A Framework for Analysing a Subset of C

Alfredo Cruz & Anders Fischer-Nielsen

December 5th 2019

Contents

Todo list

1 Introduction

Static analysis of source code allows determining possible behaviour of programs, helping detection of errors in the programs. This analysis enables error reporting and optimization of programs e.g. by performing dead code elemination utilizing information known about variables in an input program. Providing information about programs allows a developer of such programs to discover programming errors and helps the developer in writing optimized programs.

This report describes our work in implementing an analysis framework for a subset of C supporting dynamic loading of user-provided analyses.

1.1 Language

Our analysis works on a a subset of the C language and supports a subset of the TIP language described by Schwartzbach in [3] as C. The following sections give an overview of the supported language.

1.1.1 Expressions

The basic expressions all denote integer values:

$$I \rightarrow -1 \mid 0 \mid 1 \mid 2 \mid ...$$
$$X \rightarrow x \mid y \mid z \, ...$$
$$E \rightarrow I$$
$$\rightarrow X$$
$$\rightarrow E\text{+}E \mid E\text{-}E \mid E\text{*}E \mid E/E \mid E\text{>}E \mid E\text{==}E$$
$$\rightarrow (E)$$
$$\rightarrow \texttt{input}$$

1.1.2 Statements

The simple statements are similar to C:

$$S \rightarrow X = E$$
$$\rightarrow \texttt{output} \ E;$$
$$\rightarrow S\,S$$
$$\rightarrow \texttt{if}(E)\{S\}$$
$$\rightarrow \texttt{if}(E)\{S\} \ \texttt{else} \ \{S\}$$
$$\rightarrow \texttt{while} \ (E)\{S\}$$

1.1.3 An Example Program

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

```
int main()
{
    int x;
    x = 1;
    int y;
    y = 0;
    int z;
    z = 0 + 1;
    if (y == 0) {
        while(1) {
            x = 0;
        }
    }
    else {
        x = 1;
    }
    return x + y;
}
```

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

2 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.
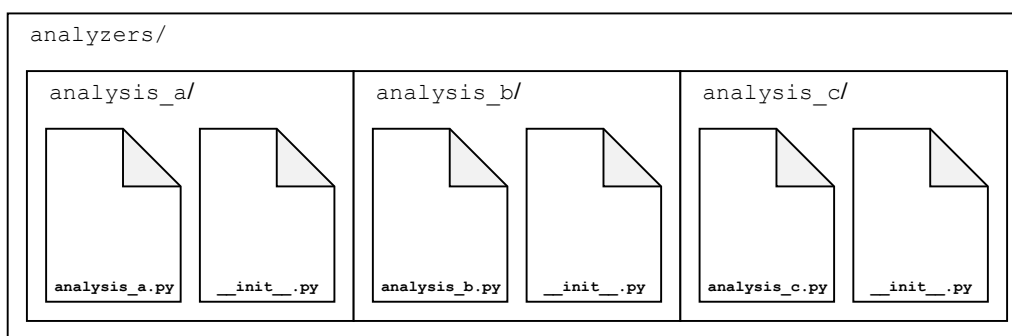

Figure 2: The folder structure for three analyses provided by the user.

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.
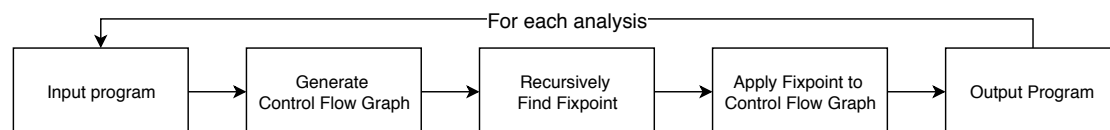

Figure 3: The program flow for analyzing a program with multiple provided analyses.

Only set-based analyses have been implemented at the time of writing and these are the only implementations supported by the dynamic loading of the framework, as described in the following section.

3 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 [3] 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 analyses relay 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]).

CFG generation based on the parsed Abstract Syntax Tree (AST) has been implemented in Python, since existing AST-to-CFG parser libraries did not support generating single-statement blocks as required by this project. The CFG generation will generate a CFG with single-statement blocks given a generated TIP/C subset AST.

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 and 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: `analysis = Analysis(cfg, monotone_functions)`. To find the analysis' fix point to the given CFG, is just a matter of executing: `analysis.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.monotone_functions`) 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_least_upper_bound(analysis, cfg_node)` in which `analysis` is a reference to the AE and `cfg_node` 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 least_upper_bound(self, left, right)` and
`def greatest_lower_bound(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:

```
def join_least_upper_bound(analysis, cfg_node):
    club = frozenset()
    for successor in cfg_node.successors:
        club = analysis.least_upper_bound(club, analysis.state(successor))
    return club
```

3.1 Lattice based analyses

To be able to handle general lattices and not only power set based analyses, certain changes must be made to the current implementation of `Analysis`. First, a new constructor must be added it will take the CFG and monotone functions as parameters but also will take as an additional parameter a list of pairs `lattice` that represent all the edges in the lattice. If $(x, y) \in$ `lattice` then $x < y$. From this list is possible to express the partial order completely as a matrix and use this matrix to calculate the least and greatest bounds. These changes are described in the Future Work[5] section of this report.

4 Results

4.1 Detectable Error Types

The detectable error types are highly dependent on the input analyses, given that if an analysis for an error type is not implemented and dynamically loaded when running the Analysis Engine, the error will not be detected. At the time of writing, only analyses described in this section have been implemented for the described error types.

4.1.1 Liveness Analysis

Liveness analysis has been implemented over the given input program, utilizing the powerset of the set of variables in the input program for the analysis. This analysis is implemented as described by Møller and Schwartzbach in [3], using set unions and set intersections for keeping track of liveness and a rule for assignment modeling the fact that *"... the set of live variables before the assignment is the same as the set after the assignment, except for the variable being written to and the variables that are needed to evaluate the right-hand-side expression."*.

Møller and Schwartzbach define the auxilary definition

$$JOIN(v) = \bigcup_{w \in succ(v)} [[w]] \tag{1}$$

This is implemented in the analysis as a union on the successors of a node and the state for the node, w, in the `join_lub` function, seen below.

```python
def join_lub(analysis, cfgn):
    club = frozenset()
    for n in cfgn.succ:
        club = analysis.least_upper_bound(club, analysis.state[n])
    return club
```

`least_upper_bound` is defined on `Analysis` as a union on two sets, making the code equivalent to the auxilary definition.

```python
def least_upper_bound(self, left, right):
    return left.union(right)
```

The assignment rule is defined by Møller and Schwartzbach as

$$(X = E : [[v]] = JOIN(v)\setminus\{X\} \cup \text{vars}(E)) \tag{2}$$

This in turn is implemented as the rule `assign`, seen below, matching the aforementioned rule.

```python
def assign(analysis,cfgn):
    jvars = (join_lub(analysis,cfgn)).difference(frozenset([cfgn.lval]))
    return analysis.least_upper_bound(jvars,cfgn.rval.vars_in)
```

4.1.2 Available Expressions

As stated by Møller and Schwartzbach in [3], an "expression in a program is available at a program point if its current value has already been computed earlier in the execution", and this information can be used for program optimization. Similar to liveness analysis, a lattice of all available expressions for all program points is used to find a lattice for analysis. Again, similar to liveness analysis, the available expressions are tracked using sets intersections and unions, as described by Møller and Schwartzbach.

Møller and Schwartzbach define the auxilary definition

$$JOIN(v) = \bigcap_{w \in \text{pred}(v)} [[w]] \tag{3}$$

This is implemented in the analysis as the intersection of the predecessors of a node and the state for the node, w, in the `join_glb` function, seen below.

```python
def join_glb(analysis, cfgn):
    if len(cfgn.pred) == 0:
        return frozenset()
    cglb = analysis.state[cfgn.pred[0]]
    for n in cfgn.pred:
        cglb = analysis.greatest_lower_bound(cglb, analysis.state[n])
    return cglb
```

`greatest_lower_bound` is defined on `Analysis` as the intersection of two sets, making the code equivalent to the auxilary definition.

```python
def greatest_lower_bound(self, left, right):
    return left.intersection(right)
```

The assignment rule is defined by Møller and Schwartzbach as

$$[v] = (JOIN(v) \cup \exp s(E)) \downarrow X \tag{4}$$

This in turn is implemented as the rule `assign`, seen below, matching the aforementioned rule.

```python
def assign(analysis, cfgn):
    jexprs = join_glb(analysis, cfgn)
    tset = analysis.least_upper_bound(jexprs, cfgn.rval.sub_exprs())
    fset = frozenset()
    for x in tset:
        if not x.var_in(cfgn.lval):
            fset = fset.union(frozenset([x]))
    return fset
```

4.1.3 Busy Expressions

As stated by Møller and Schwartzbach in [3], an "*expression is very busy if it will definitely be evaluated again before its value changes*", and can again be used for program optimization by reordering a given computation to the earliest point in time in order to optimize execution, e.g. by moving a computation outside of a loop and therefore only perform the computation once. This analysis is similar to the analysis of available expressions, since it operates over the same lattice structure and tracks busy expressions using sets.

Møller and Schwartzbach define the auxilary definition

$$JOIN(v) = \bigcap_{w \in \text{succ}(v)} [[w]] \tag{5}$$

This is implemented in the analysis as the intersection of the predecessors of a node and the state for the node, w, in the `join_glb` function, seen below.

```python
def join_glb(analysis, cfgn):
    if len(cfgn.succ) == 0:
        return frozenset()
    cglb = analysis.state[cfgn.succ[0]]
    for n in cfgn.succ:
        cglb = analysis.greatest_lower_bound(cglb,analysis.state[n])
    return cglb
```

greatest_lower_bound is, as mentioned previously, defined on `Analysis` as the intersection of two sets, making the code equivalent to the auxilary definition.

```python
def greatest_lower_bound(self, left, right):
    return left.intersection(right)
```

The assignment rule is defined by Møller and Schwartzbach as

$$[v] = JOIN(v) \downarrow X \cup \exp s(E) \tag{6}$$

This in turn is implemented as the rule `assign`, seen below, matching the aforementioned rule.

```python
def assign(analysis, cfgn):
    jexprs = join_glb(analysis, cfgn)
    tset = analysis.least_upper_bound(jexprs, cfgn.rval.sub_exprs())
    fset = frozenset()
    for x in tset:
        if not x.var_in(cfgn.lval):
            fset = fset.union(frozenset([x]))
    return fset
```

4.2 Undetectable Error Types

Though the implemented analyses supported by the Analysis Engine provide information about the input programs, examples can still be crafted where the analysis provides less information than we ourselves would get from reading the input program. Examples of such "false positives" can be seen below.

List false positive/negative code examples we have discovered

```c
int examples_here;
```

5 Future Work

As previously mentioned, as of writing, only set-based analyses are supported by the framework. A generalized lattice structure has not been determined by us, and a user can therefore not load arbitrary lattice structures and accompanying monotone transfer functions. This restricts which analyses can be run on the input programs greatly. The current set-based analysis implementation should be generalized in future work to support arbitrary lattices and transfer functions in order to make the framework more flexible in allowing other analyses types to be supported.

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.

Employing a more robust parsing implementation for parsing the input language and generating the CFG would improve the stability of the analysis. The choice of the Python package ecosystem could be reconsidered, and a different language could be chosen. Implementing a parser for a different input language could be promising, again due to the instability of the current C parsing. The Python language has elegant built-in support for parsing Python itself and writing analyses for subsets of Python could provide a positive end result. Implementing an analysis engine in a language with support for generating an AST and converting this to a CFG with relative ease for a given input language would save time when implementing the analysis, unlike this project.

Extending the input language could provide more interesting analyses provided that an extended language having constructs similar to popular programming languages in use, such as JavaScript,

Java and Python [2]. These analyses could then provide interesting feedback for developers in practical scenarios.

6 Conclusion

In this report we have shown how an Analysis Engine supporting dynamic loading of analyses can be implemented. Furthermore, we have evaluated this Analysis Engine on the implemented analyses, showing that we are able to detect the error types which these analyses support.

The Analysis Engine only detects error types for which a suitable analysis has been implemented and loaded, and there are therefore error types which are undetectable by the implementation. We have detailed how these analyses can be implemented and how the Analysis Engine could be improved in future work to improve its stability and error detection rate by supporting different input languages or implementing the core structure of the Analysis Engine differently.

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] Thomas Elliott, Samantha Rosenberg, Fred Jennings, and Jeremy Epling. The state of the octoverse: Top programming languages of 2018, Jan 2019. Available at `https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/`.

[3] 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