Safely Supporting Probabilistic Data: PL Techniques as Part of the Story

Dan Grossman
University of Washington
Executive summary

• Language design+implementation to help wrangle uncertainty
  – PL concepts+tools are a huge help
  – But also need to learn from statisticians
  ➢ Validate these claims with UW’s approximate computing work

Acknowledgments: 9 co-authors [papers at end]
  – Especially Adrian Sampson, Luis Ceze
  – Including Kathryn and Todd
Types for information flow

```plaintext
int<x> x;
int<y> y;
if(x)
  y = 7; ✗
```

Symbolic execution

```
z = x;
if(x!=0)
  z = x*y;
```

Type inference

```
let f = \( \lambda \) y. y+7
let z = f 9
let q = z && true ✗
```

Function inlining/specialization

```plaintext
int f(int x, int y){
  return x*y;
}
f(0,a)           0
f(3,b)           f(3,b)
f(1,c)           c
```
Background (2/2): Approximation

• Full bit-precision is unnecessary and wastes energy

• Allowing probabilistic [in]correctness can work!
  – Let ALUs and memory produce garbage with low-nonzero probability
  – But most code/programmers want nothing to do with that...
Information flow is *exactly* the right high-level abstraction

- Type qualifier for `@approx`
- Explicit `endorse` as needed

- Convenient:
  - Opt-in with precise default
  - Overloaded operations and methods

- Strong guarantee: Approximate data has no effect on precise data except via `endorse` (classic *non-interference theorem*)

```java
@approx int x = 12;
int y = 27;
✓ y = x*2;
x = y*3;
@approx int z = f(x);
if(looks_okay(z))
  int w = endorse(z);
...```
EnerJ limitations

1. Only “best effort” semantics for approximate computation
   – Encapsulated all the probability, and then ignored it!

2. No approximate control-flow (without **endorse**)
   – Stronger limitation to ensure non-interference, no crashes, no extra non-termination, ...
Adding probabilities (2015)

Address limitation #1 directly:

- `@approx<p>` int: static guarantee that at run-time value will be correct with at least probability p
- Operator uses (e.g., +) also have correctness probability

- EnerJ’s `@approx` is `@approx<0.0>`
- Precise is `@approx<1.0>`
- Natural subtyping: `@approx<p> t <: @approx<q> t if p >= q`

[See also Mike’s Rely and Chisel work (2014)]
Essential additions

- Type inference
  - Programmer can add type information, or prover can provide more annotations
- Problem definitions
  - We use MIP/SMT to ensure logic is correct and get as much energy as possible

Diagram:

Type Annotations -> Z3 SMT Solver
Hardware Parameters -> Z3 SMT Solver
Objective Function -> Z3 SMT Solver

Z3 SMT Solver

- Over-constrained?
  - Type error
- Under-constrained?
  - Great! Try lower objective target
Essential additions

• Type inference, part 1:
  – Programmer states probabilities at key points (inputs, outputs)
  – Automatic solver fills in the rest, and/or programmer can provide more annotations

• Type inference, part 2:
  – Problem often under-constrained; goal is to save as much energy as possible within constraints
  – We use Microsoft’s Z3 solver with a custom objective function

Method specialization
  – Up to $k$ approximation settings for each method

• Opt-in dynamic tracking for loop-carried dependencies
Still not much statistics

• Additions are all “PL bread-and-butter”
• Uses only one trivial statistical fact:
  
  \[
  \begin{align*}
  \approx \int x &= \ldots; \\
  \approx \int y &= \ldots; \\
  x +_\approx y &= \approx \int \pi \approx \int 
  \end{align*}
  \]

  – Result type is precise if \( x, y \) (and addition) are independent
  – Result type is sound regardless of [in]dependence

• Other panelists all make much better use of statistics, like in our probabilistic assertions work...
Probabilistic assertions (2014)

Much richer setting:

- Inputs/values can have arbitrary distributions, not just “Bernouilli failure”
- Dependence tracked via symbolic execution, even through if-statements and some loops

Evaluate arbitrary probabilistic assertions:

```
assert(e, p, c)
```

Key insight:

- Data-structure produced by symbolic execution is an “expression DAG” and a “Bayesian network”
- So apply compiler and statistical optimizations to it
  - Followed by hypothesis testing
The limitation

- EnerJ and follow-on work gave static guarantees regardless of input

- Probabilistic assertions either revalidates for each input (testing) or needs probabilistic assumptions (distributions) of inputs

- Need more research on:
  - Bridging this gap
  - Supporting unbounded loops by soundly trimming low-probability paths
The big context

• “Early days”
  – Excited by the panel’s work, but many open questions...
  – *Technical* questions: (loops, modularity, scale, ...)
  – *Tools* questions, also with some preliminary work
    • Debugging, profiling, monitoring
    • Error messages

• Is “adding statistical properties” to modern language design The True Way Forward or a local optimum to avoid?
To learn more

- **EnerJ: Approximate Data Types for Safe and General Low-Power Computation.** Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, Dan Grossman. PLDI2011

- **Expressing and Verifying Probabilistic Assertions.** Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, Luis Ceze. PLDI2014

- **Probability Type Inference for Flexible Approximate Programming.** Brett Boston, Adrian Sampson, Dan Grossman, Luis Ceze. OOPSLA 2015