

# Incremental Computation for Efficient Programmable Inference in Probabilistic Programs

**Fabian Zaiser**

MIT



**Jack Czenszak**

Yale



**Martin C. Rinard**

MIT



**Vikash Mansinghka**

MIT



**Alexander K. Lew**

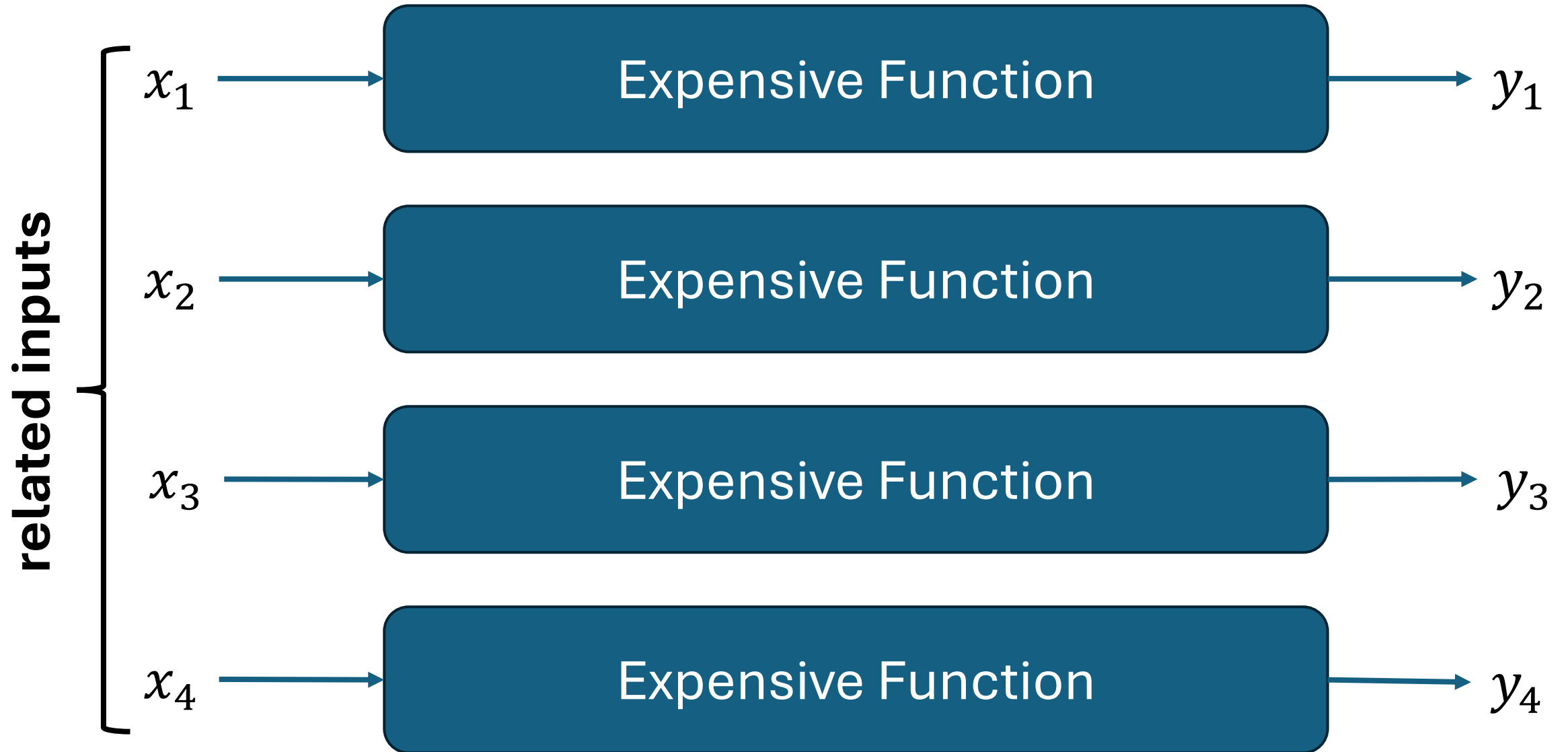
Yale



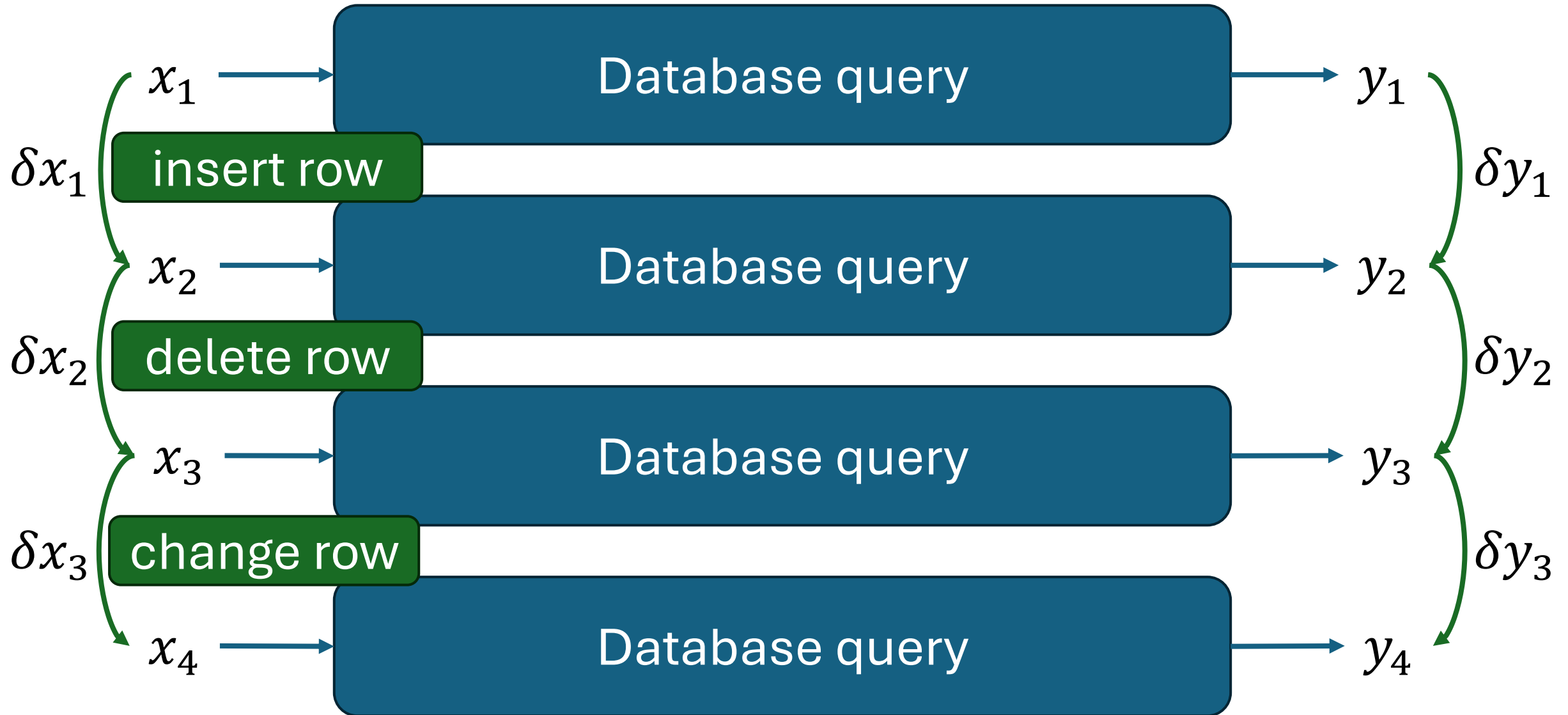
PLDI 2026, Boulder

2026-06-19

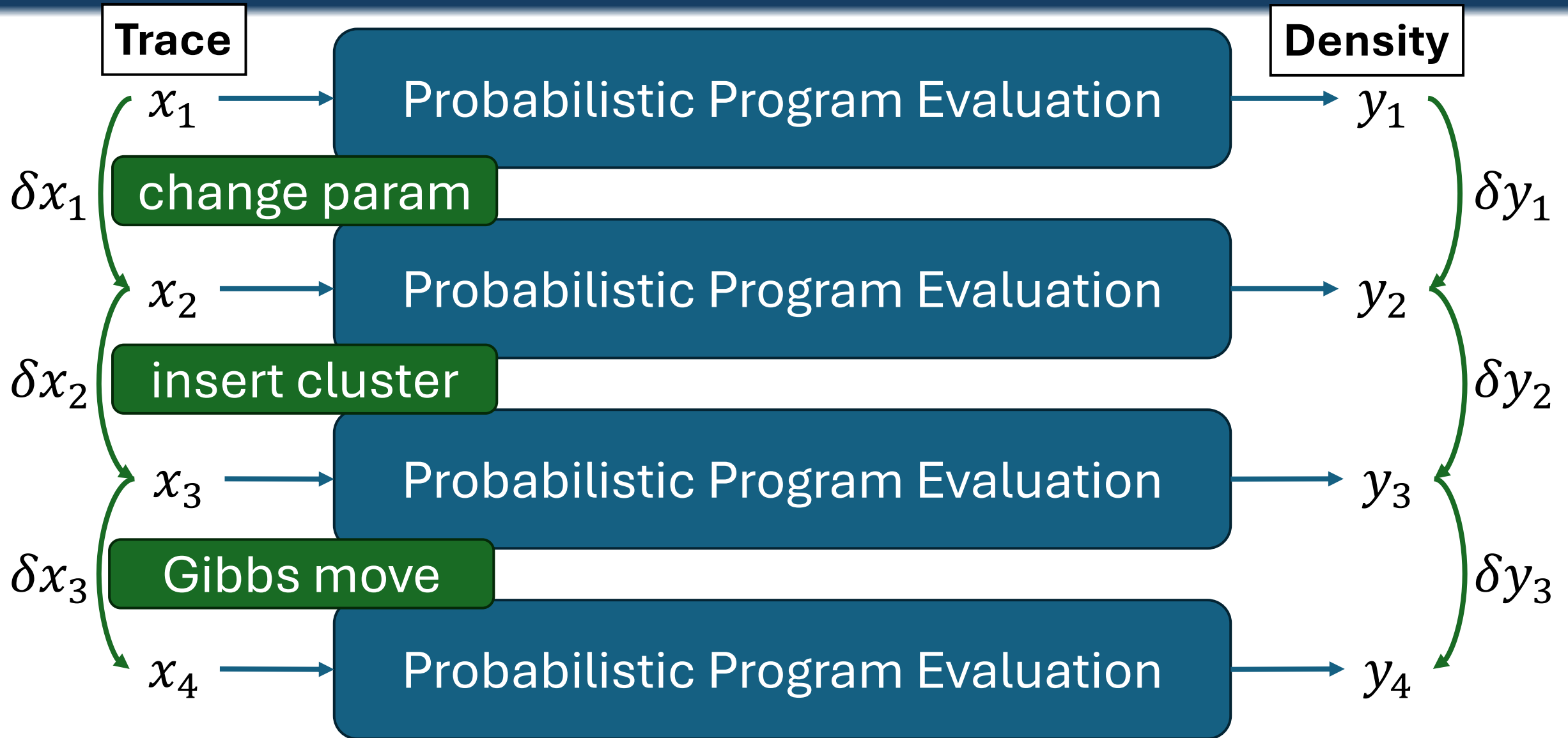
# Incremental Computation



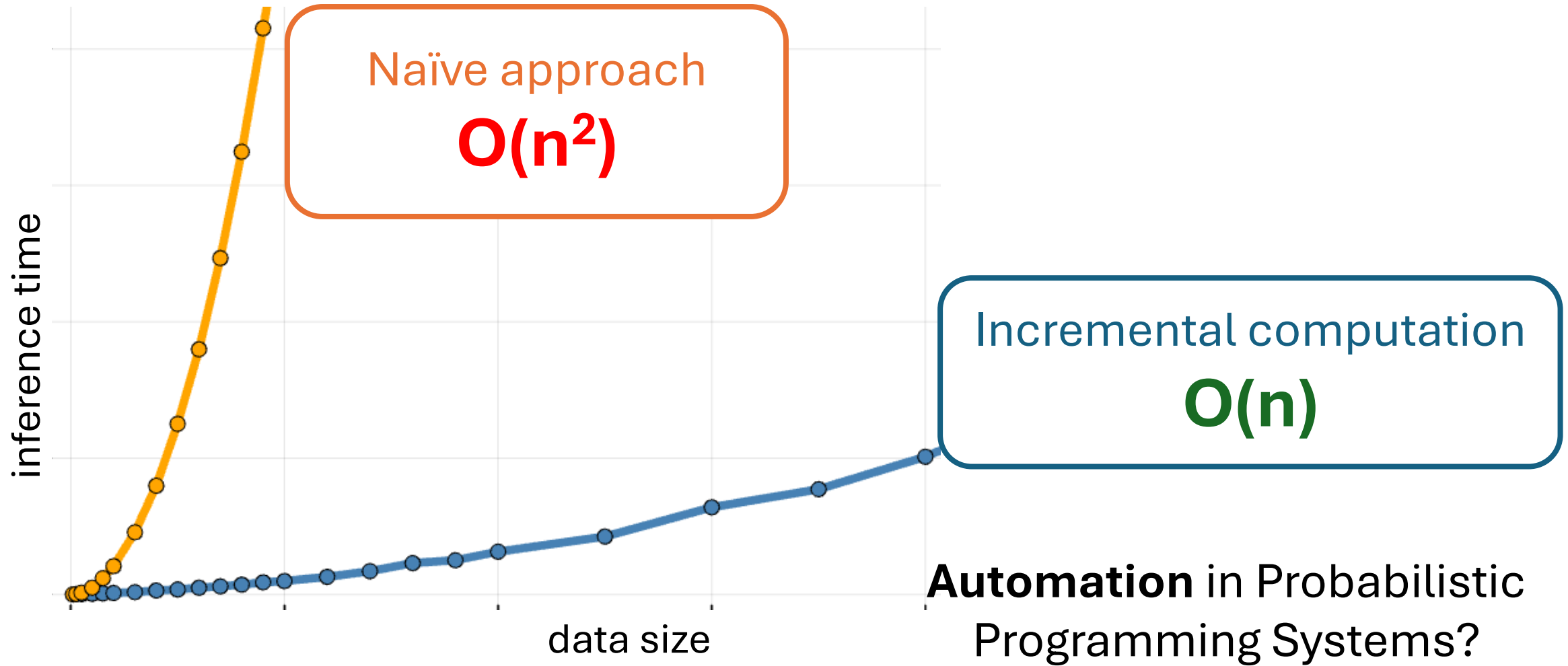
# Example: Databases



# Inference for Probabilistic Programs



# Incremental computation is critical



# Incremental Computation in Existing Systems

## ✗ Complex dependency tracking

Venture [Mansinghka et al. 2014]

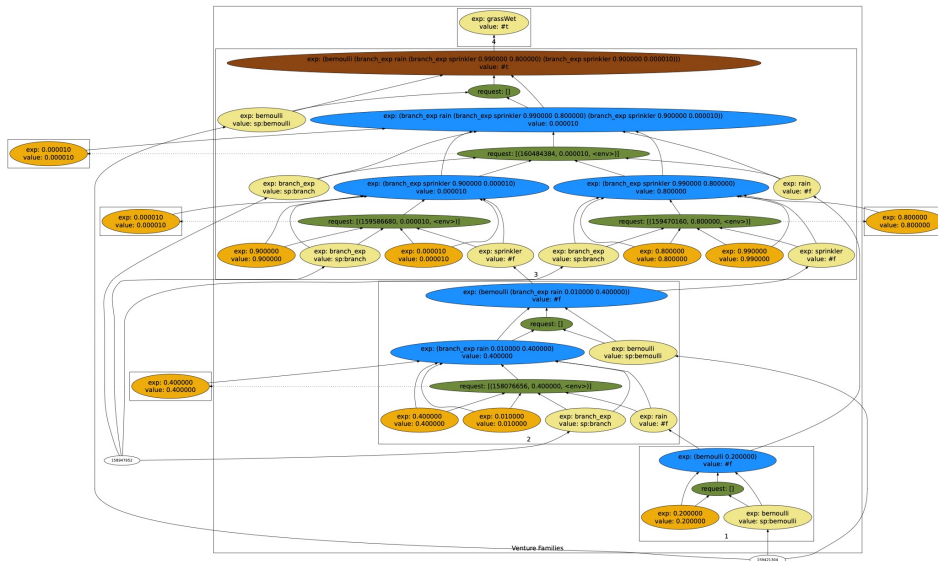
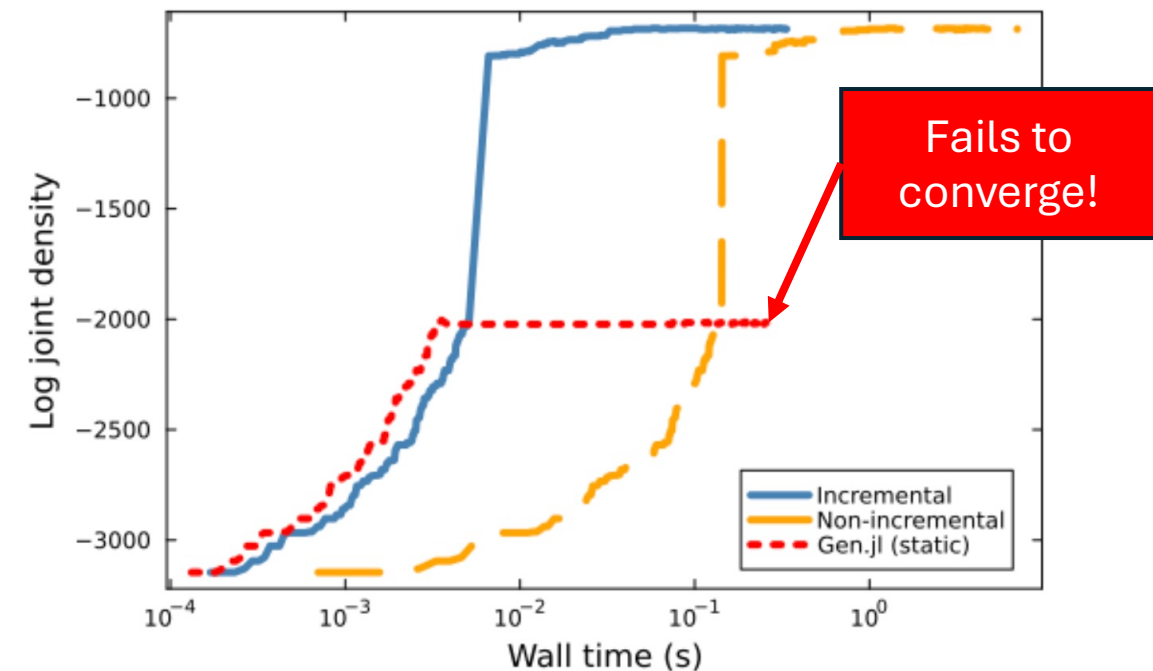


Figure 2: The PET corresponding to a three-node Bayesian network. Each of the three numbered families correspond to a node in the Bayesian network, capturing the execution history needed to simulate the node given its parents.

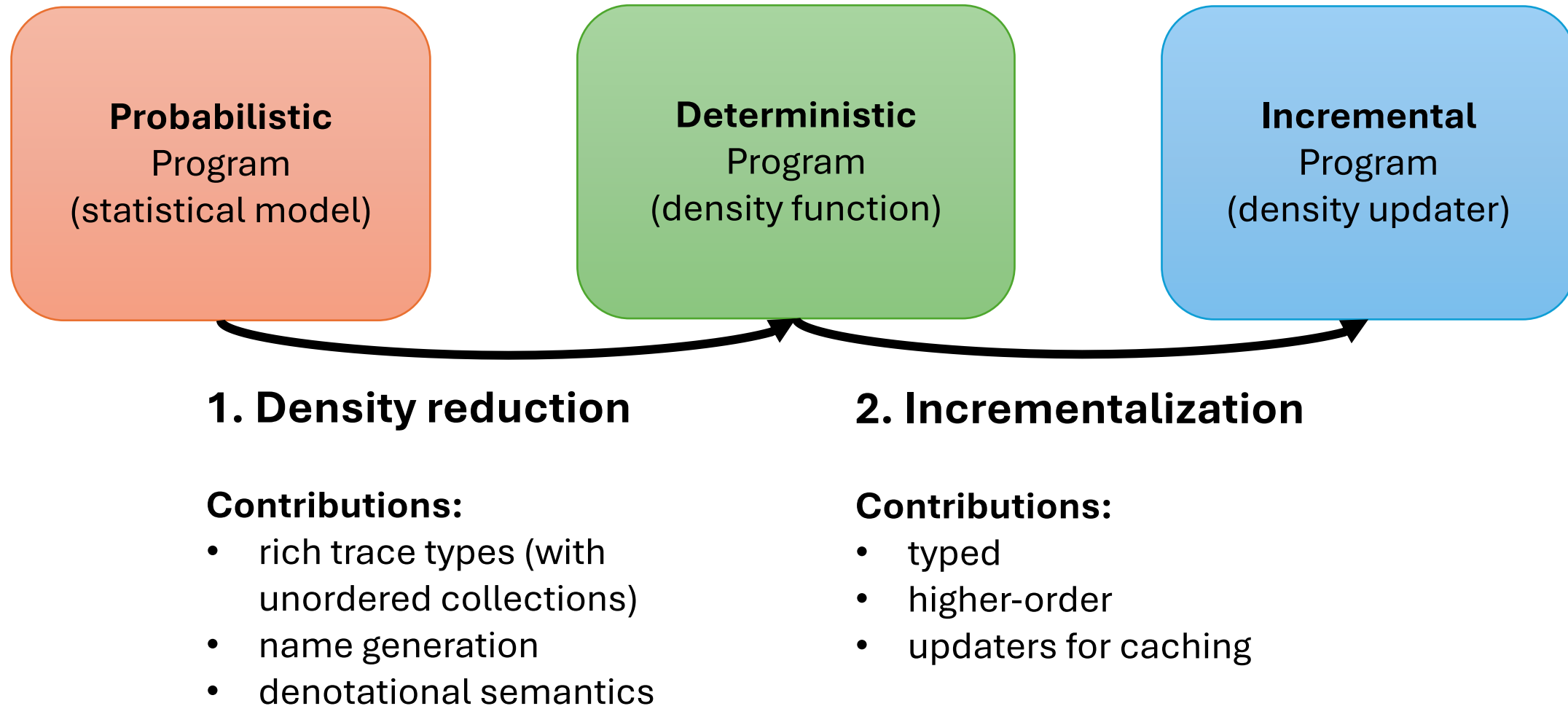
## ✗ Bugs in Gen's incrementalization

Gen.jl [Cusumano-Towner, et al. 2019]



## ✗ Sometimes suboptimal time complexity

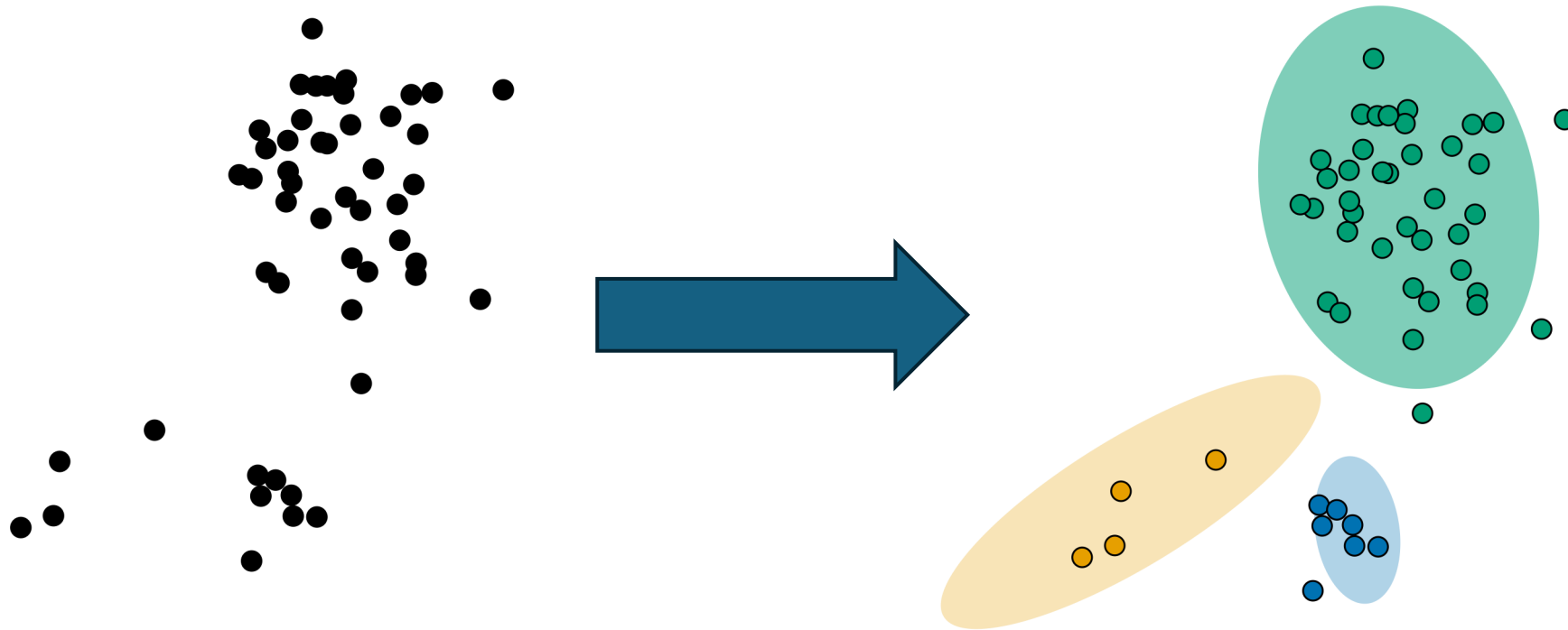
# Reduce to deterministic incrementalization



# Overview

- **Motivating Example**
- Key ideas of our approach
- Theoretical guarantees
- Experimental results
- Summary

# Clustering

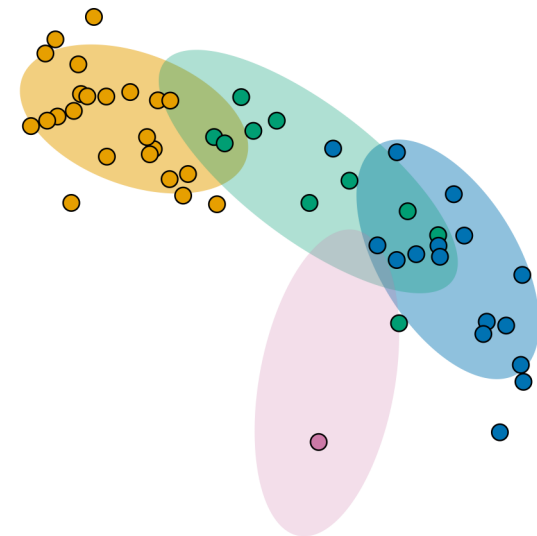


# Probabilistic Programs & Execution Traces

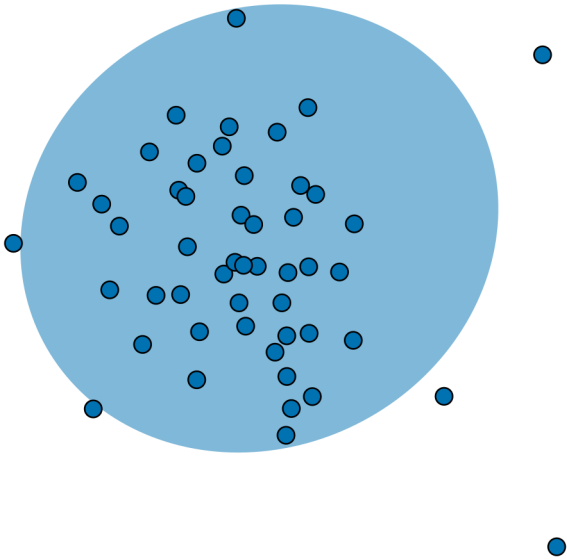
```
def model(n: Nat):  
  clusters ~ geometric(0.5)  
  params ~ for i in 1..clusters:  
    (mean, covariance) ~ paramPrior  
  weights ~ dirichlet(clusters, ...)  
  assignments ~ multinomial(weights, n)  
  data ~ for cluster in assignments:  
    normal(params[cluster])  
  return data
```

## Trace:

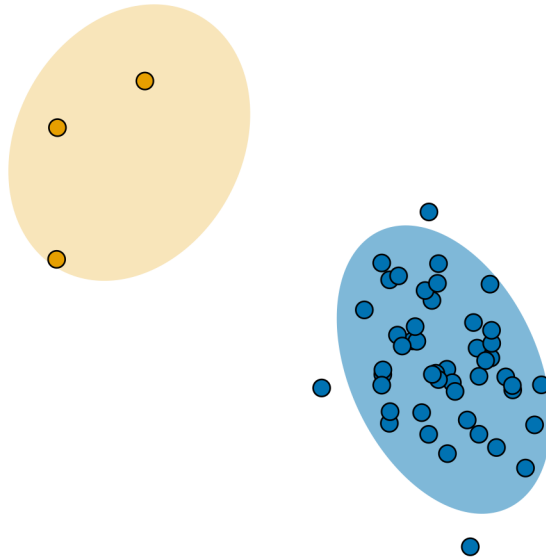
```
{ clusters: 4,  
  params: [  
    { mean: (1.4, 0.6), cov: ... },  
    ... ],  
  weights: [0.34, 0.51, 0.11, 0.04],  
  assignments: [1, 1, 2, 4, 3, ...]  
  data: [(1.4, 0.2), (0.7, -0.1), ...] }
```



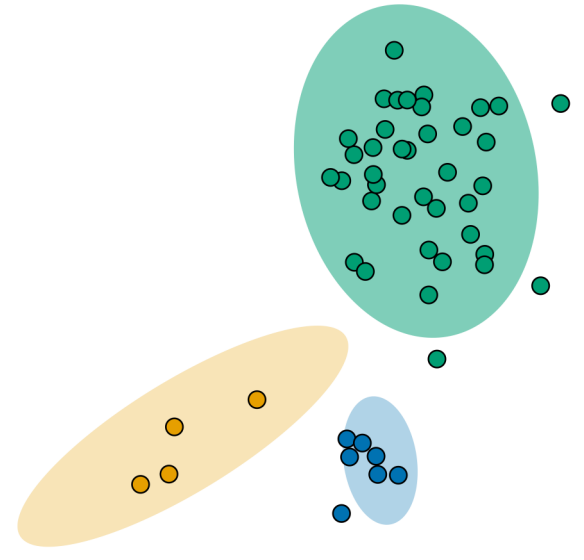
# Other possible execution traces



```
{ clusters: 1,  
  weights: [1],  
  ... }
```



```
{ clusters: 2,  
  weights: [0.89, 0.11],  
  ... }
```



```
{ clusters: 3,  
  weights: [0.15,  
            0.11, 0.74],  
  ... }
```

# Inference

```
def model(n: Nat):  
  clusters ~ geometric(0.5)  
  params ~ for i in 1..clusters:  
    (mean, covariance) ~ paramPrior  
  weights ~ dirichlet(clusters, ...)  
  assignments ~ multinomial(weights, n)  
  data ~ for cluster in assignments:  
    normal(params[cluster])  
  return data
```

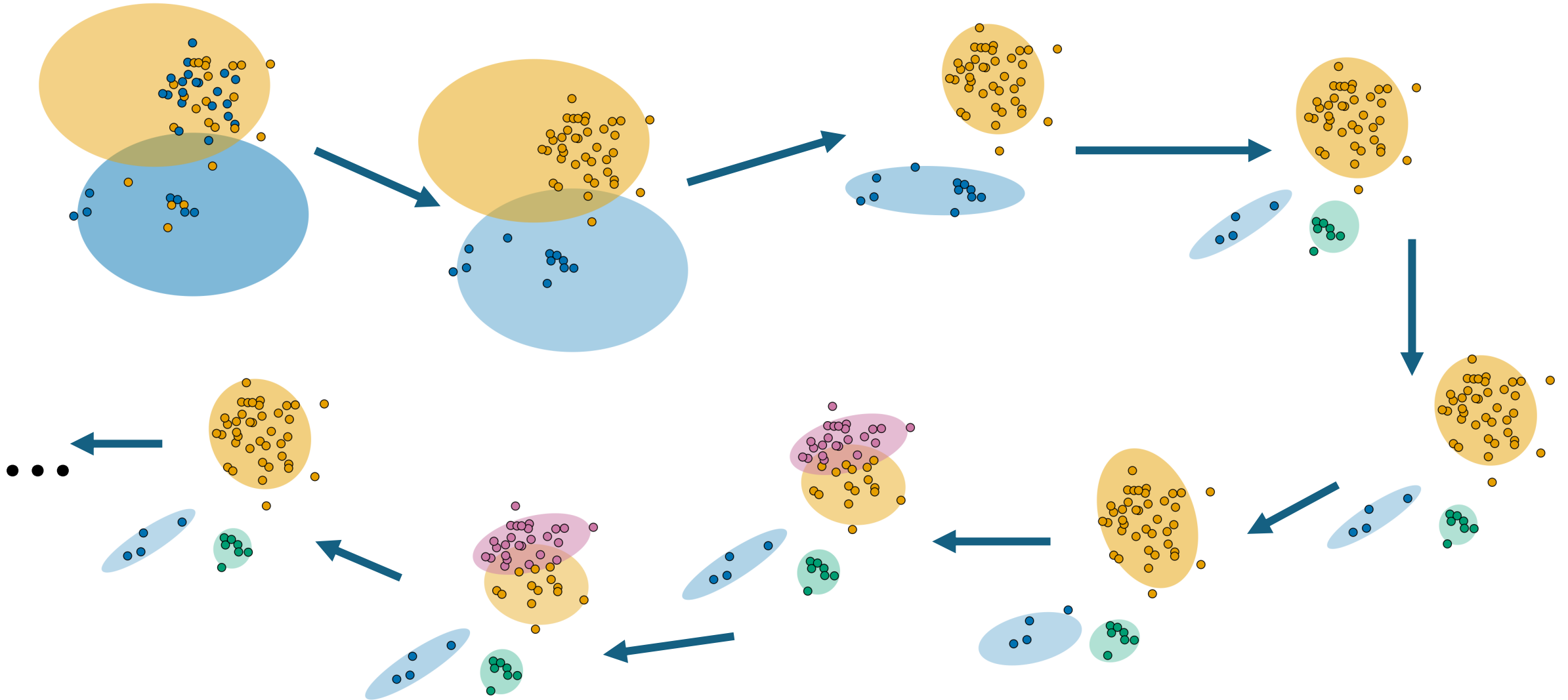
## Trace:

```
{ clusters: ???  
  params: ???,  
  weights: ???,  
  assignments: ???,  
  data: [(1.4, 0.2), (0.7, -0.1), ...] }
```

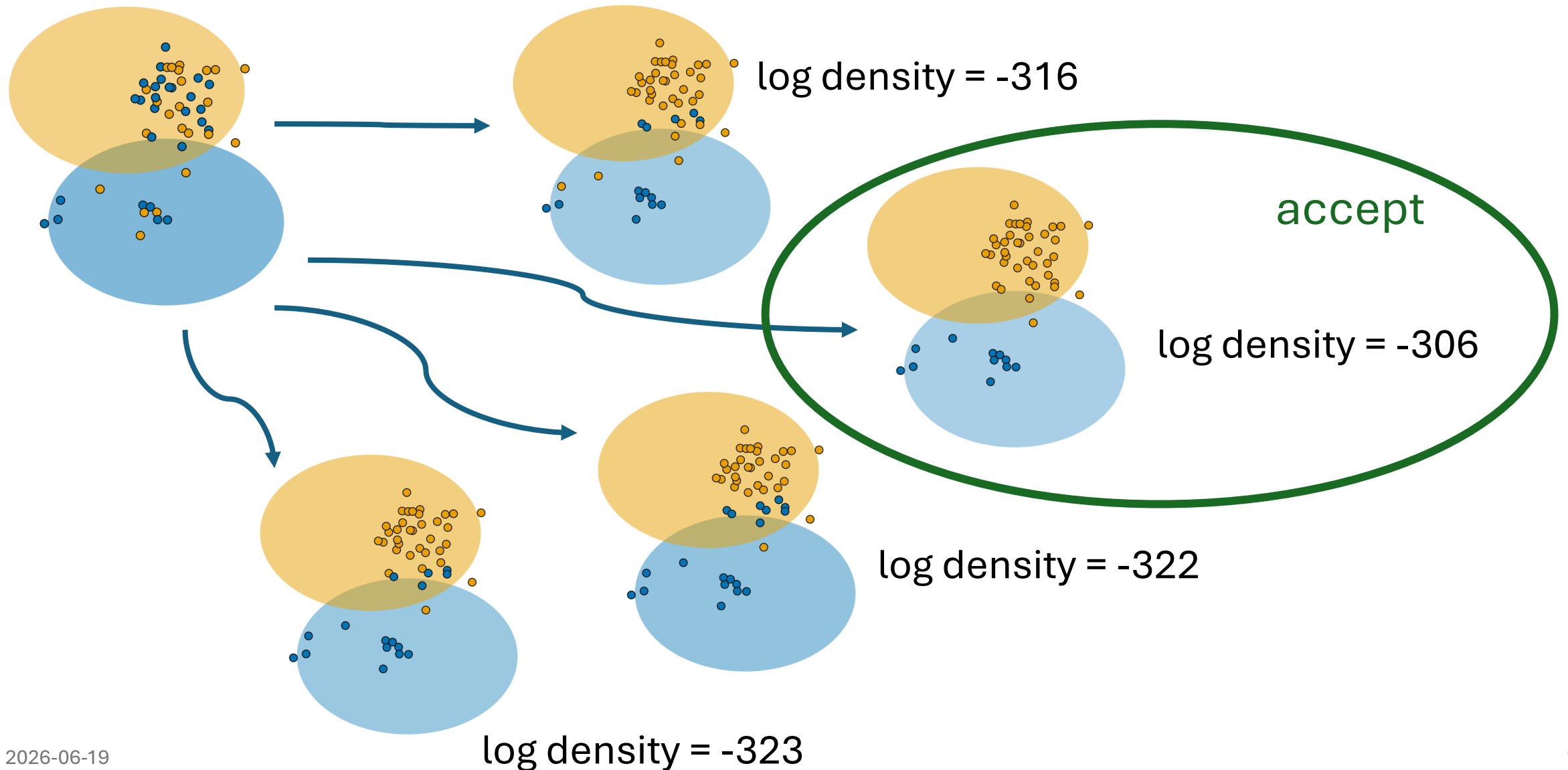


Find traces that explain the given data well!

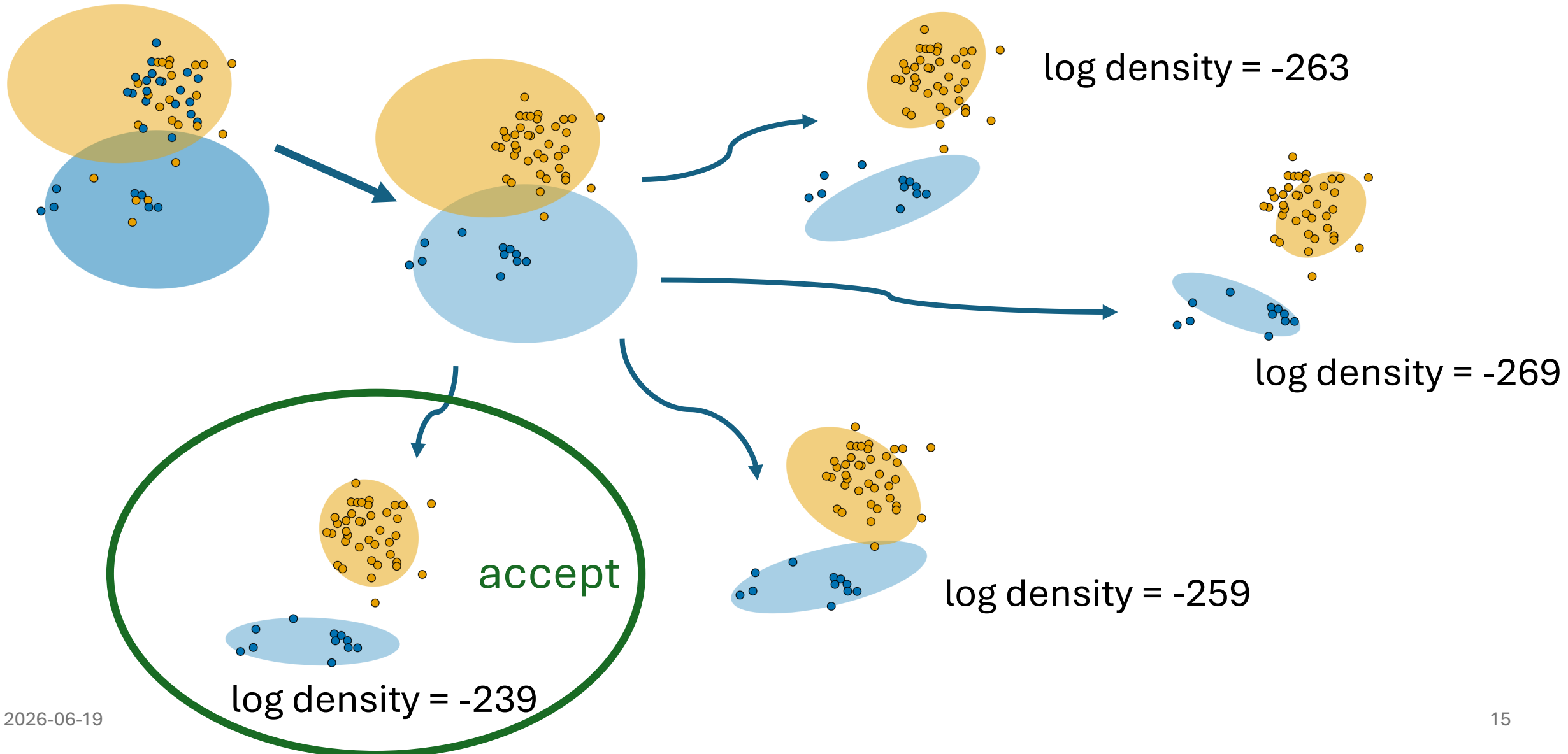
# Markov Chain Monte Carlo (MCMC): searching for good traces



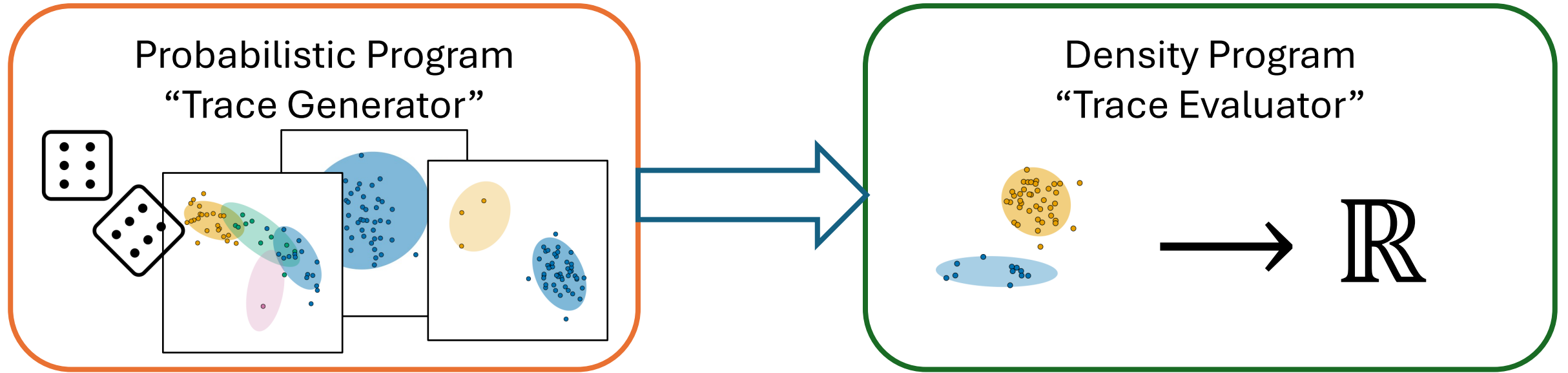
# MCMC: searching for good traces



# MCMC: searching for good traces



# The task: two program transformations



Deterministic Program

$$A \longrightarrow B$$

Incremental Program

$$A \longrightarrow B$$
$$\& \quad \Delta A \longrightarrow \Delta B$$

# Overview

- Motivating example
- **Key ideas of our approach**
- Theoretical guarantees
- Experimental results
- Summary

# Background on the Incremental $\lambda$ -Calculus [Cai et al. 2014]

- **Formal framework** for incremental computation in functional languages
- **Change types  $\Delta\tau$** : for each type  $\tau$ , defined recursively on the type grammar

# Change Types $\Delta\tau$ : Our Choices

| Type $\tau$                                                                     | Change type $\Delta\tau$                                                                                              |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <b>Base types:</b><br>$\text{Bool}, \mathbb{Z}, \mathbb{R}, \text{Name}$        | <b>New value &amp; flag:</b><br>$\{ \text{new}: \tau, \text{same?}: \text{Bool} \}$                                   |
| $\tau_1 \times \tau_2$                                                          | $\Delta\tau_1 \times \Delta\tau_2$                                                                                    |
| <b>Record:</b><br>$\{ \text{field}_1: \tau_1, \text{field}_2: \tau_2, \dots \}$ | <b>Subrecord:</b><br>$\text{Sub}\{ \text{field}_1: \Delta\tau_1, \text{field}_2: \Delta\tau_2 \}$                     |
| $\text{Map Name } \tau$                                                         | $\{ \text{remove}: \text{Set Name}, \text{change}: \text{Map Name } \Delta\tau, \text{add}: \text{Map Name } \tau \}$ |

Changes to **collections** are **finer-grained** so we can **skip loop iterations!**

# Background on the Incremental $\lambda$ -Calculus [Cai et al. 2014]

- **Formal framework** for incremental computation in functional languages
- **Change types  $\Delta\tau$** : for each type  $\tau$ , defined recursively on the type grammar
- Every **primitive** function comes with an **incremental** version
  - Goal 1: **propagate** change information
  - Goal 2: **exploit** change information to compute output incrementally
- **Program transformation**
  - composes incremental primitives
  - propagates changes downstream

# Original Incremental $\lambda$ -Calculus [Cai et al. 2014]

Function:

$f : \text{In} \rightarrow \text{Out}$

Derivatives are **change propagators**

Derivative:

$\delta f : \text{In} \times \Delta \text{In} \rightarrow \Delta \text{Out}$

**✗ Problem:** Only efficient for **self-maintainable** functions

# Incremental $\lambda$ -Calculus with Caching [Giarrusso et al. 2019]

Function:

$f : \text{In} \rightarrow \text{Out}$

Caching Function

$f' : \text{In} \rightarrow \text{Out} \times \text{Cache}_f$

Caching derivative:

$\delta f : \Delta \text{In} \times \text{Cache}_f$   
 $\rightarrow \Delta \text{Out} \times \text{Cache}_f$

**Idea: cache reusable  
intermediate results**

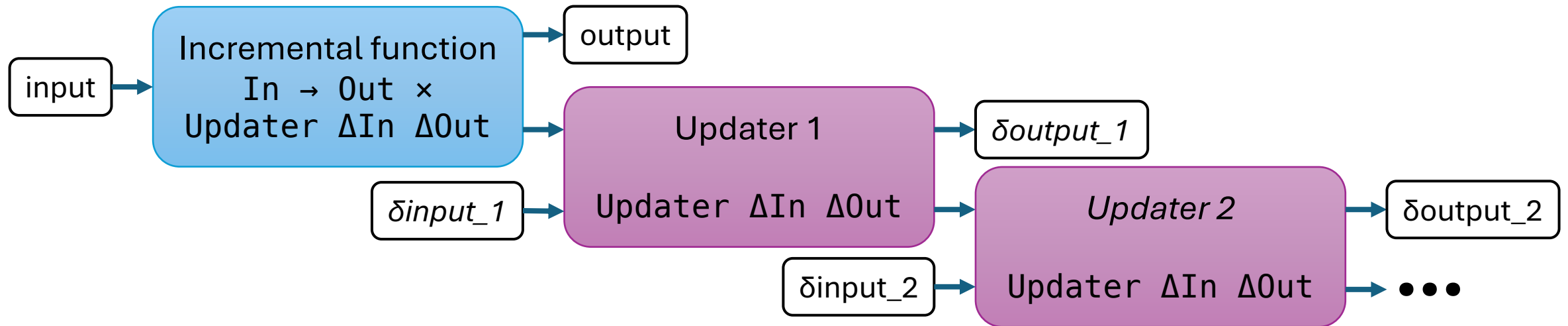
Composing functions tuples their  
cache types.

**✗ Problems:**

- huge cache types everywhere
- plays badly with typing
- correctness proofs are hard

# Our Contribution: Coinductive Updater Types

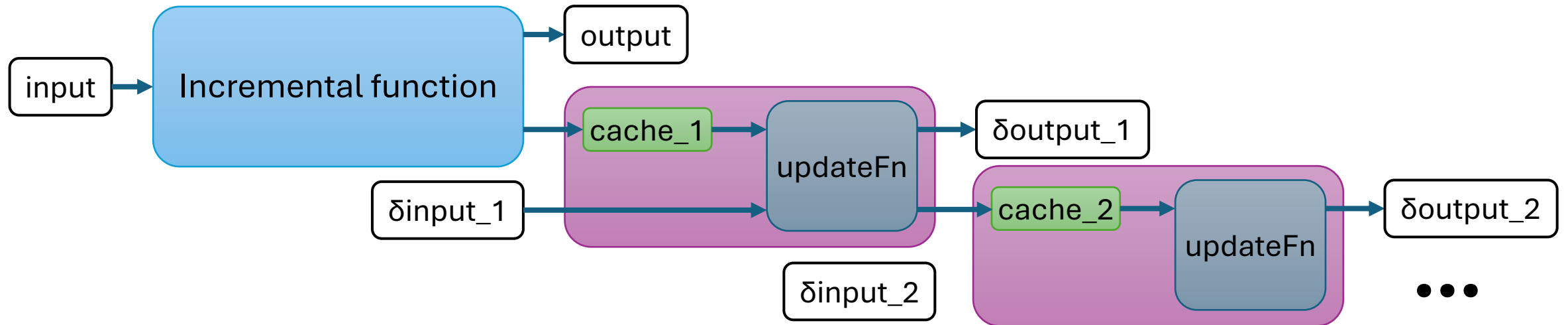
**coinductive Updater**  $\Delta\text{In}$   $\Delta\text{Out}$   $\approx \Delta\text{In} \rightarrow \Delta\text{Out} \times \text{Updater } \Delta\text{In } \Delta\text{Out}$



# Constructing Updaters

**coinductive Updater**  $\Delta\text{In}$   $\Delta\text{Out}$   $\approx \Delta\text{In} \rightarrow \Delta\text{Out} \times \text{Updater } \Delta\text{In } \Delta\text{Out}$

**mkUpdater**(**cache**: **Cache**, **updateFn**:  $\Delta\text{In} \times \text{Cache} \rightarrow \Delta\text{Out} \times \text{Cache}$ ): **Updater**  $\Delta\text{In}$   $\Delta\text{Out}$



# Updater Example

**coinductive Updater**  $\Delta\text{In}$   $\Delta\text{Out}$   $\approx \Delta\text{In} \rightarrow \Delta\text{Out} \times \text{Updater } \Delta\text{In } \Delta\text{Out}$

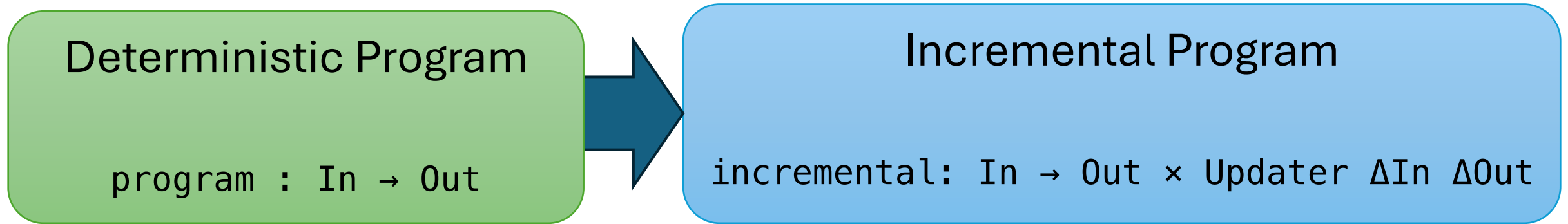
**mkUpdater**(**cache**: **Cache**, **updateFn**:  $\Delta\text{In} \times \text{Cache} \rightarrow \Delta\text{Out} \times \text{Cache}$ ): **Updater**  $\Delta\text{In}$   $\Delta\text{Out}$

```
def incrementalLength(in: List Int)
  : Int × Updater  $\Delta(\text{List Int})$   $\Delta\text{Int}$ 
= let len = length(in)
  return (len,
    mkUpdater(len, updateLength))
```

```
def updateLength( $\delta\text{in}$ :  $\Delta(\text{List Int})$ ,
  cachedLen: Int)
  :  $\Delta\text{Int} \times \text{Int}$ 
= let len' = length( $\delta\text{in}$ .insert)
  - length( $\delta\text{in}$ .remove)
  + cachedLen
  return ({ new: len',
    same?: len' == cachedLen },
    len')
```

- ✓ Encapsulate cache types
- ✓ Stackable

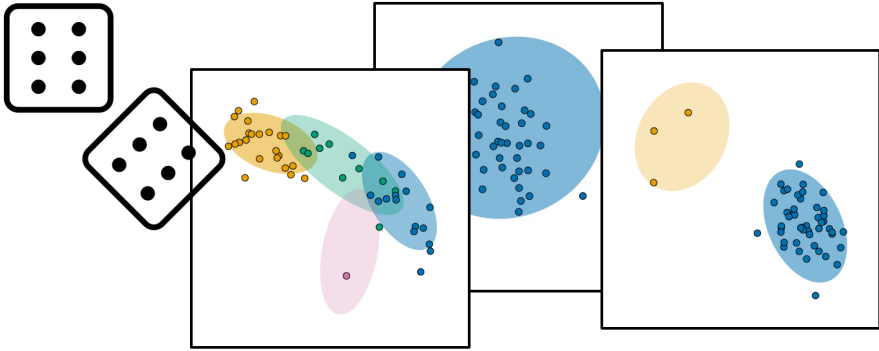
# Summary: Incrementalization



- Inspired by the **incremental  $\lambda$ -calculus** [Cai et al. 2014, Giarrusso et al. 2019]
  - **Typed, higher-order** setting
  - Every primitive operation has an incremental version
  - Loops are a special *higher-order* primitive: Track enough change information to **skip loop iterations**
- **asymptotic speedups**

# The task: two program transformations

Probabilistic Program  
“Trace Generator”



Density Program  
“Trace Evaluator”



$\mathbb{R}$

Deterministic Program

$A \longrightarrow B$

Incremental Program

$A \longrightarrow B$   
&  $\Delta A \longrightarrow \Delta B$

# Stage 1: Density Transformation

Probabilistic Program

model : Prob Trace Result

```
def model:  
  x ~ p1  
  y ~ p2  
  ...  
  return result
```

Deterministic Program

density : Trace  $\rightarrow$  Result  $\times \mathbb{R}$

```
def model_dens(trace):  
  (x, d_x) = p1_dens(trace.x)  
  (y, d_y) = p2_dens(trace.y)  
  ...  
  return (result,  
          d_x * d_y * ...)
```

***Novel aspects:***

- rich trace types with *unordered* collections
- unique *name generation*

# Trace Types [Lew et al. 2020]

The shape of the traces depends on the control flow!

| Program                                                                               | Trace type                                                                                              |
|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Named samples<br>$x \sim \dots; \quad y \sim \dots$                                   | Record type:<br>$\{x: \text{Subtrace}_1, y: \text{Subtrace}_2\}$                                        |
| Ordered iteration<br><b>for</b> $x$ <b>in</b> $\text{list}$ $\{ \dots \}$             | List Subtrace                                                                                           |
| Unordered iteration:<br><b>for</b> $\text{name}$ <b>in</b> $\text{set}$ $\{ \dots \}$ | Map Name Subtrace  |

# Why do we want names?

## Indexed clusters (1, 2, 3, ...)

```
{ clusters: 4,  
  weights: ...,  
  assignments: [4, 2, 3, 4, 1, ...] }
```

Remove  
first  
cluster  
(1)

```
{ clusters: 3,  
  weights: ...,  
  assignments: [3, 1, 2, 3, 1, ...] }
```

# Why do we want names?

## Indexed clusters (1, 2, 3, ...)

```
{ clusters: 4,  
  weights: ...,  
  assignments: [4, 2, 3, 4, 1, ...] }
```

Remove  
first  
cluster  
(1)

✗ Destroys incrementality!

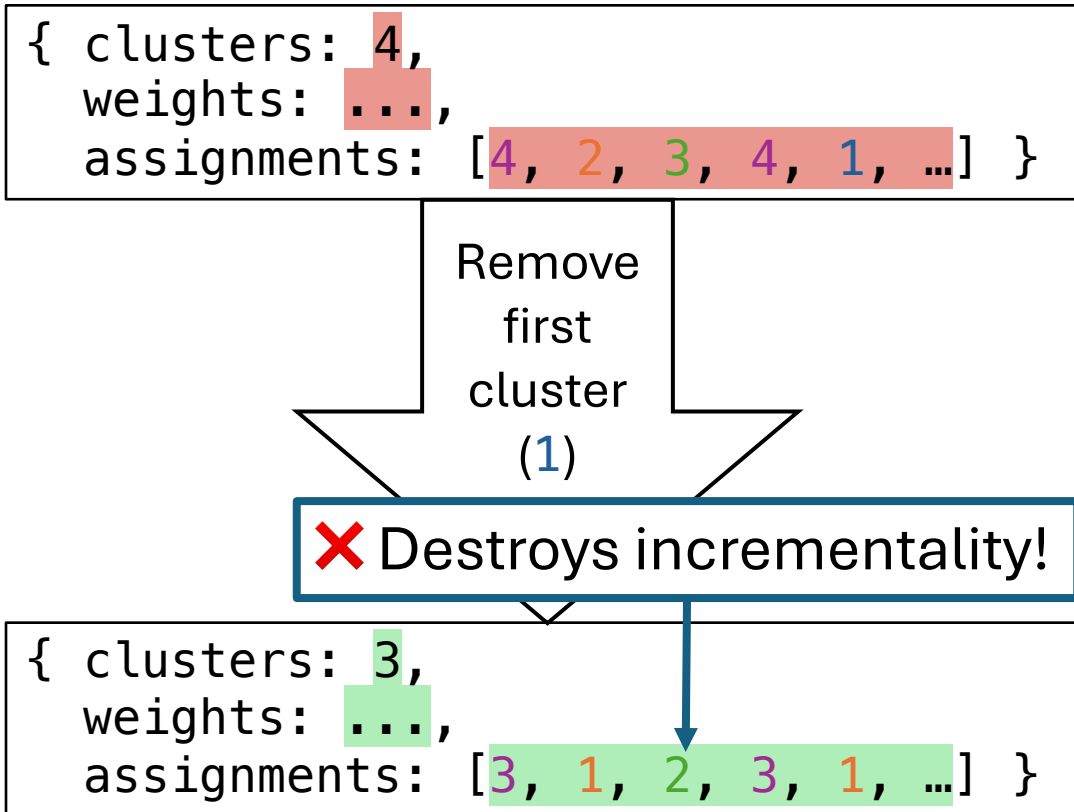
```
{ clusters: 3,  
  weights: ...,  
  assignments: [3, 1, 2, 3, 1, ...] }
```

## Named clusters (unique IDs)

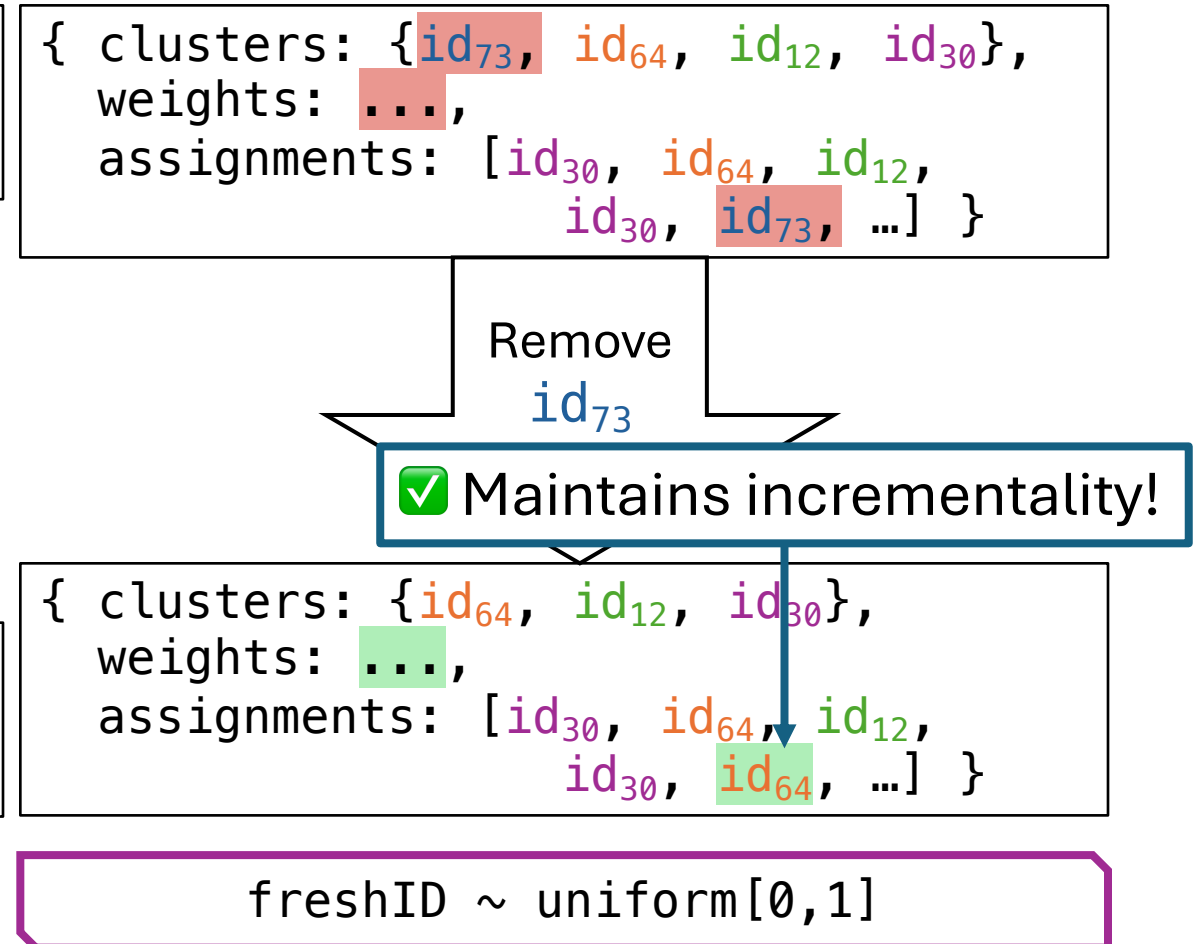
```
{ clusters: {id73, id64, id12, id30},  
  weights: ...,  
  assignments: [id30, id64, id12,  
                id30, id73, ...] }
```

# Names are better than indices

## Indexed clusters (1, 2, 3, ...)



## Named clusters (unique IDs)



# Overview

- Motivating Example
- Key ideas of our approach
- **Theoretical guarantees**
- Experimental results
- Summary

# Correctness

## Theorem.

Our **density transformation** computes a **valid density** of the program distribution with respect to the reference measures.

## Theorem.

Our **incrementalization** transformation yields updaters that turn input changes into output changes in a way **equivalent to a full recomputation**.

## Proof.

Uses **denotational** semantics and **logical relations**.

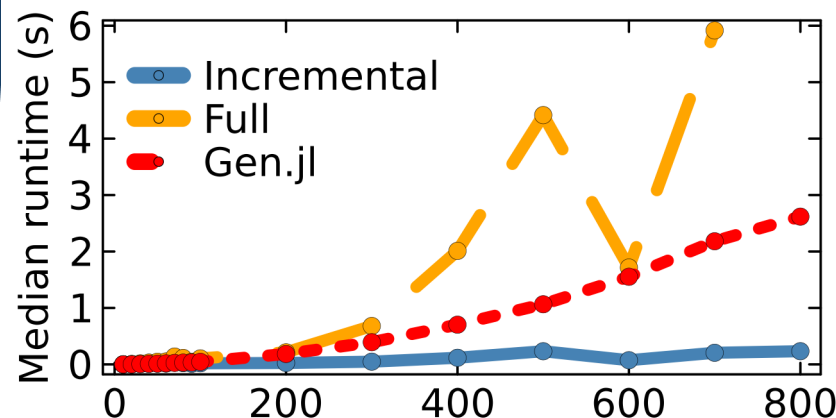
# Overview

- Motivating example
- Key ideas of our approach
- Theoretical guarantees
- **Experimental results**
- Summary

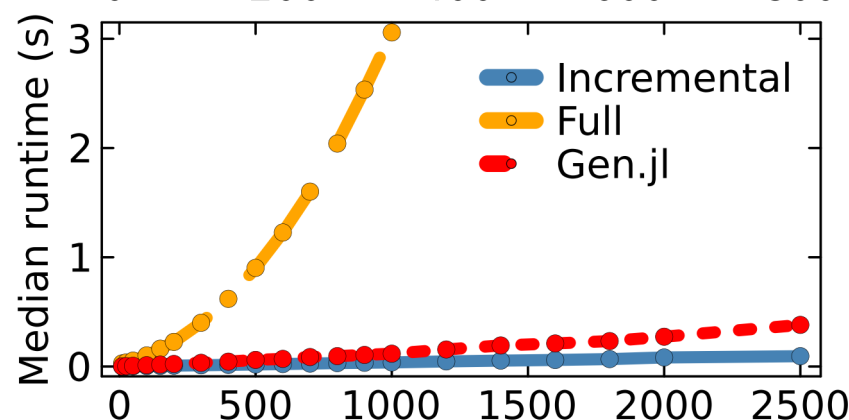
# Experiments

- MCMC inference using
  1. Our implementation
  2. Full recomputation
  3. Gen's experimental fine-grained incrementalization
- **Takeaway:** *Only our method achieves optimal asymptotic scaling for all models and inference algorithms.*

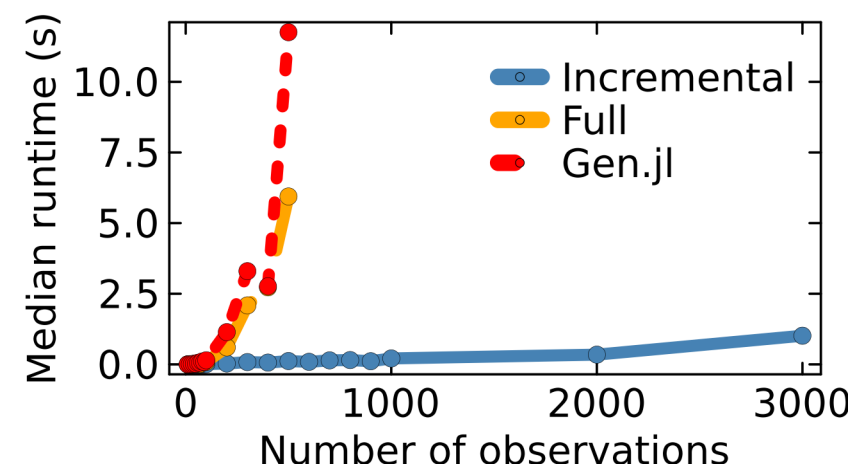
→ More details in the paper!



**Gaussian Mixture Model** in two dimensions with Poisson prior on number of components.



**Hidden Markov Model** with unknown  $100 \times 100$  transition matrix and continuous observations.



**Dirichlet Process Mixture Model** with infinitely many clusters, marginalized via a Chinese Restaurant Process.

# What's in the paper for you ...

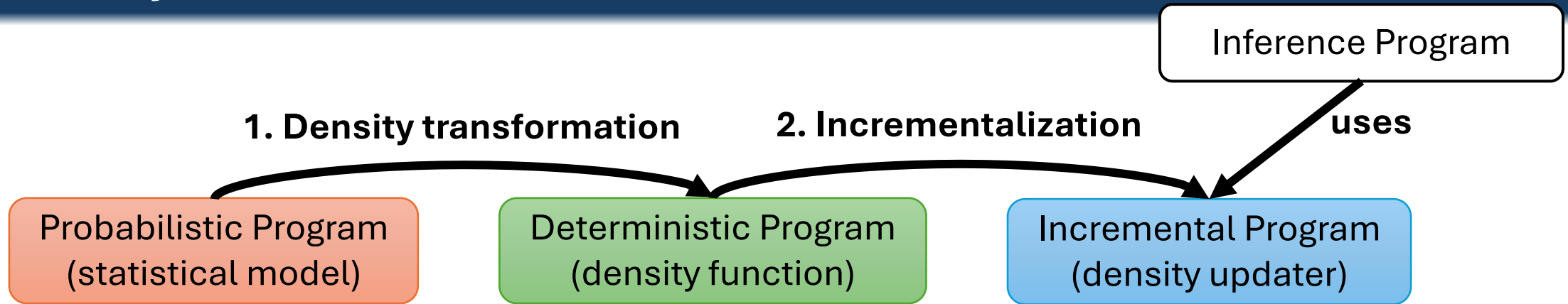
... if you care about  
**incremental  $\lambda$ -calculus:**

- **New application** domain
- **Typed** higher-order incrementalization with caching
- via **coinductive updater** types
- **Denotational** correctness proofs

... if you care about  
**probabilistic programming:**

- **Principled** approach to incrementalization in probabilistic programming
- Asymptotic efficiency gains for **open-universe & nonparametric** models
- **Unordered** collections
- Unique **name generation**

# Summary



- **Two stages:** decouples probability and incrementalization
- **Trace types & change types:** flexible change representations
- Improvements to the **caching incremental  $\lambda$ -calculus**
- **Correctness** proofs based on a **denotational** semantics
- Prototype implementation is **asymptotically faster** than SOTA for several models and inference algorithms

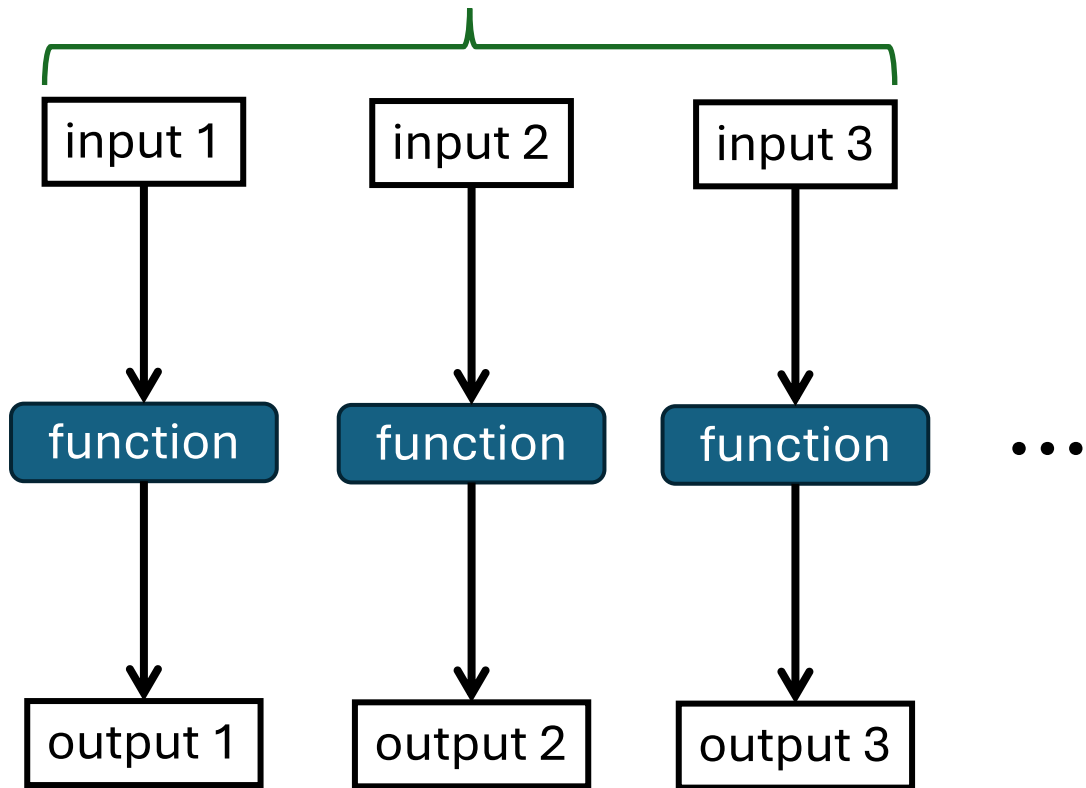
# Backup Slides

# New structure

- start with general structure of incremental computation (occurs in databases ...)
- but also, it occurs in MCMC!
- explain MCMC as before
- our approach is to \*reduce\* this probabilistic thing to deterministic incrementalization (show picture; we have novelties in both parts)
- motivating example, but with MCMC visualized (bad trace to good trace, with candidates excluded)
- so we want to incrementalize this deterministic functional program computing Trace -> Real
- background on incremental computation and the problems
- introduce updaters
- introduce our incrementalization (each primitive has its own incremental version; those are composed)
- then talk about the probabilistic densities

# Incremental Computing

**related** inputs

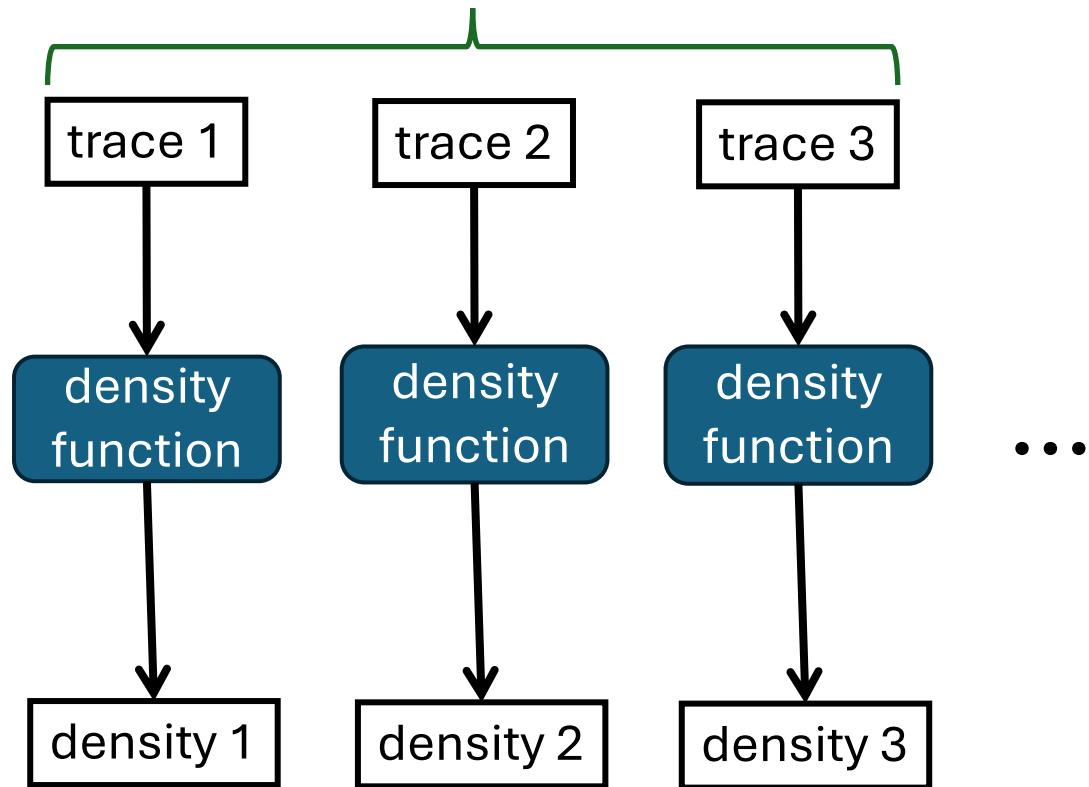


Needed for:

- database queries
- DOM layout
- compilers & IDEs
- ...

# Incremental Computing in Probabilistic Programming

**related** traces



Needed for:

- database queries
- website layout
- compilers & IDEs
- ...
- **Probabilistic Inference (MCMC)**

# Markov Chain Monte Carlo (MCMC)

**Goal:** sample from distributions

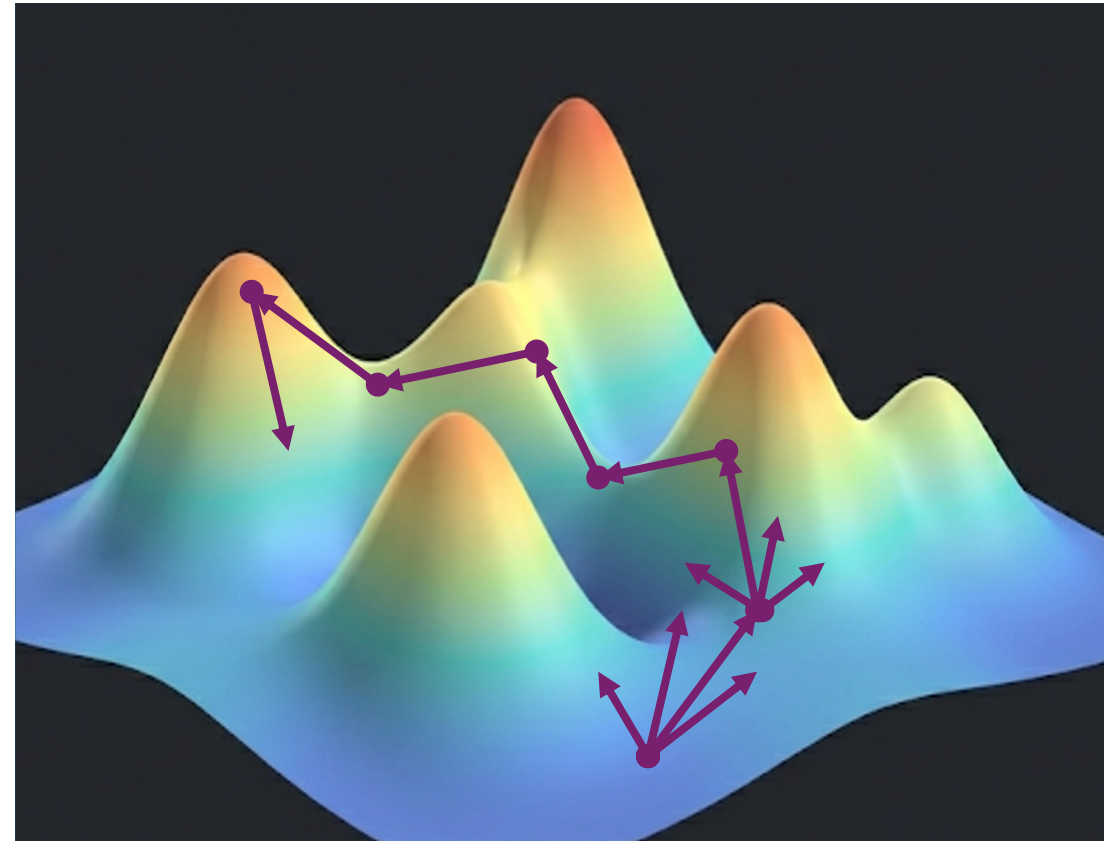
- In Bayesian statistics, physics, cosmology, phylogenetics, epidemiology, machine learning, econometrics, finance, ...



## Markov Chain Monte Carlo

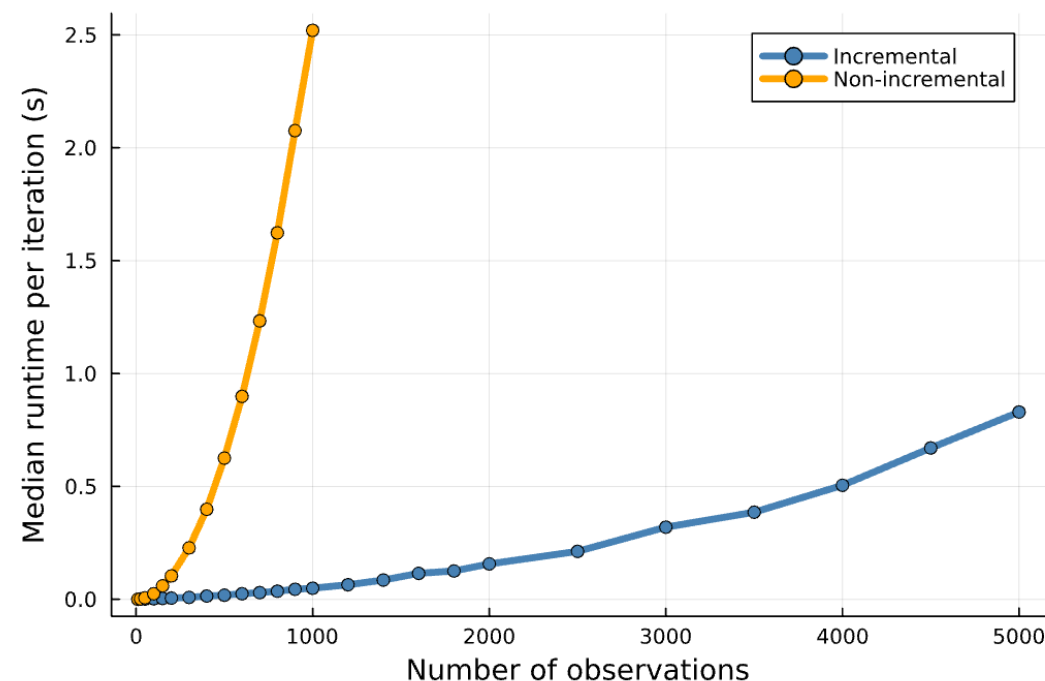
- Construct a **chain** of samples
- Designed to converge to the desired distribution
- Requires evaluating the likelihood at each step (**expensive!**)
- **Incremental computation** to avoid repeated work

Density plot of a distribution (height = density):



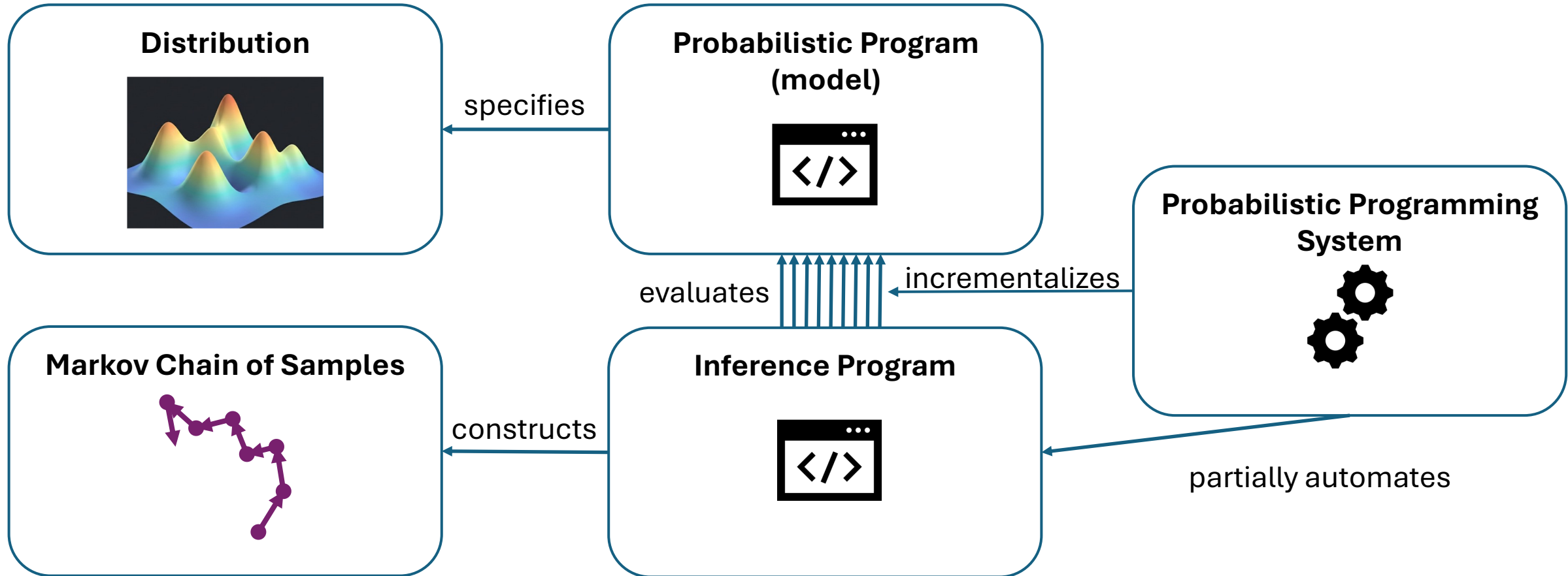
# Desiderata

- **Incremental** density evaluations
- **Automatic** program transformation
- Same **asymptotic efficiency** as hand-written
- **Programmable** inference
- **Compositional** correctness arguments



incremental vs. non-incremental

# Probabilistic Programming



Our approach computes densities incrementally ...

- ✓ ... in a **compositional** and provably **correct** way
- ✓ ... in  **$O(\text{hand-written})$  time**
- ✓ ... for **complex models** (e.g., Dirichlet Process Mixtures)
- ✓ ... for **sophisticated inference** programs

# Density functions

```
def model(n: Nat):  
  clusters ~ geometric(0.5)  
  params ~ for i in 1..clusters:  
    (mean, covariance) ~ paramPrior  
  weights ~ dirichlet(clusters, ...)  
  assignments ~ multinomial(weights, n)  
  data ~ for cluster in assignments:  
    normal(params[cluster])  
  return data
```

density(trace) = 1

**pmf** = Probability *Mass* Function  
**pdf** = Probability *Density* Function

- $\text{pmf}_{\text{geometric}(0.5)}(\text{trace.clusters})$
- $\prod_{i=1}^{\text{trace.clusters}} \text{pdf}_{\text{paramPrior}}(\text{trace.params}[i])$
- $\text{pdf}_{\text{dirichlet}(\dots)}(\text{trace.weights})$
- $\text{pmf}_{\text{multinomial}(\text{weights}, n)}(\text{trace.assignments})$
- $\prod_{i=1}^n \text{pdf}_{\text{normal}}(\text{params}[\text{assignments}[i]])(\text{trace.data}[i])$

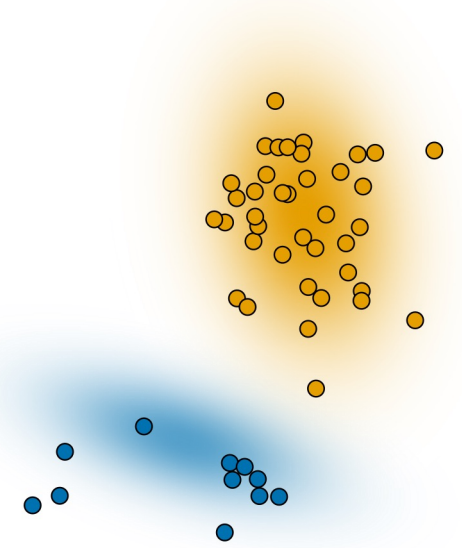
density = 1  
density = 0.25

density =  $2.34 \cdot 10^{-12}$   
density =  $2.34 \cdot 10^{-12}$   
density =  $6.65 \cdot 10^{-25}$

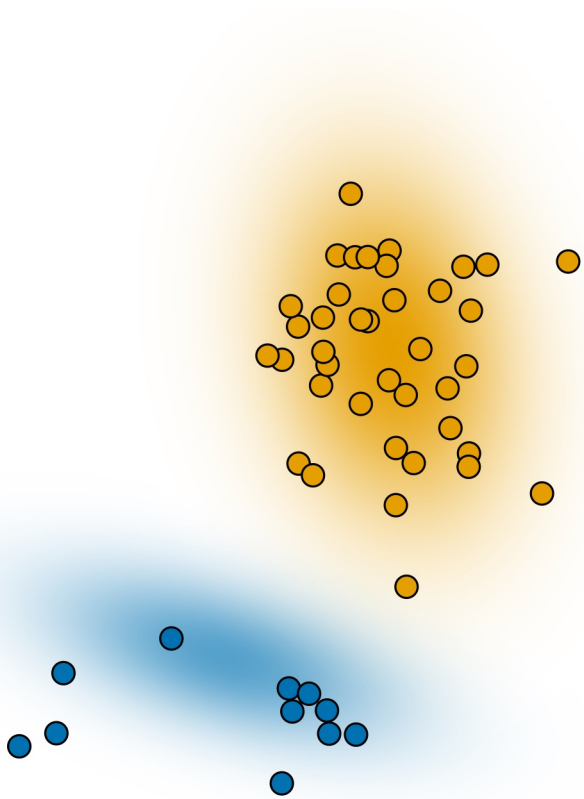
density =  $1.29 \cdot 10^{-111}$

## Trace:

```
{ clusters: 2,  
  params: [  
    { mean: (-1.4, -1.1), cov: ... },  
    { mean: (1.7, 6.0), cov: ... } ],  
  weights: [0.26, 0.74],  
  assignments: [2, 2, 1, 2, 1, ...]  
  data: [(1.7, 8.2), (2.8, 4.2), ...]  
}
```

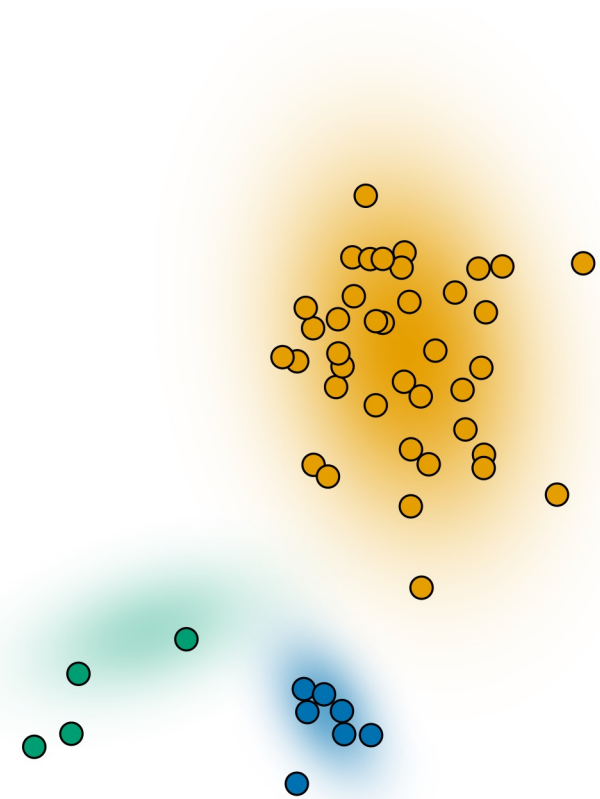


# Density comparison



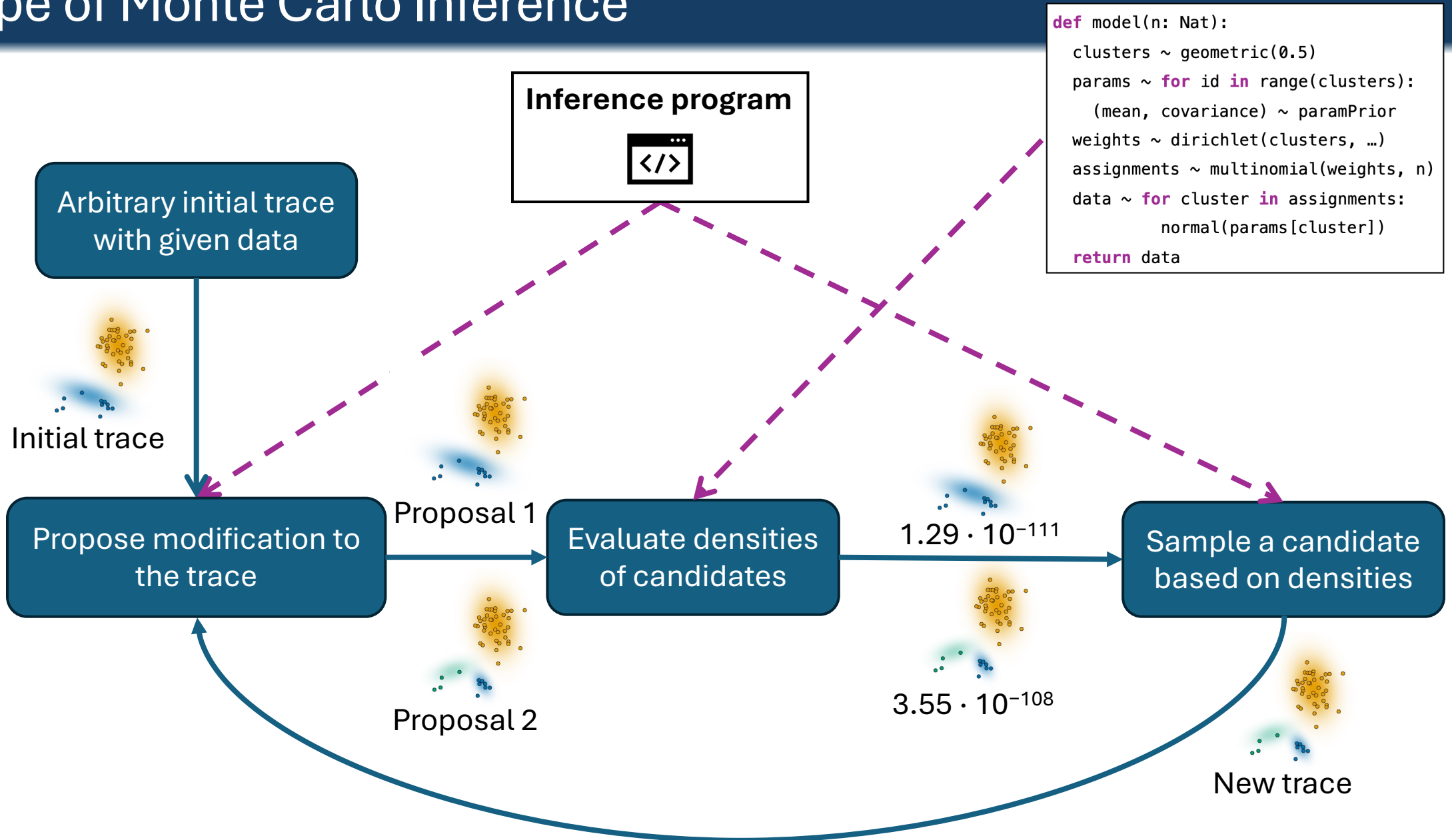
$$\text{Density} = 1.29 \cdot 10^{-111}$$

>

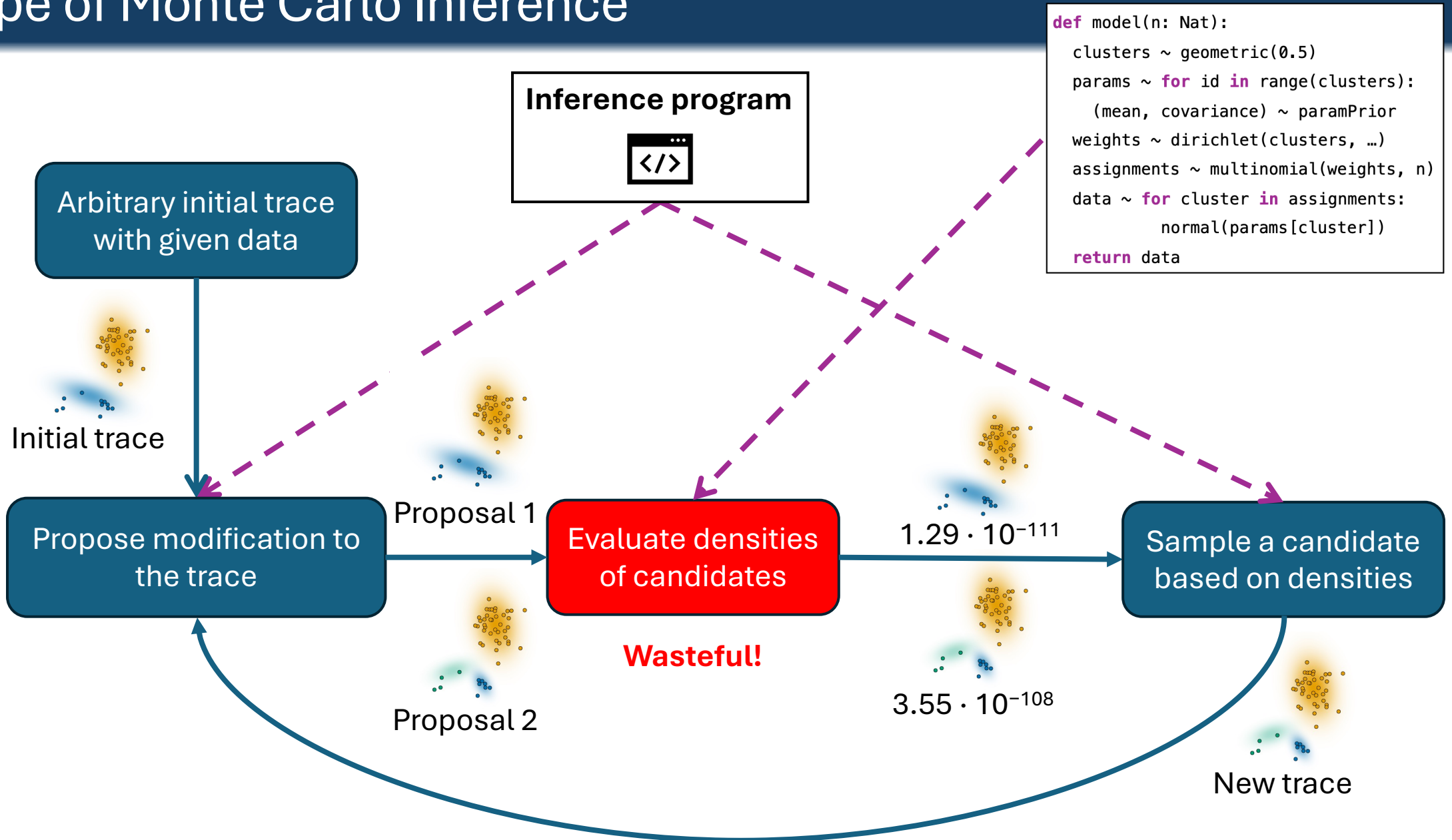


$$\text{Density} = 3.55 \cdot 10^{-108}$$

# Shape of Monte Carlo Inference



# Shape of Monte Carlo Inference



# Recomputing densities

```
def model(n: Nat):
    clusters ~ geometric(0.5)
    params ~ for i in 1..clusters:
        (mean, covariance) ~ paramPrior
    weights ~ dirichlet(clusters, ...)
    assignments ~ multinomial(weights, n)
    data ~ for cluster in assignments:
        normal(params[cluster])
    return data
```

density(trace) = 1

**pmf** = Probability *Mass* Function

**pdf** = Probability *Density* Function

•  $\text{pmf}_{\text{geometric}(0.5)}(\text{trace. clusters})$

$$\cdot \prod_{i=1}^{\text{trace.clusters}} \text{pdf}_{\text{paramPrior}}(\text{trace.params}[i])$$
 $\cdot \text{pdf}_{\text{dirichlet}(\dots)}(\text{trace.weights})$  $\cdot \text{pmf}_{\text{multinomial}(\text{weights}, n)}(\text{trace. assignments})$ 
$$\cdot \prod_{i=1}^n \text{pdf}_{\text{normal}}(\text{params}[\text{assignments}[i]](\text{trace.data}[i]))$$

Suppose we reassign the first data point from cluster 1 to cluster 2.

## Only `trace.assignments[1]` changes!

And only two factors in the density depend on it!

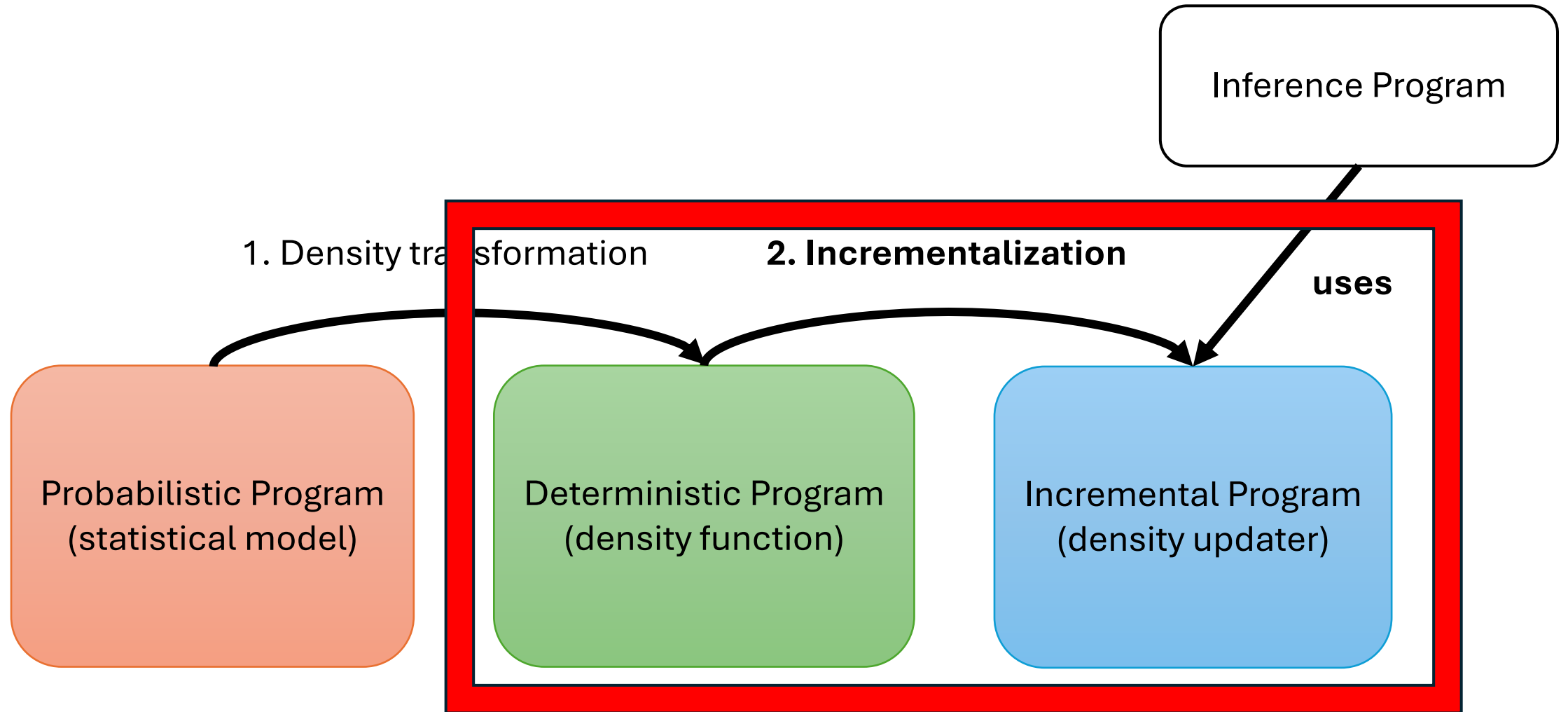
$$\text{pmf}_{\text{multinomial}(\text{weights}, n)}(\text{trace. assignments})$$

```
· pdf_normal(params[1])(trace.data[1])
```


$$\text{pmf}_{\text{multinomial}(\text{weights}, n)}(\text{trace. assignments}')$$

```
· pdf_normal(params[1])(trace.data[1])
```

# Stage 2: Incrementalization



# Example of an incremental function

```
length : List Int -> Int
```

```
incrementalLength(in: List Int) : Int × (Δ(List Int) → ΔInt)  
  let len = length(in)  
  return (len, updateLength)
```

```
updateLength(δin: Δ(List Int)) : ΔInt  
  let len' = length(δin.insert) - length(δin.remove) + len  
  return { new: len', same?: len' == len }
```

→ We need **caching**!

## Example of an incremental function with caching

```
updateLength( $\delta$ in:  $\Delta$ (List Int), cachedLen: Int) :  $\Delta$ Int  $\times$  Int  
  let len' = length( $\delta$ in.insert) - length( $\delta$ in.remove)  
           + cachedLen  
  return ({ new: len', same?: len' == cachedLen }, len')
```

### Problems:

**✗ Caches everywhere.**

**✗ We cannot stack changes** on top of each other.

# Example trace and its type

## Trace

```
{ clusters: {id73, id64,  
            id12, id30},  
  weights: {  
    id73: 0.34, id64: 0.51,  
    id12: 0.11, id30: .04},  
  assignments: [id73, id73,  
                id64, id30, id12, ...]  
  data: [(1.4, 0.6), ...]  
}
```

## Trace Type

```
{ clusters: Set Name,  
  weights: Map Name  $\mathbb{R}$ ,  
  assignments: List Name,  
  data: List ( $\mathbb{R} \times \mathbb{R}$ ),  
}
```

- 
-

# Example of a trace change

Reassigning the first data point from cluster `id73` to cluster `id64`:

```
δtrace = Sub {  
  assignments: {  
    change: {  
      1 => {  
        new: id64,  
        same?: false  
      }  
    }  
  }  
}
```

# An example where we're worse by a constant factor

