Slice: Type-Directed Discretization of Continuous Probabilistic Programs

ANONYMOUS AUTHOR(S)

Probabilistic programming languages (PPLs) are powerful tools for statistical modeling, but performing exact inference on programs with continuous random variables remains a major challenge. Many PPLs capable of exact inference are restricted to purely discrete models, while those supporting continuous distributions typically rely on approximate inference. This paper introduces Slice, a language and compiler that bridges this gap by automatically and soundly transforming probabilistic programs with continuous distributions into equivalent, purely discrete counterparts. The core of our approach is a novel type system that statically analyzes how continuous variables are compared against constant values throughout the program. This type-directed analysis determines a set of cut points used to partition continuous domains into a finite number of intervals. Slice then replaces the original continuous distributions with discrete distributions over these intervals, enabling the transformed program to be solved by off-the-shelf exact inference engines for discrete PPLs. We formalize this transformation, prove its soundness, and provide an open-source implementation. Our evaluation on a range of benchmarks demonstrates that this approach is significantly faster than existing symbolic inference techniques for hybrid programs, effectively combining the expressiveness of continuous modeling with the power of exact discrete inference.

1 Introduction

Probabilistic programming languages (PPLs) have emerged as a powerful tool for building complex statistical models and reasoning about uncertainty [1–8, 10–18]. A key challenge in PPLs is inference: computing the probability of an event given a model. The tractability of inference often depends on the kinds of random variables used. For programs with only discrete random variables, powerful exact inference engines can compute probabilities precisely. However, many real-world models require continuous random variables to represent quantities like time, distance, or sensor readings. For these programs, exact inference is often intractable, forcing practitioners to rely on approximate methods like Monte Carlo simulation or variational inference.

Existing discrete PPLs like Dice [10] and Roulette [12] offer powerful, exact inference but cannot handle continuous distributions. At the other end of the spectrum, popular general-purpose PPLs like Stan [2] and Pyro [1] excel at approximate inference for continuous models but sacrifice exactness. Symbolic inference systems like SPPL [14] can reason exactly about certain mixed discrete-continuous models, but for a restricted class of programs.

Another related line of work is on probabilistic program abstraction. For example, [9] use abstract interpretation to translate a hybrid discrete-continuous program into a finite Boolean program. Their approach is distributionally sound, but it relies on the user to provide a set of predicates to abstract the program with. In contrast, our work automatically infers discretization points from the program text itself.

This paper introduces Slice, which occupies a unique position in the PPL design space: it automatically and soundly translates a broad class of programs with continuous variables into equivalent discrete programs. This automated, sound discretization unlocks the ability to use powerful discrete exact inference engines on models that were previously out of their reach. The key insight is to analyze how continuous variables are used within the program. Specifically, Slice uses a type system to track all the constant thresholds against which continuous variables are compared. These comparison points are then used to intelligently discretize the continuous distributions into a finite number of buckets. When the resulting program is purely discrete it can then be solved by an off-the-shelf exact inference engine such as Dice [10]. This type-directed

discretization allows developers to model with continuous distributions while still benefiting from the speed and precision of exact discrete inference.

For instance, consider a simple program that checks if a uniform random variable is less than 0.5:

```
uniform(0, 1) < 0.5
```

Slice's type system determines that 0.5 is the only relevant "cut point." It automatically transforms the program into an equivalent discrete version:

```
discrete(0.5, 0.5) < 1
```

The continuous uniform distribution is replaced by a discrete one with two equally likely outcomes (0 and 1), and the comparison is transformed accordingly.

The true power of this approach, however, lies in its ability to handle programs where these relationships are not immediately obvious. A single continuous variable might be compared against multiple different constants across separate parts of the program. Further, these constraints are propagated through complex control flow. For example, a value drawn from a continuous distribution might be returned from one branch of a conditional, passed through a higher-order function, or stored in a data structure before it is eventually compared to a constant. Slice's type inference systematically tracks these dependencies, ensuring that all relevant cut points are associated with the original distribution, regardless of how the value flows through the program. This robust, whole-program analysis is the key to handling expressive models, as we demonstrate in §2.

The Slice language itself is a statically-typed functional language that supports a rich set of features, including higher-order functions, recursion, pairs, lists, and even mutable references. Our transformation handles all of these features, allowing programmers to write expressive, high-level models that can still be analyzed with exact discrete inference.

Contributions. We present and evaluate our framework for type-directed discretization via the following contributions:

- **Type System (§3)** We introduce a novel type system that identifies which expressions are discretizable by analyzing how they are compared against constant values throughout the program.
- **Type Inference (§4)** We develop a type inference algorithm that automatically collects the set of constant thresholds needed for discretization for each expression.
- **Discretization (§5)** We present a type-directed program transformation that soundly converts continuous distributions and their comparisons into discrete counterparts.
- **Soundness (§6)** We prove that our discretization method is sound, establishing that the discretized program preserves the semantics of the original program.
- **Implementation (§7)** We provide an open-source implementation of Slice that compiles programs for the Dice exact inference engine. Our implementation also includes a direct interpreter for partially discretized programs.
- **Evaluation (§8)** We conduct an empirical evaluation on a range of benchmarks, demonstrating that our approach is significantly faster than prior symbolic techniques on hybrid programs while maintaining correctness.

Finally, we discuss related and future work (§9).

2 Slice by Example

This section presents simple examples that highlight the key features of our probabilistic programming language Slice, aiming to emphasize how Slice meets the challenge of performing

exact inference efficiently. Slice extends a core functional language with traditional programming language constructs such as conditionals, local variables, and functions, augmented with constructs for probabilistic programming such as sampling and conditioning. Each example in the subsections is intended to showcase these features. Later in the section, we also mention examples that can only be partially discretized but nonetheless preserve the semantics across both the original and discretized program.

2.1 Simple branching on a uniform distribution

We start with a simple example that illustrates the core idea of Slice. Consider the following program, which models a fair coin flip by checking if a random number from a uniform distribution is less than 0.5:

```
uniform(0,1) < 0.5
```

Slice's type system analyzes this program and identifies that the only comparison point for the continuous variable is 0.5. This single split point partitions the range $\mathbb{R} = (-\infty, +\infty)$ into two intervals of equal probability mass: $(-\infty, 0.5)$ and $[0.5, +\infty)$. Slice then transforms the program into an equivalent discrete version:

```
discrete(0.5, 0.5) < 1
```

Here, `discrete(0.5, 0.5)` represents a choice between two outcomes (0 or 1), each with probability 0.5. The outcome 0 corresponds to the original value being in $(-\infty, 0.5)$, and 1 corresponds to it being in $[0.5, +\infty)$. The comparison is updated to check if the outcome is less than 1, which is true only for outcome 0, correctly preserving the original program's meaning. This transformation is guided by tracking comparison points, which are determined by type inference as shown in the annotated program below:

```
(uniform(0, 1) : float[{<0.5}; T]) < (0.5 : float[{<0.5}; {0.5}])
```

The type `float[{<0.5}; T]` for the uniform sample indicates that the bound bag contains the comparison bound "< 0.5" and the value bag is $\top$ (can take any value in the range). The type `float[{<0.5}; {0.5}]` for the constant 0.5 indicates that the bound bag contains < 0.5 and the value bag contains only the single value 0.5.

In general, a type **float**`[bound-bag; value-bag]` indicates:

$$\textbf{float}[\ \underbrace{\text{bound-bag}}_{\substack{\text{set of comparison bounds}\\ \text{(e.g., } \{<0.1, <0.5, \leq 0.8\}\text{)}\\ \text{or } \top \text{ (unbounded)}}}\ ;\ \underbrace{\text{value-bag}}_{\substack{\text{set of concrete values}\\ \text{(e.g., } \{0.1, 0.4, 0.5\}\text{)}\\ \text{or } \top \text{ (any value)}}}\ ]$$

The bound bag is determined by *how the expression is used* (i.e., what it's compared against) and the value bag is determined by *what concrete values the expression can take*. The discretization is driven by the bound bag: a float value with n comparison bounds is discretized into $n + 1$ possible discrete values.

In the example above, the expression **uniform**`(0,1)` is discretized to **discrete**`(0.5, 0.5)` which returns 0 with probability 0.5 and 1 with probability 0.5. The constant 0.5 on the right hand side is discretized to the constant 1 because it falls in the second range $[0.5, +\infty)$. The less-than comparison is translated to an identical less-than comparison on the discrete values.

2.2 Branching on multiple continuous variables

Now let's consider a more complex example with multiple continuous variables and conditional branches:

```
let x = uniform(0, 1) in
let y = uniform(0, 2) in
if x < 0.5 then x < 0.1 else y < 0.1
```

In this program, the variable `x` is drawn from `uniform(0, 1)` and is compared against two different constants: `0.5` and `0.1`. These two cut points partition the real line $\mathbb{R}$ into three intervals: $(-\infty, 0.1)$, $[0.1, 0.5)$, and $[0.5, +\infty)$. Similarly, the variable `y`, drawn from `uniform(0, 2)`, is compared only with `0.1`. This single cut point partitions the real line $\mathbb{R}$ into two intervals, $(-\infty, 0.1)$ and $[0.1, +\infty)$.

This analysis is done by the type inference process, which collects all comparison points for each continuous variable. For `x`, the type inference algorithm discovers the cut points `<0.1` and `<0.5`, giving it the type `float[{<0.1,<0.5}; T]`. For `y`, it only finds `<0.1`, giving it the type `float[{<0.1}; T]`. This information is captured in the annotated program below.

```
let x = uniform(0, 1) : float[{<0.1,<0.5}; T] in
let y = uniform(0, 2) : float[{<0.5}; T] in
if x < (0.1 : float[{<0.1,<0.5}; {0.1}]) then
  x < (0.5 : float[{<0.1,<0.5}; {0.5}])
else
  y < (0.5 : float[{<0.5}; {0.5}])
```

Slice replaces the continuous distribution for `x` with a discrete one that has three outcomes, where the probability of each outcome is the probability mass of the original uniform distribution within the corresponding interval. This results in `discrete(0.1, 0.4, 0.5)`. Slice replaces the continuous distribution for `y` with a discrete one that has two outcomes, where the probability of each outcome is the probability mass of the original uniform distribution within the corresponding interval. This results in `discrete(0.05, 0.95)`. The comparisons in the program are then transformed to operate on the integer indices of these new discrete intervals, leading to the following discretized program:

```
let x = discrete(0.1, 0.4, 0.5) in
let y = discrete(0.05, 0.95) in
if x < 1 then
  x < 2
else
  y < 1
```

Note that the original constant 0.5 is discretized in two different ways: for $x < 0.5$ it is discretized to 2 and for $y < 0.5$ it is discretized to 1. This is because the constant 0.5 falls in the second interval of the split $\mathbb{R} = (-\infty, 0.1) \cup [0.1, 0.5) \cup [0.5, +\infty)$ for x and the first interval for the split $\mathbb{R} = (-\infty, 0.5) \cup [0.5, +\infty)$ for y.

2.3 If else with continuous values

In the program above, we always directly compare the continuous samples to constants. However, the discretization process still works if the continuous samples and constants flow through the program, for instance via an if-else statement. Consider the following program:

```
let x = if uniform(0,1) < 0.5
```

```
        then uniform(0,2)
        else gaussian(0,1) in
let y = if 1.5 < x
        then 1.8
        else 0.3 in
x <= y
```

This program demonstrates how comparison points propagate through control flow. The type inference annotates the program as follows:

```
let x = if (uniform(0,1) : float[{<0.5}; T]) < (0.5 : float[{<0.5}; {0.5}])
        then (uniform(0,2) : float[{<=0.3,<1.5,<=1.8}; T])
        else (gaussian(0,1) : float[{<=0.3,<1.5,<=1.8}; T])
        : float[{<=0.3,<1.5,<=1.8}; T] in
let y = if (1.5 : float[{<=0.3,<1.5,<=1.8}; {1.5}]) < x
        then (1.8 : float[{<=0.3,<1.5,<=1.8}; {1.8}])
        else (0.3 : float[{<=0.3,<1.5,<=1.8}; {0.3}])
        : float[{<=0.3,<1.5,<=1.8}; {0.3,1.8}] in
x <= y
```

Notice that all comparison points from the entire program (0.3, 1.5, and 1.8) are collected for the variable `x`, regardless of which branch is taken. These comparison points with type `<=0.3,<1.5,<=1.8` split the real line into four intervals: $(-\infty, 0.3]$, $(0.3, 1.5)$, $[1.5, 1.8]$, and $(1.8, +\infty)$. Note that 1.5 is excluded from the second interval and included in the third because it comes from a strict '<' comparison. The discretized version becomes:

```
let x = if discrete(0.5, 0.5) < 1 then
    discrete(0.15, 0.6, 0.15, 0.1)
  else
    discrete(0.617911, 0.315281, 0.0308769, 0.0359303) in
let y = if 1 < x then 2 else 0 in
x <= y
```

The four probabilities in each `discrete` distribution correspond to the probability mass in each of the four intervals. For example, `uniform(0,2)` has probability 0.15 in $(-\infty, 0.3]$, 0.6 in $(0.3, 1.5]$, 0.15 in $(1.5, 1.8]$, and 0.1 in $(1.8, +\infty)$.

2.4 Discrete latent variable

Continuous distributions can have parameters that depend on discrete random choices. In this example, a Gaussian distribution's standard deviation is determined by a random choice between two values:

```
let x = if uniform(0,1) < 0.5
        then 0.5
        else 1.5 in
gaussian(0, x) < 0.5
```

This example shows how discrete choices can affect the parameters of continuous distributions. The type-annotated version reveals how the discrete latent variable `x` determines the standard deviation of the Gaussian:

```
let x = if (uniform(0,1) : float[{<0.5}; T]) < (0.5 : float[{<0.5}; {0.5}])
```

```
            then (0.5 : float[{<=0.5,<=1.5}; {0.5}])
            else (1.5 : float[{<=0.5,<=1.5}; {1.5}])
            : float[{<=0.5,<=1.5}; {0.5,1.5}] in
    (gaussian(0, x) : float[{<0.5}; T]) < (0.5 : float[{<0.5}; {0.5}])
```

The discretized program handles the distribution parameter that depends on a discrete choice:

```
let x = if discrete(0.5, 0.5) < 1 then 0 else 1 in
(if x == 0 then
    discrete(0.841345, 0.158655)
else
    discrete(0.630559, 0.369441)) < 1
```

2.5 Conditioning

Slice supports conditioning on events through the `observe` construct, allowing for Bayesian inference. The following program computes the probability that a uniform random variable is less than 0.2, given that it is less than 0.5:

```
let x = uniform(0.0, 1.0) in
observe (x < 0.5);
x < 0.2
```

The type-annotated version shows how observation points become comparison points:

```
let x = (uniform(0.0, 1.0) : float[{<0.2,<0.5}; T]) in
observe (x < (0.5 : float[{<0.2,<0.5}; {0.5}]));
x < (0.2 : float[{<0.2,<0.5}; {0.2}])
```

The discretized version preserves the conditioning semantics:

```
let x = discrete(0.2, 0.3, 0.5) in
observe (x < 2);
x < 1
```

In the discrete version, the observation `x < 2` rules out the case where `x = 2` (corresponding to the original interval $[0.5, +\infty)$), effectively conditioning on `x` being in $[0.0, 0.5)$.

2.6 Higher-order functions

Slice supports higher-order functions, enabling functional programming patterns with probabilistic computations. This example shows a higher-order function `mappair` that applies a function to both elements of a pair:

```
let mappair = fun f -> fun p -> (f (fst p), f (snd p)) in
let f = fun x -> x < 0.5 in
let g = fun x -> x < 1.5 in
let p = (uniform(0,2), gaussian(0,2)) in
let q = (mappair f) p in
let r = (mappair g) p in
if fst q then snd q else
    if fst r then snd r else fst q
```

The type system correctly infers that comparison points from both `f` and `g` (0.5 and 1.5) must be collected for the distributions in `p`, yielding type `float[{<0.5,<1.5}; T]` for both components. The discretized version preserves the higher-order structure:

```
let mappair = fun f -> fun p -> (f (fst p), f (snd p)) in
let f = fun x -> x < 1 in
let g = fun x -> x < 2 in
let p = (discrete(0.25, 0.5, 0.25),
        discrete(0.598706, 0.174666, 0.226627)) in
let q = (mappair f) p in
let r = (mappair g) p in
if fst q then snd q else
    if fst r then snd r else fst q
```

The continuous distributions are discretized into three-valued discrete distributions based on the intervals $(-\infty, 0.5)$, $[0.5, 1.5)$, and $[1.5, +\infty)$, while the functional structure remains intact.

2.7 Sequential processes with iterate

While Slice is purely functional without mutable state, it provides the `iterate` construct to model sequential probabilistic processes. This example simulates a simple weather model where tomorrow's weather depends on today's:

```
let weather_transition = fun today ->
    if today < 0.5 then
        (* If sunny today, likely stays sunny *)
        uniform(0.2, 0.4)
    else
        (* If rainy today, likely stays rainy *)
        gaussian(0.7, 0.1) in
let weather_after_3_days =
    iterate(weather_transition, uniform(0,1), 3) in
weather_after_3_days < 0.5  (* Is it sunny? *)
```

The `iterate` function applies the weather transition function three times, starting from a random initial weather state. Each iteration represents one day, with values below 0.5 representing sunny weather and values above 0.5 representing rain. Sunny days tend to stay sunny (uniform between 0.2 and 0.4), while rainy days tend to stay rainy (Gaussian centered at 0.7). The type system ensures that all comparison points are properly collected across the iterations, e.g. yielding type `float[{<0.5}; T]` for `today`, `uniform(0.2,0.4)`, `gaussian(0.7,0.1)`, and `weather_after_three_days` each. This enables accurate discretization of this sequential process:

```
let weather_transition = fun today ->
  if today < 1 then
      discrete(1, 0)
  else
      discrete(0.0227501, 0.97725) in
let weather_after_3_days =
  iterate(weather_transition, discrete(0.5, 0.5), 3) in
weather_after_3_days < 1
```

2.8 Mutable state

Slice supports mutable references, allowing imperative-style updates within a probabilistic context. This example models temperature that can be updated based on weather events:

```
let temperature = ref (uniform(15.0, 25.0)) in
let weather_event = uniform(0.0, 1.0) in
(if weather_event < 0.3 then
    (* Cold front moves in *)
    temperature := gaussian(10.0, 2.0)
else
    ());
temperature < 12.0
```

The reference `temperature` initially holds a uniform temperature between 15°C and 25°C. With 30% probability, a cold front updates it to a Gaussian-distributed temperature centered at 10°C. The type system tracks that the dereferenced value will be compared to 12.0, ensuring proper discretization of both the initial uniform distribution and the potential Gaussian update, both with float types `float[{<12}; T]`:

```
let temperature = ref (discrete(0, 1)) in
let weather_event = discrete(0.3, 0.7) in
(if weather_event < 1 then
    temperature := discrete(0.841345, 0.158655)
else
    ());
temperature < 1
```

2.9 Lists and recursion

Slice supports lists and pattern matching, enabling functional list processing with probabilistic elements. This example demonstrates closures capturing stochastic values:

```
let map = fun f -> fix loop lst :=
    match lst with
      nil -> nil
      | head :: tail -> (f head) :: (loop tail)
    end in
(* Stochastically choose threshold *)
let threshold = if uniform(0,1) < 0.5 then 0.3 else 0.7 in
(* Closure captures the stochastic threshold *)
let check_threshold = fun x -> x < threshold in
let measurements = gaussian(0.0, 1.0) ::
                   gaussian(0.5, 1.0) ::
                   gaussian(1.0, 1.0) :: nil in
map check_threshold measurements
```

The closure `check_threshold` lexically captures the stochastic `threshold` value, which is randomly chosen to be either 0.3 or 0.7. The type system correctly infers that both threshold values (0.3 and 0.7) must be included as comparison points for all Gaussian measurements, yielding type `float[{<0.3,<0.7}; T]` for list elements. This demonstrates how Slice handles higher-order functions with captured stochastic values while maintaining sound discretization.

2.10 Indian GPA Problem

This canonical example involves both discrete and continuous random variables and models a student's GPA based on nationality (India or USA) and whether they have a perfect record. The GPA is deterministic for perfect students (10.0 for India, 4.0 for USA) but drawn from a uniform distribution otherwise. The program computes the probability that a student's GPA falls below 1.0:

```
let nationality = discrete(0.5, 0.5) in
let perfect = discrete(0.01, 0.99) in
let gpa = if nationality <= 0 then
        if perfect <= 0 then
          (10.0 : float[{<1}; {10}])
        else
          (uniform(0, 10) : float[{<1}; T])
      else
        if perfect <= 0 then
          (4.0 : float[{<1}; {4}])
        else
          (uniform(0, 4) : float[{<1}; T])
      : float[{<1}; T] in
gpa < (1.0 : float[{<1}; {1.0}])
```

The type system collects the comparison point < 1 for all GPA expressions. This splits each uniform distribution into two intervals at the threshold 1.0. The discretized version becomes:

```
let nationality = discrete(0.5, 0.5) in
let perfect = discrete(0.01, 0.99) in
let gpa = if nationality <= 0 then
        if perfect <= 0 then
          1
        else
          discrete(0.1, 0.9)
      else
        if perfect <= 0 then
          1
        else
          discrete(0.25, 0.75) in
gpa < 1
```

Here, `uniform(0,10)` becomes `discrete(0.1, 0.9)` representing 10% probability for $[0, 1)$ and 90% for $[1, 10]$. Similarly, `uniform(0,4)` becomes `discrete(0.25, 0.75)`. Constants like 10.0 and 4.0 that fall in the interval $[1, +\infty)$ are mapped to index 1. The discretized program can then be processed by exact inference engines like Dice.

2.11 Partial discretization

Not all programs can be fully discretized. Consider a program where continuous variables are compared directly to each other rather than to constants:

```
let x = uniform(0, 1) in
let y = gaussian(0, 1) in
if uniform(0,1) < 0.5 then x < y else y < x
```

The type-annotated version reveals that x and y have type `float[T; T]`, indicating that no finite set of comparison points suffices:

```
let x = (uniform(0, 1) : float[T; T]) in
let y = (gaussian(0, 1) : float[T; T]) in
if (uniform(0,1) : float[{<0.5}; T]) < (0.5 : float[{<0.5}; {0.5}])
then x < y else y < x
```

Slice performs partial discretization, discretizing only the parts it can:

```
let x = uniform(0, 1) in
let y = gaussian(0, 1) in
if discrete(0.5, 0.5) < 1 then x < y else y < x
```

The branching condition gets discretized, but x and y remain continuous because they are only compared to each other, not to constants. This partially discretized program cannot be compiled to Dice but can still be executed using Monte Carlo simulation, potentially benefiting from the discrete branching structure.

3 The Slice Language and Type System

The syntax of the Slice language is shown in Figure 1.
where x ranges over variable names, c ranges over floating point constants, i ranges over integer constants, k and n are non-negative integers with $0 \leq k < n$, h and t are variables in list patterns, and $p_0, \ldots, p_n$ are probabilities that should sum to 1.

The uniform distribution $\textbf{uniform}(e_1, e_2)$ represents a continuous random value uniformly distributed in the range $[e_1, e_2)$. Other continuous distributions are defined according to their standard statistical definitions. The implementation supports over 17 distributions including Gaussian, exponential, beta, gamma, Laplace, Cauchy, and others. The discrete distribution $\textbf{discrete}(p_0, \ldots, p_n)$ represents a random value that returns integer $i \in \{0, \ldots, n\}$ with probability p_i.

Beyond basic functional constructs (pairs, projections, lambda abstractions, and function application), the language includes:

- **Finite types**: Values of the form $k\#_n$ represent finite type constants, where k is a value in the finite type with n elements.
- **Boolean operations**: Logical operators &&, ||, and not for boolean expressions.
- **Fixed-point recursion**: The fix construct enables recursive function definitions.
- **Observations**: The observe construct conditions the program on a boolean expression being true.
- **Lists**: Constructed with nil and ::, and deconstructed with pattern matching.
- **Mutable references**: Created with ref, dereferenced with !, and updated with :=.
- **Sequencing**: The semicolon operator sequences expressions with side effects.

When we discretize continuous programs, we convert expressions involving continuous distributions into expressions with discrete distributions, and convert less-than comparisons into less-than-or-equal comparisons on the corresponding discrete intervals.

3.1 Type System

We introduce a type system that analyzes both the comparison points and concrete values of floating point expressions. We have the following types:

- **`bool`**: the expression is a boolean value (true or false)
- **`int`**: the expression is an integer value

- $\mathrm{fin}(n)$: the expression is a finite type value in $\{0, 1, \ldots, n-1\}$
- $\mathbf{float}\langle B; V\rangle$: the expression is a floating point value with comparison bounds B and value set V
- $\tau_1 * \tau_2$: the expression is a pair with elements of type τ_1 and τ_2
- $\tau_1 \to \tau_2$: the expression is a function that takes an argument of type τ_1 and returns a result of type τ_2
- unit: the unit type with single value ()
- $\mathrm{list}(\tau)$: list of elements of type τ
- $\mathrm{ref}(\tau)$: mutable reference containing a value of type τ

The float type $\mathbf{float}\langle B; V\rangle$ uses a two-bag approach:

- B is the *bound bag*, containing comparison bounds of the form $< c$ or $\leq c$
- V is the *value bag*, containing concrete float values that the expression can evaluate to

Both B and V can be either a finite set or $\top$ (representing an unknown/unbounded set). This forms a product lattice structure that we describe in detail below.

3.1.1 Lattice Structure. Each bag forms a join-semilattice with the following structure:

- Elements: $\{\mathrm{Finite}(S) \mid S \text{ is a finite set}\} \cup \{\top\}$
- Ordering: $\mathrm{Finite}(S_1) \sqsubseteq \mathrm{Finite}(S_2)$ if $S_1 \subseteq S_2$, and $\mathrm{Finite}(S) \sqsubseteq \top$ for any S
- Join: $\mathrm{Finite}(S_1) \sqcup \mathrm{Finite}(S_2) = \mathrm{Finite}(S_1 \cup S_2)$, and $x \sqcup \top = \top$ for any x
- Bottom: $\mathrm{Finite}(\emptyset)$
- Top: $\top$

The float type forms a product lattice:

$$\mathrm{FloatLattice} = \mathrm{BoundLattice} \times \mathrm{ValueLattice} \tag{1}$$

$$(B_1, V_1) \sqsubseteq (B_2, V_2) \iff B_1 \sqsubseteq B_2 \wedge V_1 \sqsubseteq V_2 \tag{2}$$

$$(B_1, V_1) \sqcup (B_2, V_2) = (B_1 \sqcup B_2, V_1 \sqcup V_2) \tag{3}$$

This product lattice is also a join-semilattice with:

- Bottom element: $(\mathrm{Finite}(\emptyset), \mathrm{Finite}(\emptyset))$
- Top element: $(\top, \top)$

The lattice structure enables type inference to systematically collect and propagate information about both how values are used (bounds) and what values they can take (values).

The typing rules are as follows:

$$\textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \textsc{If}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

$$\textsc{Float}\ \frac{}{\Gamma \vdash c : \textbf{float}\langle \text{Finite}(\emptyset); \text{Finite}(\{c\})\rangle} \qquad \textsc{ContDist}\ \frac{}{\Gamma \vdash cdistr : \textbf{float}\langle \text{Finite}(\emptyset); \top\rangle}$$

$$\textsc{Discrete}\ \frac{}{\Gamma \vdash \textbf{discrete}(p_0, \ldots, p_n) : \textbf{int}} \qquad \textsc{Less}\ \frac{\Gamma \vdash e : \textbf{float}\langle B; V\rangle}{\Gamma \vdash e < c : \textbf{bool}}$$

where B is constrained to include $< c$

$$\textsc{LessEq}\ \frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash e \leq i : \textbf{bool}} \qquad \textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \textbf{fst}\ e : \tau_1} \qquad \textsc{Snd}\ \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \textbf{snd}\ e : \tau_2}$$

$$\textsc{Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\ x\ \rightarrow\ e : \tau_1 \rightarrow \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \text{true} : \textbf{bool}}$$

$$\textsc{False}\ \frac{}{\Gamma \vdash \text{false} : \textbf{bool}} \qquad \textsc{And}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1\ \&\&\ e_2 : \textbf{bool}} \qquad \textsc{Or}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1\ ||\ e_2 : \textbf{bool}}$$

$$\textsc{Not}\ \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \text{not}\ e : \textbf{bool}} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \text{unit}} \qquad \textsc{FinConst}\ \frac{0 \leq k < n}{\Gamma \vdash k\#_n : \text{fin}(n)} \qquad \textsc{FinLess}\ \frac{\Gamma \vdash e_1 : \text{fin}(n) \qquad \Gamma \vdash e_2 : \text{fin}(n)}{\Gamma \vdash e_1 <_{\#n} e_2 : \textbf{bool}}$$

$$\textsc{FinLessEq}\ \frac{\Gamma \vdash e_1 : \text{fin}(n) \qquad \Gamma \vdash e_2 : \text{fin}(n)}{\Gamma \vdash e_1 \leq_{\#n} e_2 : \textbf{bool}} \qquad \textsc{Observe}\ \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \text{observe}\ e : \text{unit}} \qquad \textsc{Seq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}$$

$$\textsc{Fix}\ \frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fix}\ f\ x := e : \tau_1 \rightarrow \tau_2} \qquad \textsc{Nil}\ \frac{}{\Gamma \vdash \text{nil} : \text{list}(\tau)} \qquad \textsc{Cons}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \text{list}(\tau)}{\Gamma \vdash e_1 :: e_2 : \text{list}(\tau)}$$

$$\textsc{Match}\ \frac{\Gamma \vdash e : \text{list}(\tau_1) \qquad \Gamma \vdash e_1 : \tau_2 \qquad \Gamma, h : \tau_1, t : \text{list}(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \text{match}\ e\ \text{with nil} \rightarrow e_1 \mid h :: t \rightarrow e_2\ \text{end} : \tau_2} \qquad \textsc{Ref}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{ref}\ e : \text{ref}(\tau)}$$

$$\textsc{Deref}\ \frac{\Gamma \vdash e : \text{ref}(\tau)}{\Gamma \vdash !e : \tau} \qquad \textsc{Assign}\ \frac{\Gamma \vdash e_1 : \text{ref}(\tau) \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}$$

4 Type Inference

Type inference for Slice aims to assign a type (e.g., **bool**, **int**, fin(n), **float**$\langle \dots \rangle$, $\tau_1 * \tau_2$, $\tau_1 \to \tau_2$, unit, list(τ), ref(τ)) to every subexpression while simultaneously collecting all relevant comparison threshold points for each float expression. This process works by traversing the abstract syntax tree (AST) of the program and generating type constraints based on the structure of the code and the typing rules.

To manage the constraints on float types **float**$\langle B; V \rangle$, we maintain two separate "bags" for each float expression:

- The *bound bag* B collects comparison bounds (e.g., < 0.5, ≤ 1.2) from all comparison sites where the expression is used
- The *value bag* V tracks concrete float values that the expression can evaluate to

Both bags start at the bottom of their respective lattices and are updated through constraint propagation during type inference.

We have two different kinds of constraints:

Type constraints: types $\tau = \tau'$ being equal.
Bag constraints: bags $B = B'$ being equal or $c \in B$ being a member of bag B.

Constraints are generated as follows:

- In an **if** e_1 **then** e_2 **else** e_3 expression, e_1 must have type **bool**, and the types inferred for e_2 and e_3 must be equal
- A comparison $e < c$ requires e to have a **float**$\langle B; V \rangle$ type. The comparison adds the bound $< c$ to the bound bag B. If c is a constant expression, its value is also tracked in its own value bag.
- A comparison $e \leq i$ requires e to have an **int** type.
- A **let** $x = e_1$ **in** e_2 expression requires the type inferred for e_1 to be used for the variable x when inferring the type of e_2.
- A continuous distribution sampling expression has type **float**$\langle B; V \rangle$ where B starts as Finite($\emptyset$) and V is $\top$ (since it can produce any value in its range).
- A **discrete**$(p_0, \dots, p_n)$ expression has type **int**, representing an integer in the range $[0, n]$ with probability distribution given by the probabilities.
- A pair (e_1, e_2) has type $\tau_1 * \tau_2$ if e_1 has type τ_1 and e_2 has type τ_2.
- For **fst** e, e must have a pair type $\tau_1 * \tau_2$, and the result has type τ_1.
- For **snd** e, e must have a pair type $\tau_1 * \tau_2$, and the result has type τ_2.
- For **fun** $x \to e$, if e has type τ_2 under the assumption that x has type τ_1, then the function has type $\tau_1 \to \tau_2$.
- For $e_1\ e_2$, e_1 must have a function type $\tau_1 \to \tau_2$, and e_2 must have type τ_1. The result has type τ_2.
- Boolean constants true and false have type **bool**.
- Boolean operations (&&, ||, not) require their operands to have type **bool** and return type **bool**.
- The unit value () has type unit.
- A finite constant $k\#_n$ has type fin(n) if $0 \leq k < n$.
- Finite comparisons $e_1 <_{\#n} e_2$ and $e_1 \leq_{\#n} e_2$ require both operands to have type fin(n) and return type **bool**.
- observe e requires e to have type **bool** and returns type unit.
- For $e_1; e_2$, the result has the type of e_2.

- For fix $f\ x := e$, if e has type τ_2 under the assumptions that f has type $\tau_1 \to \tau_2$ and x has type τ_1, then the result has type $\tau_1 \to \tau_2$.
- nil can have type list(τ) for any type τ.
- For $e_1 :: e_2$, if e_1 has type τ and e_2 has type list(τ), the result has type list(τ).
- For list pattern matching, the scrutinee must have type list(τ_1), and both branches must have the same type τ_2.
- For ref e, if e has type τ, the result has type ref(τ).
- For !e, if e has type ref(τ), the result has type τ.
- For $e_1 := e_2$, e_1 must have type ref(τ) and e_2 must have type τ. The result has type unit.

Bag constraint solving is implemented using a variant of the disjoint-set data structure, commonly known as union-find. A union-find data structure maintains a collection of disjoint sets (our bags). Each bag is represented by a tree, where the root is the canonical representative of the set. It supports three main operations:

- `find(b)`: Returns the canonical representative (root) of the bag b, containing the set of threshold points currently known for b. Path compression is used for efficiency: during the traversal from b to the root, all nodes encountered are made direct children of the root. This flattens the tree and speeds up future `find` operations.
- `union(b1, b2)`: Merges the bags containing $b1$ and $b2$. It first finds the roots of both bags. If they are different, one root is made a child of the other. Crucially, when merging bags associated with **`float`** types, the sets of threshold points stored at the roots are combined (using set union).
- `add(b, c)`: Adds a new threshold point c to the bag b.

Type constraints are solved using unification. When two types t_1 and t_2 must be unified:

- If $t_1 =$ **`bool`** and $t_2 =$ **`bool`**, unification succeeds.
- If $t_1 =$ **`int`** and $t_2 =$ **`int`**, unification succeeds.
- If $t_1 =$ unit and $t_2 =$ unit, unification succeeds.
- If $t_1 = \text{fin}(n)$ and $t_2 = \text{fin}(m)$, unification succeeds if $n = m$.
- If $t_1 =$ **`float`**$\langle B_1; V_1 \rangle$ and $t_2 =$ **`float`**$\langle B_2; V_2 \rangle$, we must unify both components:
  - Bound bags: B_1 and B_2 are unified to ensure both expressions share the same comparison context
  - Value bags: $V_1 \sqcup V_2$ is computed to combine value information
- If $t_1 = \tau_{1a} * \tau_{1b}$ and $t_2 = \tau_{2a} * \tau_{2b}$, unification succeeds if τ_{1a} unifies with τ_{2a} and τ_{1b} unifies with τ_{2b}.
- If $t_1 = \tau_{1a} \to \tau_{1b}$ and $t_2 = \tau_{2a} \to \tau_{2b}$, unification succeeds if τ_{1a} unifies with τ_{2a} and τ_{1b} unifies with τ_{2b}.
- If $t_1 = \text{list}(\tau_1)$ and $t_2 = \text{list}(\tau_2)$, unification succeeds if τ_1 unifies with τ_2.
- If $t_1 = \text{ref}(\tau_1)$ and $t_2 = \text{ref}(\tau_2)$, unification succeeds if τ_1 unifies with τ_2.
- If the types are incompatible (e.g., **`bool`** and **`float`**, **`int`** and **`float`**, pair and function), a type error occurs. Meta-variables (placeholder types) are handled by instantiation during unification.

The inference algorithm recursively walks the expression AST. It maintains an environment mapping variables to their inferred types. At each node, it generates and solves constraints using unification and the bag operations. The final result is an AST annotated with types, where each **`float`**$\langle B; V \rangle$ type carries both bags populated with the relevant bounds and values determined by the inference process.

5 Discretization

After type inference, we have a Slice expression annotated with types, where each $\texttt{float}\langle B; V\rangle$ type includes:

- A bound bag B containing all comparison bounds (e.g., < 0.5, ≤ 1.2)
- A value bag V containing known constant values or $\top$

The discretization process uses the bound bag B to determine how to partition continuous distributions. For each continuous distribution with type $\texttt{float}\langle B; V\rangle$, we extract the numeric thresholds from B to create discretization intervals.

The core idea is to map the continuous range of a float variable onto a finite set of integers, representing intervals defined by the threshold points in its bag. A comparison against a threshold constant c_k becomes a comparison against the corresponding interval index k.

Let e be a subexpression with inferred type $\texttt{float}\langle B; V\rangle$. From the bound bag B, we extract the set of numeric thresholds $\{c_0, \ldots, c_n\}$ (sorting and removing duplicates from bounds like $< c_i$ and $\leq c_i$). Let $\texttt{discretize}(e)$ be the corresponding discretized expression.

- **Continuous Distribution**: If $e = \mathit{cdistr}$ with type $\texttt{float}\langle B; V\rangle$, we extract threshold points $\{c_0, \ldots, c_n\}$ from the bound bag B. We define $n + 1$ intervals based on these points: $I_0 = (-\infty, c_0)$, $I_1 = [c_0, c_1)$, $\ldots$, $I_n = [c_{n-1}, c_n)$, $I_{n+1} = [c_n, +\infty)$. The discretization is $\texttt{discretize}(e) = \textbf{discrete}(p_0, \ldots, p_{n+1})$, where p_i is the probability mass of the original continuous distribution cdistr within the interval I_i. This is calculated using the Cumulative Distribution Function (CDF) of the specific distribution:

  $$p_i = \mathrm{CDF}(\mathrm{right}_i) - \mathrm{CDF}(\mathrm{left}_i)$$

  where left_i and right_i are the bounds of interval I_i (using $-\infty$ and $+\infty$ appropriately). This **discrete** distribution yields an integer i with probability p_i, signifying that the original continuous value fell within interval I_i. Special handling is needed for degenerate cases (e.g., zero-width intervals for uniform).
- **Discrete Distribution**: If $e = \textbf{discrete}(p_0, \ldots, p_n)$ with type **int**, the discretization simply preserves the discrete distribution as is: $\texttt{discretize}(e) = \textbf{discrete}(p_0, \ldots, p_n)$. This expression returns an integer value $i \in \{0, 1, \ldots, n\}$ with probability p_i.
- **Less Than Comparison**: If $e = e' < c_k$, where e' has type $\texttt{float}\langle B; V\rangle$. We extract the sorted threshold set $\{c_0, \ldots, c_n\}$ from the bound bag B, and c_k is the k-th smallest element in this set. The discretization is $\texttt{discretize}(e) = \texttt{discretize}(e') \leq k$. The discretized subexpression $\texttt{discretize}(e')$ evaluates to an integer i representing an interval I_i. The comparison $i \leq k$ checks if the value falls into any of the intervals $I_0, \ldots, I_k$. The union of these intervals is $(-\infty, c_k)$. Thus, the comparison correctly determines if the original value of e' was less than c_k.
- **Less Than or Equal Comparison**: If $e = e' \leq i$, where e' has type **int**, the discretization preserves the comparison: $\texttt{discretize}(e) = \texttt{discretize}(e') \leq i$. This form is used for comparing integer values, particularly the results of discrete distributions.
- **Variables, Let, If, Pairs, Projections, Functions, Application**: These constructs are translated recursively, preserving their structure:
  - $\texttt{discretize}(x) = x$
  - $\texttt{discretize}(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) = \textbf{let}\ x = \texttt{discretize}(e_1)\ \textbf{in}\ \texttt{discretize}(e_2)$
  - $\texttt{discretize}(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \textbf{if}\ \texttt{discretize}(e_1)\ \textbf{then}\ \texttt{discretize}(e_2)\ \textbf{else}\ \texttt{discreti}$
  - $\texttt{discretize}((e_1, e_2)) = (\texttt{discretize}(e_1), \texttt{discretize}(e_2))$
  - $\texttt{discretize}(\textbf{fst}\ e) = \textbf{fst}\ (\texttt{discretize}(e))$
  - $\texttt{discretize}(\textbf{snd}\ e) = \textbf{snd}\ (\texttt{discretize}(e))$

- $\texttt{discretize}(\textbf{fun}\ x\ \rightarrow\ e) = \textbf{fun}\ x\ \rightarrow\ (\texttt{discretize}(e))$
- $\texttt{discretize}(e_1\ e_2) = \texttt{discretize}(e_1)\ \texttt{discretize}(e_2)$

- **Boolean, Unit, Finite Constants**: These are already discrete and pass through unchanged.
- **Finite Comparisons**: Comparisons like $e_1 <_{\#n} e_2$ are preserved as-is since they already operate on discrete finite types.
- **Lists, References, Boolean Operations**: These constructs are translated recursively, preserving their structure.

This process effectively translates continuous distributions into discrete distributions based on the thresholds used in the program, while preserving the original discrete distributions and program structure, including functional constructs, pairs, lists, references, and finite types.

6 Soundness Proof

This section establishes the soundness of our discretization transformation. We proceed in three steps: first, we define a nondeterministic small-step semantics for the base language; second, we extend this to a probabilistic semantics that tracks distributions; and third, we prove that discretization preserves the probabilistic semantics.

6.1 Nondeterministic Small-Step Semantics

We begin by defining a small-step operational semantics for Slice that treats probabilistic sampling nondeterministically. This semantics captures the possible values that expressions can take without tracking their probabilities.

6.1.1 Values. First, we define the set of values that expressions can evaluate to:

$v ::= c$	float constant
$\mid$ true $\mid$ false	boolean value
$\mid k\#_n$	finite type value
$\mid i$	integer value
$\mid ()$	unit value
$\mid (v_1, v_2)$	pair value
$\mid \textbf{fun}\ x\ \rightarrow\ e$	function closure
$\mid$ nil	empty list
$\mid v_1 :: v_2$	list cons value
$\mid \ell$	location (for references)

6.1.2 Evaluation Contexts. We use evaluation contexts to specify the order of evaluation:

$$
\begin{aligned}
E ::= {} & [\cdot] \\
\mid {} & \textbf{let}\ x = E\ \textbf{in}\ e \\
\mid {} & E < e \mid v < E \\
\mid {} & E \le e \mid v \le E \\
\mid {} & E <_{\#n} e \mid v <_{\#n} E \\
\mid {} & E \le_{\#n} e \mid v \le_{\#n} E \\
\mid {} & E \mathbin{\&\&} e \mid v \mathbin{\&\&} E \\
\mid {} & E \mid\mid e \mid v \mid\mid E \\
\mid {} & \text{not } E \\
\mid {} & \textbf{if}\ E\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
\mid {} & (E, e) \mid (v, E) \\
\mid {} & \textbf{fst}\ E \mid \textbf{snd}\ E \\
\mid {} & E\ e \mid v\ E \\
\mid {} & \text{observe } E \\
\mid {} & E; e \\
\mid {} & E :: e \mid v :: E \\
\mid {} & \text{match } E \text{ with } \ldots \\
\mid {} & \text{ref } E \\
\mid {} & !E \\
\mid {} & E := e \mid v := E
\end{aligned}
$$

6.1.3 Small-Step Relation. We define the small-step relation $e \to \mathcal{P}(e')$ that maps an expression to a set of possible next expressions. For deterministic constructs, this set is a singleton; for probabilistic sampling, it contains all possible values in the distribution's support.

Basic Rules

$$\textsc{Let}\quad \frac{}{\textbf{let}\ x = v\ \textbf{in}\ e \to \{e[v/x]\}} \qquad \textsc{IfTrue}\quad \frac{}{\textbf{if}\ \text{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \{e_1\}}$$

$$\textsc{IfFalse}\quad \frac{}{\textbf{if}\ \text{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \{e_2\}} \qquad \textsc{Fst}\quad \frac{}{\textbf{fst}\ (v_1, v_2) \to \{v_1\}} \qquad \textsc{Snd}\quad \frac{}{\textbf{snd}\ (v_1, v_2) \to \{v_2\}}$$

$$\textsc{App}\quad \frac{}{(\textbf{fun}\ x\ \to\ e)\ v \to \{e[v/x]\}}$$

Comparison Rules

$$\textsc{LessTrue}\ \frac{c_1 < c_2}{c_1 < c_2 \longrightarrow \{\text{true}\}} \qquad \textsc{LessFalse}\ \frac{c_1 \geq c_2}{c_1 < c_2 \longrightarrow \{\text{false}\}} \qquad \textsc{LeqTrue}\ \frac{i_1 \leq i_2}{i_1 \leq i_2 \longrightarrow \{\text{true}\}} \qquad \textsc{LeqFalse}\ \frac{i_1 > i_2}{i_1 \leq i_2 \longrightarrow \{\text{false}\}}$$

Boolean Operations

$$\textsc{AndTrue}\ \frac{}{\text{true \&\& } e \longrightarrow \{e\}} \qquad \textsc{AndFalse}\ \frac{}{\text{false \&\& } e \longrightarrow \{\text{false}\}} \qquad \textsc{OrTrue}\ \frac{}{\text{true || } e \longrightarrow \{\text{true}\}} \qquad \textsc{OrFalse}\ \frac{}{\text{false || } e \longrightarrow \{e\}}$$

$$\textsc{NotTrue}\ \frac{}{\text{not true} \longrightarrow \{\text{false}\}} \qquad \textsc{NotFalse}\ \frac{}{\text{not false} \longrightarrow \{\text{true}\}}$$

List Operations

$$\textsc{MatchNil}\ \frac{}{\text{match nil with nil} \longrightarrow e_1 \mid h :: t \longrightarrow e_2 \text{ end} \longrightarrow \{e_1\}}$$

$$\textsc{MatchCons}\ \frac{}{\text{match } v_1 :: v_2 \text{ with nil} \longrightarrow e_1 \mid h :: t \longrightarrow e_2 \text{ end} \longrightarrow \{e_2[v_1/h, v_2/t]\}}$$

Probabilistic Sampling (Nondeterministic)

For continuous distributions, the small-step relation returns the set of all possible values in the distribution's support:

$$\textsc{Uniform}\ \frac{v_1, v_2 \text{ are float values} \qquad v_1 < v_2}{\textbf{uniform}(v_1, v_2) \longrightarrow \{c \mid v_1 \leq c < v_2\}} \qquad \textsc{Gaussian}\ \frac{\mu, \sigma \text{ are float values} \qquad \sigma > 0}{\textbf{gaussian}(\mu, \sigma) \longrightarrow \{c \mid c \in \mathbb{R}\}}$$

$$\textsc{Exponential}\ \frac{\lambda \text{ is a float value} \qquad \lambda > 0}{\textbf{exponential}(\lambda) \longrightarrow \{c \mid c \geq 0\}} \qquad \textsc{Beta}\ \frac{\alpha, \beta \text{ are float values} \qquad \alpha > 0 \qquad \beta > 0}{\textbf{beta}(\alpha, \beta) \longrightarrow \{c \mid 0 \leq c \leq 1\}}$$

$$\textsc{Discrete}\ \frac{p_0, \ldots, p_n \text{ are probabilities}}{\textbf{discrete}(p_0, \ldots, p_n) \longrightarrow \{0, 1, \ldots, n\}}$$

Observations

$$\textsc{ObserveTrue}\ \frac{}{\text{observe true} \longrightarrow \{()\}} \qquad \textsc{ObserveFalse}\ \frac{}{\text{observe false} \longrightarrow \emptyset}$$

Note that observing a false condition results in an empty set, representing program termination without producing a value.

References and State

For references, we extend the semantics with a store σ that maps locations to values. The relation becomes $\langle e, \sigma\rangle \to \mathcal{P}(\langle e', \sigma'\rangle)$:

$$\textsc{Ref}\ \frac{\ell \text{ is a fresh location}}{\langle \text{ref } v, \sigma\rangle \to \{\langle \ell, \sigma[\ell \mapsto v]\rangle\}} \qquad \textsc{Deref}\ \frac{\sigma(\ell) = v}{\langle !\ell, \sigma\rangle \to \{\langle v, \sigma\rangle\}} \qquad \textsc{Assign}\ \frac{}{\langle \ell := v, \sigma\rangle \to \{\langle (), \sigma[\ell \mapsto v]\rangle\}}$$

Context Rule

$$\textsc{Context}\ \frac{e \to S}{E[e] \to \{E[e'] \mid e' \in S\}}$$

This nondeterministic semantics captures all possible execution paths of a probabilistic program without tracking the probability of each path. In the next subsection, we will extend this to a probabilistic semantics that properly tracks distributions.

7 Implementation

We have implemented Slice as an OCaml-based compiler that transforms continuous probabilistic programs into discrete ones, which can then be analyzed using exact inference engines. The implementation consists of two main components: the Slice compiler (`cdice`) and the Dice inference engine [10].

7.1 Architecture Overview

Figure 3 shows the overall architecture of the Slice system.

7.2 Implementation Details

Parser and Lexer The frontend uses OCaml's Menhir parser generator to parse `.cdice` source files. The parser (`parser.mly`) defines the concrete syntax and produces an abstract syntax tree (AST) represented by the `expr` type. The lexer (`lexer.mll`) handles tokenization, including special tokens for finite type comparisons (e.g., `<#n`).

Type Inference The type inference module (`inference.ml`) implements the two-bag type system described in Section 4. It uses a constraint-based approach with unification to infer types while collecting comparison bounds. The implementation features:

- A union-find data structure for managing bags efficiently
- Listener patterns for propagating constraints between related expressions
- Subtyping rules that ensure proper information flow

Bags Module The `bags.ml` module provides the lattice-based data structures for tracking bounds and values. It implements:

- Generic bag operations (union, equality checking)
- Specialized `BoundBag` for comparison bounds ($< c, \leq c$)

- `FloatBag` for tracking concrete float values
- Efficient representation using either finite sets or $\top$

Discretization The discretization module (`discretization.ml`) transforms typed continuous programs into discrete ones. For each continuous distribution, it:

(1) Extracts comparison thresholds from the bound bag
(2) Computes interval probabilities using distribution CDFs
(3) Generates equivalent discrete distributions
(4) Transforms comparisons to operate on interval indices

Code Generation The `to_dice.ml` module generates Dice source code from the discretized AST. It handles the translation of Slice constructs to their Dice equivalents, including special handling for finite types that become integers in Dice.

Runtime and Evaluation The implementation includes:

- An interpreter (`interp.ml`) for sampling-based execution
- Statistical testing to verify discretization correctness
- Support for multiple output formats (Dice, SPPL)

7.3 Integration with Dice

The Dice inference engine [10] provides exact inference for discrete probabilistic programs using Binary Decision Diagrams (BDDs) and weighted model counting. Our discretized programs are designed to work seamlessly with Dice's capabilities:

- Discrete distributions map directly to Dice's `discrete` construct
- Finite types become bounded integers
- Observations translate to Dice's conditioning primitives

The complete workflow is automated through the `run_contdice.sh` script, which pipes the output of Slice directly to Dice for inference.

7.4 Engineering Considerations

The implementation prioritizes correctness and clarity:

- Extensive use of OCaml's type system for safety
- Modular design allowing independent testing of components
- Clear separation between syntax-directed and semantic phases
- Comprehensive test suite including adversarial examples

The Slice compiler consists of approximately 3,400 lines of OCaml code (including the lexer, parser, type inference engine, discretization engine, and code generator), demonstrating that powerful program transformations can be implemented concisely.

The system is open-source and available at https://github.com/[repository-url], along with benchmarks and documentation.

8 Evaluation

8.1 Synthetic benchmarks

- Describe the benchmarks, the structure they have - Scaling graphs - Divide cdice and Dice
  * We compare all benchmarks that sppl runs on

8.2 Psi benchmarks

- describe benchmarks, cite paper - simplications we made such as expanding out arrays - graphs (cdice + dice vs sppl vs bitblast)

8.3 Fairness benchmarks

- describe benchmarks, cite paper - simplications we made such as expanding out arrays - graphs (cdice + dice vs sppl vs bitblast)

9 Related Work

Discrete-only probabilistic languages Several recent systems achieve *exact inference* by restricting models to finite, discrete random variables. *Roulette* extends Racket with first-class support for finitely-supported distributions and leverages symbolic evaluation to compile queries into weighted model-counting problems, enabling scalable exact conditioning on complex programs [12]. *Dice* follows a similar philosophy in an OCaml DSL, compiling discrete programs to weighted model counting to perform exact Bayesian updates even on thousands of Boolean and finite-categorical variables [10]. Earlier work in probabilistic logic programming, most prominently *ProbLog*, annotates Prolog facts with probabilities and reduces inference to weighted Boolean formulas that can likewise be solved exactly [4]. These languages demonstrate the power of exact reasoning, but by construction they *cannot express continuous random quantities*; consequently they offer no built-in path for analysing models that naturally mix real-valued and discrete structure.

Symbolic treatment of mixed models *SPPL* occupies an intermediate point on the design spectrum. By enforcing syntactic restrictions that guarantee every program can be translated into a finite *sum-product expression*, SPPL supports models with both discrete and continuous components while still admitting *symbolic* exact inference [14]. Rather than discretising the continuous parts, SPPL derives closed-form integrals when possible; exactness is retained for a narrow class of models. There is no mechanism for turning the remaining continuous structure into a discrete program that could reuse the powerful inference back-ends of Dice, Roulette, or ProbLog.

Continuous and hybrid PPLs with approximate inference The majority of widely-used PPLs treat *continuous* distributions as first-class and rely on approximate inference. Systems such as *Stan* (C++), *PyMC* (Python), *Pyro* (Python on PyTorch), *TensorFlow Probability* and its precursor *Edward* expose rich continuous distributions and obtain posteriors with Hamiltonian Monte Carlo, variational inference, or sequential Monte Carlo [1, 2, 5, 15, 18]. These frameworks deliver high accuracy for differentiable densities but can struggle with discrete latent structure or discontinuities that arise after naïve discretisation. Crucially, none of them offers an automated pathway to convert continuous sub-expressions into discrete replacements that would make exact inference feasible.

Universal languages supporting both regimes Universal or "hybrid" languages aim for expressiveness by embedding probabilistic primitives into general-purpose hosts. *Anglican* (Clojure), *WebPPL* (JavaScript), *Figaro* (Scala) and *Infer.NET* (C#) permit arbitrary mixtures of discrete and continuous variables with inference based on importance sampling, Gibbs, expectation propagation, or lightweight MCMC [8, 11, 13, 17]. More recently, Julia-based systems such as *Turing.jl* and *Gen.jl* expose programmable inference interfaces, while *Bean Machine* offers a declarative syntax atop PyTorch [3, 6, 16]. Classic languages like *Church* pioneered the "evaluation-as-sampling" view that underlies many of these systems [7]. Discretization transformations are left entirely to the user and are not integrated with the inference engine.

Probabilistic program abstraction The work of [9] is perhaps most similar to our own in its aim to simplify probabilistic programs for easier inference. They use ideas from abstract interpretation to transform a program with continuous and discrete random variables into a finite Boolean program based on user-provided predicates. This *predicate abstraction* is proven to be *distributionally sound*, meaning that for any query expressed in the predicate vocabulary, the abstract program gives the same probability as the concrete one. Their method works by taking a set of predicates and automatically adding *completion* predicates to create a tight abstraction. Inference then proceeds by querying the abstract Boolean program and using a concrete inference engine for sub-queries. While this approach is powerful, it has several limitations. The selection of predicates is left to the user, and a poor choice can lead to a coarse abstraction or a combinatorial explosion in the number of completion predicates. Furthermore, the full automation of their technique is limited to loop-free programs. Our work, in contrast, does not require user-provided predicates. Instead, Slice automatically infers the necessary "cut points" from the program text using a novel type system, and our approach supports features like recursion.

Positioning of our work Our contribution lies precisely at the intersection left open by the above lines of research. We propose a *language-level transformation* that automatically discretises continuous sub-programs while preserving the semantics required for exact Bayesian reasoning in the discrete fragment. When discretization successfully translates all continuous distributions, we can translate the program into the finite domain accepted by Dice (or Roulette, or ProbLog), and inherit their fast exact inference without sacrificing the modelling convenience of continuous distributions. Empirically, our approach of discretizing programs and running them through Dice's inference engine achieves significantly faster runtimes compared to SPPL's symbolic methods, while maintaining exactness. Unlike continuous PPLs that settle for stochastic approximations, our approach bridges the gap between expressive modelling and fast exact computation. Partial discretization is also possible, and could be used to speed up inference without sacrificing expressiveness.

In summary, existing discrete PPLs excel at exactness but lack continuous support, symbolic languages like SPPL require restrictive structure and are slower than model counting approaches, and continuous or hybrid frameworks favour approximate inference. To our knowledge none of these systems automates the conversion of continuous programs into a discrete representation amenable to exact inference; our work addresses precisely that missing piece.

Future work In the future, we would like to extend our work to symbolically integrate tractable combinations of continous distributions beyond the cumulative distribution function. We would also like to improve mixed continuous-discrete inference by partial discretization, summing out the discrete variables using weighted model counting techniques, and running the remaining continuous inference using standard continuous inference techniques.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *JMLR* (2019).

[2] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *J. Stat. Softw.* (2017). doi:10.18637/jss.v076.i01

[3] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *PLDI*. doi:10.1145/3314221.3314642

[4] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *IJCAI*.

[5] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alexander A. Alemi, Matthew D. Hoffman, and Rif A. Saurous. 2017. TensorFlow Distributions. In *SysML*.

[6] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *AISTATS*.

[7] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *UAI*.

[8] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Online textbook demonstrating WebPPL.

[9] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2018. Sound Abstraction and Decomposition of Probabilistic Programs. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 1999–2008. https://proceedings.mlr.press/v80/holtzen18a.html

[10] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *PACMPL* OOPSLA (2020). doi:10.1145/3428208

[11] Tom Minka, John M. Winn, John Guiver, Yordan Zaykov, David Fabian, and Jim Bronskill. 2018. Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[12] Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. 2025. Roulette: A Language for Expressive, Exact, and Efficient Discrete Probabilistic Programming. In *PLDI*. doi:10.1145/3729334

[13] Avi Pfeffer. 2009. Figaro: An Object-Oriented Probabilistic Programming Language. In *CRA Tech Report*.

[14] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *PLDI*. doi:10.1145/3453483.3454078

[15] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic Programming in Python Using PyMC3. *PeerJ CS* (2016). doi:10.7717/peerj-cs.55

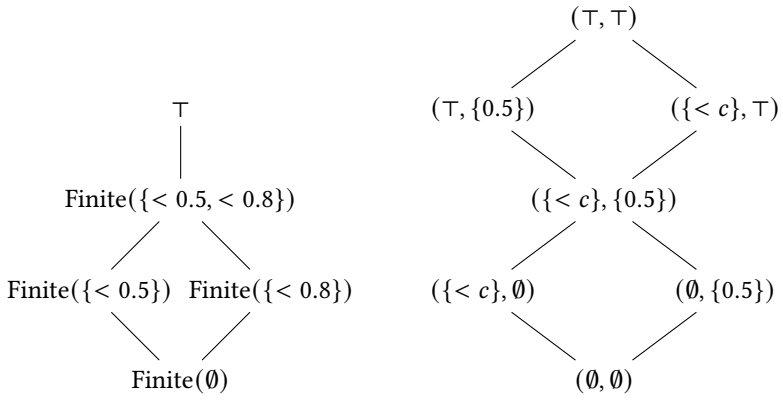
[16] Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal D. Shah, David Noursi, Michael Tingley, Narjes Torabi, Eric Lippert, and Erik Meijer. 2020. Bean Machine: A Declarative Probabilistic Programming Language for Efficient Programmable Inference. In *PGM*.

[17] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. *CoRR* abs/1608.05263 (2016). http://arxiv.org/abs/1608.05263

[18] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A Library for Probabilistic Modeling, Inference, and Criticism. *arXiv preprint arXiv:1610.09787* (2016).

$e ::= x$	variable
$\mid c$	float constant
$\mid$ true $\mid$ false	boolean constant
$\mid k\#_n$	finite type constant (k of type fin(n))
$\mid ()$	unit
$\mid$ **let** $x = e_1$ **in** e_2	let binding
$\mid$ cdistr	continuous distribution
$\mid$ **discrete**$(p_0, \ldots, p_n)$	discrete distribution
$\mid e_1 < e_2$	less-than (floats)
$\mid e_1 \leq e_2$	less-or-equal (integers)
$\mid e_1 <_{\#n} e_2$	less-than (finite type n)
$\mid e_1 \leq_{\#n} e_2$	less-or-equal (finite type n)
$\mid e_1$ && e_2	logical and
$\mid e_1 \mid\mid e_2$	logical or
$\mid$ not e	logical not
$\mid$ **if** e_1 **then** e_2 **else** e_3	conditional
$\mid (e_1, e_2)$	pair construction
$\mid$ **fst** e	first projection
$\mid$ **snd** e	second projection
$\mid$ **fun** $x \rightarrow e$	function abstraction
$\mid e_1\ e_2$	function application
$\mid$ fix $f\ x := e$	fixed-point recursion
$\mid$ observe e	observation/conditioning
$\mid e_1; e_2$	sequencing
$\mid$ nil	empty list
$\mid e_1 :: e_2$	list cons
$\mid$ match e with nil $\rightarrow e_1 \mid h :: t \rightarrow e_2$ end	list match
$\mid$ ref e	reference creation
$\mid\ !e$	dereference
$\mid e_1 := e_2$	assignment
cdistr ::= **uniform**(e_1, e_2)	uniform distribution
$\mid$ **gaussian**(e_1, e_2)	gaussian distribution
$\mid$ **exponential**(e)	exponential distribution
$\mid$ **beta**(e_1, e_2)	beta distribution
$\mid \ldots$	(and 13+ more distributions)

Fig. 1. Syntax of the Slice language.



(a) Bound Lattice Example (b) Product Lattice Example

Fig. 2. Lattice structure for float types. (a) shows an example bound lattice with comparison bounds. (b) shows the product lattice combining bounds and values.

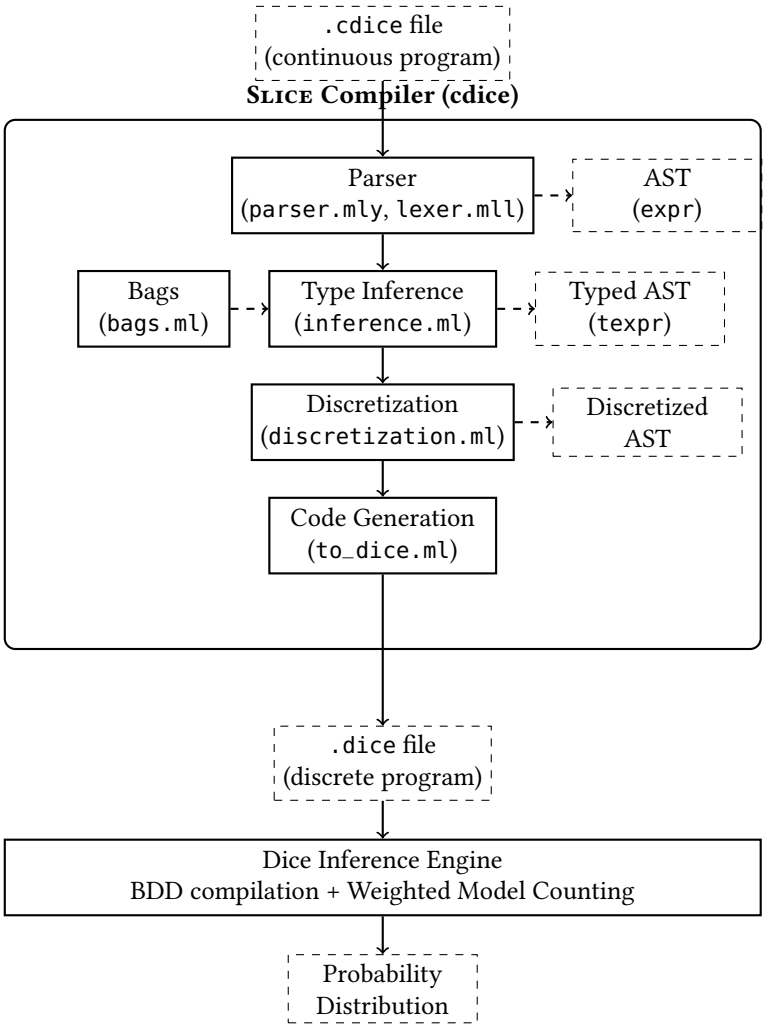


Fig. 3. Architecture of the SLICE system. The SLICE compiler transforms continuous probabilistic programs into discrete ones through type-directed discretization. The resulting discrete program is then processed by the Dice inference engine to compute exact probability distributions.