White-Box Path Generation in Recursive Programs

Ricardo Peña(✉) and Jaime Sánchez-Hernández

Computer Science School, Complutense University of Madrid, Madrid, Spain
{ricardo,jaime}@sip.ucm.es

Abstract. We present an algorithm for generating paths through a set of mutually recursive functions. The algorithm is part of a tool for white-box test-case generation. While in imperative programs there is a well established notion of path depth, this is not the case in recursive programs. We define what we mean by path and path depth in these programs and propose an algorithm which generates all the static paths up to a given depth. When the algorithm is applied to transformed iterative programs, giving as a result tail-recursive functions, the defined depth corresponds to the maximum number of times the loop condition is evaluated. When applied to non-tail recursive functions, the meaning is their maximum unfolding depth along the path, for each initial call to the function. It can also be applied to hybrid programs where both iteration and recursion are present.

Keywords: White-box testing · Static path · Recursion depth

1 Introduction

Testing is very important for increasing program reliability. Thorough testing ideally exercises all the different situations described in the specification, and all the instructions and conditions of the program under test, so that it would have a high probability of finding bugs, if they are present in the code. There is a general agreement that automatic tools can alleviate most of the tedious and error prone activities related to testing. One of them is test-case generation (TCG). Traditionally (see, for instance [1]), there are two TCG variants: black-box TCG and white-box TCG. In the first one, test-cases are solely based on the program specification, and in the second one, they are additionally based on a particular reference implementation. Each one complements each other, so both are needed if we aim at performing thorough testing.

White-box TCG is concerned with first defining a coverage criterion for the Unit Under Test (UUT), and then generating a set of test-cases which, when executed, will implement this criterion. A usual criterion is to require the test

Research paper. Partially funded by the Spanish Ministry of Science, Innovation and Universities, under the grant TIN2017-86217-R.

suite to exercise all the statements or all the condition outcomes of the UUT. A more complete criterion is to exercise all the execution paths through the UUT. When there exist loops in the code, the number of paths is potentially infinite, so a limit on the number of allowed iterations executed in each loop must be established. This limit is usually referred to as the path *depth*. For instance, depth-1 paths will be those that never execute the loop bodies; depth-2 paths will execute each loop body at most once; and so on. Once the paths are generated, each one is defined by the sequence of decisions that the UUT takes when executing that path. By collecting the conditions involved, and the expected outcome of evaluating each one, a test-case for the path can be synthesised by using for instance symbolic execution [2].

Functional programs do not contain loops, but use recursion instead. There is no established criterion on what a path and its path depth mean in a recursive program. Intuitively, a path should require a complete execution of the UUT main function and it would involve a sequence of decisions taken by the UUT along the execution. In a path, the UUT functions may be recursively invoked a number of times, either directly or indirectly. So, multiple and mutual recursion should be taken into account when defining the meaning of a path. By analogy to the iterative case, in recursive programs the path depth should be related to the number of invocations the UUT, or any of its auxiliary functions, undergo in the path.

In this work, we define a UUT to be a collection of (potentially) mutually recursive functions with a main, or top, visible one, which may invoke any of the other ones. Then, we define a notion of path and of path depth for that UUT, and present an algorithm for generating an exhaustive set of paths up to a given depth.

This work has been developed in the context of our computer assisted validation platform CAVI-ART [5,6]. The platform uses a functional language as its Intermediate Representation (IR) to which both imperative and functional programs are translated. Imperative loops are translated into a set of mutually tail recursive functions and, if recursion is present in the input program, it is preserved by the translation. In the end, only recursion remains in the IR. Given a UUT, the platform automatically generates white-box paths, and synthesises a test-case for each one, by using an SMT solver [7] which checks the satisfiability of the conditions required by the path, and assigns appropriate values to the involved variables.

In Fig. 1 we show a picture of the complete testing system. As the platform is also intended for formal verification, the UUT visible function is endowed with a formal precondition and postcondition. These are translated by the tool into SMT constraints. In this way, the test cases generated by the tool, not only satisfy the path constraints, but also the precondition ones. By requiring to only satisfy the precondition constraints, the tool can also generate black-box test cases. In our approach, the user fixes some sizes for the data types of the UUT arguments (for instance, the range of the integers, the length of an array, the cardinal of a tree, etc.), and the tool generates an exhaustive set of cases up to the

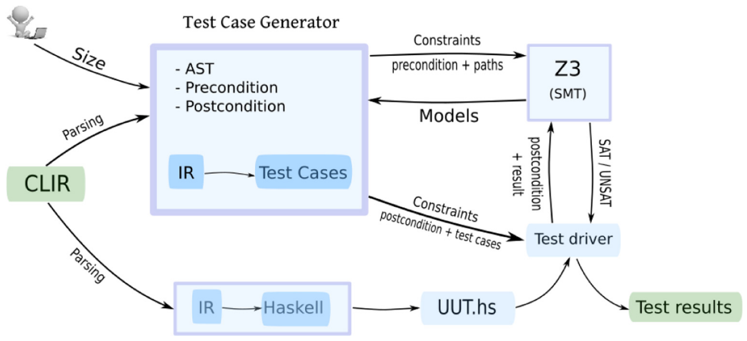


Fig. 1. CAVI-ART testing tool

given size. The reason for fixing these sizes is that we seek for exhaustiveness but also for getting a finite number of cases. Additionally, the validity of the test cases is automatically checked by the tool, just by checking whether the result returned by the UUT satisfies the postcondition constraints. So, the user responsibilities are only to provide a precondition and a postcondition, to fix sizes for the arguments and to choose a maximum depth for the white-box paths. After that, the whole testing process—test case generation, test execution and test validation—is automatically performed by the tool.

In principle, the IR language is not executable. In order to actually run the tests, the tool provides a translation from the IR to Haskell (see the bottom part of Fig. 1). The generic test driver is also written in this language. The translation is straightforward since, as we have already said, the IR is in fact a functional language.

2 Functional Intermediate Representation

Our intermediate representation is a first-order eager core functional language which supports mutually recursive function definitions. In Fig. 2 we show its abstract syntax. Notice that all expressions are flattened in the sense that all the arguments in function and constructor applications, and also the **case** discriminants, are atoms. An additional feature is that IR programs are in **let** A-normal form [4], and also in SSA[1] form, i.e. all **let** bound variables in a nested sequence of **let** expressions are distinct, and also different to the function arguments.

In Fig. 3 we show a Java version of the *quicksort* algorithm. This algorithm is translated by the platform into the IR code shown in Fig. 4. As there is no mutable state in the IR, the array updates of the Java version are simulated by passing an array as an argument, both to the *partition* and *qsort* functions,

[1] Static Single Assignment.

$$
\begin{array}{lll}
a & ::= c & \{\text{ constant }\} \\
 & \mid x & \{\text{ variable }\} \\
\mathit{be} & ::= a & \{\text{ atomic expression }\} \\
 & \mid f\ \overline{a_i} & \{\text{ function/primitive operator application }\} \\
 & \mid \langle\overline{a_i}\rangle & \{\text{ tuple construction }\} \\
 & \mid C\ \overline{a_i} & \{\text{ constructor application }\} \\
e & ::= \mathit{be} & \{\text{ binding expression }\} \\
 & \mid \mathbf{let}\ \langle\overline{x_i :: \tau_i}\rangle = \mathit{be}\ \mathbf{in}\ e & \{\text{ sequential let. Left part of the binding can be a tuple }\} \\
 & \mid \mathbf{letfun}\ \overline{\mathit{def}_i}\ \mathbf{in}\ e & \{\text{ let for mutually recursive function definitions }\} \\
 & \mid \mathbf{case}\ a\ \mathbf{of}\ \overline{\mathit{alt}_i}[;\ \_ \rightarrow e] & \{\text{ case distinction with optional default branch }\} \\
\mathit{tldef} & ::= \mathbf{define}\ \{\psi_1\}\ \mathit{def}\ \{\psi_2\} & \{\text{ top level function definition with pre- and post-conditions }\} \\
\mathit{def} & ::= f\ (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau_j}) = e & \{\text{ function definition. Output results are named }\} \\
\mathit{alt} & ::= C\ \overline{x_i :: \tau_i} \rightarrow e & \{\text{ case branch }\} \\
\tau & ::= \alpha & \{\text{ type variable }\} \\
 & \mid T\ \overline{\tau_i} & \{\text{ type constructor application }\}
\end{array}
$$

Fig. 2. CAVI-ART IR abstract syntax

```
public class Quick {
  public static void quicksort (int [] v) {
     int n = v.length;
     qsort (v, 0, n-1);
  }
  public static int partition (int [] v, int a, int b) {
    int i = a+1; int j = b; int piv = v[a];
    while (i <= j) {
      if (v[i] <= piv) {
         i = i+1;
      } else if (v[j] >= piv) {
         j = j-1;
      } else { // v[i] > piv && v[j] < piv
         int temp = v[i]; v[i] = v[j]; v[j] = temp;
         i = i+1; j = j-1;
      }
    }
    int temp = v[a]; v[a] = v[j]; v[j] = temp;
    return j;
  }
  public static void qsort (int [] v, int a, int b)  {
    if (a < b) {
       int p = partition(v, a, b);
       qsort(v, a, p-1);
       qsort(v, p+1, b);
    }
  }
}
```

Fig. 3. A Java version of *quicksort*

```
define quicksort (v::Array Int)::(res::Array Int) =
  letfun
    qsort (v::Array Int, a::Int, b::Int)::(res::Array Int) =
      let b::Bool = a < b in
      case b of
        True  -> f1 v a b
        False -> v
    f1 (v::Array Int, a::Int, b::Int)::(res::Array Int) =
      let (p::Int, v1::Array Int) = partition v a b in
      let p1::Int = p - 1 in
      let v2::Array Int = qsort v1 a p1 in
      let p2::Int = p + 1 in
        qsort v2 p2 b
    partition (v::Array Int, a::Int, b::Int)::(p::Int, res::Array Int) =
      let i::Int = a + 1 in
      let j::Int = b in
      let piv::Int = get-array v a in
        f2 v a piv i j
    f2 (v::Array Int, a::Int, piv::Int, i::Int, j::Int)::(res::Array Int) =
      let b::Bool = i <= j in
      case b of
        True  -> let ei::Int = get-array v i in
                 let b2::Bool = ei <= piv in
                 case b2 of
                   True  -> let i2::Int = i + 1 in
                              f2 v a piv i2 j
                   False -> let ej::Int = get-array v j in
                            let b3::Bool = ej >= piv in
                            case b3 of
                              True  -> let j2::Int = j - 1 in
                                         f2 v a piv i j2
                              False -> let temp::Int = get-array v i in
                                       let ej::Int = get-array v j in
                                       let v1::Array Int = set-array v i ej in
                                       let v2::Array Int = set-array v1 j temp in
                                       let i2::Int = i + 1 in
                                       let j2::Int = j - 1 in
                                         f2 v2 a piv i2 j2
        False -> let temp::Int = get-array v a in
                 let ej::Int = get-array v j in
                 let v1::Array Int = set-array v a ej in
                 let v2::Array Int = set-array v1 j temp in
                   (j, v2)
  in let n::Int  = length v in
     let n1::Int = n - 1    in
       qsort v 0 n1
```

Fig. 4. CAVI-ART IR for function *quicksort*

and returning a different one as a result. The predefined function `set-array`[2] simulates the assignments of the form $v[i] = exp$. Notice also that the **while** loop has been translated into the tail-recursive function *f2*, and that the conditional **if** statements are translated into **case** expressions. In the IR program, there is also mutual recursion between the functions *qsort* and *f1*, being *qsort* non-tail recursive and showing double recursion. The four mentioned functions are

[2] `set-array v i e` builds a new array identical to v, except the i-th component which is replaced by the value e. An assignment of the form `v[i]=e` in the source program is translated to `set-array v i e`. This translation may have an impact on the execution time of the (translated to Haskell) IR version with respect to the Java one, but not on the correctness of the algorithm.

defined in a **letfun** expression within the top level visible function *quicksort*. We will use this IR program as a running example of UUT along the paper, since it illustrates some features we are interested in: mutual recursion, non-tail recursion, and dealing with arrays.

We define a *static path* through a set of mutually recursive functions declared together in a UUT, as a potential execution path starting at the top level function, and ending when this function produces a result. Not all the static paths correspond to actual execution paths, since some static paths may be infeasible. The detection of infeasible paths will be done in a subsequent phase by checking the satisfiability of the set of boolean conditions defining the path.

We define the *depth* of a static path, as the maximum unfolding depth of the UUT recursive functions when performing the actions implied by the path. When all the UUT functions are tail recursive, this definition corresponds to the number of times the loop condition is evaluated in imperative loops. When there is at least one non-tail recursive function in the UUT, the depth path is the depth of the call tree deployed during the path execution, considering only the calls to the non-tail recursive function. A depth-1 path is one in which each non-tail recursive function executes one of its base cases, i.e. an invocation to them immediately returns without triggering further recursive invocations; depth-2 ones correspond to executions in which at least one recursive function has executed a recursive case, by generating one or more recursive calls, and then these recursive calls have executed a base case; and so on.

For instance, there is one depth-1 path in function *f2* of Fig. 4, namely the one in which the condition of the first **case** expression is evaluated to *false*, i.e. the condition $i > j$ holds. This corresponds to an execution of the *partition* function in the Java program of Fig. 3 in which the **while** body is not entered. There are also three depth-2 paths in function *f2*, each one ending in a tail recursive call to itself. These recursive calls then execute the depth-1 path.

In the mutually recursive set formed by *qsort* and *f1*, there is one depth-1 path, namely the one in which the **case** condition of *qsort* is evaluated to *false*. There are also four depth-2 paths. In all of them, the **case** condition of *qsort* is evaluated to *true*, and the path continues by calling to *partition*, then to *qsort*, and then to *qsort* a second time. The two latter calls execute the only *qsort* depth-1 path, and the call to *partition* may execute either its depth-1 path, or one of its three depth-2 paths. However, only the *qsort* depth-1 path, and the *qsort* depth-2 path in which *f2* executes any of its depth-2 paths, are feasible. The first one corresponds to an empty array segment as input, and the other three correspond to an array segment having 2 or 3 elements. The infeasible path (the depth-2 path in *qsort* combined with the depth-1 path in *f2*) forces the array segment to have at least two elements in *qsort* (i.e. $a < b$), and to have at most one element in *partition* (i.e. $i = a + 1,\ j = b,\ i > j$), which is contradictory.

Given a UUT written in the IR language, and a maximum depth fixed by the user, our tool generates all the static paths having a depth smaller than or equal to this maximum depth.

3 Two-Level Representation and Assumed Properties

Generating paths in recursive programs cannot be regarded as just a graph problem, as it is the case in iterative programs. In the latter, the control flow graph (CFG) can be depicted as a directed planar graph. Loops become the strongly connected components (SCC) of those graphs. Computing paths is then a combination of computing the graph SCCs, then collapsing them into single nodes, and then computing paths in the resulting DAG (directed acyclic graph), which is an easy problem. The path depth corresponds to the number of iterations the path undergoes in each SCC.

Recursive programs cannot be depicted as planar graphs, unless recursive calls are represented as non-expanded nodes in those graphs. But, as soon as recursive calls are unfolded and replaced by their bodies, the graph representation is not possible anymore. A different number of unfoldings would result in different graphs. When iterative programs are translated into recursive ones, all the resulting functions are tail recursive, as we have seen in the *partition* example above. But, as long as there are non-tail recursive functions in the source program, analyzing the CFG in order to extract the paths is not enough. One may wonder why not to translate non-tail recursive programs into tail-recursive ones by using the well-known continuation passing style translation of functional programs (see, for example [4]). By doing that, the non-tail recursive functions are somehow ‘hidden’ in the continuation and will be unfolded when the continuation is applied. So, it seems rather difficult to extract the paths from a static code which may undergo a variable number of unfoldings. Also, the resulting program is higher-order, which makes the approach more difficult.

We follow here a different strategy: to propose a graph representation of recursive programs at two different levels:

– One level consists of the CFG of each function body. These are DAGs in which the internal calls are represented as non-expanded nodes.
– The other level is the call-graph (CG) of the whole function collection. An edge (f, g) here represents one or more calls from function f to function g. An SCC in this graph represents a loop of functions calling each other in a mutually recursive way.

These two kinds of representations are not arbitrary directed graphs. By knowing that they come from code written in a programming language, we can assume them to satisfy the following properties:

1. Each SCC in the CG has at least one entry node. This is because we assume each function in the CG to be reachable from the top one. Otherwise, it would represent dead code. When the UUT is the result of transforming an iterative program, each SCC has a unique entry node. This holds because iterative loops in conventional programming languages have a unique entry point.

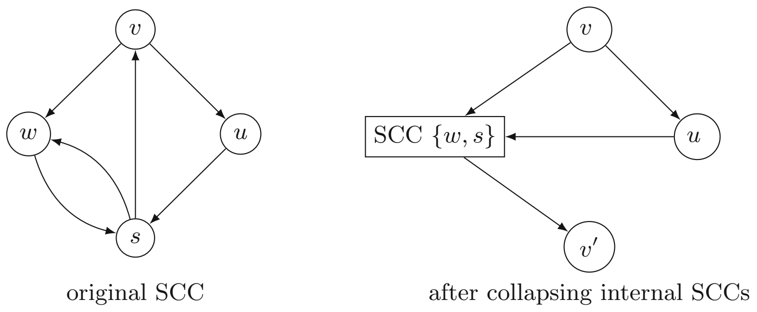


Fig. 5. Illustration of Proposition 1.

2. On the other hand, a SCC in the CG may have more than one exit to nodes external to it. This is because a loop or a function may abruptly terminate by sentences such as **break**, **continue**, **return**, or an exception, which interrupt the normal flow. Also, a function belonging to an SCC may issue a call to a function external to it.
3. In a SCC, all paths starting at an entry node, not exiting the SCC, and not infinitely iterating inside a nested SCC, must eventually reach again that entry node. We justify this statement by Propositon 1 below.
4. Each function CFG has a single source node and a single sink node. Each internal node is reachable from the source and the sink is reachable from each internal node. This is because we assume no dead code, (statically) terminating loops, and also that recursive functions have at least a base case.

A SCC may contain nested SCCs. Our path generation algorithm will not infinitely loop inside any SCC. In order to control the number of iterations inside a SCC, we will count the number of visits the path pays to a single node per SCC. This will be one of its entry nodes. So, it is important to prove that, after visiting all the internal SCCs, a path will reach again the entry node of the external SCC.

Proposition 1. *Given a SCC $G = (V, E)$, subgraph of a directed graph, and an entry vertex $v \in V$, all paths starting at v, not exiting V, and not infinitely looping inside any nested SCC, must go again through v.*

Proof. By definition of SCC, V is a maximal set of vertices such that $\forall v, w \in V$ there exist paths $v \rightsquigarrow w$ and $w \rightsquigarrow v$ not leaving V. G may contain proper subgraphs which are themselves SCCs. Let us assume that we remove v from V and all its incoming and outgoing edges. In the resulting graph, we compute its internal SCCs, and collapse them to single vertices. Let $G' = (V', E')$ be the resulting DAG. Then, we build the following graph $G'' = (V'', E'')$, as follows:

– $V'' = V \cup \{v'\}$, where v' is a fresh vertex aiming at being a duplicate of v.

– E'' is built from E', by adding all the outgoing edges $(v, u) \in E$ from v to vertices $u \in V'$ but, for each incoming edge $(u, v) \in E$, we add instead an edge (u, v') to the duplicate v' of v.

It is clear the G'' is a DAG. Moreover, all paths in G'' starting at v can be extended to paths ending up in v'. This is because G is a SCC and so there exist paths from every vertex of V' to v. The only paths not reaching v' are those exiting G'' or iterating infinitely often in one internal SCC of G. In Fig 5, we illustrate the above construction with an example. □

4 Path Generation Algorithm

As said above, a UUT consists of a top function—we will call it *top*—and a set of internal functions defined in a **letfun** expression within it. Function *top* may call any of them, but not the other way around. The path generation algorithm consists of the following phases:

1. Generating the CFG of each function.
2. Generating the *template paths* (TP) of each function. These are all the paths from its source to its sink.
3. Generating the UUT CG.
4. Computing the CG SCCs.
5. Computing the paths by expanding the TP.

The essential idea of phase 5 is to continuously replace nodes representing calls to functions by all the paths across the bodies of those functions. In order to guarantee termination, an additional control is needed on the unfolding level of some particular functions.

4.1 Generating the CFGs and the TP

We assume a fresh name supplier so that every node of any graph is given an identifying key which is unique in the whole set of graphs. The CFG of each function is a DAG having five types of nodes:

Source. This is the unique entry point of the function. It may have some code associated to it.

Block. This is either a basic block node—having associated sequential code consisting of a sequence of **let** bindings not containing calls, and ending in an edge to another node—, or it is a conditional block node having as associated code a **case** expression. In this case, the node has outgoing edges to more than one node.

Call g**.** Node exactly containing a call to a function, where g is the key of the called function source node.

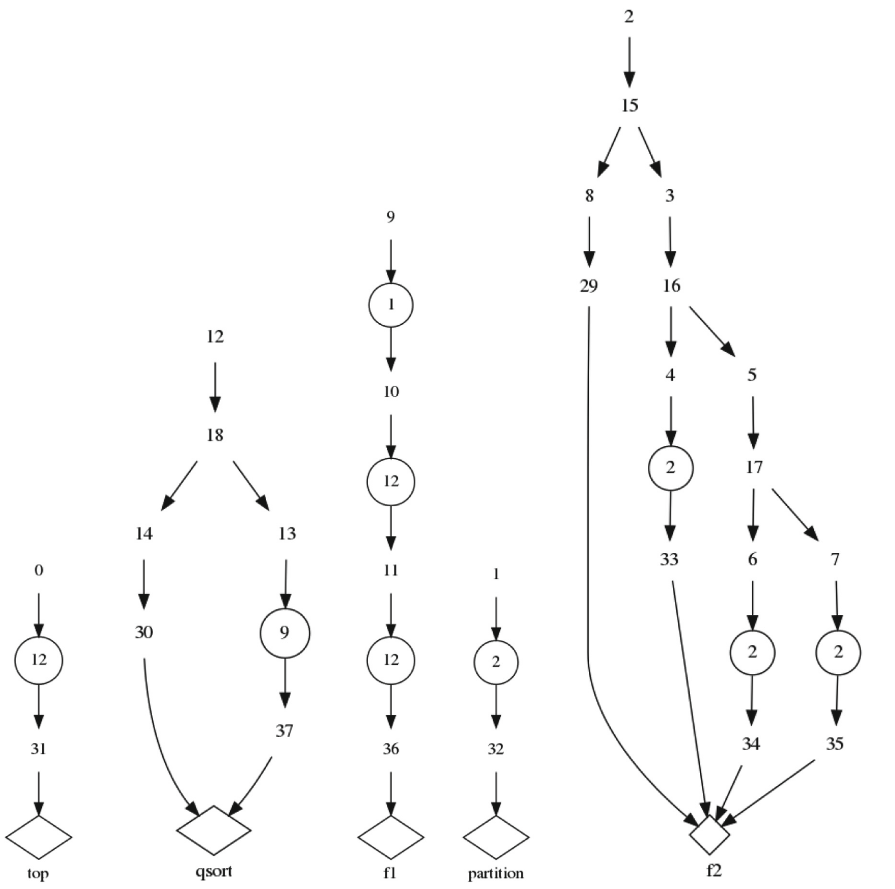


Fig. 6. Control flow graphs for functions top, qsort, f1, partition, and f2.

Return. In our tool, there is one such a node at the end of each execution path. Their next node is always the sink. We associate some code to these nodes related to returning to the caller the result of the function. For the purpose of this paper, it will be dealt with as a block node.

Sink. Node with no associated code representing the unique sink of the function.

From the UUT IR, the algorithm computes the CFG of each function by using conventional techniques such as those one can find in compilers. The resulting graphs are DAGs. In Fig. 6 we show the CFGs computed for all the functions of the *quicksort* UUT. Uncircled nodes are source, block or return nodes; circled nodes are calls; and the diamond node is the sink. For convenience, we only show the keys associated to the nodes.

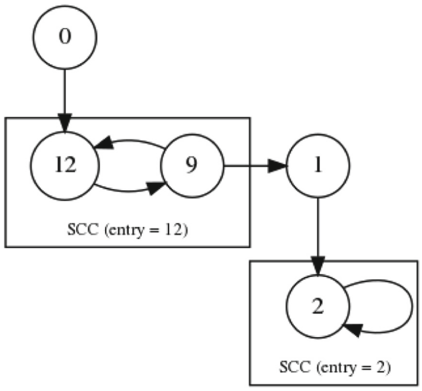


Fig. 7. Call graph and SCCs for function *quicksort*.

Then, a simple recursive algorithm computes all the paths from the source to the sink in each DAG. By convention, the sink node is not included in any path. These are called *template paths* (TP). The algorithm will return a list of TP associated to each function. We call them *template* because they may contain call nodes which should be later expanded in order to compute the final paths. A final path does contain neither call nodes, nor sink ones. It just consists of a sequence of source, block and return nodes.

4.2 Generating the CG and Computing the SCCs

From the TP, the next phase of the algorithm generates the CG. This one consists only of call nodes. An edge (f, g) represents one or more calls from function f to function g. We include in the graph a node with an initial call to the *top* function. In Fig. 7 we show the CG computed by this algorithm for our running example.

Now, all the SCCs of the CG are computed, included the nested ones. This phase uses standard algorithms. Thanks to the restrictions assumed in Sect. 3, each SCC has at least an entry point. Moreover, by Proposition 1, all the paths not exiting the SCC must pass again through that entry point. So, these entry points provide us a convenient way for controlling the depth of the paths: each time we unfold an entry point corresponding to a call to a function f, the subsequent calls to f in the unfolded path represent a depth one unit smaller than the depth of the unfolded f.

Consequently, our algorithm computes an entry point for each SCC and returns the set of them. In Fig. 7 we show the SCCs computed for *quicksort* and the entry points computed.

```
expand ds tp []                    = [[]]
expand ds tp ((Source,k): pth)     = map ((Source,k):)  $ expand ds tp pth
expand ds tp ((Block,k): pth)      = map ((Block,k):)   $ expand ds tp pth
expand ds tp ((RetNode,k): pth)    = map ((RetNode,k):) $ expand ds tp pth
expand ds tp ((CallNode k,_):pth) = concatMap (expand ds tp)
                                              [p1 ++ pth | p1 <- kPaths]
   where kPaths = map (map $ annotate ds) (tp ! k)
expand ds tp ((EntryCall k d,_):pth)
   | d == 0    = []                          -- remove this path
   | otherwise = [p1 ++ p2 | p1 <- concatMap (expand ds' tp) kPaths,
                             p2 <- expand ds tp pth]
   where kPaths = map (map $ annotate ds') (tp ! k)
         ds'    = insert k (d - 1) ds        -- decrease k's recursion depth

annotate :: Map Key Int -> (Node,Key) -> (Node,Key)
annotate ds (CallNode f,i)
         | member f ds = (EntryCall f (ds ! f),i)
annotate ds other      = other
```

Fig. 8. The *expand* algorithm.

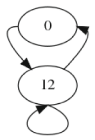
4.3 Computing the Paths

The algorithm starts by creating a map from the SCC's entry points to their initial depth. This is the maximum depth established by the user for the intended white-box paths. Then, the recursive algorithm *expand* is invoked. It is written in Haskell, and its signature is the following:

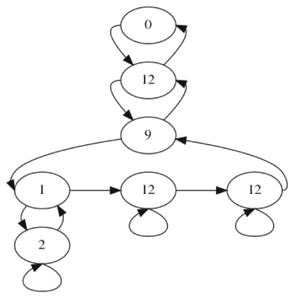
```
expand :: Map Key Int -> TemplatePaths -> [(Node,Key)] -> [[(Node,Key)]]
```

It receives as arguments the depth map, the template paths, and a single path to be expanded. It traverses the path from left to right and *expands* it by systematically replacing call nodes by their template paths. So, in general, the expansion of a path produces a list of paths as a result. Initially, *expand* is invoked with a path consisting of a single call node *Call top*. It processes a path node at a time and invokes itself recursively on the rest of the path.

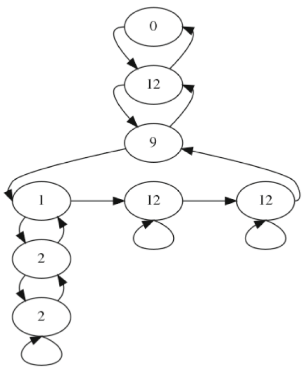
Its complete code is shown in Fig. 8. Nodes of type *Source*, *Block*, and *Return* are simply bypassed, and the expansion continues on the remaining path. Call nodes are replaced by their template paths but, prior to that, the possible calls to the SCC's entry points are annotated with their current remaining depth registered in the depth map. When a node corresponding to an entry node calling to function *f* is expanded, the expansion of its template paths is done with a depth for *f* one unit smaller than the current depth for that entry. However, the expansion of the remaining path is done with the original path depth, as they may occur new calls to *f* in that path, and these should be expanded by starting at the initial path depth. If an entry node is found with its current depth being zero, then the complete path is removed.



path 1 = [0,12,18,14,30,31]



path 2 = [0,12,18,13,9,1,2,15,8,29,32,10,12,18,
14,30,11,12,18,14,30,36,37,31]



path 3 = [0,12,18,13,9,1,2,15,3,16,4,2,15,8,29,33,32,
10,12,18,14,30,11,12,18,14,30,36,37,31]

path 4 = [0,12,18,13,9,1,2,15,3,16,5,17,6,2,15,8,29,34,
32,10,12,18,14,30,11,12,18,14,30,36,37,31]

path 5 = [0,12,18,13,9,1,2,15,3,16,5,17,7,2,15,8,29,35,
32,10,12,18,14,30,11,12,18,14,30,36,37,31]

Fig. 9. The 5 paths of *quicksort* up to depth 2.

In Fig. 9 we show the 5 paths generated by the algorithm for the *quicksort* UUT when the initial depth is set to 2. We have depicted them graphically on the left part of the figure as the call tree deployed by each path. Notice that, by setting the depth to 2, also the paths of depth 1 are generated (in this example, there is only one such path). Moreover, all the combinatorics arising when varying the depths of the different SCCs from 1 to the maximum depth, and building the cartesian product of the corresponding paths, are also achieved by our expansion algorithm. For that reason, setting the maximum depth to 3 in this example results in a combinatorial explosion and produces 2 549 paths (see Fig. 10).

5 Experiments

We have applied the above algorithm to a collection of UUTs including purely functional algorithms such as the insertion of a key in an AVL tree, or computing the union of two leftist heaps; purely iterative ones, such as inserting and searching a value in a sorted array, or the Dutch National Flag problem [3]; and to hybrid ones, such as our *quicksort* running example, which includes an iterative version of *partition*.

In Fig. 10 we show the results of these experiments. The number of static white box paths computed by our algorithm at several depths is shown in the first three columns. The column T/NT specifies whether all functions in the UUT are tail-recursive (T), or there exists some non-tail recursive (NT) ones. The column S/D tells us whether all functions are simple recursive or at least one is double recursive. The last column shows the number of functions included in the UUT, excluding the top visible one. We make note that the number of paths grows very quickly for units having more than one function.

UUT	**depth=1**	**depth=2**	**depth=3**	**T/NT**	**S/D**	**#f**
quicksort	1	5	2 549	NT	D	4
binSearch	1	3	7	T	S	1
DutchNationalFlag	1	4	13	T	S	1
linearSearchArray	2	4	6	T	S	1
insertArray	2	4	6	T	S	1
insertList	2	4	6	NT	S	1
deleteList	3	6	9	NT	S	1
searchBST	2	6	14	T	S	1
insertBST	2	6	14	NT	S	1
searchAVL	2	6	14	T	S	1
insertAVL	2	10 306	(*)	NT	S	6
searchRedBlack	2	6	14	T	S	1
unionLeftist	2	34	20 202	NT	D	3
insertLeftist	2	34	20 202	NT	D	4

(*) It runs out of memory after 40 minutes running.

Fig. 10. White-box paths in a suit of UUTs

We have confirmed in the experiments that the notion of path depth defined here coincides both with the number of iterations in loops incremented by 1, and with the unfolding depth of the call tree in recursive functions. A single algorithm suffices, then, to generate exhaustive white-box paths in iterative, recursive, and hybrid programs.

6 Conclusions

We have not found papers in the literature on the specific subject of white-box path generation in recursive programs. In a comprehensive study conducted in 2014 [8] on white-box and black-box test-case generation, and surveying 85 papers published in journals between 2004 and 2013, there is no mention to this problem. The papers on white-box paths are mainly devoted to how to generate concrete values satisfying a particular path, or to compare the path coverage of a suit of test-cases generated by using a variety of techniques. But, consistently, all of them assume that paths are directly obtained from the program control flow graph. That is, they assume an iterative unit under test.

We believe that the approach presented here is simple enough to be implemented in a practical tool and it can deal with all kinds of recursive programs: simple and multiple recursive, tail and non-tail recursive, and directly and mutually recursive. It is appropriate for unit white-box testing, and could be easily adapted to bottom-up integration testing. For that purpose, the UUT must distinguish between the externally called functions and the ones belonging to the UUT (for instance, by declaring the former in an import clause). Then, the external functions would simply not be expanded by the algorithm.

References

1. Anand, S., et al.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013). https://doi.org/10.1016/j.jss.2013.02.061
2. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
3. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
4. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the Conference on Programming Language Design and Implementation (PLDI 1993), pp. 237–247. ACM (1993). https://doi.org/10.1145/155090.155113
5. Montenegro, M., Nieva, S., Peña, R., Segura, C.: Liquid types for array invariant synthesis. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 289–306. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_20
6. Montenegro, M., Peña, R., Sánchez-Hernández, J.: A generic intermediate representation for verification condition generation. In: Falaschi, M. (ed.) LOPSTR 2015. LNCS, vol. 9527, pp. 227–243. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27436-2_14
7. de Moura, Leonardo, Bjørner, Nikolaj: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, Jakob (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Mustafa, S., Deris, S., Mohamad, R.: Systematic Mapping Study in Automatic Test Case Generation, pp. 703–720. IOS Press, Amsterdam (2014)