Program Analysis

jw3354

February 2019

1 Static Analysis & Abstract Interpretation

1.1 Context from last class:

The goal of program analysis is to extract program properties which can be leveraged to optimize programs (e.g. remove dead code), ensure security (e.g. buffer out-of-bound access), or fixing and finding functionality bugs. Note we can also do this analysis on binaries albeit harder to do.

1.2 Motivation

What are the things in all programs that can be leverage? All programs have statements for data and control flow.

- Data Flow statements: Manipulate data update data
- Control Flow statements: if/while

An analyzer's must find a representation of the program that can manipulate and reason with efficiently. At the very least, our representation should normalize syntactic sugar. This is similar to abstract syntax trees in compilers. In fact traditionally, we can just leverage the existing IR of compilers to do program analysis.

Since in the general case static analysis is resource intensive, Abstract Interpretation offers a way to simplify the computation.

2 Control Flow Analysis

Implicitly the Control Flow Graph (CFG) is an abstract syntax tree (AST) in fact a refinement of AST (a special case). Each nodes represent linear block of statements called basic block. An edge represents the relationship between basic blocks and flow of control.
NB: example in slide has no/yes flipped.
What does the CFA tell us?

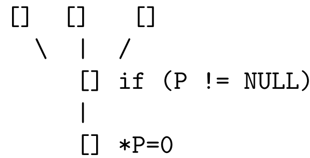
- In principle it allows you to enumerate all the distinct execution paths.
- Identify points where paths merge.
- When we merge the states, we lose information, ie might start considering cases that are not reachable.
- Potentially also how many times a variable is set to some value or some property of the values.

Algorithm for determining basic blocks:
Find leaders then greedily add instructions to the block until it gets to a branch.
Note we can detect loops by looking for back edges in the CFG.

Dominators:

- Definition: A basic block dominates another if the latter is only reachable going through the former.
- Why is it helpful? - Can tell you if there are checks that protect a certain dangerous statement.

- Example:



Using the CFG to determine properties
Clearly we can't do explicit walks cause we can have while loops which mean potentially indefinite looping or until you reach a fixed point (ie no change in state).
So we want to define a logic that lets us use transfer rules (logical reasoning) and propagate the properties from one basic block to another.
Faster because you can throw away excess information. Effectively, throwing out some execution paths. E.G. if there are blocks that don't change the state of interest.

3 Data Flow Analysis: Example with Liveness

Data Flow Analysis

Determine dynamic data properties by only looking at static code.
Property Definition:
A variable is live at a particular point if the variable be access again after that point.

Motivation:
Register Allocation (fixed resources so we can free up registers). Clearly we can optimize if we determine a variable is dead; we can just ignore the instruction there by saving register space. Since the property we want doesn't care about the actual value, so we can avoid actually computing.
Algorithm:

1. maintain a read set and a write set.
2. propagate based on the following observations/rules:
   - A variable is live at a node if it's used.
   - A variable is not live if it's defined in the same node but never used in the node or after.
   - A variable is live in a node, N, if it's live in a successor of N. Unless the successor, defines the variable.
3. repeat 2 until fixed point is reached.

Observations:

- Note the difficult part of building any analysis program is determining the update rule. Formally for liveness, we have the following update rules

$$in[n] = use[n] \cup (out[n] - def[n]); out[n] = \bigcup_{s \in Succ[n]} in[s]$$

  . Note the merge actually has to be efficient or else it's not practical.
- Solution is always complete (ie no FP or FN) but for large programs might take forever to reach fixed point (maybe never).
- Key weakness: Perhaps, let's go on but then halt after a fixed number of iterations, can we have the program say something about the program. No, a partial trace won't give us any guarantees.

- We can iterate backwards faster in this case; since the network is well defined we can just write new rules. Since it depends on the property and particular CFG if it will converge faster forwards or backwards propagated, a good opportunty for ML.
- **Can we see this happen in real compilers?** GCC is black box when it comes to this, but LLVM have separate passes and you can print out the pass. So we can collect the trace, e.g. liveness pass in LLVM.

3.1 Abstract Interpretation

3.1.1 Motivation

Can we get some kind of correctness guarantee without reaching the fixed point? What are the compromises we make to get it to terminate faster without giving nothing useful?

3.1.2 Static vs Dynamic Liveness

Note that it gets even harder if the values live in the memory. E.g. dereferenced pointers. Depending on dynamic data. In that case, the graph might change structures based on the input. E.g. with Just-In-Time Compilers, a program can compile itself and optimize itself. Restrict to simpler programs it's still time consuming to reach a fixed point.
There exist algorithms for Pointer Analysis, but take cubic complexity in program size.

3.1.3 Allow for some FP

The trade off of FP and computational time was widely investigated in the 80s and 90s. E.g. to avoid buffer overflow just not allow any kind of copy. This way there are false positives but definitely still sound.
Since we flag some correct programs, basically the analysis reports reasonable warnings. Instead of a compiler rejecting programs, allowing FP is more useful for warnings in an IDE or compiler warnings, e.g. Microsoft's Visual Studios' warnings and tab complete or Google's Lint that checks style.
In short, we keep soundness but give up completeness.

3.1.4 Where are the FP coming from?

- hard to detect feasible paths
Static vs Dynamic :: once it's been split then the states will be merged in static. but in the case of dynamic we consider the paths as a whole. We are too generous that every combination of branching is possible ignoring dependencies, for example:

```
if (x):
  do A;
if (x):
  do B;
```

The second branch is dependent on the first.

3.1.5 Alternative Approaches

Note that a natural alternative to allowing FP is to restrict the expressiveness of the language at the cost of making programming much harder; for example functional languages. Another option is to make assumptions on the programs as you analyze, e.g. maybe ignore the order of instructions.

3.2 Finally Introducing Abstract Interpretation

3.2.1 Goal:

The goal of abstract interpretation is to reach the fixed point faster in a principled manner by reasoning over equivalence classes over variable feasible sets. Instead of modeling variables as values between the maximum and minimum values (as defined by the language), we consider different partitions of the space.
Now instead of just writing rules that describe how to merge feasible sets, we determine how to reason over subsets/equivalence classes of the feasible set. A simulation approach would be a machine interpretation of the program. While a

data flow network is a data interpretation. Thus, each rule is an interpretation of the operations. Giving us an abstract interpreation of the program.
Goal is for Abstract domain to keep just enough information to reason about the property

3.2.2 Example:

```
Property is whether b is pos or neg
Partitions := {} +, -, 0}
a = -50 // P(a) = -
d = a * a /2 // P(d) = (-) * (-)/ (+) = +
b = d+ 0 // P(b) = (+) plus (0) = +
```

3.2.3 Why it Converges faster

Abstract domain should reach fixed point faster. Since there are less distinct states, this means we can reach a stable state faster. In other words, it's not as fine grained and sensitive to small scale modulations. The trade-off is that FP are introduced because we end up not knowing exactly where a program is in the covering partition. Said in another way, we can't distinguish between behaviors within a class.