14] Insa, D., Silva, J., Tomás, C., sep 2012. Enhancing Declarative Debugging with Loop Expansion and Tree Compression. In: Albert, E. (Ed.), Proceedings of the 22th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2012). Vol. 7844 of Lecture Notes in Computer Science (LNCS). Springer, pp. 71–88.

[15] Liu, Y. A., Stoller, S. D., 2000. From recursion to iteration: what are the optimizations? In: Proceedings of the 2000 ACM-SIGPLAN Workshop on Partial Evaluation and semantics-based Program Manipulation (PEPM'00). ACM, New York, NY, USA, pp. 73–82.
URL `http://doi.acm.org/10.1145/328690.328700`

[16] Manolios, P., Moore, J. S., dec 2003. Partial Functions in ACL2. Journal of Automated Reasoning 31 (2), 107–127.
URL `http://dx.doi.org/10.1023/B:JARS.0000009505.07087.34`

[17] McCarthy, J., 1962. Towards a Mathematical Science of Computation. In: Proceedings of the 2nd International Federation for Information Processing Congress (IFIP'62). North-Holland, pp. 21–28.

[18] Myreen, M., Gordon, M., jul 2009. Transforming Programs into Recursive Functions. Electronic Notes in Theoretical Computer Science 240, 185–200.
URL `http://dx.doi.org/10.1016/j.entcs.2009.05.052`

[19] Nielson, H. R., Nielson, F., 1992. Semantics with Applications: A Formal Introduction. John Wiley & Sons, Inc., New York, NY, USA.

[20] Yi, Q., Adve, V., Kennedy, K., 2000. Transforming Loops to Recursion for Multi-level Memory Hierarchies. In: Proceedings of the 21st ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00). pp. 169–181.

Appendix A. Proof of correctness

In this section we provide a formal semantics-based specification of our transformation in order to prove the correctness of the basic transformation (we skip exceptions and labels in this section). For this, we provide a BNF syntax specification and an operational semantics of Java. We consider the subset of Java that is needed to implement the transformation (*if-then-else*, *while*, method calls, *return*, etc.), and we ignore the rest of syntax elements for the sake of simplicity (they do not have any influence because any command inside the body of the loop remains unchanged in the transformed code). Moreover, in this section, we center the discussion on *while*-loops and prove properties for this kind of loop. The proof for the other kinds of loops is omitted, but it would be in all cases analogous or slightly incremental.

We start with a BNF syntax of Java:

P	$::=$	$M_1, \ldots, M_n, S_p$	(program)	*Domains*
				$x, y, z \ldots \in \mathbb{V}$ (variables)
				$a, b, c \ldots \in \mathbb{C}$ (constants)
M	$::=$	$m(x_1, \ldots, x_n)$ { S_p; S_r }	(method definition)	where $x_1, \ldots, x_n \in \mathbb{V}$ and
				m is the name of the method
S_p	$::=$	$x := E$	(assignment)	
	$\mid$	$x := m(E_0, \ldots, E_n)$	(method invocation)	
	$\mid$	if E_b then S_p	(if-then)	
	$\mid$	if E_b then S_p else S_p	(if-then-else)	
	$\mid$	while E_b do S_p	(while)	
	$\mid$	S_p; S'_p	(sequence)	
S_r	$::=$	return E	(return)	
E	$::=$	$E_a \mid E_b$	(expresion)	
E_a	$::=$	$E_a + E_a \mid E_a - E_a \mid V$	(arithmetic expresion)	
E_b	$::=$	$E_b \mathrel{!=} E_b \mid E_b == E_b \mid V$	(boolean expresion)	
V	$::=$	$x \mid a$	(variables or constants)	

A program is a set of method definitions and at least one initial statement (usually a method invocation). Each method definition is composed of a set of statements followed by a *return* statement. For simplicity, the arguments of a method invocation can only be expressions (not statements). This is not a restriction, because any statement can be assigned to a variable and then be passed as argument of the method invocation. However, this simplification allows us to ease the semantics of method invocations and, thus, it increases readability.

In the following we consider two versions of the same program shown in Algorithms 8 and 9. We assume that in Algorithm 8 there exists a variable x already defined before the loop and, for the sake of simplicity, it is the only variable modified inside S. Therefore, Algorithm 9 is the recursive version of the *while*-loop in Algorithm 8 according to our transformation, and hence, $p_0, \ldots, p_n$ represent all variables defined before the loop and used in S (the loop statements) and *cond* (the loop condition). In the case that more than one variable are modified, then the output would be an array with all variables modified. We avoid this case because it is not necessary for the proof.

Algorithm 8 While version

```
1: while $cond$ do
2:     $S$;
```

Algorithm 9 Recursive version

```
1: $m(p_0, \ldots, p_n)$ {
2:     $S$;
3:     if $cond$ then
4:         $x := m(a_0, \ldots, a_n)$;
5:     return $x$;
6: }
7: if $cond$ then
8:     $x := m(a_0, \ldots, a_n)$;
```

In order to provide an operational semantics for this Java subset, which allows recursion, we need a stack to push and pop different frames that represent individual method activations. Frames, $f_0, f_1, \ldots \in \mathbb{F}$, are sequences of pairs variable-value. States, $s_0, s_1, \ldots \in \mathbb{S}$, are sequences of frames ($\mathbb{S}: \ \mathbb{F}$ x ... x $\mathbb{F}$). We make the program explicitly accessible to the semantics through the use of an environment, $e \in \mathbb{E}$, represented with a sequence of functions from method names $\mathbb{M}$ to pairs of parameters $\mathbb{P}$ and statements $\mathbb{I}$ ($\mathbb{E}: \ (\mathbb{M} \to (\mathbb{P}$ x $\mathbb{I}))$ x ... x $(\mathbb{M} \to (\mathbb{P}$ x $\mathbb{I})))$. Our semantics is based on the Java semantics described in [19] with some minor modifications. It uses a set of functions to update the state, the environment, etc.

Function Upd_v is used to update a variable (*var*) in the current frame of the state (s) with a value (*value*). The current frame in the state is always the last frame introduced (i.e., the last element in the sequence of frames that represent the state). We use the standard notation $f[var \to value]$ to denote that variable *var* in frame f is updated to value *value*.

$$Upd_v(s,\ var \to value) = \begin{cases} error & if\ s = [] \\ [f_0, \ldots, f_n[var \to value]] & if\ s = [f_0, \ldots, f_n] \end{cases}$$

Function Upd_r records the returned value (*value*) of the current frame of the state (s) inside a fresh variable $\Re$ of this frame, so that other frames can consult the value returned by the current frame.

$$Upd_r(s,\ value) = \begin{cases} error & if\ s = [] \\ [f_0, \ldots, f_n[\Re \to value]] & if\ s = [f_0, \ldots, f_n] \end{cases}$$

Function Upd_{vr} is used to update a variable (*var*) in the penultimate frame of the state (s) taking the value returned by the last frame in the state (which must be previously stored in $\Re$). This happens when a method calls another method and the latter finishes returning a value. In this situation, the last frame in the state should be removed and the value returned should be updated in the penultimate frame. We use the notation $f_n(\Re)$ to consult the value of variable $\Re$ in frame f_n.

$$Upd_{vr}(s,\ var) = \begin{cases} error & if\ \ s = []\ \ or\ \ s = [f] \\ [f_0, \ldots, f_{n-1}[var \to f_n(\Re)], f_n] & if\ \ s = [f_0, \ldots, f_{n-1}, f_n] \end{cases}$$

Function Upd_e is used to update the environment (env) with a new method definition ($m \to (P, I)$). The environment is used in method invocations to know the method that should be executed.

$$Upd_e(env,\ m \to (P, I)) = env[m \to (P, I)]$$

Function $AddFrame$ adds a new frame to the state (s). This frame is a sequence of mappings from parameters $(p_0, \ldots, p_n)$ to the evaluation of arguments $(a_0, \ldots, a_n)$. To evaluate an expression we use function $Eval$: a variable is consulted in the state, a constant is just returned, and a mathematical or boolean expression is evaluated with the standard semantics. We use this notation because the evaluation of expressions does not have influence in our proofs, but it significantly reduces the size of derivations, thus, improving clarity of presentation.

$$AddFrame(s, [p_0, \ldots, p_n], [a_0, \ldots, a_n]) = \begin{cases} [[p_0 \to Eval(a_0), \ldots, p_n \to Eval(a_n)]] & if\ \ s = [] \\ [f_0, \ldots, f_m, [p_0 \to Eval(a_0), \ldots, p_n \to Eval(a_n)]] & if\ \ s = [f_0, \ldots, f_m] \end{cases}$$

Analogously, function $RemFrame$ removes the last frame inserted into the state (s).

$$RemFrame(s) = \begin{cases} error & if\ \ s = [] \\ [] & if\ \ s = [f] \\ [f_0, \ldots, f_n] & if\ \ s = [f_0, \ldots, f_n, f_{n+1}] \end{cases}$$

We are now in a position ready to introduce our Java operational semantics. Essentially, the semantics is a big-step semantics composed of a set of rules of the form: $\frac{p_1 \cdots p_n}{env\ \vdash\ <st,\ s>\ \Downarrow\ s'}$ that should be read as "The execution of statement st in state s under the environment env can be reduced to state s' provided that premises $p_1 \ldots p_n$ hold". The rules of the semantics are shown below.

New method

$$\frac{env' = Upd_e(env,\ m_0 \rightarrow (P,\ I)) \wedge env' \vdash <m_1\ i,\ s> \Downarrow s'}{env \vdash <m_0(P)\{I\}\ m_1\ i,\ s> \Downarrow s'}$$

Empty statement

$$\frac{}{env \vdash <\surd,\ s> \Downarrow s}$$

Asignment

$$\frac{s' = Upd_v(s,\ x \rightarrow Eval(op,\ s))}{env \vdash <x{:=}op,\ s> \Downarrow s'}$$

Method invocation

$$\frac{(P,\ I) = env(m) \wedge s' = AddFrame(s,P,A) \wedge env \vdash <I,\ s'> \Downarrow s'' \wedge s''' = Upd_{vr}(s'',x) \wedge s'''' = RemFrame(s''')}{env \vdash <x{:=}m(A),\ s> \Downarrow s''''}$$

If

$$\frac{env \vdash <if\ cond\ then\ i_0\ else\ \surd,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0,\ s> \Downarrow s'}$$

$$\frac{<cond,\ s> \Rightarrow true \wedge env \vdash <i_0,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0\ else\ i_1,\ s> \Downarrow s'}$$

$$\frac{<cond,\ s> \Rightarrow false \wedge env \vdash <i_1,\ s> \Downarrow s'}{env \vdash <if\ cond\ then\ i_0\ else\ i_1,\ s> \Downarrow s'}$$

While

$$\frac{<cond,\ s> \Rightarrow false}{env \vdash <while\ cond\ do\ i,\ s> \Downarrow s}$$

$$\frac{<cond,\ s> \Rightarrow true \wedge env \vdash <i,\ s> \Downarrow s' \wedge env \vdash <while\ cond\ do\ i,\ s'> \Downarrow s''}{env \vdash <while\ cond\ do\ i,\ s> \Downarrow s''}$$

Sequence

$$\frac{env \vdash <i_0,\ s> \Downarrow s' \wedge env \vdash <i_1,\ s'> \Downarrow s''}{env \vdash <i_0;\ i_1,\ s> \Downarrow s''}$$

Return

$$\frac{s' = Upd_r(s,\ Eval(op,\ s))}{env \vdash <return\ op,\ s> \Downarrow s'}$$

We can now prove our main result.

Theorem Appendix A.1 (Correctness). *Algorithm 9 is semantically equivalent to Algorithm 8.*

Proof. We prove this claim by showing that the final state of Algorithm 8 is always the same as the final state of Algorithm 9. The semantics of a program P is:

$$S(P) = s \quad \text{iff} \quad [\,] \vdash < P,\ [\,] > \Downarrow s$$

Therefore, we say that two programs P_1 and P_2 are equivalent if they have the same semantics:

$$S(P_1) = S(P_2) \quad \text{iff} \quad [\,] \vdash < P_1,\ [\,] > \Downarrow s_1 \quad \wedge \quad [\,] \vdash < P_2,\ [\,] > \Downarrow s_2 \quad \wedge \quad s_1 = s_2$$

For the sake of generality, in the following we consider that the loops can appear inside any other code. Therefore, the environment and the state are not necessarily empty. Thus, we will assume an initial environment env_0 and an initial state s: $env_0 \vdash < P,\ s >$. We proof this semantic equivalence analyzing two possible cases depending on whether the loop is executed or not.

1) <u>Zero iterations</u>

This situation can only happen when the condition *cond* is not satisfied the first time it is evaluated. Hence, we have the following semantics derivation for each program:

<u>Iterative version</u>

$$\frac{< cond,\ s > \ \Rightarrow\ false}{env_0 \ \vdash\ < while\ cond\ do\ S,\ s > \ \Downarrow\ s}$$

<u>Recursive version</u>

$$\frac{env = Upd_e(env_0,\ m\ \rightarrow\ (P,\ S; I)) \qquad \dfrac{\dfrac{< cond,\ s > \ \Rightarrow\ false \quad \dfrac{}{env\ \vdash\ < \surd,\ s > \ \Downarrow\ s}}{env\ \vdash\ < if\ cond\ then\ t\ else\ \surd,\ s > \ \Downarrow\ s}}{env\ \vdash\ < if\ cond\ then\ t,\ s > \ \Downarrow\ s}}{env_0\ \vdash\ < m(P)\ \{\ S; I\ \}\ if\ cond\ then\ t,\ s > \ \Downarrow\ s}$$

Clearly, the state is never modified neither in the iterative version nor in the recursive version. Therefore, both versions are semantically equivalent.

2) <u>One or more iterations</u>

This means that the condition *cond* is satisfied at least once. Let us consider that *cond* is satisfied n times, producing n iterations. We proof that the final state of the program in Algorithm 9 is equal to the final state of the program in Algorithm 8 by induction over the number of iterations performed.

(Base Case) In the base case, only one iteration is executed. Hence, we have the following derivations:

<u>Iterative version</u>

$$\frac{< cond,\ s > \ \Rightarrow\ true \quad env_0\ \vdash\ < S,\ s > \ \Downarrow\ s^1 \quad \dfrac{< cond,\ s > \ \Rightarrow\ false}{env_0\ \vdash\ < while\ cond\ do\ S,\ s^1 > \ \Downarrow\ s^1}}{env_0\ \vdash\ < while\ cond\ do\ S,\ s > \ \Downarrow\ s^1}$$

Recursive version

$$\dfrac{\mathit{env} \ \vdash \ < S, \ s^1 > \ \Downarrow \ s^2 \qquad \dfrac{\dfrac{\dfrac{< \mathit{cond}, \ s^2 > \ \Rightarrow \ \mathit{false} \quad \dfrac{}{\mathit{env} \ \vdash \ < \surd, \ s^2 > \ \Downarrow \ s^2}}{\mathit{env} \ \vdash \ < \mathit{if\ cond\ then\ t\ else} \ \surd, \ s^2 > \ \Downarrow \ s^2}}{\mathit{env} \ \vdash \ < \mathit{if\ cond\ then\ t}, \ s^2 > \ \Downarrow \ s^2} \quad \dfrac{s^3 \ = \ \mathit{Upd}_r(s^2, \ \mathit{Eval}(x, s^2))}{\mathit{env} \ \vdash \ < \mathit{return}\ x, \ s^2 > \ \Downarrow \ s^3}}{\mathit{env} \ \vdash \ < \mathit{if\ cond\ then\ t};\ \mathit{return}\ x, \ s^2 > \ \Downarrow \ s^3}}{\dfrac{\mathit{env} \ \vdash \ < S; I, \ s^1 > \ \Downarrow \ s^3}{\triangle}}$$

$$\dfrac{\dfrac{< \mathit{cond}, \ s > \ \Rightarrow \ \mathit{true} \qquad \dfrac{(P, I) = \mathit{env}(m) \quad s^1 \ = \ \mathit{AddFrame}(s, P, [a_0, \ldots, a_n]) \quad \triangle \quad s^4 \ = \ \mathit{Upd}_{vr}(s^3, x) \quad s^5 \ = \ \mathit{RemFrame}(s^4)}{\mathit{env} \ \vdash \ < x := m(a_0, \ldots, a_n), \ s > \ \Downarrow \ s^5}}{\mathit{env} \ \vdash \ < \mathit{if\ cond\ then\ t\ else} \ \surd, \ s > \ \Downarrow \ s^5}}{\mathit{env} \ \vdash \ < \mathit{if\ cond\ then\ t}, \ s > \ \Downarrow \ s^5}$$

We can assume that variable x has an initial value z^0, which must be the same in both versions of the algorithm. Then, states are modified during the iteration as follows:

Iterative version	Recursive version
$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$
$s^1 = [f_0] \Rightarrow f_0 = \{x \to z^1\}$	$s^1 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^0\}$
	$s^2 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^1\}$
	$s^3 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^1, \Re \to z^1\}$
	$s^4 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^1\} \wedge f_1 = \{x \to z^1, \Re \to z^1\}$
	$s^5 = [f_0] \Rightarrow f_0 = \{x \to z^1\}$

Clearly, with the same initial states, both algorithms produce the same final state.

(Induction Hypothesis) We assume as the induction hypothesis that executing i iterations in both versions with an initial value z^0 for x then, if the iterative version obtains a final value z^n for x then the recursive version correctly obtains and stores the same final value z^n for variable x in the top frame.

Iterative version	Recursive version
$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$
...	...
$s' = [f_0] \Rightarrow f_0 = \{x \to z^n\}$	$s' = [f_0] \Rightarrow f_0 = \{x \to z^n\}$

(Inductive Case) We now prove that executing $i + 1$ iterations in both versions with an initial value z^0 for x then, if the iterative version obtains a final value z^n for x then the recursive version correctly obtains and stores the same final value z^n for variable x in the top frame.

The derivation obtained for each version is the following:

Iterative version

$$\dfrac{< \mathit{cond}, \ s > \ \Rightarrow \ \mathit{true} \quad \mathit{env}_0 \ \vdash \ < S, \ s > \ \Downarrow \ s^1 \quad \dfrac{\mathit{Induction\ hypotesis}}{\mathit{env}_0 \ \vdash \ < \mathit{while\ cond\ do}\ S, \ s^1 > \ \Downarrow \ s^2}}{\mathit{env}_0 \ \vdash \ < \mathit{while\ cond\ do}\ S, \ s > \ \Downarrow \ s^2}$$

Recursive version

$$\frac{\dfrac{env \ \vdash \ < S,\ s^1 > \ \Downarrow \ s^2 \qquad \dfrac{\dfrac{Induction\ hypotesis}{env \ \vdash \ < if\ cond\ then\ t,\ s^2 > \ \Downarrow \ s^3} \quad \dfrac{s^4 \ = \ Upd_r(s^3,\ Eval(x, s^3))}{env \ \vdash \ < return\ x,\ s^3 > \ \Downarrow \ s^4}}{env \ \vdash \ < if\ cond\ then\ t;\ return\ x,\ s^2 > \ \Downarrow \ s^4}}{env \ \vdash \ < S; I,\ s^1 > \ \Downarrow \ s^4}}{\triangle}$$

$$\frac{\dfrac{< cond,\ s > \ \Rightarrow \ true \qquad \dfrac{(P, I) = env(m) \quad s^1 \ = \ AddFrame(s, P, [a_0, \ldots, a_n]) \quad \triangle \quad s^5 \ = \ Upd_{vr}(s^4, x) \quad s^6 \ = \ RemFrame(s^5)}{env \ \vdash \ < x := m(a_0, \ldots, a_n),\ s > \ \Downarrow \ s^6}}{env \ \vdash \ < if\ cond\ then\ t\ else\ \surd,\ s > \ \Downarrow \ s^6}}{env \ \vdash \ < if\ cond\ then\ t,\ s > \ \Downarrow \ s^6}$$

Because both algorithms have the same initial value z^0 for x then the states during the iteration are modified as follows (the * state is obtained by the induction hypothesis):

Iterative version	Recursive version
$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$	$s = [f_0] \Rightarrow f_0 = \{x \to z^0\}$
$s^1 = [f_0] \Rightarrow f_0 = \{x \to z^1\}$	$s^1 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^0\}$
$s^2 = [f_0] \Rightarrow f_0 = \{x \to z^n\}*$	$s^2 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^1\}$
	$s^3 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^n\}*$
	$s^4 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^0\} \wedge f_1 = \{x \to z^n, \Re \to z^n\}$
	$s^5 = [f_0, f_1] \Rightarrow f_0 = \{x \to z^n\} \wedge f_1 = \{x \to z^n, \Re \to z^n\}$
	$s^6 = [f_0] \Rightarrow f_0 = \{x \to z^n\}$

Hence, Algorithm 9 and Algorithm 8 obtain the same final state, and thus, they are semantically equivalent. $\square$