Presentation Notes

Ana Nora Evans

February 5, 2017

These notes are prepared for the research group meeting discussing the ASE 2016 paper, *Inferring annotations for device drivers from verification histories* [1].

1 PL Background

1.1 Axiomatic Semantics

The theory one uses to reason about program correctness is "Axiomatic Semantics". It consists of a language for stating assertions about programs, usually first order predicate logic on involving program variables, and rules for establishing the truth of assertions.

The notation used is: $[A]c[B]$. A is a logical predicate and is called **precondition**. B is also a predicate and is called **postcondition**. c is a command or instruction.

The triple $[A]c[B]$ is called a Hoare triple. It's meaning is: "If the the predicate A is true then the command c terminates the predicate B is true".

For example, $[x < y]x := x + 1; [x \leq y]$.

How hard is it to prove program termination?

We may be able to prove termination for some toy programs, and if we work very hard even for some programs with loops, but this is an undecidable problem.

Hence, we downgrade our goals a little bit. We introduce a new notation $\{A\}c\{B\}$, which means that "If the predicate A is true and the command c terminates then the predicate B is true. If the command c does not terminate the assertion is valid".

What kind of assertions can we make?

- $x > 0$

- this program is memory safe
- the array accesses are within the array bounds
- this program is deadlock free
- this program is race free
- this program does not leak memory

For more interesting properties the language of assertions would use other logics like separation logic, temporal, linear.

These are all things we can say about a program, but that means nothing if we can't actually prove them. We need a set of rules for establishing the truth of assertions:

- usual rules from first order logic like

$$\frac{A \qquad B}{A \wedge B}$$

- rule of consequence

$$\frac{A' \Rightarrow A \qquad \{A\}c\{B\} \qquad B \Rightarrow B'}{\{A'\}c\{B'\}}$$

- one rule for each command

$$\frac{\{A\}c_1\{B\} \qquad \{B\}c_2\{C\}}{\{A\}c_1; c_2\{C\}}$$

$$\frac{\{A \wedge b\}c_1\{B\} \qquad \{A \wedge \neg b\}c_2\{B\}}{\{A\}if\ b\ then\ c_1\ else\ c_2\{B\}}$$

$$\frac{A \wedge b \Rightarrow C \qquad \{C\}c\{A\} \qquad A \wedge \neg b \Rightarrow B}{\{A\}while\ b\ do\ c\{B\}}$$

  C is the **loop invariant**.

Here is an example:

$$\{x \leq 0\}while\ x \leq 5\ do\ x := x + 1\{x = 6\}$$

with invariant $x \leq 6$.

Axiomatic semantics is

- **sound**: we can only prove true predicates

- **complete**: we can prove all the true predicates (only if the underlying logic is complete).

Where are the difficulties?

1. Application of the rule of consequence
2. Proving the implications

   With modern SMT solvers, the two above are possible to solve in most cases. Cook's theorem proves that boolean satisfiability if NP-complete, but, luckily the hard instances are not dense in the input space.
3. loop invariants
4. preconditions and postconditions for functions

   Solution for the last two: let's call them **annotations** and make the programmers write them.

This is not impossible. In the world of formal methods and verification, one uses proof assistants like Coq, Isabelle, PVS, to combine code and proofs about code. Notably, Xavier Leroy and his team implemented a verified C compiler [2] in Coq.

1.2 Static Program Checkers

In the world of C and Java, pointers, inheritance, legacy code and libraries complicate the analysis. Usually, the tools are neither sound nor complete.That means they will allow buggy programs and will give spurious warnings. Tools like ESC/-Java [3] and Microsoft Static Driver Verifier [4] are not really verifiers, in the sense that they prove the analyzed program is correct, but they will find some categories of programming errors.

For example, ESC, can give static warnings about null deferences, array bounds errors, type cast errors, race conditions deadlocks. These are all undecidable problems in general, but the kinds of programs that occur in practice are not the one from the undecidability proofs.

TODO Say something about assume-guarantee style and modular checking

1.3 Houdini

2 The Paper

Annotations are candidate predicates.

Invariants are logical formulas over program variables that are always true. For example: $x > 0$.

3 Key Insight

Programs that use the same API probably require similar annotations for verifying contracts of that API. By using information from prior verification runs (i.e. invariants already checked) of the tool on potentially different sources, candidate invariants will be generated and checked. Previous work, such as SLAM, used only the given specification and the program to be analyzed.

Key challenges:

- invariants and annotations are formulas over program variables, using local variables that will be out of scope on a different location
- keep the size of inferred annotations small. The generated set is minimal w.r.t. subset relation and generates all the invariants from previous runs.

References

[1] Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. Inferring annotations for device drivers from verification histories. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 450–460, New York, NY, USA, 2016. ACM.

[2] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[3] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[4] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 119–122, Berlin, Heidelberg, 2010. Springer-Verlag.