A Framework for Analysing a Subset of C

Alfredo Cruz & Anders Fischer-Nielsen

December 5th 2019

Contents

Todo list

1 Introduction

Analyzing a program can help detect errors in the program. Elaborate

Our analysis works on a a subset of the *C* language. Our analyzer supports a subset of the *TIP* language described by Schwartzbach in [2] as C. In the following, we give an overview of the supported language.

1.1 Expressions

The basic expressions all denote integer values:

$$\begin{aligned} E &\rightarrow intconst \\ &\rightarrow id \\ &\rightarrow E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid E\texttt{>}E \mid E\texttt{==}E \\ &\rightarrow \texttt{(}E\texttt{)} \end{aligned}$$

1.2 Statements

The simple statements are similar to C:

$$\begin{aligned} S &\rightarrow id = E \\ &\rightarrow \texttt{ printf } E; \\ &\rightarrow S\ S \\ &\rightarrow \texttt{if(}E\texttt{)}\{S\} \\ &\rightarrow \texttt{if(}E\texttt{)}\{S\} \texttt{ else } \{S\} \\ &\rightarrow \texttt{ while (}E\texttt{)}\{S\} \\ &\rightarrow \texttt{ int } id_1, \ldots, id_n \end{aligned}$$

1.3 An Example Program

A program implemented in the supported subset of C can be seen in Fig. 1.

Figure 1: An example program in *TIP*-inspired subset of C.

2 Analysis

The type of analyses we are interested in are *intra-functional*, that is, we do not consider function calls. To root the design of the analysis infrastructure we considered the four basic analyses in [2] which were also covered during the second part of the course, these are *liveness*, *available expressions*, *very busy expressions* and *reaching definitions*.

From a pure algorithmic point of view, these four analyses are striking similar: all of them operate on sets on a data structure that has a notion of successor and predecessor and both use set union, intersection and difference as operations; this suggest an analysis framework with parametric analyses instead of a framework with *hardcoded* ones. This approach promotes code reuse, hides the details of the solver and underlying operations, e.g. set union, and it exposes an interface expressive enough to write *power set based* analyses.

The intermediate representation (of the program) upon which the analysis relays on is the Control Flow Graph with single-statement blocks (CFG). A *Control Flow Graph* with *single-statement blocks* is a digraph in which the control flow between non-control flow statements is modeled. This differ from a *pure* control flow graph in the block (node) definition; in a typical control flow graph a block is a maximal sequence of linear statements. ([1]).

From each node in the CFG we require an interface to its successors (forward flow), predecessors (backward flow), left-hand side variable (if any), right-hand side expressions and type of statement. An extra requirement for the CFG is the variables set, i.e. all the variables declared in the program, the expressions set.

The *working unit* of an analysis is a set of monotone functions, and each monotone function take as an argument a node in the CFG.

To guide the construction and interfacing of the Analysis Engine (AE), we wrote `blue_print.py` that focus on the design and inner workings of the analysis engine (represented by the class `Analysis`) while abstracting away all the implementation details about the components interfacing with it.

At the top level the AE requires a CFG and a non-empty list of monotone functions: `ana = Analysis(cfg, mon_funs_list)`. To find the analysis' fix point to the given CFG, is just a matter of executing: `ana.fix_point()`. Also we designed the AE so the writing of monotone functions is as close as the mathematical formulation as possible.

To be able to find the fix point the AE keeps a list of all user provided monotone functions (`self.mon_funs`) and a list `self._state` of size the number of nodes in the CFG. Each entry of the `self._state` list holds the current result of the monotone function that analysed the node. The signature of a monotone function looks like: `def join_lub(anai,cfgn)` in which `anai` is a reference to the AE and cfgn is a reference to the CFG node to be analysed. We send a reference to the AE to the monotone function so the user can have access the `self._state` variable; this is useful when a function requires information from other nodes. To find the fix point the AE iterates over all blocks in the CFG, per each block, each monotone function is applied in the same order as they were provided by the user when the AE was constructed, a block is considered to be analysed when a monotone function returns a non `None` value. Hence, a requirement is set to all monotone functions: if a monotone function does not apply to the given CFG node, then it must return `None`. This approach can be improved by caching, per CFG node, the function that returned a non `None` value.

The `Analysis` class exposes two functions to the monotone functions: `def lub(self, left, right)` and `def gub(self,left,right)` corresponding to the lattice functions *least upper bound* and *greatest lower bound* respectively.

As an example of a monotone function implementation, we show the JOIN function w.r.t. the least upper bound:

```python
def join_lub(anai,cfgn):
    club = frozenset()
    for n in cfgn.succs:
        club = anai.lub(club,ana.state(n))
    return club
```

2.1 Lattice based analyses

To be able to handle general lattices, not just power set based ones, certain changes must be made to the `Analysis`. First a new constructor must be added this will take as an additional parameter a list of pairs `lattice` that represent all the edges in the lattice. If $(x, y) \in$ `lattice`

3 Finding Double Unlock Bugs

A requirement of this project was that the user of the analyzer should be able to supply their own analyses. We decided to enable this by having the user define a lattice and transfer functions. These structures are then used by the analyzer in order to gain knowledge about input programs.

Due to the requirement of a user passing their own analyses to the framework and the fact that these analyses need to be loaded at runtime, Python was used for the implementation of our analyzer. Given the dynamic nature of the Python programming language, evaluating Python scripts dynamically at runtime is relatively easy and supported by the language. Using a compiled language, e.g. OCaml, turned out to be too unstable in practice given the static compiled nature of the language. Loading in arbitrary user code is to a great extent not supported in compiled languages and this dynamic loading of arbitrary modules is where a dynamic language shines.

The user supplies our framework with a list of file names containing Python class definitions. The framework then dynamically loads these analyses which the user is expected to formulate as individual Python scripts. These scripts must implement and expose an `Analysis` class containing the components our framework implementation expects to be present in order to analyze.

A disadvantage in using a dynamic language is the lack of type checks. Python supports type annotations, but these are generally completely ignored by the interpreter. The type annotations read more like comments in the source code than actual safety guarantees. This means that we have no way of enforcing that the analyses supplied by the user adhere to an interface or a class definition, and we can only provide examples for the user to follow when implementing their analysis. This is unfortunate, since the framework will simply throw an `Error` when attempting to evaluate the users' analyses. Using a compiled language would allow us to specify a signature for the analyses of the user to implement, guaranteeing that the analysis could be run without errors — provided that the users' own implementations does not raise errors.

The framework is invoked with a list of filenames of analysis files. These files are implemented as very basic Python modules, which are then imported at runtime. An invocation of the two analyses `AnalysisA` and `AnalysisB` would require the user to provide the list of analyses as to the framework as `"AnalysisA:AnalysisB"`. This list of names is split on commas, and the framework then attempts to import these modules. User analyses are expected to be located in the `analyzers` subfolder of the framework implementation in a corresponding subfolder. Due to the way Python expects modules to be structured, a file named `__init__.py` needs to be co-located with the analysis implementation file. This is merely a practicality in the way the Python module system works. An illustration of the module structure can be seen in Fig. 2.

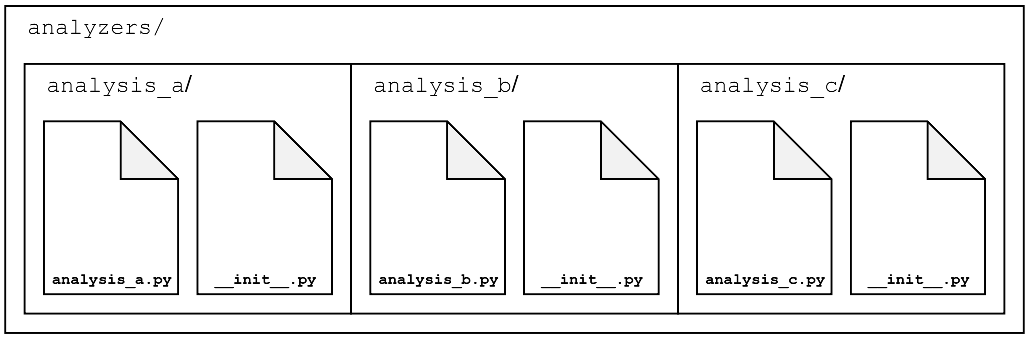

Figure 2: The folder structure for three analyses provided by the user.

An analysis provided by the user must expose an instance of an `Analysis` class. This class requires a list of transfer functions, variables and expressions to be provided for instantiation. The analyzer extracts this instance and applies the transfer functions for each provided `Analysis` to the input file. The result of applying an analysis results in a transformed input program, which is fed forward to the next provided `Analysis`. When no more analyses are present, the transformed input program is given as output to the user. An illustration of this can be seen in Fig. 3.

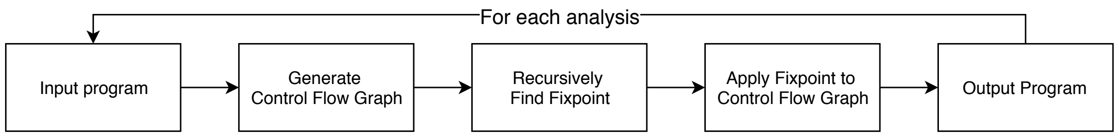
For each analysis
Input program
Generate Control Flow Graph
Recursively Find Fixpoint
Apply Fixpoint to Control Flow Graph
Output Program

Figure 3: The program flow for analyzing a program with multiple provided analyses.

`Analysis` implementation

Fixpoint finding

4 Results

4.1 Detectable Error Types

4.2 Undetectable Error Types

5 Future Work

The dynamic nature of the implementation should be made more type-safe when users provide analyses. Validating whether a given input analysis matches the expected structure of the framework should be implemented in order to reduce errors on the user's end. This could be accomplished either by dynamically verifying that members are present on the input analyses or attempting to implement the framwork in a strongly typed language, though this might not prove to be possible due to the requirement of dynamic loading.

6 Conclusion

References

[1] Keith D. Cooper and Linda Torczon. Chapter 5 - intermediate representations. In *Engineering a Compiler (Second Edition)*, pages 221 – 268. Morgan Kaufmann, Boston, second edition edition, 2012.

[2] Anders Møller and Michael I. Schwartzbach. Lecture notes on static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

7 Appendix