ISL Memory Management Using Clang Static Analyzer

Malhar Thakkar - Indian Institute of Technology Hyderabad, India
cs13b1031@iith.ac.in | malhar1910@gmail.com

Abstract

Maintaining consistency while manual reference counting is very difficult. Languages like Java, C#, Go and other scripting languages employ garbage collection which automatically performs memory management. On the other hand, there are certain libraries like ISL (Integer Set Library) which use memory annotations in function declarations to declare what happens to an object's ownership, thereby specifying the responsibility of releasing it as well. However, improper memory management in ISL leads to invocations of runtime errors. Hence, my proposal is to employ a robust static analyzer in clang which raises warnings in case there is a possibility of a memory leak, double free, etc.

Background

Clang Static Analyzer

Typically, static analysis does work similar to compilers in raising warnings with varieties of code. But, static analysis takes it further and finds bugs which are traditionally found using run-time debugging. Clang Static Analyzer [3] is a source code analysis tool which finds bugs in C, C++ and Objective-C programs. Among the many analyses, Clang Static Analyzer also includes a mechanism to check if the memory management is done correctly in an Objective-C code.

`RetainCountChecker.cpp`

An implementation of memory management checker algorithm is included in the source file *`clang/lib/StaticAnalyzer/Checkers/RetainCountChecker.cpp`*. Although this checker does a pretty good job at raising warnings for every possible memory management problems in Objective-C, it is not generic in the sense that it was designed for only a very specific use namely, Objective-C. In addition to Objective-C, this checker also works for C functions. It checks for annotations in the Apple C API called CoreFoundation (CF) that requires a reference counting discipline. This discipline is similar to those used in Objective-C but the rules are a bit different. At the moment, this checker is not aware of the annotations used for function parameters in ISL. Hence, these annotations need to be incorporated to check for proper memory management in ISL.

ISL Annotations

The following annotations for function parameters and arguments have been mentioned in the ISL developer manual [4]:

- **`__isl_give`:** This annotation for a function argument means that a new object should be returned by the function. It is the user's duty to see to it that the object returned in such a manner is used exactly once as an `__isl_take` argument and in between it can be used for any number of `__isl_keep` arguments.
- **`__isl_null`:** This annotation for a function means that a `NULL` value is returned.
- **`__isl_take`:** This annotation for a function argument means that the object the argument points to is taken over by the function and may no longer be used by the user as an argument to any other function. The pointer value must be one returned by a function returning an `__isl_give` pointer.
- **`__isl_keep`:** This annotation for a function argument means that the function will only use the object temporarily. After the function has finished, the user can still use it as an argument to other functions.

In addition to the aforementioned annotations, if memory is allocated for any ISL object inside a function, then either that object should be returned from the function, or the memory allocated to it should be freed inside the function.

From the above explanation, it is quite evident that in case of ISL, the static analyzer should care about what comes in and what goes out of a function.

ISL and Objective-C

Similar to ISL, the Objective-C rules for reference counting are intraprocedural. The retain count checker checks each Objective-C method at the top level to make sure this is enforced. Hence, memory management for each function in ISL can be performed in a similar manner. However, this will require further discussion on the cfe-dev mailing list.

Objective

Modify `RetainCountChecker.cpp` in Clang Static Analyzer to incorporate new attributes in addition to CoreFoundation attributes to enable/improve memory management for ISL and C codebases.

Benefits to Community

Successfully carrying out this project will benefit the open-source community in the following ways:

- Provide a robust bug-finder/static analyzer to the community, so that everyone can verify their C-style reference counter.

- Fix existing bugs in ISL.
    - Creating an analyzer which checks for issues like memory leaks, double frees, etc. will help point out bugs in the existing ISL codebase.
- Help developers to use ISL correctly.
    - Many novice ISL programmers encounter runtime errors due to improper memory management done by them. Hence, performing static analysis on the code before execution can help prevent such bugs for developers reducing development time significantly.
- Benefit the wider Clang Static Analyzer community because it will add callee-side parameter checking to RetainCountChecker. This will find a new class of bugs on Objective-C and CoreFoundation codebases in addition to to ISL projects.

Milestones and Deliverables

I am committed to LLVM's incremental development policy and that the criterion for completion of coding milestones is that a patch has been reviewed and accepted by the Clang community and committed to llvm.org trunk. Also, I am planning to achieve all the below mentioned milestones chronologically starting from May 5th, 2017.

Time period	Milestone
May 5 - May 29	Community bonding period.
May 30 - June 5	1. **Evaluation of Existing Checker on Representative Codebases:** Apply the existing analyzer to the ISL and Polly codebase with the annotations `#define`'d to their CoreFoundation attributes (i.e., `cf_consumed` for `__isl_take`, `cf_returns_retained` for `__isl_give`). Prepare a report on the kinds and relative frequency of false positives and false negatives found. This will help prioritize later work (What heuristics will be needed? What diagnostics will need to be modified?) The deliverable will be an email writeup characterizing the false positives, false negatives, and diagnostic quality sent to the cfe-dev mailing lists.
June 6 - June 19	2. **Initial Annotation Support:** Support for ISL-specific annotations (with the

	clang ‘annotate' attribute) when creating summaries. At this point the analysis would still be treating ISL annotations as core foundation annotations.
June 20 - July 3	3. **Diagnostic Customization:** Add custom diagnostic text for ISL-originated data types. This will add variants of the diagnostics where the jargon is changed from CoreFoundation terms to ISL terms.
July 4 - July 17	4. **Add Callee-Side Parameter Checking:** The RetainCountChecker already does caller-side checking, but it doesn’t check parameters in the callee. (For example, it won’t warn if you forget to free an `__isl_take` parameter). For this milestone, add parameter checking (Dr. Tobias’ patch) behind a flag. This will be off by default initially (at least for the non-ISL checker).
July 18 - July 24	5. **Evaluate ISL Checker on C and C++ Codebases:** Re-evaluate with the new diagnostic text and parameter checking on C and C++ code. Produce a second report on false positives and diagnostic issues. Again, it will be an email writeup characterizing the false positives, false negatives, and diagnostic quality sent to the cfe-dev mailing lists.
July 25 - Aug 14	6. **Fix Issues from C Evaluation:** Add heuristics and improve diagnostics from 5) for C codebases (but not C++).
Aug 15 - Aug 28	7. **Turn it on!:** Turn the checker (and parameter checking) on by default.

The primary challenge in this project will be adapting the existing RetainCheckCounter heuristics to the idioms that ISL clients use due to the following reasons:

- Need to teach the analyzer about different scenarios in which ownership of an `__isl_take` parameter is transferred to a storage location in a data structure.
- Need to ensure that the above mentioned added functionality does not introduce new false positives or interfere with existing heuristics. It is really important to avoid regressions in RetainCountChecker as it is Clang Static Analyzer's most important and widely used checker.

Planned result artifacts (Code, Documentation, Experiments, ...)

- Patches for implementation reviewed by mentor(s) and/or the Clang community.
- Patches to fix bugs in ISL.
- Documentation
- [Optional] Tutorial
- [Optional] Bug reports to other projects like GObject.

Criteria of Success

- Patches should be committed to llvm.org trunk, evaluated on real ISL codebases, and determined to be in good enough shape that the ISL checker can be enabled with a flag.

Related/Similar Work

Splint

It is a tool for statically checking C programs for security vulnerabilities and coding mistakes [5] . Splint's memory management algorithm detects errors like using storage that may have been deallocated, memory leaks, returning a pointer to a stack allocated object, etc [6]. In addition to static analysis of programs containing explicit memory management annotations for functions and function parameters, Splint also performs analysis on programs containing no memory management annotations by assuming an implicit memory management annotation for all the declarations which do not have an explicit one. Additionally, it can also perform reference counting by using the annotation `refcounted` to constrain the use of reference counted storage. Only pointer to struct types may be declared as refcounted, since reference counted storage must have a field to count the references. As the objective of this project and Splint's functionalities have many things in common, Splint may provide insights on some heuristics which need to be implemented in Clang Static Analyzer.

Testing Methodology

- Run the new checker on existing code bases like Polly, Pluto, ISL, GObject.
- Fix found bugs in ISL.
- Ensure that introducing new attributes and heuristics to perform memory management doesn't result in regressions in RetainCountChecker.
- Extend the existing Clang Static Analyzer 'lit' tests to cover added functionality.

Biography

I am a final year undergraduate studying Computer Science and Engineering at Indian Institute of Technology Hyderabad, India. I am reasonably familiar with the LLVM architecture. I have written various LLVM passes as part of my regular courses. I also took Compiler Engineering in which I implemented an analysis pass which collects various loop properties and also a transformation pass which performs basic loop invariant code motion. Also, as part of my Advanced Compiler Optimization course project, I wrote a Global Code Motion pass in LLVM which implements Cliff Click's PLDI 1995 paper [1]. All of these projects can be found on my GitHub profile [2].

PS: I do not have any summer commitments other than this project and if my application is accepted, I am willing to work full time on it.

References

1. Cliff Click. 1995. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (PLDI '95). ACM, New York, NY, USA, 246-257.
2. Malhar Thakkar's GitHub Profile: https://github.com/malhar1995
3. https://clang-analyzer.llvm.org
4. http://compsys-tools.ens-lyon.fr/iscc/isl.pdf
5. http://www.splint.org/
6. http://www.splint.org/manual/html/sec5.html