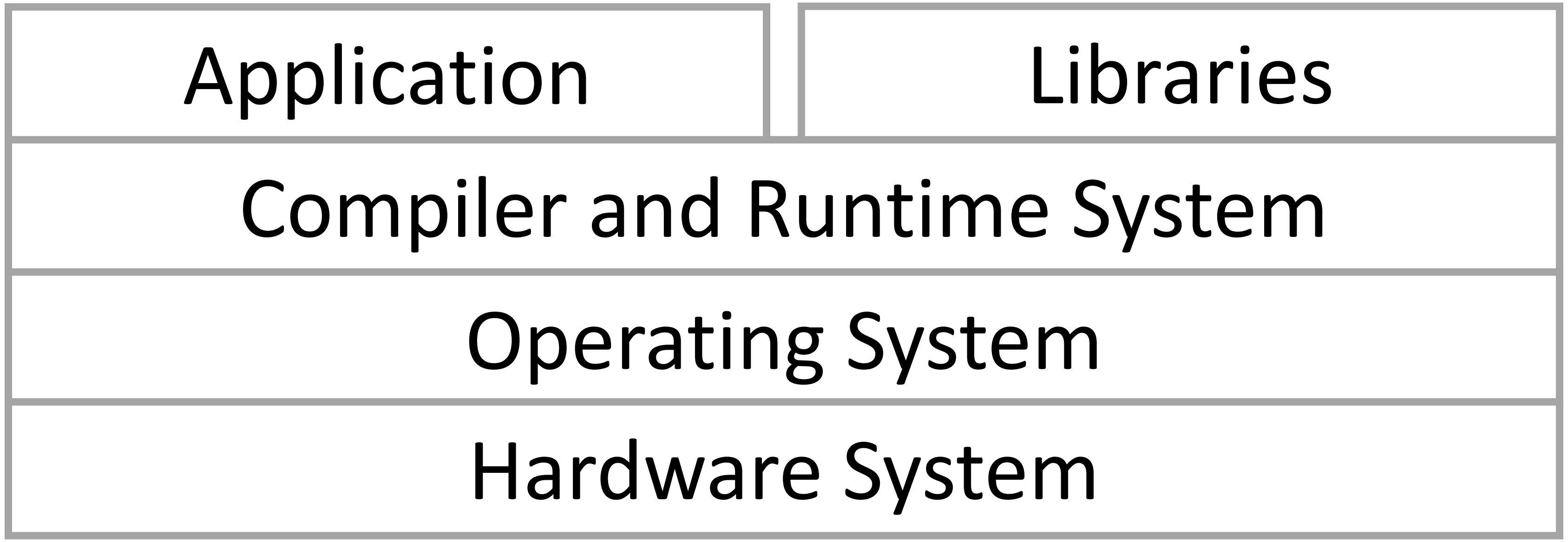
From Reliability to Resilience via Program Verification

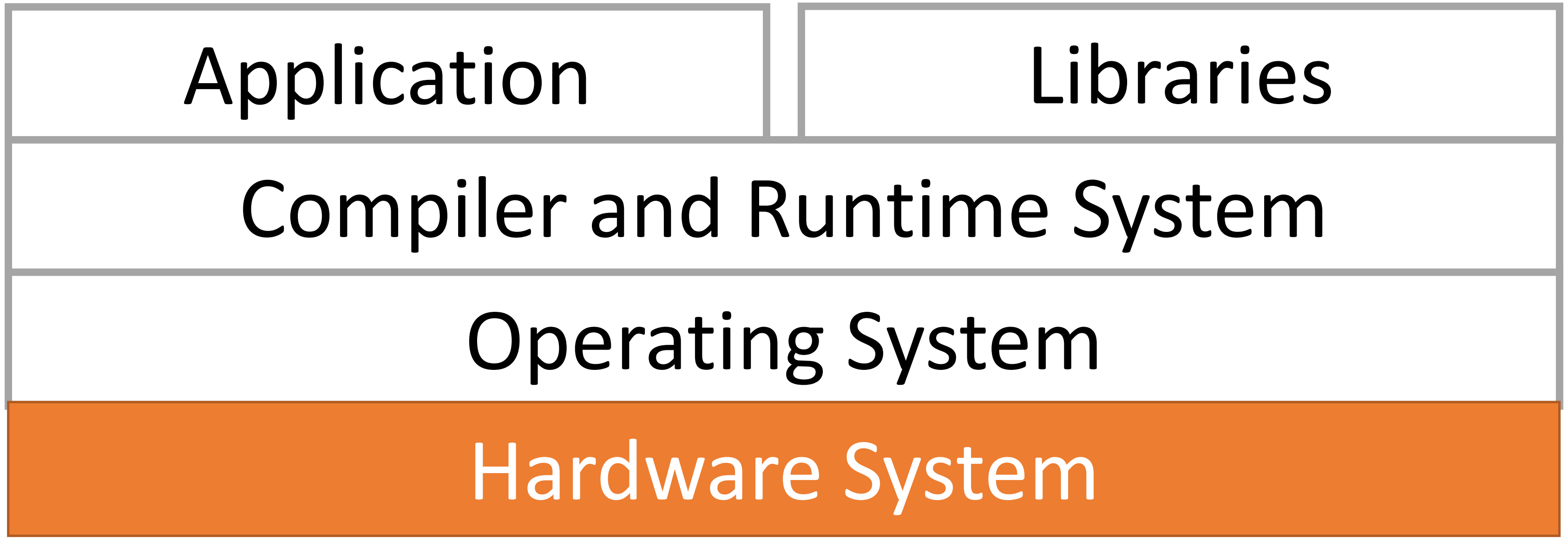
Michael Carbin

MIT EECS & CSAIL

Software/Hardware Stack



Software/Hardware Stack



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.”
  - http://arstechnica.com/gadgets/2016/01/intel-skylake-bug-causes-pcs-to-freeze-during-complex-workloads/
- **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.”
  - http://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder

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;
```

⇨

```
do {
    z  = x + y;
    z’ = x + y;
} while (z != z’)
```

Checkable Computations:

```
x = newton_method(f, guess);
```

⇨

```
do {
    x = newton_method(f, guess);
} while (abs(f(x)) > eps);
```

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

  $\Pr(5 + 5 = 10)$ too low

- Produce an invalid result

  5 + 5 = "hello"

- Crash or do something nefarious

  5 + 5 = 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

$$f(x) = h(g(x))$$

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

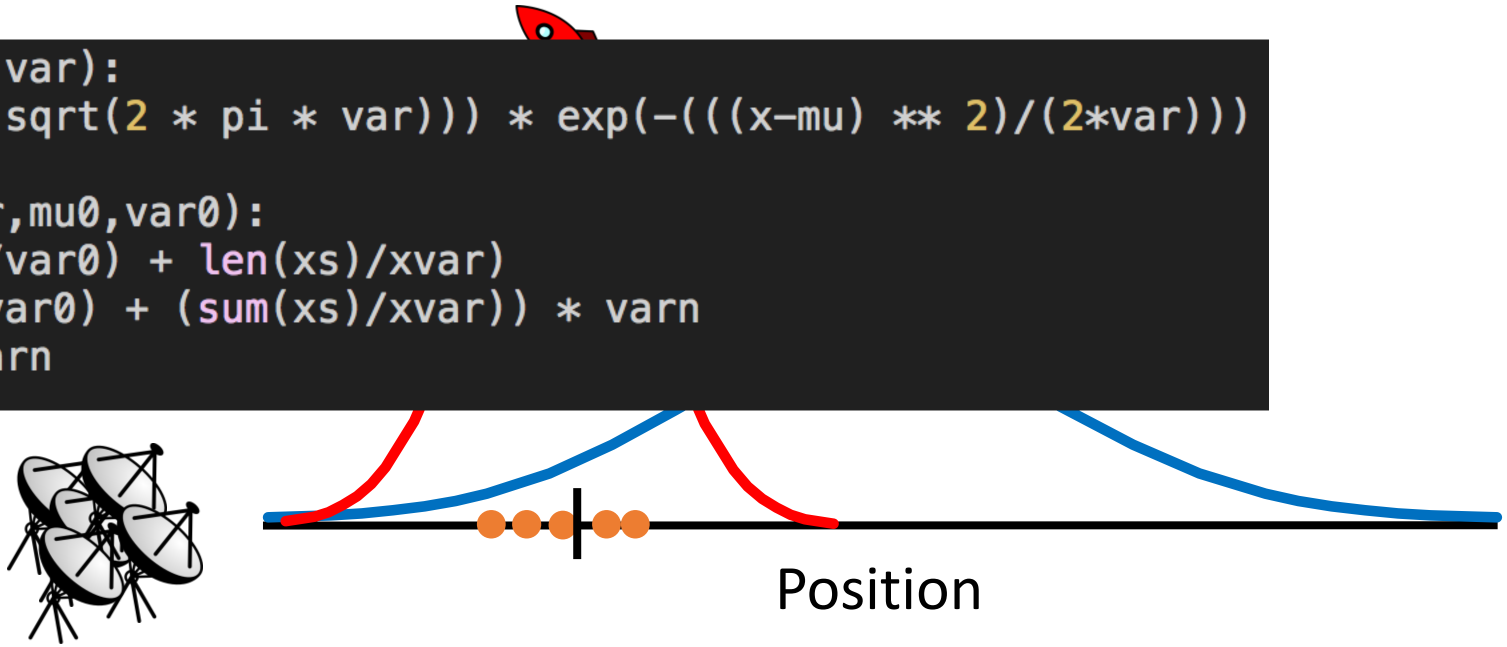
- Distribution of Error: $P\left(f(x) - \hat{f}(x)\right)$
- Expected Error : $E[f(x) - \hat{f}(x)]$
- Variance of Error $\mathrm{Var}[f(x) - \hat{f}(x)]$
- Estimation: $P(\mu_1, \mu_2 | \ obs)$ where obs are outputs from the noisy computation

Probabilistic Inference

$\Pr(\mu)$

```
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
```

$\Pr(x_i|\mu)$



$\Pr(\mu|x)$

Probabilistic Inference

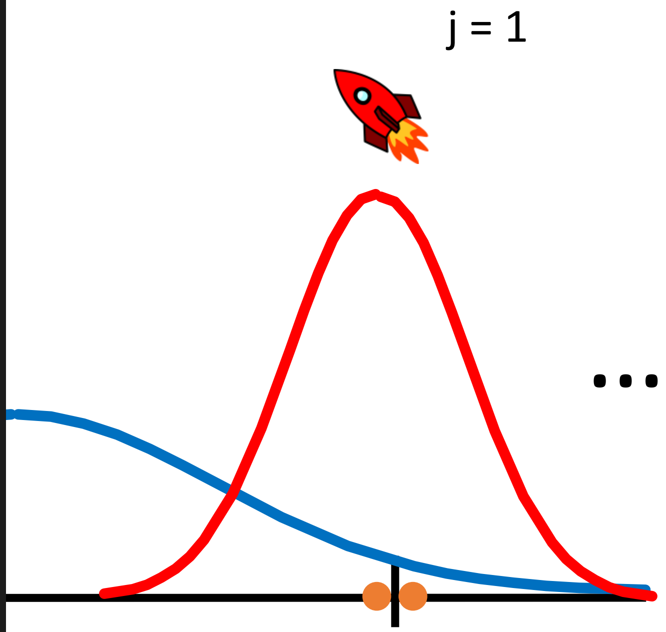
$$\Pr(\mu_j) = N(\mu_j, 0{,}10)$$

$$\Pr(z_i) = U(z_i)$$

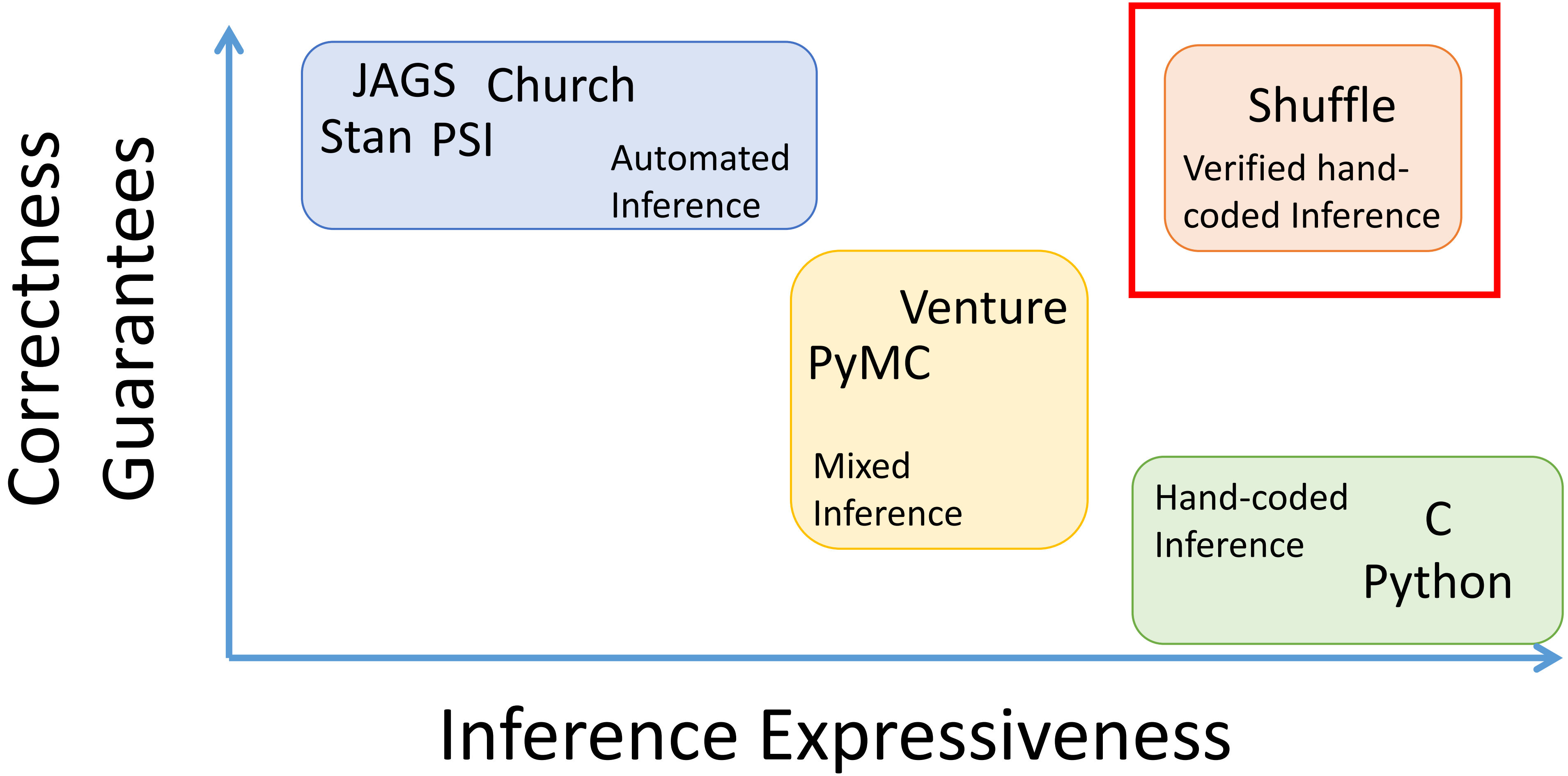
$$\Pr(x_i|\mu_j, z_i = j) = N(x$$

$$\Pr(\mu|x) = \cdots$$

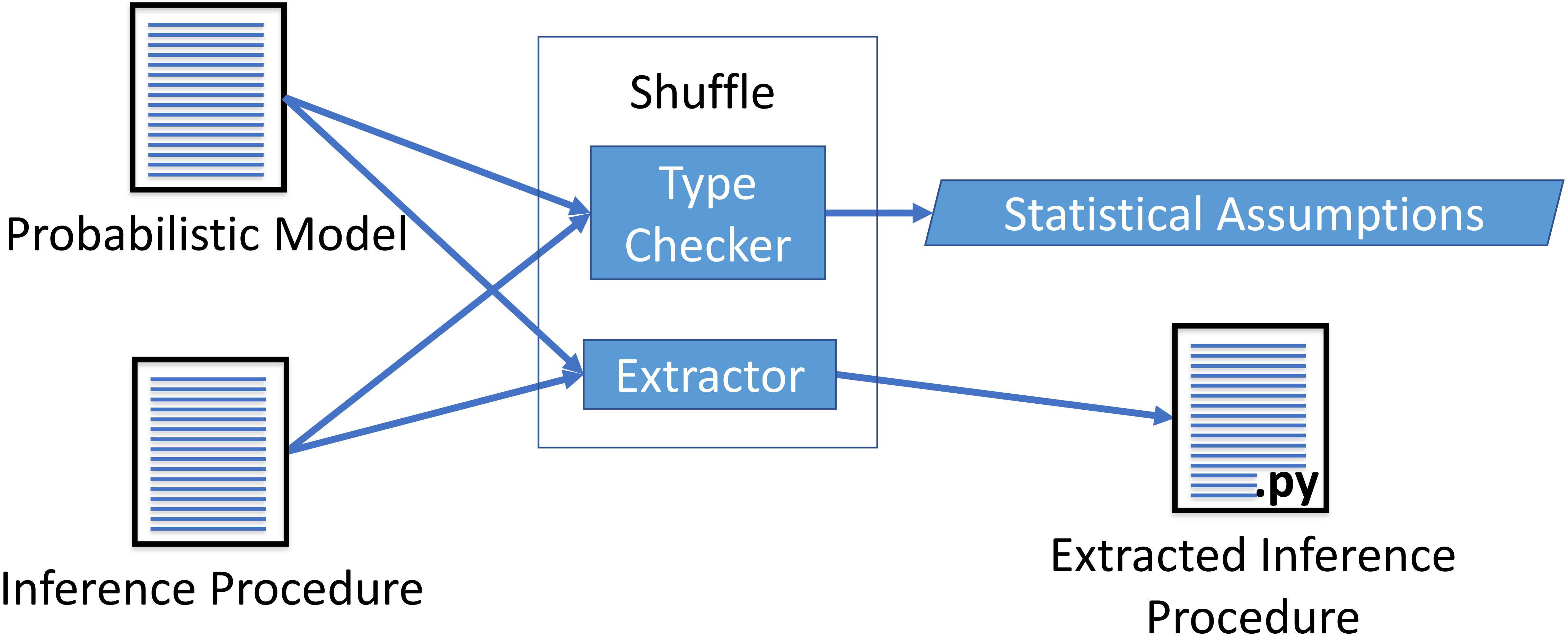
```
def normald(x,mu,var):
    return (1 / (sqrt(2 * pi * var))) * exp(-(((x-mu) ** 2)/(2*var)))

def conjugate(xs,xvar,mu0,var0):
    varn = 1/((1/var0) + len(xs)/xvar)
    mun = ((mu0/var0) + (sum(xs)/xvar)) * varn
    return mun,varn

def obscond(obsi,i,obsprev,z,xvar,var0,mu0):
    relobs = []
    for ii in range(0,i):
        if z[ii] == z[i]:
            relobs += [obs[ii]]

    muc, varc = conjugate(relobs,xvar,mu0,var0)
    return normald(obsi,muc,varc + xvar)

def obsdens(obs,z,xvar,var0,mu0):
    p = 1
    obsprev = []
    for i in range(len(obs)):
        p *= obscond(obs[i],i,obsprev,z,xvar,var0,mu0)
        obsprev += [obs[i]]
    return p

def zdens(z,obs,xvar,var0,mu0):
    s = 0
    for z0 in range(0,2):
        for z1 in range(0,2):
            for z2 in range(0,2):
                for z3 in range(0,2):
                    for z4 in range(0,2):
                        for z5 in range(0,2):
                            zp = [z0,z1,z2,z3,z4,z5]
                            s += obsdens(obs,zp,xvar,var0,mu0)
    return obsdens(obs,z,xvar,var0,mu0) / s

def mupostf(mu,j,z,obs,xvar,var0,mu0):
    relobs = []
    for i in range(len(obs)):
        if z[i] == j:
            relobs += [obs[i]]

    return conjugate(relobs,xvar,mu0,var0)

def finalpost(mu,j,obs,xvar,var0,mu0):
    s = 0
    for z1 in range(0,2):
        for z2 in range(0,2):
            for z3 in range(0,2):
                for z4 in range(0,2):
                    for z5 in range(0,2):
                        z = [0,z1,z2,z3,z4,z5]
                        mup,varp = (mupostf(mu,j,z,obs,xvar,var0,mu0))
                        s += 2 * (zdens(z,obs,xvar,var0,mu0)) * normald(mu,mup,varp)
    return s
```



Existing Approaches

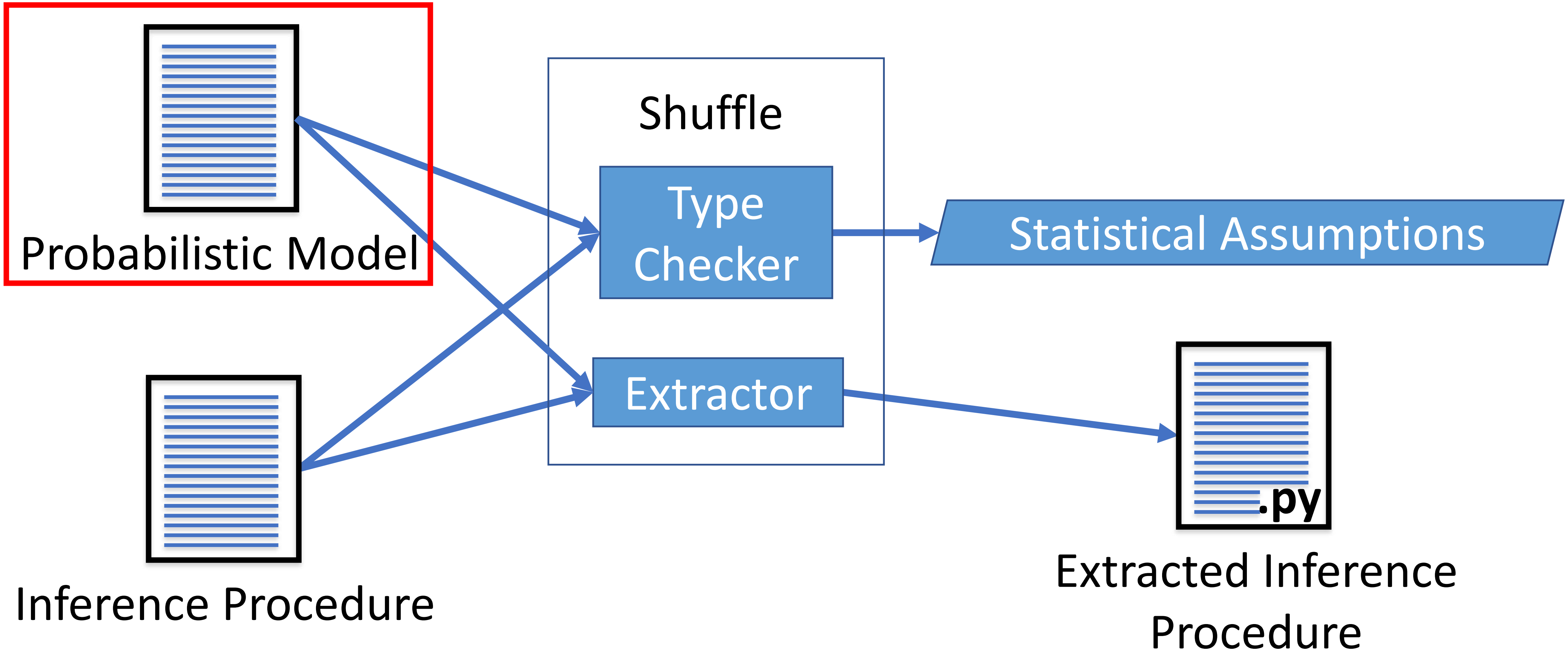


Shuffle



Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Shuffle



Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Model Specification

```
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

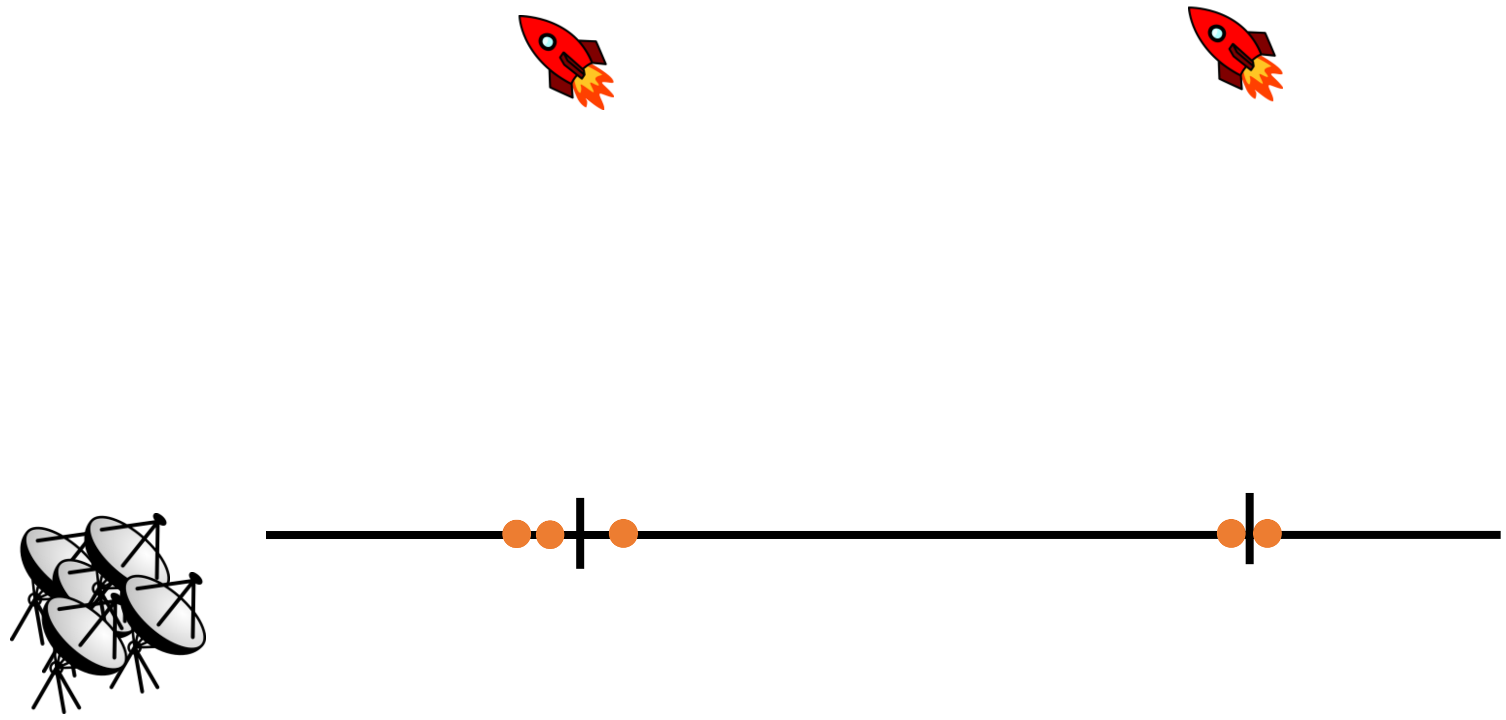
```
model GMM {
   domain Samples, Mus;
   variable R[Samples] obs;
   variable R[Mus] mu;
   variable Mus[Samples] z;

   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

```
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)
}
```

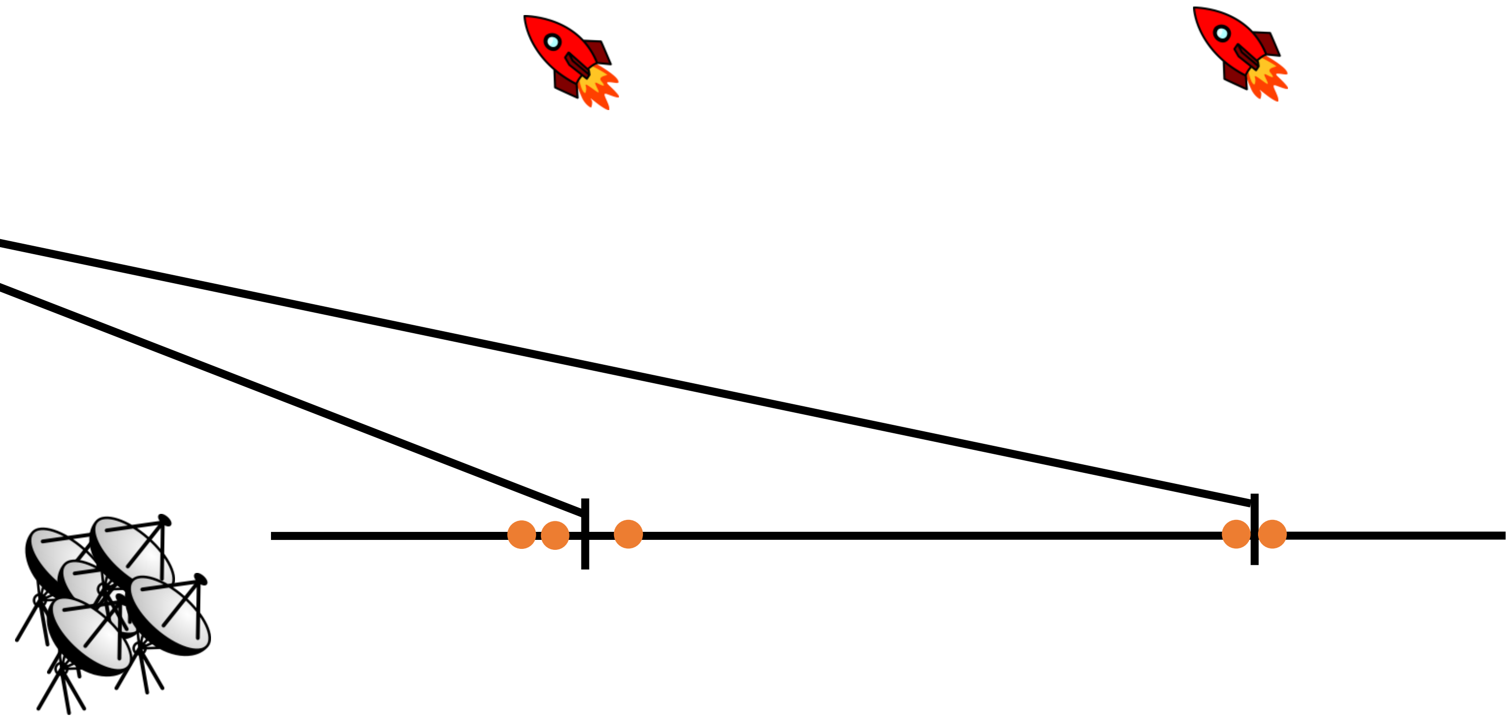
Random Variable Specification

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



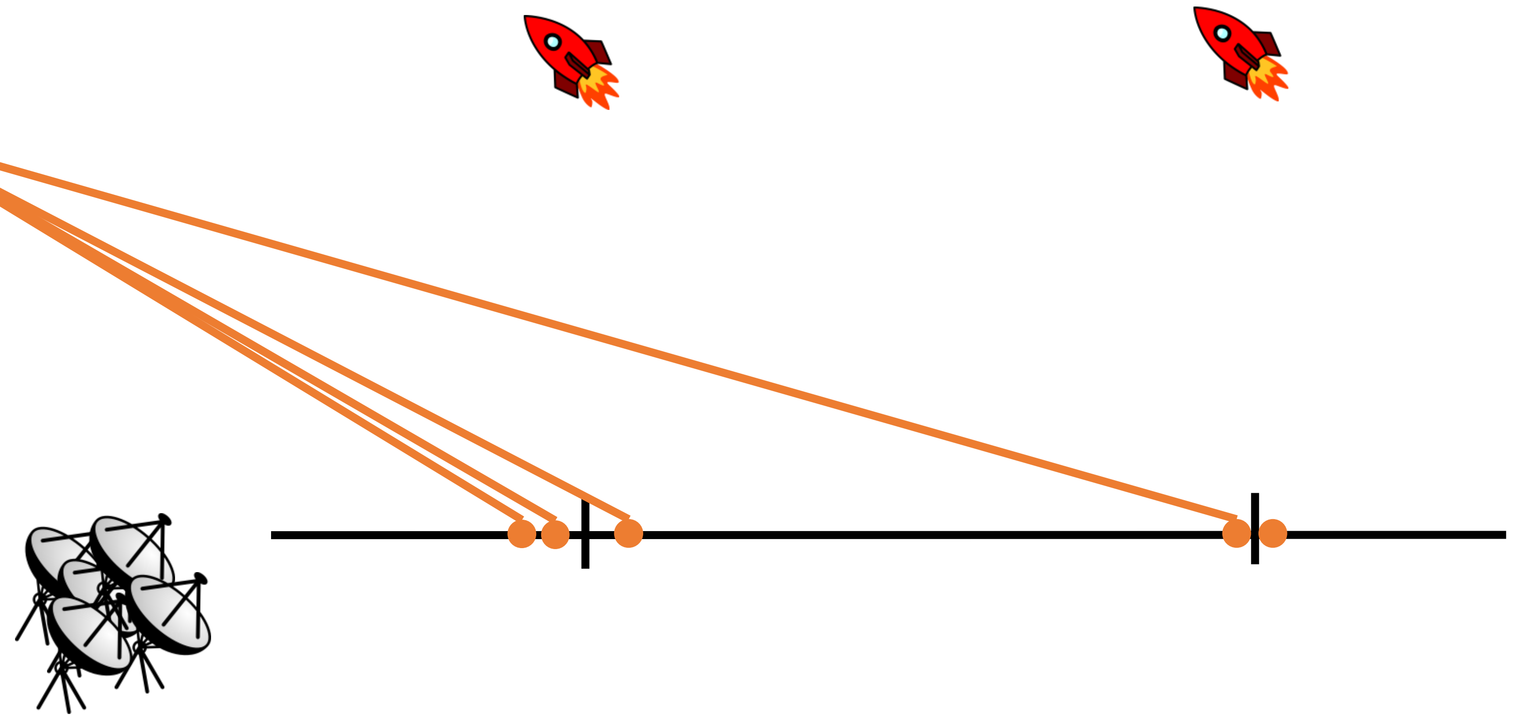
Random Variable Specification

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



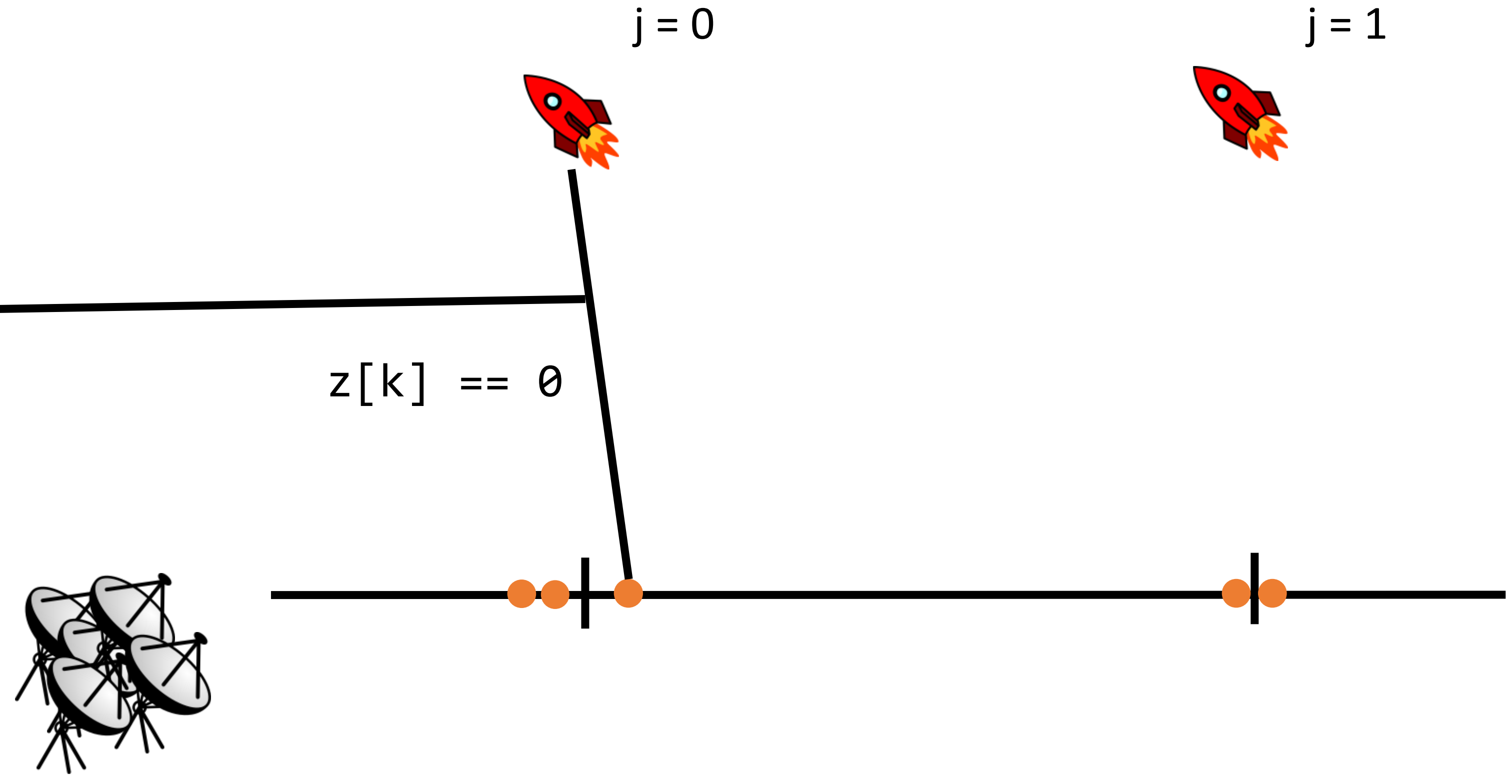
Random Variable Specification

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



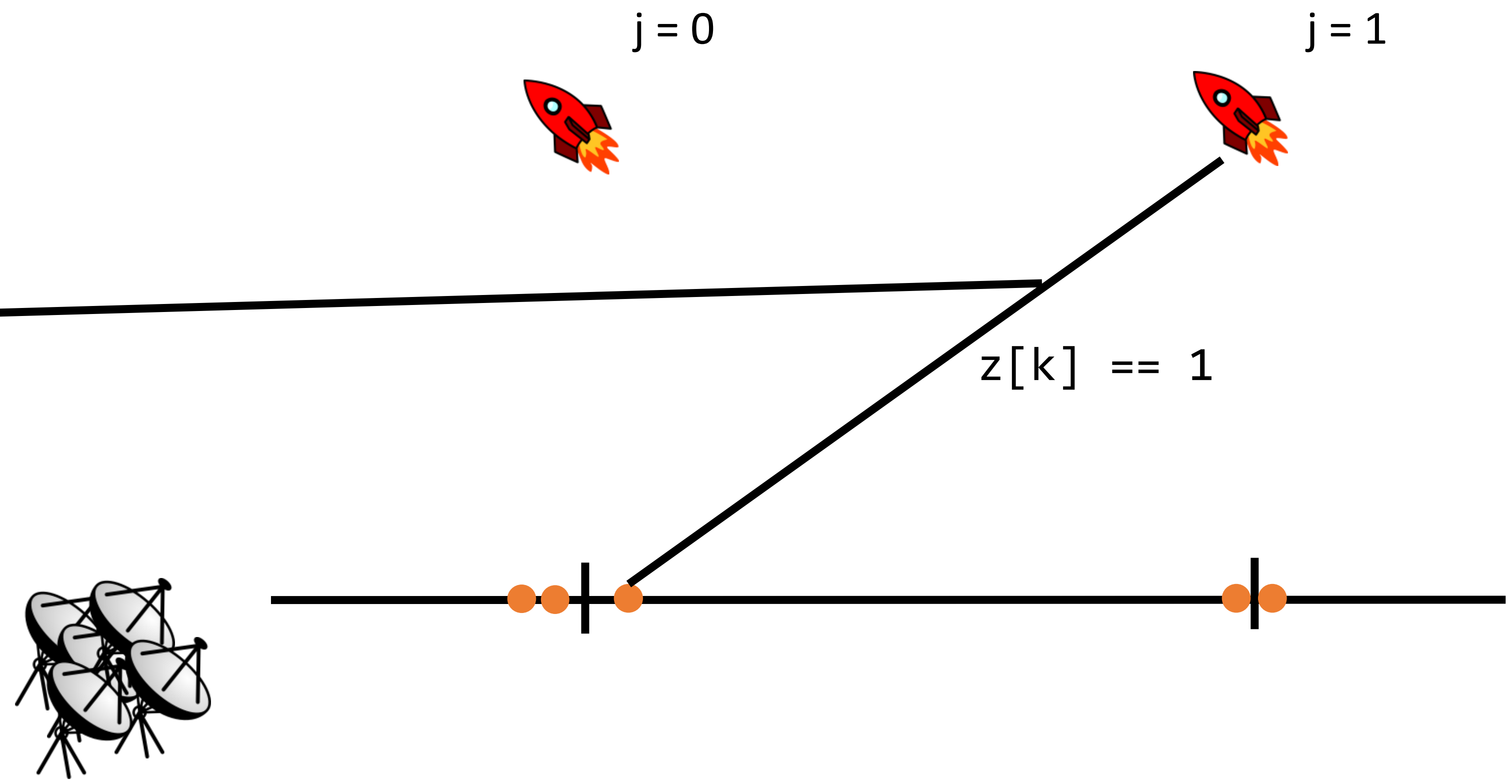
Random Variable Specification

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



Random Variable Specification

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



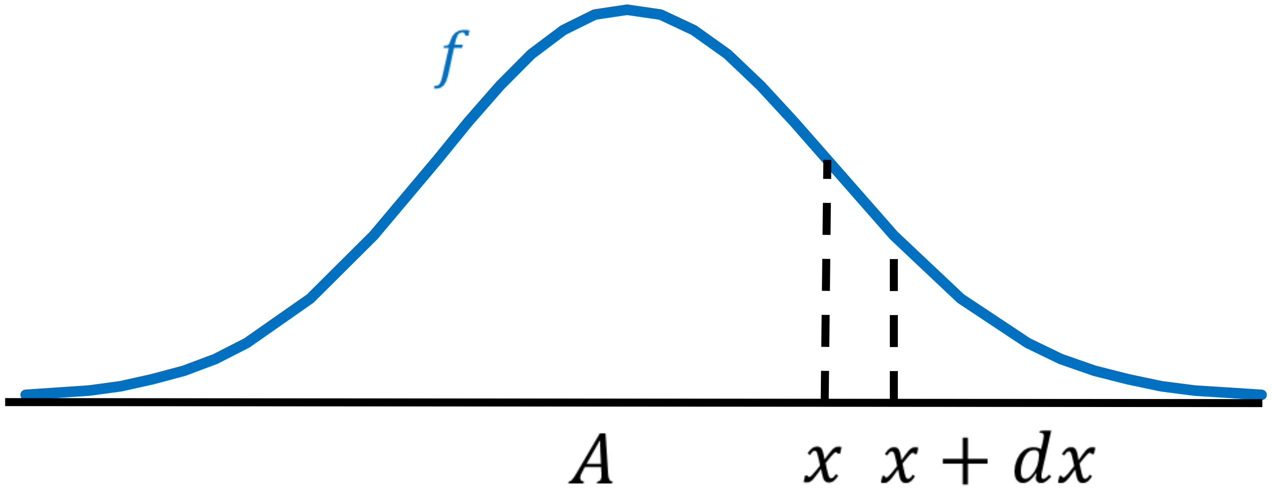
Model Specification

```
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

```
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)
}
```

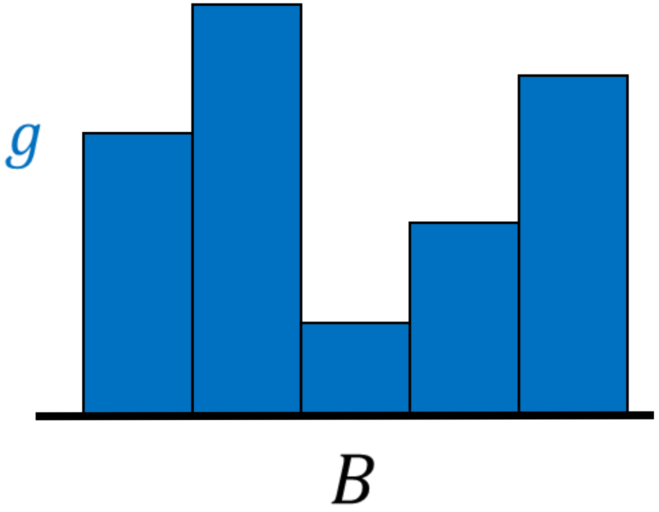
Probability Densities



$v : Rand(A)$

$f : A \to \mathbb{R}$

$f(x) = \Pr(v \in [x, (x + dx)])$



$w : Rand(B)$

$g : B \to \mathbb{R}$

$g(x) = \Pr(w = x)$

Density Semantics

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

Density Semantics

```
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 Primitives

Random Variables

Density Semantics

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

Density Primitives   Random Variables   Quantified Variables

Model Specification

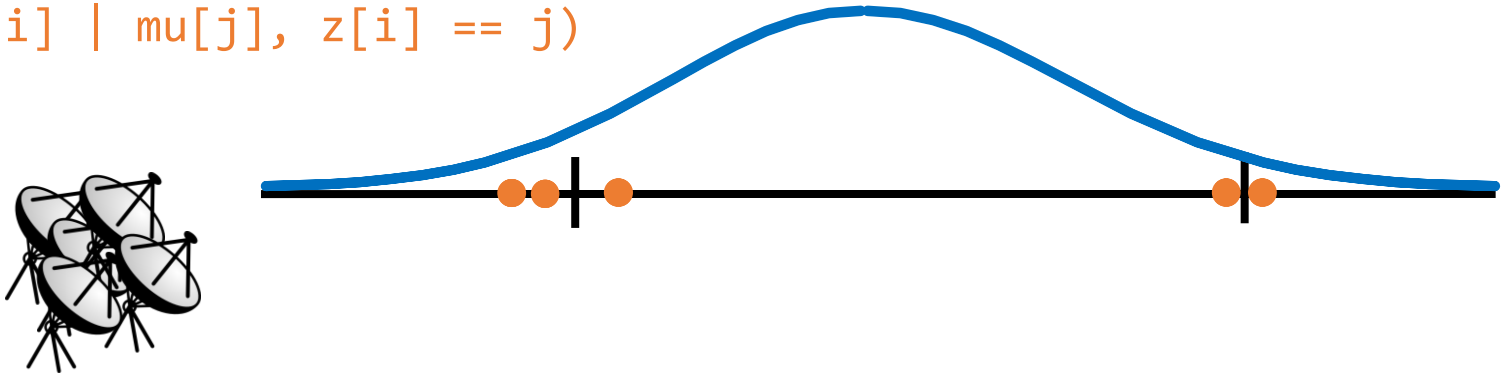
```
model GMM {
   domain Samples, Mus;
   variable R[Samples] obs;
   variable R[Mus] mu;
   variable Mus[Samples] z;

   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

```
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)
}
```

Types of Probability Densities

```
muiPrior : i. density(mu[i])

 ziPrior : i. density(z[i])

obsiDensity : i,j. density(obs[i] | 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

```
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

```
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

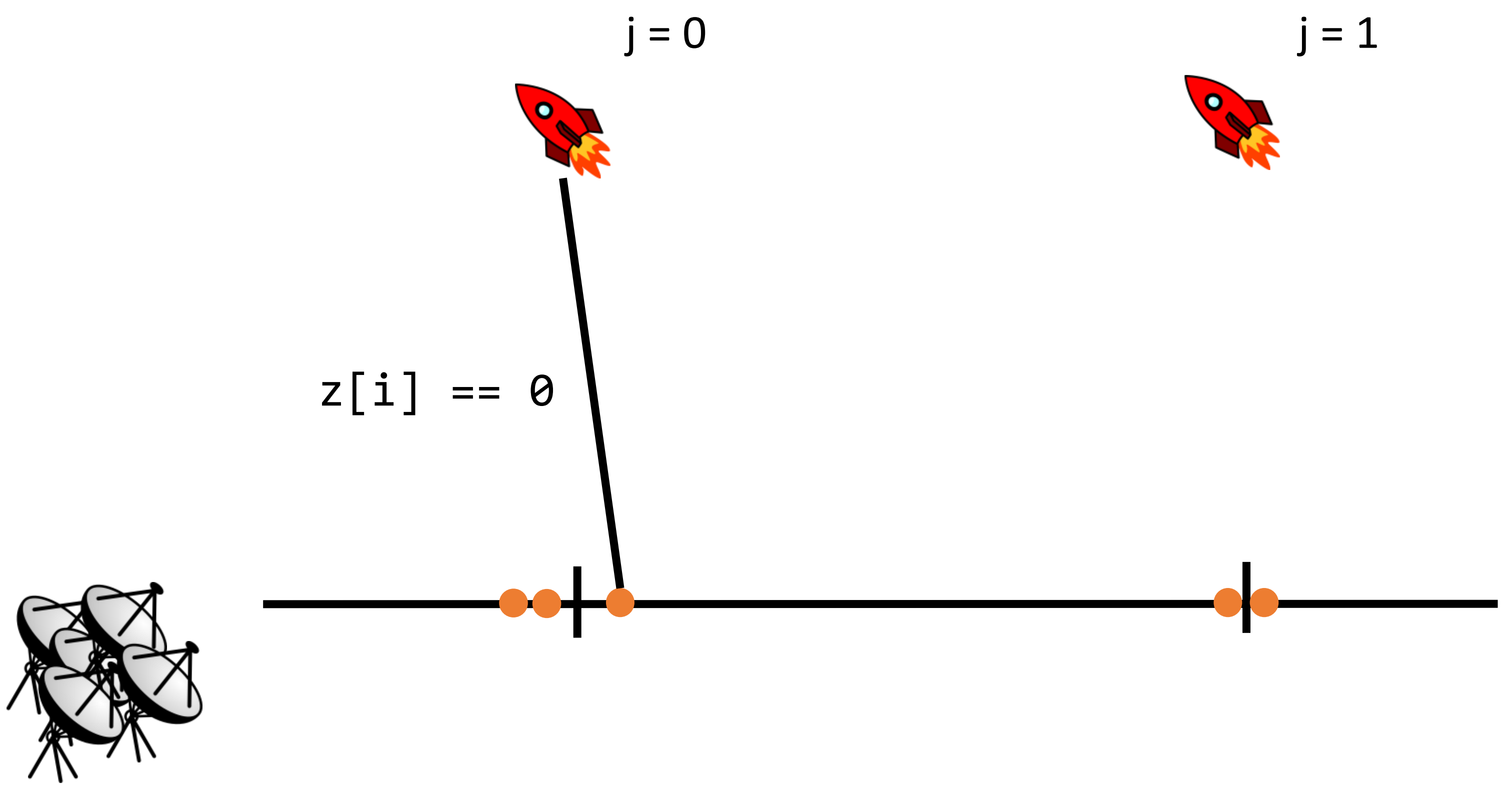
`density(A | B, ϕ)`

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

```
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)
```

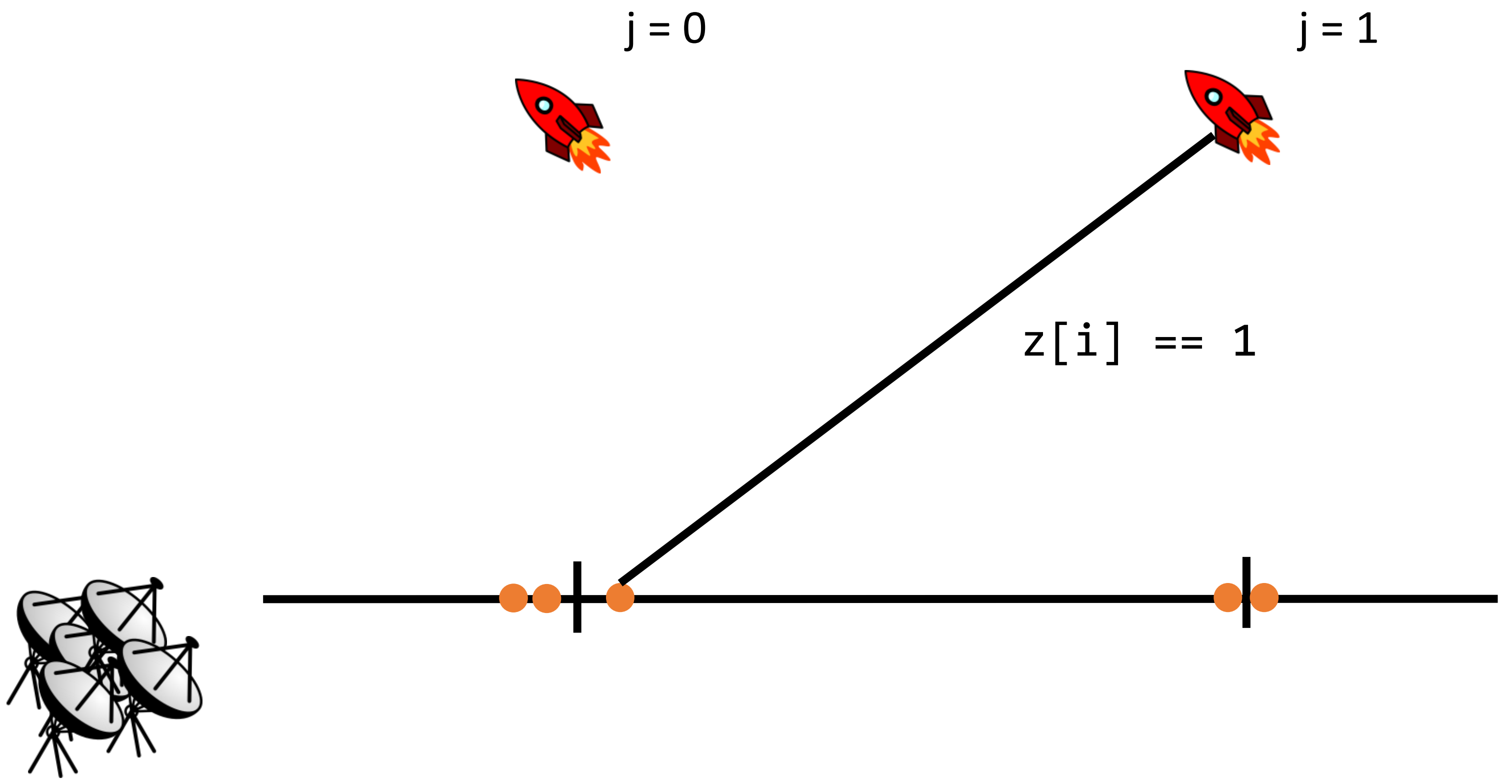
Constraints

```
def obsiDensity(i in Samples, j in Mus)
     : density(obs[i] | 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)
```



Model Specification

```
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 Semantics

```
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

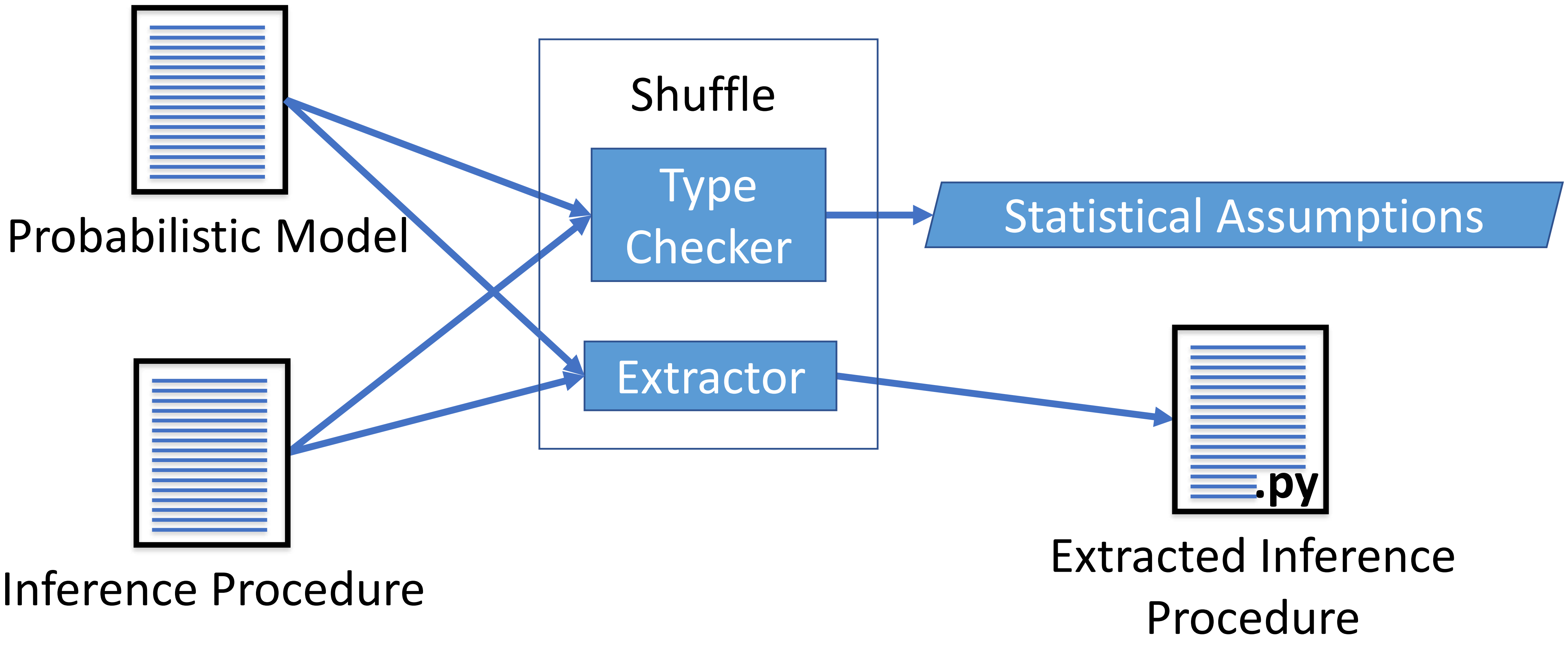
```
def muiPrior(i in Mus) : density(mu[i]) =
     normal(mu[i], 0, 100);
```

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

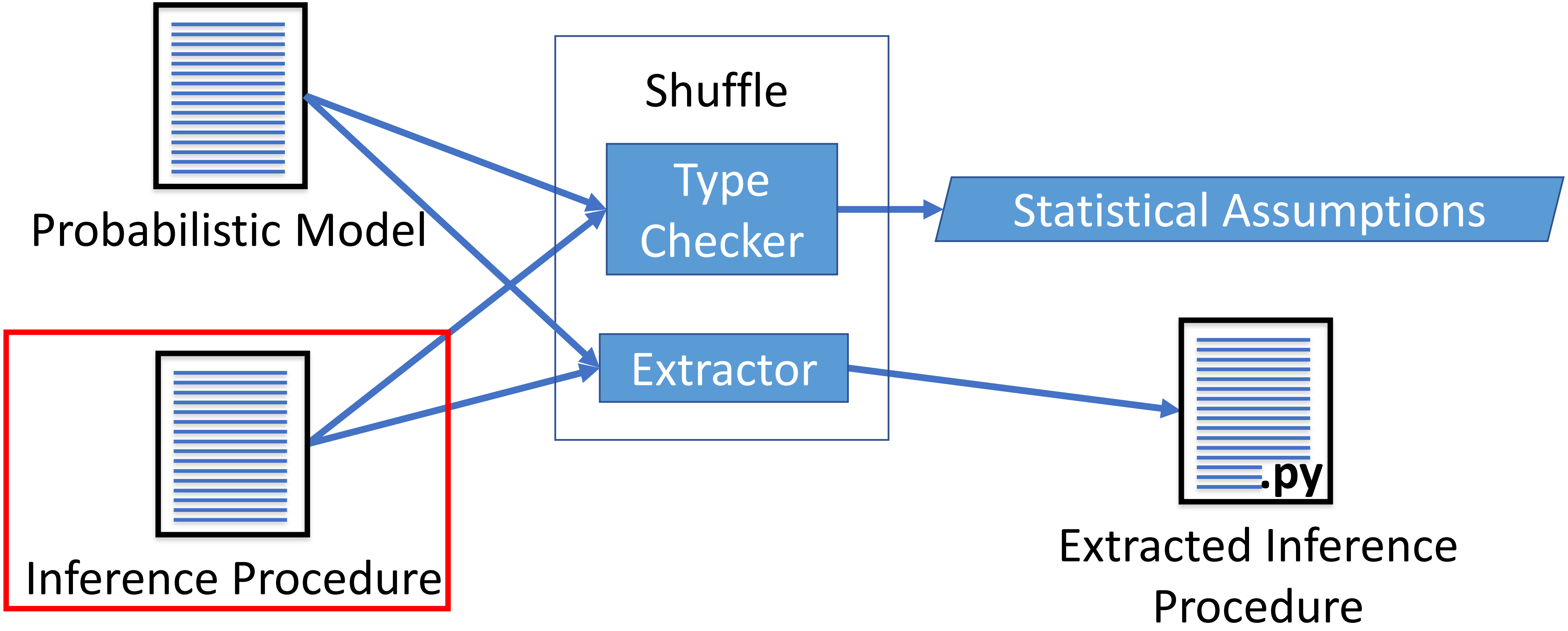
```
def obsiDensity(i in Samples, j in Mus)
     : density(obs[i] | mu[j], z[i] == j)
      = normal(obs[i], mu[j], 1)
```

$$\Pr(\texttt{obs},\texttt{mu},\texttt{z}) = \prod_{j}^{\texttt{Mus}} \texttt{muiPrior}(j) \prod_{i}^{\texttt{Samples}} \texttt{ziPrior}(i) \cdot \begin{cases} \texttt{obsiDensity}(i,j), & \texttt{z[i] == j} \\ 1, & \texttt{else} \end{cases}$$

Shuffle


Probabilistic Model
Inference Procedure
Shuffle
Type
Checker
Extractor
Statistical Assumptions
.py
Extracted Inference
Procedure

Shuffle

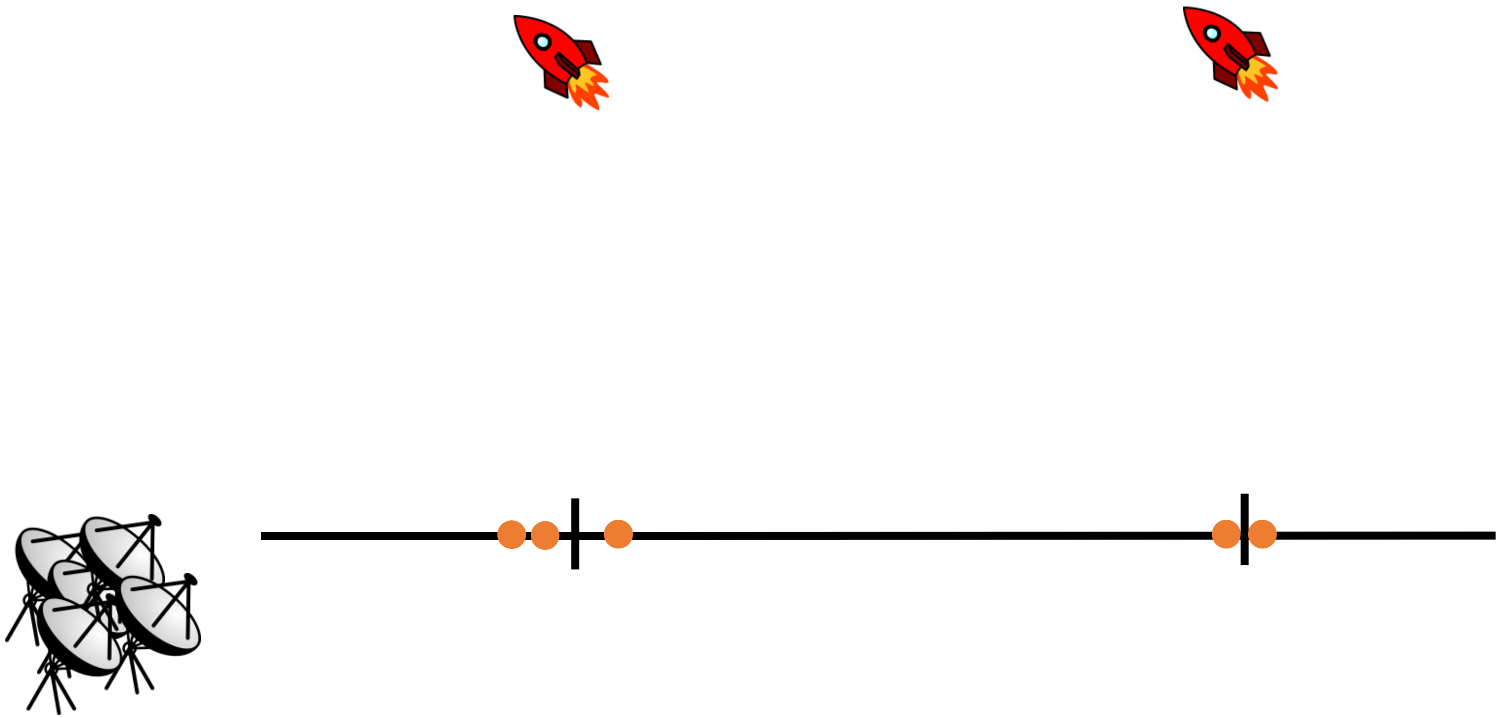

Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Inference Procedures

Goal Distribution:

`density(mu|obs)`

`density(z|obs)`



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;
```

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;
```

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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 likelihood of data given an assignment

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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 joint probability of data and assignments

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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 joint probability of data and assignments

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

Multiplication Operator

```
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

$$\frac{\Gamma \vdash \texttt{obsLikelihood} : \texttt{density(obs | z)} \qquad \Gamma \vdash \texttt{zPrior} : \texttt{density(z)}}{\Gamma \vdash \texttt{obsLikelihood * zPrior} : \texttt{density(obs, z)}}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{density(A | B)} \qquad \Gamma \vdash \texttt{e2} : \texttt{density(B)}}{\Gamma \vdash \texttt{e1 * e2} : \texttt{density(A, B)}}$$

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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 $\mathrm{Mus}^{\mathtt{Samples}}$

Typing

$$\frac{\Gamma \vdash \mathtt{obsZJoint} : \mathtt{density\ (obs,\ z)}}{\Gamma \vdash \mathtt{int\ obsZJoint\ z} : \mathtt{density(obs)}} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{density(A,\ B)}}{\Gamma \vdash \mathtt{int\ e\ B} : \mathtt{density(A)}}$$

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

Division (Bayes' Rule)

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

Python

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

Typing

$$\frac{\Gamma \vdash \texttt{obsZJoint} : \texttt{density(obs, z)} \qquad \Gamma \vdash \texttt{obsMarginal} : \texttt{density(obs)}}{\Gamma \vdash \texttt{zJoint / obsPrior} : \texttt{density(z | obs)}}$$

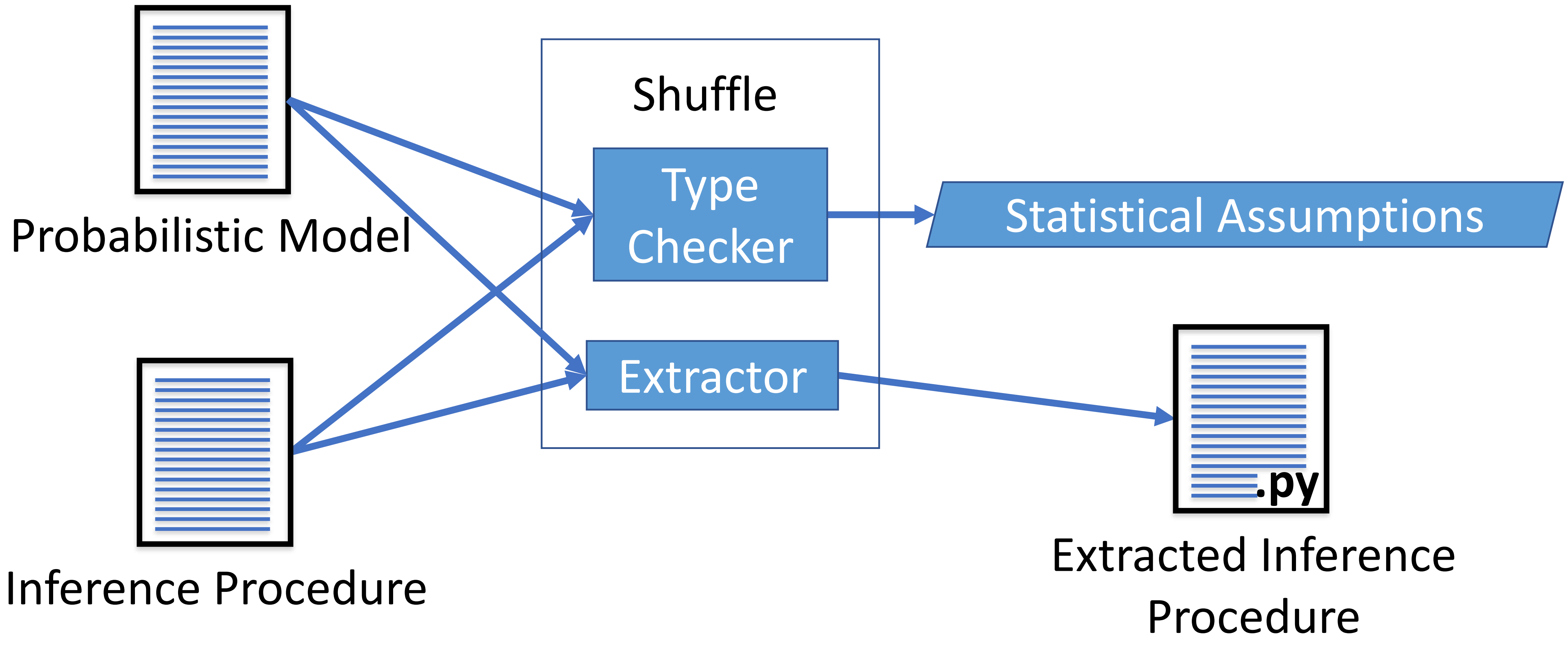
$$\frac{\Gamma \vdash \texttt{e1} : \texttt{density(A, B)} \qquad \Gamma \vdash \texttt{e2} : \texttt{density(B)}}{\Gamma \vdash \texttt{e1 / e2} : \texttt{density(A | B)}}$$

Inference Pro

- Arithmetic operat
- Integrals
- Definition
- Invocation
- Independence dent
- Primitive recursion
- Conditionals } else { d2 }

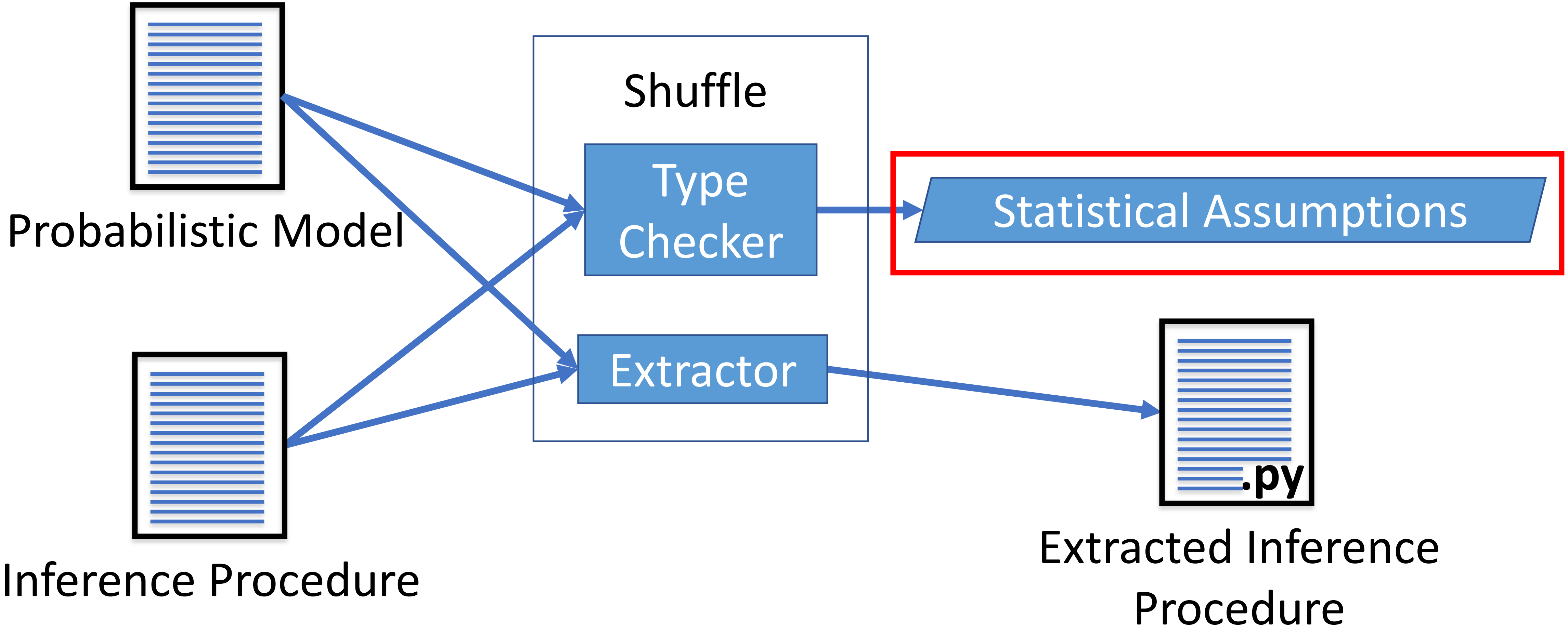
```
def independent obsDens1 (i in dataPoints, j in Mus) :
    density(obs[i] | mu[j], z; z[i] == j) =
    obsDens(i,j) in

def obsDens2 (i in dataPoints, j in Mus) :
    density(obs[i] | mu[j] , z ; z[i] == j) =
    obsDens1(i,j) in

def independent obsDens3 (i in dataPoints, j in Mus) :
    density(
        obs[i] |
        obs{i0 in dataPoints: i0 < i && z[i0] == j}, mu[j], z ;
        z[i] ==j
    ) =
    obsDens2(i,j) in

def independent muDens1(j in Mus) : density(mu[j] | z, none; true)
    = muPrior(j) in

def muDens2(j in Mus) :
    density(mu[j] | z; true) = muDens1(j) in

def rec obsProdHelper(i in dataPoints, j in Mus) :
    density(obs{i0 in dataPoints: i0 <= i && z[i0] == j}
        | mu[j], z; true) =
    if (z[i] == j) {
        obsDens3(i,j) * obsProdHelper(i-1,j)
    } else {
        obsProdHelper(i-1,j)
    } in

def obsProd(j in Mus) :
    density(mu[j], obs{i0 in dataPoints: z[i0] == j} | z; true) =
    obsProdHelper(max(dataPoints),j) * muDens2(j) in

def muPost (j in Mus) :
    density(mu[j] | obs{i0 in dataPoints: z[i0] == j}, z; true)
    = obsProd(j) / int obsProd(j) by mu[j] in

def independent obsMarg(j in Mus) :
    density(obs{i0 in dataPoints: z[i0] == j} | obs{i0 in dataPoints: z[i0] < j},
        z; true) =
    obsProd(j) / muPost(j) in

def rec obsMargAllHelper(j in Mus) :
    density(obs{i0 in dataPoints: z[i0] <= j} | z; true) =
    obsMarg(j) * obsMargAllHelper(j - 1) in

def obsLikelihood() : density(obs | z; true) =
    obsMargAllHelper(max(Mus)) in

def independent ziPriorI(i in dataPoints) :
    density(z[i] | z{i0 in dataPoints: i0 < i}; true) =
    zPrior(i) in

def rec zPriorHelper(i in dataPoints) :
    density(z{i0 in datAPoints: i0 <= i} | none; true) =
    ziPriorI(i) * zPriorHelper(i - 1) in

def zPriorAll() : density(z | none; true) =
    zPriorHelper(max(dataPoints)) in

def zPost() : density(z | obs; true) =
    obsLikelihood() * zPriorAll() / int (obsLikelihood() * zPriorAll()) by z in

def export zPost1() : density(z | obs; true) =
    zPost()
```

Shuffle



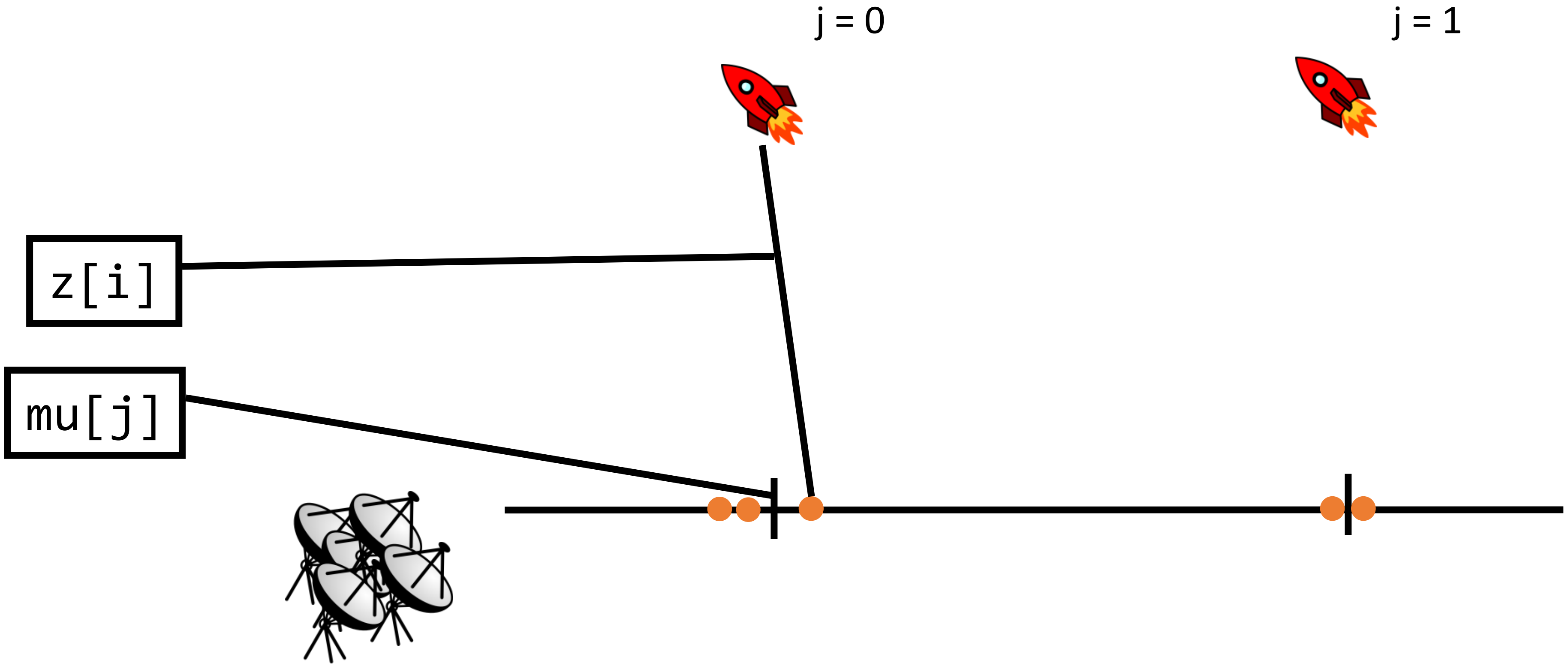
Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Shuffle



Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Independence



z[i] _||_ mu[j]

Assumptions

- Independence

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

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

Assumptions

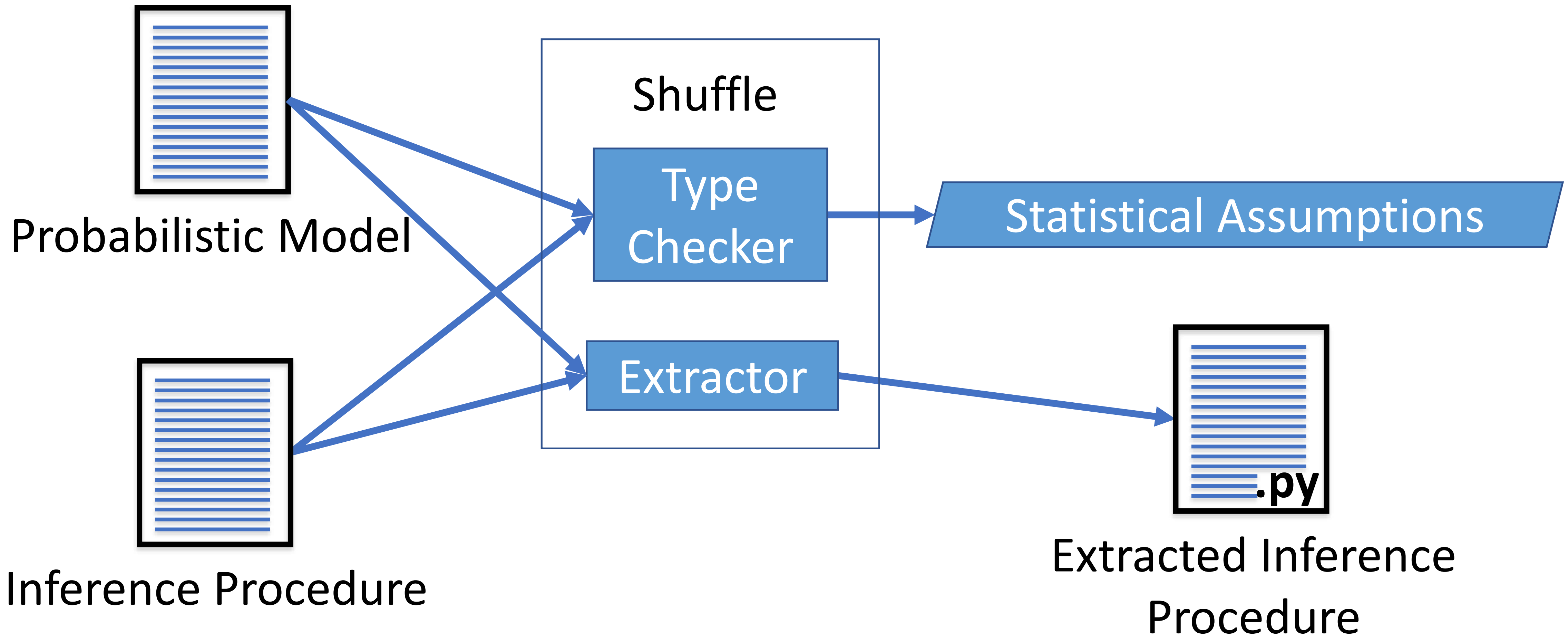
- Independence

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

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

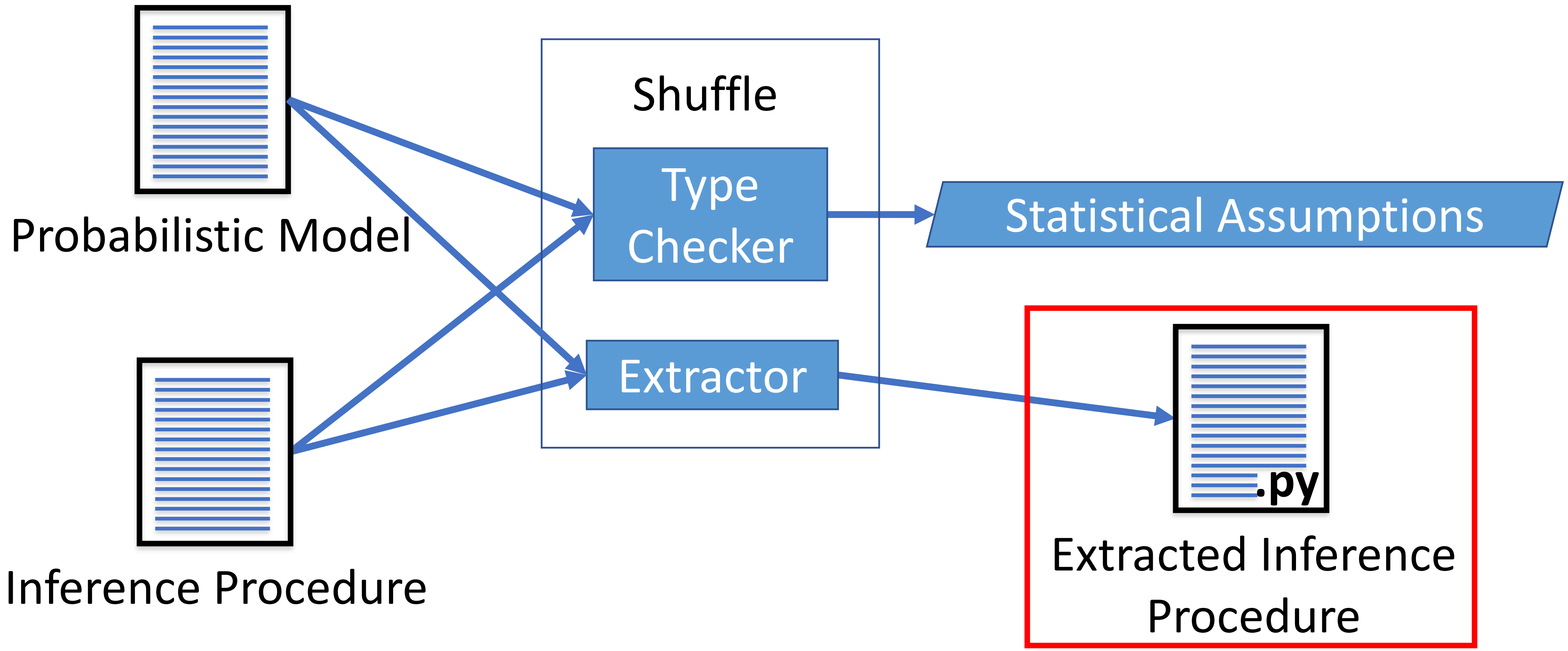
- Saturation

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

Shuffle


Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Shuffle



Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Extraction

```
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:

Prior

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

Likelihood

- If normal, $\Pr(\mathrm{mu}[j] \mid obs[i]) = \dfrac{\frac{\mu_0}{\sigma_0^2} + \frac{obs[i]}{\sigma^2}}{\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}}$

Pattern Matching

```
(normal(obs[i],mu[j],1) * normal(mu[j],0,10)) /
int normal(obs[i],mu[j],1) * normal(mu[j],0,10) by mu[j]
```

Likelihood

Prior

```
normal(mu[j],(obs[i]/1)/(1/10 + 1/1), 1/(1/10 + 1/1))
```

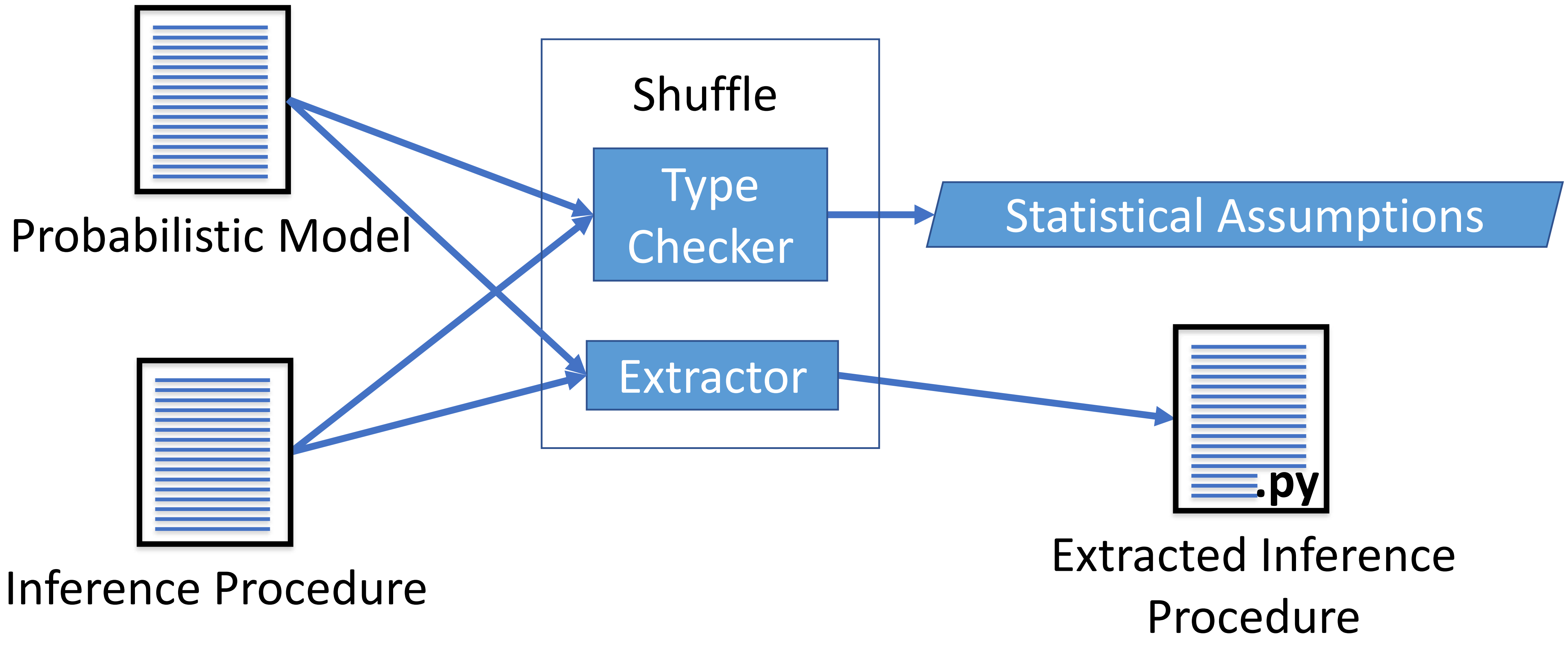
Automatic Incrementalization

```
accs = []
for i in range(N) :
    acc = 0

    for j in in range(i)
        acc += obs[j]

    accs += [acc]
```

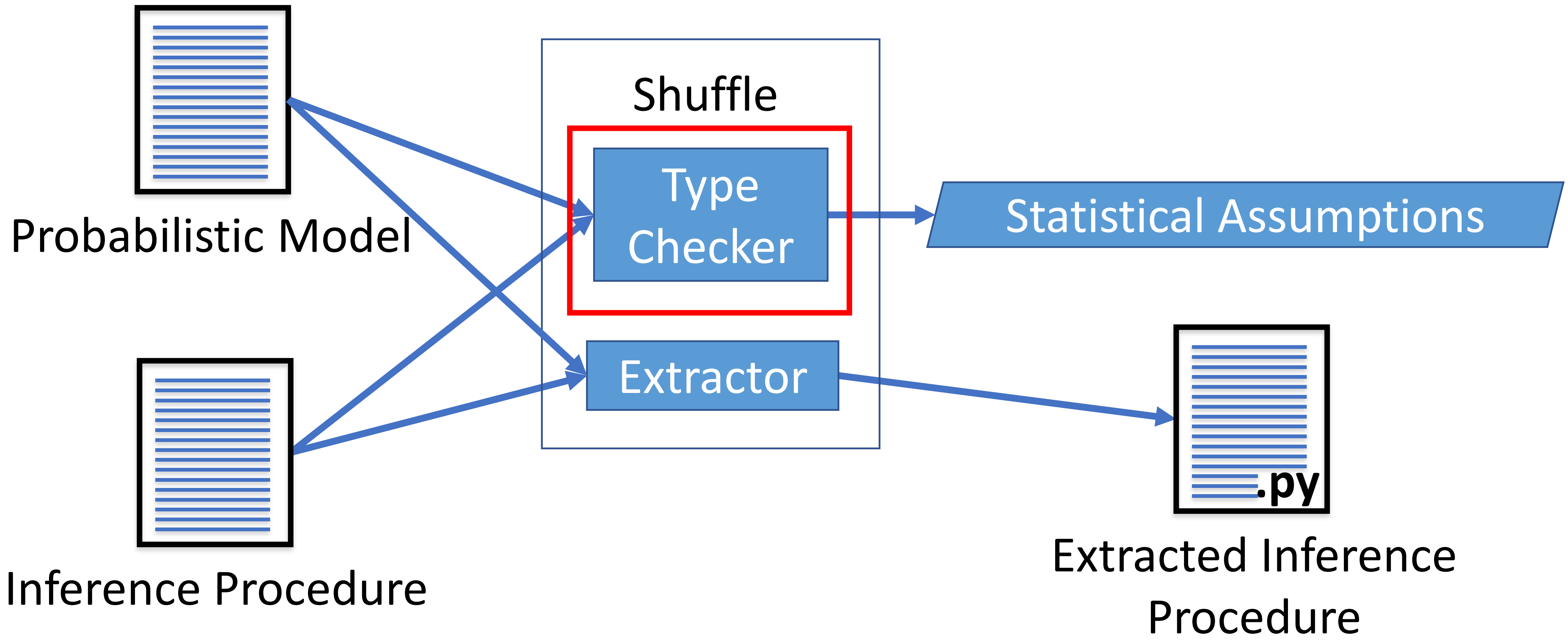
```
accs = []

acc = 0

for i in range(N) :
    acc += obs[i]

    accs += [acc]
```

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

Shuffle


Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Shuffle


Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Type Checking

- Preservation:

$$\Gamma \vdash d\colon \texttt{density}(A \mid B) \Rightarrow \forall \sigma.\, d(\sigma) = \Pr(A \mid B)$$

- Progress:
  - Well-typed terms evaluate successfully

Type Checking

`density(A | B, ϕ)`

- Must verify that $A \cap B = \emptyset$ and `FRV(ϕ)` $\subseteq \mathrm{B}$

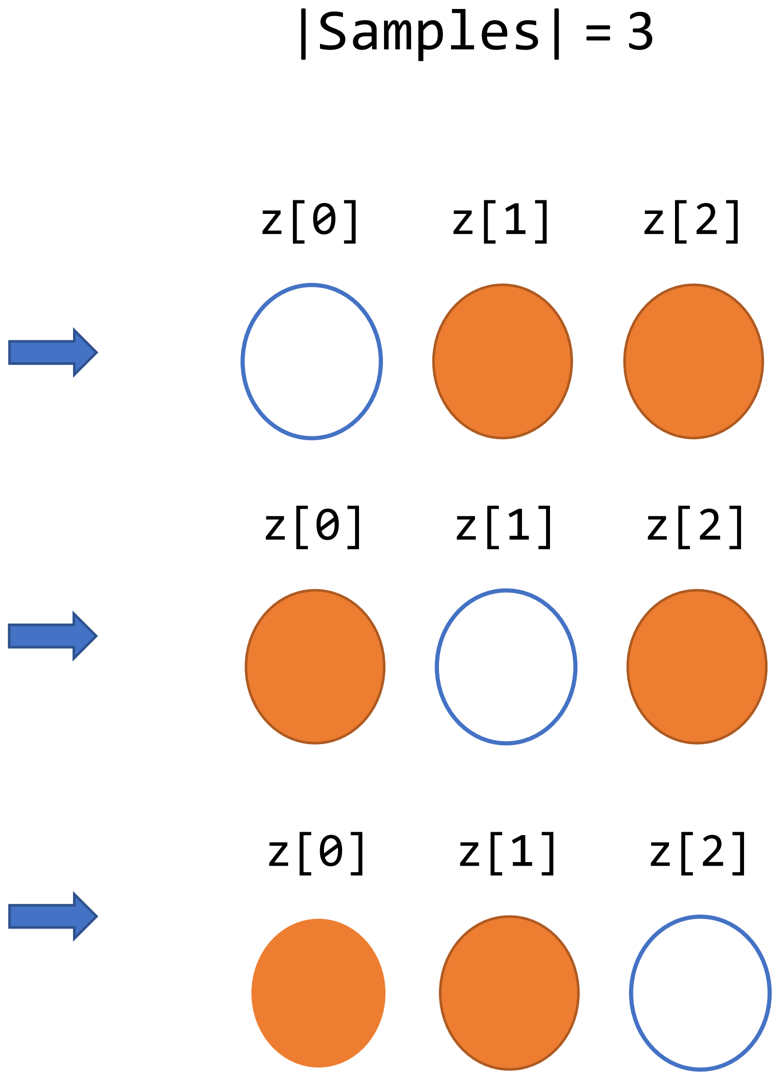
`density(A | B, ϕ)` → `density(A' | B', ϕ')`

- 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] | z{k : k != i}, obs)

density(z[0] | z{k : k != 0}, obs)

density(z[1] | z{k : k != 1}, obs)

density(z[2] | z{k : k != 2}, obs)
```



Benchmarks

Benchmark Name	Shuffle Code (Model/Inference LoC)	Generated Code (LoC)
Burglary	28/30	36
Context-Specific Inference	26/30	24
Dirichlet-Categorical	11/20	20
Healthiness	76/101	118
Hurricane	29/29	12
Naïve Bayes	29/229	318
Normal-Normal	11/20	27
Weather	12/20	24

Benchmarks

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

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

$$\mathrm{P}(\mathrm{z} \mid \mathrm{obs}) = \frac{P(\mathrm{obs} \mid \mathrm{z}) * P(\mathrm{z})}{\int P(\mathrm{obs} \mid z_1) * P(z_1) dz_1} = \frac{P(\mathrm{obs},\ \mathrm{z})}{\int P(\mathrm{obs},\ z_1) dz_1}$$

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 $\text{Mus}^{\texttt{Samples}}$

Approximate Inference

- Alternative: build a *sampler* for P(z | obs)

- For example, given a boolean predicate pred : Mus[Samples] -> bool

```
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] != z[1])
```

Approximate Inference

$$\Gamma \vdash d : \texttt{density}(A \mid B) \Rightarrow \forall \sigma.\, d(\sigma) = \Pr(A \mid B)$$

$$\Gamma \vdash s : \texttt{sampler}(A \mid B) \Rightarrow \forall \sigma. \int_{sr} f\big(s(\sigma, sr)\big) = \int_{A} f(x) * P(x|B)$$

$$\Gamma \vdash k : \texttt{kernel}(A \mid B) \Rightarrow \forall \sigma. \int_{sr_1} \int_{sr_2} f(k(s(\sigma, sr_1), sr_2)) = \int_{A} f(x) * P(x|B)$$

MCMC

$$\Gamma \vdash k : \texttt{estimator}(A \mid B) \Rightarrow \forall \sigma. \int_{sr} \frac{fst\big(e(\sigma, sr)\big) * f\big(snd\big(e(\sigma, sr)\big)\big)}{\int_{sr} fst(e(\sigma, sr))} \int_{A} f(x) * P(x|B)$$

Importance Sampling

Approximate Infe

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

```
def independent obsDens1 (i in dataPoints, j in Mus) :
    density(obs[i] | mu[j], z; z[i] == j) =
    obsDens(i,j) in

def obsDens2 (i in dataPoints, j in Mus, k in dataPoints) :
    density(obs[i] | mu[j] , z ; z[i] == j && i != k) =
    obsDens1(i,j) in

def independent obsDens3 (i in dataPoints, j in Mus, k in dataPoints) :
    density(
        obs[i] |
        obs{i0 in dataPoints: i0 < i && (z[i0] == j && i0 != k)}, mu[j], z ;
        z[i] ==j && i != k
    ) =
    obsDens2(i,j,k) in

def independent muDens1(j in Mus) : density(mu[j] | z, none; true)
    = muPrior(j) in

def muDens2(j in Mus, k in dataPoints) :
    density(mu[j] | z; true) = muDens1(j) in

def rec obsProdHelper(i in dataPoints, j in Mus, k in dataPoints) :
    density(obs{i0 in dataPoints: i0 <= i && z[i0] == j && i0 != k}
        | mu[j], z; true) =
    if (z[i] == j && i != k) {
        obsDens3(i,j,k) * obsProdHelper(i-1,j,k)
    } else {
        obsProdHelper(i-1,j,k)
    } in

def obsProd(j in Mus, k in dataPoints) :
    density(mu[j], obs{i0 in dataPoints: z[i0] == j && i0 != k} | z; true) =
    obsProdHelper(max(dataPoints),j,k) * muDens2(j,k) in

def muPost (j in Mus, k in dataPoints) :
    density(mu[j] | obs{i0 in dataPoints: z[i0] == j && i0 != k}, z; z[k] == j)
    = obsProd(j,k) / int obsProd(j,k) by mu[j] in

def independent obsDensNew (j in Mus, k in dataPoints) :
    density(obs[k] | mu[j], z; z[k] == j) =
    obsDens(k,j) in

def independent obsDensNew2 (j in Mus, k in dataPoints) :
    density(
        obs[k] |
        obs{i0 in dataPoints: z[i0] == j && i0 != k}, mu[j] , z ; z[k] == j
    ) = obsDensNew(j,k) in

def muJoint (k in dataPoints, j in Mus) :
    density(
        mu[j], obs[k] |
        obs{i0 in dataPoints: z[i0] == j && i0 != k}, z; z[k] == j
    ) = obsDensNew2(j,k) * muPost(j,k) in

def independent obsPred(k in dataPoints, j in Mus) :
    density(
        obs[k] | obs{i0 in dataPoints: i0 != k}, z; z[k] == j) =
    int muJoint(k,j) by mu[j] in

def obsPred1(k in dataPoints) :
    density(
        obs[k] | z[k],
        z{i0 in dataPoints : i0 != k},
        obs{i0 in dataPoints: i0 != k}; true
    ) =
    obsPred(k, z[k]) in

def independent zDensNew(i in dataPoints) :
    density(z[i] |
        z{i0 in dataPoints: i0 != i}, obs{i0 in dataPoints: i0 != i}; true) =
    zPrior(i) in

def zJoint(k in dataPoints) :
    density(
        z[k], obs[k] |
        z{i0 in dataPoints: i0 != k}, obs{i0 in dataPoints: i0 != k};
        true
    ) = obsPred1(k) * zDensNew(k) in

def zPost(k in dataPoints) :
    density(z[k] | z{i0 in dataPoints: i0 != k}, obs; true) =
    zJoint(k) / int zJoint(k) by z[k] in

def rec zPostHelper(k in dataPoints) :
    kernel(z{i0 in dataPoints: i0 <= k} |
        z{i0 in dataPoints: k < i0}, obs; true) =
    zPostHelper(k-1); lift z[k] := sample zPost(k) in

def export zPostFinal() : kernel(z | obs; true) =
    zPostHelper(max(dataPoints))
```

sample d

ifte

e by d

ependent

{ d1 } else { d2 }

Small Scale Performance

Benchmark	Algorithm	Assumptions		Speedup vs Venture
		Ind.	Reach.	
Gaussian Mixture Model	Collapsed Gibbs Sampler	7	1	6x
Latent Dirichlet Allocation (LDA)	Collapsed Gibbs Sampler	5	1	.01x
Simultaneous Mapping and Location (SLAM)	Rao-Blackwellized Particle Filter	13	0	20x

Automatic Incrementalization

```
accs = []
for i in range(N) :
    acc = 0

    for j in in range(i)
        acc += obs[j]

    accs += [acc]
```

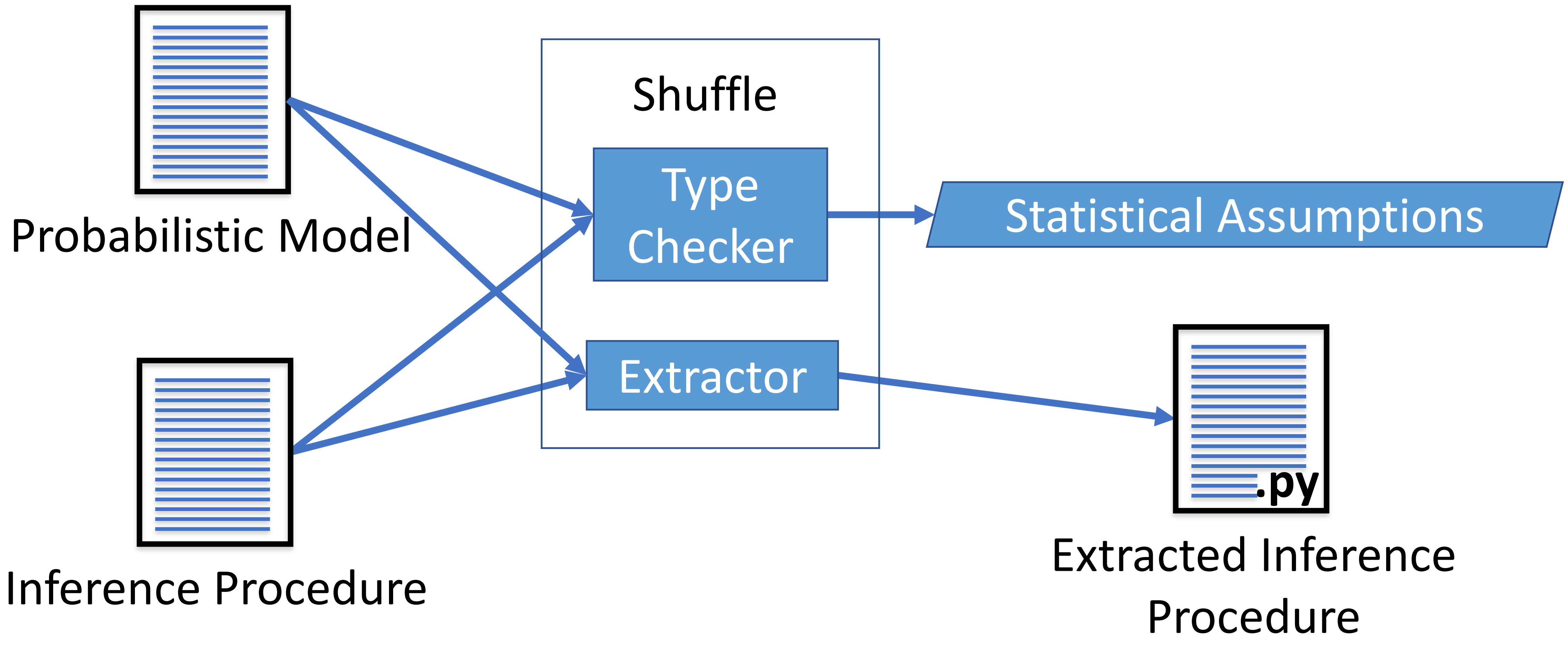
```
accs = []

acc = 0

for i in range(N) :
    acc += obs[i]

    accs += [acc]
```

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

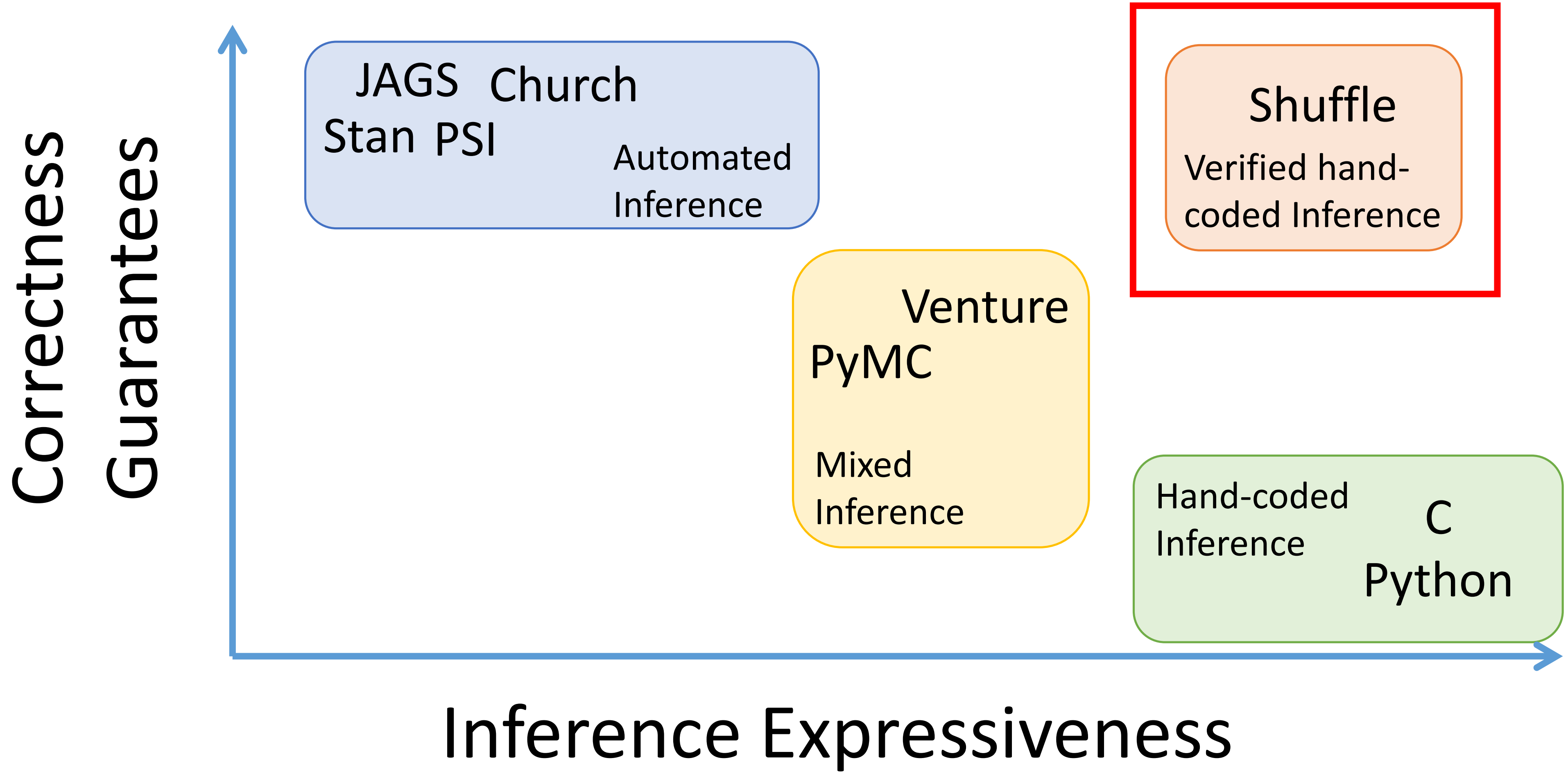
Performance at Scale

Benchmark	Algorithm	Assumptions		Speedup vs Venture
		Ind.	Reach.	
Gaussian Mixture Model	Collapsed Gibbs Sampler	7	1	34x
Latent Dirichlet Allocation (LDA)	Collapsed Gibbs Sampler	5	1	50x
Simultaneous Mapping and Location (SLAM)	Rao-Blackwellized Particle Filter	13	0	1.3x

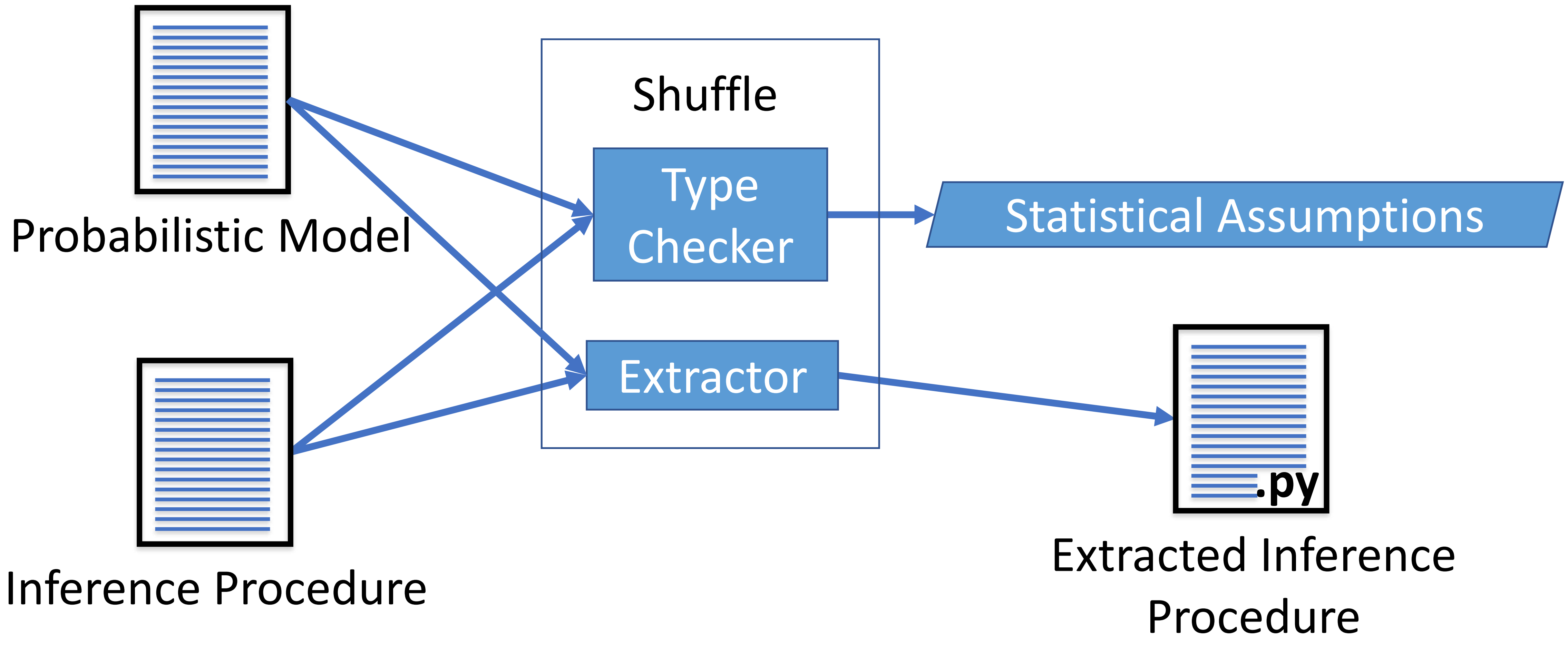
Shuffle


Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

Existing Approaches



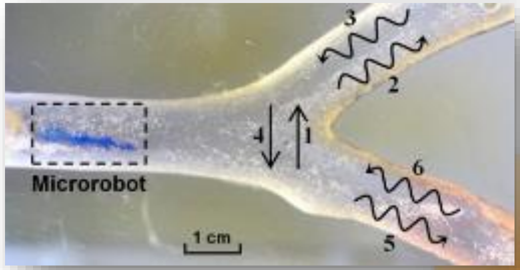
Shuffle

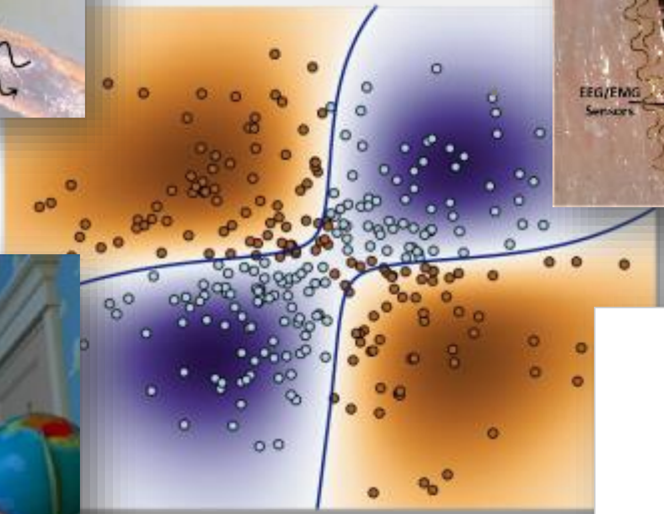

Probabilistic Model
Inference Procedure
Shuffle
Type Checker
Extractor
Statistical Assumptions
.py
Extracted Inference Procedure

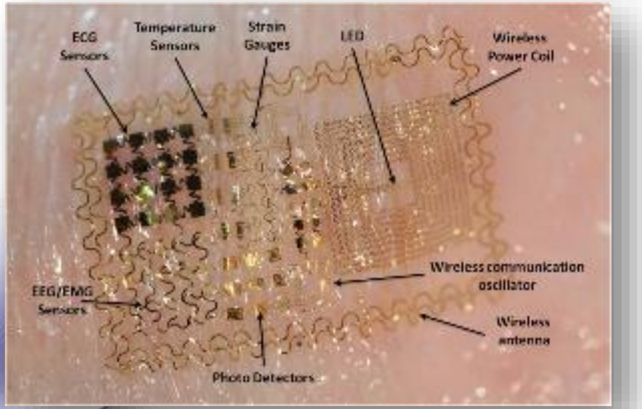
Conclusion

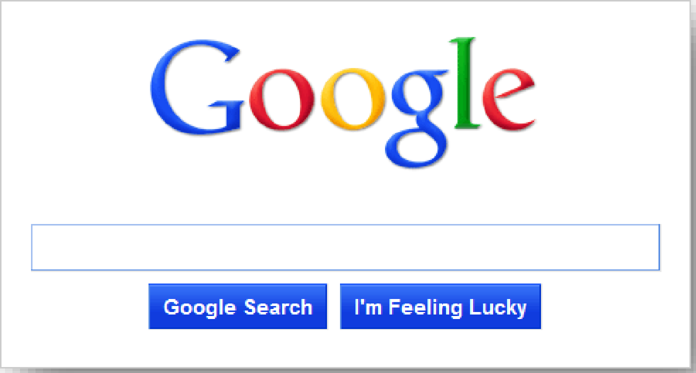
- Many opportunities for resilience
  - Mechanism for dealing with inherently unreliable hardware
  - Mechanism for increased performance (up to 7x)
  - It’s also possible to verify the resulting applications

Takeaway: Methodology for Programming General Uncertain Computations









First-Class Execution Models (SEU)

```
model
{
  bool upset = false;

  operator +(x1, x2) :
    ensures result == (x1 + x2)

  operator +(x1, x2) :
    when !upset,
    ensures upset,
    ensures |result - (x1 + x2)| <= e

  ...
}
```

Model State

Postcondition

Guard

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)

```
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

Relational Program Logic

```
assert (x != null) ∧ x<o> == x<r> ⊨ x<r> != null
```

2. Accuracy – worst-case difference in program result

Relational Program Logic

```
assert_r |res<o> - res<r>| <= .02 * 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