Result graphs for an abstract interpretation-based static analyzer[*]

Pascal Cuoq
cuoq@trust-in-soft.com

Raphaël Rieu-Helft
raphael.rieu-helft@trust-in-soft.com

Trust-In-Soft

ABSTRACT

TIS-Analyzer is a static analysis platform based on Frama-C. It integrates C analyzers in a plugin architecture and can be used to soundly detect undefined behaviors in C programs. The plugins communicate with each other to increase their precision. The Value analysis is an important TIS-Analyzer plugin. It uses dataflow analysis to produce a sound representation of the memory state at each control point of the program. Its abstract domain allows to represent disjunctions of non-relational value states. Further plugins then use this information to conduct derived analyses (operational inputs, dependencies...) However, some information is lost in the process: the derived analyses cannot know which disjuncts of the representation of an abstract state can be reached from each disjunct of the state at the preceding statement. We propose to represent the state of the Value analysis with a graph that refines the control flow graph of the analyzed program and that accurately represents the disjunction of abstract memory states at each statement in separate nodes. This avoids the aforementioned loss of information and leads to precision gains for the derived analyses. This new domain also allows us to formalize the Value analysis and its use of disjunctions in terms of abstract interpretation. Finally, result graphs are suited for human review and allow users to find the root cause of alarms raised by the analysis.

1. A MOTIVATING EXAMPLE

The operational inputs (or *inout*) plugin of Frama-C [4] computes the operational inputs and sure outputs of each function call in a C program. The operational inputs are defined as the memory zones that may be read by the program without having been previously overwritten, and the sure outputs are the memory zones that are written to for sure. The inout plugin runs a dataflow analysis on the control flow graph, with knowledge of the results of Value.

In the program represented in Figure 1, we therefore expect the operational inputs to be empty (`t[1]` is read, but not before having been written to) and the sure outputs to be `t`, `i` and `x`. However, the program's control flow graph is not expressive enough to reflect that the program goes through the full three iterations of the `for` loop. The dataflow analysis therefore concludes that it is possible to read `t[1]` at line 8 without having initialized it in, which makes it an operational input. Although the value analysis did iterate through the loop precisely, that information was lost in translation, and the derived inout analysis loses precision as a result.

```
1 char t[3];
2 void main() {
3   int i,x;
4   for (i=0; i<3; i++)
5     {
6       t[i] = i;
7     }
8   x = t[1];
9   }
```

```
[inout] InOut (internal) for function main:
        Operational inputs:
          t[1]
        Operational inputs on termination:
          t[1]
        Sure outputs:
          i; x
```

Figure 1: Initializing an array with a `for` loop

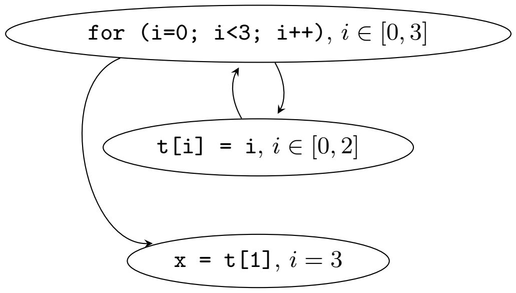

Figure 2: Control flow graph of the program

2. RESULT GRAPHS AND VALUE

The Value plugin is an abstract interpreter [2] that runs a dataflow analysis on the control flow graph of the program. When a new abstract state is discovered for a statement, it is maintained separately instead of being merged with the previous ones. The transfer functions of the statements are applied to each disjunct independently. We propose to represent a run of Value with a graph whose nodes are (statement, abstract state) pairs. When a transfer function is applied to a state, we create new nodes representing the results (or reuse existing ones if relevant) and add edges from the node representing the old state to the new ones. This does not change the behaviour of the value analysis, but only the way its results are transmitted to the derived plugins.

[*]This work was partially supported by the French ANR-14-CE28-0014 AnaStaSec.

The Value analysis also needs to merge abstract states to save time and space when there are too many for a single control point. Exactly how many can be maintained at the same time is controlled by the `slevel` parameter. When it is exceeded for a statement, the abstract states of that statement are joined. This is also easy to represent in the result graph by adding an edge from each node representing one of the abstract state to the new node that represents the joined state. The result graph thus transparently represents a run of the Value analysis.

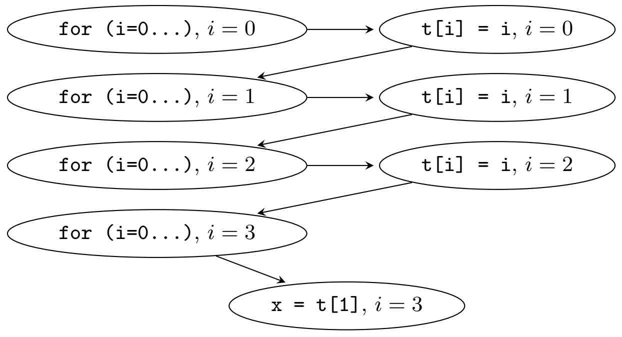

Figure 3: Result graph for the program in Figure 1

The result graph corresponding to the program from Figure 1 is represented in Figure 3. It accurately represents the fact that the only way to reach the assignment of `x` is to go through the three iterations of the `for` loop in order. By adapting the inout plugin to run a dataflow analysis on the result graph rather than the control flow graph, we obtain a precise result.

3. FORMALIZING VALUE

We argue that the value analysis should be seen as an abstract interpreter that computes a transition relation between abstract memory states. Representing Value results as a result graph is therefore only natural. We give programs a trace semantics. Given a base abstract domain on memory states, we then define our abstract domain for programs as the set of transition relations on (statement, abstract memory state) pairs similarly to [5]. The concretization of an abstract element is the set of traces that start at the initial state and take only valid transitions. We can see transition relations as graphs and traces as paths in the graph starting at the initial node. The semantics of statements are transfer functions that enrich a transition relation by adding more pairs of elements to it, analogously to adding new nodes and edges to the result graph. We define the abstract semantics of the program as the fixpoint of the transfer functions corresponding to all the statements in the program. This soundly approximates the concrete trace semantics.

An approach similar to the one described in [1] is used to avoid storing a state multiple times: when a pair (i_1, s_1), (i_2, s_2) should be added to the transition relation, if an element (i_2, s) with $s_2 \sqsubseteq s$ already exists for the same statement i_2, then the edge $(i_1, s_1), (i_2, s)$ is added instead and the node for s_2 is never created. This, as well as the `slevel` limitation that creates a merged node when there are more than a certain number of abstract states for the same statement, fits into the framework and can be seen as a widening on the graph abstract domain. Indeed, once a merge has occurred (because the `slevel` limit has been reached), states that are included in the joined state aren't added to the graph anymore, and the widening on the base abstract domain on memory states prevents an infinitely increasing chain of memory states from being encountered. This ensures convergence in a finite number of steps.

4. CONCLUSIONS AND FUTURE WORK

We have formalized the Value static analyzer as an abstract interpreter on graphs that refine the control-flow graph of a C program, and implemented this idea in TIS-Analyzer by representing its results as a result graph rather than a map from statements to abstract memory states. This allowed us to adapt some of the derived analyses that depend on the Value results to run a dataflow analysis on the result graphs directly, rather than the control flow graph. This has lead to noticeable precision gains. The tradeoff is that the result graphs are much larger than the control flow graph when the `slevel` is high, and the time and memory costs follow suit. However, thanks to the compactness of the non-relational abstract domains involved and to optimisations such as hashconsing [3], they remain manageable.

We expect further gains from the availability of a graph representing a full run of the Value analysis on a program. For example, it is a valuable tool for the user to trace back to the root cause of alarms raised by Value or of imprecisions in the results: it is now possible, given an imprecise node, to access its predecessors without having to sift through a large number of states when the `slevel` is high.

It also becomes possible to implement derived analyses that follow the evaluation order throughout the whole program. For example, we used the result graph to implement a whole-program variant of the dependencies analysis. It is able to express the dependencies of each variable in the program relative to the globals and inputs.

5. REFERENCES

[1] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*. Springer, 2007.

[2] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[3] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in ocaml 3.10. 2. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 13–22. ACM, 2008.

[4] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.