A Framework for Analysing a Subset of C

Alfredo Cruz & Anders Fischer-Nielsen

December 5th 2019

Contents

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.

2 Design of the Framework

Our analysis supports input programs in the *TIP* language described by Schwartzbach [3]. The *TIP* language resembles a subset of *C* and we have therefore implemented support for a subset of *C* resembling *TIP*. The following sections give an overview of the supported language.

2.1 Expressions

The basic expressions all denote integer values:

$$
\begin{aligned}
I &\to -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
X &\to x \mid y \mid z \ldots \\
E &\to I \\
&\to X \\
&\to E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid E\texttt{>}E \mid E\texttt{==}E \\
&\to \texttt{(}E\texttt{)} \\
&\to \texttt{input}
\end{aligned}
$$

2.2 Statements

The simple statements are similar to C:

$$
\begin{aligned}
S &\to X = E \\
&\to \texttt{output}\ E; \\
&\to S\ S \\
&\to \texttt{if(}E\texttt{)}\{S\} \\
&\to \texttt{if(}E\texttt{)}\{S\}\ \texttt{else}\ \{S\} \\
&\to \texttt{while (}E\texttt{)}\{S\}
\end{aligned}
$$

2.3 An Example Program

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

```c
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.

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 which 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 supported by the language and therefore relatively easy. An implementation in a compiled language,OCaml, was initially attempted, but this turned out to be too unstable in practice, given the static strongly typed nature of the language. Loading arbitrary user code is to a great extent not supported in compiled languages and this dynamic loading of arbitrary modules is where dynamic languages shine.

The user supplies our framework with a list of file names containing Python class definitions. The framework 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 ignored by interpreters. 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 analyses. 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"` 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 consequence of using the Python module system. An illustration of the module structure can be seen in Fig. 2.

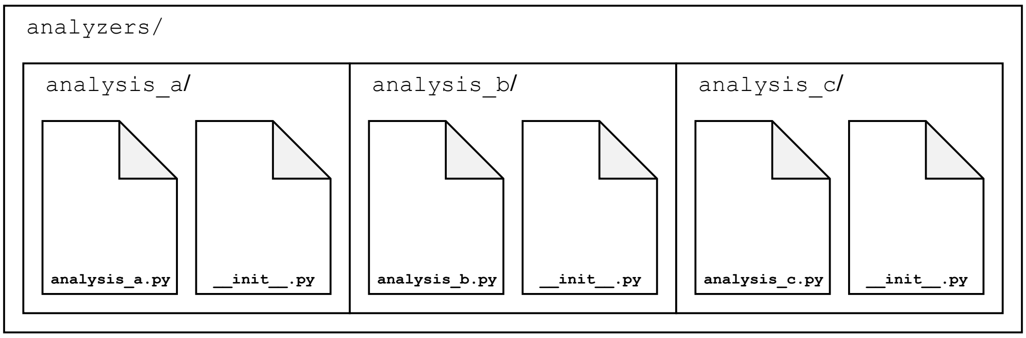

Figure 2: The folder structure for three analyses provided by the user.

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 all analyses have been run, the transformed input program is given as output to the user. An illustration of this can be seen in Fig. 3.

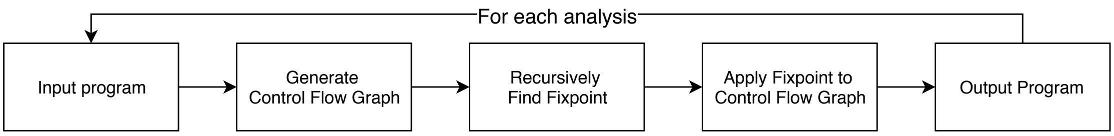

Figure 3: The program flow for analyzing a program with multiple provided analyses.

The implementation of this project, instructions for running the analyses and example input files are available on GitHub, at `https://github.com/andersfischernielsen/ASA-Analyzer/`.

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
```

To find the least fix-point, we iterate over the nodes of the CFG, applying the corresponding monotone function depending on the *type* of the node. A `_state` variable keeps track of the latest monotone function result per CFG node. The least fix-point is found when the `_state` variable does not change. The `_state` variable is updated each time a monotone function is executed, as a consequence a small improvement on the time complexity might be achieved depending on the iteration order of the CFG nodes.

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 section.

4 Results

4.1 Implemented Analyses

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 $JOIN(v)$, seen below.

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

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.

```
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.

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

The assignment rule is defined by Møller and Schwartzbach as

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

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

```
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 $JOIN(v)$, seen below.

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

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.

```
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.

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

The assignment rule is defined by Møller and Schwartzbach as

$$[v] = (JOIN(v) \cup \text{exps}(E)) \downarrow X$$

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

```
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 $JOIN(v)$, seen below.

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

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.

```
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.

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

The assignment rule is defined by Møller and Schwartzbach as

$$[v] = JOIN(v) \downarrow X \cup \text{exps}(E)$$

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

```
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 Precision of Analyses

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.

Analyses are not always precise, for example consider the following example:

```
int x,a,b;
x = input();
if(x > 0){
        x = -1;
}
a = x - 1;
b = x - 2;
while (x>0){
        output a*b-x;
        x = x-1;
}
output a*b;
```

A very busy expressions analysis would determine that a*b is a busy expression inside the while loop even though the loop is *dead-code*, that is, code that is never executed. This seemingly limitation of the analysis actually suggest an *analysis-transformation* pattern in which an user supplies a transformation function that would receive as arguments the least fix-point (e.g. _state) and the CFG and it will return a transformed CFG. It is worth to mention that the order in which the analyses and their respective transformations are executed might impact the performance of the overall CFG transformation.

To illustrate this point at an intuitive level, let us assume we have a very busy expression and sign analyses and a constrain solver with their respective transformations. If the busy expressions analysis is executed first, then the transformation would factor `a*b` out of the `while` loop and and add a temporal variable. Then the sign analysis would determine that `x` is less than or equal to 0, in this case the transformation might be empty. Finally the constrain solver would determine that the condition of the `while` is impossible to satisfy and mark the loop as dead-code. On the other hand, if the sign analysis is executed first, then the constrain solver and finally the very busy expressions analysis the result would not have an extra (temporal) variable. However, both programs are semantically equivalent.

5 Future Work

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

The dynamic nature of the implementation would benefit from having a stricter, type-safe way for users to provide analyses. Validating whether a given input analysis matches the expected structure of the framework could 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 and allow for greater expandability of the framework. The choice of the Python package ecosystem could be reconsidered, and a different language could be chosen to make traversing the input language easier. Implementing a parser for a different input language might give better results due to the instability of the C parsing of the current implementation. Python has elegant built-in support for parsing Python itself and writing analyses for subsets of Python might result in a better end result.

Extending the input language could provide more interesting analyses. If the extended language has constructs similar to popular programming languages in use, such as JavaScript, Java and Python [2], these analyses could provide interesting results for developers using these languages.

6 Conclusion

In this report we have shown how an Analysis Engine supporting dynamic loading of analyses can be implemented in Python. 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, albeit with some imprecision due to the nature of the analyses.

The Analysis Engine only detects error types for which a suitable analysis has been implemented, and there are therefore error types which are undetectable by the implementation. We have detailed how these analyses should 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