Interprocedural Static Analysis by Abstract Interpretation

Master's Thesis Project Description

Flurin Rindisbacher
Supervisors: Jérôme Dohrau, Dr. Caterina Urban
ETH Zurich

February 28, 2017

1 Introduction

Static analysis provides mathematical guarantees about the behavior of programs and can also be used to infer program specifications, such as pre-/postconditions and loop invariants. Sample [1] is a generic static analyzer developed within the Chair of Programming Methodology at ETH Zurich. It is based on abstract interpretation [2] a generic framework for the sound approximation of the semantics of programs. Sample supports generic/parametrizable intraprocedural analyses and one special purpose interprocedural analysis [3]. An intraprocedural analysis looks at each procedure in isolation. Due to the isolated analysis information from the calling context cannot be leveraged and the results may be less precise than an analysis that would take the calling context into account. The goal of this project is to extend Sample with a generic/parametrizable interprocedural analysis.

2 Some Approaches and Difficulties

This section gives a short overview over some approaches in interprocedural analysis and their trade-off between precision and cost of analysis. A *trivial treatment* of procedures as suggested by [4] analyzes each procedure of a program separately and after an invocation only conservative information about global variables, return values and the heap is used. This information is set to unknown value after a procedure call. This analysis is very easy to implement but imprecise because all information about global variables is lost.

Another simple and sometimes successful technique is *inlining*. The calls to `increment()` in Listing 1 could be inlined by textual replacement of the call with the body of the called procedure. Since there are no procedure calls anymore, `main()` can be analyzed with any intraprocedural analysis. But the same technique would fail for `rec()` because inlining would not terminate due to the recursive call.

A more involved *context insensitive* analysis that uses the control flow graph of the program is presented in [4] as the "naïve solution". Each procedure invocation is treated as a goto from the call instruction to the called procedure. The return statements are then treated as a non deterministic goto to the next instruction after all calls to this procedure. The calls to `increment()` in Listing 1 would result in the gotos from-line-number $\rightarrow$ to-line-number: $4 \rightarrow 18$, $5 \rightarrow 18$, $18 \rightarrow 5$ and $18 \rightarrow 6$. Adding these goto statements may introduce impossible paths (e.g the path $5 \rightarrow 18 \rightarrow 5$) leading to a loop which will result in *top* as the value of the tracked variable. This analysis would therefore not be useful for `main()`.

Using a *context sensitive* approach that remembers where a call came from these impossible paths can be avoided. [4] uses a call string that simulates the stack of procedure calls. Line numbers 16, 17, 19 and 20 in Listing 1 show that for such an analysis the possible values of the variable x would be differentiated using the call strings L4 and L5 (standing for Line number 4 and 5 where the call came from). The call string for `rec(x+1)` on line number 11 would be L11.L11.L11... or using a regular expression L11* because of the recursive call.

Out of the so far described approaches only the context sensitive analysis would be able to deduce that on line 6 in `main()` x has the value 3 and will therefore be within the bounds of the array. The other analyses are cheaper in terms of analysis cost but more imprecise. The call string approach itself is efficient for short strings. When the call stack becomes deep or in the case of recursion grows indefinitely it has to be limited resulting in a loss of precision.

```
1  int main(){
2      int array[5] = {0, 1, 2, 3, 4};
3      int i = 1;
4      i = increment(i);
5      i = increment(i);
6      return array[x];
7  }
8
9  int rec(int x) {
10     if(..)
11         return rec(x+1);
12     return x;
13 }
14
15 int increment(int x){
16     // L4: [x→1]
17     // L5: [x→2]
18     return x+1;
19     // L4: [x→2]
20     // L5: [x→3]
21 }
```

Listing 1: Small demo program for the different approaches. The comments in `increment()` show the call string and the value of the variable a numerical analysis would keep track of.

Furthermore Cousot and Cousot [5] present methods for compositional separate modular static analysis of programs by abstract interpretation. Those approaches could be more costly but also more precise than the methods above. They also discuss the fact that it is very difficult to analyze programs with an unknown dependence graph. For functional languages with higher-order functions this is the case because a function could for example call a function that was passed in as a parameter and the call destination is not clear by only looking at the call itself.

3 Core Goals

The core goals of the project consist of evaluating existing work, adapting it to our needs, instantiating a numerical analysis in Sample and evaluating the implementation.

3.1 Evaluation of Existing Approaches

As a first step existing approaches to interprocedural analysis should be evaluated. As a starting point we look at methods presented in [4] and more sophisticated solutions from [5].

3.2 Implementation of a Generic Interprocedural Analysis

Sample should be extend with a generic, extensible and tunable interprocedural analysis designed suitable to our needs:

- The analysis can be instantiated with different numerical and non-numerical domains.
- The implementation in Sample should be extensible such that it can be used as a basis for other analyses in the future.
- It should be possible to adjust the cost vs. precision ratio of the analysis. For example a faster but less precises analysis (e.g. context insensitive approach) or a more precise but slower approach (e.g. call string) should be usable depending on the use case.

3.3 Baseline for Precision

The naïve interprocedural solution in [4], which is quite imprecise but fast, will be used as a baseline to compare to. The implemented approach should beat the baseline in terms of precision.

3.4 Instantiation of the Analysis with Numerical Domains

An interprocedural analysis for standard numerical domains (intervals, octagons, etc.) should be instantiated. The analysis should keep track of the possible values of numerical variables. If Sample has been extended with a solution that was generic enough this step should be straight forward.

3.5 Evaluation of Different Parameterization of the Analysis

The numerical analysis should be evaluated using (artificial and real world) programs in terms of cost and precision. Strengths / weaknesses of the implementation should be shown. It would be interesting to see the scalability of the implementation with respect to the size, measured in number of methods, of the analyzed programs.

4 Possible Extensions

Based on progress and outcome of the core goals it will be decided which extension will be worked on. Possible project extensions could be the following.

4.1 Design of a Variant of the Interprocedural Analysis Dedicated to Numerical Domains

The core goal targets to build a generic and extensible variant of an interprocedural analysis. This extension's goal is to design an interprocedural analysis dedicated to numerical domains. By relaxing the requirement of having a generic analysis a more precise interprocedural analysis specialized for numerical domains could be designed.

4.2 Interprocedural Analysis with Non-Numerical Domains

The implemented interprocedural analysis could be instantiated with a non-numerical domain. One possible example is a domain for alias analysis. The existing alias analysis in Sample uses an unsound assumption that parameters to a method do not alias. In Listing 2 the body of the if block would not be reachable under this assumption.

```
foo(x: Ref, y: Ref) {
    if(x == y) {
        // unreachable under assumption
    }
}
```

Listing 2: Assumption that x and y do not alias

Using an interprocedural analysis we can discharge this assumption by leveraging the knowledge about aliases at the entry point of the method.

4.3 Interprocedural Specification Inference

Sample is able to infer some specifications of a program. For instance it supports the automatic inference of access permissions [6]. A possible project extension would be to adapt the existing inference to the interprocedural setting. One difficulty that would arise is the inference of a method interface (pre-/postconditions). Multiple calls to the same method could lead to different assumptions about its interface. Then plugging the inferred pre- and postconditions back into the calling method(s) may make it difficult to deduce useful information from the method calls.

4.4 Programs with an Unknown Dependence Graph

Analysing programs with higher-order functions pose a difficulty because of the possibly unknown dependence graph. A project extension could be to handle some of the difficulties that arise when the dependence graph is not fully known.

References

[1] "Sample." `http://www.pm.inf.ethz.ch/research/sample.html`. [Online; accessed 15-February-2017].

[2] P. Cousot, "Abstract interpretation in a nutshell." `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`. [Online; accessed 15-February-2017].

[3] L. Brutschy, P. Ferrara, and P. Müller, "Static analysis for independent app developers," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 847–860, 2014.

[4] L. Ramati and D. Nir, "Interprocedral analysis class notes program analysis course given by prof. Mooly Sagiv, fourth lecture given by Noam Rinetzky." `http://www.cs.tau.ac.il/~msagiv/courses/pa07/Interprocedural%20Analysis4.pdf`, 2007. [Online; accessed 15-February-2017].

[5] P. Cousot and R. Cousot, "Compositional separate modular static analysis of programs by abstract interpretation," in *Proc. SSGRR*, pp. 6–10, 2001.

[6] P. Ferrara and P. Müller, "Automatic inference of access permissions," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 202–218, Springer, 2012.