CS738: PROJECT REPORT (GROUP 1)
INTER-PROCEDURAL DATA FLOW ANALYSIS IN LLVM USING VALUE CONTEXTS

Deeksha Arora
20111017
deeksha20@iitk.ac.in

Sambhrant Maurya
20111054
samaurya20@iitk.ac.in

Shruti Sharma
20111061
shruti20@iitk.ac.in

ABSTRACT

This project is a re-implementation of the VASCO framework available here[1], although this time in LLVM. The original VASCO framework was implemented in SOOT, the implementation details of which are described in this paper [2] by Rohan Padhye and Uday P.Khedker presented at *2nd International Workshop on State Of the Art in Java Program analysis (SOAP '13)—Seattle, WA*.

Keywords: Interprocedural analysis, context-sensitive analysis, sign analysis

1. INTRODUCTION

Interprocedural analysis extends the scope of data flow analysis across procedures. It incorporates the effects of procedure calls in the caller procedures and calling contexts in the callee procedures. A context-insensitive analysis does not distinguish between distinct calls to a procedure while a context-sensitive analysis considers the calling context when analyzing the target of a function call (callee). A context-sensitive analysis restricts the propagation of data values to invalid paths and hence is more precise. Although, the complexity of analysis increases. An interprocedural analysis is precise if it is flow sensitive and fully context-sensitive even in the presence of recursion.

Two dominant approaches of interprocedural analysis are the Functional (summary-based) approach and the Call Strings approach. The functional approach is a bottom-up approach that computes summary flow functions for each procedure and uses them as the flow function for a call block. The limitation here is that constructing summary flow functions may not be always possible.

The call strings approach is the most general context-sensitive dataflow analysis method.It ensures the validity of interprocedural paths by maintaining a history of calls in terms of call strings. However, this method requires an exponentially large number of call strings and it cannot be used with frameworks with infinite lattices.

It is often easier to implement an intraprocedural analysis before extending to interprocedural analysis. Intra/inter-procedural property is highly related to context-sensitivity since a context-sensitive analysis has to be an interprocedural analysis.

Our framework uses the concept of value based context sensitivity which combines the tabulation method of the functional approach - enumeration of functions as <input, output> pairs and value based termination of call strings approach - use data flow values to restrict the explosion of call strings. Hence if two or more calls to a procedure P have the same dataflow value at entry of P , then all calls will have same value at exit of P. Our framework hence reaps the benefits of the best of both worlds.

2. RELEVANCE TO CS738

This project is based on Interprocedural Analysis which is an important mechanism for performing *compiler optimizations* across function boundaries. Practical compiling systems apply Interprocedural Analysis to produce more efficient object programs.

Sign analysis and Constant Propagation are very important components of Program Analysis. Some of its most notable applications include code optimization and security.

3. IMPLEMENTATION

Compiler Infrastructure : LLVM v11 or later
Operating System : Linux/MacOS
Language : C/C++

The following assumptions have been made:

- Lattice is finite
- Flow functions are finite
- Lattice may or may not be distributive

Approach

1. **Tabulation method used in functional approach:** This enumerates the function as a pair of (input, output) data flow values for each procedure, and then constructs summary flow functions, by reducing compositions and meets of flow functions of individual statements into one.

2. **Value-based termination used in call string approach:** The call strings method remembers calling contexts in terms of unfinished calls as call string. Value-based termination uses data flow values to limit the number of contexts and thus improves efficiency and maintains precision.

We have implemented a generic framework which uses value-based contexts and is based on both tabulation method of the functional approach and the modified call strings method.

4. INTERPROCEDURAL ANALYSIS USING VALUE CONTEXTS

A value context can be defined as *X* = <*method, entryValue*> where *entryValue* is the data flow value at the entry of the method. We also define a mapping of *X* to *exitValue(X)* which gives the data flow value at the exit of the method. All these details can be stored and maintained using tabulation, and modified call strings method would partition them using input values.

As data flow analysis is an iterative process, this mapping may change over time. The Entry value of each context of a procedure is different. The Exit value of a context changes as the analysis proceeds. The new value is propagated to all callers of method if and when this mapping changes. We create a map of contexts to a list of callsites to get all the callsites from the caller context. We re-analyze the call sites if the exit value of caller context changes. Using this information, interprocedural analysis can be performed for each value context independently, handling transfer functions in the usual way, and hence only procedure calls need special treatment. If at any instance, a procedure call is made to a method, and its context data X_1 = <*method, entryValue*> is already available in the context table, then it can simply return *exitValue(X)* without any recomputation. This would not only save the computation time and resources, but will also simplify the call graphs. It is experimentally observed that in the worst case, although the number of value contexts created per procedure is theoretically proportional to the size of the lattice, in practice the number of distinct data flow values reaching each procedure is often very small.

LLVM Instructions to be handled

- Load Instruction - An LLVM IR load instruction is used to read value from memory. It has syntax: <result> = load <ty>, <ty>* <pointer>
- Store Instruction - An LLVM IR store instruction is used to write value to memory. It has syntax: store <ty> <value>, <ty>* <pointer>
- Call Instruction - Since we are implementing an interprocedural analysis, we need to handle function call instructions. The call instruction has syntax: <result> = call <ty> <fnptrval>(<function args>) The *result* is the return value of the function call with type *ty*. *fnptrval* is the identifier of the called function, and *function args* is a list of arguments that gets passed in to the function.
- PHI Instruction - LLVM IR uses static single assignment form (SSA) to represent variables. This splits existing variables into many variations. SSA allows compilers to perform many kinds of optimizations such as constant propagation, value

Algorithm 1: IPA using Value Contexts

```
Procedure initContext(X):
    ADD(Contexts, X)
    Set ExitValue(X) := ⊤
    for node n in METHOD(X) do
        ADD(workList, <X, n>)
        IN(X, n) := ⊤
        OUT(X, n) := ⊤
    end
    IN(X, ENTRYNODE(m)) :=
      ENTRYVALUE(X)
Procedure doAnalysis():
    initContext(<main,BI>)
    while !workList.empty() do
        <X,n> := REMOVENEXT(workList)
        if n != entryNode then
            IN(n) := ∩_{p∈pred(n)} OUT(p)
        end
        a:= IN(X,n)
        if n contains a method call then
            m := TARGETMETHOD(n)
            x := CALLENTRYFLOWFUNC-
              TION(X, m, n, a)

            X' := <m,x>
            ADD transition (X' <- <X, n>)
            if X' ∈ contexts then
                y := EXITVALUE(X')
                b1 := CALLEXITFLOWFUNC-
                  TION(X, m, n, y)

                b2 := CALLLOCALFLOW-
                  FUNCTION(X, n, a)

                OUT(X, n) := b1 ∩ b2
            end
            else
                initContext(X')
            end
        end
        else
            OUT(X, n):=
              NORMALFLOWFUNCTION(X,
              n, a)
        end
        if OUT(X, n) has changed then
            for all successor s of n do
                ADD(worklist, <X, s>)
            end
        end
        if n==exitNode then
            exitValue(X, n) := OUT(X, n)
            ∀ < X', c >∈ transition[X], add c to
              workList[X']
        end
    end
```

range propagation, etc. However, when there are control branches in the program, we need some kind of mechanism to determine which branch was executed. PHI node in LLVM IR has syntax:
<result> = phi <ty> [<val0>, <label0>], ...
where *<result>* is the new SSA variable, *<ty>* is the type of the variable, and *[<val0>, <label0>]* is the list of SSA variables with labels of the corresponding control flow branches of each SSA variable.

Implementation Details

- **Context.cpp** Defines a value based context, *Context <M,N,A>* where
  - M is the type of a method
  - N is the type of a node in the CFG
  - A is the type of the data flow value

  A value-based context is identified as a pair of a method and the data flow value at the entry of the method, for forward flow, or the data flow value at the exit of the method, for backward flow. Thus, if two distinct calls are made to a method and each call-site has the same data flow value then it is considered that the target of that call is the same context. This concept allows termination in the presence of recursion as the number of contexts is limited by the size of the lattice (which must be finite).

  Each value context has its own work-list of CFG nodes to analyse, and the results of analysis are stored in a map from nodes to the data flow values before/after the node.

- **ContextTransitionTable.cpp** A record of transitions between contexts at call-sites. The context transition table records a bidirectional one-to-many mapping of call-sites to called contexts parameterised by their methods. Defines a map from call-sites to contexts, parameterised by the called method: *std::unordered_map < CallSite < M, N, A > >, std::unordered_map < M, Context < M, N, A > > > transitions* and a map of contexts to call-sites present within their method bodies: *std::unordered_map<Context < M,N,A >, std::vector<CallSite < M,N,A > > > call_sites_of_contexts*.

  If a call-site transition is not traversed in an analysis (e.g. a call to a native method) then it is listed as a "default site" which this table also records.

- **InterproceduralAnalysis.cpp** Performs interprocedural dataflow analysis which is fully context-sensitive even in the presence of recursion. It is the base-class for any kind of dataflow analysis, whether forward or backward and uses dataflow values to distinguish contexts.

- **ForwardInterproceduralAnalysis.cpp** A generic forward-flow inter-procedural analysis which is fully context-sensitive. This essentially captures a forward data flow problem which can be solved using the context-sensitive inter-procedural analysis framework as described in InterProceduralAnalysis.cpp .
  - ***initContext(method, entryValue)***: Creates new contexts. Here, *method* is the method for which the context is to be created and *entryValue* is the data flow value at the entry of this method. The following steps are performed:
    1. Create the context.
    2. Initialise IN/OUT for all nodes to $\top$ and add them to the work-list
    3. Initialise the IN of entry points with a copy of the given entry value.
    4. Add this new context to the given method's mapping.
    5. Add this context to the global work-list.
  - ***doAnalysis()***: Performs dataflow analysis. A work-list of contexts is maintained, each with it's own work-list of CFG nodes to process. Remove a node from the work-list using lexicographical ordering of contexts (newer first).For each node removed from the work-list, compute the meet of values along incoming edges (in the direction of analysis) and then depending on whether the node contains a function call or not, it's OUT value is computed. If the OUT for this node has changed, then it's successor nodes (in the direction of analysis) are added to the work-list. Analysis starts with the context for the program entry points with the given boundary values and ends when the work-list is empty.

- **SignAnalysis.cpp** Performs an inter-procedural simplified sign analysis by mapping numeric variables to their sign (negative, positive or zero), if it is statically determined to be singular, or else to bottom (represented by null).

Figure 1 shows the flow of control of our framework.

5. GLOBAL DATA STRUCTURES

- ***map < M, List<Context > > contexts***: mapping from methods to a list of contexts
- ***map <CallSite, map <M,Context> > transitions***: map from call-sites to contexts, parameterised by the called method M.
- ***vector < Contexts > workList***: work-list of contexts to process.

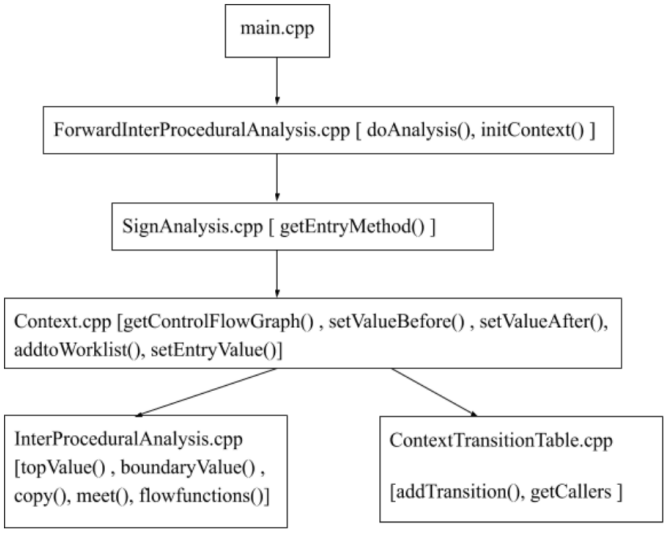

Figure 1. Flow of Control

6. SIGN ANALYSIS

This analysis maps numeric variables to a sign (negative, positive or zero), if it is statically determined to be singular, or else bottom (represented by null). For statements involving sums or products of two variables, the flow functions are non-distributive.
For this analysis we have an abstract domain consisting of the five abstract values {+, -, 0, ⊤, ⊥}, which can be organized as follows with the least precise information at the top and the most precise information at the bottom:

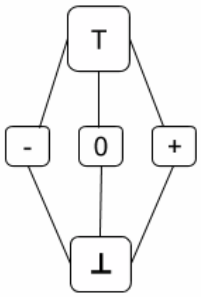

Figure 2. Lattice for single variable (Sign Analysis)

7. COPY CONSTANT PROPAGATION

We have implemented an inter-procedural copy constant propagation analysis. This analysis uses a mapping of Locals to Constants as data flow values. The flow functions consider assignments of constants to locals (immediate operands) as well as assignments of locals to locals where the operand has a constant value. This type of analysis is commonly referred to as copy constant propagation. After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned). This may also "set up" dead code elimination. After that, expressions or statements whose values or effects are unused may be eliminated. Example: Without other assignments to X, it is valid to treat the red parts as if they were in the same basic block

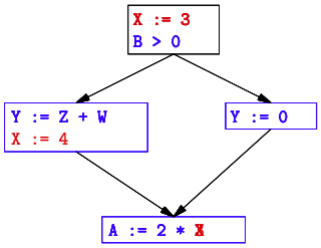

Figure 3. Example

but as soon as one other block on the path to the bottom block assigns to X, we can no longer do so.

We can apply copy propagation to a variable x defined at statement A to its use at statement B only if the last assignment for x on every path to B is A.

8. EVALUATION

There are mainly three areas of interest when evaluating an analysis: precision, time and memory usage.

We used the LLVM built-in pass -time-passes to measure the execution time of each of our analysis passes on the test example.

Figure 4 shows the Pass execution time and LLVM IR parsing time required by the framework. The figure gives details of User time, System time, User+System Time and Wall time required by SignAnalysis pass, Bitcode Writer, Module Writer and assigning names to anonymous instructions. The LLVM IR Parsing time gives details about User time, System time, User+System Time and Wall time required to parse the IR.

```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0047 seconds (0.0047 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0045 ( 95.5%)   0.0045 ( 95.5%)   0.0045 ( 95.5%)  Sign Analysis
   0.0002 (  3.8%)   0.0002 (  3.8%)   0.0002 (  3.7%)  Bitcode Writer
   0.0000 (  0.7%)   0.0000 (  0.7%)   0.0000 (  0.6%)  Module Verifier
   0.0000 (  0.1%)   0.0000 (  0.1%)   0.0000 (  0.1%)  Assign names to anonymous instructions
   0.0047 (100.0%)   0.0047 (100.0%)   0.0047 (100.0%)  Total

===-------------------------------------------------------------------------===
                                LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0002 wall clock)

   ---User Time---   --User+System--   ---Wall Time---  --- Name ---
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Parse IR
   0.0002 (100.0%)   0.0002 (100.0%)   0.0002 (100.0%)  Total
```

Figure 4. Timing Information

9. CONCLUSION AND FUTURE WORK

We have presented a framework to perform value-based context-sensitive interprocedural sign analysis and copy constant propagation analysis in LLVM. The research provides extensive and comprehensive study of some of the data flow analysis. This framework does not require distributivity of flow functions and is thus applicable to a large class of analyses. Being context-sensitive, it can be useful in dynamic optimizations. This analysis also shows that it is practical to use data flow values as contexts because the number of distinct data flow values reaching each method is often very small and this helps in reducing significant computations.

This framework can be extended to perform fully context-sensitive points-to analysis, backward interprocedural analysis (with interProceduralAnalysis as the base class) and interprocedural heap analysis.

10. REFERENCES

[1] Rohan Padhye, Uday P. Khedkar. Interprocedural Data Flow Analysis in Soot using Value Contexts https://dl.acm.org/doi/10.1145/2487568.2487569

[2] Data Flow Analysis: Theory and Practice: https://www.cse.iitb.ac.in/ uday/dfaBook-web/

[3] LLVM tutorial from Carnegie Mellon University: http://www.cs.cmu.edu/afs/cs/academic/class/15745-s15/public/lectures/L6-LLVM2-1up.pdf

[4] VASCO: https://github.com/rohanpadhye/vasco