Document downloaded from:

http://hdl.handle.net/10251/36674

This paper must be cited as:

Insa Cabrera, D.; Silva Galiana, JF.; Tomás Franco, C. (2013). Enhancing declarative debugging with loop expansion and tree compression. En Logic-Based Program Synthesis and Transformation. Springer Verlag (Germany). 71-88. doi:10.1007/978-3-642-38197-3_6.

The final publication is available at

http://link.springer.com/chapter/10.1007%2F978-3-642-38197-3_6

Copyright Springer Verlag (Germany)

Enhancing Declarative Debugging with Loop Expansion and Tree Compression*

David Insa, Josep Silva, and César Tomás

Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.
{dinsa,jsilva,ctomas}@dsic.upv.es

Abstract. Declarative debugging is a semi-automatic debugging technique that allows the programer to debug a program without the need to see the source code. The debugger generates questions about the results obtained in different computations and the programmer only has to answer them to find the bug. Declarative debugging uses an internal representation of programs called execution tree, whose structure highly influences its performance. In this work we introduce two techniques that optimize the execution trees structure. In particular, we expand and collapse the representation of loops allowing the debugger to find bugs with a reduced number of questions.

Keywords: Declarative Debugging, Tree Compression, Loop Expansion

1 Introduction

Debugging is one of the most difficult and less automated tasks of software engineering. This is due to the fact that bugs are usually hidden under complex conditions that only happen after particular interactions of software components. Programmers cannot consider all possible computations of their pieces of software, and those unconsidered computations usually produce a bug. In words of Brian Kernighan, the difficulty of debugging is explained as follows:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> The Elements of Programming Style, 2nd edition

The problems caused by bugs are highly expensive. Sometimes more than the product development price. For instance, the NIST report [17] calculated that

* This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. David Insa was partially supported by the Spanish *Ministerio de Eduación* under FPU grant AP2010-4415.

undetected software bugs produce a cost to the USA economy of $59 billion per year.

There have been many attempts to define automatic techniques for debugging, but in general with poor results. One notable exception is Declarative Debugging [15, 16]. In this work we present a technique to improve the performance of Declarative Debugging reducing the debugging time.

Declarative debugging is a semi-automatic debugging technique that automatically generates questions about the results obtained in subcomputations. Then, the programmer answers the questions and with this information the debugger is able to precisely identify the bug in the source code. Roughly speaking, the debugger discards parts of the source code associated with correct computations until it isolates a small part of the code (usually a function or procedure). One interesting property of this technique is that programmers do not need to see the source code during debugging. They only need to know the actual and intended results produced by a computation with given inputs. Therefore, if we have available a formal specification of the pieces of software that is able to answer the questions, then the technique is fully automatic.

In declarative debugging, a data structure called *Execution Tree* (ET) represents a program execution, where each node is associated with a particular method execution.[1] Moreover, declarative debugging uses a navigation strategy to select ET nodes, and to ask the programmer about their validity. Each node contains a particular method execution with its inputs and outputs. If the programmer marks a node as wrong, then the bug must be in its subtree; and it must be in the rest of the tree if it is marked as correct. We find a *buggy node* when the programmer marks a node as wrong, and all its children are marked as correct. Hence, the debugger reports the associated method as buggy.

Let us explain the technique with an example. Consider the Java program shown in Fig. 1. This program initializes the elements of a matrix to 1, and then traverses the matrix to sum all of them. The ET associated with this example is shown in Fig. 2 (left). Because it is not relevant in our technique, we omit the information of the nodes in the ETs of this paper, and instead we label them (inside) with the number of the method associated with them. Observe that `main` (0) calls once to the constructor `Matrix` (1), and then calls nine times to `position` (2) inside two nested loops. In the ET, every node has a number (outside) that indicates the number of questions needed to find the bug when it is the buggy node. To compute this number, we have considered the navigation strategy Divide and Query [15], that always selects the ET node that minimizes the difference between the numbers of descendants and non-descendants (i.e., better divides the ET by the half).

Declarative debugging can produce long series of questions making the debugging session too long. Moreover, it works at the level of methods, thus the

[1] Nodes represent computations. Hence, depending on the underlying paradigm, they can represent methods, functions, procedures, clauses, etc. Our technique can be applied to any paradigm, but, for the sake of concreteness, we will center the discussion on the object-oriented paradigm and our examples on Java.

```
   public class Matrix {
     private int numRows;
     private int numColumns;
     private int[][] matrix;

(1)  public Matrix(int numRows, int numColumns) {
       this.numRows = numRows;
       this.numColumns = numColumns;
       this.matrix = new int[numRows][numColumns];
       for (int i = 0; i < numRows; i++)
         for (int j = 0; j < numColumns; j++)
           this.matrix[i][j] = 1;
     }

(2)  public int position(int numRow, int numColumn) {
       return matrix[numRow][numColumn];
     }
   }

   public class SumMatrix {
(0)  public static void main(String[] args) {
       int result = 0;
       int numRows = 3;
       int numColumns = 3;
       Matrix m = new Matrix(numRows, numColumns);
       for (int i = 0; i < numRows; i++)
         for (int j = 0; j < numColumns; j++)
           result += m.position(i, j);
       System.out.println(result);
     }
   }
```

Fig. 1. Iterative Java program.

granularity level of the bug found is a method. In this work we propose two new techniques that reduce (i) the number of questions needed to detect a bug, (ii) the complexity of the questions, and also (iii) the granularity level of the bug found. The first technique improves a previous technique called *Tree Compression* [4], and the second one is a new technique named *Loop Expanssion*. While the first technique is based on a transformation of the ET, the second one is based on a transformation of the source code. In both cases, the produced ET can be debugged more efficiently using the standard algorithms. Therefore, the techniques are conservative with respect to previous implementations and they can be integrated in any debugger as a preprocessing stage. The cost of the transformations is low compared with the cost of generating the whole ET. In fact, they are efficient enough as to be always used in all declarative debuggers before the ET exploration phase. According to our experiments (summarized in Table 1), as an average, they reduce the number of questions in 27.65 % with a temporal cost of 274 ms for loop expansion and 123 ms for tree compression.

The rest of the paper has been organized as follows. In Section 2 we discuss the related work. Section 3 introduces some preliminary definitions that will be used in the rest of the paper. In Section 4 we explain our techniques and their main applications, and we introduce the algorithms that improve the structure of the ET. Then, in Section 6 we provide details about our implementation and some experiments carried out with real Java programs. Finally, Section 7 concludes.

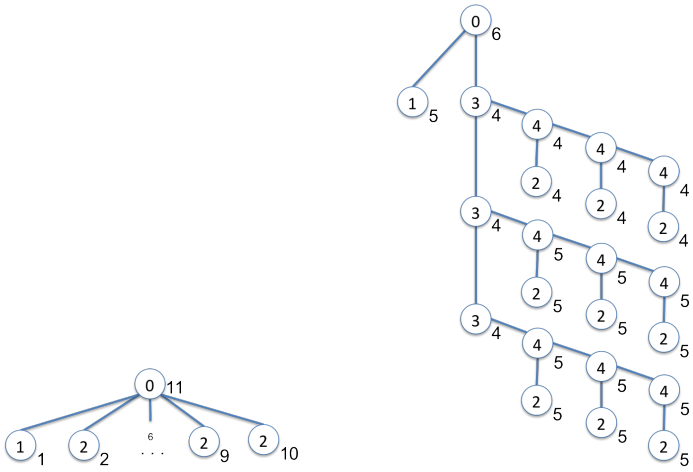

Fig. 2. ETs of the examples in Fig. 1 (left) and 8 (right).

2 Related Work

Reducing the number of questions asked by declarative debuggers is a well-known objective in the field, and there exist several works devoted to achieve this goal. Some of them face the problem by defining different ET transformations that modify the structure of the ET to explore it more efficiently.

For instance, in [14], authors improve the declarative debugging of Maude balancing the ET by introducing nodes. These nodes represent transitivity inferences done by their inference system. Although these nodes could be omitted, if they are kept, the ET becomes more balanced. Balanced ETs are very convenient for strategies such as Divide and Query [16], because it is possible to prune almost half of the tree after every answer, thus obtaining a logarithmic number of questions with respect to the number of nodes in the ET. This approach is related to our technique, but it has some drawbacks: it can only be applied where transitivity inferences took place while creating the ET, and thus most of the parts of the tree cannot be balanced, and even in these cases the balancing only affects two nodes. Our techniques, in contrast, balance loops, that usually contain many nodes.

Our first technique is based on Tree Compression (TC) introduced by Davie and Chitil [4]. In particular, we define an algorithm to decide when TC must be applied (or partially applied) in an ET. TC is a conservative approach that transforms an ET into an equivalent (smaller) ET where we can detect the same bugs. The objective of this technique is essentially different to previous ones: it tries to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls. For each recursive call, TC removes the child node associated with the recursive call and all its children become children of the parent node. Let us explain it with an example.

Example 1. Consider the ET in Fig. 3. Here, TC removes six nodes, thus statically reducing the size of the tree. Observe that the average number of questions has been reduced ($\frac{72}{17}$ vs $\frac{42}{11}$) thanks to the use of TC.

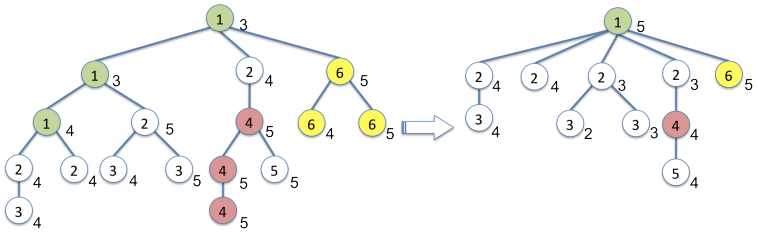

Fig. 3. Example of Tree Compression.

Unfortunately, TC does not always produce good results. Sometimes reducing the number of nodes causes a worse ET structure that is more difficult to debug and thus the number of questions is increased, producing the contrary effect to the intended one.

Example 2. Consider the ET in Fig. 4 (top). In this ET, the average number of questions needed to find the bug is $\frac{33}{9}$. Nevertheless, after compressing the recursive calls (the dark nodes), the average number of questions is augmented to $\frac{28}{7}$ (see the ET at the left). The reason is that in the new compressed ET we cannot prune any node because its structure is completely flat. The previous structure allowed us to prune some nodes because deep trees are more convenient for declarative debugging. However, if we only compress one of the two recursive calls, the number of questions is reduced to $\frac{27}{8}$ (see the ET at the right).

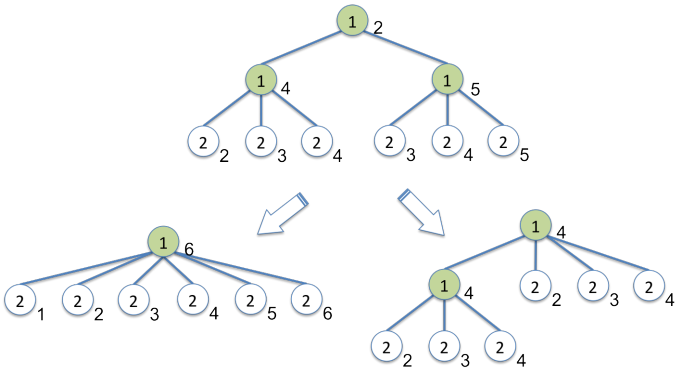

Fig. 4. Negative and positive effects of Tree Compression.

Example 2 clearly shows that TC should not be always applied. From the best of our knowledge, there does not exist an algorithm to decide when to apply TC, and current implementations always compress all recursive calls [3, 7]. Our new technique solves this problem with an analysis to decide when to compress them.

A similar approach to TC is declarative source debugging [2], that instead of modifying the tree prevents the debugger from selecting questions related to nodes generated by recursive calls. Another related approach was presented in [13]. Here, authors introduced a source code (rather than an ET) transformation for list comprehensions in functional programs. Concretely, this technique transforms list comprehensions into a set of equivalent methods that implement the iteration. The produced ET can be further transformed to remove the internal nodes reducing the size of the final ET as in the TC technique. Even though this technique is used in other paradigm and only works for a different program construct (list comprehensions instead of loops), it is very similar to our loop expansion technique because it transforms the program to implement the list comprehension iterations with recursive functions. This is somehow equivalent to our transformation for `for-each` loops. However, the objective of their technique is different. Their objective is to divide a question related to a list comprehension in different (probably easier) questions, while our objective is to balance the tree, and thus they are optimized in a different way. Of course, their transformation is orthogonal to our technique and it can be applied before.

Even though the techniques discussed can be applied to any language, they only focus on recursion. This means that they cannot improve ETs that use loops, avoiding their use in the imperative or the object-oriented paradigm where loops predominate. Our second technique is based on an automatic transformation of loops into recursive methods. Hence, it allows the previously discussed transformations to work in presence of iteration.

3 Preliminaries

Our ET transformations are based on its structure and the signature of the method in each node. Therefore, for the purpose of this work, we can provide a definition of ET whose nodes are labeled with a number referring to a specific method.

Definition 1 (Execution Tree). *An* execution tree *is a labeled directed tree* $T = (N, E)$ *whose nodes* N *represent method executions and are labeled with method identifiers, where the label of node* n *is referenced with* $l(n)$*. Each edge* $(n \to n') \in E$ *indicates that the method associated with* $l(n')$ *is invoked during the execution of the method associated with* $l(n)$*.*

We use numbers as method identifiers that uniquely identify each method in the source code. This simplification is enough to keep our definitions and algorithms precise and simple. Given an ET, *simple recursion* is represented with a branch of chained nodes with the same identifier. *Nested recursion* happens

when a recursive branch is descendant of another recursive branch. *Multiple recursion* happens when a node labeled with an identifier n has two or more children labeled with n.

The weight of a node is the number of nodes contained in the tree rooted at this node. We refer to the weight of node n as w_n. In the following, we will refer to the two most used navigation strategies for declarative debugging: Top-Down [12] and Divide and Query (D&Q) [15]. In both cases, we will always implicitly refer to the most efficient version of both strategies, respectively named, (i) *Heaviest First* [1], which always traverses the ET from the root to the leaves selecting always the heaviest node; and (ii) *Hirunkitti's Divide and Query* [6], which always selects the node in the ET that better divides the number of nodes in the ET by the half. A comparative study of these techniques can be found in [16].

For the comparison of strategies we use function $\mathcal{Q}uestions(T, s)$ that computes the number of questions needed (as an average) to find the bug in an ET T using the navigation strategy s.

4 Execution Trees Optimization

In this section we present two new techniques for the optimization of ETs: Tree Compression (TC) and Loop Expansion (LE). TC was defined and described in [4]. Here, we introduce an algorithm to compress a recursive branch of the ET in any case (i.e., simple recursion, nested recursion, and multiple recursion). We also discuss how navigation strategies are affected by TC. The other technique introduced is Loop Expansion that essentially transforms a loop into an equivalent recursive method. Then, TC adequately balances the iterations to obtain a new ET as optimized as possible. We explain each technique separately and propose algorithms for them that can work independently.

4.1 When to Apply Tree Compression

Tree compression was proposed as a general technique for declarative debugging. However, it was defined in the context of a functional language (Haskell) and with the use of a particular strategy (Hat-Delta). The own authors realized that TC can produce wide trees that are difficult to debug and, for this reason, defined strategies that avoid asking about the same method repeatedly. These strategies do not prevent to apply TC. They just assume that the ET has been totally compressed and they follow a top-down traversal of the ET that can jump to any node when the probability of this node to contain the bug is high. This way of proceeding somehow partially mitigates the bad structure of the produced ET when it is totally compressed. Our approach is radically different: We do not create a new strategy to avoid the bad ET structure; but we transform the ET to ensure a good structure.

Even though TC can produce bad ETs (as shown in Example 4), its authors did not study how this technique works with other (more extended) strategies

such as Top-Down or D&Q. So it is not clear at all when to use it. To study when to use TC, we can consider the most general case of a simple recursion in an ET. It is shown in Fig. 5 where clouds represent possibly empty sets of subtrees and the dark nodes are the recursion branch with a length of $n \geq 2$ calls.

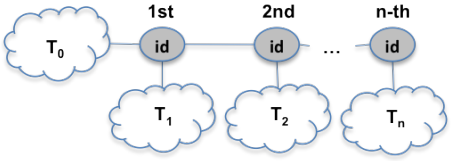

Fig. 5. Recursion branch in an ET.

It should be clear that the recursion branch can be useful to prune nodes of the tree. For instance, in the figure, if we ask for the node $n/2$, we prune $n/2$ subtrees. Therefore, in the case that the subtrees $T_i, 1 \leq i \leq n$, are empty, then no pruning is possible. In that case, only the nodes in the recursion branch could be buggy; but, because they form a recursive chain, all of them have the same label. Thus no matter which one is buggy because all of them refer to the same method. Hence, TC must be used to reduce the recursive branch to a single node avoiding navigation strategies to explore this branch. Hence, we can conclude that *we must compress every node whose only child is a recursive call.* This result can be formally stated for Top-Down as follows.

Theorem 1. *Let T be an ET with a recursion branch $R = n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ where the only child of a node n_i, $1 \leq i \leq m-1$, is n_{i+1}. And let T' be an ET equivalent to T except that nodes n_i and n_{i+1} have been compressed. Then, $\mathcal{Q}uestions(T', \textit{Top-Down}) < \mathcal{Q}uestions(T, \textit{Top-Down})$.*

The associated proof can be found in [10].

This theorem shows us that, in some situations, TC must be used to statically improve the ET structure. But TC is not the panacea, and we need to identify in what cases it should be used. D&Q is a good example of strategy where TC has a negative effect.

Tree Compression for D&Q

In general, when debugging an ET with the strategy D&Q, TC should only be applied in the case described by Theorem 1 ($T_i, 1 \leq i \leq n$, are empty). The reason is that D&Q can jump to any node of the ET without following a predefined path. This allows D&Q to ask about any node of the recursion branch without asking about the previous nodes in the branch. Note that this does not happen in other strategies such as Top-Down. Therefore, D&Q has the ability to use the recursion branch as a mean to prune half of the iterations.

Observe in Fig. 6 that, except for very small recursion branches (e.g., $n \leq 3$), D&Q can take advantage of the recursion branch to prune half of the iterations. The greater is n the more nodes pruned. Observe that D&Q can prune nodes even in the case when there is only one child for every node in the recursion branch (e.g., $T_i, 1 \leq i \leq n$, is a single node). Therefore, if we add more nodes to the subtrees Ti, then more nodes can be pruned and D&Q will behave even better.

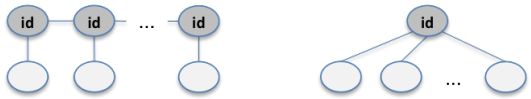

Fig. 6. TC applied to a recursive method.

Tree Compression for Top-Down

In the case of Top-Down-based strategies, it is not trivial at all to decide when to apply TC. Considering again the ET in Fig. 5, there are two factors that must be considered: (i) the length n of the recursive branch, and (ii) the size of the trees $T_i, 1 \leq i \leq n$. In order to decide when TC should be used, we provide Algorithm 1 that takes an ET and compresses all recursion branches whenever it improves the ET structure.

Essentially, Algorithm 1 analyzes for each recursion what is the effect of applying TC, and it is finally applied only when it produces an improvement. This analysis is done little by little, analyzing each pair of parent-child (recursive) nodes in the sequence separately. Thus, it is possible that the final result is to only compress one (or several) parts of one recursion branch. For this, variable `recs` initially contains all nodes of the ET with a recursive child. Each of these nodes is processed with the loop in line 1 bottom-up (lines 2-3). That is, the nodes closer to the leaves are processed first. In order to also consider multiple recursion, the algorithm uses the loops in lines 5 and 8. These loops store in the variable `improvement` the improvement achieved when compressing each recursive branch. In addition to the functions (`Cost` and `Compress`) shown here, the algorithm uses three more functions whose code has not been included because they are trivial: function `Children` computes the set of children of a node in the ET (i.e., `Children(m) = {n | (m → n) ∈ E}`); function `Sort` takes a set of nodes and produces an ordered sequence where nodes have been decreasingly ordered by their weights; and function `Pos` takes a node and a sequence of nodes and returns the position of the node in the sequence.

Given two nodes `parent` and `child` candidates to make a TC, the algorithm first sorts the children of both the `parent` and the `child` (lines 9-10) in the order in which Top-Down would ask them (sorted by their weight). Then, it combines the children of both nodes simulating a TC (line 11). Finally, it compares the

Algorithm 1 Optimized Tree Compression

Input: An ET $T = (N, E)$
Output: An ET T'
Inicialization: $T' = T$ and $recs = \{n \mid n, n' \in N \wedge (n \rightarrow n') \in E \wedge l(n) = l(n')\}$

begin
1) **while** ($recs \neq \emptyset$)
2) **take** $n \in recs$ such that $\nexists n' \in recs$ with $(n \rightarrow n') \in E^+$
3) $recs = recs \backslash \{n\}$
4) $parent = n$
5) **do**
6) $maxImprovement = 0$
7) $children = \{c \mid (n \rightarrow c) \in E \wedge l(n) = l(c)\}$
8) **for each** $child \in children$
9) $pchildren = \text{Sort}(\text{Children}(parent))$
10) $cchildren = \text{Sort}(\text{Children}(child))$
11) $comb = \text{Sort}((pchildren \cup cchildren) \backslash \{child\})$
12) $improvement = \frac{Cost(pchildren)+Cost(cchildren)}{w_{parent}} - \frac{Cost(comb)}{w_{parent}-1}$
13) **if** ($improvement > maxImprovement$)
14) $maxImprovement = improvement$
15) $bestNode = child$
16) **end for each**
17) **if** ($maxImprovement \neq 0$)
18) $T' = \text{Compress}(T', parent, bestNode)$
19) **while** ($maxImprovement \neq 0$)
20) **end while**
end
return T'

function Cost($sequence$)
begin
21) **return** $\sum\{\text{Pos}(node, sequence) * w_{node} \mid node \in sequence\} + |sequence|$
end

function Compress($T = (N, E), parent, child$)
begin
22) $nodes = \text{Children}(child)$
23) $E' = E \backslash \{(child \rightarrow n) \in E \mid n \in nodes\}$
24) $E' = E' \cup \{(parent \rightarrow n) \mid n \in nodes\}$
25) $N' = N \backslash \{child\}$
end
return $T' = (N', E')$

average number of questions when compressing or not the nodes (line 12). The equation that appears in line 12 is one of the main contributions of the algorithm, because this equation determines when to perform TC between two nodes in a branch with the strategy Top-Down. This equation depends in turn on the

formula (line 21 in function Cost) used to compute the average cost of exploring an ET with Top-Down.

If we analyze Algorithm 1, we can easily realize that its asymptotic cost is quadratic with the number of recursive calls $\mathcal{O}(N^2)$ because in the worst case, all recursive calls would be compared between them. Note also that the algorithm could be used with incomplete ETs [8] (this is useful when we try to debug a program while the ET is being generated). In this case, the algorithm can still be applied locally, i.e., to those subtrees of the ET that are totally generated.

4.2 Loop Expansion

Recursive calls group the iterations in different subtrees whose roots belong to the recursion branch. This is very convenient because it allows the debugger to prune different iterations. Therefore, recursion is beneficial for declarative debugging except in the cases discussed in the previous section. Contrarily, loops produce very wide trees where all iterations are represented as trees with a common root. In this structure, it is impossible to prune more than one iteration at a time, being the debugging of these trees very expensive.

To solve this problem, in this section we present a technique for declarative debugging that transforms loops into equivalent recursive methods. Because iteration is more efficient than recursion, there exist many approaches to transform recursive methods into equivalent loops (e.g., [5, 11]). However, there exist few approaches to transform loops into equivalent recursive methods. An exception is the one presented in [18] to improve performance in multi-level memory hierarchies. Nevertheless, we are not aware of any algorithm of this kind proposed for Java or for any other object-oriented language. Hence, we had to implement this algorithm as a Java library and made it public for the community: http://users.dsic.upv.es/∼jsilva/loops2recursion/. Moreover, it has been also integrated into a declarative debugger [7]. Due to lack of space we cannot describe here the algorithm, but we made a technical report with a detailed description [9]. This algorithm has an asymptotic cost linear with the number of loops in the program and is the basis of LE. Basically, it transforms each loop into an equivalent recursive method. The transformation is slightly different for each kind of loop (`while`, `do`, `for` or `foreach`). In the case of `for`-loops, it can be explained with the code in Fig. 7 where A, B, C and D represent blocks of code. If we observe the transformed ET we see that each iteration is represented with a different node of the recursive branch $r(1) \rightarrow r(2) \rightarrow \ldots \rightarrow r(10)$, thus it is possible to detect a bug in a single iteration. This means that, in the case that function f had a bug, thanks to the transformation, the debugger could detect that a bug exists in the code in B + C or in A + D. Note that this is not possible in the original ET where the debugger would report that A + B + C + D has a bug. This is a very important result because it augments the granularity level of the reported bugs, detecting bugs inside loops and not only inside methods.

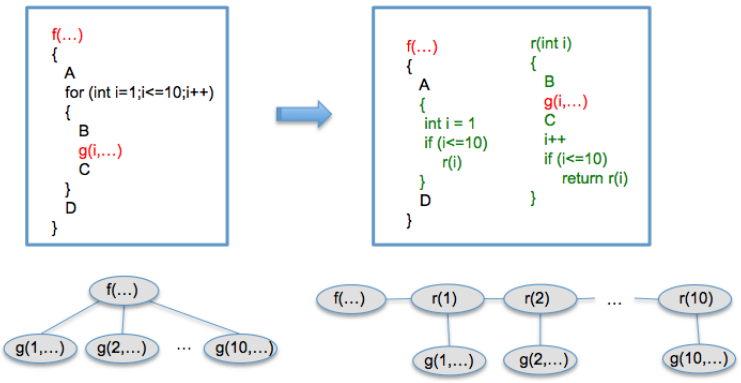

Fig. 7. ET transformation from a loop to a recursive method.

Nested recursion augments the possibilities of pruning. For instance, class Matrix in Fig. 1 can be automatically transformed[2] to the code in Fig. 8. The ET associated with the transformed program is shown in Fig. 2 (right). Observe that there is a recursion branch for each executed loop, and thus, we have recursive branches (those labelled with 4) inside a recursive branch (labelled with 3). Hence, the new nodes added by the transformation are used to represent each single iteration; and thanks to them, now it is possible to prune loops, iterations, or single calls inside an iteration.

Our implementation combines the use of TC and LE as follows: (i) Expand all loops of the source code with LE, (ii) generate the ET with the transformed code, and (iii) use Algorithm 1 to compress the ET with the current strategy—observe that if the strategy is later changed, all nodes removed by the compression can be introduced again, and the ET compressed for the new strategy—. In this way, we produce an ideal representation of loops where each individual loop is partially or totally expanded to produce an optimal debugging session.

5 Correctness

In this section we prove that after our transformations, all bugs that could be detected in the original ET can still be detected in the transformed one. An even more interesting result is that the transformed ET can contain more buggy nodes than the original one, and thus, we can detect bugs that before

[2] For the sake of clarity, in the figure we replaced the names of the generated recursive methods by `sumRows` and `sumColumns`. In the implementation, if a loop has a Java label in the original source code, the transformation uses this label to name the recursive method. If this label does not exist, then the name of the loop is the name of the method that contains this loop followed by "_loopN", where N is an autonumeric. While debugging, the user can see the source code of the loop, and she can change its name if she wants to do it.

```
   public class SumMatrix {
(0)  public static void main(String[] args) {
       int result = 0;
       int numRows = 3;
       int numColumns = 3;
       Matrix m = new Matrix(numRows, numColumns);
       // For loop
       { // Init for loop
          int i = 0;
          // First iteration
          if (i < numRows) {
             Object[] res = SumMatrix.sumRows(m, i, numRows, numColumns, result);
             result = (Integer)res[0];
          }
       }
       System.out.println(result);
     }
(3)  private static Object[] sumRows(Matrix m, int i, int numRows, int numColumns, int result) {
       // For loop
       { // Init for loop
          int j = 0;
          // First iteration
          if (j < numColumns) {
             Object[] res = SumMatrix.sumColumns(m, i, j, numColumns, result);
             result = (Integer)res[0];
          }
       }
       // Update for loop
       i++;
       // Next iteration
       if (i < numRows)
          return SumMatrix.sumRows(m, i, numRows, numColumns, result);
       return new Object[]{result};
     }
(4)  private static Object[] sumColumns(Matrix m, int i, int j, int numColumns, int result) {
       result += m.position(i, j);
       // Update for loop
       j++;
       // Next iteration
       if (j < numColumns)
          return SumMatrix.sumColumns(m, i, j, numColumns, result);
       return new Object[]{result};
     }
   }
```

Fig. 8. Recursive version of the program in Fig. 1.

were undetectable. Regarding TC, its correctness has been proved in [4]. Our algorithm does not influence this correctness property because it only decides what nodes should be compressed, but the TC algorithm is the standard one. The correctness of LE is stated in the following. The associated proofs can be found in [10].

Theorem 2. *[Completeness] Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. For each buggy node in T, there is at least one buggy node in T'.*

Theorem 3. *[Soundness] Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. If T' contains a buggy node associated with code $f \subseteq \mathcal{P}$, then T contains a buggy node associated with code $g \subseteq \mathcal{P}$ and $f \subseteq g$.*

From Theorem 2 and 3 we have a very interesting corollary that reveals that the transformed tree can find more bugs than the original ET.

Benchmark	Nodes				LE	Time		Questions				%	
	ET	LE	TC_{ori}	TC_{opt}		LE	TC	ET	LE	TC_{ori}	TC_{opt}	LETC	TC
Factoricer	55	331	51	51	5	151	105	11.62	8.50	7.35	7.35	63.25	100.0
Classifier	25	57	22	24	3	184	4	8.64	6.19	6.46	6.29	72.80	97.36
LegendGame	87	243	87	87	10	259	31	12.81	8.28	11.84	11.84	92.43	100.0
Romanic	121	171	112	113	3	191	12	16.24	7.74	10.75	9.42	58.00	87.62
FibRecursive	5378	6192	98	101	12	251	953	15.64	12.91	9.21	8.00	51.15	86.86
FactTrans	197	212	24	26	3	181	26	10.75	7.88	6.42	5.08	47.26	79.13
BinaryArrays	141	203	100	100	5	172	79	12.17	7.76	7.89	7.89	64.83	100.0
FibFactAna	178	261	44	49	7	202	33	7.90	8.29	8.50	6.06	76.71	71.29
RegresionTest	13	121	15	15	5	237	4	4.77	7.17	4.20	4.20	88.05	100.0
BoubleFibArrays	16	164	10	10	10	213	27	9.31	8.79	4.90	4.90	52.63	100.0
StatsMeanFib	19	50	23	23	6	195	21	7.79	8.12	6.78	6.48	83.18	95.58
Integral	5	8	8	8	3	152	2	6.80	5.75	7.88	5.88	86.47	74.62
TestMath	3	5	3	3	3	195	2	7.67	6.00	9.00	7.67	100.0	85.22
TestMath2	92	2493	13	13	3	211	607	14.70	11.54	15.77	12.77	86.87	80.98
Figures	2	10	10	10	24	597	13	9.00	7.20	6.60	6.60	73.33	100.0
FactCalc	128	179	75	75	3	206	46	8.45	7.60	7.96	7.96	94.20	100.0
SpaceLimits	95	133	98	100	15	786	10	36.26	12.29	18.46	14.04	38.72	76.06

Table 1. Summary of the experiments.

Corollary 1. *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. If T contains n buggy nodes, then T' contains n' buggy nodes with $n \leq n'$.*

6 Implementation

We have implemented the original TC algorithm and the optimized version presented in this paper; and also the LE algorithm in such a way that they all can work together. This implementation has been integrated into the Declarative Debugger for Java DDJ [7]. The experiments, the source code of the tool, the benchmarks, and other materials can be found at http://www.dsic.upv.es/∼jsilva/DDJ/.

All the implementation has been done in Java. The optimized TC algorithm contains around 90 LOC, and the LE algorithm contains around 1700 LOC. We conducted a series of experiments in order to measure the influence of both techniques in the performance of the debugger. Table 1 summarizes the obtained results.

The first column in Table 1 shows the name of the benchmarks. For each benchmark, column `nodes` shows the number of nodes descendant of a loop[3] in the original ET (`ET`), in the ET after applying LE (`LE`), in the ET after applying LE first and then the original version of TC—compressing all nodes—(TC_{ori}), and in the ET after applying LE first and then the optimized version of TC—Algorithm 1—(TC_{opt}); column `LE` shows the number of loops expanded; column

[3] We consider these nodes because the part of the ET that is not descendant of a loop remains unchanged after applying our technique, and thus the number of questions needed to find the bug is the same before and after the transformations.

`Time` shows the time (in milliseconds) needed to apply LE and TE; column `Questions` shows the average number of questions asked with each of the previously described ETs. Each benchmark has been analyzed assuming that the bug could be in any node of its associated ET. This means that each value in column `Questions` represents the average of a set of experiments. For instance, in order to obtain the information associated with `Factoricer`, this benchmark has been debugged 55 times with the original ET, 331 with the ET after applying loop expansion, etc. In total, `Factoricer` was debugged 55+331+51+51=488 times, considering all ET transformations and assuming each time that the bug was a different node (and computing the average of all tests for each ET); finally, column `(%)` shows, on the one hand, the percentage of questions asked after applying our transformations (LE and TC) with respect to the original ET (`LETC`); and, on the other hand, the percentage of questions asked using Algorithm 1 to decide when to apply TC, with respect to always applying TC (`TC`). From the table we can conclude that our transformations produce a reduction of 27.65 % in the number of questions asked by the debugger. Moreover, the use of Algorithm 1 to decide when to apply TC also produces an important reduction in the number of questions with an average of 9.72 %.

7 Conclusions

Declarative debugging can generate too many questions to find a bug, and once it is found, the debugger reports a whole method as the buggy code. In this work we make a step forward to solve these problems. We introduce techniques that reduce the number of questions by improving the structure of the ET. This is done with two transformations called tree compression and loop expansion. Moreover, loop expansion also faces the second problem, and it allows us to detect bugs in loops and not only in methods, augmenting in this way the granularity level of the bug found. As a side effect, being able to ask about the correctness of loops allows us to reduce the complexity of questions: the programmer can answer about a part of a method (a single loop), and not only to the whole method mixing the effects of different loops. We think that this is an interesting result that opens new possibilities for future work related to the complexity of questions. The idea of transforming loops into recursive methods in an ET is novel, and it allows us to apply all previous techniques based on recursion in the imperative and object-oriented paradigms.

8 Acknowledgements

We want to thank Rafael Caballero and Adrián Riesco for productive comments and discussions at the initial stages of this work. We also thank the anonymous referees of LOPSTR'12 for useful feedback and constructive criticism which has improved this work.

References

1. Dominic Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.
2. Miguel Calejo. *A Framework for Declarative Prolog Debugging*. PhD thesis, New University of Lisbon, 1992.
3. Thomas Davie and Olaf Chitil. Hat-delta: One Right Does Make a Wrong. In Andrew Butterfield, editor, *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, page 11, sep 2005.
4. Thomas Davie and Olaf Chitil. Hat-delta: One Right Does Make a Wrong. In *Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06)*, apr 2006.
5. Peter George Harrison and Hessam Khoshnevisan. A new approach to recursion removal. *Electronic Notes in Theoretical Computer Science*, 93(1):91–113, feb 1992.
6. Visit Hirunkitti and Christopher John Hogger. A Generalised Query Minimisation for Program Debugging. In *Proceedings of the 1st International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749, 1993.
7. David Insa and Josep Silva. An Algorithmic Debugger for Java. *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 0:1–6, 2010.
8. David Insa and Josep Silva. Debugging with Incomplete and Dynamically Generated Execution Trees. In *Proceedings of the 20th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2010)*, 2010.
9. David Insa and Josep Silva. A Transformation of Iterative Loops into Recursive Loops. Technical Report DSIC/05/12, Universidad Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/∼jsilva/research.htm#techs.
10. David Insa, Josep Silva, and César Tomás. Enhancing Declarative Debugging with Loop Expansion and Tree Compression. Technical Report DSIC/11/12, Universidad Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/∼jsilva/research.htm#techs.
11. Yanhong A. Liu and Scott D. Stoller. 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)*, pages 73–82, New York, NY, USA, 2000. ACM.
12. John Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
13. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, may 1998.
14. Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. Declarative Debugging of Rewriting Logic Specifications. *Journal of Logic and Algebraic Programming (JLAP)*, 2011.
15. E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
16. Josep Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, nov 2011.
17. Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3, National Institute of Standards and Technology (NIST), may 2002.

18. Qing Yi, Vikram Adve, and Ken Kennedy. 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)*, pages 169–181, 2000.

Appendix 1: Proofs of technical results

This section presents the proofs of Theorems 1, 2, 3 and Corollary 1.

Theorem 1 *Let T be an ET with a recursion branch $R = n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ where the only child of a node n_i, $1 \leq i \leq m-1$, is n_{i+1}. And let T' be an ET equivalent to T except that nodes n_i and n_{i+1} have been compressed. Then, $\mathcal{Q}uestions(T', \textit{Top-Down}) < \mathcal{Q}uestions(T, \textit{Top-Down})$.*

Proof. Let us consider the two nodes that form the sub-branch to be compressed. For the proof we can call them $n_1 \rightarrow n_2$. Firstly, the number of questions needed to find a bug in any ancestor of n_1 is exactly the same if we compress or not the sub-branch. Therefore, it is enough to prove that $\mathcal{Q}uestions(T_{1c}$,Top-Down$) < \mathcal{Q}uestions(T_1$,Top-Down) where T_1 is the subtree whose root is n_1 and T_{1c} is the subtree whose root is n_1 after tree compression.
Let us assume that n_2 has j children. Thus we call T_2 the subtree whose root is n_2, and T_{2_i} the subtree whose root is the i-th child of n_2. Then,

$$\mathcal{Q}uestions(T_2\text{,Top-Down}) = \frac{(j+1)+\sum_{i=1}^{j} |T_{2_i}| * (i+\mathcal{Q}uestions(T_{2_i}\text{,Top-Down}))}{|T_2|}$$

Here, $(j+1)$ are the questions needed to find a bug in n_2. To reach the children of n_2, the own n_2 and the previous $i-1$ children must be asked first, and this is why we need to add i to $\mathcal{Q}uestions(T_{2_i}$,Top-Down), Finally, $|T_x|$ represents the number of nodes in the (sub)tree T_x.
Therefore, $\mathcal{Q}uestions(T_{1c}$,Top-Down$) = \mathcal{Q}uestions(T_2$,Top-Down)
and $\mathcal{Q}uestions(T_1$,Top-Down$) = \frac{2+|T_2|*(i+\mathcal{Q}uestions(T_2\text{,Top-Down}))}{|T_2|+1}$
Clearly, $\mathcal{Q}uestions(T_{1c}$,Top-Down$) < \mathcal{Q}uestions(T_1$,Top-Down), and thus the claim follows.

Theorem 2 *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. For each buggy node in T, there is at least one buggy node in T'.*

Proof. Let us prove the theorem for an arbitrary buggy node n in T associated with a function f. Firstly, because loop expansion only transforms iterative loops into recursive loops, all functions executed in T are also executed in T'. This means that every node in T has a counterpart (equivalent) node in T' that represents the same (sub)computation. Therefore, we can call n' to the node that represents in T' the same execution than n in T. Because n is buggy, then n is wrong, and all the children of n (if any) are correct. Hence, n' is also wrong. Moreover, if f does not contain a loop, then loop expansion has no effect on the code of f and thus n and n' will have exactly the same children, and thus, trivially, n' is also buggy in T'. If we assume the existence of a loop in f, then we will have a situation as the one shown in the ETs of Figure 7. We can consider for the proof that the ET at the left is the subtree of n and the ET at the right is the subtree of n'. Then, because n is buggy, all nodes labeled with g are correct (in both ETs) and, thus, either n' or one of the nodes labeled with r are buggy.

Theorem 3 *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. If T' contains a buggy node associated with code $f \subseteq \mathcal{P}$, then, T contains a buggy node associated with code $g \subseteq \mathcal{P}$ and $f \subseteq g$.*

Proof. According to the proof of Theorem 2, every node in T has a counterpart (equivalent) node in T'. Hence, let n be the buggy node in T and let n' be the associated buggy node in T'. If f does not have a loop, then both n and n' point to the same function (f) and thus the theorem holds trivially. If f contains a loop that has been expanded, then, as stated in the proof of Theorem 2, either n' or one of its descendants (say n'') that represent the iterations of the loop are buggy. But we know that the code of n'' is the code of the loop that is included in the code of f. Therefore, in all cases $f \subseteq g$.

Corollary 1 *Let $\mathcal{P}$ be a program, let T be the ET associated with $\mathcal{P}$, and let T' be the ET obtained by applying loop expansion to T. If T contains n buggy nodes, then T' contains n' buggy nodes with $n \leq n'$.*

Proof. Trivial from Theorems 2 and 3. On the one hand, equality is ensured with Theorem 2 because for each buggy node in T, there is at least one buggy node in T'. On the other hand, if a node in T is associated with a function whose code contains more than one loop that has been expanded, then T' can contain more than one new buggy node not present in T.