Static Analysis with Abstract Interpretation

Presented by Guy Lev
06.04.2014

Static program analysis is the analysis of computer software that is performed without actually executing programs. Its main usages:

1. Compilers: in this case we want to analyze the program in order to decide whether certain optimizations or transformations are applicable.
2. Certification of programs against classes of bugs.

Last week the tool Splint was presented: this tool performs static analysis which is unsound. Abstract Interpretation (AI) is a theory which deals with sound (conservative) approximation of the semantics of computer programs.

The term *soundness* means that if we are able to prove some property, then we are sure it is true for all possible executions of the program. However, notice that if we are not able to prove a property, we cannot infer anything.

The *concrete semantics* of a program is a representation of the set of all possible executions of a program in all possible execution environments. Any non-trivial question about the concrete semantics of a program is undecidable, in particular, trying to prove safety properties of our program. One approach is testing. Testing is an under-approximation of the program semantics: only part of executions are examined, and only the prefix of executions. Therefore, some erroneous executions might be missed. Abstract Interpretation considers *abstract semantics*: a superset of the concrete semantics of the program. Therefore, this is an over-approximation of the possible executions. Applicable abstract semantics should be computer representable, and effectively computable from the program text. If the abstract semantics is safe, then so is the concrete semantics. This means we have soundness: no error can be missed. However, if the over-approximation is too large, we might get false alarms.

In the lecture we saw an example of a simple C procedure. We used its control flow graph to compute concrete semantics of this procedure. Two problems were demonstrated:

1. The representation of all possible states in a certain node in the graph may be too complex.

2. Stopping problem: it may occur that we keep discovering new information forever, so we would never stop.

Then we went on to compute abstract semantics, using the Cartesian domain. We saw how *widening*, the losing of information, helps to overcome those problems of too-complex representation and non-stopping.

We explained the terms of transfer functions, join operation, abstraction function, and concretization function.

Then, we briefly reviewed more complex and precise domains:

1. Interval Abstraction: the possible values of a variable are represented as an interval. In this case, considering 2 variables, the range of their possible values is given as a rectangle with sides parallel to the axis.
2. Octagon Abstraction: in this abstraction we maintain relations between variables: for each 2 variables x, y, inequalities of the form $\pm x \pm y \leq c$ are maintained. In this case, the range of possible values of x, y is given as an octagon, a convex polygon with at most 8 edges.
3. Polyhedron Abstraction: here we maintain more informative relations between variables: for each 2 variables x, y, inequalities of the form $ax + by \leq c$ are maintained. In this case, the range of possible values of x, y is given as a convex polygon.

A summary of the discussion which followed the presentation:

It is hard to design a sound static analyzer which performs well, i.e. yields only few false alarms, on any given program. However if we limit ourselves to a certain type of programs, it is easier to design a suitable analyzer which would perform well on this subset of programs.

An example for such analyzer is ASTREE, a static analyzer for proving the absence of runtime-errors in real-time, safety critical, embedded software written in the C programming language. ASTREE has been successfully applied to prove the absence of runtime errors in the control-command part of the primary flight control software of the fly-by-wire system of airplanes. ASTREE uses the Octagon domain.

Another example for sound static analyzer is CSSV: C String Static Verifier. This analyzer statically uncovers all string manipulation errors in a C program. This tool first converts the given C program to an integer-program, a program which manipulates integers. The conversion is done in a way that guarantees that if there is a string manipulation error in the original program, then necessarily there would be an error in the integer program. CSSV then analyzes the integer program and if an error is found, it is able to generate a

counter-example (an example for input that would cause the error) for the original program. In the abstract domain that CSSV uses, linear inequalities among numerical variables are maintained, namely, inequalities of the form: $\sum_{i=1}^{n} c_i x_i + b \geq 0$.