Precision vs. Scalability: Context Sensitive Analysis with Prefix Approximation

Raveendra Kumar Medicherla
TCS Limited, Bangalore, India
Indian Institute of Science, Bangalore, India
raveendra.kumar@tcs.com

Raghavan Komondoor
Indian Institute of Science, Bangalore, India
raghavan@csa.iisc.ernet.in

***Abstract*—Context sensitive inter-procedural dataflow analysis is a precise approach for static analysis of programs. It is very expensive in its full form. We propose a *prefix* approximation for context sensitive analysis, wherein a prefix of the full context stack is used to tag dataflow facts. Our technique, which is in contrast with *suffix* approximation that has been widely used in the literature, is designed to be more scalable when applied to programs with modular structure. We describe an instantiation of our technique in the setting of the classical call-strings approach for inter-procedural analysis. We analyzed several large enterprise programs using an implementation of our technique, and compared it with the fully context sensitive, context insensitive, as well as suffix-approximated variants of the call-strings approach. The precision of our technique was in general less than that of suffix approximation when measured on entire programs. However, the precision that it offered for outer-level procedures, which typically contain key business logic, was better, and its performance was much better.**

I. Introduction

Dataflow analysis [1] is an automated program analysis technique that has been used extensively in the context of reverse engineering and re-engineering scenarios such as program comprehension [2], [3], data model recovery and file-format recovery [4], [5], [6], and binary analysis [7]. Dataflow analysis also forms the foundation for construction of program dependence graphs and program slices, which, in turn, have numerous applications in software engineering [8].

Dataflow analysis is performed by propagating "abstract facts" from a chosen lattice, which forms the abstract domain, through the program. The final fix-point solution obtained associates an abstract fact with each program point, which encodes a conservative approximation of properties that are guaranteed to hold at that point at run-time.

When a program involves procedures and procedure calls, an *inter-procedural* dataflow analysis is required. A *feasible* path in such a program is one that does not enter a procedure from a certain call-site and then return from that invocation to a location other than the one that follows this call-site. A *context insensitive* inter-procedural analysis propagates facts along all paths, including infeasible paths, and hence potentially computes imprecise results. A context sensitive inter-procedural analysis is one that performs propagations only along feasible paths (to the extent possible). Empirical studies [9], [10] have confirmed that context sensitive analysis is more precise than context insensitive analysis.

Context sensitivity, however, increases analysis cost significantly, as has been shown in the same studies mentioned above. It can also potentially lead to non-termination in the presence of recursion. To curb this, limits are typically imposed on context sensitivity, which essentially trade off some of the precision gain for efficiency. A common way of enforcing a limit is to use *suffix approximated* calling contexts. Intuitively, what this means is that only certain number of items from the *top* of the stack of contexts are used during the analysis, as opposed to full context stacks. The primary contribution of this paper is an alternative proposal, which is to restrict context stacks to a certain number of entries from the *bottom* of the stack. In the rest of this section we first introduce suffix approximation. We then introduce our technique of prefix approximation, as well as its benefits.

A. Suffix Approximated Context Stacks

The idea of suffix approximation has been employed in two generic approaches that are aimed at making any given underlying "client" dataflow analysis context sensitive: the classical call-strings approach [11], as well as the more recent object-sensitivity approach [12]. In this paper we focus primarily on the call-strings approach. In this approach, each dataflow fact is tagged with a string (representing a stack) of call-site IDs, which serves as the context for the fact. In the full (i.e., "unlimited") call-strings approach, there is no bound on the lengths of call strings. Therefore, even in non-recursive programs, in general an exponential number of distinct dataflow facts could reach certain program points (exponential in terms of the total number of call-sites in the program). With k-suffix approximation, where $k > 0$ is a user parameter, call strings are limited to be of length k, thus limiting the number of call strings at any program point to K^k in the worst case, where K is the number of distinct call-sites in the program.

We illustrate the call-strings approach using the example program in Figure 1. This program will serve as our running example. It contains a *main* procedure (at the leftmost position) and procedures *f*, *g*, *h*, and *i*. Note that for simplicity of illustration procedure i has no statements. For simplicity we use only global variables in this program, namely, a and b. In this example, the *possibly uninitialized variable* analysis [13] is used as a client analysis. This analysis determines whether each variable at each program point is possibly uninitialized, or contains a value that was computed by an expression that itself involved possible uninitialized operands, along any path to that point. Therefore, each tagged dataflow fact would be of the

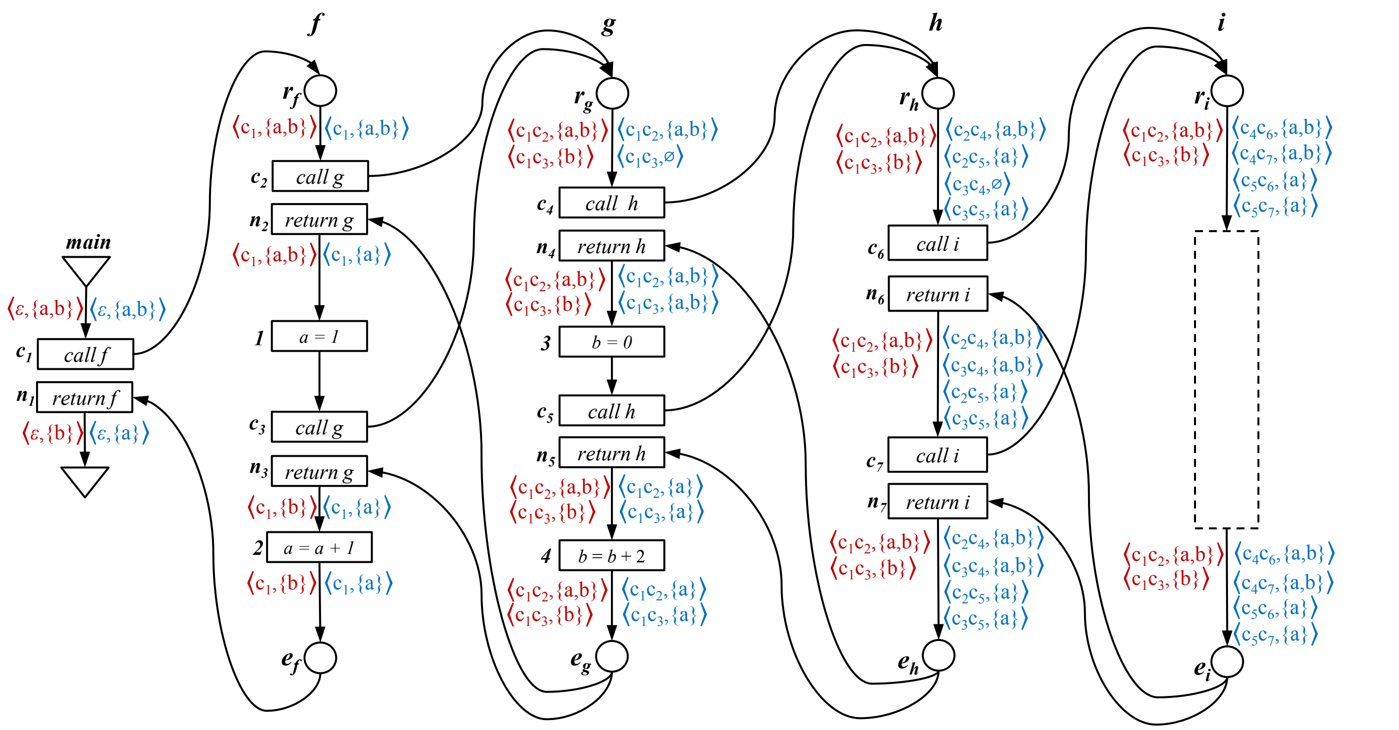


Fig. 1. Running example, with fix-point solution. Dataflow facts to the left of control-flow edges correspond to 2-prefix approximation, while facts to the right of edges correspond to 2-suffix approximation.

form $\langle \gamma, d \rangle$, where γ represents a call string and d represents a set of variables (that are possibly uninitialized).

In the figure, for brevity, the dataflow facts due to full context sensitivity are not shown. For instance, at program point r_i (in procedure i), with full context sensitivity, eight tagged dataflow facts would reach. These would include the facts $\langle c_1c_2c_4c_6, \{a, b\} \rangle$, $\langle c_1c_2c_5c_6, \{a\} \rangle$, etc.

The fix-point facts due to 2-suffix approximation are shown to the right of the control-flow edges, in blue colour, at key program points. Consider, for instance, the facts $\langle c_1c_2, \{a, b\} \rangle$ and $\langle c_1c_3, \emptyset \rangle$ at point r_g. Intuitively, the first fact is due to the path that comes directly to r_g via c_1 and c_2, while the second fact is due to the path that visits procedures g, h, and i via c_2, then returns to n_2, then flows through point 1, and then reaches r_g via c_3. Along the second path mentioned above both the variables get initialized – first variable b at point 3 and then variable a at point 1. Then, the two facts mentioned above, upon propagation via c_4, become $\langle c_2c_4, \{a, b\} \rangle$ and $\langle c_3c_4, \emptyset \rangle$, respectively, at point r_h. In other words, c_1 is removed from both the call-strings in order to restrict their lengths to 2. Now, these two facts, upon further propagation via c_6, end up having the same call-string c_4c_6. Since the approach maintains at most one fact per call-string at any program point, these two facts get *joined*, yielding the fact $\langle c_4c_6, \{a, b\} \rangle$ at r_i. (For the possibly uninitialized variables analysis, the union operation is the join.) This fact then reaches e_i.

Along return edges, the approach propagates a data fact to that caller of the current procedure that contains the call-site that is at the top of the call-string of the fact. For example, along the edge $e_i \to n_6$, the incoming data fact $\langle c_4c_6, \{a, b\} \rangle$ (that was mentioned above) is propagated to the point after n_6. During this propagation the call-site c_6 is popped from the call-string, leaving behind only c_4. Now, since procedure g, which contains the call-site c_4, could have been invoked either via call-site c_2 or via call-site c_3, the fact mentioned above is sent to n_6 as *two* different facts, namely, $\langle c_2c_4, \{a, b\} \rangle$ and $\langle c_3c_4, \{a, b\} \rangle$. Note that the latter of these two facts is *imprecise*, because when control reaches n_6 via a path that has come through c_3 both the variables would have become initialized (at points 3 and 1). In other words, this fact should have had an empty set of variables. This fact subsequently reaches e_h unmodified (via c_7, procedure i, and n_7), and then gets sent to point 4 with tag c_1c_3. This fact flows through point 3, and thus becomes $\langle c_1c_3, \{a\} \rangle$. It then flows through procedures h and i, and reaches e_g unmodified. It then gets propagated to n_3 and then point 2. Here, the presence of the variable a in the set in the fact causes the use of variable a in the rhs (i.e., right hand side) at point 2 to be declared as possibly uninitialized, which is a false warning. This false warning arises due to the imprecision mentioned earlier. On the other hand, the use of b in the rhs at point 4 is determined to be definitely initialized. Intuitively, this is because both the facts that contain b at e_i eventually get propagated only to n_4 (from where they reach the initialization at point 3), and not mistakenly to n_5.

Note that false warnings are in general unavoidable when full calling contexts are not used. With suffix approximation, precision tends to get compromised at outer-level procedures. Note that the use of a at point 2 is called possibly uninitialized, whereas the use of b at point 4 is called definitely initialized.

Secondly, when inner-level procedures have high fan-in (i.e., number of call-sites invoking them is high), the number of distinct call-strings that reach inner-level procedures can be high. In our example, procedure f has fan-in 1, while procedures g, h, and i have fan-in 2. The number of facts at each program point in the procedures h and i is 4. This can cause inefficiency (although not as much as with the full call-strings approach). It is our hypothesis that in practice inner-level procedures would often contain simpler functionality, with higher potential for reuse, than outer-level procedures. Therefore, inner-level procedures can be expected to have higher fan-in than outer-level procedures.

B. An Alternative Proposal: Prefix Approximation

Our key proposal is to restrict context stacks to a certain number of *bottom* entries, as opposed to a certain number of *top* entries as in the suffix approach. Our motivation is to obtain better scalability on programs in which fan-in is lower at outer-levels.

Consider again the example in Figure 1. Assume 2-prefix approximation, meaning only two bottom calls in each call-string are used. The fix-point dataflow facts due to this technique are shown to the left of the control flow edges, in red colour. With this technique, the dataflow facts that reach procedures g, h and i are distinguished by only two contexts – c_1c_2 and c_1c_3, which are of length 2. Within these procedures, the data facts are propagated context insensitively. Notice that due to the lower fan-in on outer-level procedures in this example, the number of tagged dataflow facts at any point is never more than two, whereas with 2-suffix approximation it is four in procedures h and i. In other words, 2-prefix approximation is more efficient.

In order to compare the precision of the two techniques, consider the dataflow fact $\langle c_1c_2, \{a, b\}\rangle$ at point r_i. This fact is intuitively the result of the joining of facts that reach r_i along all paths that begin by going via c_1 and then c_2. This fact then reaches e_i. It is then sent back both to n_6 and to n_7 (because there is no indication in the call-string as to whether the fact came to r_i via c_6 or via c_7). Then, from n_7, this fact propagates (via e_h) to both n_4 and to n_5. From n_5 this fact reaches point 4. Here, due to the presence of variable b in the fact, the occurrence of this variable in the rhs is marked as possibly uninitialized, which is a false warning. This imprecision is due to the fact that b occurs in this dataflow fact due to paths that go through c_4 (and not c_5); therefore, ideally, the fact mentioned above should have gotten propagated only to n_4, and not to n_5. However, the prefix limited call-string is unable to make this distinction. Recall that suffix approximation gave a precise result at program point 4.

Continuing on from point 4, the fact mentioned above goes via e_g to n_2, but not to n_3. This is because the call-string of this fact ends with c_2. Therefore, the variable a in the set in this fact does *not* reach point 2. (The other fact at e_g, whose call-string is c_1c_3, reaches point 2; however, this fact indicates only variable b as being possibly uninitialized.) As a result, the occurrence of a in the rhs at point 2 is declared initialized, whereas suffix approximation declared it as possibly uninitialized. In other words, prefix approximation is more precise at outer-level procedures, whereas suffix approximation is more precise at inner-level procedures.

In programs that have good modular structure, inner-level procedures are more likely to contain generic "utility" code, while outer-level procedures are more likely to contain application-specific "business logic". In such cases, higher precision at outer-level procedures would be more valuable to users in many software-analysis scenarios than higher precision at inner-level procedures.

C. Our Contributions

- The primary contribution of this paper is the insight that prefix approximation for context sensitivity is likely to (a) be more scalable than standard suffix approximation on programs that have good modular structure, and at the same time, (b) give higher precision at outer-level procedures, which are anyway more likely to contain key application-specific logic.
- We instantiate our idea specifically in the context of the classical call-strings approach, by giving details of the changes required to this approach in order to incorporate prefix approximation. We select the call-strings approach because it is a simple approach, and is yet very general, in terms of being suited to both object-oriented and non-object-oriented languages, and to any kind of underlying "client" dataflow analysis. At the same time, we posit that our approach can be extended to other approaches to context sensitivity, such as the object sensitivity approach.
- We implement our technique of k-prefix approximation, and using a set of large, real Cobol benchmarks, compare its precision and efficiency with other variants of the call-strings approach, such as zero context sensitivity, full context sensitivity, and k-suffix approximation. Our findings, in summary, are that our technique is much more scalable than k-suffix approximation, offers somewhat lower overall precision (when entire programs are considered), but offers comparable or better precision at outer-level procedures in these programs.

The rest of this paper is organized as follows. Section II gives a technical introduction to the call-strings approach, including its suffix approximation variant. We then spell out the details of prefix approximation in Section III. We describe our implementation and experimental results in Section IV, and then conclude with a discussion of related work in Section V.

II. Interprocedural Analysis Using Call Strings

In this section, we first introduce context insensitive analysis, and then the existing variants of the call-strings context-sensitive approach. This discussion forms core background material for the understanding of our own technique of prefix approximation.

A. ICFG Representation, and Underlying Analysis

Inter-procedural analysis is typically performed on an *interprocedural control flow graph* (ICFG) representation of the program. An ICFG consists of a control-flow graph (CFG) for each procedure, which are linked by *inter-procedural call* and *return* edges. Each procedure p has an *entry* node r_p and an *exit* node e_p. Each *call site* node c_i in a procedure has a corresponding *return site* node n_i where control returns from the called procedure. An edge that transfers control from a call site c_i to the entry node r_p of the called procedure is called a *call* edge. The *corresponding return* edge of this call edge is the edge that comes back from e_p to n_i. Other than call and return edges, the other edges in the CFGs are termed as *intra-procedural* edges.

As mentioned in the introduction, any context-sensitivity approach is a wrapper around a given underlying "client" dataflow analysis. We assume that this analysis is a standard intra-procedural data flow problem $A \equiv ((D, \sqsubseteq_D), F_D)$ (e.g., constant propagation, reaching definitions). D is a join semi-lattice, while F_D is a set of transfer functions with signatures $D \rightarrow D$ associated with the intra-procedural edges. We call the elements of D "dataflow facts".

We depict and illustrate various transfer functions in Figure 2. Each of the six parts of this figure depicts a transfer function of a call or return edge, first as an illustration, and then as a formal specification (below the illustration). In the illustrations c_4, c_5, etc., denote call sites, while n_4, n_5, etc., denote the corresponding return sites. In the formal specifications, f_{c_i,r_p} denotes the transfer function of a call edge from call-site c_i to the entry node r_p of procedure p, while f_{e_p,n_i} denotes the transfer function of a return edge from an exit node e_p to a return site n_i.

B. Context Insensitive Analysis

In context insensitive analysis, elements of the underlying lattice D are used directly in the analysis, without any context tagging. Therefore, call and return edges have *identity* transfer functions.

Parts (a) and (b) of Figure 2 depict the call and return transfer functions, respectively, for the context insensitive approach. Note that in effect this approach joins all facts coming into a procedure entry r_p from all call sites, propagates this joined fact through the procedure, and then returns the fact that reaches the exit e_p to all return sites. In other words, facts are sent along feasible as well as infeasible paths, causing imprecision.

C. Full Call Strings Technique

The "full" call strings approach tags each fact from D with a call-string. Each call-string intuitively represents a stack of call-site IDs of unfinished calls.

Figure 2(c) depicts the transfer function for call edges. $\langle \gamma, d \rangle$ is the incoming tagged dataflow fact into the edge, with γ being the call-string (i.e., context tag), and $d \in D$ being the underlying fact. The transfer function outputs the same fact d, tagged with the extended call-string $\gamma.c_i$; this is basically γ with the ID of the current call-site c_i pushed on to it. This is illustrated just above the transfer function specification; in the picture the concrete call-string c_1c_3 serves as γ, and c_4 serves as c_i.

Note that we model transfer functions as returning sets, rather than individual facts. This is because in other situations (which we will encounter below) transfer functions may need to return empty sets or sets with multiple tagged dataflow facts.

In case the underlying lattice D is finite, Sharir and Pneuli [11] have shown that call-strings of length greater than $K|D|^2$ can be ignored, without any impact on the correctness or precision of the analysis, where K is the number of distinct call-sites in the program. That is, the call transfer function may return the empty set if the incoming call-string γ is already of length $K|D|^2$. For simplicity of exposition, we have omitted this check from the transfer function shown in Figure 2(c). Note that although this bound ensures termination of the analysis, the number of call-strings per program point is still likely to be so high that applicability to large programs would be unlikely.

The transfer function for a return edge $e_p \rightarrow n_i$ is depicted in Figure 2(d). Say $\langle \gamma, d \rangle$ is the tagged fact that reaches the exit node e_p. Let the topmost call-symbol in γ be c_l. If c_l is the call-site corresponding to n_i (which we denote as $\sigma(n_i) = c_l$), it follows that the transfer function ought to send the fact d back to n_i. This it does, with a tag γ' that is obtained by popping c_l from γ. On other hand, if c_l does not correspond to n_i, then, in order to enforce context sensitivity, the approach sends back no dataflow fact to n_i.

This transfer function is illustrated in the picture in Figure 2(d). In the picture $c_1c_3c_4$ serves as γ. Therefore, the fact d is sent back to n_4, with tag c_1c_3. No fact is sent back to the other return site n_5 (assume that c_5 also calls the same procedure p, of which e_p is the exit node).

We do not show the transfer functions of intra-procedural edges. These transfer functions simply use the corresponding underlying transfer functions from F_D to transform the underlying fact, without modifying the call-string component.

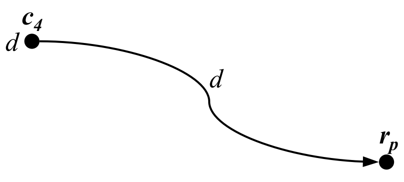
D. k-*Suffix Technique*

The full call strings analysis may not terminate if an infinite underlying lattice D is used (e.g., the constant propagation lattice). Moreover, even when D is finite, the call-string length bound of $K|D|^2$ means that a very large number of call-strings could be generated at certain program points (in the worst case, this number grows exponentially with $K|D|^2$). Therefore, for the sake of practicality, the k-suffix approximation technique has been suggested by Sharir and Pneuli, and has been used by various other researchers. This technique, basically ensures that call-string lengths are always bounded by k, where k is a user-selected parameter.

The transfer function for call edges with this technique is depicted in Figure 2(e). Basically, if the tag γ of the incoming fact into the call edge is of length less than k, then the function behaves the same as with the full technique. If not, the fact d is sent out with a new call-string, which is obtained by pushing the current call-site c_i onto γ and simultaneously dropping the bottom-most entry c_b of γ. This ensures that the length of the call-string does not increase beyond k. Implicitly, this causes a join of all facts that agree on the k-suffixes of their call strings.

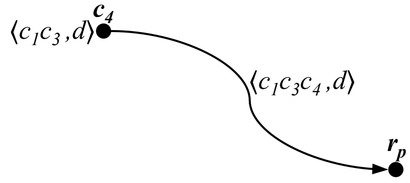
The picture in the same part of the figure illustrates this transfer function. Assuming $k = 1$, the pushing of the current call-site c_4 onto the call-string causes the bottom-most element c_3 to be dropped from the call-string.

The transfer function for the return edges (see Figure 2(f)) is similar to that used by the full call-strings technique, in that it uses the call-site symbol c_l that is on top of the call-string γ of



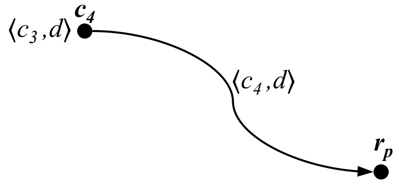
$$f_{c_i,r_p}(d) = d$$

(a)



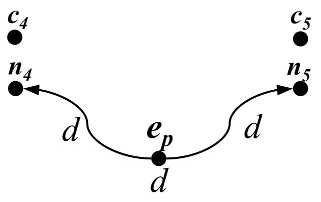
$$f_{c_i,r_p}(\langle\gamma,d\rangle) = \{\langle\gamma.c_i,d\rangle\}$$

(c)



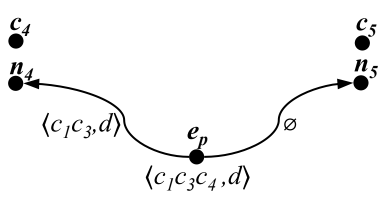
$$f_{c_i,r_p}(\langle\gamma,d\rangle) = \begin{cases} \{\langle\gamma.c_i,d\rangle\}, \text{ if } |\gamma| < k \\ \{\langle\gamma'.c_i,d\rangle \mid \gamma = c_b.\gamma'\}, \\ \quad\text{otherwise} \end{cases}$$

(e)



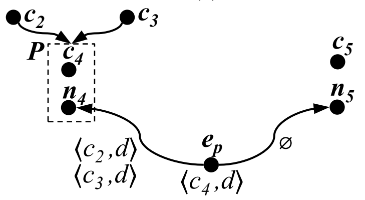
$$f_{e_p,n_i}(d) = d$$

(b)
Context-insensitive technique.



$$f_{e_p,n_i}(\langle\gamma,d\rangle) =$$
$$\{\langle\gamma',d\rangle \mid \gamma = \gamma'.c_l, \sigma(n_i) = c_l\}$$

(d)
Full context-sensitive technique.



$$f_{e_p,n_i}(\langle\gamma,d\rangle) = \begin{cases} \{\langle\gamma',d\rangle \mid \gamma = \gamma'.c_l, \\ \quad \sigma(n_i) = c_l\}, \text{if } |\gamma| < k \\ \{\langle c_f.\gamma',d\rangle \mid \gamma = c_b.\gamma'', \\ \quad \gamma = \gamma'.c_l, \\ \quad P = \mathit{ProcOf}(c_b), \\ \quad c_f \in \mathit{CallersOf}(P), \\ \quad \sigma(n_i) = c_l\}, \text{otherwise.} \end{cases}$$

(f)
k-Suffix technique.
(illustration is for k=1.)

Fig. 2. Existing call strings techniques. Transfer functions for *call* edges: (a), (c), and (e); Transfer functions for *return* edges: (b), (d), and (f).

the incoming dataflow fact into the return edge to decide which return site to send the fact d back to. However, to produce the call-string on the output side, it is not enough to pop c_l from γ. The basic reason for this is that since γ contains only a suffix of the true call stack, the output call-string cannot be allowed to become empty when active invocations still remain from which control needs to return. Let γ' represent the call-string obtained by popping c_l from γ. The technique actually returns a *set* of dataflow facts of the form $\langle c_f.\gamma', d\rangle$, where c_f is any call-site that calls the procedure that contains the bottom call-site in γ. (Note that c_f becomes the bottom entry of the call-strings of these returned facts.) In our notation $ProcOf(c_b)$ is the procedure that contains the call site c_b, while *CallersOf*(P) is the set of all call-sites that call procedure P (this would be an empty set of P is *main*).

The technique mentioned above constitutes a conservative over-approximation, because the dataflow facts that reached c_l with call-strings of the form γ' are the only possible dataflow facts that could, upon further propagation, reach e_p with the call-string $\gamma'.c_l$.

The transfer function above is illustrated in the upper part of Figure 2(f). Again, for the sake of illustration, we assume $k = 1$. In this picture, the call string associated with the fact d at e_p is c_4. Hence, the technique identifies n_4 as the correct return site, and pops c_4 from the call string (thus leaving it empty). Say c_2 and c_3 are call-sites such that they call the procedure P that contains c_4. Therefore, these are the candidate call-sites that can be inserted into the bottom of the call-string to be returned. Therefore two tagged facts – $\langle c_2, d\rangle$ and $\langle c_3, d\rangle$ – are returned to n_4.

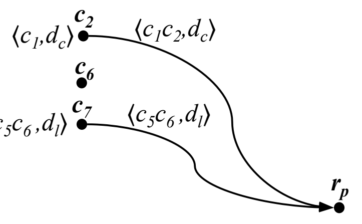
For a more complete illustration of the suffix approximated call-strings approach, we refer readers to the fix-point solution shown to the right of the control-flow edges in Figure 1. This solution was already discussed in Section I-A.

III. Prefix Approximation Call Strings Approach

In this section we describe in detail the transfer functions for our prefix approximation technique for the call-strings approach. These transfer functions are specified assuming that a prefix length limit k is specified by the user.

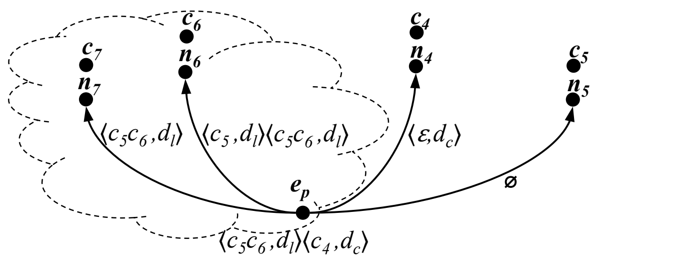
A. Transfer Function for Call Edges

Figure 3(a) shows our transfer function for call edges, as well as an illustration of this transfer function (which appears above the transfer function specification). Basically, if the length of call-string γ of the input fact is less than *k*, then we append the current call site c_i as usual to the call-string of the outgoing fact. That is, the analysis remains in a context-sensitive mode. However, if the length of the call-string γ is equal to *k*, then the analysis is in a context insensitive mode, and needs to remain in it. Therefore, the current call-site c_i is not appended to the outgoing call-string. Note that the mode switch happens whenever the length of a call-string γ becomes equals to *k* for the first time.


c_2
$\langle c_1,d_c\rangle$
$\langle c_1c_2,d_c\rangle$
c_6
c_7
$\langle c_5c_6,d_l\rangle$
$\langle c_5c_6,d_l\rangle$
r_p

$$f_{c_i,r_p}(\langle\gamma,d\rangle) = \begin{cases}\{\langle\gamma.c_i,d\rangle\}, & \text{if } |\gamma| < k\\ \{\langle\gamma,d\rangle\}, & \text{Otherwise}\end{cases}$$

(a)


c_6
n_6
c_4
n_4
c_7
n_7
c_5
n_5
$\langle c_5c_6,d_l\rangle$
$\langle c_5,d_l\rangle\langle c_5c_6,d_l\rangle$
$\langle \varepsilon,d_c\rangle$
e_p
$\varnothing$
$\langle c_5c_6,d_l\rangle\langle c_4,d_c\rangle$

$$f_{e_p,n_i}(\langle\gamma,d\rangle) = \begin{cases}\{\langle\gamma',d\rangle \mid \gamma=\gamma'.c_l, \sigma(n_i)=c_l\}, \ \text{if } |\gamma| < k\\ \begin{cases}\{\langle\gamma,d\rangle \mid \gamma=\gamma'.c_l, \sigma(n_i)=c_i,\\ \qquad c_i \in \mathit{Region}(c_l)\}\\ \qquad \cup\\ \{\langle\gamma',d\rangle \mid \gamma=\gamma'.c_l, \sigma(n_i)=c_l\},\end{cases}\\ \quad \text{otherwise.}\end{cases}$$

(b)

Fig. 3. Transfer functions for (a) *call* edges, and (b) *return* edges. Illustrations are with $k = 2$.

In the illustration that appears in Figure 3(a), say $k = 2$. Therefore, along the edge $c_2 \rightarrow r_p$, the call-string c_1 of the incoming dataflow fact into the call edge is extended by having c_2 pushed on to it. However, along the edge $c_7 \rightarrow r_p$, since the incoming fact has a call-string c_5c_6, whose length is 2, this call-string is sent out unchanged in the outgoing fact.

B. Transfer Function for Return Edges

We now discuss our transfer function for return edges, which is more complex. This transfer function is depicted in Figure 3(b). In the most straight forward scenario, the length of call string γ is less than *k*. That is, the analysis is currently in the context sensitive mode. In this scenario, the transfer function is exactly the same as in the full call strings approach. That is, if the topmost call-site c_l in the call-string γ of the incoming dataflow fact into the return edge corresponds to the return site n_i that is at the target of this return edge, then the call-string γ' for the outgoing dataflow fact that is sent to n_i is obtained by popping c_l from γ; otherwise, no fact is sent to n_i. This logic is specified in the first part of the transfer function specification in Figure 3(b) (the part that is controlled by the "if" condition).

In the illustration that appears above the transfer function specification, the dataflow fact $\langle c_4, d_c\rangle$ matches the scenario mentioned above. It is sent back only to return site n_4, with call-string ϵ.

We now discuss the case where the call-string γ of the incoming dataflow fact into the return edge is of length *k*, meaning the analysis is currently in the context insensitive mode. There are two possible scenarios while in this mode.

1) Scenario 1: c_l is the call-site corresponding to n_i. That is, $c_l = \sigma(n_i)$.

In this case, as usual, the transfer function pops c_l from γ, and returns the resultant call-string γ' along the edge to n_i. This case is covered by the last line in the transfer function specification (which is the second argument of the '$\cup$' operator). It is illustrated in the picture by the dataflow fact $\langle c_5c_6, d_l\rangle$ being propagated to n_6 as $\langle c_5, d_l\rangle$.

This does not suffice. It is possible that procedure p (of which e_p is the exit node) either contains c_l (i.e., is self-recursive), or calls (directly or transitively) the procedure that contains c_l (i.e., mutual recursion). In these cases, the fact $\langle\gamma, d\rangle$ at e_p could have actually resulted from a call-sequence that visited c_l multiple times, with some of these visits not resulting in pushes of c_l onto γ (due to the k limit). When this happens, c_l should *not* be popped from γ on the outgoing call-string to n_i (because c_l does not represent the topmost unfinished call). Therefore, $\langle\gamma, d\rangle$ itself needs to be propagated back to n_i. This case is covered by the first argument of the '$\cup$' operator in the transfer function, where $\mathit{Region}(c_l)$ denotes the procedures in the program that are reachable directly or transitively from c_l (without returning via the corresponding return site n_l). Since c_l is the same as $c_i \equiv \sigma(n_i)$ (as per the assumption of *Scenario 1*), and since c_i calls p (which follows from the presence of the edge from e_p to n_i), the test $c_i \in \mathit{Region}(c_l)$ in the transfer function actually implies that procedure p either contains c_l, or calls (directly or transitively) the procedure that contains c_l.

The situation discussed above is illustrated in the picture by the dataflow fact $\langle c_5c_6, d_l\rangle$ that gets propagated as-is to the return site n_6. Note that this happens *in addition* to this fact being sent to n_6 with tag c_5 (as was mentioned earlier, this happens due to the second argument of the '$\cup$' operator). Sending both these facts to n_6 is required, because even in the presence of mutual recursion there would exist a path that reaches e_p via a single traversal via c_l, in which setting c_l needs to be popped from the call-string. In the picture the cloud represents $\mathit{Region}(c_6)$.

2) Scenario 2: c_l is not the call-site corresponding to n_i. That is, $c_l \neq \sigma(n_i)$.

In this case, the transfer function checks whether c_i is present in $\mathit{Region}(c_l)$, where c_i is $\sigma(n_i)$. If yes, it means that some call sequence brought control to p via c_l, and γ is a prefix of this call-sequence. In this case, since c_i was not pushed onto γ earlier, the fact $\langle\gamma, d\rangle$ is returned as-is to n_i. This case is also covered by the first argument of the '$\cup$' operator. This case is illustrated in the picture by $\langle c_5c_6, d_l\rangle$ being propagated as-is to return site n_7.

C. Variable Length Prefixes

So far in this section we have discussed prefix approximation with an apriori length limit of k on the prefixes. An alternative way of using prefix approximation would be for the user of the analysis to identify a set of call-sites K_P such that they are willing to tolerate context-insensitive analysis within the regions of these call sites. For instance, if there is a library that is expensive to analyze (e.g., due to high fan-in on procedures within the library), such that analysis results within this library are not of interest to the user and such that context-insensitive analysis within this library would not impact precision in the other parts of the programs too much, then the set of call-sites to the procedures in this library could be used as K_P.

Our prefix approximation technique can be used in this scenario with the following (simple) modification to the transfer functions that were shown in Figure 3: the check "if $|\gamma| < k$" in both transfer functions be replaced with the check "if $\gamma = \gamma'.c_l \wedge c_l \notin K_P$". Note that with this variant, in the presence of recursion, termination would need to be guaranteed by other means, such as by collapsing cycles in the call-graph, or by using a call-string length bound of $K|D|^2$.

D. Soundness of Our Technique, and Other Discussions

Our prefix approximation technique is *sound*, in that in the fix-point solution, the fact (from D) that is computed for each program point is an over-approximation of (i.e., dominates) the fact that would have been computed at that point had full context sensitivity been used. We omit a detailed proof of this due to lack of space. The key property that is shown in the proof is about the return transfer function. The property is that if at all there was a call-sequence that was traversed during the analysis and that entered procedure p, such that γ is a prefix of this call sequence, and such that the topmost entry of this call sequence is $\sigma(n_i)$, then any fact of the form $\langle\gamma, d\rangle$ at e_p would definitely get propagated back to n_i. That is, no propagations that are required along return edges are missed.

Sharir and Pnueli, in their original paper [11] on the call-strings approach, have given a generic framework for creating sound approximations for this approach (Section 7-6). As an instance of this framework, they have suggested k-suffix approximation, but have not explicitly suggest prefix approximations. It is future work for us to see if the technique that we have proposed in this paper is an instance of their generic framework. If it is, then the soundness of our technique also follows from their proof of soundness of their generic framework.

It is easy to see that due to the limit on the lengths of call-strings, our approach always terminates (whenever the underlying analysis is terminating).

Finally, it is straightforward to combine our prefix approximation with k-suffix approximation. In the interest of space we omit a specification of the combined transfer functions.

IV. Implementation and Evaluation

A. Implementation

We have implemented the call-strings approach in conjunction with our k-prefix approximation, the context insensitive analysis, the full call strings approach, as well as the call-strings approach with k-suffix approximation. (We have not evaluated the variable length prefix technique that we mentioned in Section III-C, primarily because an evaluation of that technique would require user knowledge about the benchmark programs that are to be analyzed.) Our implementation is targeted at analyzing Cobol programs. Cobol programs are very prevalent in large enterprises [14]. Another motivating factor for this choice is that one of the authors of this paper has extensive professional experience with developing and maintaining Cobol applications. We have implemented our analyses on top of a proprietary program analysis framework *Prism* [15]. This framework provides the parser for Cobol, internal representation (IR), and ICFG. Our implementation is in Java. We have also implemented possibly-uninitialized variables analysis [13] as a client analyses. We performed our experiments on a 64-bit Windows desktop with an Intel i5 processor and 8 GB RAM.

B. Benchmark Programs

We have selected a set of seven large, proprietary programs, which are being used in different financial institutions, as benchmarks for evaluation. These programs all implement banking-related functionalities such as payment processing, transaction posting, and account settlement. Key statistics about these programs are provided in Table I. The meanings of columns (b), (c), and (f) are self-explanatory. In legacy Cobol programs almost all variables tend to be declared as global variables. Column (d) depicts the number of declared global variables in the programs. Note the extremely high numbers of variables in these programs, which is idiomatic of legacy Cobol programs. This poses a major challenge to efficient analysis, because each dataflow fact at each program point needs to capture information about some or all of these variables. A *variable reference* in a program is an occurrence of a variable in an expression anywhere in the program. Column (e) depicts the total number of variables references in each program.

Columns (g)-(i) provide statistics about the complexity of context-sensitive analysis. This complexity is typically a function not only of the program's structure, but also of the underlying client analysis that is chosen. Therefore, in order to report these numbers in a manner that is as independent of the client analysis as possible, we implemented a very simple *reachability* analysis. In this analysis the underlying lattice values are 0 (unreachable) and 1 (reachable), with 1 dominating 0. The numbers in columns (g)-(i) were obtained using a full call-strings approach that was wrapped around this analysis. Column (g) depicts the total number of ⟨call-string, fact⟩ pairs across all program points. Column (h) depicts the length of the longest call-string that was ever generated. Column (i) depicts the maximum number of ⟨call-string, fact⟩ pairs at any single program point. All these numbers were determined after the analysis reached a fix-point. These numbers give a feel for the high complexity of context-sensitive analysis.

We use our client analysis (possibly uninitialized variables) to compare the time requirement, memory requirement, as well as precision of the four analysis variants that were mentioned at the beginning of this section. For each of these programs, we choose apriori a value of k, and use this value of k with both the k-suffix as well as the k-prefix techniques. We chose $k = 2$ for Prog2 and Prog7, whereas we chose $k = 4$ for all

TABLE I. BENCHMARK PROGRAM DETAILS.

Program name	Total LOC	No. of CFG nodes	No. of program variables	No. of variable references	No. of call sites	Total facts	Max. call string length	Max. facts at any node
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
Prog1	9017	3353	2275	1365	299	29211	10	395
Prog2	16080	6857	4997	3110	238	15874	6	219
Prog3	17826	9376	5048	4014	518	57704	21	608
Prog4	22766	12044	4263	3724	615	760823	22	13073
Prog5	23183	13039	10115	4130	633	454595	10	25846
Prog6	37151	11999	8272	4860	562	158419	24	1484
Prog7	49857	34092	31276	4328	523	67007	5	1893

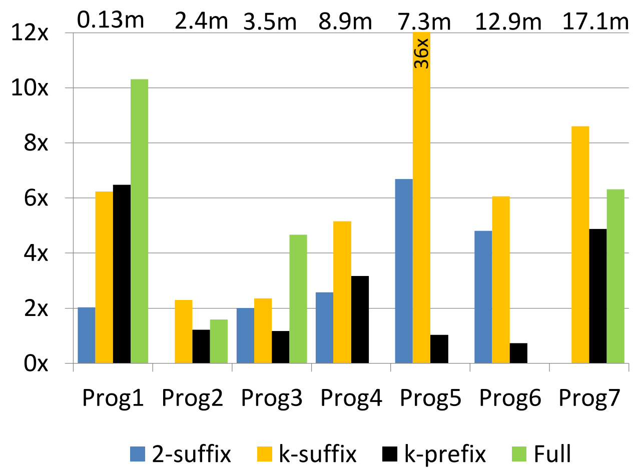


Fig. 4. Normalized running time, relative to context insensitive analysis.

other programs (because they had a higher value of maximum call sequence depth). In all our figures, whenever we use the labels "k-suffix" or "k-prefix" against a benchmark program, we refer to the chosen value of k for that program.

C. Running Time

The running time requirements of the four analysis variants are summarized in Figure 4. For each benchmark program this figure depicts the normalized running times of 2-suffix, k-suffix, k-prefix, and full context sensitive analysis, relative to the running time of context-insensitive analysis on that program. We evaluated 2-suffix approximation also on the programs for which we chose $k = 4$, because 2-suffix approximation is likely to be more efficient than 4-suffix approximation. For Prog5, the k-suffix running time is actually 36x the context insensitive running time; however, in the figure, we cut the bar off at 12x to save space. At the top of the figure, for each program, we mention the absolute running time of the context-insensitive analysis, in minutes.

The full context sensitive approach could not scale up to analyze programs Prog4, Prog5, and Prog6. We killed the analysis on these programs after it had run for nearly 15 hours.

Our observations about the running times of the various analyses are as follows:

- k-prefix approximation is very efficient. On four programs, namely, Prog2, Prog3, Prog5, and Prog6, this variant took time that was similar to or even less than that of context insensitive analysis. The worst it did was on the smallest program, Prog1, where it took 6x the time as context insensitive analysis.
- The k-suffix and 2-suffix techniques are *much* less efficient. Not counting Prog1, the k-suffix technique took between 1.6x to 36x the time of the k-prefix technique. Not counting Prog1 and Prog4, the 2-suffix technique took between 1.7x to 7x the time of the k-prefix technique.

D. Memory Requirement

We measured the memory consumption of the analyses at the point of time when the fix-point solution was just reached. Due to space constraints we do not depict the full results. The k-prefix technique was much more memory efficient than the suffix approximated variants. Not counting Prog1, 2-suffix approximation consumed between 1x to 3.5x as much memory as the k-prefix technique, while k-suffix approximation consumed between 1.4x to 9.2x as much memory as the k-prefix technique. On the program that needed the most memory for analysis, namely, Prog5, the context-insensitive, 2-suffix, k-suffix, and k-prefix techniques consumed 219 MB, 1226 MB, 3504 MB, and 350 MB, respectively.

E. Precision

We now discuss the precision of the different analysis variants. Our objective was to measure not only overall precision, but also precision separately at outer-level and inner-level procedures. Therefore, for each program, we first classified its procedures as outer-level procedures or inner-level procedures. Ideally, this should have been done in a program-specific manner, using human expertise. However, that would have required extensive manual effort. Therefore, for each program, we characterized each procedure that was at a maximum call-depth of k or less from the main procedure as an outer-level procedure (this k-value being program specific), and characterized all other procedures as inner-level procedures.

Figure 5 summarizes the precision results. Each bar in the figure represents the results from an analysis variant, namely, context insensitive (C.I.), k-suffix, k-prefix, or full context sensitive, on a benchmark program. The number on top of each bar indicates the percentage of all variable references in the

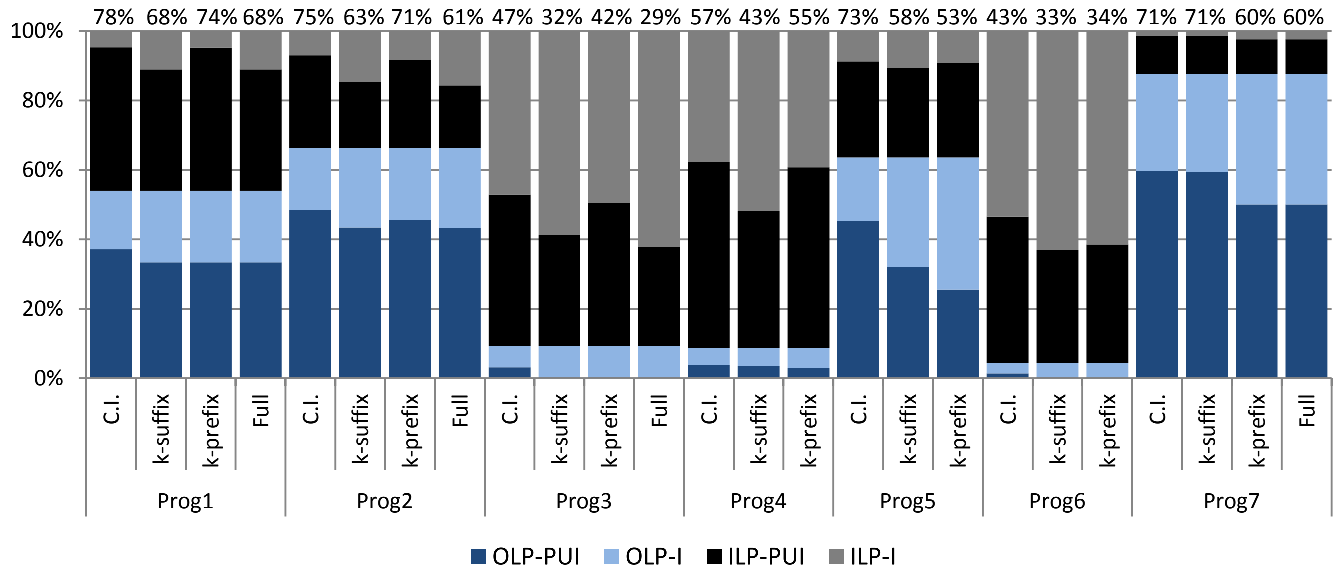


Fig. 5. %-age of variable references reported as possibly uninitialized (-PUI) or initialized (-I), for outer-level procedures (OLP) and inner-level procedures (ILP).

program that have been identified as possibly uninitialized[1]. Here, lower numbers indicate higher precision. Each bar is further split into four parts, as follows: "OLP-PUI" indicates the percentage of all variable references that are in outer-level procedures and that are identified as possibly uninitialized. "OLP-I" indicates the percentage of all variable references that are in outer-level procedures and that are identified as definitely initialized. "ILP-PUI" and "ILP-I" indicate analogous numbers, but with respect to inner-level procedures.

Our observations on the precision of the different techniques are as follows:

- k-prefix approximation resulted in equal or better precision than k-suffix approximation at outer-level procedures in all the programs except Prog2. In fact, in the case of Prog5 and Prog7, k-prefix approximation did significantly better, not only on outer-level procedures, but even overall.
- On the other hand, k-suffix approximation gave better overall precision than k-prefix approximation on the five programs other than Prog5 and Prog7. In particular, on four of these five programs (i.e., the ones excluding Prog6), the overall precision of k-prefix approximation was closer to that of the context insensitive approach than to that of k-suffix approximation. Among these four programs, on Prog1 and Prog4, k-prefix approximation also took significantly more running time than the context insensitive analysis.
- Prog3, Prog4 and Prog6 have very small numbers of variable references in outer-level procedures. This is essentially because of the deep call sequences in these programs, and also because outer-level procedures are small in size and have low fan-out. Ideally, in order to gain more precision, our technique should be applied with higher values of k on these programs. However, due to certain inefficiencies in our current implementation (especially regarding the check "$c_i \in Region(c_l)$" that was shown in Figure 3(b)), our technique currently does not scale very well to higher values of k.
- Due to lack of space, we have not explicitly depicted the precision of the 2-suffix technique in Figure 5. Surprisingly, 2-suffix approximation gave similar precision as 4-suffix approximation on four programs, and did somewhat worse only on Prog3 (where it reported 38% of all variable references as possibly uninitialized).
- The percentage of variable references reported as possibly uninitialized is unfortunately high with all of the variants. There are several reasons for this, including that we perform a standard path-insensitive analysis, and handle references to arrays and calls to unavailable external routines (which are both prevalent in Cobol programs) very conservatively. Note that our primary objective was to compare the four variants using the same client analysis, and not necessarily to engineer a very precise client analysis.
- Full context sensitivity reduces the number of reported uninitialized variables by around 13% (on Prog1) to around 38% (on Prog3) when compared to the results from context insensitive analysis. This is a significant gain, but is less than our own expectations. However, our result is generally in agreement with the detailed comparisons of context sensitive analysis vs. context insensitive analysis done by Lhotak et al. [10]. In fact, in their results, when call-graph construction and virtual-call resolution were used as clients, the improvement due to context sensitivity was less than 10% on most benchmarks. With yet

[1]In Cobol, each variable corresponds to a memory range. We report a variable reference as possibly uninitialized if any portion of the range corresponding to this reference is possibly uninitialized.

another client analysis that they evaluated, namely, cast safety, the improvement was somewhat more significant (extending to about 65% in one benchmark).

F. Takeaways

k-prefix approximation is much more efficient than the suffix approximation technique, both in terms of running time and in terms of memory consumption. Its precision on outer-level procedures was equal to or better than that of suffix approximation on six out of seven benchmarks. On programs that do not have very deep calling sequences, k-prefix approximation is overall a better performer. This is essentially because the procedures on which it gives better precision, namely procedures at a maximum call-depth of k or less, account for a large proportion of all the variable references in these programs.

V. Related Work

While suffix approximations were suggested in Sharir and Pneuli's original paper [11] itself, prefix approximations have not been explored significantly in the literature. The existing idea of analyzing regions of the call-graph that form SCCs (strongly-connected components) context insensitively [16], [17] can be thought of as a specific kind of prefix approximation. However, the objective in the two approaches mentioned above was to ensure termination, and not to use prefix approximation in general and flexible ways to make an analysis efficient while minimizing the loss of precision. The prefix approximation we propose is flexible, in that certain lower layers in a program or certain regions of the program can be treated context insensitively even in the absence of recursion. In summary, SCC collapsing is not an alternative to general prefix approximation (or suffix approximation).

It is noteworthy that in addition to collapsing SCCs, Souter et al. [16] also mention the potential to use general prefix approximation in the context of their approach. However, their approach is specifically formulated only to compute context-aware def-use information in inter-procedural programs. In contrast, we show how to incorporate prefix approximation in the generic call-strings approach, which therefore can be applied in conjunction with *any* given client analysis. Correspondingly, the changes we make to the transfer functions of the call-strings approach are very different from the changes they incorporate to their original (approximation-free) approach in order to achieve prefix approximation.

We have shown an instantiation of our prefix approximation idea for the call-strings approach. The reasons for our choice of the call-strings approach are that it is a classical approach, yet simple, flexible, and general. It is simple in that it directly wraps around a given underlying intra-procedural analysis, without requiring any changes to the lattice or transfer functions of the underlying analysis. It is flexible in that it easily admits varied approximation schemes, such as ours. It also flexible in that it readily extends to languages (such as Cobol) with overlapping procedures, procedures with return- as well as fall-through exits, etc. It is general, in that it places almost no restrictions on the form of the underlying analysis. This said, there exist other generic approaches to inter-procedural analysis such as the functional approach of Sharir and Pneuli, the IFDS approach [13], the object sensitivity approach [12], and the modified call-strings approach of Khedker et al. [18]. From among these approaches, we hypothesize that prefix approximation could be integrated into the object sensitivity approach in a natural way, and could give interesting results in practice. It is future work for us to explore this extension in-depth.

References

[1] S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[2] S. Blazy and P. Facon, "SFAC, a tool for program comprehension by specialization," in *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, nov 1994, pp. 162 –167.

[3] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field, "Dependent types for program understanding," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 157–173.

[4] J. Lim, T. Reps, and B. Liblit, "Extracting output formats from executables," in *Working Conf. on Reverse Engg. (WCRE)*, 2006, pp. 167–178.

[5] R. Komondoor and G. Ramalingam, "Recovering data models via guarded dependences," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, 2007, pp. 110–119.

[6] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proc. ACM Conf. on Computer and Communications Security (CCS)*, 2008, pp. 391–402.

[7] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.

[8] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005.

[9] F. Martin, "Experimental comparison of call string and functional approaches to interprocedural analysis," in *Compiler Construction*. Springer, 1999, pp. 63–75.

[10] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *Compiler Construction*. Springer, 2006, pp. 47–64.

[11] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Application*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall Professional Technical Reference, 1981.

[12] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[13] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, 1995, pp. 49–61.

[14] "Brain drain: Where Cobol systems go from here," *ComputerWorld*, May 21 2012, http://www.computerworld.com/s/article/9227263.

[15] S. Khare, S. Saraswat, and S. Kumar, "Static program analysis of large embedded code base: an experience," in *Proc. India Software Engg. Conf. (ISEC)*, 2011.

[16] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object-oriented systems," *Software Engineering, IEEE Transactions on*, vol. 29, no. 11, pp. 1005–1018, 2003.

[17] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. ACM Conf. on Prog. Langs. Design and Impl. (PLDI)*, 2004, pp. 131–144.

[18] U. P. Khedker and B. Karkare, "Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method," in *Compiler Construction*. Springer, 2008, pp. 213–228.