Summarizing Calling Contexts

Prathmesh Prabhu, Thomas Reps, Ben Liblit
University of Wisconsin-Madison

1 Abstract

Function summaries are one way of constructing dataflow transformers across function calls. We look at the complimentary problem of deriving the effect of calling context on functions. We develop a generic method for analyzing information flow from program exit to the program entry through the calling context and implement affine-relation analysis in this framework.

2 Introduction

Library developers often have to deal with client compatibility issues across versions of a library that are identical in their API specification. The reason for this can often be traced to some unspecified assumptions made by the client about the behaviour of exported library functions. Earlier work in this direction includes [4] . One could partially address this problem by analyzing the data flow across the context at which these library functions are called, i.e., through a summary of the flow of data from the function returns back to the function entry. As a first step in this direction, we present a generic method to implement a class of data flow analyses summarizing information flow across contexts.
The paper is organized as follows. Section 3 defines function summaries and Weighted Push Down Systems (WPDSs) and recalls the program transformation used to model program paths by runs of the WPDS and obtain function summaries. Section 4 proposes the notion of Context Summaries and in section 4.1 describes the alternative Control Flow Graph (CFG) to WPDS transformation that is the main result of this paper. Section 5 describes the implementation of our tool and section 6 contains some experimental results. Finally, section 7 concludes with some ideas for future work.

3 Background

Function summaries are a well studied way of abstracting the effect of procedures on dataflow facts across function calls. Given a dataflow problem with dataflow facts from domain $\mathcal{D}$, a summary for a procedure p is a function $f : \mathcal{P}(\mathcal{D}) \longrightarrow \mathcal{P}(\mathcal{D})$ that maps the set of facts that hold at the entry of the function to those that hold at the end.
Weighted Push Down Systems have been used by Reps et al. for inter-procedural analysis of programs for a wide range of dataflow problems [7]. Intuitively, A PDS is a finite transition system with an associated infinite stack with finite stack symbols. The transitions of this system are specified by the PDS rules that read the current state of the transition system and the top of the stack to decide the next state of the transition system. A transition may replace the top of stack with zero or more stack symbols. Additionally, a *weighted* PDS has weights associated with each rule, where these weights come from a bounded idempotent semiring domain.
Formally, a WPDS $\mathcal{Q} = \{\mathcal{P}, \Gamma, \mathcal{W}, \mathcal{R}\}$ is defined by the set of states $\mathcal{P}$, the set of stack symbols Γ, the weight domain $\mathcal{W}$ (a bounded idempotent semiring $(\mathcal{D}, \oplus, \otimes, 1, 0)$) and the PDS rules $\mathcal{R}$. We use the following notation for a general WPDS rule:

$$(i, s) \xrightarrow{w} (j, b \cdot c)$$

Here, i is the input state and s the top of stack, while j is the resulting state. The stack is modified as follows. First s is popped off the stack. Then c is pushed onto the stack. Finally b is pushed. Both b and c are optional. A rule with no stack symbols on the right-hand side simply pops the top of the stack.

3.1 Function summaries using WPDSs

The following translation from the CFG of a program to a WPDS from [7] encodes the interprocedurally valid paths in the CFG as runs of the WPDS. The WPDS consists of only one state called "prog". The stack symbols correspond to the CFG nodes. The weight domain for a given dataflow analysis is the set of abstract transformers for the involved dataflow facts, i.e., $w \in \mathcal{W}$ is a function $w : \mathcal{P}(\mathcal{D}) \longrightarrow \mathcal{P}(\mathcal{D})$. Finally, the PDS rules are defined as below.
For every intra-procedural edge $n \rightarrow m$,between nodes n and m, with the effect of the program statement captured by the weight function w, add the rule:

$$(prog, n) \xrightarrow{w} (prog, m)$$

For every call site n to the entry node s_p of procedure p and the matching return site m with the effect of the call itself captured by w, add

$$(prog, n) \xrightarrow{w} (prog, s_p \cdot m) \tag{1}$$

For every procedure p with end node e_p, add

$$(prog, e_p) \xrightarrow{1} (prog, \cdot)$$

Function summaries from WPDS runs
A configuration of a PDS is defined as the tuple (p, ζ), where p is a PDS state and ζ is a snapshot of the stack. We write $(p, \zeta) \Longrightarrow (q, \zeta^{'})$ if $\zeta = \gamma \cdot w$, $\zeta^{'} = w^{'} \cdot w$, $\gamma \in \Gamma$ and there exists a rule $r \in \mathcal{R}$, $(p, \gamma) \xrightarrow{v} (q, w^{'})$. $\Longrightarrow^*$ is the transitive closure of$\Longrightarrow$. For a set of PDS configurations $\mathcal{C} := \{(p, \zeta) | p \in \mathcal{P} \wedge \zeta \in \mathcal{Z}\}$ with $\mathcal{Z}$ a regular language over Γ, the Pre* and Post* operations developed in [3][7] can be used to obtain the (regular) language of configurations $\{(q, \zeta^{'}) | (q, \zeta^{'}) \Longrightarrow (p, \zeta) \wedge (p, \zeta) \in \mathcal{C}\}$ and $\{(q, \zeta^{'}) | (p, \zeta) \Longrightarrow (q, \zeta^{'}) \wedge (p, \zeta) \in \mathcal{C}\}$, respectively. Furthermore, for WPDSs, the Meet Over all Paths transformer from the starting configurations to any reachable configuration can be obtained via the so-called *path summary* operation on the accepting weighted automaton. The function summary for a procedure *proc* can be obtained by computing the Post* with initial configuration $(prog, s_{proc} \cdot \tau)$ where s_{proc} is the start state of proc and τ is a special end marker. The abstract transformer for the procedure is then obtained from the reachable configuration $(prog, \tau)$.

4 Context Summaries

A context summary for a procedure is defined as a function $f : \mathcal{P}(\mathcal{D}) \longrightarrow \mathcal{P}(\mathcal{D})$ that maps every set of dataflow facts at the procedure exit e_p to the meet of the set of values that can be obtained via an interprocedurally valid path that starts at e_p and ends at s_p.

Let $n \rightarrow m$ denote an (interprocedural / intraprocedural) edge between the nodes n and m in the CFG and $n \rightarrow^* m$ denote an interprocedurally valid path from n to m. Let $w(n, m)$ denote the weight that models the effects of the statement flow on the data facts along a path from n to m. Then we can obtain the weight for a valid path $n \rightarrow^* p$ as,

$$w(n, p) = w(n, m) \otimes w(m, p) such that n \rightarrow^* m \wedge m \rightarrow p$$

Now, the summary weight across all contexts f for a procedure p can be obtained by taking the meet over all unmatched paths where unmatched paths are defined by a context free grammer as below.
$um = m \cdot um \cdot m \mid)_i \cdot um \cdot (_i \mid \epsilon$
$m = m \cdot m \mid (_i \cdot m \cdot)_i \mid e \mid \epsilon$
Here, um denotes the unmatched paths of interest, m denotes matched paths of classical inter-procedural analysis, e is an expression, $(_i$ and $)_i$ are call and the corresponding return node. Now the required summary weight is

$$f = \oplus \{w(p)\} \cdot p \in um[e_p, s_p]$$

where $um[e_p, s_p]$ is the set of unmatched paths starting in e_p and ending in s_p.

Context summary: example

An example program is given in Figure 1. Since this program contains a recursive function, the call graph

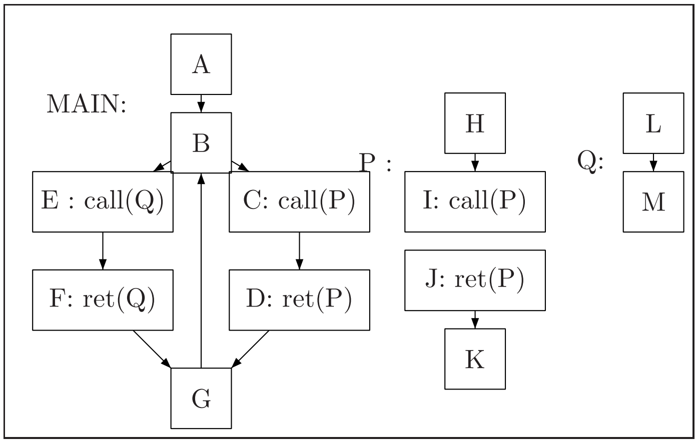

Figure 1: Example program

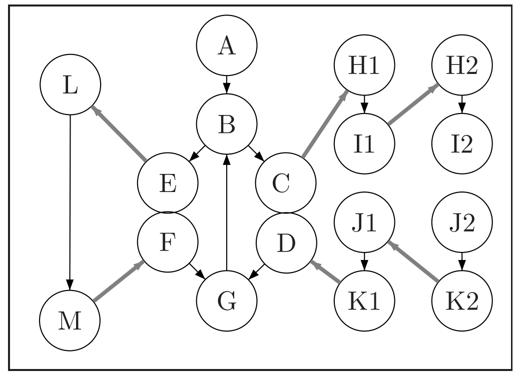

Figure 2: The unrolled program

can be unrolled by placing a copy of the function f for every call site resulting into the infinite transition system shown in Figure 2. In summarizing all context paths for the procedure P, we must consider all paths starting at K and ending at H. In particular, we must take the meet of all paths with the first unmatched return site F, D, and J and the last matching call site E, C, and I correspondingly. Note that there are an infinite number of such paths for each of these pairs. The context summary is the meet of the weights of the paths:
$K1 \to D \to G \to B \to C \to H1 \oplus$
$K2 \to J1 \to K1 \to D \to G \to B \to C \to H1 \to I1 \to H2 \oplus$
$K1 \to D \to G \to B \to E \to L \to M \to F \to G \to B \to C \to H1 \oplus \ldots$
and so on.

4.1 Context Summaries using WPDSs

We now describe a CFG to WPDS transformation similar to the one in 3.1. The aim is to obtain a WPDS from the CFG such that a Post* query on the WPDS can be used to summarize exactly the paths described above. Let the WPDS be $\mathcal{Q} = \{\mathcal{P}, \Gamma, \mathcal{W}, \mathcal{R}\}$. Let *CFGNODES* denote the set of CFG nodes, *CALLSITES* the set of call sites and *PROCS* the set of procedures. Further, *SUCC(p)* denotes the intraprocedural successors of *p*. s_p & e_p are the start and end nodes of procedure *p*. *end* is a special marker.

States

$\mathcal{P} = \{up, down\}$
The state *up* denotes that the last interprocedural edge encountered in the path was an unmatched return node and a matching call node should be found for it. state *down* denotes the part where the last interprocedural edge encountered was a call node and a matching return node must be found for it.

Stack symbols

$\Gamma = \{p^{down} | p \in CFGNODES\} \bigcup$
$\{p^{q} | p \in CFGNODES \wedge q \in CALLSITES \bigcup end\} \bigcup$
$\{end^{end}\}$
The *down* superscripted stack symbols appear with the *down* state while a stack symbol superscripted *p* denotes that the last unmatched return node corresponds to the call node *p*.

Intra-procedural rules:	
$\forall \{p,q\} \in CFGNODES - CALLSITES \wedge q \in SUCC(p) \wedge r \in CALLSITES \wedge p \neq r$:	
$(down, p^{down}) \xrightarrow{w(p,q)} (down, q^{down})$	(1.1)
$(up, p^{r}) \xrightarrow{w(p,q)} (up, q^{r})$	(1.2)
call rules:	
$\forall p,q \in CFGNODES - CALLSITES \wedge q \in SUCC(p) \wedge p \in CALLSITES(f) \wedge r \in CALLSITES \wedge p \neq r$:	
$(up, p^{r}) \xrightarrow{w(p,s_f)} (down, s_f^{down} \cdot q^{r})$	(2.1)
$(down, p^{down}) \xrightarrow{w(p,s_f)} (down, s_f^{down} \cdot q^{down})$	(2.2)
return rules:	
$\{(down, e_f^{down}) \xrightarrow{1} (down, \epsilon)\}$	(3.1)
reverse-call rules:	
$\forall \{r,p\} \in CALLSITES \wedge q \in SUCC(p) \wedge p \in CALLSITES(f) \wedge f \neq main$	
$(up, e_f^{r}) \xrightarrow{w(e_f,r)} (up, q^{p} \cdot s_f^{r})$	(4.1)
$(up, e_{main}^{r}) \xrightarrow{1} (up, \cdot)$	(4.2)
reverse-return rules:	
$\forall p \in CALLSITES$	
$(up, p^{p}) \xrightarrow{1} (up, \cdot)$	(5.1)
switch state:	
$\forall p \in CFGNODES \wedge r \in CALLSITES$	
$(down, p^{r}) \xrightarrow{1} (up, p^{r})$	(6.1)
init rules:	
$\forall p \in CALLSITES(f) \wedge q \in SUCC(p)$	
$(up, e_f^{end}) \xrightarrow{1} (up, q^{p} \cdot end^{end})$	(6.2)

Figure 3: WPDS rules for context summarization

Weight domain

Same as before.

Rules

The rules have been broken down into several categories for easier understanding in Figure 3. Rules (1.1) and (1.2) extend the paths along intraprocedural edges. (2.1) and (2.2) simulate a procedure call by pushing the return node for this call behind the start node of the called function; the pushed node has the same superscript as the current top, while the entry symbol gets the superscript *down*. This ensures that when we have traversed the side branch, we continue with the correct superscript for the remaining traversal. (4.1) comes into effect at the exit of a function in *up* state. We push the *start of current function with the current superscript* behind *any* return node for this function with the *corresponding* call site as the superscript. This along with (5.1) works exactly like the call-return rules in equation (1) but in reverse. For *main*, we simply pop off the return node in (4.2). (6.2) is the initialization rule.

Query

The context summary for a procedure f can now be obtained in a way similar to function summaries. We

S No.	Program/file name	Test case size	# of ARs discovered
1	fib1.c (prog)	19 src / 73 asm	5
2	multicall.c (prog)	22 src / 61 asm	4
3	multifunc.c (prog)	39 src / 140 asm	18
4	libprime.c (OSS)	89 src / 350 asm	2
5	formelparser_grammer.c (OSS)	400 src / 788 asm	87
6	winhlp.exe (bin)	8 KB	0
7	print.exe (bin)	9 KB	0
8	wscntfy.exe (bin)	14 KB	189

Figure 4: Experimental results

calculate Post*$((up, e_f^{end}))$ and read off the weight obtained for end^{end}.

5 Implementation

We integrated context summaries into the TSL-VSA system being developed in Prof. Tom Reps's group at University of Wisconsin-Madison.This system implements several analyses on stripped binary files on 32 and 64 bit architectures. The system supports the Extended WPDSs of Lal et al. as the WPDS engine [5]. EWPDSs provide us with a clean way of generating reverse calls using *merge functions* in rules (4.1), (4.2) and (5.1). Furthermore, it was possible to implement the rules in a slightly different way so that context summaries for all the functions could be obtained via only one Post* query, which is the costliest operation in the analysis. For experimentation, we instantiated the context summarization scheme for Affine-Relation Analysis, which summarizes linear equality relationships between register values at different program points based on [6].

6 Experiments

Although implementing ARA is the first step towards implementing more complex analyses, it creates some problems with experimentation. Since ARA does not handle memory operations, most information is lost because of the large number of loads and stores in assembly code. We get around this problem by using the '-O' & '-fcaller-saves' flags of gcc for our own test programs. Unfortunately, for Windows binaries, our analysis ends up being too conservative. The results of our experiments are shown in Figure 4. Program 1 through 3 are basic programs written to exercise the different rules of WPDS generation while programs 4 & 5 were extracted from opensource projects [2] [1]. Examples 6, 7, and 8 are binaries from the Windows XP system32 folder. For the programs, size is recorded in terms of uncommemnted lines of source and assembly code, and for binaries as space on disk. The first thing to notice is that size is not correlated to the number of affine relations discovered in a program. As expected, the complexity of the code and existence of loops with function calls inside them are the prime contributors to context paths. In particular, notice that winhlp.exe and print.exe do not have a single context path along which some information can be summarized, while wscntfy.exe has the maximum number of discovered summarized affine context summaries.

Although our experimental results demonstrate the basic path summaries as expected, they fall short on several fronts. First, the program sizes dealt with are quite small. Secondly, although the number of affine relations discovered for different function summaries is some indication of the information gathered, most of these are simple relations between the value of a register and the stack pointer arising from stack pointer arithmetic rather than the program variable relations we are likely to be more interested in. This is also the case for simple ARA. We do find some useful information for the larger programs.

7 Conclusion and Future Work

We proposed the concept of context summaries as a way to collect information about the environment of a function and developed a method using WPDSs to obtain these summaries. In particular, we implemented the Affine Relations Analysis in the TSL system and carried out some preliminary experiments validating the translation for path summaries. Significant future work remains to be done beginning with a simple reinterpretation to obtain other analyses like Local ARA (which includs local variables) and kill-use analysis on context paths. These analyses may uncover more interesting information flows and more useful summaries. With such an analysis in place, we would like to be able to bridge the gap between the ideas in this paper and their implementation in an analysis of library / client interfaces.

References

[1] http://sourceforge.net/projects/formelparser.

[2] http://sourceforge.net/projects/libprime.

[3] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. 1243:135–150, 1997. 10.1007/3-540-63141-0_10.

[4] Aditya Kanade, Rajeev Alur, Sriram Rajamani, and Ganesan Ramanlingam. Representation dependence testing using program inversion. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 277–286, New York, NY, USA, 2010. ACM.

[5] Akash Lal, Thomas Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. 3576:434–448, 2005. 10.1007/11513988_44.

[6] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[7] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206 – 263, 2005. Special Issue on the Static Analysis Symposium 2003 - SAS'03.