A GSoC 2015 Proposal

# Enhancing the Alias Analysis Passes in LLVM

***Mingxing Zhang,*** *Tsinghua University*

**Alias analysis (AA) is a prerequisite for many program analyses, and the effectiveness of these analyses depends on the precision of the alias information they receive. Thus I think it is necessary and meaningful to increase the precision of the current AA passes in LLVM by making them interprocedural; field-sensitive; and context-sensitive.**

## Background

In general, alias analysis is a technique that used to determine whether or not separate memory references point to the same area of memory. Many applications of program analysis, such as program optimization [1], automatic parallelization [4], and finding bugs [3], need the alias information. And almost all of these program analyses are more effective when given more precise alias information. Moreover, the scalability of such program analyses may also be impacted by AA's precision [5].

## Motivation

As a fundamental building block of code optimization, there already exist many alias analysis passes in LLVM, such as basicaa and steens-aa. But these standard LLVM AA passes either take a large amount of time (Anderson Analysis at cubic time and large memory requirements) or are cheap but somewhat imprecise (Steensgard Analysis). In order to address this problem, cfl-aa, which is done at Google, implements a demand-driven, CFL-based methods based on the algorithm derived from resent researches [8, 9]. It provides more accurate result than Steensgard's algorithm, while significantly faster than Anderson's. However, the current implementation of cfl-aa is intraprocedural; field-insensitive; and context-insensitive. As explained in the following subsections, these properties imply a huge space for increasing precision.

### Intraprocedural V.S. Interprocedural

In compiler theory, an intraprocedural analysis means that the analysis is done within the scope of one procedure, while an interprocedural analysis is done across the entire program. Albeit relatively more time-consuming, interprocedural alias analysis is needed in many scenarios. For example, detecting memory leaks rely on interprocedure alias information heavily, since an object with proper destructors will never leak if its life cycle is within a function.

Section 5.2 of paper [6] also gives an example of using interprocedure AA to find security vulnerabilities. It tries to assure that a security key will not flow into a String object.

### Field-insensitive V.S. Field-sensitive

Yong et al [7] gives an in-depth evaluation of the precision difference between field-sensitive and field-insensitive AA, and find that "distinguishing individual fields of structs is important". In their evaluation, the average pointer sets produced by field-insensitive AA are at least twice as large as the sets produced field-sensitive AA ($10X$ larger in the worst case).

### Context-insensitive V.S. Context-sensitive

In Section 6.3 (Figure 6) of paper [6], the author compares the effectiveness of type refinement between using context-sensitive AA and context-insensitive AA, and shows a considerable increase. Although the difference is not that significant when compared with the field sensitive, as demonstrated by Guyer et al [2], a small amount of imprecision in isolated parts of the program can significantly impact the effectiveness of the client analysis in specific cases, such as security analysis and parallelization.

## Plan

There are several things that I plain to do for enhancing the alias analysis passes in LLVM during this summer of code project.

1. Make the alias analysis field-sensitive by representing fields of a struct with separated nodes. In order to handle type casting, I intend to use the "collapse at casting" approach described in paper [7].

2. Handling special global variables, such as errno.

3. Extend cfl-aa to interprocedural analysis. Both context-insensitive and context-sensitive approach will be explored. The implementation of context-sensitive analysis may be based on the cloning technique proposed by Whaley et al [6].

The expectant schedule is given in Table 1.

**Table 1:** *Time allocation of the project*

| Work | Weeks |
|---|---|
| Investigating recent researches on alias analysis | 1 |
| Profiling the code | 1 |
| Field-sensitive analysis | 3 |
| Handling special global variables | 1 |
| Context-insensitive interprocedural analysis | 2 |
| Context-sensitive interprocedural analysis | 2 |
| Evaluating the precision | 1 |
| Scrub code, write documents | 1 |

## About Me

In this section, I'd like to introduce myself briefly. I'm a 3rd year PhD student from Tsinghua University, China. And my research area is mainly focus on software reliability and distributed computing, which means that I've used LLVM in many of my past and on-gong projects for detecting and tolerating bugs.

As an illustration, we have proposed a novel technique named Anticipating Invariant (AI), which can anticipate concurrency bugs before any irreversible changes have been made. Based on it, we implemented a LLVM-based tool to tolerate concurrency bugs on-the-fly. Experiments with 35 real-world concurrency bugs demonstrate that AI is capable of detecting and tolerating most types of concurrency bugs, including both atomicity and order violations. We also evaluate AI with 6 representative parallel programs. Results show that AI incurs negligible overhead ($< 1\%$) for many nontrivial desktop and server applications. And its slowdown on computation-intensive programs can be reduced to about $2X$ after using the bias instrumentation. The paper is published in FSE 2014 and won a SIGSOFT distinguished paper award. The source code of our implementation is also available at http://james0zan.github.io/AI.html.

I have also done a 3-months internship at Google NYC on improving database testing and a 9-months internship at Microsoft Research Asia (MSRA) for implementing a distributed (MPI-based) L-BFGS library. And I have got an excellent assessment for my MSRA internship. Moreover, I've visited the Columbia University for 6 months and working on a performance bug detection project, under the supervising of Prof. Junfeng Yang.

My email is james0zan@gmail.com and my homepage is http://james0zan.github.io/.

## References

[1] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98.

[2] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2), October 2005.

[3] David L. Heine and Monica S. Lam. A prac-

tical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03.

[4] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99.

[5] Marc Shapiro, II and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97.

[6] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04.

[7] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99.

[8] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13.

[9] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08.