Probabilistic Program Analysis

GTTSE part 4 : Don't have a PhD topic? Here take one of mine!

Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska USA

August 2015

University of Nebraska Lincoln

Explore quantitative program analyses

In security researchers moved from information flow to *quantitative* information flow in an attempt to characterize the severity of vulnerabilities.

There may be additional opportunities for adapting classic analyses to produce quantitative information that is not not necessarily probabilistic.

Quantification can shift questions from possibility/impossibility to how many, i.e., channel capacity.

For example, dependence analysis could be modified to compute edge weights that permit the capacity of dependence chains to be computed?

This would permit a "capacity threshold" notion of slicing to be performed.

Would this be useful?

Explore quantitative program analyses

In security researchers moved from information flow to *quantitative* information flow in an attempt to characterize the severity of vulnerabilities.

There may be additional opportunities for adapting classic analyses to produce quantitative information that is not not necessarily probabilistic.

Quantification can shift questions from possibility/impossibility to how many, i.e., channel capacity.

For example, dependence analysis could be modified to compute edge weights that permit the capacity of dependence chains to be computed?

This would permit a "capacity threshold" notion of slicing to be performed.

Would this be useful? You tell us in a few years.

Rank errors based on execution probability

A number of analysis techniques, e.g., model checking, provide error reports in the form of counter examples, i.e., a trace.

There can be many such traces and determining which one to look at can be a challenge.

Lots of work on error ranking in the mid-2000s, but probabilistic technique might permit a new approach where the probability of taking the path that the counter example took could allow developers to focus on "more likely" errors first.

Customizing supporting techniques for program analysis

Model counting is a general problem; the complexity of #SAT has been studied for many decades.

SAT techniques became fast by focusing on the cases that arose in practice.

This led to a proliferation of *theory fragments* with associated efficient decision procedures.

Customizing supporting techniques for program analysis

Model counting is a general problem; the complexity of #SAT has been studied for many decades.

SAT techniques became fast by focusing on the cases that arose in practice.

This led to a proliferation of *theory fragments* with associated efficient decision procedures.

The same approach seems possible for # procedures.

Simple example: polyhedra that are regular hyperrectangles are trivial to count, i.e., $\Pi_{e \in Edge} length(e)$

A first step in this direction would be to characterize the types of queries that arise in program analysis which require #.

Collecting a large corpus of such queries and classifying them would be very useful in providing a "target" for # researchers.

University of Nebraska Lincoln

DSL Design for Analysis

DSLs are being designed for many domains, e.g., robotics, ecological/biological systems, in which stochasticity is fundamental.

How can the design of these DSLs be set up to facilitate probabilistic analysis?

Can we incorporate notions of input distribution into the language? For instance as an assumption about inputs that can be exploited for analysis and checked at runtime?

Are there notions of probabilistic contract that make sense in this setting that could permit modular analysis?

Moving from explicit to implicit branch probabilities

Popular probabilistic model checking tools like like PRISM and PASS still require that probabilities annotate transitions in the model.

In some cases this is easy, e.g., modeling `bernouli(0.5)`, but in others it is not, e.g., when it is based in complex ways on input data.

Is it possible to incorporate the tracking of symbolic constraints reaching a branch and then using # techniques to eliminate some of the explicit probabilities?

This is tricky because you need to shift from a fully path-sensitive setting, like symbolic execution, to one where paths are merged, but it is possible.

At the very least it seems possible to run a separate analysis that attempts to determine the accuracy of those explicit probabilities?

Residual probabilistic program analysis

In a residual analysis one stages a series of analyses.

An analysis does what it can on the program and then passes the unanalyzed part, the residue, on to the next one in the series.

What if you were to run a probabilistic symbolic execution for whatever resource budget you can tolerate and then target the "unanalyzed" portion of the program using a data flow analysis?

Alternatively you run probabilistic data flow first, then target the regions of the program for which you have imprecise probability information for refinement using symbolic execution.

One could use slicing techniques to extract that residue.

This would combine the precision of the symbolic execution with the scalability of data flow.

Probabilistic program differencing

Probabilistic symbolic execution is particularly well suited for quantifying the difference between two versions of a program [**?**].

This means that it could be used to rank how close a program is to a given oracle program.

This has applications in mutation analysis, program repair, approximate computing or even in marking student assignments.

Symbolically executing "probabilistic programs"

Probabilistic programming is becoming very popular.

Most current approaches rely on sampling.

Statistical symbolic execution can be more efficient because each sample accumulates the `mass` of the sampled path instead of the mass of the single sampled trace.

Extending symbolic execution to support these programs requires support for conditioning, i.e., the `observe(...)` statement.

From a path generation perspective this is trivial, i.e., it is equivalent to support of `assume(...)`.

It is trickier to handle the renormalization of the output distribution; but this seems possible.

Handling incompleteness, e.g., when the probabilistic program iterates over large domains, remains an open challenge.

University of Nebraska Lincoln

Final words

Send us your ideas for improving the briefing paper!

We want it to be a useful resource for the community.