This article explores relationships between imperative and functional programming by viewing a program as a set of paths. We argue, through a small case study, that the presence of infeasible (or unexecutable) paths makes programs harder to understand. We identify two main causes of infeasible paths, “unnecessary” sequencing and destructive update, hallmarks of an imperative programming style. Functional programming eschews sequencing and destructive update, which can result in programs with fewer infeasible paths that are easier to understand. No proofs are included. We intend to provoke discussion regarding imperative and functional programming styles.

Feasible and Infeasible Paths

Regardless of the language they spring from, programs execute paths, or sequences, of instructions. Imperative programs execute paths consisting of statements. Functional programs execute paths consisting of expressions. Both imperative and functional languages have nested control structures that allow the conditional execution of program elements. The C language [HS94] has explicit statement sequencing (via the “;” and the block construct), while functional languages such as Standard ML [Pau96, MTHM97], use let-clauses and recursion in preference to explicit sequencing (although Standard ML and most functional languages contain sequencing operators).

While programs can contain huge numbers of paths, many of these paths can never execute. A path through a program is said to be feasible if there is some program execution that traverses that path. All other paths in a program are said to be infeasible; these paths cannot execute in any possible program execution. Infeasible paths generally make programs harder to understand, as programmers must sort out which paths are truly executable and which are not.

Infeasible paths arise from correlated branches: branches whose behavior is dependent on one another. Consider the C procedure foo1 in Figure 1(a), and its control-flow graph, shown in Figure 2. One path is infeasible, since if the variable x is less than 10 then x is also less than 20. In Figure 2, the infeasible path is highlighted in the control-flow graph with bold edges. The number in brackets to the right of a line of source code indicates the number of paths from that line to procedure exit. The number of paths through this procedure is four, as there are two if-then statements in sequence, resulting in $2^2=4$ possible paths. Three of the four paths are feasible.

Hedley and Hennell [HH85] discuss some causes of infeasible paths. They refer to “functional programming” as a means for reducing the number of infeasible paths:

> The functional programming method involves partitioning the input domain of a program into a number of distinct subsets associated with program logical structures and with functional requirements. Functional programming produces programs having simpler structure, with fewer paths and fewer executable statements.

How does this idea apply to our toy example? By evaluating the final value for variable y along each of the three feasible paths, we can simplify and rewrite the procedure foo1 as shown in Figure 1(b). There are three paths through the procedure foo2, each of them feasible. Our restructuring of this simple function has both reduced its complexity (by eliminating the infeasible path), and made its effect more obvious. We have eliminated the intermediate variable y and expressed the return value for each
int foo1(int x) {
    int y = x;
    if (x < 10) y++; [4]
    if (x < 20) y++; [2]
    return y; [1]
}

(a)

int foo2(int x) {
    if (x < 10) [3]
    return x+2;
    else if (x < 20) [2]
    return x+1;
    else [1]
    return x;
}

(b)

Figure 1: A simple function (a) and a restructured version of the function (b).

Figure 2: Control-flow graph of procedure foo1 in Figure 1(a). There are four paths through the graph. The path with bold edges is infeasible.

A More Complex Example

Let’s explore infeasible paths and their relationship to imperative and functional programming styles using a more complex example, shown in Figure 3. This procedure, DeleteNode, is taken directly from an ordered binary tree implementation of Cormen et al. [CLR90], a well-known book on data structures and algorithms. Each node of the tree is represented by a structure Node with four fields: an integer key; a left child pointer, right child pointer and parent pointer. A pointer to a unique Node named nilNode is used instead of the NULL pointer value, allowing many NULL pointer tests to be eliminated.

The tree obeys the following key-ordering invariant: the key value of every node is greater than the key values of all its left descendants and less than or equal to the key values of all its right descendants.

The purpose of the DeleteNode procedure is to delete the key value of the node pointed to by the parameter z, while maintaining the key ordering invariant. If z has no children or exactly one child, z may be deleted easily. However, if z has two children then the procedure determines the successor of z in key order, which is the lowest leftmost node in z’s right subtree. This is accomplished by calling the following function with z->right as argument:

    Node* leftmostNode(Node *n) {
        while (n->left != nilNode)
            n = n->left;
        return n;
    }

Node y, the result of calling this function, will be deleted instead of z, its key value replacing z’s. Child and parent pointers need to be adjusted properly. The procedure returns a pointer to the deleted node.

As before, the number in brackets to the right of a line indicates the number of paths from that line to procedure exit. There are 36 intraprocedural paths through the DeleteNode procedure, but only 8 of these paths are feasible. The rather low ratio of 8/36 is due to the following correlations in the code:

- After procedure leftmostNode is called by DeleteNode, it is guaranteed that y and z will point to different nodes, the left child of y will point to nilNode (otherwise y would not be the leftmost
Node* DeleteNode(Node* z) {
    Node *x, *y;
    if (z->left == nilNode) [9]
        return reparent(z,z->right);
    else if (z->right == nilNode) [6]
        return reparent(z,z->left);
    else {
        y = leftmostNode(z->right); [3]
        y->key = y->key;
        return reparent(y,y->right); [3]
    }
    return y;
}

Node* reparent(Node* y, Node* x) {
    x->parent = y->parent; [3]
    if (y->parent == nilNode) [3]
        root = x;
    else if (y == y->parent->left) [2]
        y->parent->left = x;
    else
        y->parent->right = x;
    return y;
}

Figure 4: Restructured DeleteNode procedure

the observation that the main goal of the DeleteNode procedure is to identify a pair of nodes (y, x) such that x is a child of y, and to replace node y in the tree by node x. In the case where z has two nilNode children or exactly one nilNode child, the deletion is carried out by immediately calling the reparent procedure. In the case where z has two children, the procedure leftmostNode is called to find the next node (y) in key order, transfer its key value to z and delete y, again using the reparent procedure.

The restructuring eliminates two predicates from the original code: the redundant predicates at lines 5 and line 14. The code controlled by these predicates is now found directly in the relevant cases of the if-then-else.

One infeasible path remains in the restructured procedure, as the reparent procedure checks to see if y is the root node of the tree, but y cannot be the root node when DeleteNode calls the procedure leftmostNode.

To achieve better efficiency we could create a specialized version of reparent for the third clause of the if-then-else that would not check if node y were the root. Furthermore, since each call to reparent is a tail call ending each of the three paths through the DeleteNode procedure, DeleteNode can jump to reparent rather than calling it, saving the pushing and popping of an activation frame.

1Since the code has been split into two procedures, we count the number of interprocedural, rather than intraprocedural, paths. However, we do not count the paths through the function leftmostNode, to maintain parity with Figure 3.
Infeasible Paths and Imperative and Functional Programming Styles

The restructured DeleteNode procedure is more functional than the original procedure in several respects:

- Much unnecessary sequencing has been eliminated. For example, consider the sequence of the first two if-then-else statements in the original procedure. There are three paths through the first statement (due to the short-circuit evaluation) and two paths through the second (which selects the left/right child of y), yielding a total of six possible paths. However, our correlation analysis showed that there are only three feasible paths through this sequence, as each of the three paths through the first if-then-else uniquely determines the path through the second if-then-else. In the restructured code, the selection of the left/right child has been moved into the calls to the reparent procedure.

- Intermediate variables and explicit assignment statements have been eliminated (variable x) and their use localized (variable y). In fact, within each procedure, each use of a variable is reached by exactly one definition, which was not the case in the original code. Thus, the restructured code is in Static-Single-Assignment form [App98].

- The common code for “reparenting” has been extracted into a procedure, invoked as a tail call by each of the three paths through DeleteNode.

Imperative languages such as C encourage programming “one step at a time”, which can result in procedures such as the original DeleteNode. Each step may be locally obvious and correct, but the resulting paths (and procedures) can be very complex, with many infeasible paths. These infeasible paths make code harder to understand, as programmers must sort out which scenarios are possible and which are not possible. Furthermore, infeasible paths may indicate inefficiencies in the code due to correlated branches.

We conjecture that infeasible paths in imperative programs arise mainly through a combination of unnecessary statement sequencing and destructive updating of state (assignment statements). Sequencing of conditionals leads quickly to an exponential blow-up in the number of paths. For example, a sequence of N if-then statements yields $2^N$ paths, as shown in the control flow graph of Figure 5(a). On the other hand, N nested if-then statements yield $N + 1$ paths, as shown in the control flow graph of Figure 5(b). Assignment statements can create correlated branches that may be widely separated in the code. In the DeleteNode procedure, the assignment statement that aliased y and z and the if-then that tested for the alias is such an example.

Summary

We have argued that infeasible paths lead to code that is harder to understand and that a primary cause of infeasible paths is the unnecessary use of statement sequencing, intermediate variables and assignment statements. By eliminating infeasible paths one can create programs that are more functional in style. Of course, infeasible paths cannot always be eliminated without greatly increasing code size. In the example of the procedure DeleteNode, we were fortunate to be able to eliminate most of the infeasible paths without increasing code size. One can also reduce the number infeasible paths created in the first place by adopting a functional programming style, which eschews the use of explicit sequencing operators and destructive state update.
Advertisement

The DeleteNode example presented here was analyzed with the aid of Dan Hoffman, using the Hot Path Browser (HPB), a GUI for helping programmers understand program behavior in terms of execution paths. HPB imports path profiles and displays them using a web-browser-like interface. HPB is a joint project between the author, Jim Larus of Microsoft Research and Genevieve Rosay of the University of Wisconsin–Madison. For more information about HPB, see: http://www.bell-labs.com/projects/HPB/

Acknowledgements

Thanks to Glenn Bruns, Dan Hoffman, Jim Larus, Genevieve Rosay and Philip Wadler for their comments.

References


Thomas Ball is a member of the technical staff in the Systems and Software Research Center of Lucent Technologies’ Bell Laboratories. His research focuses on improving the software production cycle through the application of domain specific programming languages, the dynamic analysis of programs, and software visualization. He is the program chair of the 1999 Usenix Conference on Domain Specific Languages.