Notes on converting recursion into iteration

Sherif Khattab

November 15, 2017

Theses notes give examples on two techniques for converting a recursive method into an iterative method. The first technique converts the recursive method into a tail-recursive method then converts that into a loop. The second technique uses a stack to simulate the run-time stack. Although the first technique may appear simpler, the second technique is more general. In other words, any recursive method can be turned into an iterative method using the second technique, whereas some recursive methods cannot be converted to iterative using the first technique.

Method 1: Through tail-recursion

Example: Computing the n^{th} Fibonacci number

We start with the doubly-recursive method below.

```
public static int Fibonacci(int n)
{
      if(n==0)
         return 0;
      if(n==1)
          return 1;
      return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Converting into a tail-recursive method

Converting that into a tail-recursive method involves introducing a helper method with extra parameters for partial solutions. These partial solutions are passed with each recursive call and the keep "growing" until they reach the final result with the deepest recursive call. Because we have two recursive calls here, we use two parameters for the partial solutions. (These parameters will hold a sliding window of two consecutive Fibonacci numbers. That window moves one step to the left, along the sequence of Fibonacci numbers, with every recursive call.)

The helper function is listed below. Because we have a sliding window that moves left one step per call, the firstNumber argument takes on the value of the secondNumber, and the secondNumber argument takes the value of the next Fibonacci number, that is the sum of the two numbers in the window (firstNumber+secondNumber).

```java
private static int FibonacciHelper(int n, int firstNumber, int
   secondNumber)
 {

     if(n==0){
         return firstNumber;
     }
     if(n==1){
         return secondNumber;
     }
     return FibonacciHelper(n-1, secondNumber,
        firstNumber+secondNumber);
 }
```

The original method would just make a call to FibonacciHelper. Initially, the window of partial solutions contains the two numbers 0 and 1. Note that the method Fibonacci is not a tail-recursive method because the last call is not a recursive call; it is a call to a different method.

The if conditions in the helper function mark the deepest recursive calls. One way of coming up with these conditions is thinking about the very first call to FibonacciHelper, which is `FibonacciHelper(n, 0, 1)`, when `n==1`. In this call, we set the sliding window to contain the two numbers 0 and 1. The first Fibonacci number is 1, so we should be returning 1, and 1 is the second number of the sliding window. Similarly, we can think of the `n==0` condition.

```java
public static int Fibonacci(int n)
{
   return FibonacciHelper(n, 0, 1);
}
```

Converting the tail-recursive method into an iterative method

We change the method so that the the recursive case is the then part of an if statement. Then we replace the if by a while.

```java
private static int FibonacciHelper(int n, int firstNumber, int
   secondNumber)
{
    if(n==0){
        return firstNumber;
    }
    if(n>1){
       return FibonacciHelper(n-1, secondNumber,
          firstNumber+secondNumber);
    } else {
        return secondNumber;
    }
}
```

Then we replace the last recursive call by assignment statements, one per parameter of the method. The left-hand-side is the parameter, and the right-hand-side is the corresponding argument in the recursive call. We use the variable temp because we want the value of firstNumber before it is overwritten by secondNumber.

```
private static int FibonacciHelper(int n, int firstNumber, int
   secondNumber)
{
     if(n==0){
        return firstNumber;
    }
    while(n>1){
        n = n-1;
        int temp = firstNumber;
        firstNumber = secondNumber;
        secondNumber = temp + secondNumber;

    }
    return secondNumber;
}
```

Method 2: Using a stack

To convert a recursive method into an iterative method using a simulated run-time stack, we follow the following steps.

1. First, we make sure that if a recursive call is returning a value, then that value be assigned into a local variable first. So, for example, `return Power(x, n-1)` should be spelled out as `int result = Power(x, n-1); return result;`

2. We make sure that we have at most one recursive call per statement. For example, `return Fibonacci(n-1) + Fibonacci(n-2);` should be turned into `int result = Fibonacci(n-1); result += Fibonacci(n-1); return result;`

3. If the method returns a value, we define a local variable, named result for example, to hold the return value.

4. We define a class for the activation record of the method. Remember that we are simulating the run-time stack. Activation records are pushed and popped from the stack every time a method is called and every time a method returns, respectively. The activation record contains the method parameters, all local variables, and the program counter. The program counter keeps track of the point execution reached in the method. We only need to keep track of where execution is relative to recursive calls. So, for example, if a method has two recursive calls, we only need to keep track of being up to the first recursive call, up to the second recursive call, and after the second call. So, every value of the program counter corresponds to one segment of the recursive method's code. Except for

the last segment, each segment ends by a recursive call. An example activation-record class for the Fibonacci method is:

```
enum Location {FIRST_CALL, SECOND_CALL, END}

 private class Record {

        private int n;
        private int result;
        private Location programCounter;


        Record(int n, int result, Location programCounter){
                this.n = n;
                this.result = result;
                this.programCounter = programCounter;
        }
```

5. We then create our own stack in the method. We use the stack to simulate how the run-time stack works.
6. We push an activation record into the stack representing the first call to the recursive method.
7. We then have a while-loop that iterates while the stack is not empty.
8. We start the body of the loop by popping an activation record from the stack and assigning the method parameters whatever values the activation record has for them.
9. We then have a switch statement based on the values of the program counter. Within every case, we put the code that corresponds to the respective value of the program counter. We do the below changes to that code though.
10. We end each segment (except the last one) by pushing a record to the stack with the program counter having the value of the next code segment. Everything else on the activation record is the same. We replace the recursive calls by pushes into the stack, whereby the parameters of the activation record to be pushed are the arguments of the call. The program counter points to the first code segment.
11. We replace every return statement by an assignment to the result local variable followed by a continue statement.
12. If the recursive method has a return value, we return the local variable result `return result;`.

Example: Solving the Towers of Hanoi problem

We start with the doubly-recursive method below.

```
public static void TowersOfHanoi(int n, int start, int temp, int end)
{
      if(n==0) return;

      TowersOfHanoi(n-1, start, end, temp);
      System.out.println("Move one disc from pole " + start + " to
         pole " + end);
      TowersOfHanoi(n-1, temp, start, end);
 }
```

Then we create our own stack in the method. We use the stack to simulate how the run-time stack works.

```
public static void TowersOfHanoi(int n, int start, int temp, int
   end)
{
        StackInterface<Record> runTimeStack = new
           LinkedStack<>();

        if(n==0) return;

        TowersOfHanoi(n-1, start, end, temp);
        System.out.println("Move one disc from pole " + start +
           " to pole " + end);
        TowersOfHanoi(n-1, temp, start, end);

}
```

We define a class, Record, that contains the parameters (n, start, temp, and end), the local variables (if any), and the program counter.

```
enum Location {FIRST_CALL, SECOND_CALL, END}

 private class Record {

  private int n;
  private int start;
  private int temp;
  private int end;
  private Location programCounter;

  Record(int n, int start, int temp, int end, Location
     programCounter){
```

```
                this.n = n;
                this.start = start;
                this.temp = temp;
                this.end = end;
                this.programCounter = programCounter;
        }
}
```

We then define a loop that iterates while the stack is not empty. Before entering the loop we push an activation record with the method parameters into the stack. We start the body of the loop by popping an activation record from the stack and assigning the method parameters whatever values the activation record has for them.

```
runTimeStack.push(new Record(n, start, temp, end,
   Location.FIRST_CALL));
while(! runTimeStack.isEmpty()){
                        Record r = runTimeStack.pop();
                        n = r.n;
                        start = r.start;
                        temp = r.temp;
                        end = r.end;

    }
```

Then, we put the switch statement and fill in the cases with the corresponding code segments from the recursive method.

```
        switch(r.programCounter){
                case FIRST_CALL:
                        if(n==0){
                                return;
                        }

                        TowersOfHanoi(n-1, start, end, temp);
                break;

                case SECOND_CALL:
                        System.out.println("Move a disk from pole " +
                           start + " to pole " + end);
                        TowersOfHanoi(n-1, temp, start, end);

                break;
                case END:
                break;
        }
```

Then, we end each segment (except the last one) by pushing a record to the stack with the program counter having the value of the next code segment. We replace the recursive calls by pushes into the stack. We replace every return statement by an assignment to the result local variable followed by a continue statement. We keep everything else in the method without change.

```
        switch(r.programCounter){
                case FIRST_CALL:
                        if(n==0){
                                continue;
                        }

                        runTimeStack.push(new Record(n, start, temp,
                           end, Location.SECOND_CALL));
                        runTimeStack.push(new Record(n-1, start, end,
                           temp, Location.FIRST_CALL));

                break;

                case SECOND_CALL:
                        System.out.println("Move a disk from pole " +
                           start + " to pole " + end);
                        runTimeStack.push(new Record(n, start, temp,
                           end, Location.END));
                        runTimeStack.push(new Record(n-1, temp, start,
                           end, Location.FIRST_CALL));

                break;
                case END:
                break;
        }
```

Putting that together we have the following iterative method.

```
enum Location {FIRST_CALL, SECOND_CALL, END}

public void TowersOfHanoi(int n, int start, int temp, int end)
{
        StackInterface<Record> runTimeStack = new LinkedStack<>();

        runTimeStack.push(new Record(n, start, temp, end,
           Location.FIRST_CALL));
        while(! runTimeStack.isEmpty()){
                Record r = runTimeStack.pop();
                n = r.n;
                start = r.start;
                temp = r.temp;
```

```java
                end = r.end;

                switch(r.programCounter){
                case FIRST_CALL:
                        if(n==0){
                                continue;
                        }

                        runTimeStack.push(new Record(n, start, temp,
                           end, Location.SECOND_CALL));
                        runTimeStack.push(new Record(n-1, start, end,
                           temp, Location.FIRST_CALL));

                break;

                case SECOND_CALL:
                        System.out.println("Move a disk from pole " +
                           start + " to pole " + end);
                        runTimeStack.push(new Record(n, start, temp,
                           end, Location.END));
                        runTimeStack.push(new Record(n-1, temp, start,
                           end, Location.FIRST_CALL));

                break;
                case END:
                break;
            }
        }
}

 private class Record {

  private int n;
  private int start;
  private int temp;
  private int end;
  private Location programCounter;

  Record(int n, int start, int temp, int end, Location programCounter){
          this.n = n;
          this.start = start;
          this.temp = temp;
          this.end = end;
          this.programCounter = programCounter;
```

```
  }
}
```

Example 2: Computing the n^{th} Fibonacci number

We start with the doubly-recursive method below.

```
public static int Fibonacci(int n)
{
  if(n==0)
      return 0;
  if(n==1)
      return 1;
  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

1. We "spell out" the statement involving the recursive calls. We make sure that if a recursive call is returning a value, then that value be assigned into a local variable.

   ```
   public static int Fibonacci(int n)
   {   int temp=0, result=0;

       if(n==0)
           return 0;
       if(n==1)
           return 1;
       int temp = Fibonacci(n-1);
       temp += Fibonacci(n-2);
       return temp;
   }
   ```

2. We define a local variable, result, to hold the return value.

3. We define a class Record that contains the parameter n, the local variable temp, and the program counter.

   ```
   enum Location {FIRST_CALL, SECOND_CALL, END}
   private class Record {
      private int n;
      private int temp;
      private Location programCounter;
      Record(int n, int temp, Location programCounter){
             this.n = n;
             this.temp = temp;
   ```

```
            this.programCounter = programCounter;
        }
    }
```

4. We create our own stack in the method. We use the stack to simulate how the run-time stack works.

```
StackInterface<Record> runTimeStack = new LinkedStack<>();
```

5. We then have a loop that iterates while the stack is not empty. Before entering the loop we push an activation record with the method parameters into the stack. We start the body of the loop by popping an activation record from the stack and assigning the method parameters and local variables whatever values the activation record has for them.

```
runTimeStack.push(new Record(n, 0, 0 Location.FIRST_CALL));
    while(! runTimeStack.isEmpty()){
          Record r = runTimeStack.pop();
          n = r.n;
          temp = r.temp;
          result = r.result;

          switch(r.programCounter){
          case FIRST_CALL:
      if(n==0)
          return 0;
      if(n==1)
          return 1;
      int temp = Fibonacci(n-1);
          break;

          case SECOND_CALL:
           temp += Fibonacci(n-2);
          break;
          case END:
           return temp;
          break;
      }
  }
```

6. We replace every return statement by an assignment to the result local variable followed by a continue statement.

```
runTimeStack.push(new Record(n, temp, Location.FIRST_CALL));
    while(! runTimeStack.isEmpty()){
          Record r = runTimeStack.pop();
          n = r.n;
```

```
                temp = r.temp;

                switch(r.programCounter){
                case FIRST_CALL:
            if(n==0) {
                result =  0;
                continue;
            }
            if(n==1) {
                result = 1;
                continue;
            }
            int temp = Fibonacci(n-1);
                break;

                case SECOND_CALL:
                 temp += Fibonacci(n-2);
                break;
                case END:
                 result = temp;
                 continue;
            }
        }
```

7. We end each segment (except the last one) with a push for the next code segment and replace the recursive calls by pushes into the stack. Note that the assignment temp = Fibonacci(n-1) actually happens *after* the recursive call. So, it is moved to the next segment as `temp = result;`. Finally, we add a return statement at the end of the method `return result;`.

```
        runTimeStack.push(new Record(n, temp, Location.FIRST_CALL));
        while(! runTimeStack.isEmpty()){
          Record r = runTimeStack.pop();
          n = r.n;
          temp = r.temp;

          switch(r.programCounter){
          case FIRST_CALL:
              if(n==0) {
                  result =  0;
                  continue;
              }
              if(n==1) {
                  result = 1;
                  continue;
```

```
            }
            runTimeStack.push(new Record(n, temp,
               Location.SECOND_CALL));
            runTimeStack.push(new Record(n-1, temp,
               Location.FIRST_CALL));
                break;
        case SECOND_CALL:
           temp = result;
           runTimeStack.push(new Record(n, temp, Location.END));
           runTimeStack.push(new Record(n-2, temp,
              Location.FIRST_CALL));
        break;
        case END:
         temp += result;
         result = temp;
         continue;
        }
    }
    return result;
```

8. Putting that all together, we get the following iterative method.

```
public int Fibonacci(int n){
          int result=0, temp=0;

          StackInterface<Record> runTimeStack = new LinkedStack<>();
          runTimeStack.push(new Record(n, temp,
             Location.FIRST_CALL));
          while(! runTimeStack.isEmpty()){
                Record r = runTimeStack.pop();
                n = r.n;
                temp = r.temp;

                switch(r.programCounter){
                case FIRST_CALL:
                    if(n==0) {
                    result =  0;
                    continue;
                }
                if(n==1) {
                    result = 1;
                    continue;
                }
                runTimeStack.push(new Record(n, temp,
```

```
                   Location.SECOND_CALL));
                 runTimeStack.push(new Record(n-1, temp,
                   Location.FIRST_CALL));
                       break;
                 case SECOND_CALL:
                   temp = result;
                   runTimeStack.push(new Record(n, temp,
                     Location.END));
                   runTimeStack.push(new Record(n-2, temp,
                     Location.FIRST_CALL));
                 break;
                 case END:
                  temp += result;
                  result = temp;
                  continue;
              }
           }
           return result;
           }

    private class Record {
          private int n;
          private int temp;
          private Location programCounter;
          Record(int n, int temp, Location programCounter){
                 this.n = n;
                 this.temp = temp;
                 this.programCounter = programCounter;
           }
       }
```