White-Box Path Generation in Recursive Programs

Ricardo Peña and Jaime Sánchez-Hernández

Complutense University of Madrid, Computer Science School, 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 iterative programs, the defined depth corresponds to the maximum number of iterations through any loop. When applied to non-tail recursive functions, the meaning is closely related to the maximum number of their unfoldings along the path. It can also be applied to hybrid programs where iteration and recursion are both 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 based on the program specification, and in the second one, they are 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 suite 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

[**] Research paper

limit is usually referred to as the path *depth*. For instance, depth-0 paths will be those that never execute the loop bodies; depth-1 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.

Functional programs do no 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 imply a complete execution of the recursive UUT and it would involve a sequence of decisions taken by the UUT along the execution. In a path, the UUT 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. 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 an UUT to be a collection of mutually recursive functions called from a visible top one. 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, 4]. The platform supports a functional language as its Intermediate Representation (IR) to which both imperative programs written in Java and functional programs written in Haskell are translated. Imperative loops are translated into a set of mutually recursive and 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 an UUT, the platform automatically generates white-box paths, and synthesises a test-case for each path, by using an SMT solver which checks the satisfiability of the conditions involved in the path and assigns appropriate values to the involved variables.

2 Functional Intermediate Representation

Our implementation language is a sort of core functional language which supports mutually recursive function definitions. In Fig. 1 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 [3], 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. 2 we partially show the IR code for function *insertAVL*, inserting a key into an AVL tree. In the **letfun** main expression, the code for functions *height*, *compose*, *leftBalance* and *rightBalance* is not shown. The first one computes the

[1] Static Single Assignment.

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

Fig. 1. CAVI-ART IR abstract syntax

height of an AVL with a time cost in $O(1)$, by just getting the value stored in its root node. The second one just joins two trees and a value received as arguments to form an AVL tree. The other two are responsible for performing the LL, LR, RL and RR rotations that reestablish the height invariant of AVLs. In what follows, we will use the *insertAVL* function, together with all its auxiliary functions defined in the **letfun** expression, as a running example of UUT.

We define a *static path* through a set of mutually recursive functions defined together in an 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 unfeasible.

We define the *depth* of a static path, as the maximum number of unfoldings that any recursive function may undergo in the path. When all the UUT functions are tail recursive, this definition of depth corresponds to the number of iterations in imperative loops. Depth-0 paths correspond to none iteration in all loops, depth-1 ones correspond to at most one iteration, and so on. When there is at least one non-tail recursive function in the UUT, the depth of the path is the depth of the call tree deployed during the path execution, considering only the calls to the non-tail recursive function. Depth 0 means that each non-tail recursive function executes one of its base cases, depth 1 corresponds to that at least one recursive function has executed a recursive case by generating one or more recursive calls, and then these recursive calls have executed its base case, and so on.

In the *insertAVL* example there are two depth-0 paths. In the first one, the input tree is empty and the internal function *ins* immediately terminates by creating a new node with the key x and with height 1. In the second one, the key x being inserted is already present at the root of the input tree, and the *ins*

```
define insertAVL (x::Int, t::AVL Int)::(res::AVL Int) =
  letfun
    ins (x::Int, t::AVL Int)::(res::AVL Int) =
      case t of
        leafA                                     -> nodeA x 1 leafA leafA
        nodeA y::Int h::Int l::(AVL Int) r::(AVL Int) ->
                         let b1::Bool = x < y in
                         case b1 of
                           true  -> let ia::(AVL Int) = ins x l in
                                    equil ia y r
                           false -> let b2::Bool = x > y in
                                    case b2 of
                                      true  -> let ib::(AVL Int) = ins x r in
                                               equil l y ib
                                      false -> t

    equil (l::AVL Int, x::Int, r::AVL Int)::(res::AVL Int) =
      let hl::Int  = height l in
      let hr::Int  = height r in
      let hr2::Int = hr + 2 in
      let b::Bool  = hl == hr2 in
        case b of
          true  -> leftBalance l x r
          false -> let hl2::Int = hl + 2 in
                   let b2::Bool = hr == hl2 in
                   case b2 of
                     true  -> rightBalance l x r
                     false -> compose l x r

    ...  definitions of height, compose, leftBalance and rightBalance
  in ins x t
```

Fig. 2. CAVI-ART IR for function *insertAVL*

function terminates without inserting anything. With depth 1, there are at least 4 paths through the function *ins*: two of them recursively call to *ins* with the left subtree as an argument, and another two call to *ins* with the right one as an argument. The rest of the path inside these recursive calls is one of the depth-0 paths. After inserting the key, the static paths go on with a call to function *equil*, and there are two paths in this function. Combined with the other 4, this gives 8 paths. Then, we should consider paths through *leftBalance* or *rightBalance*, depending on the branch taken by *equil*, and so on. The combinatorics soon produces an exponential number of paths when the UUT is as complex as in our example. Notice however, that many of these paths are unfeasible. For instance, when inserting a new node in the left subtree, it may not happen that *equil* takes the branch calling to *rightBalance*: if any unbalance arises after inserting to the left, it should be in the left subtree.

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

3 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.

We, then, propose a graph representation of recursive programs at two different levels:

- One level consists of the CFG of each function body. This is a DAG 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 SCCs in this graph symbolises 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 (PL), we assume them to satisfy the following properties:

1. Each SCC in the CG has a unique entry node. This property holds because iterative loops and recursive functions in conventional PLs have a unique entry point.
2. However, a SCC 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.
3. 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 and (statically) terminating loops in our code, and also that recursive functions have at least a base case.

4 Path Generation Algorithm

As said above, an 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. Generation of the CFG of each function.
2. Generation of the *template paths* (TP) of each function. These are all the paths from its source to its sink.
3. Generation of the CG.
4. Splitting the CG into a set of disjoint subgraphs.
5. Computing the SCCs of each subgraph.
6. Computing the paths by expanding the TP and the SCCs.

4.1 Generating the CFGs and the TPs

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 four types of nodes:

Source This is the entry point of the function. It has no code associated to it.
Block This is either a basic block node —having associated sequential code consisting of a sequence of **let** bindings, and ending in an edge to a call node or to a conditional block—, 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 entry node.
Sink Node with no associated code representing the 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. In Fig. 3 we show the CFG computed for function *ins* of *insertAVL*. Circle nodes labeled with a **B** are blocks, the reminder circle nodes are calls, the square node is the source and the diamond one is the sink.

Then, a simple recursive algorithm computes all the paths from the source to the sink in each DAG. These are the template paths (TP). There is a list of TPs 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 source nodes, nor sink ones. It just consists of a sequence of blocks.

4.2 Generating the CG

From the TP, the algorithm generates the CG. This one consists only of call nodes, plus an additional sink node. The graph edges are marked as *tail* or *non-tail*. This is important for the following phases. Let us assume that the algorithm is processing the TPs of a function f:

– If it finds a node *Call* g not followed by a sink node, it marks function g as non-tail, and all the edges from any other function to it are marked accordingly.

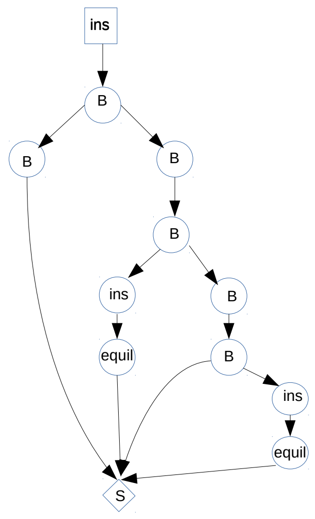

Fig. 3. Control Flow graph for function *ins* of *insertAVL*.

- If all the call nodes to g are in tail positions (i.e. followed by the sink node), then the edges to it are marked as tail ones.
- If there exists a path containing no tail-call nodes, then an edge from f to the sink is included in the graph. This path represents the execution of a base case function f.

In Fig. 4 we show the CG computed by this algorithm for function *insertAVL*. The non-tail edges are labeled NT, being tail edges the remainder ones.

4.3 Splitting the CG

The CG is not directly appropriate for generating the final paths from it. There, a set of mutually recursive functions belonging to an SCC should be dealt with as a single unit in order to control the depth of the paths. Also, if a function is non-tail recursive, their paths must be independently generated, and then embedded into the bigger paths of the functions calling it. For those reasons, it is convenient to split the CG into disjoints subgraphs, each one representing a piece of code whose paths must be generated independently of those of other pieces. Each subgraph will have a unique entry function. There exists a hierarchy between these subgraphs: if there is a call from a function of subgraph G_1 to the entry function of subgraph G_2, then it may not exist direct or indirect calls from the functions of G_2 to the entry function of G_1.

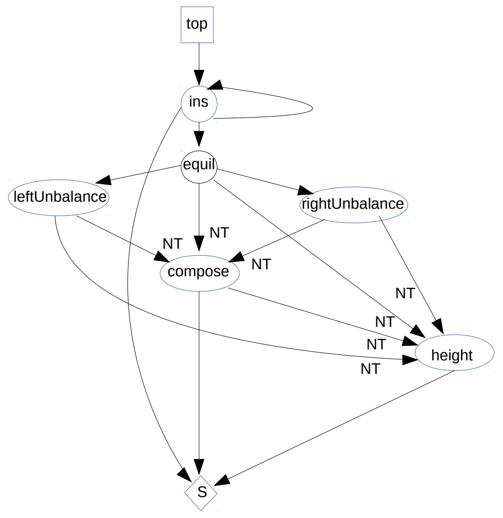

Fig. 4. Call graph for function *insertAVL*.

The splitting algorithm does a depth-first traversal of the CG starting at the *top* node. Each time a non-tail edge (f, g) is reached, it checks whether node f is reachable from g. If so, the edge is kept; otherwise, it is removed. Tail edges are kept. Additionally, if a function f is non-tail recursive, all the calls not belonging to the SCC component involving f are considered non-tail calls and the edges to them are removed.

By applying this algorithm to the CG of Fig. 4, we get the split graph of Fig 5.

4.4 Computing the SCCs

Now, the SCCs of all the subgraphs of the split CG are computed, and each one is collapsed into a single node of type *SCCnode*. In this way, each subgraph becomes a DAG. For the SCC nodes, all their internal paths from depth-0 to the maximum depth are computed. At this stage, we consider the depth of an SCC path to be the maximum number of times minus one that the path hits the entry function of the SCC. So, if a path hits the entry function only once, and then exits the SCC, it would be a depth-0 path; If it hits the entry function twice, it would be a depth-1 path; and so on.

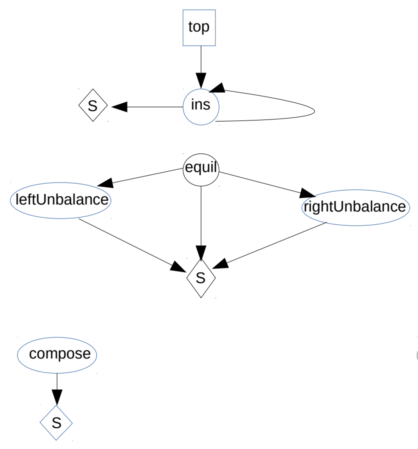

Fig. 5. Split call graph for function *insertAVL*.

4.5 Computing the paths

The algorithm starts by generating all the paths of the subgraph containing the *top* function, from the entry node to the sink. Each of these paths represents a complete execution of function *top*. The rest of the algorithm just *expands* each of these paths. The expansion consists of traversing the path and, in turn, expanding each node. The expansion of a node depends on its type:

– If it is a node *Call* g, not being g the top function of a subgraph, it means that g is part of the current subgraph. The algorithm, then, replaces the node by all the template paths of g *matching* the rest of the path. The notion of matching requires traversing the subsequent path nodes looking for the first node of the form *Call* h, of type SCCnode, or a sink.
  - If the node found is a sink, then the TPs of g selected for expansion are those not ending in a tail-call.
  - If the node found has the form *Call* h, then the TPs of g selected for expansion are those containing one or more calls to h.
  - If the node found is an SCC node with entry function h, then the TPs of g selected for expansion are those containing one or more calls to h. Additionally, the paths computed for the SCC are appended to these paths.

– If it is a node *Call* g, being g the top function of a subgraph G, it means that g is an independent function. Then, all the paths from G's entry function to its sink are computed, and the node *Call* g is replaced by them.

- If it is a node of type *Block*, it does not need further expansion and it is skipped.
- If it is a source or a sink, it is removed from the path.

5 Conclusions

We have applied the above algorithm to a collection of UUTs including purely functional algorithms such as *insertAVL*, 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 [2]; and to hybrid ones, such as the recursive quicksort algorithm including an iterative version of *partition*. We have confirmed in the examples that the notion of path depth defined here coincides both with the number of iterations in loops and with the depth of the call tree in non-tail recursive functions. A single algorithm suffices, then, to generate exhaustive white-box paths in iterative, recursive, and hybrid programs.

References

1. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86(8), 1978–2001 (2013), https://doi.org/10.1016/j.jss.2013.02.061
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
3. 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'93). pp. 237–247. ACM (1993), http://doi.acm.org/10.1145/155090.155113
4. Montenegro, M., Nieva, S., Peña, R., Segura, C.: Liquid types for array invariant synthesis. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2017. pp. 289–306. Springer, LNCS 10482 (2017)
5. Montenegro, M., Peña, R., Sánchez-Hernández, J.: A generic intermediate representation for verification condition generation. In: Falaschi, M. (ed.) Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9527, pp. 227–243. Springer (2015), http://dx.doi.org/10.1007/978-3-319-27436-2