Interprocedural Path Profiling

David Melski
Thomas Reps

Technical Report #1382

September 1998

Interprocedural Path Profiling[1]

David Melski[2] Thomas Reps[2]

September 1, 1998

Abstract

In path profiling, a program is instrumented with code that counts the number of times particular path fragments of the program are executed. This paper extends the *intra*procedural path-profiling technique of Ball and Larus to collect information about *inter*procedural paths (i.e., paths that may cross procedure boundaries).

Interprocedural path profiling is complicated by the need to account for a procedure's calling context. To handle this complication, we generalize the "path-naming" scheme of the Ball-Larus instrumentation algorithm. In the Ball-Larus work, each edge is labeled with a number, and the "name" of a path is the sum of the numbers on the edges of the path. Our instrumentation technique uses an edge-labeling scheme that is in much the same spirit, but to handle the calling-context problem, edges are labeled with *functions* instead of *values*. In effect, the edge-functions allow edges to be numbered differently depending on the calling context. A key step in the process of creating the proper edge functions is related to a method proposed by Sharir and Pnueli for solving context-sensitive interprocedural dataflow-analysis problems.

Some of the machinery that we develop to handle the calling-context problem for purposes of interprocedural path profiling suggests other variants of both intraprocedural and interprocedural path profiling, as well as a variety of hybrid intra-/interprocedural schemes.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*; D.2.2 [**Software Engineering**]: Tools and Techniques—*programmer workbench*; D.2.5 [**Software Engineering**]: Testing and Debugging—*diagnostics*; *tracing*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; *path and circuit problems*

General Terms: Algorithms, Measurement

Additional Key Words: Control-flow graph, instruction tracing, instrumentation, profiling, algebraic path problem, valid path

1 Introduction

In path profiling, a program is instrumented with code that counts the number of times particular path fragments of the program's control-flow graph—or *observable paths*—are executed. A path profile for a given run of a program consists of a count of how often each observable path was executed. This paper extends the *intra*procedural path-profiling technique of Ball and Larus [6] to collect information about *inter*procedural paths (i.e., paths that may cross procedure boundaries).

Interprocedural path profiling is complicated by the need to account for a procedure's calling context. There are really two issues:

- *What is meant by a procedure's "calling context"?* Previous work by Ammons et al. [2] investigated a hybrid intra-/interprocedural scheme that collects separate *intra*procedural profiles for a procedure's different calling contexts. In their work, the "calling context" of procedure P consists of the *sequence*

[1]Supported in part by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, by grants from Rockwell and IBM, and by a Vilas Associate Award from the University of Wisconsin.

[2]Address: Computer Sciences Department; University of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA. E-mail: {melski,reps}@cs.wisc.edu.

of call sites pending on entry to P. In general, the sequence of pending call sites is an abstraction of any of the paths ending at the call on P.

The path-profiling technique presented in this paper profiles true *inter*procedural paths, which may include call and return edges between procedures, paths through pending procedures, and paths through procedures that were called in the past and have completed execution. This means that, in general, our technique maintains finer distinctions than those maintained by the profiling technique of Ammons et al. Furthermore, if one thinks of indexing the paths that pass through procedure P by path-prefix up to a call on P, our technique can be viewed as one in which the "calling context" for a call on P consists of an observable path ending at the call on P. Again, this causes our technique to maintain finer distinctions than those maintained by the technique of Ammons et al.

- *How does the calling-context problem impact the profiling machinery?* In the method presented in this paper, the "naming" of paths is carried out via an edge-labeling scheme that is in much the same spirit as the path-naming scheme of the Ball-Larus technique, where each edge is labeled with a number, and the "name" of a path is the sum of the numbers on the path's edges. However, to handle the calling-context problem, in our method edges are labeled with *functions* instead of *values*. In effect, the use of edge-functions allows edges to be numbered differently depending on the calling context.

  At runtime, as each edge e is traversed, the profiling machinery uses the edge function associated with e to compute a value that is added to the quantity `pathNum`. At the appropriate program points, the profile is updated with the value of `pathNum`.

The principal contribution of the paper is to generalize the Ball-Larus technique to collect interprocedural path profiles. Specific technical contributions of the paper include:

- In the Ball-Larus scheme, a cycle-elimination transformation of the (in general, cyclic) control-flow graph is introduced for the purpose of numbering paths. We present the interprocedural analog of this transformation.

- In the case of intraprocedural path profiling, the Ball-Larus scheme produces a dense numbering of the observable paths within a given procedure: That is, in the transformed (*i.e.*, acyclic) version of the control-flow graph for a procedure P, the sum of the edge labels along each path from P's entry vertex to P's exit vertex falls in the range [0..number of paths in P], and each number in the range [0..number of paths in P] corresponds to exactly one such path.

  The techniques presented in this paper produce a dense numbering of interprocedural observable paths. The significance of the dense-numbering property is that it ensures that the numbers manipulated by the instrumentation code have the minimal number of bits possible.

- We present a technique that allows edges to be labeled differently depending on the calling context. This involves labeling edges with functions instead of values. (A key step in the process of creating the proper edge functions has an interesting connection to a method proposed by Sharir and Pnueli [17] for solving context-sensitive interprocedural dataflow-analysis problems.)

- Some of the machinery that we develop to handle the calling-context problem for purposes of *inter*procedural path profiling suggests other variants of both intraprocedural and interprocedural path profiling, as well as a variety of hybrid intra-/interprocedural schemes.

The profiling techniques presented in this paper have a number of applications:

- There are several possible applications in program optimization, in conjunction with the *partial-inlining* technique of Muth and Debray [12]. This allows inlining of only a selected path or paths of a procedure P at a call site on P; residual procedures are generated to handle the cases when the infrequently executed pieces of P are needed at the expanded call site. Information ascertained from interprocedural path profiling could be used to drive partial inlining, either for interprocedural trace scheduling or for an interprocedural extension of the work of Ammons et al. on path-qualified dataflow analysis [3].

- There are also several applications in software maintenance:

- Reps et al. [14] showed that when different runs of a program produce different path profiles, the differences in the path profiles can be used to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant except one, any such divergence can be attributed to this factor. This can be useful in locating program errors, by examining the profile differences between a run that does not exhibit a program error and a run that does. While any path-profiling technique can be used, the set of paths profiled—the observable paths—affects the fidelity of the technique. The path-profiling techniques described in this paper generate more detailed path profiles than any other published technique.
- During alpha- or beta-testing, path profiling can be used to detect "oddball" paths—paths that were not exercised by a program's test suite. Note that an oddball path is likely to be an infrequently-executed (*i.e.*, cold) path, and thus cannot be found by techniques that are used to estimate hot paths from edge profiles [7].
- Profiling could be used to allow a debugger to report the last path (or last several paths) executed before a breakpoint or program crash. This would provide a nice complement to other information provided by the debugger, such as the current call stack.

Our work encompasses two main algorithms for interprocedural path profiling, which we call *context path profiling* and *piecewise path profiling*, as well as several hybrid algorithms that blend aspects of the two main algorithms. Context path profiling is best suited for the software-maintenance applications listed above, whereas piecewise path profiling is better suited for providing information about interprocedural hot paths, and hence is more appropriate for the optimization applications.

The first two-thirds of the paper concentrates on context path profiling. Up through Section 4, the term "interprocedural path profiling" means "interprocedural context path profiling". We have chosen to discuss the context-path-profiling algorithm first because the method is simpler to present than the algorithm for piecewise path profiling. However, the same basic machinery—which is developed in Section 5—is at the heart of both algorithms.

The remainder of this paper is organized into eight sections: Section 2 presents background material and defines terminology needed to describe our results. Section 3 gives an overview of interprocedural context path profiling. Section 4 describes a generalization of the Ball-Larus path-numbering technique that is used in each of the interprocedural path-profiling techniques. Section 5 describes the technical details of our approach to interprocedural context path profiling. Section 6 describes interprocedural piecewise path profiling. Section 7 summarizes a novel approach to *intra*procedural path profiling that is related to our interprocedural-profiling techniques. Section 8 discusses a variety of hybrid intra-/interprocedural schemes that are possible. Section 9 discusses some issues that arise in path profiling that are not described in the rest of the paper, such as how to generate an interprocedural profile in the presence of function pointers.

2 Background

2.1 Summary of the Ball-Larus Technique for Intraprocedural Path Profiling

The Ball-Larus path-numbering scheme applies to an acyclic control-flow graph with a unique entry vertex *Entry* and a unique exit vertex *Exit*. For purposes of numbering paths, control-flow graphs that contain cycles are modified by a preprocessing step to turn them into acyclic graphs:

> Every cycle must contain one backedge, which can be identified using depth-first search. For each backedge $w \rightarrow v$, add the surrogate edges $Entry \rightarrow v$ and $w \rightarrow Exit$ to the graph. Then remove all of the backedges from the graph.

The resulting graph is acyclic. In terms of the ultimate effect of this transformation on profiling, the result is that we go from having an infinite number of unbounded-length paths in the original control-flow graph to having a finite number of acyclic bounded-length paths in the modified graph. A path p in the original graph that proceeds several times around a loop will, in the profile, contribute "execution counts" to several smaller "observable paths" whose concatenation makes up p. In particular, the paths from *Entry* to *Exit* in the modified graph correspond to observable paths in the original graph (where following the edge $Entry \rightarrow v$

that was added to the modified graph corresponds to beginning a new observable path that starts with the backedge $w \rightarrow v$ of the original graph, and following the edge $w \rightarrow Exit$ that was added to the modified graph corresponds to ending an observable path in the original graph at w). Furthermore, each path from *Entry* to *Exit* in the modified graph defines an observable path in the control-flow graph.

In the discussion below, when we refer to the "control-flow graph", we mean the transformed (*i.e.*, acyclic) version of the graph.

The Ball-Larus numbering scheme labels the control-flow graph with two quantities:

1. Each vertex v in the control-flow graph is labeled with a value, $numPaths[v]$, which indicates the number of paths from v to the control-flow graph's *Exit* vertex.

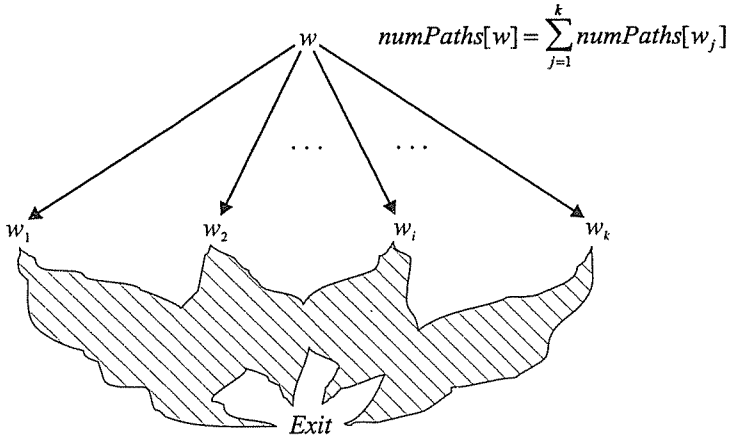
2. Each edge e in the control-flow graph is labeled with a value derived from the $numPaths[]$ quantities.

For expository convenience, we will describe these two aspects of the numbering scheme as if they are generated during two separate passes over the graph. In practice, the two labeling passes can be combined into a single pass.

In the first labeling pass, vertices are considered in reverse topological order. The base case involves the *Exit* vertex: It is labeled with 1, which accounts for the path of length 0 from *Exit* to itself. In general, a vertex w is labeled only after all of its successors $w_1, w_2, \ldots, w_k$ are labeled. When w is considered, $numPaths[w]$ is computed using the following equation:

$$numPaths[w] = \sum_{i=1}^{k} numPaths[w_i]. \tag{1}$$

This equation is illustrated in the following diagram:

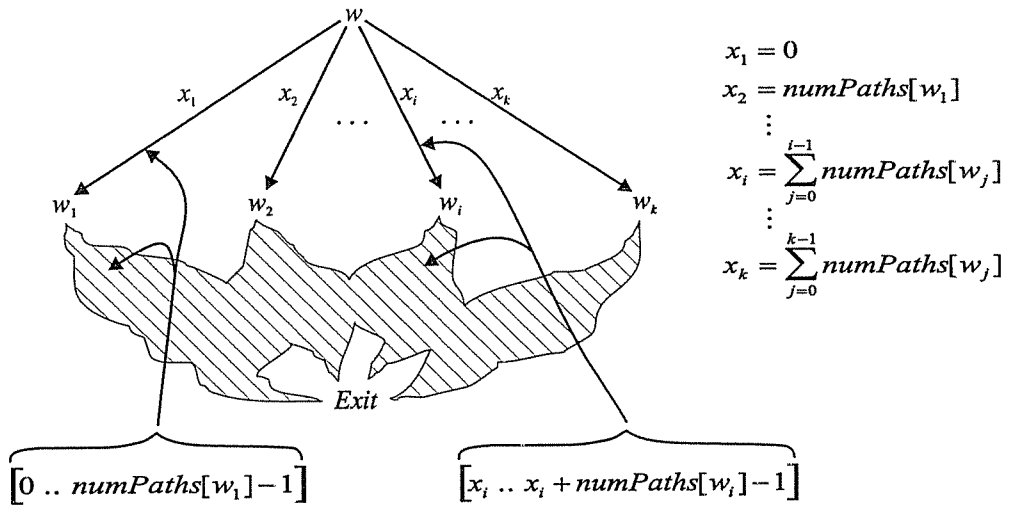


The goal of the second labeling pass is to arrive at a numbering scheme for which, for every path from *Entry* to *Exit*, the sum of the edge labels along the path corresponds to a *unique* number in the range $[0..numPaths[Entry] - 1]$. That is, we want the following properties to hold:

1. Every path from *Entry* to *Exit* is to correspond to a number in the range $[0..numPaths[Entry] - 1]$.

2. Every number in the range $[0..numPaths[Entry] - 1]$ is to correspond to some path from *Entry* to *Exit*.

Again, the graph is considered in reverse topological order. The general situation is shown below:



At this stage, we may assume that all edges along paths from each successor of w, say w_i, to *Exit* have been labeled with values such that the sum of the edge labels along each path corresponds to a unique number in the range $[0..numPaths[w_i]-1]$. Therefore, our goal is to attach a number x_i on edge $w \to w_i$ that, when added to numbers in the range $[0..numPaths[w_i]-1]$, distinguishes the paths of the form $w \to w_i \to \ldots \to Exit$ from all paths from w to *Exit* that begin with a different edge out of w.

This goal can be achieved by generating numbers $x_1, x_2, \ldots, x_k$ in the manner indicated in the above diagram: The number x_i is set to the sum of the number of paths to *Exit* from all successors of w that are to the left of w_i:

$$x_i = \sum_{j<i} numPaths[w_j]. \tag{2}$$

This "reserves" the range $[x_i..x_i + numPaths[w_i] - 1]$ for the paths of the form $w \to w_i \to \ldots \to Exit$. The sum of the edge labels along each path from w to *Exit* that begins with an edge $w \to w_j$, where $j < i$, will be a number strictly less than x_i. The sum of the edge labels along each path from w to *Exit* that begins with an edge $w \to w_m$, where $m > i$, will be a number strictly greater than $x_i + numPaths[w_i] - 1$.

The final step is to instrument the program, which involves introducing a counter variable and appropriate increment statements to accumulate the sum of the edge labels as the program executes along a path.

Several additional techniques are employed to reduce the runtime overheads incurred. These exploit the fact that there is actually a certain amount of flexibility in the placement of the increment statements [5, 6].

2.2 Supergraph

As in many interprocedural program-analysis problems, we work with an interprocedural control-flow graph called a *supergraph*. Specifically, a program's supergraph G^* consists of a unique entry vertex $Entry_{global}$, a unique exit vertex $Exit_{global}$, and a collection of control-flow graphs (one for each procedure), one of which represents the program's main procedure. For each procedure P, the flowgraph for P has a unique entry vertex, $Entry_P$, and a unique exit vertex, $Exit_P$. The other vertices of the flowgraph represent statements and predicates of the program in the usual way,[1] except that each procedure call in the program is represented in G^* by two vertices, a *call* vertex and a *return-site* vertex. In addition to the ordinary intraprocedural edges that connect the vertices of the individual control-flow graphs, for each procedure call (represented, say, by call vertex c and return-site vertex r) to procedure P, G^* contains a *call-edge*, $c \to Entry_P$, and a *return-edge*, $Exit_P \to r$. The supergraph also contains the edges $Entry_{global} \to Entry_{main}$ and $Exit_{main} \to Exit_{global}$. An example of a supergraph is shown in Figure 1.

Each execution of a program corresponds to an *execution path* in the program's supergraph, G^*, from $Entry_{global}$ to $Exit_{global}$. It is possible that some paths in G^* do not correspond to feasible execution paths. In particular, a path in G^* in which control is passed to a procedure P from one call site, but control returns

[1]The vertices of a flowgraph can represent individual statements and predicates; alternatively, they can represent basic blocks.

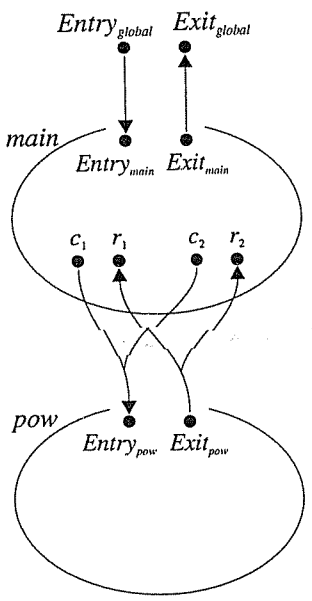


Figure 1: Schematic of the interprocedural linkages in the supergraph of a program in which *main* has two call sites on the procedure *pow*. (Figure 4 shows the source code for such a program.) Intraprocedural control-flow edges are not shown.

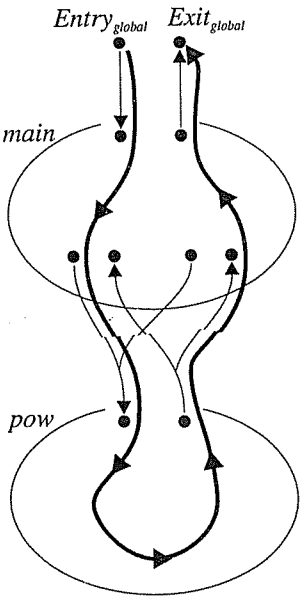


Figure 2: Example of an invalid path in a supergraph. The path enters *pow* from one call site, but returns from *pow* to a different call site.

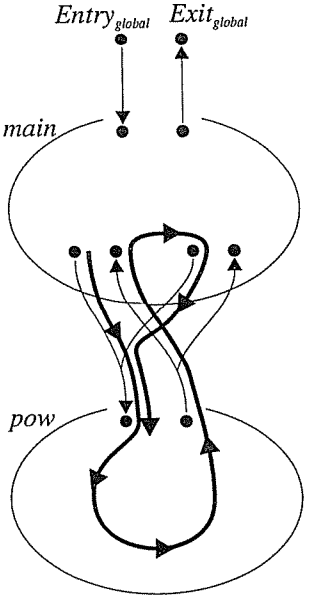


Figure 3: Example of a cycle that may occur in a valid path. The path enters *pow* from two distinct call sites.

from P to a different call site cannot represent a feasible execution path (see Figure 2). There may also be logically correlated branch statements in a program such that the outcome of one branch determines the outcome of the other branch. For example, it may be that if branch b_1 is taken, then branch b_2 (logically) cannot be taken. In this case, a path in G^* in which both branches are taken cannot represent a feasible execution path.

For purposes of profiling, we will assume that all branches are logically independent, *i.e.*, the result of one branch does not affect the ability to take any other branch; however, we do not wish to consider paths in G^* that violate the nature of procedure calls (as the path in Figure 2 does).[2] We now develop a language for describing the set of paths in G^* that we wish to consider valid. To do this, let each call site be assigned a unique index between 1 and *NumCallSites*, where *NumCallSites* is the total number of call sites in the program. Then, for each call site with index i, let the call-edge from the call vertex be labeled with the symbol "$(_i$", and let the return-edge to the corresponding return-site vertex be labeled with the symbol "$)_i$". Let each edge of the form $Entry_{global} \rightarrow Entry_P$ be labeled with the symbol "$(_P$" and each edge of the form $Exit_P \rightarrow Exit_{global}$ be labeled with the symbol "$)_P$". (In G^*, the edges $Entry_{global} \rightarrow Entry_{main}$ and $Exit_{main} \rightarrow Exit_{global}$ are the only edges that are labeled in this fashion; however, Section 3.2 introduces a transformed version of G^* that adds additional edges with the forms $Entry_{global} \rightarrow Entry_P$ and $Exit_P \rightarrow Exit_{global}$.) Let all other edges be labeled with the symbol e. Then a path p in G^* is a *same-level valid path* if and only if the string formed by concatenating the labels of p's edges is derived from the non-terminal *SLVP* in the following context-free grammar:

[2]This is similar to the assumptions that are made in much of the work on context-sensitive interprocedural dataflow analysis, *e.g.*, [17, 8, 11, 10, 15].

$SLVP ::= (_i \ \ SLVP \ \ )_i \ \ SLVP$ for $1 \leq i \leq NumCallSites$
$SLVP ::= (_P \ \ SLVP \ \ )_P \ \ SLVP$ for each procedure P
$SLVP ::= e \ \ SLVP$
$SLVP ::= \epsilon$

Here, ϵ denotes the empty string. A same-level valid path p represents an execution sequence where every call-edge is properly matched with a corresponding return-edge, and vice versa. Note that every execution of the program must follow a same-level valid path from $Entry_{global}$ to $Exit_{global}$.

We also need to describe paths that correspond to incomplete execution sequences in which not all of the procedure calls have been completed (for example, a path that begins in a procedure P, crosses a call-edge to a procedure Q, and ends in Q). Such a path is called an *unbalanced-left path*. In general, path p is an unbalanced-left path if and only if the string formed by concatenating the labels on p's edges is derived from the non-terminal *UnbalLeft* in the following context-free grammar:

$UnbalLeft ::= UnbalLeft \ \ (_i \ \ UnbalLeft$ for $1 \leq i \leq NumCallSites$
$UnbalLeft ::= UnbalLeft \ \ (_P \ \ UnbalLeft$ for each procedure P
$UnbalLeft ::= SLVP$

where *SLVP* is defined by the productions given above.

We will also use paths that are the dual of unbalanced-left paths. A path p is called an *unbalanced-right path* if and only if the string formed by concatenating the labels of p's edges can be derived from the non-terminal *UnbalRight* in the following the context-free grammar:

$UnbalRight ::= UnbalRight \ \ )_i \ \ UnbalRight$ for $1 \leq i \leq NumCallSites$
$UnbalRight ::= UnbalRight \ \ )_P \ \ UnbalRight$ for each procedure P
$UnbalRight ::= SLVP$

An unbalanced-right path represents part of an incomplete execution sequence. Specifically, an unbalanced-right path p may leave a procedure P (where the execution sequence that reached P is not part of p). For example a path that begins in a procedure P, crosses a return-edge to a procedure Q, and ends in Q is an unbalanced-right path.

Finally, we will also be interested in paths that are the concatenation of an unbalanced-right path and an unbalanced-left path. A path is called an *unbalanced-right-left path* if and only if the string formed by concatenating the labels of p's edges can be derived from the non-terminal *UnbalRtLf* in the following context-free grammar:

$UnbalRtLf ::= UnbalRight \ \ UnbalLeft$

3 Overview of Interprocedural Context Path Profiling

3.1 An Introductory Example

In this section, we give a brief example that illustrates some of the difficulties that arise in collecting an interprocedural path profile. In particular, we consider the code shown in Figure 4. Figure 1 shows a schematic of the supergraph G^* for this program. One difficulty that arises in interprocedural path profiling comes from interprocedural cycles. Even if all intraprocedural cycles are broken, G^* will still contain cyclic paths, namely, those paths that enter a procedure from distinct call sites (see Figure 3). This complicates any interprocedural extension to the Ball-Larus technique, because the Ball-Larus numbering scheme works on acyclic graphs. There are several possible approaches to overcoming this difficulty:

- One possible approach is to remove all but one pair of call and return edges to a procedure P, and insert some appropriate surrogate edges. This approach has the drawback of breaking most interprocedural paths into intraprocedural paths.
- A second approach is to inline every non-recursive procedure, and break call and return edges to recursive procedures. The Ball-Larus technique can then be used to collect intraprocedural path profiles for the procedures that are left. Each intraprocedural path will correspond to an interprocedural path in the original program. Unfortunately, such extensive inlining is usually not feasible.

```
double pow(double base, long exp) {
   double power = 1.0;
   while( exp > 0 ) {
      power *= base;
      exp--;
   }
   return power;
}

int main() {
   double t, result = 0.0;
   int i = 1;
   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      if( (i%3) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      i++;
   }
   return 0;
}
```

Figure 4: Example program used to illustrate the path-profiling technique. (The program computes the quantity $(\sum_{j=1}^{9}(2 \cdot j)^2)) + (\sum_{k=1}^{6}(3 \cdot k)^2)$.)

- A third approach is to create a unique copy of each procedure for each call site, again with recursive call and return edges removed. In our example program, we would create the copies *pow1* and *pow2* of the *pow* function. The first call to *pow* in *main* would be changed to call *pow1* and the second call would be changed to call *pow2*. Then *pow1* can be instrumented as if it had been inlined in *main*, and likewise for *pow2*. Clearly, this approach is just as impractical as extensive inlining.
- A fourth approach—which is the one developed in this paper—is to parameterize the instrumentation code added to each procedure to behave differently for different calling contexts. In our example, *pow* is changed to take an extra parameter. When *pow* is called from the first call site in *main*, the value of the new parameter causes the instrumentation of *pow* to mimic the behavior of the instrumentation of *pow1* in the third approach above; when *pow* is called from the second call site in *main*, the value of the new parameter causes *pow*'s instrumentation to mimic behavior of the instrumentation of *pow2*. Thus, by means of an appropriate parameterization, we gain the advantages of the third approach without paying the costs in space of duplicating code.

In fact, there are some technical details that make the fourth approach difficult. (In particular, it is not clear how the instrumentation of *pow* can efficiently generate the edge-increment value that, in the third approach, would occur on a surrogate edge from $Entry_{main}$ into a duplicate of *pow*.) However, the description above captures the flavor of our interprocedural path-profiling technique. There are three main issues that must be addressed:

1. how to deal with intraprocedural backedges;
2. how to deal with recursion; and
3. how parameterization is carried out.

The following section (Section 3.2) discusses transformations on the supergraph that are used in interprocedural profiling to address the first and second issues. The third issue is addressed at a high level in Section 3.3

and in detail in Section 5.5.

3.2 Modifying G^* to Eliminate Backedges and Handle Recursion

Recall that, for purposes of numbering paths, the Ball-Larus technique modifies a procedure's control-flow graph to remove cycles. This section describes the analogous step for interprocedural context profiling. Specifically, this section describes modifications to G^* that remove cycles from each procedure and from the call graph associated with G^*. The resulting graph is called G^*_{fin}. Each unbalanced-left path through G^*_{fin} corresponds to an "observable path" in G^* that can be logged in an interprocedural profile. As will be seen, the number of unbalanced-left paths through G^*_{fin} is finite, which is the reason for the subscript "*fin*".

In total, there are three transformations that are performed to create G^*_{fin}. Figure 5 shows the transformed graph G^*_{fin} that is constructed for the example program in Figure 4. The first transformation is as follows:

Transformation 1: For each procedure P, add a special vertex $GExit_P$ to the control-flow graph for P and an edge $GExit_P \rightarrow Exit_{global}$ to the supergraph. (The vertex $GExit_P$ serves as a local copy of the vertex $Exit_{global}$.)

The second transformation removes cycles in each procedure's control-flow graph. As in the Ball-Larus technique, the procedure's control-flow graph does not need to be reducible; backedges can be determined by a depth-first search of the control-flow graph.

Transformation 2: For each procedure P, perform the following steps:

1. For each backedge target v in P, add a surrogate edge $Entry_P \rightarrow v$.
2. For each backedge source w in P, add an edge $w \rightarrow GExit_P$.
3. Remove all of P's backedges.

The third transformation "short-circuits" paths around recursive call sites, effectively removing cycles in the call graph. First, each call site is classified as recursive or nonrecursive. This can be done by identifying backedges in the call graph using depth-first search; the call graph need not be reducible.

Transformation 3: The following modifications are made:

1. For each procedure R called from a recursive call site, the edges $Entry_{global} \rightarrow Entry_R$ and $Exit_R \rightarrow Exit_{global}$ are added.
2. For each pair of vertices c and r representing a recursive call site that calls procedure R, the edges $c \rightarrow Entry_R$ and $Exit_R \rightarrow r$ are removed, and the summary edge $c \rightarrow r$ is added. (Note that $c \rightarrow r$ is called a "summary" edge, but not a "surrogate" edge; this slight distinction is used when we describe the meaning of a context path profile. See Section 3.4.)

The following example discusses these transformations as they apply to Figure 5:

Example 3.1 Figure 5 shows the transformed graph G^*_{fin} that is constructed for the program in Figure 4. In Figure 5, the vertices v_{14} and u_6 are inserted by Transformation 1. The edges $v_1 \rightarrow v_4$ and $v_{13} \rightarrow v_{14}$ are inserted by Transformation 2 (and backedge $v_{13} \rightarrow v_4$ is removed). Similarly, Transformation 2 inserts the edges $u_1 \rightarrow u_3$ and $u_5 \rightarrow u_6$, and removes the backedge $u_5 \rightarrow u_3$. Transformation 3 is not illustrated in Figure 5, because the program from Figure 4 does not use recursion. □

As was mentioned above, the reason we are interested in these transformations is that each observable path—an item that we log in an interprocedural path profile—corresponds to an unbalanced-left path through G^*_{fin}. Note that the observable paths should not correspond to just the same-level valid paths through G^*_{fin}: as a result of Transformation 2, an observable path p may end in the middle of a procedure, leaving unclosed left parentheses. Furthermore, a path in G^*_{fin} that is not unbalanced-left cannot represent any feasible execution path in the original graph G^*.

A crucial fact on which this approach to interprocedural path profiling rests is that the number of unbalanced-left paths through G^*_{fin} is finite. To prove this we start with the following observation:

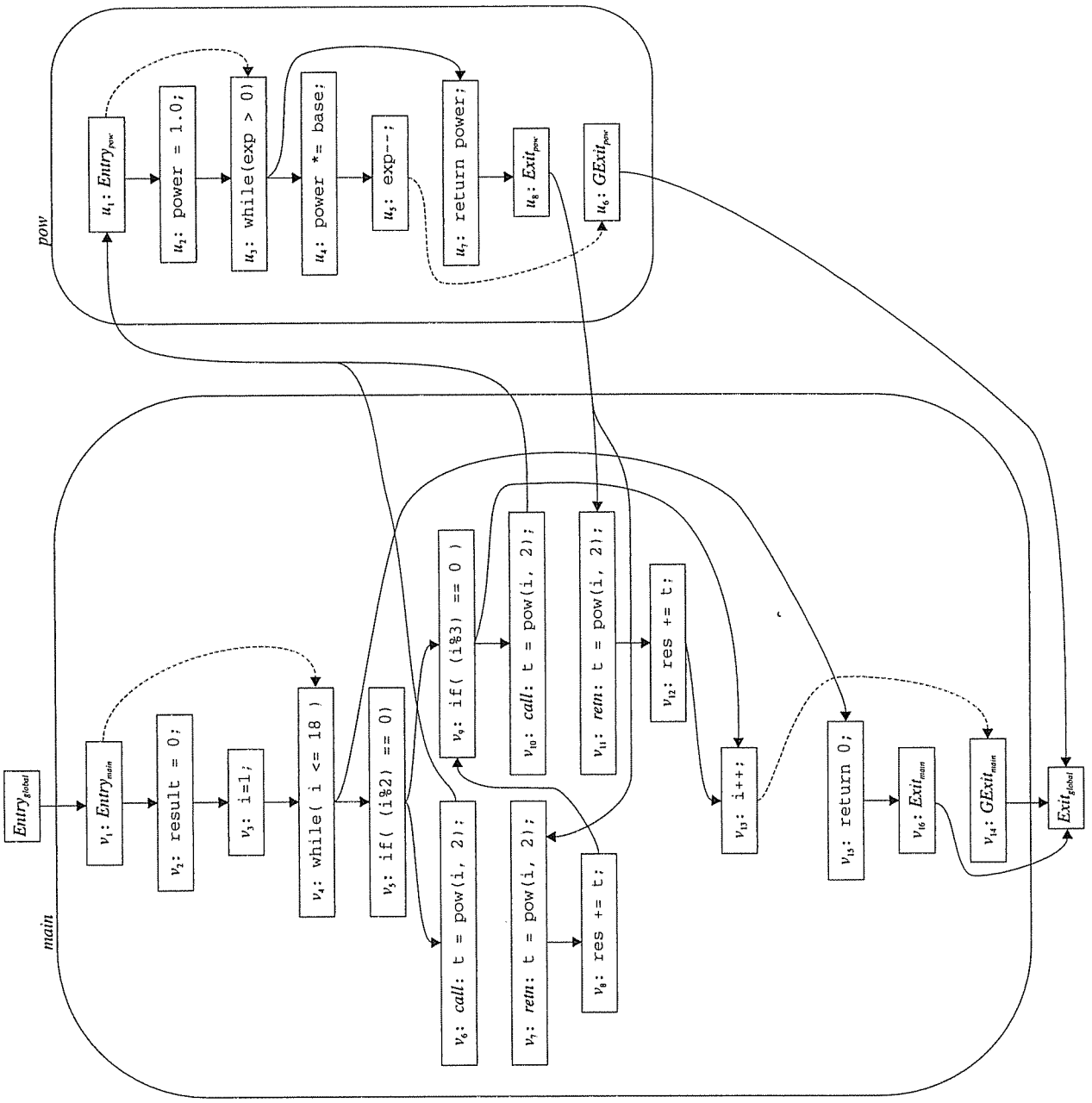


Figure 5: G^*_{fin} for the code in Figure 4. Dashed edges represent surrogate edges; the supergraph for the program in Figure 4 includes the backedges $v_{13} \rightarrow v_4$ and $u_5 \rightarrow u_3$, which have been removed here by Transformation 2.

Observation 3.1 The maximum number of times a vertex v in a procedure P can appear on an unbalanced-left path p (in G^*_{fin}) is equal to the number of times p enters the procedure P by reaching the vertex $Entry_P$. □

This follows from the fact that G^*_{fin} contains no intraprocedural loops. It also relies on the fact that p is an unbalanced-left path; a path q that is not unbalanced-left may reach some vertices an arbitrary number of times. For example, consider a vertex u in a procedure P where u lies in the middle of a path that connects one call site on Q to a second call site on Q. Then a path that is not an unbalanced-left path can contain an arbitrary number of traversals of the cycle that runs from u to the second call site on Q, enters Q from the second call site, then returns to the first call site, and then reaches u. Note that this cycle cannot occur

in an unbalanced-left path.

Next, we wish to calculate the number $maxEnters[P]$, which is an upper bound on number of times the vertex $Entry_P$ can occur on an unbalanced-left path. (As we shall see, $maxEnters[P]$ is well defined and finite.) We observe that for $P \neq main$, the number of times $Entry_P$ occurs on an unbalanced-left path p is bounded by the maximum number of times a vertex for a call site on P can occur on the path p. Together with Observation 3.1 (which applies to call vertices), this implies that for all $P \neq main$,

$$maxEnters[P] = \sum_{Q} maxEnters[Q] \cdot (\text{number of non-recursive call sites on } P \text{ in } Q).$$

We also have $maxEnters[main] = 1$. For $P \neq main$, $maxEnters[P]$ is well defined and has a finite value because there are no recursive calls in G^*_{fin}; thus, we can solve for the $maxEnters[P]$ values by considering the vertices of the call graph associated with G^*_{fin} (which is acyclic) in topological order.

Observation 3.1, together with the fact that $maxEnters[P]$ is finite for all P, means that that for all vertices v, there is a finite bound on the number of times v may occur in an unbalanced-left path p. This implies that there is an upper bound on the length of an unbalanced-left path, and that the number of unbalanced-left paths is finite.

In the following section, we motivate the use of G^*_{fin} in collecting an interprocedural context profile by considering the example code in Figure 4. Section 5.1 discusses in detail how to obtain a dense numbering of the unbalanced-left paths in G^*_{fin}, in the general case.

3.3 Numbering Unbalanced-Left Paths: A Motivating Example

Extending the Ball-Larus technique to number unbalanced-left paths in G^*_{fin} is complicated by the following facts:

1. While the number of unbalanced-left paths is finite, an unbalanced-left path may contain cycles (such as the one shown in Figure 3).

2. The number of paths that may be used to extend a given path that reaches a vertex v is dependent on the path taken to reach v: For a given path p to vertex v, not every path q from v forms an unbalanced-left path when concatenated with p.

These facts mean that it is not possible to assign a single integer value to each vertex and edge of G^*_{fin} as the Ball-Larus technique does. Instead, each occurrence of an edge e in a path p will contribute a value to the path number of p, but the value that an occurrence of e contributes will be dependent on the part of p that precedes that occurrence of e.

To motivate our approach to solving these problems, we return to the sample program shown in Figure 4. Figure 5 shows the graph G^*_{fin} constructed for this program. (For the remainder of this section, "G^*_{fin}" refers to the graph in Figure 5.) Notice that G^*_{fin} contains cyclic, unbalanced-left paths. For example, the unbalanced-left path

$$Entry_{global} \to v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to u_1 \to u_3 \to u_7 \to u_8 \to v_7 \to v_8 \to v_9 \to v_{10} \to u_1 \to u_3 \tag{3}$$

contains the cycle

$$u_1 \to u_3 \to u_7 \to u_8 \to v_7 \to v_8 \to v_9 \to v_{10} \to u_1$$

as a subpath.

Figure 6 shows a modified version of G^*_{fin} with two copies of the procedure *pow*, one for each call site on *pow* in *main*. This modified graph is acyclic, and is amenable to the Ball-Larus numbering scheme; in fact the numbers annotating the vertices and edges of the graph in Figure 6 are the Ball-Larus numbers. Note that there is a one-to-one and onto mapping between the paths through this graph and the unbalanced-left paths through G^*_{fin}. For example the path

$$Entry_{global} \to v_1 \to v_4 \to v_5 \to v_9 \to v_{10} \to u_1 \to u_3 \to u_4 \to u_5 \to u_6 \to Exit_{global}$$

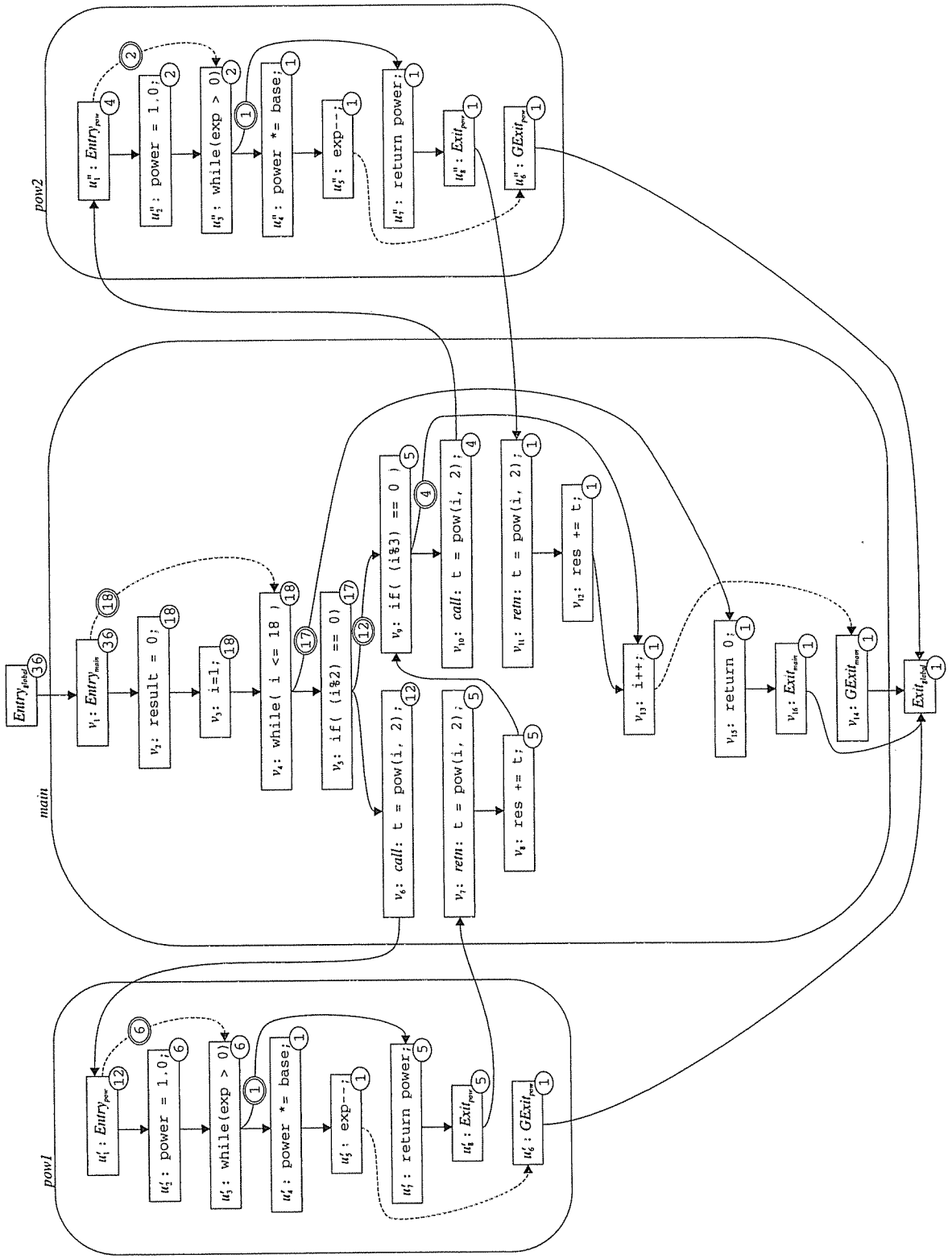


Figure 6: Modified version of G^*_{fin} from Figure 5 in which two copies of *pow* have been created. Labels on the vertices and edges show the results of applying the Ball-Larus numbering technique to the graph. Each vertex label is shown in a circle, and each edge label is shown in a double circle. Labels have not been shown for edges that are given the value 0 by the Ball-Larus numbering scheme.

in G^*_{fin} (see Figure 5) corresponds to the path

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{10} \rightarrow u''_1 \rightarrow u''_3 \rightarrow u''_4 \rightarrow u''_5 \rightarrow u''_6 \rightarrow Exit_{global}$$

in the graph from Figure 6. This correspondence can be used to number the unbalanced-left paths through G^*_{fin}: each unbalanced-left path p through G^*_{fin} is assigned the path number of its corresponding path q through the graph in Figure 6. As will be seen shortly, this is one way to view our approach to numbering unbalanced-left paths in G^*_{fin}.

The essence of our technique can be understood from the following two observations:

- Because the labeling passes of the Ball-Larus scheme work in reverse topological order, the values assigned to the vertices and edges of a procedure are dependent upon the values assigned to the exit vertices of the procedure. For instance, in Figure 6, the values assigned to the vertices and edges of *pow1* are determined by the values assigned to $Exit_{pow1}$ and $GExit_{pow1}$ (*i.e.*, the values 5 and 1, respectively), while the values assigned to the vertices and edges of *pow2* are determined by the values assigned to $Exit_{pow2}$ and $GExit_{pow2}$ (*i.e.*, the values 1 and 1, respectively). As will become apparent from the discussion in Section 5.2, $numPaths[GExit_P] = 1$ for any procedure P. Thus, thus the reason for the differences between the values on the edges and the vertices of *pow1* and the values on the corresponding edges and vertices of *pow2* is that $numPaths[Exit_{pow1}] \neq numPaths[Exit_{pow2}]$.

- Given that a program transformation based on duplicating procedures is undesirable, a mechanism is needed that assigns vertices and edges different numbers depending on the calling context. To accomplish this, each vertex u of each procedure P is assigned a linear function ψ_u that, when given a value for $numPaths[Exit_P]$, returns the value of $numPaths[u]$. Similarly, each edge e of each procedure P is assigned a linear function ρ_e that, when given a value for $numPaths[Exit_P]$, returns the Ball-Larus value for e.

The ψ functions are similar to the ϕ functions of Sharir and Pnueli's functional approach to interprocedural dataflow analysis [17]. In their work, the function ϕ_v summarizes how dataflow facts at a vertex v are related to the dataflow facts at the entry vertex $Entry_P$. In our technique, the function ψ_v summarizes how the value of $numPaths[v]$ on a vertex v is related to the value $numPaths[Exit_P]$ on the exit vertex $Exit_P$. The ρ functions have no direct analog in the Sharir and Pnueli framework, but are similar in spirit: the function ρ_e summarizes how the Ball-Larus value on the edge e is related to the value $numPaths[Exit_P]$ on the exit vertex $Exit_P$.

Figure 7 shows G^*_{fin} labeled with the appropriate ψ and ρ functions. Note that we have the desired correspondence between the linear functions in Figure 7 and the integer values in Figure 6:

- For an example of the correspondence between vertices, consider the vertex u_1 in Figure 7, with its linear function $\psi_{u_1} = \lambda x.2 \cdot x + 2$. This function, when supplied with the value $numPaths[Exit_{pow1}] = 5$ from Figure 6 evaluates to the value 12, which is equal to $numPaths[u'_1]$ in Figure 6. However, when $\lambda x.2 \cdot x + 2$ is given the value $numPaths[Exit_{pow2}] = 1$, it evaluates to 4, which is equal to $numPaths[u''_1]$ in Figure 6.

- For an example of the correspondence between edges, consider the edge $u_1 \rightarrow u_3$ and its linear function $\rho_{u_1 \rightarrow u_3} = \lambda x.x + 1$. This function, when supplied with the value $numPaths[Exit_{pow1}] = 5$ evaluates to 6, which is the value on the edge $u'_1 \rightarrow u'_3$. When $\lambda x.x + 1$ is given the value $numPaths[Exit_{pow2}] = 1$, it evaluates to 2, which is the value on the edge $u''_1 \rightarrow u''_3$.

To collect the number associated with an unbalanced-left path p in G^*_{fin}, as p is traversed, each edge e contributes a value to p's path number. As shown in the following example, the value that e contributes is dependent on the path taken to e:

Example 3.2 For example, consider the edge $u_1 \rightarrow u_3$ in G^*_{fin}, and an unbalanced-left path s that begins with the following path prefix:

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \tag{4}$$

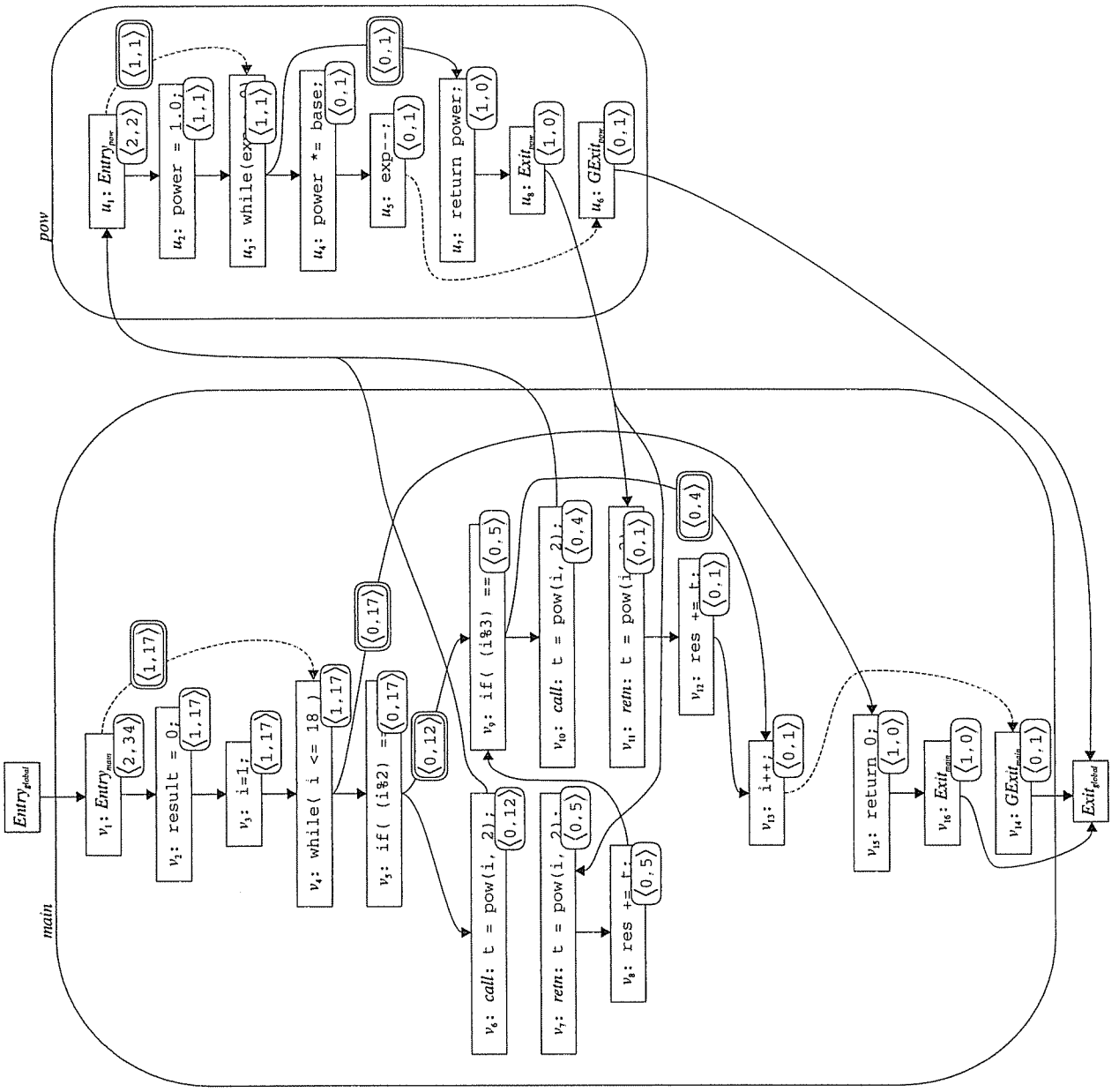


Figure 7: Labeled version of G^*_{fin} from Figure 5. Here the ordered pair $\langle a, b\rangle$ represents the linear function $\lambda x.a \cdot x + b$. Each vertex label is shown in a rounded box, and each edge label is shown in a doubled, rounded box. Unlabeled intraprocedural edges have the function $\langle 0, 0\rangle$; interprocedural edges are not labeled. Only six edges are labeled with functions other than $\langle 0, 0\rangle$: $v_1 \to v_4$ ($\langle 1, 17\rangle$), $v_4 \to v_{15}$ ($\langle 0, 17\rangle$), $v_5 \to v_9$ ($\langle 0, 12\rangle$), $v_9 \to v_{13}$ ($\langle 0, 4\rangle$), $u_1 \to u_3$ ($\langle 1, 1\rangle$), and $u_3 \to u_7$ ($\langle 0, 1\rangle$).

In this case, the edge $u_1 \to u_3$ contributes a value of 6 to s's path number. That this is the appropriate value follows from the fact that $u_1 \to u_3$ is replaced by the edge $u'_1 \to u'_3$ in the path prefix in Figure 6 that corresponds to (4):

$$Entry_{global} \to v_1 \to v_4 \to v_5 \to v_6 \to u'_1 \to u'_3$$

In Figure 6, the value on edge $u'_1 \to u'_3$ is 6.

In contrast, in an unbalanced-left path t that begins with the path prefix

$$Entry_{global} \to v_1 \to v_4 \to v_5 \to v_9 \to v_{10} \to u_1 \to u_3 \tag{5}$$

```
unsigned int profile[36];        /* 36 possible paths in total */

double pow(double base, long exp,
           unsigned int &pathNum, unsigned int numValidCompsFromExit)
{
   unsigned int pathNumOnEntry = pathNum;     /* Save pathNum's value
                                               * to capture the calling context */
   double power = 1.0;

   while( exp > 0 ) {
      power *= base;
      exp--;
      profile[pathNum]++;
      /* From surrogate edge u1->u3: */
      pathNum = 1 * numValidCompsFromExit + 1 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 1;     /* From edge u3->u7 */
   return power;
}
```

Figure 8: Part of the instrumented version of the program that computes $(\sum_{j=1}^{9}(2 \cdot j)^2) + (\sum_{k=1}^{6}(3 \cdot k)^2)$. The original program is shown in Figure 4. The instrumented version of *main* is shown in Figure 9. Instrumentation code is shown in italics.

the edge $u_1 \rightarrow u_3$ will contribute a value of 2 to t's path number. To see that this is the correct value, consider the path prefix in Figure 6 that corresponds to (5):

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1'' \rightarrow u_3''$$

In Figure 6, the value on edge $u_1'' \rightarrow u_3''$ is 2.

It can even be the case that an edge e occurs more than once in a path p, with each occurrence contributing a different value to p's path number. For example, there are some unbalanced-left paths in G^*_{fin} in which the edge $u_1 \rightarrow u_3$ appears twice, contributing a value of 6 for the first occurrence and a value of 2 for the second occurrence. (For example, see the path (3) that contains two occurrences of the edge $u_1 \rightarrow u_3$.)

To determine the value that an occurrence of the edge e should contribute to a path number, the profiling instrumentation will use the function ρ_e and the appropriate value for $numPaths[Exit_P]$, where P is the procedure containing e. Thus, as noted above, an occurrence of the edge $u_1 \rightarrow u_3$ may contribute the value $(\lambda x.x + 1)(1) = 2$ or the value $(\lambda x.x + 1)(5) = 6$ to a path number, depending on the path prefix that precedes the occurrence of $u_1 \rightarrow u_3$.

(The method for making the appropriate values for $numPaths[Exit_P]$ available at runtime is presented in Section 5.4.) □

Figures 8 and 9 show the program from Figure 4 with additional instrumentation code—based on the linear functions given in Figure 7—that collects an interprocedural path profile. (In Figures 8 and 9, various simple optimizations—such as eliminating multiplication by 1 or dropping subterms containing multiplication by 0—have not been performed in order to make the correspondence clearer between the instrumentation code shown in Figures 8 and 9, and the edge functions shown in Figure 7.) The output from the instrumented program is as follows:

```
 0: 0    1: 0    2: 0    3: 0    4: 0    5: 0    6: 0    7: 0    8: 0
 9: 0   10: 0   11: 0   12: 0   13: 0   14: 0   15: 0   16: 1   17: 0
18: 9   19: 0   20: 0   21: 0   22: 0   23: 0   24: 9   25: 3   26: 0
27: 3   28: 3   29: 6   30: 3   31: 0   32: 3   33: 3   34: 5   35: 1
```

The instrumentation in this program is based on the linear functions shown in Figure 7. In a sense, the instrumentation emulates the instrumentation that would have resulted from transforming the original program to have two copies of *pow* and using the integer values found in Figure 6.

```
int main() {
   unsigned int pathNum = 0;
   unsigned int pathNumOnEntry = 0;
   unsigned int numValidCompsFromExit = 1;
   double t, result = 0.0;
   int i = 1;

   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 5 /* From vertex v7 */);
         /* On entry to pow: pathNum is 0 or 18; fourth arg. always 5 */
         /* On exit from pow: pathNum is 1, 7, 19, or 25 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 12;
      if( (i%3) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 1 /* From vertex v11 */);
         /* On entry to pow: pathNum is 1, 7, 12, 19, 25, or 30; fourth arg. always 1 */
         /* On exit from pow: pathNum is 2, 3, 8, 9, 13, 14, 20, 21, 26, 27, 31, or 32 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 4;      /* From edge v9->v13 */
      i++;
      profile[pathNum]++;
      /* From surrogate edge v1->v4: */
      pathNum = 1 * numValidCompsFromExit + 17 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 17;          /* From edge v4->v15 */
   profile[pathNum]++;

   for (i = 0; i < 36; i++) {
      cout.width(3); cout << i << ":";
      cout.width(2); cout << profile[i] << "  ";
      if ((i+1) % 9 == 0) cout << endl;
   }
   return 0;
}
```

Figure 9: Part of the instrumented version of the program that computes $(\sum_{j=1}^{9}(2 \cdot j)^2) + (\sum_{k=1}^{6}(3 \cdot k)^2)$. The original program is shown in Figure 4. The instrumented version of the function *pow* and the global declaration of *profile* is shown in Figure 8. Instrumentation code is shown in italics.

Section 5 presents the technical details of an algorithm for generating the appropriate linear functions. This algorithm assigns linear functions to the vertices and edges of G^*_{fin} *directly*, without creating a modified version of G^*_{fin}, such as the one shown in Figure 6, in which procedures are duplicated.

3.4 What Do You Learn From a Profile of Unbalanced-Left Paths?

Before examining the details of interprocedural path profile, it is useful to understand the information that is gathered in this approach. As mentioned above, the approach to interprocedural path profiling that is presented here is centered on unbalanced-left paths through G^*_{fin}:

- Each unbalanced-left path p from $Entry_{global}$ to $Exit_{global}$ corresponds to an observable path—an object that can be logged in the profile—and is associated with a counter.

- Each unbalanced-left path p from $Entry_{global}$ to $Exit_{global}$ can be thought of as consisting of a *context-prefix* and an *active-suffix*. The active-suffix q'' of p is a maximal-size, surrogate-free subpath at the tail of p (though the active-suffix may contain summary edges of the form $c \rightarrow r$, where c and r represent a recursive call site). The context-prefix q' of p is the prefix of p that ends at the last surrogate edge before p's active suffix. (It is possible that the context-prefix q' will be the empty path at $Entry_{global}$.)

- The counter associated with the unbalanced-left path p counts the number of times during a program's execution that the active-suffix of p occurs in the context summarized by p's context-prefix.

In some sense, the Ball-Larus technique is a degenerate case of the technique that has been described here. In particular, every path profiled by the Ball-Larus technique has an empty context-prefix. Because the profiled objects in our approach to interprocedural path profiling can have non-empty context-prefixes, we call the approach illustrated above "interprocedural *context* path profiling". In contrast, we call the Ball-Larus technique an example of "intraprocedural *piecewise* path profiling". A piecewise path-profiling technique defines a set of observable paths that *partition* any execution path and, reports how many times each observable path occurred during execution.

Section 6 presents a technique for "interprocedural piecewise path profiling" in which every profiled path has an empty context-prefix. Section 7 describes a modification to the Ball-Larus technique that results in a technique for "intraprocedural context profiling"; in this technique, a typical path consists of a context-prefix from a procedure P's entry vertex to a loop header and an active-suffix through a loop body.

4 Numbering Paths in a Context-Free Directed Acyclic Graph

The Ball-Larus path-numbering technique applies to directed acyclic graphs (DAGs). In this section, we discuss how to generalize the Ball-Larus technique to apply to a *Context-Free DAG*. A context-free DAG is defined as follows:

Definition 4.1 Let CF be a context-free grammar over an alphabet Σ. Let G be a directed graph whose edges are labeled with members of Σ. Let G have a unique entry vertex, $Entry$, and a unique exit vertex, $Exit$. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is in the language L defined by CF. The graph G and the context-free grammar CF constitute a *Context-Free DAG* if and only if the number of L-paths in G from $Entry$ to $Exit$ is finite. □

In Section 3.2, we showed that the number of unbalanced-left paths through G^*_{fin} is finite. Thus, the graph G^*_{fin}, together with the context-free grammar for unbalanced-left strings, constitutes a context-free DAG where $Entry$ is the vertex $Entry_{global}$ and $Exit$ is the vertex $Exit_{global}$. Note that a context-free DAG need not be acyclic, and hence might not be a DAG; however, just as the number of paths through a DAG is finite, the number of L-paths through a context-free DAG is finite.

We are now ready to describe a mechanism for numbering L-paths in a context-free DAG. The numbering of L-paths in a context-free DAG is necessarily more complex than assigning a single integer to each vertex and edge, as is done in the Ball-Larus technique (because a context-free DAG may contain cycles). Nevertheless, a number of comparisons can be made between our technique for numbering L-paths and the Ball-Larus technique for numbering paths in an acyclic graph. In the remainder of this section, we describe the functions *numValidComps* and *edgeValueInContext* that correspond, in a sense, to the vertex and edge values of the Ball-Larus technique. (These functions are used in Section 5.1.1 to give a theoretical justification for the ψ and ρ functions that were introduced in the previous section for numbering unbalanced-left paths (*i.e.*, L-paths) in G^*_{fin} (a context-free DAG). However, in another sense, in the interprocedural path-profiling techniques it is the ψ and ρ functions that correspond to the vertex and edge values of the Ball-Larus technique, in that the ψ and ρ functions are employed at runtime to calculate the edge-increment values used by the profiling instrumentation code, whereas *numValidComps* and *edgeValueInContext* are only referred to in order to argue the correctness of the interprocedural profiling techniques.)

The following list describes aspects of the Ball-Larus technique, and the corresponding aspect of our technique for numbering L-paths:

1. In the Ball-Larus technique, each vertex v is labeled with the number $numPaths[v]$ of paths from v to *Exit*. In our technique, it is necessary to define a function $numValidComps$ that takes an L-path prefix p from *Entry* to a vertex v and returns the number of paths from v to *Exit* that form an L-path when concatenated with p. Thus, $numValidComps(p)$ returns the number of *valid completions* of p.

   Furthermore, $numValidComps$ has two properties that are similar to corresponding properties of the $numPaths$ values:

   - In the Ball-Larus technique, $numPaths[Exit]$ is 1 because the only path from *Exit* to *Exit* is the path of length 0. For an L-path q from *Entry* to *Exit*, $numValidComps(q)$ is defined to be 1 because the only valid completion of q is the path of length 0 from *Exit* to itself. (In the rest of the paper, a path of length 0 is called an *empty path*. The empty path from a vertex v to itself is denoted by "$[\epsilon : v]$.")
   - In the Ball-Larus technique, the value $numPaths[Entry]$ is the total number of paths through the acyclic control-flow graph. In our technique, the value $numValidComps([\epsilon : Entry])$ is the total number of L-paths through G.

   Note that the definition of the function $numValidComps$ is dependent on the specific context-free DAG in question.

2. In the Ball-Larus technique, each edge e is labeled with an integer value that is used when computing path numbers. We define a function $edgeValueInContext$ that takes an L-path prefix p and an edge e, and returns an integer value for the edge e in the context given by p. (As in the case for numbering unbalanced-left paths, the value that an edge e contributes to an path number may depend on the path prefix up to e.) The definition of $edgeValueInContext$ is based on the concept of a *valid successor*: let p be an L-path prefix from *Entry* to a vertex v. A vertex w is a valid successor of p if w is a successor of v, and $[p \parallel v \rightarrow w]$ is an L-path prefix.[3] We are now ready to define the function $edgeValueInContext$ in terms of the function $numValidComps$: let $w_1, \ldots, w_k$ be the valid successors of the path p, where p is an L-path prefix from *Entry* to a vertex v. Then $edgeValueInContext(p, v \rightarrow w_i)$ is defined as follows:

$$edgeValueInContext(p, v \rightarrow w_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} numValidComps(p \parallel v \rightarrow w_j) & \text{otherwise} \end{cases} \quad (6)$$

   Note that this equation is similar to Equation (2), which is used in the Ball-Larus technique to assign values to edges. Equation (6) is illustrated in Figure 10.

   As mentioned above, the function $edgeValueInContext$ is used in computing path numbers. Note that for an edge $v \rightarrow w$, the path $[p \parallel v \rightarrow w]$ must be an L-path prefix, otherwise $edgeValueInContext(p, v \rightarrow w)$ is not defined.

3. In the Ball-Larus technique, the path number for a path p is the sum of the values that appear on p's edges. We define the path number of an L-path p to be the following sum:

$$\sum_{[p' \parallel v \rightarrow w] \text{ a prefix of } p} edgeValueInContext(p', v \rightarrow w). \quad (7)$$

   (In our interprocedural path-profiling techniques, at runtime, a running total is kept of the contributions of the edges of p', and as the edge $v \rightarrow w$ is traversed, the value of $edgeValueInContext(p', v \rightarrow w)$ is added to this sum. The challenge is to devise a method by which the contribution of edge $v \rightarrow w$ to the running sum, which is a function of the path p' seen so far (namely, $edgeValueInContext(p', v \rightarrow w)$), can be determined quickly, without an expensive examination of p'. A method for doing this using ψ and ρ functions is presented in Section 5.5.)

4. Just as the Ball-Larus technique generates a dense numbering of the paths in an acyclic control-flow graph, we have the following theorem:

[3] We use the notation $[p \parallel q]$ to denote the concatenation of the paths p and q; however, when $[p \parallel q]$ appears as an argument to a function, *e.g.*, $numValidComps(p \parallel q)$, we drop the enclosing square brackets.

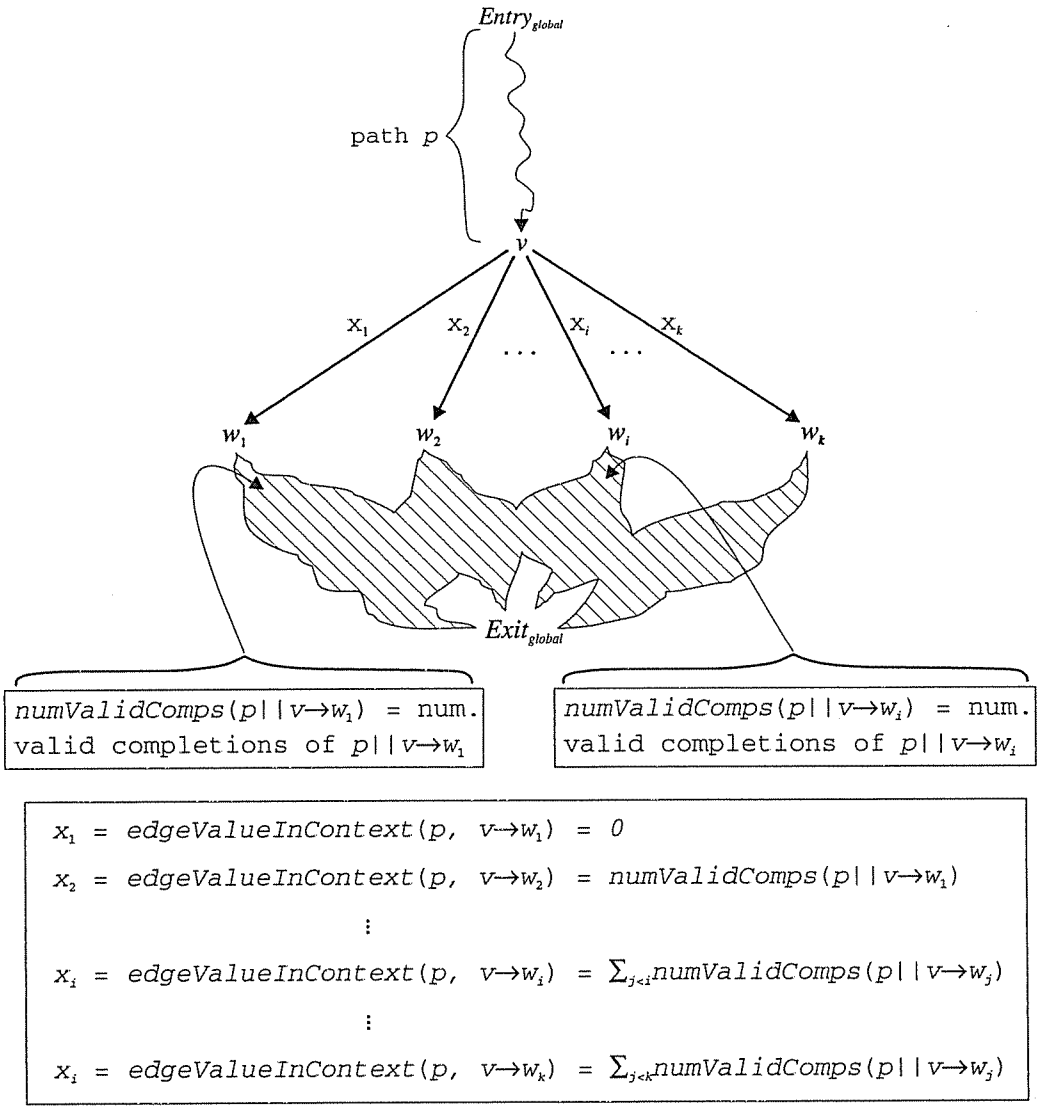


Figure 10: Illustration of the definition of *edgeValueInContext* given in Equation (6).

Theorem 4.1 *(Dense Numbering of L-paths) Given the correct definition of the function numValidComps, Equations (6) and (7) generate a dense numbering of the L-paths through G. That is, for every L-path p through G, the path number of p is a unique value in the range* $[0..(numValidComps([\epsilon : Entry]) - 1)]$. *Furthermore, each value in this range is the path number of an L-path through G.* □

Theorem 4.1 is proven in Appendix A.

Our interprocedural path-profiling techniques are based on the above technique for numbering L-paths in a context-free DAG. As mentioned above, G^*_{fin} is a context-free DAG, where unbalanced-left paths correspond to L-paths. In Section 3.3, we showed how to number unbalanced-left paths through G^*_{fin} by use of the ψ and ρ functions. As shown in Section 5.1.1, the ψ and ρ functions are actually being used to compute *numValidComps* and *edgeValueInContext*.

In Section 6, a modified version of G^*_{fin} is introduced, and we will be interested in *unbalanced-right-left paths* in this graph. In Section 6, we will show how to use the ψ and ρ functions to compute *numValidComps* and *edgeValueInContext* for the modified version of G^*_{fin}, with the context-free grammar for unbalanced-right-left paths.

5 Interprocedural Context Path Profiling

The first step in collecting an interprocedural path profile is to construct G^*_{fin}, as described in Section 3.2, and this section assumes that G^*_{fin} has been so constructed. The remainder of this section is organized as follows: Section 5.1 discusses the technical aspects of numbering the unbalanced-left paths in G^*_{fin}. Section 5.2 describes how to assign the ψ and ρ functions that are used in numbering unbalanced-left paths in G^*_{fin}. Section 5.3 describes how to compute *edgeValueInContext* for interprocedural edges of G^*_{fin}. Section 5.4 reviews how the ψ and ρ functions are used to calculate the path number of an unbalanced-left path. Section 5.5 describes the instrumentation code that is added to a program to collect an interprocedural path profile. Finally, Section 5.6 shows how to recover the path associated with a given path number.

5.1 Numbering Unbalanced-Left Paths in G^*_{fin}

5.1.1 Motivation Behind the ψ Functions

The graph G^*_{fin}, together with the context-free grammar for unbalanced-left strings, is an example of a context-free DAG. This means that the technique presented in Section 4 can be used to number unbalanced-left paths in G^*_{fin}. In particular, the L-paths of Section 4 are the unbalanced-left paths in G^*_{fin} that start at $Entry_{global}$ and end at $Exit_{global}$. The function *numValidComps* discussed in Section 4 takes an unbalanced-left path p (that starts at $Entry_{global}$ in G^*_{fin}) and return the number of valid completions of p. The function *edgeValueInContext* and the definition of a path number are exactly as described in Section 4.

In Section 5.2, we describe a technique that assigns a function ψ_v to each vertex v, and a function ρ_e to each intraprocedural edge e. These functions are used by the runtime instrumentation code to compute path numbers, which involves computing *numValidComps* and *edgeValueInContext*. This section starts by discussing properties of *numValidComps* that motivate the definition of the ψ functions. This is followed by a discussion that motivates the ρ functions.

First, observe that the following relation holds for *numValidComps*:

> Let p be an unbalanced-left path from $Entry_{global}$ to v, such that $v \neq Exit_{global}$. Let $w_1 \ldots w_k$ be the set of valid successors of p. Then
>
> $$numValidComps(p) = \sum_{i=1}^{k} numValidComps(p||v \rightarrow w_i). \tag{8}$$

This relation is very similar to the definition of *numPaths* (see Equation (1)). In particular, for any vertex v such that v is not an $Exit_P$ vertex, the set of valid successors for any path to v is the set of all successors of v, and so Equation (8) is identical to the definition of *numPaths*. For an $Exit_P$ vertex there is only one valid successor: for an unbalanced-left path p from $Entry_{global}$ to $Exit_P$, the label on the first edge of any valid completion of p must match the last open parenthesis that labels an edge of p. Note that this means that the number of valid completions for an unbalanced-left path p is completely determined by the last vertex of p and the sequence of unbalanced-left parentheses in p.

Now consider an unbalanced-left path p from $Entry_{global}$ to a vertex v and a same-level valid path q from v to a vertex u in the same procedure as v. Note that $[p \parallel q]$ is an unbalanced-left path and that the sequence of unbalanced-left parentheses in $[p \parallel q]$ is the same as in p alone. This implies that for an unbalanced-left path p from $Entry_{global}$ to v and a vertex u, the value of $numValidComps(p \parallel q)$ is the same for *any* same-level valid path q from v to u.

These observations help to motivate our approach for computing *numValidComps* for unbalanced-left paths. Consider an unbalanced-left path p from $Entry_{global}$ to a vertex v in procedure P. The path p determines (as described below) the value $numValidComps(p \parallel q')$ where q' is any same-level valid path from v to $Exit_P$. We also observe that the value $numValidComps(p \parallel q'') = 1$, where q'' is any same-level valid path from v to $GExit_P$. (The fact that $numValidComps(p \parallel q'')$ is always 1 follows from the fact that the only successor of $GExit_P$ is $Exit_{global}$.) Given the number of valid completions from $Exit_P$ and from $GExit_P$ (for a given p), it is possible to compute $numValidComps(p \parallel s)$ for any same-level valid path s from v to any vertex u (that is reachable from v) in procedure P. To aid in these computations, for each vertex v of procedure P, we define a function ψ_v that, when given the number of valid completions from $Exit_P$, returns

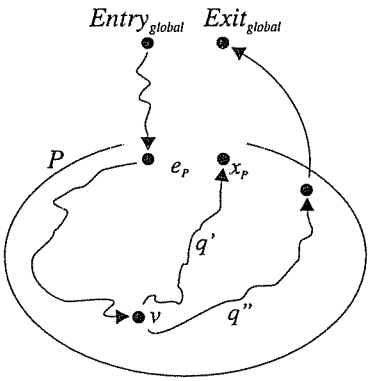


Figure 11: Schematic that illustrates the paths used to motivate the ψ functions. Vertices with labels of the form e_P, x_P, and g_P represent the vertices $Entry_P$, $Exit_P$, and $GExit_P$, respectively.

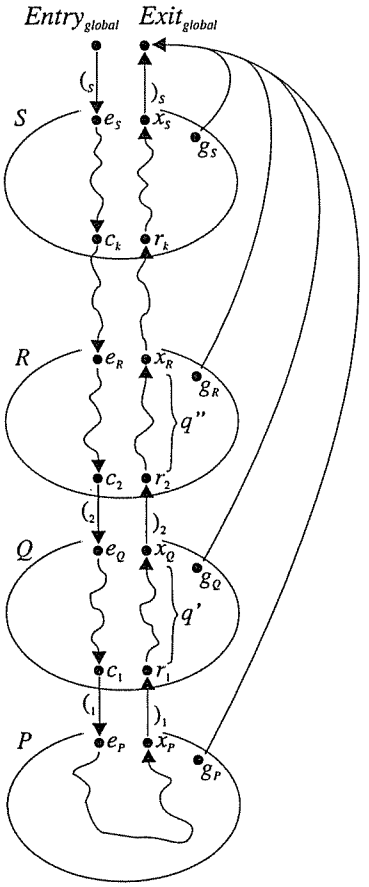


Figure 12: Schematic of the paths used to explain the use of ψ functions to compute $numValidComps(q)$, where q is an unbalanced-left path from $Entry_{global}$ to $Exit_P$. The unbalanced-left path s starts at $Entry_{global}$ and ends at $Exit_S$, and has q as a prefix. Vertices with labels of the form e_P, x_P, and g_P represent the vertices $Entry_P$, $Exit_P$, and $GExit_P$, respectively.

the number of valid completions from v. Note that the function ψ_v does not need to take the number of valid completions from $GExit_P$ as an explicit argument, because the number of valid completions of any path to $GExit_P$ is always 1.

The ψ functions will be used in calculating $numValidComps(p)$ for an unbalanced-left path p that ends

at a vertex $v \neq Exit_P$. They are also used to compute $numValidComps(q)$ for an unbalanced-left path q from $Entry_{global}$ to a vertex $Exit_P$. We now consider the latter use of the ψ functions. Recall that there is only one valid successor of q—the return vertex r_1 such that the label on the edge $Exit_P \rightarrow r_1$ matches the last open parenthesis of q. Thus, we have the following:

$$numValidComps(q) = numValidComps(q \parallel Exit_P \rightarrow r_1).$$

Suppose r_1 occurs in procedure Q. Then the above value is equal to

$$\psi_{r_1}(numValidComps(q \parallel Exit_P \rightarrow r_1 \parallel q')),$$

where q' is any same-level valid path from r_1 to $Exit_Q$. Recall that the function ψ_{r_1} counts the valid completions of $[q \parallel Exit_P \rightarrow r_1]$ that exit Q via $GExit_Q$, even though ψ_{r_1} only takes as an argument the number of valid completions for paths that exit Q via $Exit_Q$.

As before, there is only one valid successor of $[q \parallel Exit_P \rightarrow r_1 \parallel q']$: the return vertex r_2 such that $Exit_Q \rightarrow r_2$ is labeled with the parenthesis that closes the second-to-last open parenthesis in q. Suppose that r_2 is in procedure R. Then the value of $numValidComps(q)$ is equal to

$$\psi_{r_1}(\psi_{r_2}(numValidComps(q \parallel Exit_P \rightarrow r_1 \parallel q' \parallel Exit_Q \rightarrow r_2 \parallel q''))),$$

where q'' is any same-level valid path from r_2 to $Exit_R$. Again, ψ_{r_2} counts valid completions that leave R via either $GExit_R$ or $Exit_R$.

This argument can be continued until a path s has been constructed from $Entry_{global}$ to $Exit_S$, where S is the first procedure that q (and s) enters. The path s has q as a prefix, and has only one unmatched parenthesis, "$(_S$", which is the same as the first unmatched parenthesis in q. The parenthesis "$(_S$" is matched by the parenthesis "$)_S$", which can only appear on the edge $Exit_S \rightarrow Exit_{global}$. Thus, the number of valid completions of s is 1. This implies that

$$\begin{aligned} numValidComps(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(numValidComps(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(1)\ldots)) \end{aligned} \tag{9}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched parentheses in q. Figure 12 shows a schematic of the path s that is constructed to compute $numValidComps(q)$.

The ψ functions are also used to calculate the total number of unbalanced-left paths through G^*_{fin}, *i.e.*, $numValidComps([\epsilon : Entry_{global}])$:

$$numValidComps([\epsilon : Entry_{global}]) = \sum_{Entry_P \in succ(Entry_{global})} numValidComps(Entry_{global} \rightarrow Entry_P). \tag{10}$$

The value of $numValidComps(Entry_{global} \rightarrow Entry_P)$ can be computed using the function ψ_{Entry_P}: For a same-level valid path p from $Entry_P$ to $Exit_P$, the value of $numValidComps(Entry_{global} \rightarrow Entry_P \parallel p)$ is 1, because the only valid completion of $[Entry_{global} \rightarrow Entry_P \parallel p]$ is the edge $Exit_P \rightarrow Exit_{global}$; thus, for a path consisting of the edge $Entry_{global} \rightarrow Entry_P$, the value of $numValidComps(Entry_{global} \rightarrow Entry_P)$ is given by

$$numValidComps(Entry_{global} \rightarrow Entry_P) = \psi_{Entry_P}(1). \tag{11}$$

Substituting Equation (11) into Equation (10) yields the following:

$$numValidComps([\epsilon : Entry_{global}]) = \sum_{Entry_P \in succ(Entry_{global})} \psi_{Entry_P}(1).$$

5.1.2 Motivation Behind the ρ Functions

In addition to the ψ functions on vertices, functions are also assigned to edges. In particular, each intraprocedural edge e is assigned a function ρ_e. While the function ψ_v is used to compute $numValidComps(p)$ for a path p ending at vertex v, the function ρ_e is used to compute $edgeValueInContext(p, e)$ for a path p to the source vertex of e. Specifically, for each edge e of a procedure P, the function ρ_e takes the number of valid

completions from $Exit_P$ (for an unbalanced-left path p' to $Entry_P$ concatenated with any same-level valid path to $Exit_P$) and returns the value of $edgeValueInContext(p' \parallel q, e)$, where q is any same-level valid path from $Entry_P$ to the edge e.

In the following section, we first describe how to define the ψ functions and then show how to define the ρ functions. In Section 5.5, we show how to use these functions to instrument a program in order to collect an interprocedural profile.

5.2 Assigning ψ and ρ Functions

5.2.1 The Relationship of Sharir and Pnueli's ϕ Functions to ψ Functions

Recall that in the Ball-Larus technique, each vertex v is assigned an integer value $numPaths[v]$ that indicates the number of paths from v to the exit vertex. In this section, we first show that the problem of finding the value $numPaths[v]$ for each vertex v of a control-flow graph can be cast in a form that is similar to a backwards, intraprocedural dataflow-analysis problem. We then show that our scheme for assigning ψ functions to vertices is similar to Sharir and Pnueli's functional approach to interprocedural dataflow analysis [17].

A distributive, backwards dataflow-analysis problem includes a semi-lattice L with meet operator $\sqcap$, a set F of distributive functions from L to L, a graph G, and a dataflow fact c associated with the exit vertex $Exit$ of G. Each edge $v \dashrightarrow w$ of the graph is labeled with a function $f_{v \to w} \in F$. Kildall showed that the meet-over-all-paths solution to a distributive, backwards dataflow-analysis problem is given by the maximal fixed point of the following equations [9]:

$$val[v] = \prod_{w \in succ(v)} f_{v \to w}(val[w]) \quad \text{for } v \neq Exit \tag{12}$$

$$val[Exit] = c \tag{13}$$

For a DAG, finding the quantity $numPaths[v]$ amounts to a technique for summing over all paths between v and $Exit$, where each path contributes a value of one to the sum. Thus, $numPaths[v]$ can be considered to be a "sum-over-all-paths" value. To find the $numPaths$ values, the Ball-Larus technique finds the maximum fixed point of the following equations (over the integers together with ∞):

$$numPaths[v] = \sum_{w \in succ(v)} id(numPaths[w]) \qquad \text{for } v \neq Exit \tag{14}$$

$$numPaths[Exit] = 1 \tag{15}$$

The form of Equation (14) is similar to Equation (12), with the identity function id standing in for each edge function $f_{v \to w}$, and the addition operator $+$ playing the role of the meet operator $\sqcap$. When we simplify the right-hand side of Equation (14) to $\sum_{w \in succ(v)} numPaths[w]$, this is precisely the definition of $numPaths[v]$ given in Section 2.1 in Equation (1).

Thus, the problem of calculating $numPaths$ values is similar to a dataflow-analysis problem (on a DAG), differing only in that addition is not an idempotent meet operator. (In fact, the problem of calculating $numPaths$ values is an example of an algebraic path problem on a DAG. For a more general discussion of the relationship between algebraic path problems and dataflow-analysis problems, see [13].)

We now review the appropriate part of Sharir and Pnueli's work [17], with some rephrasing of their work to describe backwards dataflow-analysis problems instead of forwards dataflow-analysis problems. We then show how their ϕ functions are related to our ψ functions.

In Sharir and Pnueli's functional approach to interprocedural dataflow analysis, for each procedure P, and each vertex v of P, the function ϕ_v captures the transformation of dataflow facts from $Exit_P$ to v.[4] The ϕ functions are found by setting up and solving a system of equations. For an exit vertex $Exit_P$, ϕ_P is the identity function:[5]

$$\phi_{Exit_P} = id. \tag{16}$$

[4] Information flows counter to the direction of control-flow graph edges in a backwards dataflow-analysis problem.

[5] According to [17], this equation should be $\phi_{Exit_P} \sqsubseteq id$. Since we do not allow an exit vertex to be the source of an intraprocedural edge, it is safe to replace $\sqsubseteq$ with $=$.

For a call vertex c associated with return-site vertex r to the procedure Q, we have the following equation:

$$\phi_c = \phi_{Entry_Q} \circ \phi_r. \tag{17}$$

Finally, for any other vertex m in P, we have the following equation:

$$\phi_m = \bigsqcap_{n \in succ(m)} f_{m \to n} \circ \phi_n. \tag{18}$$

In a similar fashion, we wish to define, for each procedure P and each vertex v in P, a function ψ_v that calculates the number of valid completions from v to $Exit_{global}$ based on the number of valid completions from $Exit_P$ to $Exit_{global}$. The problem of finding the ψ functions differs from the one solved by Sharir and Pnueli in that the ϕ functions describe how dataflow facts are propagated in a dataflow-analysis problem and the ψ functions describe how values are propagated in the problem of assigning Ball-Larus-like values to vertices. However, as shown above, the problem of assigning Ball-Larus values to vertices is similar to a backwards dataflow-analysis problem, and as we show below, the equations that hold for the ψ functions are very similar to the equations that hold for the ϕ functions.

As in Sharir and Pnueli's functional approach to interprocedural dataflow analysis, several equations must hold. For an exit vertex $Exit_P$, ψ_{Exit_P} is the identity function:

$$\psi_{Exit_P} = id. \tag{19}$$

This is similar to Equation (16).

For a vertex of the form $GExit_P$, the following must hold:

$$\psi_{GExit_P} = \lambda x.1. \tag{20}$$

This equation reflects the fact that the number of valid completions from $GExit_P$ is always 1, regardless of the number of valid completions from $Exit_P$. Equation (20) does not have a direct analog in [17], however, $GExit_P$ could be thought of as a vertex that generates a constant dataflow fact.

For a call vertex c to a procedure Q associated with return-site vertex r, where c and r represent a non-recursive call site, we have the following:

$$\psi_c = \psi_{Entry_Q} \circ \psi_r. \tag{21}$$

This is similar to Equation (17).

For all other cases for a vertex m, the following must hold:

$$\psi_m = \sum_{n \in succ(m)} id \circ \psi_n. \tag{22}$$

where the addition $f+g$ of function values f and g is defined to be the function $\lambda x.f(x)+g(x)$. Equation (22) is similar to Equation (18) with the identity function id standing in for each edge function $f_{m \to n}$.

Just as Equations (16)–(18) are the interprocedural analogs of Equations (12) and (13), Equations (19)–(22) are the interprocedural analogs of Equations (14) and (15).

We now show that the solution to Equations (19)–(22) yields the desired ψ functions. Recall that, for a vertex v in procedure P, the function ψ_v takes the number of valid completions from $Exit_P$ (for an unbalanced-left path p to vertex v in procedure P concatenated with any same-level valid path from v to $Exit_P$) and returns the number of valid completions from v (for p). Given this definition of ψ_v, it is clear that Equations (19) and (20) must hold. Equation (22) must hold because of Equation (8) and the fact that for an internal vertex v and any unbalanced-left path p to v, the valid successors of p are the same as the successors of v.

Equation (21) requires more extensive justification. Let p be an arbitrary unbalanced-left path to $Entry_P$; let p' be an arbitrary same-level valid path from $Entry_P$ to a call vertex c; let c be associated with the return vertex r; let c and r represent a nonrecursive call site on procedure Q ; and let q be an arbitrary same-level valid path from $Entry_Q$ to $Exit_Q$. (Figure 13 illustrates these paths and vertices.) The function ψ_c takes

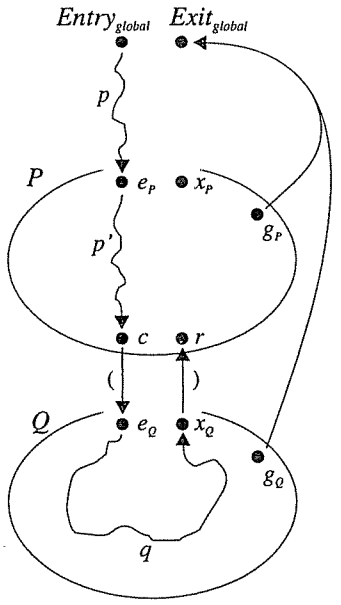


Figure 13: Paths used to illustrate the correctness of Equation (21).

the number N_p of valid completions from $Exit_P$ (for the path p concatenated with any same-level valid path from $Entry_P$ to $Exit_P$) and returns the number of valid completions from c (for $[p \parallel p']$).

By the definition of ψ_r, we know that the number of valid completions from vertex r (for p concatenated with any same-level valid path to r) is given by $\psi_r(N_p)$. In particular, we have the following:

$$numValidComps(p \parallel p' \parallel c \to Entry_Q \parallel q \parallel Exit_Q \to r) = \psi_r(N_p).$$

Now consider the path $[p \parallel p' \parallel c \to Entry_Q \parallel q]$. The last unmatched parenthesis in this path is on the edge $c \to Entry_Q$. This gives us the following (see Figure 13):

$$numValidComps(p \parallel p' \parallel c \to Entry_Q \parallel q) = \psi_r(N_p).$$

Because q is a same-level valid path from $Entry_P$ to $Exit_P$, we have (by the definition of ψ_{Entry_Q}) the following:

$$numValidComps(p \parallel p' \parallel c \to Entry_Q) = \psi_{Entry_Q}(\psi_r(N_p)).$$

Because $Entry_Q$ is the only successor of c, this gives us

$$numValidComps(p \parallel p') = \psi_{Entry_Q}(\psi_r(N_p)).$$

However, by the definition of ψ_c, $numValidComps(p \parallel p') = \psi_c(N_p)$. This gives us

$$\psi_c(N_p) = \psi_{Entry_Q}(\psi_r(N_p)) \tag{23}$$

We have shown that Equation (23) holds for an arbitrary unbalanced-left path $[p \parallel p']$ to c (to which there corresponds N_P, the number of valid completions from $Exit_P$ for p concatenated with any same-level valid path to $Exit_P$) and for an arbitrary same-level valid path $[c \to Entry_Q \parallel q \parallel Exit_Q \to r]$. It follows that

$$\psi_c = \psi_{Entry_Q} \circ \psi_r.$$

5.2.2 Solving for ψ Functions

We now describe how to find ψ functions that satisfy Equations (19)–(22). Note that each ψ function is a linear function of one variable. This follows because $id(= \lambda x.x)$ and $\lambda x.1$ are both linear functions of one variable, and the space of linear functions of one variable is closed under function composition and function addition. For the composition of two linear functions, we observe that

$$(\lambda x.a \cdot x + b) \circ (\lambda y.c \cdot y + d) = \lambda z.(a \cdot c) \cdot z + (a \cdot d + b).$$

For the addition of two linear function values (as defined above), we have

$$(\lambda x.a \cdot x + b) + (\lambda y.c \cdot y + d) = \lambda z.(\lambda x.a \cdot x + b)(z) + (\lambda y.c \cdot y + d)(z) = \lambda z.(a + c) \cdot z + (b + d).$$

The fact that each ψ function is a linear function of one variable means that they can be compactly represented as an ordered pair, with one coordinate representing the coefficient, and one coordinate representing the constant.

To find the ψ functions, each procedure P is visited in reverse topological order of the call graph, and each vertex v in P is visited in reverse topological order of P's control-flow graph. (For purposes of ordering the vertices of a procedure P, a return vertex r is considered to be a successor of its associated call vertex c.) As each vertex v is visited, the appropriate equation from Equations (19)–(22) is used to determine the function ψ_v.

The order of traversal guarantees that when vertex v is visited, all of the functions that are needed to determine ψ_v will be available. This follows from the fact that the call graph associated with G^*_{fin} is acyclic and the fact that the control-flow graph of each procedure in G^*_{fin} is acyclic. (The fact that the call graph and control-flow graphs are acyclic also explains why each vertex needs to be visited only once.) For instance, consider a call vertex c that is associated with return-site vertex r and calls procedure Q. When the vertex c is visited, the function ψ_r will be available (because vertices are visited in reverse topological order) and the function ψ_{Entry_Q} will be available (because procedures are processed in reverse topological order). Hence, Equation (21) can be used to determine ψ_c.

5.2.3 Solving for ρ functions

As mentioned above, a linear function is assigned to each intraprocedural edge e to aid in the calculation of *edgeValueInContext*. In particular, for an edge e in procedure P, we define the function ρ_e such that, when supplied with the number of valid completions from $Exit_P$ (for an unbalanced-left path $[p \,||e||\, q]$, where p is an unbalanced-left path that ends at the source vertex of edge e and q is any same-level valid path from the target vertex of e to $Exit_P$), it returns the value of *edgeValueInContext*(p, e).

Let v be an intraprocedural vertex that is the source of one or more intraprocedural edges. (That is, v cannot be a call vertex for a nonrecursive call–site, nor have the form $Exit_P$, nor have the form $GExit_P$.) Let $w_1 \ldots w_k$ be the successors of v. Recall that for a vertex such as v, for any unbalanced-left path p that ends at v, every successor of v is a valid successor of p. This is because no outgoing edge from v is labeled with a parenthesis. Given the definition of *edgeValueInContext* (see Equation (6)) and the definition of the ψ functions, it follows that the following equation holds:

$$\rho_{v \to w_i} = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} \psi_{w_j} & \text{otherwise} \end{cases} \tag{24}$$

Clearly, each ρ function is a linear function of one variable. Furthermore, Equation (24) can be used to find each ρ function after the appropriate ψ functions have been determined.

5.3 Computing *edgeValueInContext* for interprocedural edges

The ρ functions are only assigned to intraprocedural edges and they can be used to calculate *edgeValueInContext* when the second argument to *edgeValueInContext* is an intraprocedural edge. To compute the path number for an unbalanced-left path p it is also necessary to compute *edgeValueInContext* for certain interprocedural edges. This section describes how to compute *edgeValueInContext* for interprocedural edges.

In fact, for an interprocedural edge e and an unbalanced-left path p to e, the value of *edgeValueInContext*(p, e) is almost always 0. The only situation where this is not the case is when e is of the form $Entry_{global} \to Entry_Q$ and p is the path $[\epsilon : Entry_{global}]$. (Recall that as part of creating G^*_{fin}, Transformation 3 of Section 3.2 adds edges of the form $Entry_{global} \to Entry_Q$ and $Exit_Q \to Exit_{global}$ for each recursively called procedure Q.) This follows from the fact that for an unbalanced-left path p that ends at a call vertex, an $Exit_P$ vertex, or a $GExit_P$ vertex, p has only one valid successor.

Let us consider the case of an edge of the form $Entry_{global} \rightarrow Entry_Q$. The value of $edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_Q)$ is computed using Equation (6). This means that it is necessary to set a fixed (but arbitrary) ordering of the edges of the form $Entry_{global} \rightarrow Entry_P$. For convenience, we number each edge $Entry_{global} \rightarrow Entry_P$ according to this ordering, and use Q_i to refer to the procedure that is the target of the i^{th} edge. From Equation (6), the value of

$$edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{Q_i})$$

is as follows:

$$\begin{cases} 0 & \text{if } i = 0 \\ \sum_{j<i} numValidComps(Entry_{global} \rightarrow Entry_{Q_j}) & \text{otherwise} \end{cases} \tag{25}$$

As noted in Section 5.1.1, the value of $numValidComps(Entry_{global} \rightarrow Entry_{Q_j})$ is given by

$$numValidComps(Entry_{global} \rightarrow Entry_{Q_j}) = \psi_{Entry_{Q_j}}(1). \tag{26}$$

Substituting Equation (26) into Equation (25) yields the following

$$edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{Q_i}) = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j<i} \psi_{Entry_{Q_j}}(1) & \text{otherwise} \end{cases} \tag{27}$$

5.4 Calculating the Path Number of an Unbalanced-Left Path

In this section, we show how to calculate the path number of an unbalanced-left path p through G^*_{fin} from $Entry_{global}$ to $Exit_{global}$. This is be done during a single traversal of p that sums the values of $edgeValueInContext(p', e)$ for each p' and e such that $[p' \parallel e]$ is a prefix of p (*cf.* Equation (7)).

For interprocedural edges, the value of *edgeValueInContext* is calculated as described in Section 5.3. For an intraprocedural edge e in procedure P, the value of $edgeValueInContext(p', e)$ is calculated by applying the function ρ_e to the number of valid completions from $Exit_P$. (The number of valid completions from $Exit_P$ is determined by the path taken to $Entry_P$—in this case a prefix of p'.)

We now come to the crux of the matter: how to determine the contribution of an edge e when the edge is traversed (*i.e.*, how to determine the value $edgeValueInContext(p', e,))$ without incurring a cost for inspecting the path p'. The trick is that, as p is traversed, we maintain a value, `numValidCompsFromExit`, to hold the number of valid completions from the exit vertex $Exit_Q$ of the procedure Q that is currently being visited (the number of valid completions from $Exit_Q$ is uniquely determined by p'—specifically, the sequence of unmatched left parentheses in p'). The value `numValidCompsFromExit` is maintained by the use of a stack, `NVCStack`, and the ψ functions for return-site vertices. The following steps describe the algorithm to compute the path number for a path p (which is accumulated in the variable `pathNum`):

- When the traversal of p is begun, `numValidCompsFromExit` is set to 1. This indicates that there is only one valid completion from $Exit_R$, where R is the first procedure that p enters: if p reaches the exit of the first procedure it enters, then it must follow the edge $Exit_P \rightarrow Exit_{global}$. The value of `pathNum` is initialized to the value $edgeValueInContext([\epsilon : Entry_{global}], e)$ where e is the first edge of p (see Section 5.3).

- As the traversal of p crosses a call-edge $c \rightarrow Entry_T$ from a procedure S to a procedure T, the value of `numValidCompsFromExit` is pushed on the stack, and is updated to $\psi_r(numValidCompsFromExit)$, where r is the return-site vertex in S that corresponds to call vertex c. This reflects the fact that the number of valid completions from $Exit_T$ is equal to the number of valid completions from r.

- As the traversal of p crosses a return-edge $Exit_T \rightarrow r$ from a procedure T to a procedure S, the value of `numValidCompsFromExit` is popped from the top of the stack. This reflects the fact that the number of valid completions from the exit of the calling procedure S is unaffected by the same-level valid path that was taken through the called procedure T.

- As the traversal of p crosses an intraprocedural edge e, the value of `pathNum` is incremented by ρ_e(`numValidCompsFromExit`).

- At the end of the traversal of p, pathNum is output.

In essence, we have described the following algorithm:

Algorithm 5.1 (Calculate Path Number)
Input: An unbalanced-left path p from $Entry_{global}$ to $Exit_{global}$.
Output: p's path number.

initialize numValidCompsFromExit to 1
Initialize stack NVCstack to empty

Let e be the first edge of the path p. Calculate the value of $edgeValueInContext([\epsilon : Entry_{global}], e)$ as described in Section 5.3. Set pathNum to this value.

set e to the second edge of p
while e is not of the form $v \rightarrow Exit_{global}$ **do**
    if e is of the form $c \rightarrow Entry_T$ **then**
        push(NVCstack, numValidCompsFromExit)
        let r be the return vertex associated with c
        numValidCompsFromExit := ψ_r(numValidCompsFromExit)
    else if e is of the form $Exit_T \rightarrow r$ **then**
        numValidCompsFromExit := pop(NVCstack)
    else
        pathNum := pathNum + ρ_e(numValidCompsFromExit)
    fi
    set e to the next edge of p
od
output pathNum

□

5.5 Runtime Environment for Collecting a Profile

In this section, we describe the instrumentation code that is introduced to collect an interprocedural path profile. The instrumentation for a program $\mathcal{P}$ is based on the graph G^*_{fin} that is constructed for $\mathcal{P}$ as described in Section 3.2. In essence, the instrumentation code threads the algorithm described in Section 5.4 into the code of the instrumented program. Thus, the variables pathNum and numValidCompsFromExit become program variables. There is no explicit stack variable corresponding to NVCstack; instead, the program's execution stack is used. The variable pathNum and procedure parameter numValidCompsFromExit play the following roles in the instrumentation code (see Figures 8 and 9 for a concrete example):

pathNum: pathNum is a local variable of *main* that is passed by reference to each procedure $P \neq main$. It is used to accumulate the path number of the appropriate path in G^*_{fin}. As execution proceeds along the edges of the supergraph G^* of $\mathcal{P}$, the value of pathNum is updated. The profile is updated with the value in pathNum at appropriate places (*e.g.*, before an intraprocedural backedge of G^* is traversed).

numValidCompsFromExit: Each procedure $P \neq main$ is modified to take an additional parameter numValidCompsFromExit that is passed by value. This parameter is used to tell the instrumentation code in P the number of valid completions from $Exit_P$ for the path in G^*_{fin} that was used to reach P. The value in numValidCompsFromExit is used with the ρ functions to compute edge values for the edges of P.

Given a program $\mathcal{P}$ and the graphs G^* and G^*_{fin} that are associated with $\mathcal{P}$, the modifications described below are made to $\mathcal{P}$ in order to instrument it to collect an interprocedural context path profile. (As before, the ordered pair $\langle a, b \rangle$ denotes the linear function $\lambda x.a \cdot x + b$.) We use C++ terminology and syntax in the example instrumentation code.

1. A global declaration of the array `profile` is added to the program; `profile` is an array of `unsigned longs` that has $numValidComps([\epsilon : Entry_{global}])$ elements, *i.e.*, one for each unbalanced-left path through G^*_{fin}.[6]

2. Code is added to the beginning of *main* to initialize each element of `profile` to 0. Code is added just before *main* exits to output the contents of `profile`. This output constitutes the profile.

3. Declarations for the variables `pathNum`, `pathNumOnEntry`, `pathNumBeforeCall`, and `numValidCompsFromExit` are added to the beginning of procedure *main*. `pathNum` is an `unsigned long`[7] and is initialized to the value that is calculated for $edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{main})$; see Section 5.3. `pathNumOnEntry` is an `unsigned long` and is initialized to the same value as `pathNum`. `pathNumBeforeCall` is an `unsigned long` that is used to save the current value of `pathNum` before a recursive call is made. `numValidCompsFromExit` is an `unsigned long` initialized to 1. (Note that `pathNumOnEntry` and `numValidCompsFromExit` are somewhat redundant in *main*; they are added for consistency with the other procedures.)

4. For each procedure P such that $P \neq main$, P is modified to accept the following additional parameters:

```
unsigned long &numPaths                /* passed by reference */
unsigned long numValidCompsFromExit    /* passed by value */
```

   That is, a function prototype of the form

```
return_type func(...params...);
```

   becomes

```
return_type func(...params...,
                 unsigned long &numPaths,
                 unsigned long numValidCompsFromExit);
```

5. For each procedure $P \neq main$, the declarations

```
unsigned long pathNumOnEntry = pathNum;
unsigned long pathNumBeforeCall;
```

   are added to the declarations of P's local variables.

6. Each nonrecursive procedure call is modified to pass additional arguments as follows: Let the vertices c and r represent the following nonrecursive procedure call:

```
t = func(...args...);
```
(28)

   Let $\psi_r = \langle a, b \rangle$. Then the function call in (28) is replaced by the following call:

```
t = func(...args..., pathNum, a * numValidCompsFromExit + b);
```

7. Each recursive procedure call is modified as follows: Let the vertices c and r represent the following recursive procedure call from the procedure P to the function *func*:

```
t = func(...args...);
```
(29)

   Let x denote the value of $edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_{func})$ (see Section 5.3). Then the procedure call in (29) is replaced by the following code:

```
/* A: */ pathNumBeforeCall = pathNum;
/* B: */ pathNum = x;
/* C: */ t = func(...args..., pathNum, 1);
/* D: */ profile[pathNum]++;
/* E: */ pathNum = pathNumBeforeCall;
```

[6] In practice it is likely that a hash table would be used in place of the array `profile`.

[7] In practice, it is possible that the number of bits needed to represent a path number will not fit in an `unsigned long`. In this case, instead of using `unsigned longs` it may be necessary to use a `Counter` class and `Counter` objects to represent path numbers. The `Counter` class must behave like an `unsigned long` but be able to handle arbitrarily large integers.

The line labeled "A" saves the value of pathNum for the path that is being recorded before the recursive call is made. The lines "B" and "C" set up the instrumentation in func to start recording a new path number for the path that begins with the edge $Entry_{global} \rightarrow Entry_{func}$; the line "C" also makes the original procedure call in (29). The line "D" updates the profile with the unbalanced-left path in G^*_{fin} that ends with the edge $Exit_{func} \rightarrow Exit_{global}$. The line "E" restores pathNum to the value that it had before the recursive call was made, indicating that the instrumentation process resumes with the path prefix $[p \parallel c \rightarrow r]$, where p is the path taken to c.

8. For each intraprocedural edge $v \rightarrow w$ that is not a backedge, code is inserted so that as the edge is traversed, pathNum is incremented by the value $\rho_{v \rightarrow w}$(numValidCompsFromExit). For example, consider the following if statement (v, w_1, and w_2 are labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
v :if( ... ) {
     w1 : ...
  } else {                                   (30)
     w2 : ...
  }
```

   Let $\rho_{v \rightarrow w_1} = \langle a, b \rangle$ and $\rho_{v \rightarrow w_2} = \langle c, d \rangle$. Then the if statement given in (30) is replaced by

```
v :if( ... ) {
       pathNum += a * numValidCompsFromExit + b;
   w1 : ...
  } else {
       pathNum += c * numValidCompsFromExit + d;
   w2 : ...
  }
```

   Note that one of $\langle a, b \rangle$ and $\langle c, d \rangle$ will be the function $\langle 0, 0 \rangle$; clearly, no code needs to be added for an edge labeled with the function $\langle 0, 0 \rangle$.

9. For each intraprocedural edge $w \rightarrow v$ in procedure P that is a backedge, code is inserted that updates the profile for one unbalanced-left path and then begins recording the path number for a new unbalanced-left path. For a example, consider the following while statement (v and w represent labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
v :while( ... ){
       ...
   w : /* source vertex of backedge */          (31)
  }
```

   In G^*_{fin}, the backedge $w \rightarrow v$ has been replaced by the edges $Entry_P \rightarrow v$ and $w \rightarrow GExit_P$. Let $\rho_{Entry_P \rightarrow v} = \langle a, b \rangle$. In this example, $\rho_{w \rightarrow GExit_P} = \langle 0, 0 \rangle$ (because the surrogate edge $w \rightarrow GExit_P$ is the only edge out of w in G^*_{fin}). The while statement in (31) is replaced by

```
v :while( ... ){
       ...
   w : /* source vertex of backedge */
     /* A: */ profile[pathNum]++;
     /* B: */ pathNum = pathNumOnEntry + a * numValidCompsFromExit + b;
  }
```

   The line labeled "A" updates the profile for the unbalanced-left path of G^*_{fin} that ends with $w \rightarrow GExit_P \rightarrow Exit_{global}$, and the line labeled "B" starts recording a new path number for the unbalanced-left path p that consists of a context-prefix that ends at $Entry_P$ and an active-suffix that begins at v. (The context-prefix is established by the value of pathNum on entry to P, which has been saved in the variable pathNumOnEntry.)

For a second example, consider the following do-while statement (v, w and x represent labels from vertices in G^*_{fin} that correspond to the indicated pieces of code):

```
v :do {
        ...
  w : } while( /* test */ );                    (32)
x :...
```

In G^*_{fin}, the backedge $w \rightarrow v$ has been replaced by the edges $Entry_P \rightarrow v$ and $w \rightarrow GExit_P$. Let $\rho_{Entry_P \rightarrow v} = \langle a, b \rangle$, $\rho_{w \rightarrow GExit_P} = \langle c, d \rangle$, and $\rho_{w \rightarrow x} = \langle e, f \rangle$. Then the do-while in (32) is replaced by

```
v :do {
      ...
   w :if( /* test */ ){
         /* A: */ profile[pathNum + c * numValidCompsFromExit + d]++;
         /* B: */ pathNum = pathNumOnEntry + a * numValidCompsFromExit + b;
         /* C: */ continue;
     } else {
         /* D: */ pathNum += e * numValidCompsFromExit + f;
         break;
     }
  } while( 0 );
x :...
```

The line labeled "A" updates the profile for the unbalanced-left path of G^*_{fin} that ends with $w \rightarrow GExit_P \rightarrow Exit_{global}$. The line labeled "B" starts recording a new path number for the unbalanced-left path p that consists of a context-prefix that ends at $Entry_P$ and an active-suffix that begins at v. The line "D" updates pathNum using the function $\rho_{w \rightarrow x}$. Again, one of $\langle c, d \rangle$ and $\langle e, f \rangle$ will be $\langle 0, 0 \rangle$, and no code to update pathNum need be included for the function $\langle 0, 0 \rangle$.

5.5.1 Optimizing the Instrumentation

The code that calculates path numbers can be made more efficient than the implementation described above. As each non-backedge is traversed, this implementation requires one multiplication and two additions. However, within a given activation of a procedure P, the multiplication is always by the same value of numValidCompsFromExit, and the products of these multiplications are always added to the sum in pathNum. This means that the multiplication may be "factored out." As an example, consider a subpath in P consisting of the non-backedges e_1, e_2, and e_3 that are associated with functions $\langle 2, 2 \rangle$, $\langle 3, 4 \rangle$, and $\langle 5, 3 \rangle$. Let $\texttt{pathNum}_{orig}$ be the value of pathNum before this subpath is executed. After e_1, e_2, and e_3 are traversed, we have the following:

$$\begin{aligned} \texttt{pathNum} &= \texttt{pathNum}_{orig} \\ &+ 2 \cdot \texttt{numValidCompsFromExit} + 2 \\ &+ 3 \cdot \texttt{numValidCompsFromExit} + 4 \\ &+ 5 \cdot \texttt{numValidCompsFromExit} + 3 \\ &= \texttt{pathNum}_{orig} + 10 \cdot \texttt{numValidCompsFormExit} + 9 \end{aligned}$$

Instead of incrementing pathNum as each edge is traversed, two temporaries t_1 and t_2 are introduced. Both are initialized to 0. The temporary t_1 is used to sum the coefficients from the edge functions. The temporary t_2 is used to sum the constant terms of the edge functions. When pathNum absolutely must be updated (*i.e*, before the profile is updated, before a procedure call is made, or before a procedure returns), it is incremented by $t_1 \cdot \texttt{numValidCompsFromExit} + t_2$. Note that when procedure P calls procedure Q, after control returns to P, both t_1 and t_2 should be set to 0; when control returns to P, pathNum will have already been updated for the current values of t_1 and t_2.

The fact that t_1 and t_2 are used to sum values as edges are traversed allows some additional optimizations to be performed. Recall that in the Ball-Larus technique, there is some flexibility in the placement of increment statements; this is used to push the increment statements to infrequently executed edges [5]. In a similar fashion, it is possible to move the statements that increment t_1 and t_2.

5.6 Recovering a Path From a Path Number

This section describes an algorithm that takes a path number pathNum as input and outputs the corresponding path p; this is the inverse operation of computing a path number. As the algorithm traverses the path p, it decrements the value in pathNum. After a path prefix p' has been traversed, pathNum holds the value that is contributed to p's original path number by the path p'', where $p = p' \parallel p''$. At this stage, $edgeValueInContext(p', v \rightarrow w_i)$ is computed for each valid successor $w_i \in \{w_1 \ldots w_k\}$ of p'. (Note that v is the last vertex of p'.) The algorithm is based on the following observations:

- Let w_j denote the valid successor of p' that gives the largest value of $edgeValueInContext(p', v \rightarrow w_j)$ that is less than or equal to pathNum. Given the definition of *edgeValueInContext*, the edge $v \rightarrow w_j$ must be the next edge of p (*i.e.*, the first edge of p'').
- For $x > j$, the value of $edgeValueInContext(p', v \rightarrow w_x)$ is greater than pathNum and $v \rightarrow w_x$ cannot be the first edge of the continuation p''.
- For $x < j$, any valid continuation of p' that starts with $v \rightarrow w_x$ will contribute a value to the path number that is less than $edgeValueInContext(p', v \rightarrow w_j)$ (see Figure 10), which itself is less than or equal to pathNum. Since the continuation p'' makes a contribution equal to pathNum, $v \rightarrow w_x$ cannot be the first edge of p''.

Thus, the next steps of the algorithm are: (i) set p' to $p' \parallel v \rightarrow w_j$; (ii) decrement the value of pathNum by $edgeValueInContext(p', v \rightarrow w_j)$; and (iii) start considering the valid successors of the new p'.

The algorithm uses two stacks: the first is similar to the stack used by Algorithm 5.1 and keeps track of the value numValidCompsFromExit; the second keeps track of the sequence of return vertices that correspond to the current call stack.

Algorithm 5.2 (Calculate Path from Path Number)
Input: The path number pathNum for an unbalanced-left path p in G^*_{fin} from $Entry_{global}$ to $Exit_{global}$.
Output: A listing of the edges of the path p.

initialize stack NVCstack to empty
initialize stack returnStack to empty
initialize numValidCompsFromExit to 1
initialize path p to $[\epsilon : Entry_{global}]$

Find the edge $Entry_{global} \rightarrow Entry_P$ that gives the largest value for

$$x = edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \rightarrow Entry_P)$$

that is less than or equal to pathNum (see Section 5.3).

$p := p \parallel Entry_{global} \rightarrow Entry_P$
pathNum := pathNum $- x$
push(returnStack, $Exit_{global}$)
push(NVCstack, 1)

while p does not end at $Exit_{global}$ **do**
    let v be the last vertex of p
    if v is a call vertex c **then**
        let $c \rightarrow Entry_P$ be the call-edge from c
        $p := [p \parallel c \rightarrow Entry_P]$
        let r be the return vertex associated with c
        push(returnStack, r)
        push(NVCstack, numValidCompsFromExit)
        numValidCompsFromExit := ψ_r(numValidCompsFromExit)
    else if v is an exit vertex $Exit_P$ **then**

```
        numValidCompsFromExit := pop(NVCstack)
        r :=pop(returnStack)
        p := [p || Exit_P → r]
    else if v is an exit vertex GExit_P then
        p := [p || GExit_P → Exit_global]
    else
        let w_1 ... w_k denote the successors of v
        for (i := 1; i ≤ k − 1; i++) do
            if (ρ_{v→w_{i+1}}(numValidCompsFromExit) > numPaths) then
                break
            fi
        od
        p := [p || v → w_i]
        numPaths := numPaths − ρ_{v→w_i}(numValidCompsFromExit)
    fi
od
output p
```

□

6 Interprocedural Piecewise Path Profiling

As mentioned in Section 3.4, the technique described in Section 5 collects an interprocedural *context* path profile. In particular, an unbalanced-left path p through G^*_{fin} may contain a non-empty context-prefix that summarizes the context in which the active-suffix of p occurs. This section shows how to modify the technique of Section 5 to give a technique that collects an interprocedural *piecewise* path profile.

In piecewise path profiling, we use a fixed set of observable paths that partition any execution path, and report how many times each observable path occurs during program execution. An execution path corresponds to a same-level valid path in the original supergraph from $Entry_{global}$ to $Exit_{global}$. Since we are interested in observable paths that may begin and end in the middle of an execution path, we are interested in unbalanced-right-left paths.

Another way to see that we are interested in unbalanced-right-left paths for interprocedural piecewise path-profiling is to compare them with the observable paths used for interprocedural context path-profiling: to modify an observable path p from the context path-profiling technique for use in a piecewise path-profiling technique, we are interested in throwing out the prefix of p that contains the context-prefix, and keeping the suffix of p that has the active-suffix; the suffix of an unbalanced-left (observable) path that is used for context profiling is an unbalanced-right-left path.

For technical reasons, the algorithm for piecewise path profiling uses a slightly different version of G^*_{fin} from that used in Sections 3 and 5. In particular, a new transformation must be performed before the transformations given in Section 3.2:

Transformation 0: For each procedure P add the special vertex $GEntry_P$ to the flow graph for P and an edge $Entry_{global} \rightarrow GEntry_P$.

In addition, Transformations 2 and 3 are modified so that each surrogate edge of the form $Entry_P \rightarrow v$ is replaced by a surrogate edge of the form $GEntry_P \rightarrow v$. For the purpose of classifying a path as a same-level valid path, an unbalanced-left path, an unbalanced-right path, or an unbalanced-right-left path, edges of the form $Entry_{global} \rightarrow Entry_P$ are labeled "$(_P$", edges of the form $Exit_P \rightarrow Exit_{global}$ are labeled "$)_P$", and edges of the form $Entry_{global} \rightarrow GEntry_P$ and $GExit_P \rightarrow Exit_{global}$ are labeled "e". The observable paths correspond to the unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ in the graph G^*_{fin} created by Transformations 0–3.

Example 6.1 Figure 14 shows the graph G^*_{fin} that is constructed for collecting a piecewise profile for the program in Figure 4. In Figure 14, the vertices v_{17} and u_9 are inserted by Transformation 0. The vertices v_{14} and u_6 are inserted by Transformation 1. The edges $v_{17} \rightarrow v_4$ and $v_{13} \rightarrow v_{14}$ are inserted by (the

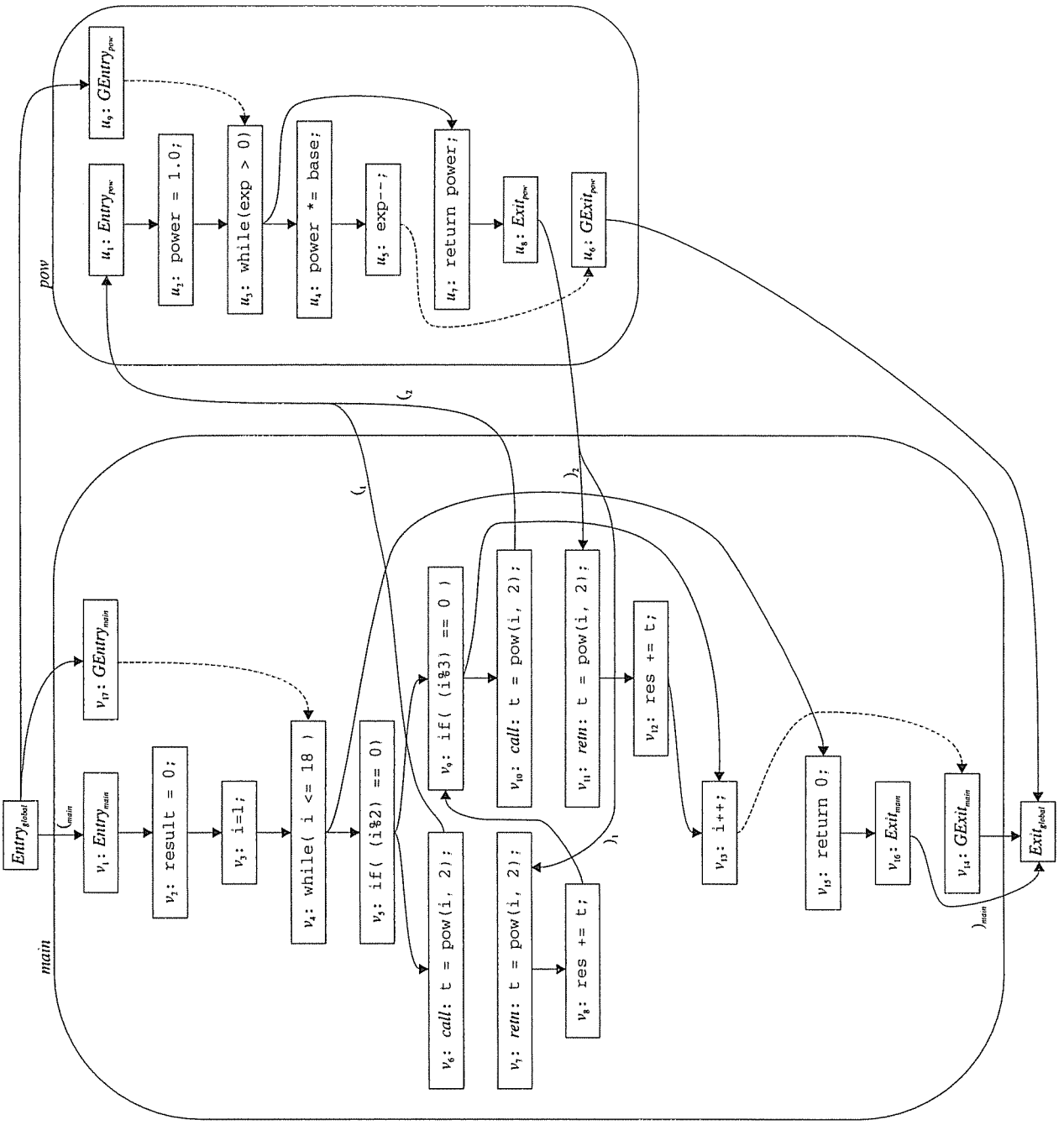


Figure 14: G^*_{fin} for piecewise-profiling instrumentation for the program given in Figure 4. Dashed edges represent surrogate edges; the supergraph for the program in Figure 4 includes the backedges $v_{13} \to v_4$ and $u_5 \to u_3$, which have been removed here by Transformation 2.

modified) Transformation 2 (and the backedge $v_{13} \to v_4$ is removed). Similarly, Transformation 2 adds the edges $u_9 \to u_3$ and $u_5 \to u_6$, and removes the backedge $u_5 \to u_3$. Transformation 3 is not illustrated in this example, because the program is non-recursive. □

As noted above, under this construction of G^*_{fin}, the observable paths correspond to unbalanced-right-left paths through G^*_{fin}. In particular, the observable paths no longer correspond just to the unbalanced-left paths in G^*_{fin}: an unbalanced-right-left path that begins $Entry_{global} \to GEntry_P \to v \to \ldots$ corresponds to an observable path that begins at vertex v (*i.e.*, in the middle of procedure P). For instance, consider the

following path in Figure 14:

$$Entry_{global} \to u_9 \to u_3 \to u_7 \to u_8 \to v_7 \to v_8 \to v_9 \to v_{10} \to u_1 \to u_2 \to u_3 \to u_4 \to u_5 \to u_6 \to Exit_{global}$$

This path corresponds to an execution sequence that begins at vertex u_3 in *pow*, returns to the first call site in *main*, and then re-enters *pow* from the second call site in *main*. The sequence of parentheses generated by this path consists of an unmatched right parenthesis on the return-edge $u_4 \to v_7$ and an unmatched left parenthesis on the call-edge $v_{10} \to u_1$. Thus, the path is an unbalanced-right-left path, but is not an unbalanced-left path nor a same-level valid path.

6.1 Numbering Unbalanced-Right-Left Paths in G^*_{fin}

In this section, we describe how to obtain a dense numbering of the unbalanced-right-left paths in G^*_{fin}. The number of unbalanced-right-left paths in G^*_{fin} is finite. Thus, the graph G^*_{fin} together with the context-free grammar for unbalanced-right-left strings of parentheses constitute a context-free DAG. We will use the technique presented in Section 4 for numbering L-paths to number unbalanced-right-left paths in the modified G^*_{fin}. For this section, the L-paths of Section 4 correspond to unbalanced-right-left paths. The function *numValidComps* takes an unbalanced-right-left path p that starts at $Entry_{global}$ and returns the number of valid completions of p. The definitions of *edgeValueInContext* and path number are exactly as in Section 4.

The task of computing *numValidComps* and *edgeValueInContext* for an unbalanced-right-left path is similar to the task of computing these functions for an unbalanced-left path in Section 5. Let p be an unbalanced-right-left path from $Entry_{global}$ to a vertex v in procedure P. Our technique is based on the following observations (which are essentially the same as those made in Section 5):

1. The number of valid completions of p, *numValidComps*(p), is determined by the sequence of unmatched left parentheses in p and the vertex v. If $v = Exit_P$ and p contains an unmatched left parenthesis, then there is only one valid successor of p: the return vertex r such that $Exit_P \to r$ closes the last open parenthesis in p. Otherwise, any successor of v is a valid successor of p.

2. For a vertex u in P, the value of *numValidComps*$(p \parallel q)$ is the same for *any* same-level valid path q from v to u. In particular, this holds for $u = Exit_P$.

3. If $v = GExit_P$, then the number of completions of p is 1, because $GExit_P \to Exit_{global}$ is the only path out of $GExit_P$.

4. If $w_1, \ldots, w_k$ are the valid successors of p, then the value of *numValidComps*(p) is given by the following sum:
$$numValidComps(p) = \sum_{i=1}^{k} numValidComps(p \parallel v \to w_i).$$

These observations imply that the ψ and ρ functions described in Section 5 are also useful for interprocedural piecewise path profiling. Let q be any same-level valid path from v to $Exit_P$. This means that

$$numValidComps(p) = \psi_v(numValidComps(p \parallel q)).$$

Furthermore, for an intraprocedural edge $v \to w$, we have the following:

$$edgeValueInContext(p, v \to w) = \rho_{v \to w}(numValidComps(p \parallel q)).$$

As we will see in Sections 6.2 and 6.3, this allows us to use the same device we used in Sections 5.4 and 5.5—namely, to maintain a value `numValidCompsFromExit` so that *edgeValueInContext*$(p, v \to w)$ can be computed efficiently, as $\rho_{v \to w}$(`numValidCompsFromExit`.

Even though the same ψ and ρ functions will be used, there are two key differences in how the functions *numValidComps* and *edgeValueInContext* are computed when dealing with unbalanced-right-left paths:

1. The first difference is in how *numValidComps* is computed for an unbalanced-right-left path that ends with an $Exit_P$ vertex.

2. The second difference is in how *edgeValueInContext* is computed for an interprocedural edge.

Both of these differences stem from the fact that an unbalanced-right-left path p from $Entry_{global}$ to $Exit_P$ may have no unmatched left parentheses (*i.e.*, p may be an unbalanced-right path). In contrast, in Section 5, when dealing with unbalanced-left paths, we never "ran out" of unmatched left parentheses.

6.1.1 Calculating *numValidComps* from $Exit_P$

Let q be an unbalanced-right-left path from $Entry_{global}$ to $Exit_P$. In this section, we discuss how to compute $numValidComps(q)$. If q contains an unmatched left parenthesis, then there is only one valid successor of q: the return vertex r_1 such that the edge $Exit_P \to r_1$ matches the last unmatched left parenthesis of q. This gives us the following:

$$numValidComps(q) = numValidComps(q \parallel Exit_P \to r_1).$$

Suppose r_1 occurs in procedure Q. Then the above value is equal to

$$\psi_{r_1}(numValidComps(q \parallel Exit_P \to r_1 \parallel q')),$$

where q' is any same-level valid path from r_1 to $Exit_Q$. Recall that the function ψ_{r_1} counts the valid completions of p that exit Q via $GExit_Q$, even though it only takes as an argument the number of valid completions for paths that exit Q via $Exit_Q$. As before, if $[q \parallel Exit_P \to r_1 \parallel q']$ has an unmatched left parenthesis, then it will have only one valid successor: the return vertex r_2 such that $Exit_Q \to r_2$ is labeled with the parenthesis that closes the second-to-last unmatched left parenthesis in q. Suppose that r_2 is in procedure R. Then the above value becomes

$$\psi_{r_1}(\psi_{r_2}(numValidComps(q \parallel Exit_P \to r_1 \parallel q' \parallel Exit_Q \to r_2 \parallel q''))),$$

where q'' is any same-level valid path from r_2 to $Exit_R$. Again, ψ_{r_2} counts valid completions that leave R via either $GExit_R$ or $Exit_R$. This argument can be continued until a path s has been constructed from $Entry_{global}$ to $Exit_S$, and one of two cases holds:

Case 1: The parenthesis "$(_S$" that appears on the edge $Entry_{global} \to Entry_S$ is the only unmatched left parenthesis in s. In this case, the only valid successor of s is $Exit_{global}$, and the number of valid completions of s is 1. This means that

$$\begin{aligned} numValidComps(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(numValidComps(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(1)\ldots)) \end{aligned}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched left parentheses in q. (Note that this equation is the same as Equation (9), which makes sense because q must have been an unbalanced-left path.)

Case 2: There are no unmatched left parentheses in s. (In this case, q does not contain the parenthesis "$(_S$"; note that this case cannot happen for a path q in Section 5, because each unbalanced-left path q starts with a parenthesis of the form "$(_S$".) In this case, s is an unbalanced-right path (which has the unbalanced-right-left path q as a prefix), and so every return vertex that is a successor of $Exit_S$ is a valid successor of s. Furthermore, any unbalanced-right-left path from $Exit_S$ to $Exit_{global}$ is a valid completion of the path s. (In contrast, consider an unbalanced-left path p that is used for interprocedural context path profiling: each valid completion of p is an unbalanced-right-left path; however, only an unbalanced-right-left path p' where the sequence of unmatched right parentheses in p' match the sequence of unmatched left parentheses in p can be a valid completion of p.)

We define the value $numUnbalRLPaths[v]$ to be the total number of unbalanced-right-left paths from the vertex v to $Exit_{global}$. This gives us the following:

$$\begin{aligned} numValidComps(q) &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(numValidComps(s))\ldots)) \\ &= \psi_{r_1}(\psi_{r_2}(\ldots\psi_{r_k}(numUnbalRLPaths[Exit_S])\ldots)) \end{aligned}$$

where $r_1 \ldots r_k$ is the sequence of return vertices determined by the unmatched left parentheses in q. (Note that if there are no unmatched left parentheses in q, then $Exit_S = Exit_P$ and the above equation simplifies to $numValidComps(q) = numUnbalRLPaths[Exit_P]$.)

We now show how to compute $numUnbalRLPaths[v]$. First, note that if $numUnbalRLPaths[Exit_P]$ is known, then the ψ functions can be used to compute $numUnbalRLPaths[v]$ for any vertex v in procedure P:

$$numUnbalRLPaths[v] = \psi_v(numUnbalRLPaths[Exit_P])$$

This follows from the definition of ψ_v: let p be any unbalanced-right-left path from $Entry_{global}$ to v such that p does not contain any unmatched left parentheses and let q be a same-level valid path from v to $Exit_P$. The number of valid completions of p is equal to the number of unbalanced-right-left paths from v to $Exit_{global}$. This implies that

$$\begin{aligned} numUnbalRLPaths[v] &= numValidComps(p) \\ &= \psi_v(numValidComps(p \parallel q)) \\ &= \psi_v(numUnbalRLPaths[Exit_P]) \end{aligned}$$

The value of $numUnbalRLPaths[Exit_P]$ is given by the following equation:

$$numUnbalRLPaths[Exit_P] = \sum_{r \in succ(Exit_P)} numUnbalRLPaths[r] \tag{33}$$

where $numUnbalRLPaths[r]$ is given by

$$numUnbalRLPaths[r] = \begin{cases} 1 & \text{if } r = Exit_{global} \\ \psi_r(numUnbalRLPaths[Exit_Q]) & \text{if } r \text{ is a return-site vertex in procedure } Q \end{cases} \tag{34}$$

Equations (33) and (34) can be used to compute $numUnbalRLPaths[Exit_P]$ for each $Exit_P$ vertex and $numUnbalRLPaths[r]$ for each return-site vertex r during a traversal of the call graph associated with G^*_{fin} in topological order: Because the call graph associated with G^*_{fin} is acyclic, whenever a vertex $Exit_P$ is reached, each value $numUnbalRLPaths[r]$ that is needed to compute $numUnbalRLPaths[Exit_P]$ will be available.

6.1.2 Computing *edgeValueInContext* for interprocedural edges

For an unbalanced-right-left path p from $Entry_{global}$ to $Exit_P$, and an edge $Exit_P \to r$, the value of $edgeValueInContext(p, Exit_P \to r)$ is not always zero, as it was in Section 5.3. Let $r_1 \ldots r_k$ be the successors of $Exit_P$. If the path p does contain an unmatched left parenthesis, then there is only a single r_i that is a valid successor of p. This means that

$$edgeValueInContext(p, Exit_P \to r_i) = 0.$$

Now suppose that p has no unmatched left parentheses (*i.e.*, p is an unbalanced-right path). In this case, every r_i is a valid successor of p. Then the definition of *edgeValueInContext* in Equation (6) yields the following:

$$edgeValueInContext(p, Exit_P \to r_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} numUnbalRLPaths[r_j] & \text{otherwise} \end{cases} \tag{35}$$

Notice that the value computed by Equation (35) is the same for any unbalanced-right-left path p to $Exit_P$ that has no unmatched left parentheses (*i.e.*, for any unbalanced-right path). For an edge $Exit_P \to r_i$, we define $edgeValue[Exit_P \to r_i]$ to be the value computed by the right-hand side of Equation (35) (for any unbalanced-right-left path p to $Exit_P$ that has no unmatched left parentheses).

As in Section 5.3, we must also calculate *edgeValueInContext* for the path $[\epsilon : Entry_{global}]$ and an edge $Entry_{global} \to v$. We observe that

$$numValidComps(Entry_{global} \to Entry_P) = \psi_{Entry_P}(1)$$

and

$$numValidComps(Entry_{global} \to GEntry_P) = \psi_{GEntry_P}(numUnbalRLPaths[Exit_P]).$$

The first of these equations is discussed in Section 5.3. The second holds because the path consisting of the edge $Entry_{global} \to GEntry_P$ is an unbalanced-right path. With these observations, it is possible to apply

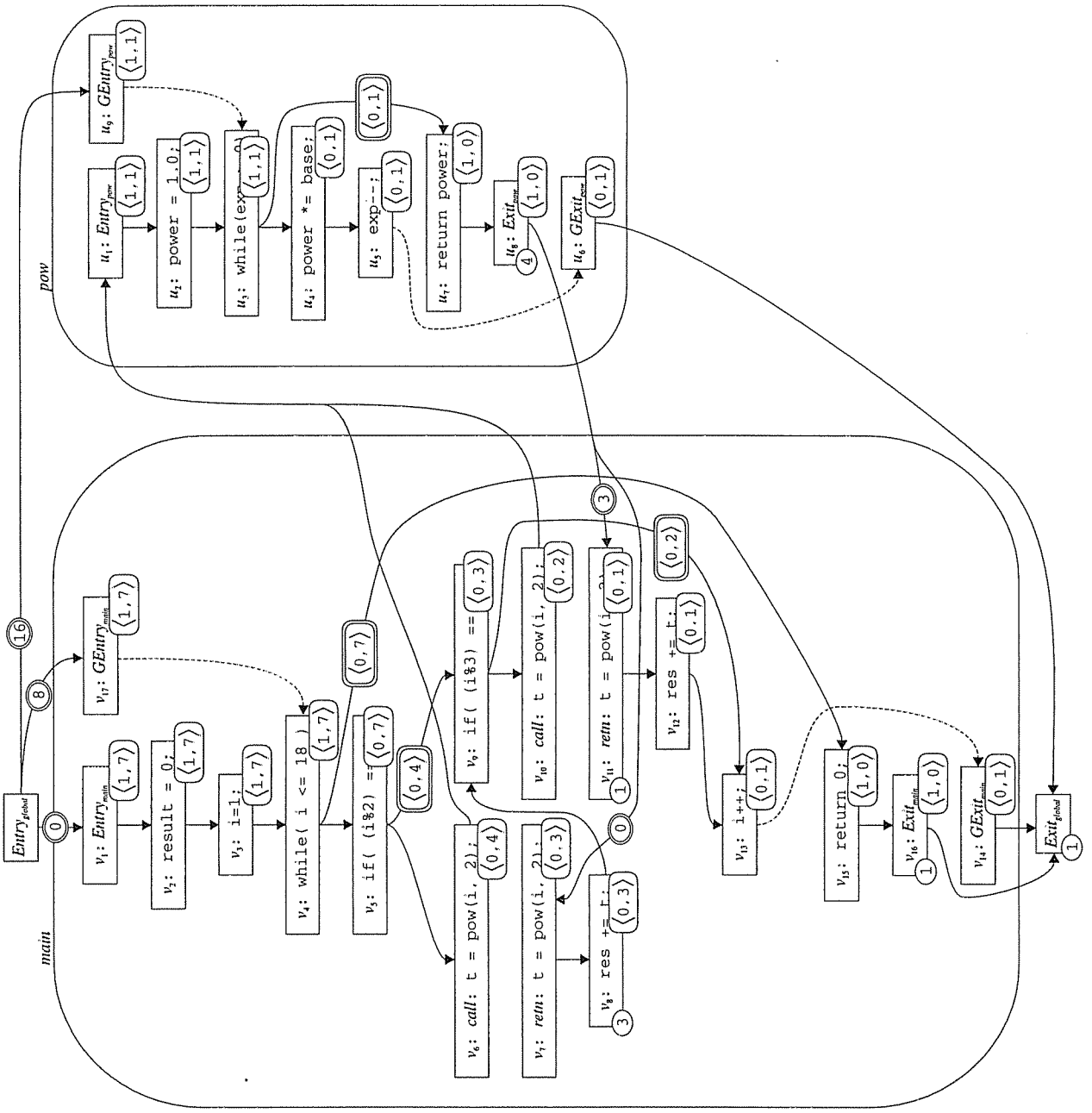


Figure 15: Labeled version of G^*_{fin} from Figure 14. For each intraprocedural vertex v, the function ψ_v is shown in a rounded box. For each intraprocedural edge e, the function ρ_e is shown in a doubled, rounded box; intraprocedural edges that are not labeled have the function $\langle 0,0\rangle$. For the exit vertices $Exit_{global}$, $Exit_{main}$, and $Exit_{pow}$, and the return-site vertices v_7 and v_{11}, the value of *numUnbalRLPaths* is shown in a circle. For each interprocedural edge $v \to w$, the value of $edgeValueInContext(p, v \to w)$ for an unbalanced-right path p ending at v is shown in a doubled rounded circle. There are a total of $\psi_{Entry_{main}}(1) + \psi_{GEntry_{main}}(1) + \psi_{GEntry_{pow}}(4) = 8 + 8 + 5 = 21$ unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ in G^*_{fin}.

Equation (6) to find the appropriate values of $edgeValueInContext([\epsilon : Entry_{global}], Entry_{global} \to v)$ for the $Entry_{global} \to Entry_P$ and $Entry_{global} \to GEntry_P$ edges.

Figure 15 shows a labeled version of G^*_{fin} from Figure 14. Figure 15 shows the values on the interprocedural edges that are calculated as discussed in this section. In addition, *numUnbalRLPaths* is shown for

certain vertices.

In general, the total number of unbalanced-right-left paths through G^*_{fin} is given by

$$numValidComps([\epsilon : Entry_{global}]) = \sum_{v \in succ(Entry_{global})} numValidComps(Entry_{global} \rightarrow v)$$

where

$$numValidComps(Entry_{global} \rightarrow Entry_P) = \psi_{Entry_P}(1)$$

and

$$numValidComps(Entry_{global} \rightarrow GEntry_P) = \psi_{GEntry_P}(numUnbalRLPaths[GExit_P]).$$

For the graph G^*_{fin} shown in Figure 15, the total number of unbalanced-right-left paths from $Entry_{global}$ to $Exit_{global}$ is

$$\begin{aligned} numValidComps([\epsilon : Entry_{global}]) &= numValidComps(Entry_{global} \rightarrow Entry_{main}) + \\ &\quad numValidComps(Entry_{global} \rightarrow GEntry_{main}) + \\ &\quad numValidComps(Entry_{global} \rightarrow GEntry_{pow}) \\ &= \psi_{Entry_{main}}(1) + \psi_{GEntry_{main}}(1) + \psi_{GEntry_{pow}}(4) \\ &= 8 + 8 + 5 \\ &= 21 \end{aligned}$$

6.2 Calculating the Path Number of an Unbalanced-Right-Left Path

We are now ready to give the algorithm for computing the path number of an unbalanced-right-left path p. This algorithm is very similar to the algorithm given in Section 5.4 for calculating the path number of an unbalanced-left path. One additional program variable, `cntOpenLfParens`, is used. This variable is used to keep track of the number of open left parentheses in the prefix p' of p that has been traversed. If `cntOpenLfParens` is zero (indicating that p' is an unbalanced-right path) and the algorithm traverses a return-edge e, then `pathNum` may be incremented by a non-zero value (see Section 6.1.2). If `cntOpenLfParens` is non-zero and the algorithm traverses a return-edge e, then `pathNum` is not incremented (which represents an increment by the value 0).

Algorithm 6.1 (Calculate Path Number for an Unbalanced-right-left Path)
Input: An unbalanced-right-left path p from $Entry_{global}$ to $Exit_{global}$.
Output: p's path number.

Initialize stack `NVCstack` to empty

Let e be the first edge of the path p. Calculate the value of $edgeValueInContext([\epsilon : Entry_{global}], e)$ as described in Section 6.1.2. Set `pathNum` to this value.

if e is of the form $Entry_{global} \rightarrow Entry_P$ **then**
    `numValidCompsFromExit` := 1
    `cntOpenLfParens` := 1
else /* e is of the form $Entry_{global} \rightarrow GEntry_P$ */
    `numValidCompsFromExit` := $numUnbalRLPaths[Exit_P]$
    `cntOpenLfParens` := 0
fi

set e to be the second edge of p
while e is not of the form $v \rightarrow Exit_{global}$ **do**
    if e is of the form $c \dashrightarrow Entry_T$ **then**
        push `numValidCompsFromExit` on `NVCstack`
        let r be the return vertex associated with c
        `numValidCompsFromExit` := ψ_r(`numValidCompsFromExit`)

```
        cntOpenLfParens++
    else if e is of the form Exit_T → r then
        if cntOpenLfParens == 0 then
            pathNum += edgeValue[e]
            let S be the procedure that contains r
            numValidCompsFromExit := ψ_r(numUnbalRLPaths[Exit_S])
        else
            numValidCompsFromExit := pop(NVCstack)
            cntOpenLfParens--
        fi
    else
        pathNum := pathNum + ρ_e(numValidCompsFromExit)
    fi
    set e to the next edge of p
od
output pathNum
```

□

6.3 Runtime Environment for Collecting a Profile

As in Section 5.5, the instrumentation code for collecting an interprocedural piecewise path profile essentially threads Algorithm 6.1 into the code of the instrumented program. The instrumentation code for collecting an interprocedural piecewise path profile differs from the instrumentation code described in Section 5.5 in the following ways:

- there is no variable `pathNumOnEntry`;
- there is a new parameter `cntOpenLfParens` that is passed to every procedure except *main*, which has `cntOpenLfParens` as a local variable; and
- both `pathNum` and `cntOpenLfParens` are saved before a recursive call is made and restored after a recursive call returns.

Figures 16 and 17 show the program from Figure 4 with additional instrumentation code to collect an interprocedural piecewise path profile. The output from the instrumented code is as follows:

```
 0: 0    1: 0    2: 0    3: 0    4: 0    5: 0    6: 1    7: 0    8: 9
 9: 0   10: 0   11: 0   12: 3   13: 0   14: 5   15: 1   16:15   17: 3
18: 0   19: 6   20: 6
```

(The algorithm for decoding a path number to obtain the corresponding unbalanced-right-left path is left as an exercise for the reader.)

7 Intraprocedural Context Path Profiling

This section describes how to modify the Ball-Larus path-profiling technique to collect an intraprocedural context profile. In Section 5.5, each observable path is divided into a context-prefix and an active-suffix. When these definitions are adapted to the observable paths in the Ball-Larus (intraprocedural) technique, each observable path has an empty context-prefix. We now show how to modify the Ball-Larus technique so that an observable path may have a non-empty context-prefix. Under this new technique, a typical observable path will consist of a context-prefix that summarizes the path taken to a loop header and an active suffix that is a path through the loop.

The new technique gives more detailed profiling information than the Ball-Larus path-profiling technique. For example, suppose there is a correlation between the path taken to a loop header and the path(s) taken during execution of the loop body. Intraprocedural context profiling will capture the relationship between

```
unsigned int profile[21];        /* 21 possible paths in total */

double pow(double base, long exp,
           unsigned int &pathNum, unsigned int numValidCompsFromExit,
           unsigned int &cntOpenLfParen) {
   double power = 1.0;

   while( exp > 0 ) {
      power *= base;
      exp--;
      profile[pathNum]++;

      /* Start a new path with the edges entry_global->u9 and u9->u1 */
      cntOpenLfParen = 0;
      numValidCompsFromExit = 4; /* from numUnbalRLPaths(u8) */
      pathNum = 16;              /* from the edge entry_global->u9 */
                                 /* no additional code is needed for the */
                                 /*   function <0,0> on edge u9->u3 */
   }

   pathNum += 0 * numValidCompsFromExit + 1; /* from edge u3->u7 */

   return power;
}
```

Figure 16: Part of the instrumented version of the program that computes $(\sum_{j=1}^{9}(2 \cdot j)^2) + (\sum_{k=1}^{6}(3 \cdot k)^2)$. The original program is shown in Figure 4; the instrumentation collects an interprocedural piecewise profile. The instrumented version of *main* is shown in Figure 17. Instrumentation code is shown in italics.

the path taken to the loop header and the paths taken on each iteration of the loop body. The Ball-Larus (piecewise) profiling technique will only capture the correspondence between the path taken to the loop header and the path taken on the first iteration of the loop; for all subsequent iterations, the Ball-Larus technique records an observable path that begins at the loop header, and ignores the context information provided by the path used to reach the loop header.

Just as in the Ball-Larus technique, modifications are made to the procedure's control-flow graph. Unlike the Ball-Larus technique, we require that the control-flow graph be reducible.[8] There are two transformations:

Transformation 1 (split backedge targets): Each vertex v that is a backedge target is split into two vertices v_a and v_b. All edges into v are changed to point to vertex v_a. All edges that have v as a source vertex are changed to have v_b as the source. An edge $v_a \rightarrow v_b$ is added to the graph; this edge is *not* considered to be a surrogate edge in the following discussion. Figures 18(a) and 18(b) illustrate this transformation.

Transformation 2 (replace backedges): For each backedge target v_a, a second edge $v_a \rightarrow v_b$ is added to the graph; this edge is considered to be a surrogate edge. (Thus, for each pair of vertices v_a and v_b introduced by Transformation 1, there are two edges of the form $v_a \rightarrow v_b$, one of which is considered to be a surrogate edge, and one that is not.) For each backedge source w, the surrogate edge $w \rightarrow Exit$ is added to the graph. Each backedge $w \rightarrow v_a$ is removed from the graph. Figure 18(c) illustrates this transformation.

The graph that results from performing these transformations is acyclic. Once the graph has been modified,

[8] If the control-flow graph is not reducible, then the graph can be transformed to make it reducible (see, for example, Aho et al. [1]).

```
int main() {
   unsigned int pathNum = 0;
   unsigned int numValidCompsFromExit = 1;
   unsigned int cntOpenLfParen = 1;

   double t, result = 0.0;
   int i = 1;

   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         cntOpenLfParen++;
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 3, cntOpenLfParen );
         /* On entry to pow: pathNum is 0 or 8; 4th arg. always 3 */
         /* On exit from pow: pathNum is 1, 9, or 17 */
         if( 0 == cntOpenLfParen ) {
            numValidCompsFromExit = 1;     /* from vertex v16 */
         } else
            cntOpenLfParen--;
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 4;  /* from edge v5->v9 */
      if( (i%3) == 0 ) {
         cntOpenLfParen++;
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 1, cntOpenLfParen );
         /* On entry to pow: pathNum is 1, 4, 9, 12 or 17; 4th arg. always 1 */
         /* On exit from pow: pathNum is 2, 5, 10, 13, 18, or 20 */
         if(0 == cntOpenLfParen) {
            pathNum += 3;              /* from edge u8->v11 */
            numValidCompsFromExit = 1; /* from vertex v16 */
         } else
            cntOpenLfParen--;
         result += t;
      } else
          pathNum += 0 * numValidCompsFromExit + 2;     /* from edge v9->v13 */
      i++;

      profile[pathNum]++;
      /* Start a new path with edges global_entry->v17 and v17->v4 */
      cntOpenLfParen = 0;
      numValidCompsFromExit = 1; /* from vertex v16 */
      pathNum = 8;               /* from edge entry_global->v17 */
   }
   pathNum += 0 * numValidCompsFromExit + 7;  /* from edge v4->v15 */
   profile[pathNum]++;
   for (i = 0; i < 21; i++) {
      cout.width(3); cout << i << ":";
      cout.width(2); cout << profile[i] << "  ";
      if ((i+1) % 9 == 0) cout << endl;
   }
   cout << endl;

   return 0;
}
```

Figure 17: Part of an instrumented version of the program that computes $(\sum_{j=1}^{9}(2 \cdot j)^2) + (\sum_{k=1}^{6}(3 \cdot k)^2)$. The original program is shown in Figure 4; the instrumentation collects an interprocedural piecewise profile. The instrumented version of the function *pow* and the global declaration of *profile* is shown in Figure 16. Instrumentation code is shown in italics.

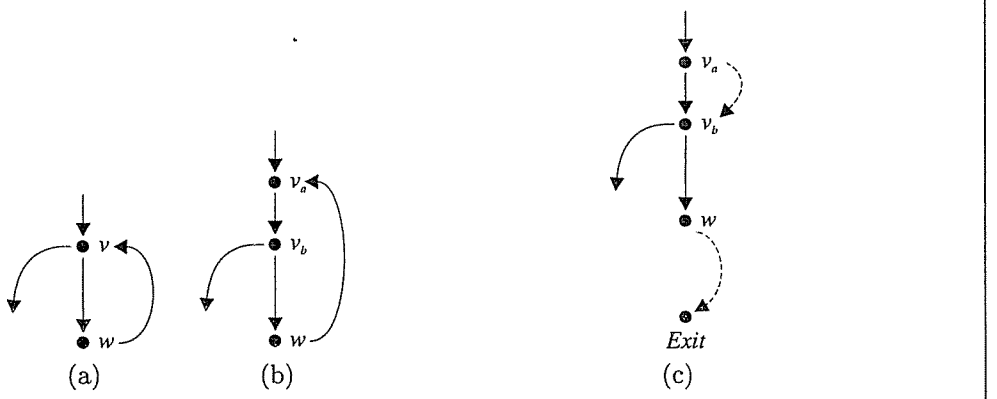


Figure 18: Illustration of Transformations 1 and 2 from Section 7. Figure (a) shows a piece of the control-flow graph for a loop before the transformation, and Figure (b) shows the same piece of the control-flow graph after the transformation. Figure (c) shows the two surrogate edges that replace the backedge $w \rightarrow v_a$. Note the difference from the Ball-Larus technique: Section 7 uses the surrogate edge $v_a \rightarrow v_b$ instead of the surrogate edge *Entry* $\rightarrow v$.

the Ball-Larus edge-numbering scheme is used as before. As in the Ball-Larus technique, the path number for a path p from *Entry* to *Exit* is the sum of the values on p's edges.

We are now ready to describe the instrumentation that is used to collect a profile. As in the Ball-Larus technique, an integer variable `pathNum` is introduced that is used to accumulate the path number of the currently executing path. At the beginning of the procedure, `pathNum` is initialized to 0.

Let v be a backedge target in the original control-flow graph, and let v_a and v_b be the vertices that represent v in the modified control-flow graph (after Transformation 1). A new integer variable called `pathNumOnEntryToV` is introduced. When control reaches v, `pathNumOnEntryToV` is set to the current value in `pathNum`. `pathNum` is then incremented by the value on the non-surrogate edge $v_a \rightarrow v_b$ in the modified graph. When the backedge $w \rightarrow v$ is traversed, the following steps are taken:

1. `pathNum` is incremented by the value on the surrogate edge $w \rightarrow$ *Exit*. The profile is updated with this value of `pathNum`.

2. `pathNum` is set to `pathNumOnEntryToV`, plus the value on the surrogate edge $v_a \rightarrow v_b$.

The second step starts recording the path number for a new path p. The path p contains the edge $v_a \rightarrow v_b$ and will have a context-prefix that ends at v_a and an active-suffix that begins at v_b. Note that this instrumentation relies on the fact that the original control-flow graph is reducible. In particular, it assumes that the backedge target v is reached—and the value of `pathNumOnEntryToV` is set—before the backedge $w \rightarrow v$ is traversed.

The remaining instrumentation is similar to the instrumentation used in the standard Ball-Larus technique. In particular, as each edge e is traversed, the value in `pathNum` is incremented by the value on e.

8 Hybrid Inter/Intra-procedural Approaches

This section describes some additional variations on our interprocedural path-profiling techniques, based on splitting some of the interprocedural paths by removing certain call and return edges. In other words, we consider the effect of applying Transformation 3 of Section 3.2 to *nonrecursive* call sites, with the appropriate modifications in instrumentation. In one sense, this will have the effect of making an interprocedural technique become closer to an intraprocedural technique because interprocedural paths are split; in the degenerate case when Transformation 3 is applied to all call sites, the result is an intraprocedural profiling technique. However, when Transformation 3 is applied to only some call sites, the resulting technique is a

hybrid of the interprocedural and intraprocedural techniques. In particular, the set of observable paths still includes paths that cross procedure boundaries.

One effect of this approach is illustrated in the following example, which shows how the paths in a leaf procedure P can (in effect) contribute to an *intra*procedural profile when called from one call site, and contribute to an *inter*procedural profile when called from another call site. The example discussed below is a modification of the interprocedural context profiling technique of Section 5. (The corresponding modification of the interprocedural piecewise profiling technique from Section 6 would be similar.)

Example 8.1 Let P be a leaf procedure (*i.e.*, P does not contain any procedure calls). Let the call vertex c_1 and the return-site vertex r_1 represent one call site on procedure P, and let c_2 and r_2 represent a second call site on P. Now consider the effect of applying Transformation 3 to the first call site and treating the second call site as nonrecursive call sites are normally treated. For the first call site, the edges $c_1 \rightarrow Entry_P$ and $Exit_P \rightarrow r_1$ are removed and the appropriate surrogate edges are added, including the edges $Entry_{global} \rightarrow Entry_P$ and $Exit_P \rightarrow Exit_{global}$. The instrumentation at the first call site will reflect the effects of this transformation: when P is called from the first call site, the instrumentation begins recording a new path that begins with the surrogate edge $Entry_{global} \rightarrow Entry_P$; when control returns from P to the first call site, the instrumentation records a path that ends with the edge $Exit_P \rightarrow Exit_{global}$.

Note that since P is a leaf procedure, each unbalanced-left path $[Entry_{global} \rightarrow Entry_P \parallel p]$ through G^*_{fin} corresponds to an intraprocedural path through P. Furthermore, each unbalanced-left path $[Entry_{global} \rightarrow Entry_P \parallel p]$ has a path number in the range

$$[x..(x + \psi_{Entry_P}(1) - 1)] \tag{36}$$

where x is the value on the edge $Entry_{global} \rightarrow Entry_P$. These observations show that when P is called from the first call site, an intraprocedural profile is, in essence, recorded for P. Specifically, the path-counts for the paths in the range in (36) give an intraprocedural profile for P. In contrast, when P is called from the second call site, the paths in P are profiled as interprocedural paths (with a context-prefix consisting of some path to the second call site). □

There many different strategies that can be devised to control when to apply Transformation 3 to a call site. The following is a partial list:

1. One possibility is to only profile "interprocedurally" for calls to procedures that have file scope (*e.g.*, static functions in C). Transformation 3 is applied to call sites that represent calls to procedures that do not have file scope.

2. A second possibility is to only profile interprocedurally for calls to procedures within the same file. Transformation 3 is applied to call sites representing calls to procedures outside of the current file.

3. A third possibility in to only profile interprocedurally for calls to leaf procedures.

As noted below (see Section 9.1), profiling may become impractical if there are too many observable paths. The effect of applying Transformation 3 will usually be to decrease the total number of observable paths. This means that using any one of strategies 1–3 listed above may yield a more practical technique than the pure strategies described in Sections 5 and 6. In addition, strategies 1 and 2 may simplify the task of instrumenting code. In particular, a utility that takes a source program $\mathcal{P}$ and outputs the instrumented version of $\mathcal{P}$ can determine the ψ and ρ functions on a file-by-file basis; otherwise, all files must be considered simultaneously.

9 Discussion

9.1 Keeping the Numbering Dense–Saving Bits

In all of the path-profiling techniques discussed in this paper, it is possible for the number of observable paths to be exponential in the size of the graph that is used for numbering paths (see Figure 19). Thus, for the interprocedural path-profiling techniques, it is possible for the number of observable paths to be exponential

Figure 19: Schematic of a graph where the number of paths from $Entry_{global}$ to $Exit_{global}$ is exponential in the number of nodes in the graph.

Figure 20: Schematic of a graph where the number of paths from $Entry_{global}$ to $Exit_{global}$ is linear in the number of nodes in the graph.

in the size of G^*_{fin}, and hence in the size of the program that is being profiled. This can be problematic because it means that the number of bits needed to represent a path number can be proportional to the size of the program.[9] Clearly, if the number of bits needed to represent an observable path is larger than the number of bits in the architectural word of the machine being used, then the profiling technique in question will be expensive, and—if the speed of the instrumented code is an issue—possibly infeasible, because the arithmetic operations that the instrumentation code performs as each edge is traversed become more expensive.

These observations served as motivation for the work in this paper. Section 5 gives a technique for densely numbering the unbalanced-left paths in G^*_{fin}. Likewise, Section 6 presents a technique for densely numbering the unbalanced-right-left paths in G^*_{fin}.

In a technique that does not use a dense numbering, even more bits are needed to represent a path number, and care must be taken that the problem described above is not exacerbated. However, it is not always the case that a non-dense numbering scheme leads to less efficient profiling. In fact, it may be that the profiling techniques described in this paper can be modified to use a non-dense numbering scheme and to operate (on average) on fewer bits during each edge traversal. Further research is need to explore this idea.

Clearly, it is also advantageous to limit the number of observable paths. This means that the interprocedural technique in Section 6 is likely to be more feasible than the interprocedural technique given in Section 5. The technique in Section 6 (which creates graphs that are more similar to Figure 20) has strictly fewer observable paths that the technique in Section 5 (which creates graphs that are more similar to Figure 19). Furthermore, the need to reduce the number of observable paths may make it desirable to use a hybrid technique, such as those discussed in Section 8.

Finally, the experimental results of the Ball-Larus path-profiling technique suggest that there are, in general, too many observable paths to create an array that has one entry for each observable path because of memory restrictions. Rather, it is necessary to use a more complicated data structure (*e.g.*, a hash table), and only store the path counters for those paths that are actually executed. Fortunately, the number of observable paths that are actually executed tends to be a small fraction of the set of all executable paths, and use of an appropriate data structure avoids memory difficulties. It remains to be seen whether this is still the case for our interprocedural path-profiling techniques.

[9] $\log_2 x$ bits are needed to represent a path number, where x is the number of observable paths. If there are 2^n observable paths, where n is the number of vertices in a program's supergraph, then n bits are needed to represent an observable path.

9.2 Handling other Language Features

In this section, we describe how to handle some additional language features that were not explicitly addressed in Section 5. Specifically, Section 9.2.1 discusses some of the complications that arise because of signals and signal handlers. Section 9.2.2 describes how exceptions can be handled, and Section 9.2.3 describes how to take care of indirect function calls.

9.2.1 Signals

Program signals can cause a problem for path profiling because of their asynchronous nature. For example, it is possible for a signal handler to be invoked while the program is in the middle of executing an observable path p. Because it is possible that the signal handler will never return, it is possible that the program will never complete execution of the (hypothetical) path p, and hence p will not be recorded. Furthermore, it is possible that there are paths, called *pending* paths, that are "on hold" at a recursive call site that will only be completed once control returns to the call site. Thus, the current and pending path prefixes that are active at the time of the signal will not be recorded in the profile. This is a problem if the purpose of gathering a path profile is to aid in debugging. Instead, we want to record the *prefixes* of p and the pending paths that have executed at the time the signal occurs.

For either of the interprocedural techniques described in Sections 5 and 6, this can be accomplished by making the following modifications:

- For each procedure P, for each vertex v of P that is not a call vertex and is not $Exit_P$, add a surrogate edge $v \rightarrow GExit_P$.
- For each new surrogate edge $v \rightarrow GExit_P$, the assignment of ρ functions is done such that $\rho_{v \rightarrow GExit_P} = \langle 0, 0 \rangle$. This guarantees that whenever execution reaches the vertex v, the value in `pathNum` is the path number for the current path to v concatenated with $[v \rightarrow GExit_P \rightarrow Exit_{global}]$.
- Add a global stack of `unsigned longs` called `pendingPaths`. For each recursive call site, modify the instrumentation to push the current value of `pathNum` on `pendingPaths` before the call is made and to pop `pendingPaths` after the call returns.
- For every possible signal, a signal handler is written (or modified) in which the first action of the signal handler is to update the path profile with the current value of `pathNum` and with every value that appears on the stack `pendingPaths`. Thus, if a signal s interrupts execution of a procedure P at a vertex v, the signal handler for s will record a path that ends with $v \rightarrow GExit_P \rightarrow Exit_{global}$ and, for every pending path prefix on `pendingPaths` that ends at the call vertex c, a path that ends with $c \rightarrow GExit_P \rightarrow Exit_{global}$.

These modifications guarantee that signals will not cause a loss of information in the profile, which, as mentioned above, is important for some profiling applications. Unfortunately, they also increase the number of observable paths, which creates its own problems (see Section 9.1). Note that the technique for handling signals in the intraprocedural case is similar.

It is possible to avoid the issue of pending paths by changing Transformation 3 in the construction of G^*_{fin} (see the construction for context profiling in Section 3.2). In particular, for a recursive call site represented by vertices c and r, rather than adding the summary edge $c \rightarrow r$, the surrogate edges $c \rightarrow GExit_P$ and $Entry_P \rightarrow r$ are added.[10] In this way, every observable path that contains the summary edge $c \rightarrow r$ is split into two observable paths. The instrumentation code at a recursive call site records the path that ends with $c \rightarrow GExit_P \rightarrow Exit_{global}$ before making the recursive call, and begins recording a new path when the recursive call returns. When this version of Transformation 3 is used, there are never any pending paths during runtime.

9.2.2 Exceptions

Programming languages that have exceptions (*e.g.*, C++, Java) can cause complications, particularly for intraprocedural path-profiling techniques. In particular, consider a procedure P that calls procedure Q,

[10] For piecewise profiling, the edge $Entry_P \rightarrow r$ becomes $GEntry_P \rightarrow r$.

which in turn calls procedure R. If R throws an exception that is caught in P, then there will be an incomplete path in Q that is not recorded. One way to address this is to break every observable path at each call site.

In an interprocedural path-profiling technique, exceptions can be handled by adding a surrogate edge from the $Entry_P$ vertex (or $GEntry_P$ vertex, depending on the interprocedural technique being used) to the vertex v where the exception is caught, and adding a surrogate edge from the vertex u where the exception is thrown to $GExit_R$. (Here P is the procedure containing v, and R is the procedure containing u.) When the exception is thrown, the profiling instrumentation updates the profile for the path that ends with $[u \rightarrow GExit_R \rightarrow Exit_{global}]$. When the exception is caught, the profiling instrumentation uses the edge $Entry_P \rightarrow v$ (or the edge $GEntry_P \rightarrow v$) to begin recording a new observable path.

Note that in the interprocedural path-profiling techniques we must also deal with the issue of pending path prefixes. This can be handled in the same way it is handled in Section 9.2.1: a surrogate edge $c \rightarrow GExit_P$ is added to each recursive call vertex c in each procedure P, the assignment of ρ functions is done such that the $\rho_{c \rightarrow GExit_P}$ functions are all $\langle 0, 0 \rangle$, and a stack of pending path prefixes is maintained. When an exception is thrown, the profile is updated for each value on the stack of pending path prefixes. A stack of pending path prefixes can also be used to handle exceptions for intraprocedural path profiling techniques (where pending paths may include paths that are "on hold" at call sites).

9.2.3 Indirect Procedure Calls

The easiest way to handle indirect procedure calls is to treat them as recursive procedure calls, and not allow interprocedural paths that cross through an indirect procedure call. Another possibility is to turn each indirect procedure call through a procedure variable `fp` into an if-then-else chain that has a separate (direct) procedure call for each possible value of `fp`. Well-known techniques (*e.g.*, such as flow insensitive points-to analysis [4, 18, 16]) can be used to obtain a reasonable (but still conservative) estimate of the values that `fp` may take on.

A Proof of Theorem 4.1

Before restating Theorem 4.1, we review some definitions.

Let the graph G and the context-free grammar CF be a context-free DAG. Let L be the language described by CF. Let the function $numValidComps$ take an L-path prefix q in G and return the number of valid completions of q.

Let q be an L-path prefix in G from $Entry$ to a vertex v. Let $w_1, \ldots, w_k$ be the valid successors of the path q. Recall that $edgeValueInContext(q, v \rightarrow w_i)$ is defined as follows:

$$edgeValueInContext(q, v \rightarrow w_i) = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{j<i} numValidComps(q \parallel v \rightarrow w_j) & \text{otherwise} \end{cases} \quad (37)$$

Equation (37) is the same as Equation (6) and is illustrated in Figure 10.

Let p be an L-path through G. Recall that the path number for p is given by the following sum:

$$\sum_{[p' \parallel v \rightarrow w] \text{ a prefix of } p} edgeValueInContext(p', v \rightarrow w) \quad (38)$$

Equation (38) is the same as Equation (7).

We are now ready to restate Theorem 4.1:

Theorem 4.1 *(Dense Numbering of L-paths) Given the correct definition of the function numValidComps, the Equations (37) and (38) generate a dense numbering of the L-paths through G. That is, for every L-path p through G, the path number of p is a unique value in the range* $[0..(numValidComps([\epsilon : Entry]) - 1)]$. *Furthermore each value in this range is the path number of an L-path through G.* □

The Ball-Larus technique achieves a dense numbering by maintaining the following invariant when assigning values to edges:

Ball-Larus Invariant: For any vertex v, for each path q from v to $Exit$, the sum of the edges in q is a unique number in the range $[0..(numPaths[v] - 1)]$.

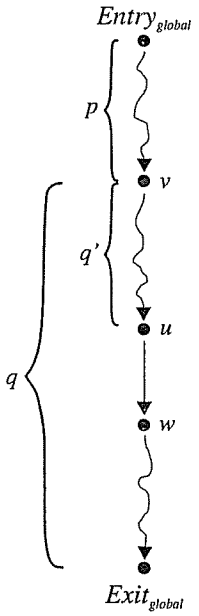


Figure 21: Schematic of the paths referred to in Equation (39). Roughly speaking, for a valid completion q of the path p, the value $\sum_{q' \| u \to w \text{ a prefix of } q} edgeValueInContext(p \parallel q', u \to w)$ is the sum of the "*edgeValueInContext*" values for the edges of q (where each edge e of q is considered with the appropriate context—part of which is supplied by p).

A consequence of this invariant is that each path from *Entry* to *Exit* has a unique path number in the range $[0..(numPaths[Entry] - 1)]$.

To prove Theorem 4.1, we show that the definition of *edgeValueInContext* given in Equation (37) maintains a similar invariant. We have the following lemma:

Lemma A.1 *The definition of edgeValueInContext given by Equation (37) satisfies the following invariant:*

Invariant 1: *For any nonempty L-path prefix p from Entry to a vertex v, let setOfValidComps(p) be the (finite) set of valid completions of p. That is, for every path q in setOfValidComps(p), p concatenated with q (denoted by p || q) is an L-path from Entry to Exit. Note that numValidComps(p) = |setOfValidComps(p)|. Then, for every nonempty path q in setOfValidComps(p), the sum*

$$\sum_{q' \| u \to w \text{ a prefix of } q} edgeValueInContext(p \parallel q', u \to w) \tag{39}$$

is a unique number in the range $[0..(numValidComps(p) - 1)]$. (Figure 21 shows the paths referred to in Equation (39).)

That is, for each valid completion q of the path p, q contributes a "unique" value n in the range $[0..(numValidComps(p) - 1)]$ to the path number associated with $[p \parallel q]$. The value n is unique in that for every valid completion $s \neq q$, the value that s contributes to the path number of $[p \parallel s]$ is different from the value that q contributes to the path number of $[p \parallel q]$.

Proof: The proof is by induction on path length (from longest path to shortest path). Let the length of a path p be the number of edges in p, and let *maxLength* be the maximum length of an L-path through G. (Note that it is not possible to have an infinite L-path, since derivations under a context-free grammar must be finite; thus, the fact that there are a finite number of L-paths through G means that there is a bound on the length of L-paths in G.)

For the base case of the induction, we show that for any L-path prefix of length *maxLength*, Invariant 1 is satisfied. An L-path prefix p of length *maxLength* must be an L-path and hence must start at *Entry* and end at *Exit*. An L-path that ends at *Exit* has only the empty path as a valid continuation, and hence satisfies Invariant 1 vacuously. It follows that any path of length *maxLength* satisfies Invariant 1.

For the inductive step, suppose that Invariant 1 is satisfied by any L-path prefix of length n that starts at *Entry*, where $1 \leq n \leq maxLength$. Consider an L-path prefix p of length $n-1$ that starts at *Entry* and ends at a vertex v. If $v = Exit$, then p satisfies Invariant 1 vacuously. Otherwise, let $w_1, \ldots, w_k$ be the valid successors of p. By the inductive hypothesis, for any valid successor w_i of p, $[p \parallel v \rightarrow w_i]$ satisfies Invariant 1; that is, for an L-path prefix of the form $[p \parallel v \rightarrow w_i]$, for every valid completion q of the path $[p \parallel v \rightarrow w_i]$, q contributes a unique value in the range $[0..(numValidComps(p \parallel v \rightarrow w_i) - 1)]$ to the path number of $[p \parallel v \rightarrow w_i \parallel q]$. This fact, combined with the definition of *edgeValueInContext*, gives us that any valid completion $[v \rightarrow w_i \parallel q]$ of the path p will contribute to the path number of $[p \parallel v \rightarrow w_i \parallel q]$ a unique value in the range:

$$[edgeValueInContext(p, v \rightarrow w_i)..(edgeValueInContext(p, v \rightarrow w_i) + numValidComps(p \parallel v \rightarrow w_i) - 1)]$$

By Equation (37), this is equal to the following range:

$$\left[\left(\sum_{j<i} numValidComps(p \parallel v \rightarrow w_j)\right)..\left(\left(\sum_{j<i} numValidComps(p \parallel v \rightarrow w_j)\right) + numValidComps(p \parallel v \rightarrow w_i) - 1\right)\right]$$

Because this holds for each successor w_i, $1 \leq i \leq k$, every valid completion of p contributes a unique value in the range

$$\left[0..\left(\left(\sum_{i=1}^{k} numValidComps(p \parallel v \rightarrow w_i)\right) - 1\right)\right] = [0..(numValidComps(p) - 1)]$$

(see Figure 10). It follows that Invariant 1 holds for the path p.

In other words, the definition of *edgeValueInContext* works for the same reason that the Ball-Larus edge-numbering scheme works—for each valid successor w_i of p, a range of numbers is "reserved" for valid completions of p that start with $v \rightarrow w_i$.

Consequently, the definition of *edgeValueInContext* in Equation (37) satisfies Invariant 1. □

Theorem 4.1 is a consequence of Lemma A.1.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*, June 1997.

[3] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proc. of the ACM SIGPLAN 98 Conf. on Program. Lang. Design and Implementation*, June 1998.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] T. Ball. Efficiently counting program events. In *TOPLAS 1994*, 1994.

[6] T. Ball and J. Larus. Efficient path profiling. In *MICRO 1996*, 1996.

[7] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1998. ACM Press.

[8] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 47–56, New York, NY, 1988. ACM Press.

[9] G.A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM Press.

[10] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *International Conference on Compiler Construction*, pages 125–140, 1992.

[11] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, New York, NY, January 1991. ACM Press.

[12] R. Muth and S. Debray. Partial inlining. (Unpublished technical summary).

[13] G. Ramalingam. *Bounded Incremental Computation.* Springer-Verlag, 1996.

[14] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proc. of ESEC/FSE '97: Sixth European Softw. Eng. Conf. and Fifth ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, volume 1301 of *Lecture Notes in Computer Science*, pages 432–449. Springer-Verlag, 1997.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, New York, NY, 1995. ACM Press. Available at "http://www.cs.wisc.edu/wpis/papers/popl95.ps".

[16] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[17] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[18] B. Steensgaard. Points-to analysis in almost-linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.