Probabilistic Calling Context and Efficient Call-Trace Collection CS701

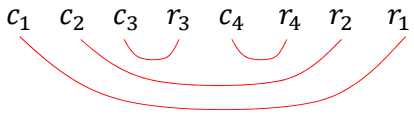
Thomas Reps
[Based on notes taken by Ashkan Forouhi on November 17, 2015]

Abstract

This lecture concerns the material in two papers: "Probabilistic Calling Context" by Bond and McKinley [3], and "Efficient Call-Trace Collection" by Wu et al. [5].

1 Introduction

In this lecture, we switch our focus from the previous profiling topics covered (path profiling, whole program paths, and statistics recovered from whole program paths) to focus on issues related to calling contexts. There are two kinds of things we can imagine a tool making use of. First, it might see (and process) the whole sequence of matching calls and returns with a nesting relationship, as shown below:



Alternatively, it might not gather the whole trace right away, but for efficiency gather enough of the trace so that the whole trace can be recovered after execution finishes, via post-processing. The trick here is that there is some redundancy, so we will have the tool put in instrumentation in a certain number of places, and after execution finishes perform an inference process to recover the information about the full trace.

The first situation is addressed in the paper by Bond and McKinley [3]. The second situation is addressed in the paper by Wu et al. [5]. We present results from both papers in this lecture.

2 Probabilistic Calling Context

When logging or profiling, the goal is to gather some kind of information, and you may be interested in having each of the pieces of information tagged with the calling context that holds at the time the information was generated. Thus, every time the program or profiler emits a piece of information, it needs to have a calling-context tag available.

Consider the stack shown in Fig. 1. Suppose that something happens after calling R. We want to emit that information tagged with the calling context $\langle \cdot, \text{Info} \rangle$, where $\cdot$ denotes some kind of calling-context tag. The problem is that the obvious technique, which is brute force, is very expensive and causes your program to slow down a lot. Thus, the question is "What kind of tracking can we do to make this information available whenever we want it?" We obviously have to do something when calls and returns occur, because the stack keeps changing; however, we want to do as little work as possible.

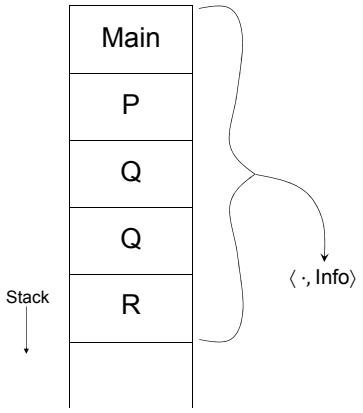
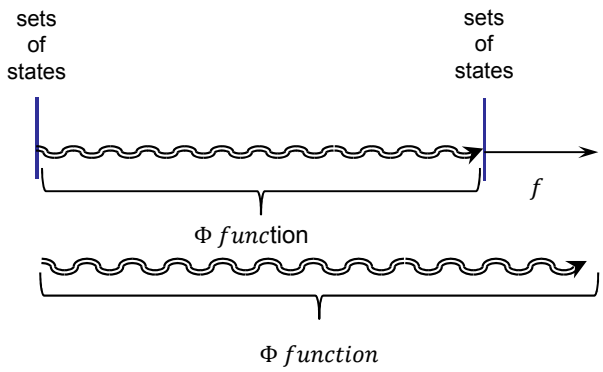


Figure 1: A stack representing the calling context.

2.1 Representing and Maintaining Calling Context in Static Analysis

Let's start by talking about so-called "call-strings." They were introduced 34 years ago in the second part of the paper on interprocedural dataflow analysis by Sharir and Pnueli [4]. The actual principle is pretty simple (and what I'll actually describe is a method of manipulating call-strings that improves on what was given by Sharir and Pnueli).

The "call-strings approach" to dataflow analysis is appropriate for static-analysis domains for which it is hard to create transfer functions that meet the conditions needed to apply the so-called "functional approach" to dataflow analysis that is described in the first part of the Sharir-Pnueli paper—i.e., dataflow analysis based on Φ-functions. Recall that for the functional approach, when you have the transfer function associated with a path and the transfer function associated with an edge that extends the path, you have to be able to mimic the composition of the two functions.



That means that you have to have an abstract domain in which an abstract-domain element represents a transition relation (of possible transitions) between two sets of states:

For some abstract domains, you may have abstract values that represent sets of states, but for which there is no convenient way to represent possible transitions between sets of states. That is, the abstract domain lets you over-approximate the set of states that can arise at a given program point, but not a way to represent the possible state transitions that can occur in going from program point p_1 to program point p_2 (such as the procedure's entry node and exit node, respectively). However, you may still want to do some kind of interprocedural analysis. The most difficult issue to deal with is procedure calls. In particular, when the analyzer considers a called procedure, it may have gotten to that procedure from multiple call-sites. Each call-site may cause different abstract values

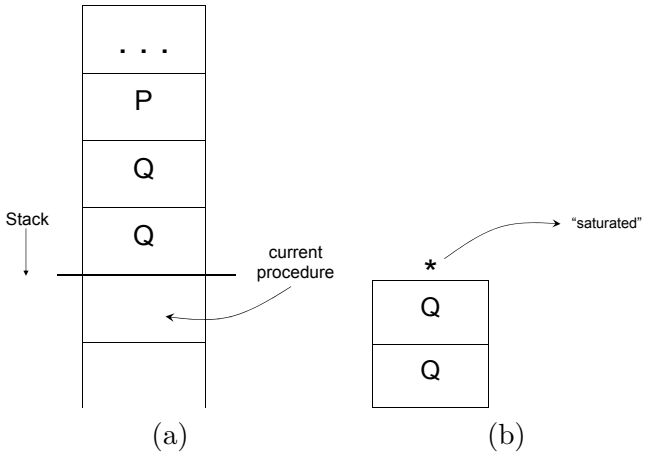
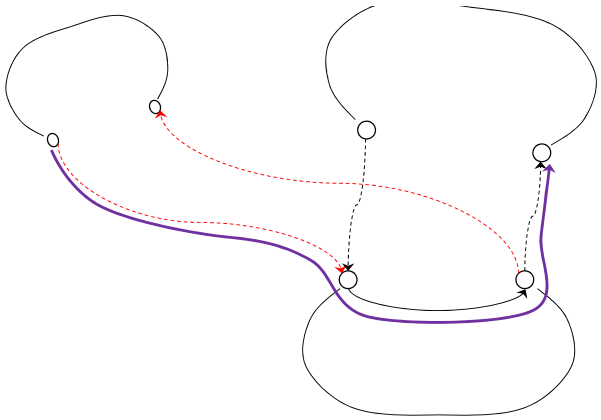


Figure 2: (a) A stack suffix; (b) the abstraction of the stack suffix with a call-string of length 2.

to be passed into the entry node of the callee. Unfortunately, you do not have a summary function for the callee—and it is that function that allows us to re-use the results of the analysis of the callee in the functional approach to interprocedural dataflow analysis. Consequently, the analyzer will have to analyze the callee multiple times, with different information each time. Moreover, when the analyzer has no way to track what initial information came from what call-site, it will pass irrelevant or imprecise answers back up to different call sites along invalid paths like the one shown below:



The mechanism that Sharir and Pnueli came up with to address this problem is essentially to equip each dataflow fact with a finite-state machine (FSM), where each FSM state is an abstraction of the contents of the call stack on entry to the procedure. You first pick a length that you are going to allow the call stack to grow to (for our example, we've picked 2).

Consider the call stack and activation record for the current procedure shown in Fig. 1(a). The call string is an abstraction of caller information, not including the current procedure (if you want the current procedure, you can always add it to the end of the current call-string). Thus, if the length is 2, the call-string would be the one shown in Fig. 1(b).

The question is now "What does the analyzer do after it analyzes a callee and needs to propagate information back to a calling procedure?" The idea is that at each point in the program, you have a set of dataflow facts, and each fact will be tagged with some call-string. Suppose that you are at C_1, the call-string length is 2, and you have two call-string/value pairs: $< \begin{array}{|c|}\hline M \\ \hline R \\ \hline \end{array}, v_1 >$ and $< \boxed{M}$

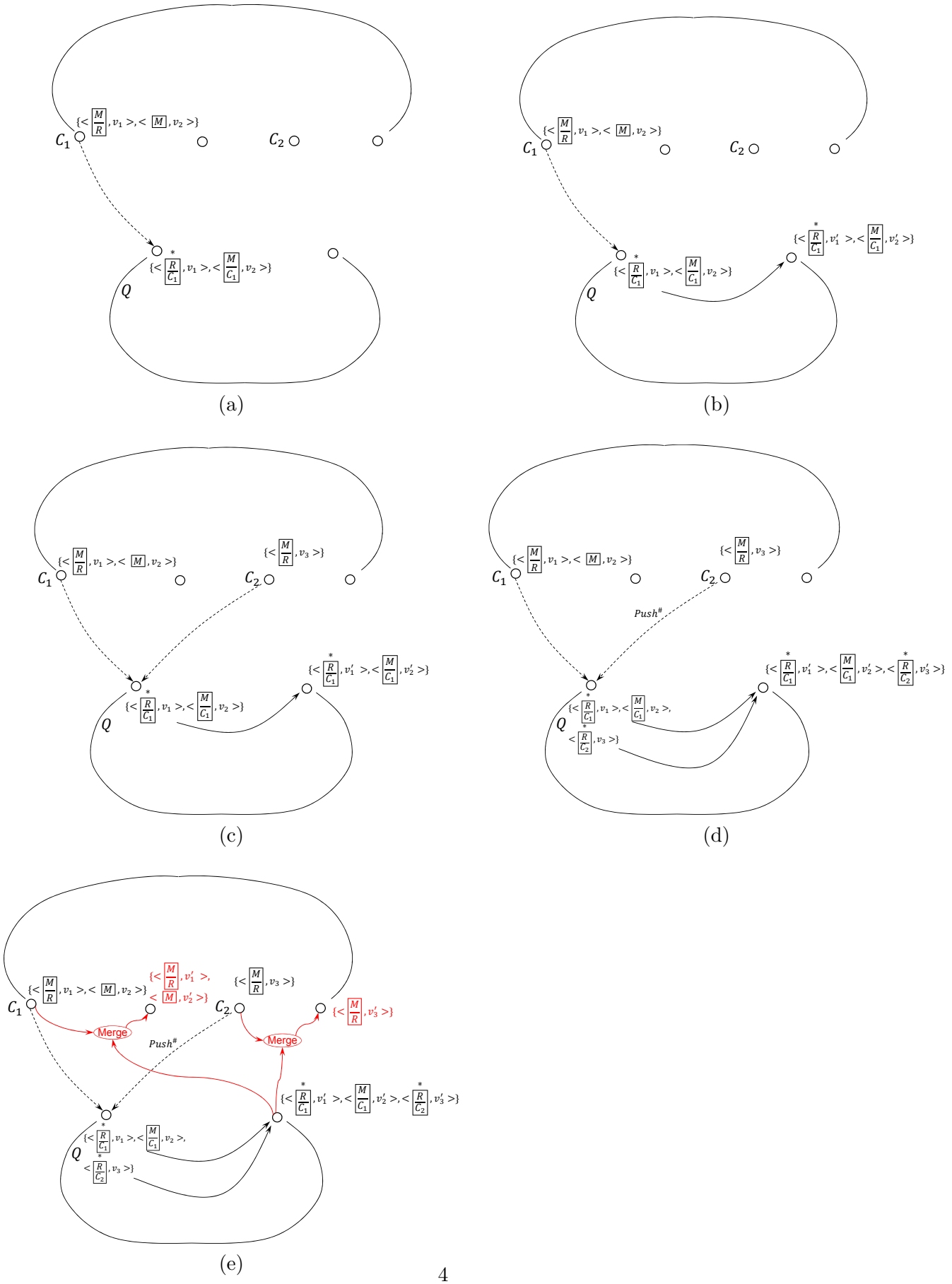


Figure 3: Propagation—over call-edges and return-edges—of dataflow facts tagged with call-strings.

$, v_2 >$. When you enter procedure Q, $\boxed{\begin{array}{c} M \\ \hline R \end{array}}$ temporarily becomes $\boxed{\begin{array}{c} M \\ \hline R \\ \hline C_1 \end{array}}$, but because this call-string is of length 3, it is turned into the saturated call-string $\overset{*}{\boxed{\begin{array}{c} R \\ \hline C_1 \end{array}}}$. Consequently, we obtain the tagged dataflow items shown in Fig. 3(a). When the analyzer reaches the end of Q, the call-strings remain unchanged. (They may have changed as the analyzer propagates information into procedures called by Q, but they will be restored when the called procedure returns to Q.) Thus, by the time the analyzer gets to the exit node of Q, we have the situation shown in Fig. 3(b).

Meanwhile, there may have been dataflow information propagated into Q from call-site C_2. Suppose that there is only one dataflow value there, namely, $< \boxed{\begin{array}{c} M \\ \hline R \end{array}}, v_2 >$. (See Fig. 3(c).) When that dataflow fact is propagated to the entry node of Q, the same process is applied. (We call this processes of modifying the call-string *push*$^{\#}$ because the call-strings are an abstraction of the stack, and—in the concrete semantics—the call on Q involves pushing a new activation record onto the stack.)

After the information from call-site C_2 propagates to the exit node of Q, we have the situation depicted in Fig. 3(d). Now the question is, "At the exit node, how do we propagate the tagged dataflow facts back up to callers?" Part of the solution is easy: just look at the "top" of the stack. (Note that in the figures, the stack-top is shown at the bottom.) In the example, the analyzer only transfers back the first two pairs to the return-site that corresponds to call-site C_1, and the last pair is transferred to the return-site that corresponds to call-site C_2. Just as we had a *push*$^{\#}$ before, we need to define a *pop*$^{\#}$. What does a *pop*$^{\#}$ do? For the case of the unsaturated pair $< \boxed{\begin{array}{c} M \\ \hline C_1 \end{array}}, v_2' >$ the answer is clear: we just remove C_1 from the call-string, and go back to having $< \boxed{M}, v_2' >$. But what about the other pairs?

Here is where we cheat a little bit. Normally, when you are doing path problems, you are only allowed to "follow your nose" by proceeding along edges of the graph. However, for the return from a procedure, we let the return-node in the caller have *two* predecessors, and use a two-argument *Merge* function to obtain the value that is passed into the return-node. The *Merge* function attempts to match each one of the call-strings at the call-node with each one of the call-strings at the exit node to check if it is possible that a *push*$^{\#}$ of the caller fact's call-string creates the call-string of the callee fact's call-string. If there is a match, then the call-string used at the return-node is the one obtained from the call-node (whereas the second component of the pair is obtained from the exit-node's pair—modulo any adjustments needed to handle the change of scopes from the callee to the caller).

For example, when checking against $\boxed{\begin{array}{c} M \\ \hline R \end{array}}, \overset{*}{\boxed{\begin{array}{c} R \\ \hline C_1 \end{array}}}$ matches, but $\overset{*}{\boxed{\begin{array}{c} R \\ \hline C_2 \end{array}}}$ does not: *push*$^{\#}\left(\boxed{\begin{array}{c} M \\ \hline R \end{array}}, C_1\right) = \overset{*}{\boxed{\begin{array}{c} R \\ \hline C_1 \end{array}}}$, but *push*$^{\#}\left(\boxed{\begin{array}{c} M \\ \hline R \end{array}}, C_1\right) \neq \overset{*}{\boxed{\begin{array}{c} R \\ \hline C_2 \end{array}}}$. The result of this process is depicted in Fig. 3(e).

The manipulations of the call-string are reminiscent of state changes in a finite-state machine. However, they differ from what goes on in a finite-state machine because the state change at a return-node involves a *two-argument* transition function. In fact, the kind of mechanism pioneered by *Merge* functions has been formalized as a purely automata-theoretic device called a Visibly

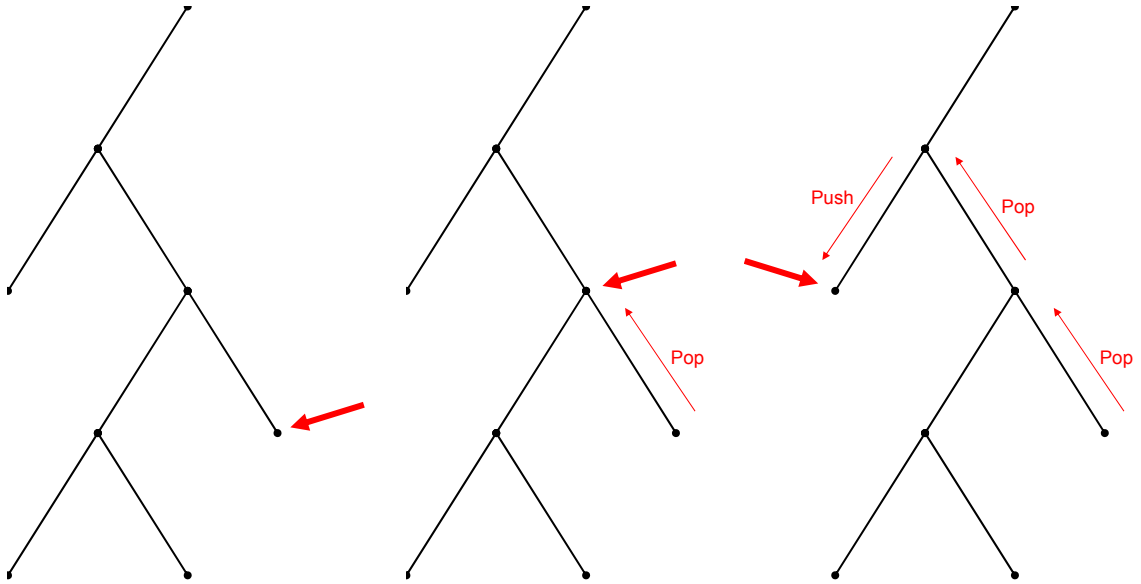


Figure 4: Positions in a calling-context tree.

Pushdown Automaton (VPA) [1]. A VPA accepts words whose alphabet symbols are labeled to distinguish them as coming from three disjoint sets of symbols: *internal symbols*, *call symbols*, and *return symbols*.[1] In our case, the transitions for call-strings have a particularly simple structure:

- Internal Edges: $\delta_i = \{(cs, e, cs)\}$ (i.e., δ_i does not change the call-string state)
- Call Edges: $\delta_c = \{(cs_c, c, push^{\#}(cs_c, c))\}$ (i.e., δ_c performs a $push^{\#}$)
- Return Edges: $\delta_r = \{(cs_x, cs_c, r, cs_c) \mid push^{\#}(cs_c, c_r) = cs_x\}$, where c_r denotes the call-site that corresponds to the target of r. (In other words, δ_r restores cs_c whenever cs_x is one $push^{\#}(\cdot, c_r)$ away from cs_c.)

2.2 Representing and Maintaining Calling Context in Dynamic Analysis

What we discussed in §2.1 was one way in which calling context can be represented and maintained by a static analyzer. Let's now return to the problem of representing and maintaining calling context during dynamic analysis. As discussed at the beginning of the lecture, we want to set things up in the run-time environment so that, at any moment, we can tag information with a label that represents the current calling context (see Fig. 1).

Obviously, the call stack is part of the run-time environment, so one approach would be to walk the stack and emit a list of the pending calls. Typically, a stack frame has a stored frame pointer, which points into the caller's activation record; moreover, one would typically have the ability to move from that address to the stored frame pointer in the caller's activation record, and so on. By this means you can traverse the entire stack—in essence, the collection of activation records on the stack is just a singly linked list. You also have to have some way of identifying the call-site to which a given activation record corresponds. However, in each activation record, the return address in the caller is typically stored at a given offset from the frame pointer, which provides us with the information that we need for emitting the list of pending calls.

Another thing you could do is to have the program build a calling-context tree. Suppose that

[1]Technically, an interprocedural dataflow analyzer that uses *Merge* functions acts like a mixture of VPAs and a related formalism called Nested Word Automata (NWA) [2]. In particular, the use of three kinds of alphabet symbols is similar to VPAs; the use of a two-argument state-transition function for returns is similar to NWAs.

at a given moment, the program's calling context is represented by the node indicated in Fig. 4(a). If the program now returns from the current called procedure, performing a pop, you would leave that node, but would not de-allocate it—thus moving to the situation depicted in Fig. 4(b). If the program does another pop and then a push that happens to go to something already contained in the calling-context tree, you would not have to build anything, but just move to an already-existing node, as shown in Fig. 4(c). Thus, at any moment that you want to write out the current calling-context information, you can just write the address of the current node in the calling-context tree.

The overhead that the authors quote is a 2–4× slow-down of the program if you use a calling-context tree. In contrast, with their method, the overheads are very low; they give three numbers, corresponding to three hypothetical clients that require calling-context information in different circumstances:

A query at every system call	0% overhead
A query at every `java.util` call	3% overhead
A query at every `java.API` call	9% overhead

It must be said that with their method, you will not have a firm guarantee that you are getting completely accurate information, but their numbers are such that with either 64-bit or 32-bit values, there is a good chance that no imprecision arises.

Before getting to what they do, let's think a little bit about what the requirements are:

- You need to make it efficiently computable.
- You need to make it deterministic. (This requirement is not actually a firm requirement. You need to make it re-playable, meaning that if you are given whatever the calling-context descriptor turns out to be, you can figure out unambiguously what set of call stacks it corresponds to.)
- You need to avoid "stupid" (i.e., avoidable) collisions. For example, you do not want the calling context for "Main calls A calls B calls C" to have the same descriptor as "Main calls B calls C calls A." This example suggests that whatever operation is performed, it needs to be non-commutative.

What Bond and McKinley do is simple and cute: You keep a numeric value; perform a very simple calculation on it at each call; and perform its inverse at each return. For an initial value v, a push is going to take you to $3 * v +$ hash(call-site). The hash function they use is a hash of the signature, plus the line number. On a pop, they do the reverse: subtract the hash, and divide it by 3. Alternatively, you can store a copy of the value in a local variable, so that when you return, you can just pick it up from the local variable. In the paper, they say that they use this technique to help "... correctly [maintain v] in the face of exception control flow;" however, the paper does not explain what they do for exceptions, nor whether exceptions cause the technique to lose temporarily the ability to provide a usable calling-context tag.

In each Java method, they instrument the code using the scheme shown below, where the boxed code indicates the instrumentation that is introduced, and f is the function $\lambda v.\lambda CS.3*v+\text{hash}(CS)$:

```
method () {
    int temp = v;
    ...
    v = f(temp, CS_1);
CS_1: A();
    ...
    v = f(temp, CS_2);
CS_2: B();
```

```
        ...
}
```

One good feature of the approach is that one adds the instrumentation code and then compiles. This order allows the method to work even if inline-expansion optimizations are performed. For instance, suppose that the optimizer performs an in-line expansion of `method_2` in `method_1` at call-site `CS_i`:

```
    method_1 () {
        int temp = v;
        ...
        v = f(temp, CS_i);
CS_i: method_2();
        ...
}

method_2 () {
        int temp = v;
        ...
        v = f(temp, CS_j);
CS_j: method_3();
        ...
}
```

Then at the in-lined instance of `CS_j`, we will have `v = f(f(temp, CS_i), CS_j);`, which equals $9*v+3*\text{hash}(\texttt{CS\_i})+\text{hash}(\texttt{CS\_j})$. Note that $3*\text{hash}(\texttt{CS\_i})+\text{hash}(\texttt{CS\_j})$ is a compile-time constant.

Like the call-strings used by Sharir and Pnueli, v does not include the contribution from the current procedure. However, the appropriate calling-context tag can be computed by applying f with the current program point:

```
    method () {
        int temp = v;
        ...
        v = f(temp, CS_1);
pp: query(f(temp,pp));  // query is the user-specified monitoring function
        ...
}
```

Bond and McKinley had 19 benchmark programs, and reported measurements when the benchmarks were run with medium and large input sets. What they found was that when they went from medium to large inputs, they encountered as many as 6× as many calling contexts. However, in 10–12 of the 19 benchmarks—depending on which of the hypothetical clients (mentioned earlier) they assumed was in use—there were *no* new calling contexts. In other words, for a significant number of the examples, the calling context is pretty stable, but for some it can go up substantially.

One issue that is not addressed in the paper is that, if you are given some number, you would like to know what stack configuration(s) it corresponds to. To find this out, you will need to carry out some kind of search of the space of calling-context tags; however, the algorithm to carry this out is not discussed in the paper.

Another interesting matter is the numbers provided about the expected number of collisions, for both 32-bit and 64-bit calling-context tags. They just considered a random model, with n values

selected randomly. The resulting table, which shows the number of collisions for 10^k numbers drawn at random, for varying values of k, is as follows:

	32 bit	64 bit
10^3	0	0
10^4	0	0
10^5	1	0
10^6	116	0
10^7	11K	0
10^8	1.1M	0
10^9	107M (11%)	0
10^{10}	6B (61%)	3

One drawback of the method is that for certain kinds of applications, you can imagine wanting to compare two runs of slightly different programs. For this situation, you would like to have a fingerprint that is resilient to whatever small changes were made to the program. By using the line number of the call-site in the hash function, their implementation technique will often destroy the ability to compare calling-context tags across runs of slightly different programs. For instance, if there is a one-line change in a file, all call-sites later in the file will have different line numbers, and thus any calling context that involves any of those functions will likely not have the same value of v. Even worse, if you add a one-line comment to `main`, the value of v for *all* of the calling contexts in the program are likely to change. Thus, for applications in which one desires to be able to compare runs of slightly different programs, one would want to use a hash function that is more resilient to such changes, such as a hash of the text of the whole function, minus all comments and whitespace.

3 Efficient Call-Trace Collection

We now turn to the paper by Wu et al. [5] about Casper, a tool for call-trace collection. In this work, the goal is to recover an entire trace of calls and returns. The baseline method is to log information at every call and return transition, which on average has a runtime overhead of 213%, and produces about 50 GB of data per hour. A simple optimization is to log only the first call in each branch, because in a piece of non-branching code, once you are on a path, you will continue on that path (at least if the program does not crash). This one technique alone reduces the average overhead to 128%. The authors claim that Casper has only a 68% overhead.

3.1 Overview

The technique of Wu et al. involves a context-free language that is an over-approximation of the traces of calls and returns in executions of the program. (The context-free language is the same as what you have seen before in interprocedural dataflow analysis, where returns must match with preceding calls.) Note that in an execution of the program, the actual calls and returns that are made are controlled by the concrete execution state—and thus the branches taken by the program can be correlated; that is, it might be the case that whenever there is a call to P, the program also calls Q, or alternatively, whenever there is a call to P the program never calls R. Consequently, there can be strings in the context-free language that could never be traces generated by an execution of the program.

For the moment, just accept that there is an appropriate context-free language L that we want to use. We will show that there is an easy way to construct a grammar G, such that:

1. $L = L(G)$
2. G is LL(1).

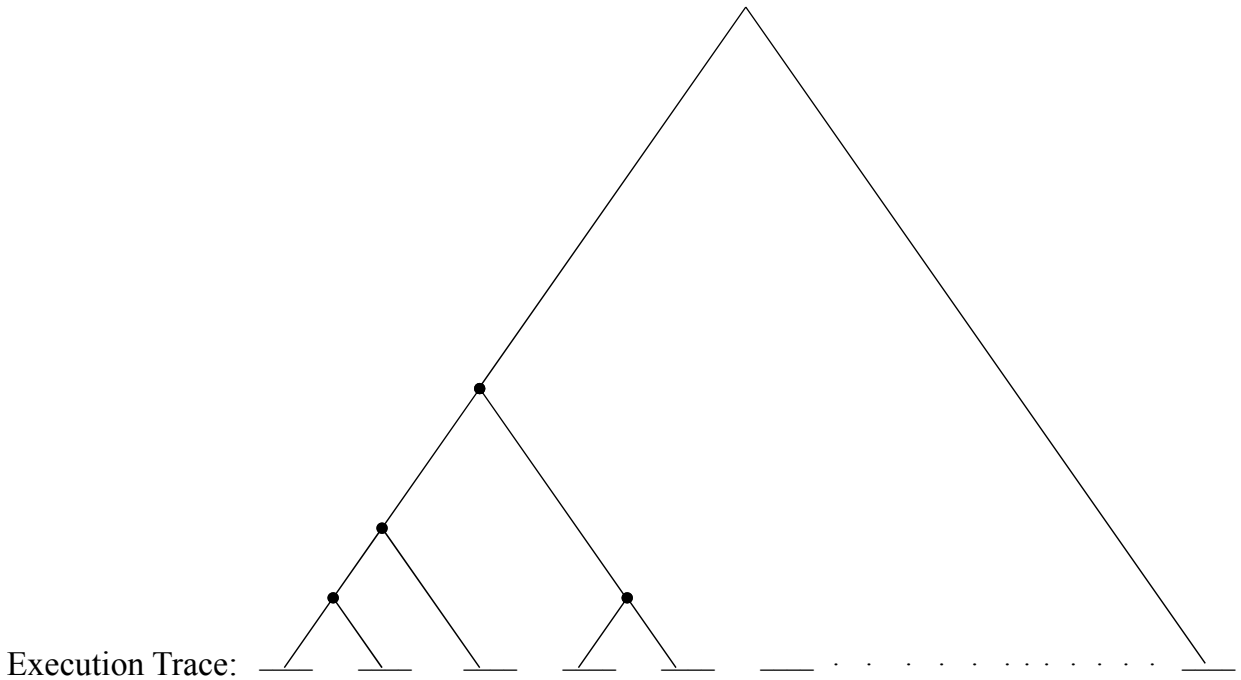


Figure 5: A parse tree for an execution trace.

In other words, there is way of constructing a description of L, such that all strings of L can be parsed by a predictive parser.

One approach, which corresponds to the baseline method, would be to log calls and returns and parse the resulting string to obtain a parse tree for the string (with respect to grammar G). Fig. 5 shows this in outline form: one has an execution trace with a set of symbols (calls and returns); above that, there is the parse tree for this execution trace.

What Wu et al. do is to log only a subset of the calls and returns. The string that is obtained is thus just a portion of the execution trace. They instrument a subset of the call and return points, called the Instrumentation-Points set I, and only the calls and returns in I are logged. In other words, only elements of I in the input string are "visible," and so instead of string σ we have σ projected on I (which we will denote by $\sigma|_I$).

The question is "Which symbols should be dropped?" Wu et al. aim to use a predictive parser on $\sigma|_I$, and want to arrange things so that the predictive parser builds a tree that is essentially isomorphic to the parse tree of the full string. The difference is that at places where there are missing symbols, the parse tree will not have a child. However, because this is a predictive parser, and because there is a known correspondence between the two productions, we can use the correspondence to restore the parse tree, and therefore, restore the full string! This goal is depicted in rather sketchy form in Fig. 6.

Let's now take a look at the example they provide in the paper, which is shown in Fig. 7. Suppose that the actual string is:

$$c_1 r_2 c_2 c_5 r_4 r_3 c_2 c_5 r_4 r_3 c_3 c_7 r_4 r_6 c_4 r_7 r_1,$$

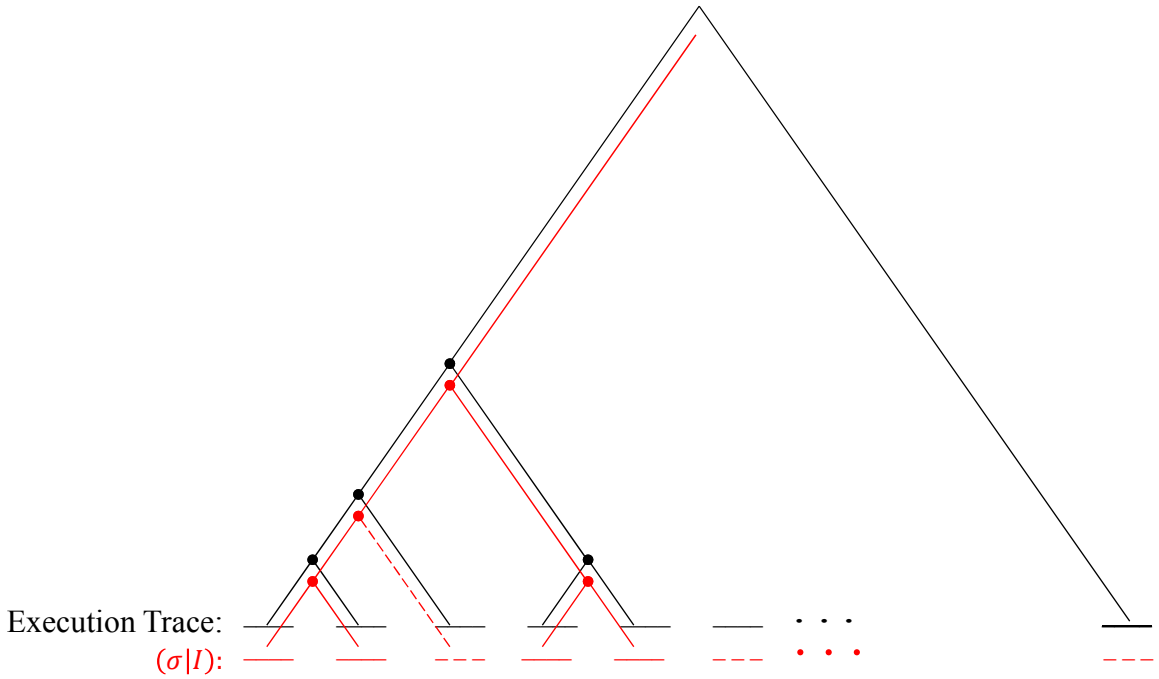


Figure 6: A parse tree for string σ and the string $\sigma|_I$. $\sigma|_I$ is missing the input symbols indicated by dashes, and its parse tree is missing the associated subtrees. (One such subtree is indicated by the dashed sloping line in the tree.)

```
1  void main(String[] args){
2    A();  // c1
3    do{
4      if (p1)
5        B();  // c2
6      else
7        E();  // c3
8    }while(p2);
9    H();  // c4
10   return;  // r1
11 }
12
13 void A(){
14   ...//no call sites
15   return;  // r2
16 }
17
18 void B(){
19   if (p3)
20     C();  // c5
21   else
22     D();  // c6
23   return;  // r3
24 }
25 void C(){
26   ...//no call sites
27   return;  // r4
28 }
29
30 void D(){
31   ...//no call sites
32   return;  // r5
33 }
34
35 void E(){
36   if (p3)
37     C();  // c7
38   else
39     D();  // c8
40   return;  // r6
41 }
42
43 void H(){
44   ...//no call sites
45   return;  // r7
46 }
```

Figure 7: An example program [5].

but all you are given is:

$$c_5 c_5 c_7$$

Basically, we can see that from the first c_5, we can infer the first part of the string:

$$[c_1 r_2 c_2 \overleftarrow{c_5}] r_4 r_3 c_2 c_5 r_4 r_3 c_3 c_7 r_4 r_6 c_4 r_7 r_1$$

How do we do that? Starting from `main()`, the only action that one can do is call `A()`, which generates c_1. `A()` does nothing but return, which generates r_2. Now the program has a choice of either going to c_2 or c_3. The authors note that we can recover that the program must have gone down the then-branch at line 5 to call `B()`. The call to `B()` would generate c_2, and `B()` would make a call to `C()` at c_5. If the program had called `E()` from `A()` instead, a c_3 would have been generated, and the program would have performed a call on `E()`, at which point the program would have had to perform either a return or another call before we got back to `B()`, where our call site c_5 was. We can continue to apply the same reasoning method, and recover the entire string:

$$[c_1 r_2 c_2 \overleftarrow{c_5}][r_4 r_3 c_2 \overleftarrow{c_5}][r_4 r_3 c_3 \overleftarrow{c_7}][r_4 r_6 c_4 r_7 r_1 \overleftarrow{]}$$

The final segment, $[r_4 r_6 c_4 r_7 r_1 \overleftarrow{]}$ is recovered using the fact that there was nothing else in string $\sigma|_I$ after c_7, plus the information that the program did not crash.

3.2 Problem Formalization

In a nutshell, the problem is addressed as follows:

1. The set of possible traces is modeled (over-approximated) by a context-free language.
2. There is an easy way to give a grammar for the language that is LL(1), and hence parsable by a predictive parser.
3. It is possible to devise a way to pick instrumentation sites I such that the parse of any "trimmed-down" string (i.e., which only has alphabet symbols in I) is 1-for-1 with the parse of the full string. One uses the former to recover the latter, which fills in the symbols missing in the trimmed-down string.

The starting point is the LL(1) description of the full language of the program's call-trace strings. Wu et al. express the grammar as productions of a special form: each right-hand side consists of a terminal symbol for some call/return site, followed by other symbols.

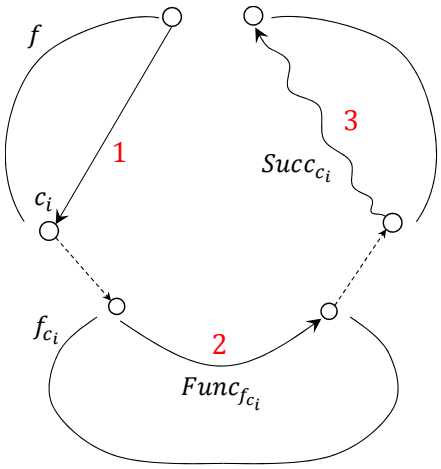
At this point we need to introduce some terminology and notation. From now on, I will use the term "transfer site" when I want to refer to call nodes and exit nodes, but do not need to distinguish between the two. Wu et al. work with what they call the *call-site control-flow graph* (CSCFG), which is a version of the control-flow graph that is projected down to just procedure entry nodes, procedure exit nodes, and all call sites. The CSCFG has an edge $m \to n$ if there is a path in a procedure's CFG from m to n that does not pass through another call-site.

The next step is to define the appropriate LL(1) context-free grammar. The grammar consists of the following kinds of elements

1. t_i : t_i is a terminal symbol that represents a transfer-site (call or return) in some function f.
2. $Func_{f_{c_i}}$: $Func_{f_{c_i}}$ is a nonterminal whose language is the set of call-trace fragments generated for the function f_{c_i} at call-site c_i.
3. $Succ_{t_i}$: $Succ_{t_i}$ is a nonterminal whose language is the set of call-trace fragments generated for the path from (i) the successor of transfer-site t_i in function f to (ii) the exit of f.

It is now possible to define our grammar $G = (V, \Sigma, S, Productions)$:

- S : $Func_{main}$
- Σ : The program's calls and returns



(a)

(1) $Func_{main} \to c_1 Func_A Suc_{c_1}$
(2) $Func_A \to r_2$
(3) $Func_B \to c_5 Func_C Suc_{c_5}$
(4) $Func_B \to c_6 Func_D Suc_{c_6}$
(5) $Func_C \to r_4$
(6) $Func_D \to r_5$
(7) $Func_E \to c_7 Func_C Suc_{c_7}$
(8) $Func_E \to c_8 Func_D Suc_{c_8}$
(9) $Func_H \to r_7$
(10) $Suc_{c_1} \to c_2 Func_B Suc_{c_2}$
(11) $Suc_{c_1} \to c_3 Func_E Suc_{c_3}$
(12) $Suc_{c_2} \to c_4 Func_H Suc_{c_4}$
(13) $Suc_{c_2} \to c_2 Func_B Suc_{c_2}$
(14) $Suc_{c_2} \to c_3 Func_E Suc_{c_3}$
(15) $Suc_{c_3} \to c_4 Func_H Suc_{c_4}$
(16) $Suc_{c_3} \to c_2 Func_B Suc_{c_2}$
(17) $Suc_{c_3} \to c_3 Func_E Suc_{c_3}$
(18) $Suc_{c_4} \to r_1$
(19) $Suc_{c_5} \to r_3$
(20) $Suc_{c_6} \to r_3$
(21) $Suc_{c_7} \to r_6$
(22) $Suc_{c_8} \to r_6$

(b)

Figure 8: (a) A path described by the production $Func_f \to c_i \; Func_{f_{c_i}} \; Succ_{c_i}$. (b) The grammar created for the program in Fig. 7 [5].

- $V = \{Func_f \mid f \text{ is a function in the program}\} \cup \{Succ_{c_i} \mid c_i \text{ is a call site}\}$

The grammar has productions for the call-sites that are upwards-exposed in each procedure, as well as for cases where there are no calls.[2] These productions have the following form:

$$\begin{array}{lcll} Func_f & \to & c_i \;\; Func_{f_{c_i}} \;\; Succ_{c_i} & (c_i \text{ is an upwards-exposed call site in } f) \\ Func_f & \to & r_j & (r_j \text{ is an upwards-exposed return site in } f) \end{array}$$

To combine both kinds of productions into ones of a single form, we have

$$Func_f \;\to\; t_i \;\; \gamma_{t_i},$$

where γ_{t_i} is ε for an upwards-exposed return-site.

The grammar has similar rules for the $Succ_{c_i}$ nonterminals, which represent the set of call-trace fragments in the procedure after the call on c_i returns:

$$\begin{array}{lcll} Succ_{c_i} & \to & c_j \;\; Func_{f_{c_j}} \;\; Succ_{c_j} & (\text{there is an edge } c_i \to c_j \text{ in the CSCFG}) \\ Succ_{c_i} & \to & r_j & (\text{there is an edge } c_i \to r_j \text{ in the CSCFG}) \end{array}$$

or

$$Succ_{c_i} \;\to\; t_i \;\; \gamma_{t_i}.$$

Our first observation is that this grammar is LL(1). The two properties of an LL(1) grammar are:

1. The grammar does not have any left-recursion: In our case, all the rules start with a terminal t_i, so this requirement is fulfilled.
2. Whenever we have two productions $A \to \alpha$ and $A \to \beta$:
   (a) $First(\alpha) \cap First(\beta) = \emptyset$. In our case, for each pair of productions of the form $N \to t_i \; \gamma_{t_i}$ and $N \to t_j \; \gamma_{t_j}$, t_i and t_j must be different.

[2] A transfer-site t is *upwards-exposed* in a procedure P when there is a path from the entry node of P to t that does not pass through another transfer-site.

(b) At most one of α and β can derive ε. In our case, this requirement is vacuously satisfied because none of the right-hand sides of the productions can derive ε.

(c) If $\beta \xrightarrow{*} \varepsilon$, then $\mathit{First}(\alpha) \cap \mathit{Follow}(A) = \emptyset$. This requirement is vacuously satisfied because none of the right-hand sides of the productions can derive ε.

(d) If $\alpha \xrightarrow{*} \varepsilon$, then $\mathit{First}(\beta) \cap \mathit{Follow}(A) = \emptyset$. Again, this requirement is vacuously satisfied.

We restrict the strings to alphabet symbols in I, which is a subset of the set of transfer-sites. The way to do that is via a grammar transformation: each production in G of the form $N \to t_i \gamma_{t_i}$, it is transformed as follows:

$$\begin{array}{ll} N' \to t_i \gamma'_{t_i}, & \text{if } t_i \in I \\ N' \to \gamma'_{t_i}, & \text{if } t_i \notin I \end{array}$$

This transformation maps G to G', where the set of nonterminals is mapped one-to-one from N to N'. The problem is that *Productions* can be mapped many-one to *Productions'*, which can cause G' to fail to be LL(1), and hence one could not use a predictive parser.

However, the only situations in which the mapping is many-one is when there are two productions of the form $P_i : N \to t_i$ and $P_j : N \to t_j$, where $t_i, t_j \notin I$. Thus, the first requirement on us is that we must find a way to avoid such cases. The second requirement on us is that we need the grammar G' that one obtains to be parsable by a predictive parser.[3]

The actual algorithm they propose for finding G' in the face of the NP-hardness result uses the following idea: take your program, and for the purposes of finding an instrumentation scheme, take out all the back-edges created by recursive calls; you are then left with something that is an (artificial) non-recursive program. They show that even for this restricted situation, the optimal-instrumentation problem is a hard problem. The reason is that if you just look at each individual procedure, you have to solve a vertex-cover problem, which is NP-hard. However, there is an approximation algorithm for the vertex-cover problem, and Wu et al. use that algorithm as a subroutine in the construction of an appropriate G'.

There are several other issues that they deal with in the paper. One is dealing with crashes. If the program crashes, one will not necessarily have an input string that generates a whole parse tree. Thus, they lift the definition of a feasible call trace to be a *prefix* of a string that parses. In the case of traces created using the reduced instrumentation set I, there is a problem, because even if you can complete the parse tree up to the places where you have symbols, you won't necessarily be able to reconstruct the entire tail of the trace beyond the point of the crash. To handle this problem, they assume that they have the stack at the crash point, and they use the stack information to help fill in the missing portion of the parse tree.

They also need to deal with callbacks; virtual calls; finalize and static initializers; and exceptions. There are short paragraphs in the paper describing what they do for for each of these constructs.

References

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.

[3] M.D. Bond and K.S. McKinley. Oopsla. In *Probabilistic Calling Context*, 2007.

[4] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[3] I think the paper makes a claim that is not correct. They show that the problem of minimizing the size of I is NP-hard, if you restrict the tool to use an LL(1) grammar. They then make the argument that it would be NP-hard if the tool were allowed to use an LL(k) grammar. However, a parser for an LL(k) grammar has more freedom to look ahead than a parser for an LL(1) grammar, and thus a minimal solution to the LL(1) version of the problem does not, in general, give you a minimal solution to the LL(k) version of the problem.

[5] R. Wu, X. Xiao, S.-C. Cheung, H. Zhang, and C. Zhang. Princ. of prog. lang. In *Casper: An Efficient Approach to Call Trace Collection*, 2016.