IDEs-Friendly Interprocedural Analyser

Ilya Nozhkin
Saint Petersburg State University
St. Petersburg, Russia
nozhkin.ii@gmail.com

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
semen.grigorev@jetbrains.com

ABSTRACT

TODO: ABSTRACT

ACM Reference Format:
Ilya Nozhkin and Semyon Grigorev. 2019. IDEs-Friendly Interprocedural Analyser. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

TODO: INTRODUCTION

2 PRELIMINARIES

TODO: PRELIMINARIES

3 SOLUTION

3.1 Main idea

The solution involves using of the conception that is close to CFL-reachability to solve the problems mentioned above.

The classic CFL-r approach consider analysis as a search of paths in a graph, edges concatenation of which is a word of a certain context-free language which is defined by grammar. However, such definition has a few drawbacks when it comes to static analyses. The first of them is that grammars have a slightly unnatural structure in comparison with program interpreter. For example, call-return edges pairing is defined using grammars as brackets that contain anything else between them. In interpreter semantics, in contrast, calls and returns are usually defined separately from each other using only stack concept. So, the main idea is to formulate analyses in terms of pushdown automata that in turn should emulate original interpreter semantics but using less computational power. Moreover, it is quite useful to define automata as abstract as possible to allow to use any objects as states, input and stack symbols and by this get closer to interpreter in contrast to grammars which are based on strings.

Therefore we can define analysis as a pair of a control-flow graph which contains information about analysed program and a pushdown automaton which accepts only sequences of operations that leads to an error. The result of such analysis is a set of paths in the graph each of which can be traversed during the normal execution and also accepted by automaton, thus it can cause an erroneous behaviour.

3.2 Example

Now, let's consider the construction of a simple analysis including definition of graph and PDA.

Let nodes have no special meaning and can be described as states or positions in program. Each edge, in contrast, has label which defines an operation and translates the program from one state to another. So, three different procedures in the following program produces three separate graphs. Next step is to determine how they will be interconnected to make interprocedural analysis possible.

TODO: PROGRAM AND PICTURE

One way to connect procedures is to replace invocations with calling and returning edges that are directed to the beginning or from the end of an invoked procedure respectively and then add different labels for each pair to distinguish one pair from another. Of course, such approach allows to perform further analysis but has some disadvantages. First of them is that if one of procedures changes then all connections that have been added instead of invocations inside it should be removed and then new edges need to be added again. But the second one is more significant. It is not obvious how to support dynamically forming connections such as invocations of delegates.

Thus, instead of it, we offer to modify PDA concept and add an ability to jump to any point of input graph during the transition. Actually, it can be interpreted as adding of fake edges. So, there is no need to change produced control flow graph at all and the only requirement is to have an opportunity to find the entry point of an invoked procedure during the simulation.

Now let's define PDA that checks that there is access to an undefined variable. To simplify definition we give it in informal way which still can be translated into strict rules. Let set of states be set of variables with one dummy initial state. Set of stack symbols be just edges with one dummy edge that indicates the bottom of the stack. And the transition rules be following (SHOULD I REWRITE THEM IN PSEUDOCODE???):

- If current state is initial and next operation is variable declaration then just switch state to this variable.
- If current state is variable and operation assigns this variable then abort transition because further computations are not interesting.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- If current state is variable and operation is invocation that pass this variable as argument then push current edge to the stack, switch state to the variable that corresponds to the argument and jump to the entry point of called procedure.
- If current state is variable and operation is return of this variable from function then pop from the stack the edge, change state to the variable that is assigned by popped invocation and jump to the target of the popped edge.
- If current state is variable and operation assigns some another variable to this variable then just switch state.
- And finally, if there is met some computation using the current variable then accept the path.
- In all other cases just skip the operation

Therefore, if this PDA is runned from the whole programs's entry point then each accepted path is the sequence of operations that since the certain one pass some uninitialized variable from its declaration to its usage.

Next step is to implement interfaces that allows to reproduce such construction in code.

3.3 Solution structure

In order to provide the most common interface for interaction with IDE, the solution is offered to be devided into two separate entities. The first of them is the thin plugin for IDE that translates methods into the graphs, sends them to the second entity and then gets analysis results by request. The second one is the remote service that aggregates graphs into the full graph of the program, is able to update it incrementally and finally can perform any available analysis by request. This side also takes care about PDA simulation and results extraction and provides interfaces that require only the PDA implementation itself.

3.4 Service

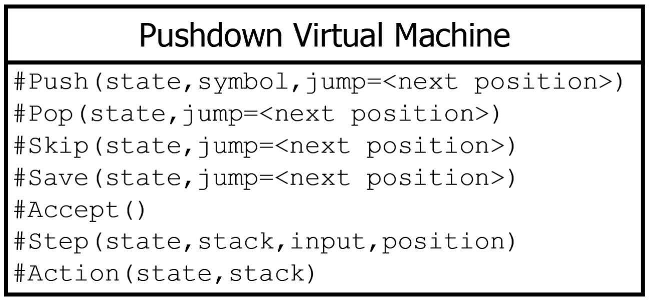
Service is implemented in C# but due to socket-based connection can be used with any other side implementation that supports interchange protocol.

TODO: IN PROGRESS

3.5 Automata construction

The main of them is the abstract PDA class that is needed to be inherited by concrete analysis implementation. Due to generalization of PDA and its slightly different meaning in case of static analyses it is named pushdown virtual machine (PDVM) (fig. 1).

It is designed to provide a set of specific methods that control PDA-like behaviour. The first important subset of them are Push, Pop, Skip and Save. Each of them corresponds to one transition of classic PDA but supports jumps as well. I.e they perform corresponding stack modification, change the state and then jump somewhere or just step to the next position if jump target is not specified. The only odd thing here is difference between Skip and Save. Both of them just stay stack unchanged and then performs other actions, however skipped input symbols are not added into traces during further results extraction and saved ones are.



```
#Push(state,symbol,jump=<next position>)
#Pop(state,jump=<next position>)
#Skip(state,jump=<next position>)
#Save(state,jump=<next position>)
#Accept()
#Step(state,stack,input,position)
#Action(state,stack)
```

Figure 1: Abstract Pushdown Virtual Machine

Another contol method is Accept. Invocation of it leads to the accepting of all paths execution of which finishes in the current position and in the current state.

Finally, two remaining methods are needed to be implemented by developer using control methods. Step method is invoked for each next input symbol in a configuration. Action method is called in each new configuration just before input symbols reading. It can be used when it is needed to perform a chain of transitions without any movement.

Since a PDVM is constructed it can be runned from any set of positions.

3.6 Plugin

TODO: PLUGIN

4 EVALUATION

TODO: EVALUATION

5 CONCLUSION

TODO: CONCLUSION