From Reliability to Resilience via Program Verification

Michael Carbin
MIT EECS & CSAIL
# Software/Hardware Stack

<table>
<thead>
<tr>
<th>Application</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler and Runtime System</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td></td>
</tr>
<tr>
<td>Hardware System</td>
<td></td>
</tr>
</tbody>
</table>
Software/Hardware Stack

<table>
<thead>
<tr>
<th>Application</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler and Runtime System</td>
<td>Operating System</td>
</tr>
<tr>
<td>Hardware System</td>
<td></td>
</tr>
</tbody>
</table>
Examples

• ~107 Issues in Current Errata for Intel 6th Generation (May 2016)

• SKL057: Cache Performance Monitoring Events May be Inaccurate

• SKL082: Processor May Hang or Cause Unpredictable System Behavior

• “Under complex microarchitecture conditions, processor may hang with an internal timeout error ... or cause unpredictable system behavior.”
In Practice

• **Consumer:** “Intel Skylake bug causes PCs to freeze during complex workloads: Bug discovered while using Prime95 to find Mersenne primes.”
  

• **Supercomputer:** “Researchers performed a study in 2010 on the then most powerful supercomputer, called Jaguar. The study found that [uncorrectable] errors occurred about once every 24 hours in Jaguar’s 360 TB of memory.”
  
What to do?

- **Distinction**: some errors are transient (non-deterministic)

- Logic: complex workloads with multiple architectural events happening concurrently triggers corner case in design (Programming Bugs).

- Electrical: increasingly smaller transistors are more sensitive to physical variation in fabrication process (Physical Bugs)

- Environmental: cosmic-rays
What do you do?

Replication:

\[ z = x + y; \]

Checkable Computations:

\[ x = \text{newton\_method}(f, \text{guess}); \]

\[
\begin{align*}
\text{do}\{ \\
\quad z &= x + y; \\
\quad z' &= x + y; \\
\} \text{ while } (z \neq z')
\end{align*}
\]

\[
\begin{align*}
\text{do}\{ \\
\quad x &= \text{newton\_method}(f, \text{guess}); \\
\} \text{ while } (\text{abs}(f(x)) > \text{eps});
\end{align*}
\]
What if we just let errors *happen*?
Faster and consumes less energy! 

May give the wrong result. 

A different
Different Results

• Produce an inaccurate result
  \[5 + 5 = 8\]

• Produce correct results too infrequently
  \[\Pr(5 + 5 = 10) \text{ too low}\]

• Produce an invalid result
  \[5 + 5 = \text{“hello”}\]

• Crash or do something nefarious
  \[5 + 5 = \text{exec “/bin/launch_missiles”}\]
Approaches

• Self-Stabilizing Algorithms
  • Algorithm-based Fault Tolerance for Matrix Operations (Huang and Abraham, 1984)
  • Fault-Tolerant GMRES (Sao et al., 2011)
  • Self-Stabilizing Conjugate Gradient (Sao et al., 2013)
  • Self-Correct Connected Components (Sao et al., 2016)

• Non-interference + Empirical Guarantees
  • Enerj (Sampson et al. 2011), Truffle (Esmaeilzadeh et al., 2012)
  • ExpAx (Park et al., 2014), FlexJava (Park et al., 2015)

• Traditional Verification
  • Faulty Logic (Meola and Walker, 2010)
  • Relaxed Programs (Carbin et al. 2012)

• Probabilistic Analysis
  • Rely (Carbin et al. 2013), Chisel (Misailovic et al. 2014)
  • Uncertainty Quantification
In Progress Systems

• Leto: Verifying Fault Tolerance with First-Class Execution Models
  • Programmer, platform designer specifies an stateful execution model that prescribes a semantics for the platform
  • Verification system weaves in model and enables check fault tolerance properties such as memory safety,, non-interference, and accuracy
  • Student: Brett Boston

• Shuffle: Typesafe Handcoded Probabilistic Inference
  • NIPS – Machine Learning Systems Dec, 2016
  • Students: Eric Atkinson and Cambridge Yang
Noise Model

\[ f(x) = h(g(x)) \]
\[ \hat{f}(x) = h(g(x) + e_1) + e_2 \quad e_1 \sim N(\mu_1, 1) \quad e_2 \sim N(\mu_2, 1) \]

- Distribution of Error: \( P \left( f(x) - \hat{f}(x) \right) \)
- Expected Error: \( E[f(x) - \hat{f}(x)] \)
- Variance of Error: \( \text{Var}[f(x) - \hat{f}(x)] \)
- Estimation: \( P(\mu_1, \mu_2 | obs) \) where \( obs \) are outputs from the noisy computation
Probabilistic Inference

\[
\Pr(\mu) \\
\Pr(x_i|\mu) \\
\Pr(\mu|x)
\]

```python
def normald(x, mu, var):
    return (1 / (sqrt(2 * pi * var))) * exp(-(((x-mu) ** 2)/(2*var)))

def infer(xs, xvar, mu0, var0):
    varn = 1/((1/var0) + len(xs)/xvar)
    mun = ((mu0/var0) + (sum(xs)/xvar)) * varn
    return mun, varn
```
Probabilistic Inference

\[
\Pr(\mu_j) = N(\mu_j, 0,10)
\]

\[
\Pr(z_i) = U(z_i)
\]

\[
\Pr(x_i | \mu_j, z_i = j) = N(x_i | \mu_j, \sigma^2)
\]

\[
\Pr(\mu | x) = \ldots
\]
Existing Approaches

Correctness Guarantees

Inference Expressiveness

- JAGS, Church, Stan, PSI (Automated Inference)
- Venture, PyMC (Mixed Inference)
- Shuffle (Verified hand-coded Inference)
- C, Python (Hand-coded Inference)
Shuffle

Probabilistic Model

Inference Procedure

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure
model GMM {
    domain Samples, Mus;
    variable R[Samples] obs;
    variable R[Mus] mu;
    variable Mus[Samples] z;

    def muiPrior(i in Mus) : density(mu[i]) =
        normal(mu[i], 0, 100);

    def ziPrior(i in Samples) : density(z[i]) =
        uniform(Mus, z[i]);

    def obsiDensity(i in Samples, j in Mus)
        : density(obs[i] | mu[j], z[i] == j)
        = normal(obs[i], mu[j], 1)
}
model GMM {
  domain Samples, Mus;
  variable R[Samples] obs;
  variable R[Mus] mu;
  variable Mus[Samples] z;

  def muiPrior(i in Mus) : density(mu[i]) = normal(mu[i], 0, 100);

  def ziPrior(i in Samples) : density(z[i]) = uniform(Mus, z[i]);

  def obsiDensity(i in Samples, j in Mus) : density(obs[i] | mu[j], z[i] == j) = normal(obs[i], mu[j], 1)
}
model GMM {
    domain Samples, Mus;
    variable R[Samples] obs;
    variable R[Mus] mu;
    variable Mus[Samples] z;

    def muiPrior(i in Mus) : density(mu[i]) =
        normal(mu[i], 0, 100);

    def ziPrior(i in Samples) : density(z[i]) =
        uniform(Mus, z[i]);

    def obsiDensity(i in Samples, j in Mus)
        : density(obs[i] | mu[j], z[i] == j)
        = normal(obs[i], mu[j], 1)
}
Random Variable Specification

```
variable R[Samples] obs;
variable R[Mus] mu;
variable Mus[Samples] z;
```
variable R[Samples] obs;
variable R[Mus] mu;
variable Mus[Samples] z,
variable R[Samples] obs;
variable R[Mus] mu;
variable Mus[Samples] z;
Random Variable Specification

```plaintext
variable R[Samples] obs;
variable R[Mus] mu;
variable Mus[Samples] z;

j = 0

z[k] == 0

j = 1
```
variable R[Samples] obs;
variable R[Mus] mu;
variable Mus[Samples] z;

j = 0
z[k] == 1

j = 1
```model
GMM {
    domain Samples, Mus;
    variable R[Samples] obs;
    variable R[Mus] mu;
    variable Mus[Samples] z;

    def muiPrior(i in Mus) : density(mu[i]) =
        normal(mu[i], 0, 100);

    def ziPrior(i in Samples) : density(z[i]) =
        uniform(Mus, z[i]);

    def obsiDensity(i in Samples, j in Mus) :
        density(obs[i] | mu[j], z[i] == j)
        = normal(obs[i], mu[j], 1)
}
```
model GMM {
  domain Samples, Mus;
  variable R[Samples] obs;
  variable R[Mus] mu;
  variable Mus[Samples] z;

  def muiPrior(i in Mus) : density(mu[i]) = normal(mu[i], 0, 100);

  def ziPrior(i in Samples) : density(z[i]) = uniform(Mus, z[i]);

  def obsiDensity(i in Samples, j in Mus)
    : density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
}
Probability Densities

\[ f : A \to \mathbb{R} \]
\[ f(x) = \Pr(v \in [x, (x + dx)]) \]

\[ w : \text{Rand}(B) \]
\[ g : B \to \mathbb{R} \]
\[ g(x) = \Pr(w = x) \]
def muiPrior(i in Mus) =
    normal(mu[i], 0, 100);
Density Semantics

```python
def muiPrior(i in Mus) =
    normal(mu[i], 0, 100);
```

Density Primitives
Density Semantics

```
def muiPrior(i in Mus) =
    normal(mu[i], 0, 100);
```
Density Semantics

def muiPrior(i in Mus) =
    normal(mu[i], 0, 100);
model GMM {
    domain Samples, Mus;
    variable R[Samples] obs;
    variable R[Mus] mu;
    variable Mus[Samples] z;

    def muiPrior(i in Mus) : density(mu[i]) =
        normal(mu[i], 0, 100);

    def ziPrior(i in Samples) : density(z[i]) =
        uniform(Mus, z[i]);

    def obsiDensity(i in Samples, j in Mus)
        : density(obs[i] | mu[j], z[i] == j)
        = normal(obs[i], mu[j], 1)
}
model GMM {
    domain Samples, Mus;
    variable R[Samples] obs;
    variable R[Mus] mu;
    variable Mus[Samples] z;

    def muiPrior(i in Mus) : density(mu[i]) =
        normal(mu[i], 0, 100);

    def ziPrior(i in Samples) : density(z[i]) =
        uniform(Mus, z[i]);

    def obsiDensity(i in Samples, j in Mus) :
        density(obs[i] | mu[j], z[i] == j)
        = normal(obs[i], mu[j], 1)
}

Model Specification
Types of Probability Densities

\[ \text{muiPrior} : i. \text{density}(\mu[i]) \]

\[ \text{ziPrior} : i. \text{density}(z[i]) \]

\[ \text{obsiDensity} : i,j. \text{density}(\text{obs}[i] \mid \mu[j], z[i] == j) \]
Type Structure

density(A | B, φ)

• Compare to Pr(A | B)

• A and B are disjoint sets of random variables

• Constrained: supports dynamic affine dependencies

```python
def muiPrior(i in Mus) : density(mu[i])
    = normal(mu[i], 0, 100);

def ziPrior(i in Samples) : density(z[i])
    = uniform(Mus, z[i]);

def obsiDensity(i in Samples, j in Mus) : density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
```
Type Structure

density(A | B, ϕ)

- Compare to Pr(A | B)
- A and B are disjoint sets of random variables
- Constrained: supports dynamic affine dependencies

```python
def muiPrior(i in Mus) :
    density(mu[i]) = normal(mu[i], 0, 100);

def ziPrior(i in Samples) :
    density(z[i]) = uniform(Mus, z[i]);

def obsiDensity(i in Samples, j in Mus) :
    density(obs[i] | mu[j], z[i] == j) = normal(obs[i], mu[j], 1)
```
Type Structure

density(A \mid B, \phi)

• Compare to Pr(A \mid B)

• A and B are disjoint sets of random variables

• Constrained: supports dynamic affine dependencies

```
def muiPrior(i in Mus) : density(mu[i])
    = normal(mu[i], 0, 100);

def ziPrior(i in Samples) : density(z[i])
    = uniform(Mus, z[i]);

def obsiDensity(i in Samples, j in Mus) : density(obs[i] \mid mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
```
Constraints

def obsiDensity(i in Samples, j in Mus):
    density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)

j = 0

j = 1

z[i] == 0
Constraints

def obsiDensity(i in Samples, j in Mus):
    density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
model GMM {
  domain Samples, Mus;
  variable R[Samples] obs;
  variable R[Mus] mu;
  variable Mus[Samples] z;

  def muiPrior(i in Mus) : density(mu[i]) =
      normal(mu[i], 0, 100);

  def ziPrior(i in Samples) : density(z[i]) =
      uniform(Mus, z[i]);

  def obsiDensity(i in Samples, j in Mus)
    : density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
}
def muiPrior(i in Mus) : density(mu[i]) = normal(mu[i], 0, 100);
def ziPrior(i in Samples) : density(z[i]) = uniform(Mus, z[i]);

def obsiDensity(i in Samples, j in Mus)
    : density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
Model Semantics

```python
def muiPrior(i in Mus) : density(mu[i]) = normal(mu[i], 0, 100);
def ziPrior(i in Samples) : density(z[i]) = uniform(Mus, z[i]);

def obsiDensity(i in Samples, j in Mus):
    : density(obs[i] | mu[j], z[i] == j)
    = normal(obs[i], mu[j], 1)
```

\[
Pr(\text{obs}, \text{mu}, \text{z}) = \prod_{j} \text{muiPrior}(j) \prod_{i} \text{ziPrior}(i) \cdot \begin{cases} \text{obsiDensity}(i,j), & z[i] == j \\ 1, & \text{else} \end{cases}
\]
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure

.py
Inference Procedures

Goal Distribution:

density(\mu | obs)

density(z | obs)
Inference Procedures

```python
def zPrior : density(z) = …;
def obsLikelihood : density(obs | z) = …;
def obsZJoint : density(obs, z) = obsLikelihood * zPrior;
def obsMarginal : density(obs) = int obsZJoint z;
def zPosterior : density(z | obs) = obsZJoint / obsMarginal;
```
Inference Procedures

\[
\text{def } z\text{Prior : density}(z) = \ldots ;
\]
\[
\text{def } \text{obsLikelihood : density}(\text{obs} \mid z) = \ldots ;
\]
\[
\text{def } \text{obsZJoint : density}(\text{obs}, z) = \text{obsLikelihood} \times z\text{Prior} ;
\]
\[
\text{def } \text{obsMarginal : density}(\text{obs}) = \text{int } \text{obsZJoint } z ;
\]
\[
\text{def } z\text{Posterior : density}(z \mid \text{obs}) = \text{obsZJoint} / \text{obsMarginal} ;
\]

\[
P(z \mid \text{obs}) = \frac{P(\text{obs} \mid z) \times P(z)}{\int P(\text{obs} \mid z_1) \times P(z_1)dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1)dz_1}
\]
Inference Procedures

```
def zPrior : density(z) = ...;

def obsLikelihood : density(obs | z) = ... ;

def obsZJoint : density(obs, z) = obsLikelihood * zPrior;

def obsMarginal : density(obs) = int obsZJoint z;

def zPosterior : density(z | obs) = obsZJoint / obsMarginal;
```

Compute *prior probability* of entire z vector

\[
P(z | \text{obs}) = \frac{P(\text{obs} | z) \cdot P(z)}{\int P(\text{obs} | z_1) \cdot P(z_1) \, dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1) \, dz_1}
\]
Inference Procedures

\[
P(z \mid obs) = \frac{P(obs \mid z) \cdot P(z)}{\int P(obs \mid z_1) \cdot P(z_1)dz_1} = \frac{P(obs, z)}{\int P(obs, z_1)dz_1}
\]

---

```
def zPrior : density(z) = ...;

def obsLikelihood : density(obs | z) = ...;

def obsZJoint : density(obs, z) = obsLikelihood * zPrior;

def obsMarginal : density(obs) = \int obsZJoint z;

def zPosterior : density(z | obs) = obsZJoint / obsMarginal;
```
Inference Procedures

def zPrior : density(z) = ...;

def obsLikelihood : density(obs | z) = ... ;

def obsZJoint : density(obs, z) = obsLikelihood * zPrior;

def obsMarginal : density(obs) = int obsZJoint z;

def zPosterior : density(z | obs) = obsZJoint / obsMarginal;

\[
P(z | obs) = \frac{P(obs | z) \cdot P(z)}{\int P(obs | z_1) \cdot P(z_1) dz_1} = \frac{P(obs, z)}{\int P(obs, z_1) dz_1}
\]
Inference Procedures

```
def zPrior : density(z) = ...;

def obsLikelihood : density(obs | z) = ... ;

def obsZJoint : density(obs, z) = obsLikelihood * zPrior;

def obsMarginal : density(obs) = int obsZJoint z;

def zPosterior : density(z | obs) = obsZJoint / obsMarginal;
```

Computes marginal likelihood of data

$$P(z | \text{obs}) = \frac{P(\text{obs} | z) \cdot P(z)}{\int P(\text{obs} | z_1) \cdot P(z_1) dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1) dz_1}$$
Inference Procedures

```
def zPrior : density(z) = ...;

def obsLikelihood : density(obs | z) = ...;

def obsZJoint : density(obs, z) = obsLikelihood * zPrior;

def obsMarginal : density(obs) = int obsZJoint z;

def zPosterior : density(z | obs) = obsZJoint / obsMarginal;
```

Apply Bayes’ Rule

\[
P(z | obs) = \frac{P(obs | z) \cdot P(z)}{\int P(obs | z_1) \cdot P(z_1)dz_1} = \frac{P(obs, z)}{\int P(obs, z_1)dz_1}
\]
Inference Procedures

\[
P(z \mid \text{obs}) = \frac{P(\text{obs} \mid z) \times P(z)}{\int P(\text{obs} \mid z_1) \times P(z_1) \, dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1) \, dz_1}
\]
def obsZJoint : density (obs, z) = obsLikelihood * zPrior

Python

#input state assigns values to all random variables
#returns a real number
def obsZJoint(state):
    return obsLikelihood(state) * zPrior(state)

Typing

Γ ⊢ obsLikelihood : density(obs | z)  Γ ⊢ zPrior : density(z)

Γ ⊢ obsLikelihood * zPrior : density(obs, z)

Γ ⊢ e1 : density(A | B)  Γ ⊢ e2 : density(B)

Γ ⊢ e1 * e2 : density(A, B)
Inference Procedures

\[
P(z | \text{obs}) = \frac{P(\text{obs} | z) * P(z)}{\int P(\text{obs} | z_1) * P(z_1)dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1)dz_1}
\]

def zPrior : \text{density}(z) = ...;

def obsLikelihood : \text{density}(\text{obs} | z) = ...;

def obsZJoint : \text{density}(\text{obs}, z) = \text{obsLikelihood} * \text{zPrior};

def obsMarginal : \text{density}(\text{obs}) = \text{int} \text{obsZJoint} \text{ z};

def zPosterior : \text{density}(z | \text{obs}) = \text{obsZJoint} / \text{obsMarginal};

Computes marginal likelihood of data
Integration (Summation)

\[
\text{def obsMarginal : density (obs) = int obsZJoint z;}
\]

Python

```
def obsMarginal(state):
    ret = 0
    for z in exprange(Samples, Mus):
        state' = state.clone()
        state'.z = z
        ret += obsZJoint(state')
    return ret
```

Typing

\[
\Gamma \vdash \text{obsZJoint : density (obs, z)} \quad \Gamma \vdash \text{e : density(A, B)}
\]

\[
\Gamma \vdash \text{int obsZJoint z : density(obs)} \quad \Gamma \vdash \text{int e B : density(A)}
\]

Computes the set \( \text{Mus}^{\text{Samples}} \)
Inference Procedures

\[
P(z | \text{obs}) = \frac{P(\text{obs} | z) \cdot P(z)}{\int P(\text{obs} | z_1) \cdot P(z_1)dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1)dz_1}
\]

Apply Bayes’ Rule
Division (Bayes’ Rule)

\[
def \text{zPost} : \text{density} (z \mid obs) = \text{obsZJoint} / \text{obsMarginal};
\]

Python

```python
def zPost(state):
    return obsZJoint(state) / obsMarginal(state)
```

Typing

\[
\begin{align*}
\Gamma & \vdash \text{obsZJoint} : \text{density}(\text{obs}, z) & \Gamma & \vdash \text{obsMarginal} : \text{density}(\text{obs}) \\
\Gamma & \vdash \text{zJoint} / \text{obsPrior} : \text{density}(z \mid \text{obs}) \\
\Gamma & \vdash \text{e1} : \text{density}(\text{A}, \text{B}) & \Gamma & \vdash \text{e2} : \text{density}(\text{B}) \\
\Gamma & \vdash \text{e1} / \text{e2} : \text{density}(\text{A} \mid \text{B})
\end{align*}
\]
Inference Procedure

- Arithmetic operators
- Integrals
- Definition
- Invocation
- Independence
- Primitive recursion
- Conditionals
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure
Independence

\[ \text{z[i]} \parallel \text{mu[j]} \]
Assumptions

• Independence

    // Assuming ziPrior : i. density(z[i])

    def independent ziPriorI(i in dataPoints):
        density(z[i] | mu[j]) = ziPrior(i);
Assumptions

• Independence

```python
// Assuming ziPrior : i. density(z[i])

def independent ziPriorI(i in dataPoints):
    density(z[i] | mu[j]) = ziPrior(i);
```

• Saturation

• Annotated by developer, but recorded and reported in a log by Shuffle.
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure

.py
Shuffle

- Probabilistic Model
- Inference Procedure
- Type Checker
- Extractor
- Statistical Assumptions

Extracted Inference Procedure
def zPosterior : density(z | obs) = obsZJoint / obsMarginal;

def zPosterior(state) :
    return obsZJoint(state) / obsMarginal(state);
Integrals

• Simplify known opportunities for closed forms: For example, the conjugate prior in the posterior distribution:

\[
P(mu[j] \mid obs[i]) = \frac{P(obs[i] \mid mu[j]) \cdot P(mu[j])}{\int P(obs[i] \mid mu_1) \cdot P(mu_1) \cdot dmu_1}
\]

• If normal, \( Pr(mu[j] \mid obs[i]) = \frac{\mu_0}{\sigma_0^2} + \frac{obs[i]}{\sigma^2} \)

\[
= \frac{1}{\sigma_0^2 + \frac{1}{\sigma^2}}
\]
Pattern Matching

\[
\frac{\text{normal}(\text{obs}[i], \mu[j], 1) * \text{normal}(\mu[j], 0, 10)}{\text{int normal}(\text{obs}[i], \mu[j], 1) * \text{normal}(\mu[j], 0, 10) \text{ by } \mu[j]}
\]

\[
\text{normal}(\mu[j], (\text{obs}[i]/1)/(1/10 + 1/1), 1/(1/10 + 1/1))
\]
Automatic Incrementalization

- Commutative and associative reductions with overlapping ranges
- Shuffle’s language is such that determining if two iteration ranges overlap is computable

```python
accs = []
for i in range(N):
    acc = 0
    for j in range(i):
        acc += obs[j]
    accs += [acc]
```

```python
accs = []
acc = 0
for i in range(N):
    acc += obs[i]
accs += [acc]
```
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure
Shuffle

- Probabilistic Model
- Inference Procedure
- Type Checker
- Extractor
- Statistical Assumptions
- Extracted Inference Procedure
Type Checking

• Preservation:

\[ \Gamma \vdash d : \text{density}(A \mid B) \Rightarrow \forall \sigma. d(\sigma) = \Pr(A \mid B) \]

• Progress:

  • Well-typed terms evaluate successfully
Type Checking

density(A | B, \phi)
• Must verify that A \cap B = \emptyset and FRV(\phi) \subseteq B

\text{density}(A | B, \phi) \rightarrow \text{density}(A' | B', \phi')
• Must verify A \equiv A', B \equiv B' and \phi' \Rightarrow \phi

• Formalize with QF theory of arrays and BitVectors (Solve with Z3)
Variable Sets

\[
density(z[i] \mid z\{k : k \neq i\}, \text{obs})
\]
\[
density(z[0] \mid z\{k : k \neq 0\}, \text{obs})
\]
\[
density(z[1] \mid z\{k : k \neq 1\}, \text{obs})
\]
\[
density(z[2] \mid z\{k : k \neq 2\}, \text{obs})
\]

\[|\text{Samples}| = 3\]
# Benchmarks

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Shuffle Code</th>
<th>Generated Code (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Model/Inference LoC)</td>
<td></td>
</tr>
<tr>
<td>Burglary</td>
<td>28/30</td>
<td>36</td>
</tr>
<tr>
<td>Context-Specific Inference</td>
<td>26/30</td>
<td>24</td>
</tr>
<tr>
<td>Dirichlet-Categorical</td>
<td>11/20</td>
<td>20</td>
</tr>
<tr>
<td>Healthiness</td>
<td>76/101</td>
<td>118</td>
</tr>
<tr>
<td>Hurricane</td>
<td>29/29</td>
<td>12</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>29/229</td>
<td>318</td>
</tr>
<tr>
<td>Normal-Normal</td>
<td>11/20</td>
<td>27</td>
</tr>
<tr>
<td>Weather</td>
<td>12/20</td>
<td>24</td>
</tr>
</tbody>
</table>
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Shuffle Code (Model/Inference LoC)</th>
<th>Generated Code (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burglary</td>
<td>28/30</td>
<td>36</td>
</tr>
<tr>
<td>Context-Specific Inference</td>
<td>26/30</td>
<td>24</td>
</tr>
<tr>
<td>Dirichlet-Categorical</td>
<td>11/20</td>
<td>20</td>
</tr>
<tr>
<td>Healthiness</td>
<td>76/101</td>
<td>118</td>
</tr>
<tr>
<td>Hurricane</td>
<td>29/29</td>
<td>12</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>29/229</td>
<td>318</td>
</tr>
<tr>
<td>Normal-Normal</td>
<td>11/20</td>
<td>27</td>
</tr>
<tr>
<td>Weather</td>
<td>12/20</td>
<td>24</td>
</tr>
<tr>
<td>Gaussian Mixture Model</td>
<td>19/93</td>
<td>55</td>
</tr>
<tr>
<td>Latent Dirichlet Allocation</td>
<td>21/160</td>
<td>293</td>
</tr>
<tr>
<td>Simultaneous Localization and Mapping</td>
<td>40/68</td>
<td>38</td>
</tr>
</tbody>
</table>
Inference Procedures

\[
P(z | \text{obs}) = \frac{P(\text{obs} | z) * P(z)}{\int P(\text{obs} | z_1) * P(z_1) dz_1} = \frac{P(\text{obs}, z)}{\int P(\text{obs}, z_1) dz_1}
\]

- \text{def } z\text{Prior} : \text{density}(z) = \ldots;
- \text{def } \text{obsLikelihood} : \text{density}(\text{obs} | z) = \ldots;
- \text{def } \text{obsZJoint} : \text{density}(\text{obs}, z) = \text{obsLikelihood} * z\text{Prior};
- \text{def } \text{obsMarginal} : \text{density}(\text{obs}) = \text{int } \text{obsZJoint} \text{ z};
- \text{def } z\text{Posterior} : \text{density}(z | \text{obs}) = \text{obsZJoint} / \text{obsMarginal};

Computes marginal likelihood of data
Integration (Summation)

def obsMarginal : density (obs) = int obsZJoint z;

```python
def obsMarginal(state):
    ret = 0
    for z in exprange(Samples,Mus):
        state’ = state.clone()
        state’.z = z
        ret += obsZJoint(state’)
    return ret
```

Computes the set Mus^{Samples}
Approximate Inference

• Alternative: build a sampler for $P(z \mid obs)$

• For example, given a boolean predicate $pred : Mus[Samples] \rightarrow bool$

```python
def muApprox (state, count, pred):
    sum = 0
    total = 0

    for state in [zSample(state) for x in range(count)]:
        sum = sum + (1 if pred(state.z) else 0)
        total = total + weight

    return sum / total
```

Consider $pred = \lambda x : (z[0] \neq z[1])$
Approximate Inference

\[ \Gamma \vdash d : \text{density}(A \mid B) \Rightarrow \forall \sigma. d(\sigma) = \Pr(A \mid B) \]

\[ \Gamma \vdash s : \text{sampler}(A \mid B) \Rightarrow \forall \sigma. \int_{sr} f(s(\sigma, sr)) = \int_{A} f(x) \ast P(x \mid B) \]

\[ \Gamma \vdash k : \text{kernel}(A \mid B) \Rightarrow \forall \sigma. \int_{sr_1} \int_{sr_2} f(k(s(\sigma, sr_1), sr_2)) = \int_{A} f(x) \ast P(x \mid B) \]

\[ \Gamma \vdash k : \text{estimator}(A \mid B) \Rightarrow \forall \sigma. \int_{sr} \frac{f(s(t(e(\sigma, sr))) \ast f(snd(e(\sigma, sr))))}{\int_{sr} f(st(e(\sigma, sr))} \int_{A} f(x) \ast P(x \mid B) \]

MCMC

Importance Sampling
Approximate Inference

• Sampling
• Composition
• Lifting
• Factor
• Definition
• Invocation
• Independence
• Primitive recursion
• Conditionals

\[
\text{z}[i] := \text{sample } d
\]
\[
\text{lift, lift } e \text{ by } d
\]
\[
\text{d}(x,y) = \text{independent}
\]
\[
\text{if (c) } \text{d1 } \text{else } \text{d2}
\]
# Small Scale Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Algorithm</th>
<th>Assumptions</th>
<th>Speedup vs Venture</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ind.</td>
</tr>
<tr>
<td>Gaussian Mixture Model</td>
<td>Collapsed Gibbs Sampler</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Latent Dirichlet Allocation (LDA)</td>
<td>Collapsed Gibbs Sampler</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Simultaneous Mapping and Location (SLAM)</td>
<td>Rao-Blackwellized Particle Filter</td>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>
Automatic Incrementalization

- Commutative and associative reductions with overlapping ranges
- Shuffle’s language is such that determining if two iteration ranges overlap is computable

```python
accs = []
for i in range(N):
    acc = 0
    for j in range(i):
        acc += obs[j]
    accs += [acc]

accs = []
acc = 0
for i in range(N):
    acc += obs[i]
    accs += [acc]
```
# Performance at Scale

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Algorithm</th>
<th>Assumptions</th>
<th>Speedup vs Venture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Mixture Model</td>
<td>Collapsed Gibbs Sampler</td>
<td>7</td>
<td>34x</td>
</tr>
<tr>
<td>Latent Dirichlet Allocation (LDA)</td>
<td>Collapsed Gibbs Sampler</td>
<td>5</td>
<td>50x</td>
</tr>
<tr>
<td>Simultaneous Mapping and Location (SLAM)</td>
<td>Rao-Blackwellized Particle Filter</td>
<td>13</td>
<td>1.3x</td>
</tr>
</tbody>
</table>
Existing Approaches

Correctness

Guarantees

Inference Expressiveness

- JAGS
- Church
- Stan
- PSI
- Automated Inference

- Venture
- PyMC
- Mixed Inference

- Shuffle
- Verified hand-coded Inference

- Hand-coded Inference
- C
- Python

Verified hand-coded Inference
Shuffle

Probabilistic Model

Inference Procedure

Shuffle

Type Checker

Extractor

Statistical Assumptions

Extracted Inference Procedure
Conclusion

• Many opportunities for resilience
  • Mechanism for dealing with inherently unreliable hardware
  • Mechanism for increased performance (up to 7x)
  • It’s also possible to verify the resulting applications
Takeaway: Methodology for Programming General Uncertain Computations
Verifying Fault Tolerance with First-class Execution Models.
Brett Boston and Michael Carbin. In Submission
Leto: Verifying Application-Specific Fault Tolerance with First-Class Execution Models
First-Class Execution Models (Jacobi)

```java
spec bool last_upset = model.upset;
while (...) {
    for (int i = 0; i < x.length; ++i) {
        float sigma = 0;
        for (int j = 0; j < x.length; ++j) {
            if (j != i) {
                float delta = A[i][j] * . last_x[j];
                sigma = sigma + . delta;
            }
        }
        float num = b[i] -. sigma;
        x[i] = num /. A[i][i];
    }
    assert (!last_upset && model.upset) ->
        (norm2(x<o> - x<r>) < eps))
    last_x = x;
    last_upset = model.upset;
}
```

- Reflect on fault model state
- Unreliable Computation
- Relational Assertion: bound difference in solution vector
Logics for Verifying Properties

1. Safety – properties required to produce a valid result
   \[ \text{assert } (x \neq \text{null}) \land x_{<o>} = x_{<r>} \models x_{<r>} \neq \text{null} \]

2. Accuracy – worst-case difference in program result
   \[ \text{assert}_r \ |res_{<o>} - res_{<r>}| \leq 0.02 \times res \]
Result

• **Model:** First class fault model specification

• **Semantics:** system weaves fault model into program semantics

• **Verification:** automatically generate *relational* weakest preconditions and discharge using SMT

• **Broad Motivation:** model adversarial environments such as hardware faults (unreliable compute, memory (RowHammer)) and system attackers
First-Class Execution Model

• Rowhammer
• Approximate Multiplication
• Chaos