Hi everyone! My name is Alex Lew, and today I’ll be presenting some joint work with Mathieu Huot, my advisor Vikash Mansinghka, and Mathieu’s advisor Sam Staton.
So, I don’t think I need to convince anyone in this room that probabilistic programs are a very expressive way of writing down measures.

Just by incorporating the uniform distribution on the unit interval to a deterministic programming language, we can express a huge variety of probability distributions.
For example, we can use an if statement to define a Bernoulli distribution by checking a random number against a threshold.

```plaintext
fair_flip = do
    u <- sample
    if u < 0.5 then
        return 1
    else
        return 0
```

Probabilistic programs are an expressive representation for measures.
Probabilistic programs are an expressive representation for measures

```plaintext
std_normal = do
u1 <- sample
u2 <- sample
let r=sqrt(-2*ln(u1))
let t=2*pi*u2
return (r*cos(t),
        r*sin(t))
```

We can use the Box Muller transform to define a standard bivariate Gaussian distribution.
Probabilistic programs are an expressive representation for measures

\[
\text{uniform\_band} = \text{do} \\
\quad x \leftarrow \text{sample} \\
\quad y \leftarrow \text{sample} \\
\quad \text{return} (x, x + y)
\]

We can sample from restricted regions of a space,
Probabilistic programs are an expressive representation for measures

\[
\text{mcdonalds} = \begin{do}
    \text{x} \leftarrow \text{sample} \\
    \text{let} \quad t = 2\pi x - \pi \\
    \text{return} \quad (t \cos(t), \quad t \sin(t))
\end{do}
\]

Or from low-dimensional curves in an ambient space, even when they have measure zero,
Probabilistic programs are an expressive representation for measures

```haskell
bowtie_heart = do
  x <- std_normal
  let t = x * pi / 2
  r <- if |t| < 0.4
        then std_normal
        else return t
  return (r*cos(t), r*sin(t))
```

And by using probabilistic branches, we can create mixtures that are supported on really weird sets.
A natural question: just how weird can they get?

And a natural question we might ask is: just how weird can a probabilistic program get?
For example, using recursion, we can write a program that samples points within the complement of the Cantor Dust fractal, but is there a program that samples from the fractal itself?

We know that Cantor Dust takes up a measure-zero subset of the unit square, but that shouldn’t be an obstacle: some of the programs we already looked at had that property, too.
We can similarly write a recursive program that samples from the upside-down triangles that are punched out during the iterative construction of the Sierpinski triangle. But can we write a program supported on the Sierpinski triangle itself, a program that can sample all and only the points that are never punched out?

A natural question: just how weird can they get?
Now, these questions might seem kind of esoteric, so I am going to start by connecting them to some practical concerns we might have as language designers and theorists.

Then I’ll make the question of “how weird a probabilistic program can be” more precise, before presenting our main result. Finally I’ll try and tie things back to the motivation with some concrete consequences of the result for PPL design.
Why do we care?

- As PPL designers:
  Reasoning about coverage of our **automation tools** or **interfaces**

- As PPL theorists:
  Alternative **characterization / representation theorem**
  for working with denotations of probabilistic programs

So, why do we care about how weird a probabilistic program can be?

Well, as PPL designers, we might want to reason about the coverage of our automation tools or interfaces: do they handle any program a user could throw at us?

And as theorists, it is always nice to have a new representation theorem or characterization of the definable objects in some language, that helps us abstract away details of how a program was constructed.
Why do we care?

**Example:** Primitive Distribution Interface

```haskell
data Distribution a = Dist {
    sample  : M a,
    density : a -> R,
}
```

**Specification:** The density method should implement a density of the measure denoted by sample, with respect to a type-dependent base measure. (e.g., Lebesgue for \( a = \mathbb{R}^n \))

As an example of a design question that might be relevant here, consider the primitive distribution interface. Most PPLs have some notion of “primitive distribution,” and to add a new one, you have to implement some methods. For example, in many languages, those methods are sample, which draws a sample from the distribution in question, and density, which computes the density at a given point of the sampling distribution with respect to some base measure. For example, if you have a distribution over vectors of real numbers, the interface might require you to give a density with respect to the Lebesgue measure, which is just a fancy way of saying a standard Probability-101 density function.
Why do we care?

**Example:** Primitive Distribution Interface

data Distribution a = Dist {
    sample : M a,
    density : a -> R.
}

**Specification:** The density method should implement a density of the
measure denoted by sample, with respect to a type-dependent base measure.
(e.g., Lebesgue for \(\mathbb{R}^n\))

But a challenge with this interface is that a user might come to us and say, hey, I’m
trying to implement a new primitive, and I’ve written my sampler, but I’m not sure
how to implement the density function. Because the distribution that I’m sampling,
for example the uniform distribution over this curve here, doesn’t have a density
function in the plane. Its entire support is confined to a measure-zero set.

And while this example might seem contrived because this curvy M is not exactly a
primitive distribution that users are clamoring to use, we actually have run into this
issue in our lab.
Why do we care?

**Example**: Primitive Distribution Interface

```kotlin
data Distribution a = Dist {
  sample : M a,
  density : a -> R.
}
```

**Specification**: The density method should implement a density of the measure denoted by sample, with respect to a type-dependent base measure. (e.g., Lebesgue for $\alpha = \mathbb{R}^n$)

when trying to extend Gen with probability distributions for modeling the poses and rotations of objects in 3D space. We wanted to add various primitive distributions over unit vectors representing rotations, but just like the curvy M from before, these distributions don’t have densities in the standard sense. We went ahead and implemented them anyway, but parts of Gen’s design makes assumptions that are incompatible with them, so when you use them, you need to steer clear of certain features, which is of course not ideal.

So we would love to fix the primitive distribution interface to cover a broader set of programs.
And there are various proposals for how to do that. For example, in this really wonderful paper, Praveen and Ken Shan define a small language of base measures, and we could say that a Primitive Distribution needs to have a sampler, a density, and an explicitly defined base measure, with respect to which the density is correct. This idea of explicitly storing a base measure is also implemented by Chad Scherrer’s Measure Theory libraries in Julia, and a different set of Hausdorff base measures was also proposed by Alexey Radul and Boris Alexeev in an AISTATS paper in 2021.

Depending on how expressive the set of base measures is, this new interface will have higher coverage than our previous interface. But is it universal?

Why do we care?

**Example: Primitive Distribution Interface**

```haskell
data Distribution a = Dist { sample : M a,
    density : a -> R,
    base : Base a }
```

**Specification:** The `density` method should implement a density of the measure denoted by `sample`, with respect to `base`.
Why do we care?

**Example:** Primitive Distribution Interface

```haskell
data Distribution a = Dist {
  sample : M a,
  density : a -> R,
  base    : Base a
}
```

**Is this interface universal?**
Can we now, in theory, create primitives for any sampler a user writes?

If a user comes to us with some weird sampler program, can we always tell them a base measure that they can go implement a density function for?
A natural question: just how weird can they get?

So, that’s one example of why, even if you’re not compelled by weirdly-shaped distributions, you might care about the answer to this question.
But before we can answer it, we should probably clarify what exactly mean by the question.
First, we should set some ground rules: what language features and primitive functions are we considering?

We start with the deterministic simply-typed lambda calculus, with real vectors and Booleans as ground types, as well as tuples and functions. The symbol $c$ here ranges over primitive functions, which we'll talk more about in a moment.

Ground rules:
what language features + primitives?

$$\tau ::= 1 \mid \mathbb{R}^k \mid \mathbb{B} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

$$e ::= c \mid x \mid e_1 e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \mid \lambda x: \tau. e$$
We then add support for general recursion, via this mu operator, which binds the name \( f \) for recursive calls within the body \( e \) of the function being defined.

Ground rules: what language features + primitives?

\[
\begin{align*}
\tau & ::= \mathbf{1} \mid \mathbb{R}^k \mid \mathbb{B} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
 e & ::= c \mid x \mid e_1, e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \\
 & \mid \lambda x : \tau. e \mid \mu f : \tau_1 \rightarrow \tau_2. \lambda x : \tau_1. e
\end{align*}
\]
And finally we add the standard constructs for probabilistic computation, including the type $M\,\tau$ of measures over $\tau$, the terms \texttt{return}, \texttt{sample}, and \texttt{score}, and \texttt{do} notation inspired by Haskell for sequencing probabilistic computations.

At POPL 2019, Matthijs and Ohad and Sam showed how to give a denotational semantics to a language with these features, and even though they use various fancy techniques to interpret the language compositionally, a nice property is that closed probabilistic terms returning real vectors always denote some measure on $\mathbb{R}^n$.
So our question is just: what can we say about the definable measures, that is, the measures that lie in the image of this semantic function?

And a key factor controlling the answer to this question is going to be
Ground rules: what **primitives**?

**We will assume:** all primitives $c : \tau_1 \rightarrow \tau_2$ have PAP denotations

<table>
<thead>
<tr>
<th>Inclusions (for example):</th>
<th>Does not include (for example):</th>
</tr>
</thead>
<tbody>
<tr>
<td>• All primitives in numpy</td>
<td>• Indicator function for Cantor set</td>
</tr>
<tr>
<td>• arithmetic, trigonometry, comparison, floor, ceil, …</td>
<td>• Measurable bijections betw. $\mathbb{R}, \mathbb{R}^2$</td>
</tr>
<tr>
<td>• All functions in SpecialFunctions.jl</td>
<td>• Hilbert curve</td>
</tr>
<tr>
<td>• All functions in Haskell’s numeric typeclasses</td>
<td></td>
</tr>
<tr>
<td>• Closed under if, recursion</td>
<td></td>
</tr>
</tbody>
</table>

is going to be what **primitives** we let into our language.

There are lots of sensible choices, but for this work, we are going to assume that the primitives are all piecewise analytic under analytic partition, or PAP. This class of functions was introduced by Wonyeol and his colleagues in a great NeurIPS 2020 paper, and we extended the definition to cover higher-order and partial functions in a workshop last year.

To give you an intuitive sense of what’s being included here vs. what’s being left out, all of the operations exposed by numpy and Julia’s math libraries and Haskell’s numeric libraries all fit the definition of PAP, and any program you could possibly write that composes those operations using branching and recursion also fit the definition. Some examples of things that don’t fit the definition of PAP are the indicator function for the Cantor set, the Hilbert space-filling curve, or any measurable bijection between the one-dimensional number line and the plane.

So our question about how weird a probabilistic program can get is really about programs that you write down using the kinds of primitives that real-world libraries tend to expose. If you instead were to allow arbitrary measurable functions as
primitives, we already know that any s-finite measure is definable, including all probability measures, so it makes sense to search for a slightly more restricted setting still general enough to encompass all sorts of strange programs people might write in practice.
Now, with this choice in hand, we have a more precise problem statement, which is: when the deterministic primitives in our language are PAP, what can we say about the definable measures?

\[[[e]] \mid e \in M \mathbb{R}^n\]
Outline

- Why do we care?
- Defining the problem
  - Main result
- Consequences for PPL designers and theorists

So, what did we find?
Let me first state our main result informally: For any probabilistic program returning a vector of reals, its denotation is a measure supported on a countable union of smooth submanifolds of Euclidean space.

So for example, for this strange program we looked at earlier, we can carve up its support into a bunch of different smooth manifolds, each colored differently in this illustration. Some of the manifolds are 1-dimensional, and some are 2-dimensional. This program doesn’t, but we could also have zero-dimensional manifolds, which are individual points.

Now, this is an informal statement, because we haven’t really explained what we mean by “supported on.” We’ll make it formal in a moment, but I want to point out two features of this decomposition of the plane into manifolds:

First, sampling this probabilistic program will always land us somewhere in one of these manifolds. That’s one of the things we mean by “is supported on”: we never land outside of the colored regions and curves. But it’s also important that these manifolds are “minimal” in some way: if we removed any area from the 2-dimensional manifolds, or any length from the one-dimensional manifolds, then we...
would no longer catch all the samples from our program; the program could sample in the deleted regions.

In order to make this result formal, we’ll need a couple definitions.
Hausdorff measure

**Definition.** The *d*-dimensional Hausdorff measure assigns to a set \( A \subseteq \mathbb{R}^n \) the value

\[
\mathcal{H}^d (A) = \frac{d}{2^d} \sup_{\delta > 0} \inf \left\{ \sum_{i=1}^{\infty} (\text{diam } U_i)^d \mid A \subseteq \bigcup_{i=1}^{\infty} U_i, \text{diam } U_i < \delta \right\}
\]

**Intuition:**
- \( d = 0 \): Counts the number of points in \( A \)
- \( d = 1 \): Sums arc lengths of curves in \( A \)
- \( d = 2 \): Sums surface areas of surfaces in \( A \)
- \( d = 3 \): Sums volumes of solids in \( A \)

First, the \( d \)-dimensional Hausdorff measure, which is a function that takes as input a set, and outputs a notion of the size of the set that’s specialized to a dimension \( d \).

We have a formal definition on the slide here, to reassure you that this can be rigorously defined, but what we need for this talk is a much simpler intuition. When the dimension \( d = 0 \), we’re just counting the number of points; when \( d = 1 \), we’re measuring the lengths of any line segments or curves in \( A \); when \( d = 2 \), we’re measuring the area of any surfaces or shapes in \( A \), and so on.
Hausdorff measure

**Definition.** The \( d \)-dimensional Hausdorff measure assigns to a set \( A \subseteq \mathbb{R}^n \) the value

\[
\mathcal{H}^d(A) = \frac{a_d}{\delta^d} \cdot \sup_{\delta > 0} \left\{ \sum_{i=1}^{\infty} (\text{diam } U_i)^d \mid A \subseteq \bigcup_{i=1}^{\infty} U_i, \text{diam } U_i < \delta \right\}
\]

**Intuition:**
- \( d = 0 \): Counts the number of points in \( A \)
- \( d = 1 \): Sums arc lengths of curves in \( A \)
- \( d = 2 \): Sums surface areas of surfaces in \( A \)
- \( d = 3 \): Sums volumes of solids in \( A \)

So, for example, this set outlined in pink here has zero-dimensional Hausdorff measure infinity, because it has infinitely many points, but it has finite 1-dimensional Hausdorff measure, equal to the sum of the arc lengths of the curves that it contains. Its 2- and 3-dimensional Hausdorff measures are 0, because the set has no area or volume.
By contrast, this set A has 1-dimensional Hausdorff measure infinity, intuitively because it has infinitely many curves whose lengths would be summed. Its 2-dimensional Hausdorff measure is finite, and equal to its area, and above dimension 2, the measure is 0.
s-Hausdorff measures

**Definition.** An s-Hausdorff measure on $\mathbb{R}^n$ is a measure $\mu$ that decomposes as

$$\mu(A) = \sum_{d=0}^{n} \mathcal{H}^d (A \cap M_d),$$

where each $M_d$ is a countable union of $d$-dimensional smooth manifolds.

Building on the definition of the Hausdorff measure, we define a class of measures called s-Hausdorff measures. $\mu$ is an s-Hausdorff measure if, when given a set $A$, it spits out a sum of Hausdorff measures of each dimension up to $n$. But for each dimension, it only looks at the size of $A$’s intersection with a particular set $M_d$, and for each $d$, this set $M_d$ is a union of $d$-dimensional manifolds.

So for example, if we look at this probabilistic program whose support we carved up earlier, we can define an s-Hausdorff measure based on the way we carved up the support.

All of our one-dimensional manifolds – the curves highlighted here – together will form the set $M_1$, and the 2D manifolds, the regions highlighted in green, red, blue, and yellow, will together form the set $M_2$. The green, red, blue, and yellow areas together from $M_2$, and all the 1D curves here together form $M_1$.

If we take some other set $A$ and ask for its measure under $\mu$, we would get the area of its intersection with the triangular highlighted regions, PLUS the length of its intersection with the blue curve.
Main result (formal)

**Theorem.** If $\vdash e : M \mathbb{R}^n$, then there exists an $s$-Hausdorff measure $\mu_e$ such that:

- $\mu_e(A) = 0 \iff \|e\|(A) = 0$
- $\mu_e(A) > 0 \iff \|e\|(A) > 0$
- $\mu_e(\{e\}) \setminus \{\mu_e\} = 0$

(implies $\|e\|$ is $(0, \infty)$-absolutely continuous with respect to $\mu_e$).

Using $s$-Hausdorff measures, we can state a formal version of our theorem, which is that if we have a program $e$, then there exists an $s$-Hausdorff measure $\mu_e$ with these properties: $\mu$ assigns measure zero to a set exactly when our program does, $\mu$ assigns non-zero measure to a set exactly when our program does, and then there’s this third part of the result that’s more complicated, and becomes relevant when our program is not sigma-finite.
Main result (formal)

**Theorem.** If $\vdash e : M \mathbb{R}^n$, then there exists an s-Hausdorff measure $\mu_e$ such that:

- $\mu_e(A) = 0 \iff \|e\|(A) = 0$
- $\mu_e(A) > 0 \iff \|e\|(A) > 0$
- $\mu_e(\infty) \setminus \infty = 0$

For each dimension $d = 0, \ldots, n$, a choice of $M_d$ (countable union of $d$-dim. manifolds)

To unpack this, remember from the previous slide that choosing an s-Hausdorff measure is the same as choosing, for each dimension $d$, a countable union of manifolds of that dimension.

Then this notation $\mu(A)$ is just saying, the total size of $A$ as measured separately on each of these manifolds: the area of $A$'s intersection with the 2-dimensional manifolds, plus the length of its intersection with 1-dimensional manifolds, plus the number of points that intersect with zero-dim manifolds.

And the notation $e(A)$ is saying, the total probability or measure of $A$ under the user’s program.

So what this is saying is exactly what we said informally before: anything the program assigns mass to, is part of these manifolds, and every part of these manifolds – every bit of length of the 1D manifolds, every bit of area from the 2D manifolds, and so on – have some probability of being hit by the program.
Returning to the Cantor example...

Now if we return to the example from the beginning of the talk, we can definitively say no program samples from the Cantor dust fractal.

Why? Well, any such program would assign positive measure to the Cantor Dust fractal, so we need an $s$-Hausdorff measure that assigns positive measure. The Cantor set has 1-dimensional Hausdorff measure 0. So in order for an $s$-Hausdorff measure $\mu$ to assign positive measure to the fractal, it has to do so using only zero-dimensional manifolds, that is, isolated points. But we can only stuff countably many points into the zero-dimensional part of an $s$-Hausdorff measure, and the Cantor set has uncountably many points.
So we can answer some of our abstract questions, but how does this connect back to the more concrete motivation we looked at in the beginning?
Consequences for PPL designers

**Theorem.** If $\vdash e : M \mathbb{R}^n$, then there exists an s-Hausdorff measure $\mu_e$ such that:

- $\mu_e(A) = 0 \iff \llbracket e \rrbracket(A) = 0$
- $\mu_e(A) > 0 \iff \llbracket e \rrbracket(A) > 0$
- $\mu_e(\infty (\llbracket e \rrbracket) \setminus \infty [\mu_e]) = 0$

**This implies:**

- Every program $e$ has a **density** $\rho_e$ (possibly infinite) with respect to $\mu_e$.

Well, one immediate consequence of our theorem is that for every program $e$, there is some s-Hausdorff base measure with respect to which it has a density.

And these densities are quite convenient.
Convenient properties of s-Hausdorff Densities

1) **S-Hausdorff densities are essentially unique.**
   
   If \([e]\) is absolutely continuous w.r.t. two s-Hausdorff measures \(\mu_1\) and \(\mu_2\), then \(\frac{d[e]}{d\mu_1} = \frac{d[e]}{d\mu_2}\), \([e]\)-almost-everywhere.
   
   So we can write \(\rho_e : \mathbb{R}^n \to [0, \infty)\), and, just like usual densities, this picks out a function up to a measure-zero set.

Suppose we have a couple programs \(e_1\) and \(e_2\), and that \(e_2\) “covers the support” of \(e_1\) – which is what this absolute continuity notation implies.

The first thing our corollary says is just about the individual program \(e_1\).

It says, there might be lots of s-Hausdorff measures that \(e_1\) has a density with respect to, but all those densities will be almost everywhere equal to one another.

So there really is no choice here: for every program \(e\), there is an essentially unique density function \(\rho_e\) with respect to the various s-Hausdorff measures that \(e\) is absolutely continuous w.r.t.

Next, if the supports of \(e_2\) and \(e_1\) actually match each other, because the absolute continuity condition applies in both directions, then we can just divide their densities to get valid Radon-Nikodym derivatives. That is, these densities are compatible with one another.

And even if the support condition only holds in one direction, then the we can still divide the densities by one another, we just need to also multiply by an indicator.
function. That multiplication by the indicator function is essentially doing a “dimension check” as has been previously proposed to correctly compute density ratios of programs with discrete and continuous components.
Convenient properties of s-Hausdorff Densities

2) S-Hausdorff densities can be divided.

Let \( e_1, e_2 : M \to \mathbb{R}^n \), and suppose both \( \| e_1 \| \ll \| e_2 \| \) and \( \| e_2 \| \ll \| e_1 \| \).

Then \( \frac{d[e_1]}{d[e_2]} = \frac{\rho_{e_1}}{\rho_{e_2}} \).

Suppose we have a couple programs \( e_1 \) and \( e_2 \), and that \( e_2 \) "covers the support" of \( e_1 \) – which is what this absolute continuity notation implies.

The first thing our corollary says is just about the individual program \( e_1 \).

It says, there might be lots of s-Hausdorff measures that \( e_1 \) has a density with respect to, but all those densities will be almost everywhere equal to one another.

So there really is no choice here: for every program \( e \), there is an essentially unique density function \( \rho_{e} \) with respect to the various s-Hausdorff measures that \( e \) is absolutely continuous w.r.t.

Next, if the supports of \( e_2 \) and \( e_1 \) actually match each other, because the absolute continuity condition applies in both directions, then we can just divide their densities to get valid Radon-Nikodym derivatives. That is, these densities are compatible with one another.

And even if the support condition only holds in one direction, then we can still divide the densities by one another, we just need to also multiply by an indicator.
function. That multiplication by the indicator function is essentially doing a “dimension check” as has been previously proposed to correctly compute density ratios of programs with discrete and continuous components.
Convenient properties of s-Hausdorff Densities

3) S-Hausdorff densities can be divided even if supports don’t match.

Let $e_1, e_2 : M \to \mathbb{R}^n$, and suppose only that $\|e_1\| \ll \|e_2\|$.

Then there exists a set $A$ with $\|e_1\|A^c = 0$, s.t.

$$\frac{d\|e_1\|}{d\|e_2\|} = \frac{\rho_{e_1}}{\rho_{e_2}} \cdot 1_A.$$ 

Checking membership in $A$ implements a “dimension check” like that in Wu et al. 2018 or Radul and Alexeev 2021.

Suppose we have a couple programs $e_1$ and $e_2$, and that $e_2$ “covers the support” of $e_1$ – which is what this absolute continuity notation implies.

The first thing our corollary says is just about the individual program $e_1$.

It says, there might be lots of s-Hausdorff measures that $e_1$ has a density with respect to, but all those densities will be almost everywhere equal to one another.

So there really is no choice here: for every program $e$, there is an essentially unique density function $\rho_{e_1}$ with respect to the various s-Hausdorff measures that $e$ is absolutely continuous w.r.t.

Next, if the supports of $e_2$ and $e_1$ actually match each other, because the absolute continuity condition applies in both directions, then we can just divide their densities to get valid Radon-Nikodym derivatives. That is, these densities are compatible with one another.

And even if the support condition only holds in one direction, then the we can still divide the densities by one another, we just need to also multiply by an indicator
function. That multiplication by the indicator function is essentially doing a “dimension check” as has been previously proposed to correctly compute density ratios of programs with discrete and continuous components.
Consequences for PPL designers

data Distribution a = Dist {
    sample : M a,
    density : a -> R,
    dim : a -> Nat
}

Specification: The density method should implement a density of the measure denoted by sample, with respect to some s-Hausdorff measure. The dim method should report the lowest $d$ for which the given point is in $M_d$ for the chosen s-Hausdorff measure.

All of this suggests that a distribution interface based on these densities might be reasonable. Our new specification can be that density should tell you the density with respect to an s-Hausdorff measure, and maybe we’d also want something like a dimension method that tells us the dimension of the manifold that a particular point lies on.
Consequences for PPL designers

Our work shows:

• It really is a general solution: doesn’t leave any programs out
• Can divide these densities soundly to get importance weights, up to dimension check
• Can extend to programs that return lists of reals with no maximum dimension

In fact, a very similar interface was proposed by Alexey Radul and Boris Alexeev in their AISTATS paper called “The Base Measure Problem and its Solution.”

Our work shows that their solution really is a general solution: it doesn’t leave any programs out.

We also show that the densities involved can be safely divided by one another to get density ratios, and though we haven’t talked about it today, our result does extend to programs that return lists of reals with no maximum dimension.
What do the posteriors of probabilistic programs look like?

So, the title of our talk is what do the posteriors of probabilistic programs look like? And what we’ve shown is that in an expressive language with recursion and PAP primitives, any program’s support will decompose into a countable union of smooth manifolds.

We’ve also shown that this has interesting implications for designers of PPLs, because it implies that all programs have essentially canonical densities with respect to a certain kind of base measure.
We would like to explore the efficient implementation of a density interface based on s-Hausdorff measures in Gen. Our focus in this talk has been on primitive distributions, but it would also be great to find ways to compositionally automate joint s-Hausdorff densities and the estimation of marginal s-Hausdorff densities.

That’s all I have for today, but I’m happy to take questions!

Our next steps…

- Constructing s-Hausdorff densities compositionally
- Efficient implementation within PPL (like Gen.jl)