

Type-Directed Discretization of Probabilistic Programs

KATHERINE WU, Cornell University, USA

JULES JACOBS, ETH Zurich, Switzerland and Jane Street, USA

KEVIN BATZ, University of Münster, Germany

ALEXANDRA SILVA, Cornell University, USA

We study *exact discretization* as a semantics-preserving transformation for recursive, higher-order probabilistic programs with continuous distributions. We target programs where continuous values are compared against finitely many constants, so exact inference reduces to a discrete problem. Our central technical contribution is a non-local, type-directed analysis that infers where continuous values can be partitioned into finitely many observationally relevant regions, then rewrites sampling and comparison behavior over those regions. We call this transformation SLICE. Because this construction is global and type-directed, correctness requires reasoning beyond the local syntax: we formalize the transformation and prove soundness for boolean queries using a coupling-style logical relations argument over operational semantics. As an application, transformed programs can be executed by discrete engines such as DICE, ROULETTE, and STORM. Our empirical evaluation shows two complementary strengths of SLICE when paired with discrete backends: it enables *exact inference* for challenging continuous programs that lie beyond the reach of previous exact systems, and, on benchmarks where direct comparison is possible, it is competitive with state-of-the-art exact inference systems for continuous programs.

CCS Concepts: • **Mathematics of computing** → **Probabilistic representations; Probabilistic inference problems**; • **Theory of computation** → **Operational semantics**.

Additional Key Words and Phrases: probabilistic programming, exact inference, discretization, type systems

ACM Reference Format:

Katherine Wu, Jules Jacobs, Kevin Batz, and Alexandra Silva. 2026. Type-Directed Discretization of Probabilistic Programs. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 402 (October 2026), 140 pages. <https://doi.org/10.1145/3839534>

1 Introduction

Probabilistic programming languages (PPLs) are widely used to model uncertainty [5, 9, 13, 14, 16, 20, 23, 24, 26, 34, 35, 37, 39–41, 45–47]. Programs with *continuous* distributions capture rich models but require reasoning about uncountable domains, whereas purely *discrete* programs often admit fast and precise inference but cannot express real-valued models.

SLICE studies *exact discretization*: a semantics-preserving source-to-source transformation that converts, if possible, continuous programs into discrete ones. We target higher-order recursive programs with conditioning and continuous sampling, without arithmetic. The key mechanism is *type-directed cut inference*: our type system tracks each constant threshold (or *cut*) used in comparisons and uses this information to partition continuous domains into finitely many intervals. For example, $\text{uniform}(0, 1) < 0.5$ discretizes to $\text{discrete}(0.5, 0.5) < 1$, yielding a discrete program that existing exact inference tools can process.

Authors' Contact Information: Katherine Wu, Cornell University, Ithaca, USA, kaw324@cornell.edu; Jules Jacobs, ETH Zurich, Zurich, Switzerland and Jane Street, New York, USA, julesjacobs@gmail.com; Kevin Batz, University of Münster, Münster, Germany, kevin.batz@uni-muenster.de; Alexandra Silva, Cornell University, Ithaca, USA, alexandra.silva@cornell.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/10-ART402

<https://doi.org/10.1145/3839534>

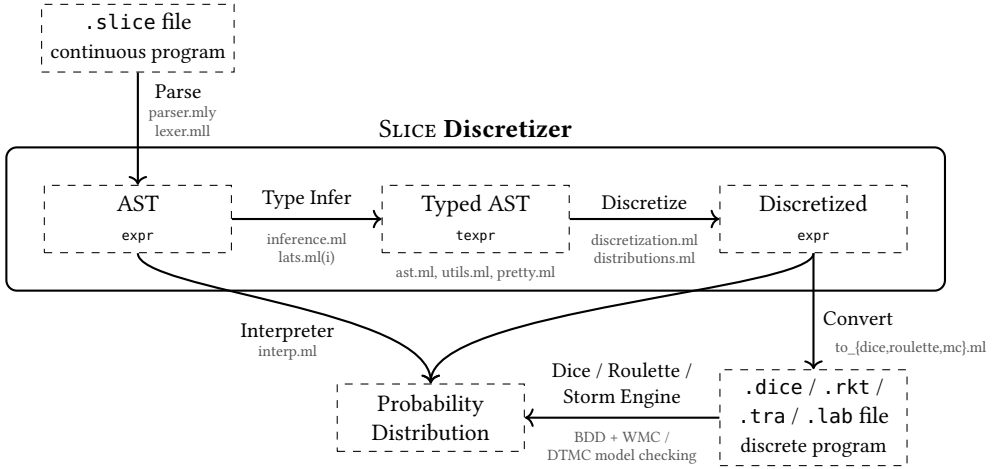


Fig. 1. Pipeline for exact discretization: typed SLICE programs are transformed into semantically equivalent discrete programs, then evaluated by discrete engines. The transformation stage (type inference + discretization) is backend-agnostic; DICE, ROULETTE, and STORM are examples of target backends.

Cuts may arise far from their source expressions: a sampled value may flow through higher-order functions, conditionals, recursive calls, data structures, and observations before it is eventually compared. To discretize soundly, SLICE propagates cut information along these non-local flows, collecting all observationally relevant cut points for each continuous value and using them to partition its domain into finitely many intervals (see §2 for illustrative examples). A type system provides a natural framework for this analysis: it gives a compositional account of how cut information propagates through each language construct (via one typing rule per language feature); it separates the analysis result, i.e. a type derivation, from the type inference algorithm used to infer it; and it supports our correctness argument, since semantic preservation can be stated as a logical relation indexed by our types carrying cut and value sets.

Our discretization method is subject to a principled restriction: a continuous value must ultimately be compared against constants; direct comparisons between two unknown reals such as `uniform(0, 1) < uniform(0, 1)` cannot be discretized exactly and remain continuous. Nonetheless, benchmarks from the literature and our novel case studies show that many programs *can* be discretized by our system automatically.

Discretization itself is not new. Garg et al. [19] encode real-valued variables via user-chosen bit-precisions, trading accuracy for tractability. Other approximate discretization schemes exist in the literature [4, 11, 27]. In SLICE, *exact discretization* means that the transformation is semantics-preserving: the discrete program preserves the relevant probabilities of the original continuous program. This differs from approximate discretization, which approximates the original semantics and may provide error bounds or convergence guarantees, but does not prove equality of source and transformed program probabilities in the sense of our main theorem.

Figure 1 shows the concrete pipeline from typed source programs to discrete backends. Because discretization is source-to-source, any discrete PPL that supports finite distributions can serve as a backend; we instantiate this with DICE [26], ROULETTE [35], and STORM [15].

Outline. Starting from examples (§2) and concluding with related and future work in §8, we:

Type System (§3) Characterize when programs are discretizable.

Type Inference (§4) Infer types automatically to guide discretization.

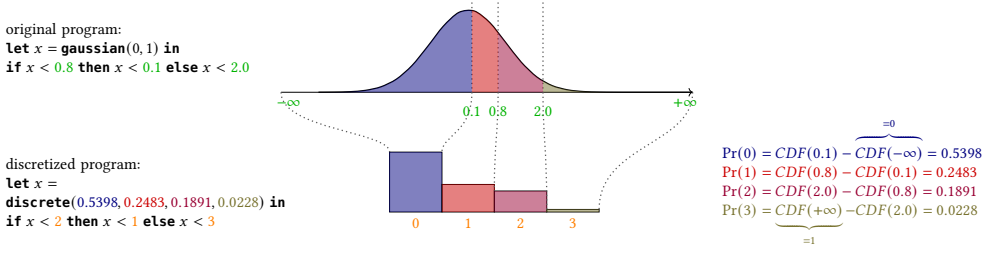


Fig. 2. Discretization of the original program into its semantically equivalent discretized program for §2.1. The shaded regions show how the cuts at 0.1, 0.8, and 2.0 partition the continuous distribution into four intervals $(-\infty, 0.1)$, $[0.1, 0.8)$, $[0.8, 2.0)$, and $[2.0, +\infty)$, whose masses become the probabilities of outcomes 0, 1, 2, and 3, i.e., interval indices, in the discrete distribution.

Exact Discretization (§5) Define a type-directed transformation from continuous programs to discrete ones using inferred type information.

Soundness (§6) Prove semantic preservation of the discretization transformation with a coupling-style relational argument [6, 12, 48].

Implementation, Case Studies, and Evaluation (§7) Implement SLICE type inference and the discretization transformation in OCaml, and provide DICE, ROULETTE, and STORM backends for running the discretized programs. We provide (i) case studies combining continuous sampling, higher-order functions, and unbounded recursion; to the best of our knowledge, there is no prior exact inference system that can cover programs *combining* all three of these features and (ii) case studies where SLICE+DICE and SLICE+ROULETTE show favorable scaling and empirical performance on new and existing benchmarks [21, 40].

2 Slice by Example

To build intuition for SLICE, we start with simple examples demonstrating the key ideas, and then provide a more involved example containing higher-order recursion and lists in §2.6. Our pipeline performs *fully automatic exact inference* on all provided programs.

2.1 Branching on a Single Continuous Variable

Consider a program that samples a real value from a standard Gaussian distribution and then performs two nested comparisons: first checking whether the sample is less than 0.8, then comparing it to either 0.1 or 2.0:

```
let x = gaussian(0,1) in if x < 0.8 then x < 0.1 else x < 2.0
```

We discretize this program by replacing the real-valued sample with a discrete interval index, as shown below and in Figure 2:

```
let x = discrete(0.5398, 0.2483, 0.1891, 0.0228) in if x < 2 then x < 1 else x < 3
```

Here, `discrete(0.5398, 0.2483, 0.1891, 0.0228)` already represents all four observationally distinguishable outcomes of the Gaussian sample. The precise real value of x never matters to the rest of the program; only the interval in which it falls does. This is the regime we target: continuous quantities observed only through finitely many comparison outcomes. *Slice’s goal is to detect such situations automatically.* We partition the continuous distribution into intervals that are indistinguishable to the rest of the program, then map every real to the interval it inhabits.

Our type system guides the transformation, in particular through a *cut set* — the set of comparison thresholds extracted from the program. A continuous distribution is annotated with its cut set, which records the values it is compared against. For our example, the type annotation is:

```
let x = (gaussian(0,1) : float[{<0.1,<0.8,<2}]) in if x < 0.8 then x < 0.1 else x < 2.0
```

Intuitively, the cuts $\{<0.1, <0.8, <2.0\}$ partition the real line $\mathbb{R}$ into four intervals $(-\infty, 0.1)$, $[0.1, 0.8)$, $[0.8, 2.0)$, $[2.0, +\infty)$, and only the interval containing the Gaussian sample x affects the rest of the computation. SLICE uses these cuts to replace the continuous distribution with a discrete one whose outcomes correspond exactly to these intervals, and each floating-point constant is mapped to the index of the interval that contains it.

2.2 Branching on Multiple Continuous Variables

Different continuous quantities may induce different cut sets, and SLICE computes the appropriate set for each one independently. Consider an example with multiple continuous variables:

```
let x = uniform(0, 1) in
let y = uniform(0, 2) in
if x < 0.5 then x < 0.2 else y < 0.1
```

In this program, the variable x is drawn from `uniform(0, 1)` and compared against two constants: 0.5 and 0.2. These cuts <0.5 and <0.2 partition the real line $\mathbb{R}$ into three intervals: $(-\infty, 0.2)$, $[0.2, 0.5)$, and $[0.5, +\infty)$. Similarly, the variable y , drawn from `uniform(0, 2)`, is compared only with 0.1. This single cut <0.1 partitions the real line $\mathbb{R}$ into $(-\infty, 0.1)$ and $[0.1, +\infty)$.

Type inference performs this analysis by collecting all cuts for each continuous variable. For x , it discovers the cuts <0.2 and <0.5 , giving it the type `float[{<0.2,<0.5}]`. For y , it only finds <0.1 , giving it the type `float[{<0.1}]`. This information is captured in the program below (left):

(a) Type-Annotated Program

```
let x = uniform(0, 1) : float[{<0.2,<0.5}] in
let y = uniform(0, 2) : float[{<0.1}] in
if x < 0.5 then x < 0.2 else y < 0.1
```

(b) Discretized Program

```
let x = discrete(0.2, 0.3, 0.5) in
let y = discrete(0.05, 0.95) in
if x < 2 then x < 1 else y < 1
```

SLICE therefore discretizes x into three outcomes `discrete(0.2, 0.3, 0.5)` and y into two outcomes `discrete(0.05, 0.95)`, where each probability in the discrete distribution is the measure of the corresponding original interval. Comparisons then operate over the integer indices of those intervals. The constants 0.5 and 0.2 map to indices 2 and 1, respectively, in the partition $(-\infty, 0.2) \cup [0.2, 0.5) \cup [0.5, +\infty)$ for x ; the constant 0.1 maps to index 1 in the partition $(-\infty, 0.1) \cup [0.1, +\infty)$ for y .

2.3 Control Flow

We now consider comparisons against a continuous quantity whose value is determined by control flow. Additionally, continuous values can flow through conditionals:

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

Here, y evaluates to either 1.8 or 0.3 depending on x . Because $x \leq y$ may use either value, SLICE collects both constants and treats the comparison as ranging over this finite set. This motivates enriching our float types with a *value set*: a finite collection of concrete values that an expression

float[cut-set ; value-set]
 set of cuts (e.g., $\{<0.1, <0.5, \leq 0.8\}$) or $\top$ (uncut) set of concrete values (e.g., $\{0.1, 0.4, 0.5\}$) or $\top$ (any value)

Fig. 3. The type **float**[cut-set; value-set] captures two aspects of a floating-point expression. The cut set is determined either by (i) the constants the expression is compared against or (ii) the constant itself if the expression is a float constant. The value set records the concrete values an expression can take. The discretization is driven by the cut set: the n cuts define $n + 1$ intervals over the real line.

may evaluate to. Figure 3 presents the *full* float type implemented by SLICE; each floating-point expression is annotated with such a type. The type-annotated program is below (left):

(a) Type-Annotated Program

```
let x =
  (if flip() 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
```

(b) Discretized Program

```
let x =
  if flip() then
    discrete(0.15, 0.6, 0.15, 0.1)
  else
    discrete(0.617911, 0.315281,
             0.030877, 0.035930)
in
let y =
  if 2 > x then 2 else 0
in
x <= y
```

Value sets refine an expression's cut set by adding cuts induced by comparisons. For example, in the test $x \leq y$, the value set of y contributes boundary points that must also appear in the cut set of x . This dependency is non-local: constants introduced in one branch can constrain how a sampled value from another branch must be discretized.

Whenever two expressions are compared, the type system equates their cut sets so they are discretized over the same intervals. These cut sets then propagate backward through the control flow to all contributing subexpressions. The extended version [51, Figure 18] illustrates this non-local propagation, which the soundness proof must relate through a semantic coupling relation.

The discretized program is shown on the right. The cuts ≤ 0.3 , < 1.5 , ≤ 1.8 induce the intervals $(-\infty, 0.3]$, $(0.3, 1.5)$, $[1.5, 1.8]$, $(1.8, +\infty)$. Each **discrete** statement lists the mass of these intervals rounded to six decimal places.

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 such a random choice:

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

The type annotations show how the variable x determines the Gaussian's standard deviation, and the extended version [51, Figure 19] illustrates how type information is propagated.

(a) Type-Annotated Program

```

let x =
  (if flip() 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}]))

```

(b) Discretized Program

```

let x =
  if flip() then 0 else 1
in
(if x == 0 then
  discrete(0.84, 0.16)
else
  discrete(0.63, 0.37)) < 1

```

Because x is used as a parameter to the **gaussian** distribution, SLICE generates a separate case for each possible value that x can take (namely 0.5 and 1.5). To support this, the type system uses non-strict inequalities in cut sets, allowing these values to map to unique integer indices starting at 0. Specifically, 0.5 maps to index 0 via `float[{<=0.5,<=1.5}; {0.5}]`, and 1.5 maps to index 1 via `float[{<=0.5,<=1.5}; {1.5}]`. The resulting discretized program then branches on these values and instantiates the corresponding discrete distribution, depending on which standard deviation was selected by the initial flip, as shown in the program on the right.

2.5 Conditioning

SLICE supports conditioning via **observe**. Consider the following:

(a) Type-Annotated Program

```

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

```

(b) Discretized Program

```

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

```

In the discretized program, the observation $x < 2$ eliminates the discrete state corresponding to the original interval $[0.5, +\infty)$, thereby preserving the effect of conditioning on $x < 0.5$.

Exact equalities are encoded with **observe** ($x \leq c \ \&\& \ x \geq c$). The two cuts $<c$ and $\leq c$ induce the partition $(-\infty, c)$, $\{c\}$, (c, ∞) , so SLICE can track probability mass assigned exactly to c as its own discrete outcome. This supports point observations when the observed value has strictly positive probability mass; the extended version [51, § A.3] gives such an example.

Measure-zero observations remain delicate [28]. What matters for **observe**(e) is whether e has positive probability, not whether it is written as an equality. When an observation has probability zero, SLICE records observation-failure mass instead of renormalizing (cf. §6); if all observation-satisfying executions have probability zero, the result is the empty subdistribution. Scoring semantics for true measure-zero observations are out of scope, so users should approximate them with narrow intervals when that is the intended model. [51, § A.4] gives details and examples.

2.6 Unbounded Recursion, Higher-Order Functions, and Lists

We now turn to the more involved example in Figure 4, which SLICE discretizes automatically. This example emphasizes how our discretization procedure operates across higher-order functions, unbounded recursion, and lists.

```

(* Returns true if y is in [lo,hi] *)      1
let within = fun lo -> fun hi -> fun y -> 2
  (y >= lo) && (y < hi) in                3

(* Function for sampling interval bounds *) 4
let b = fun _ ->                          5
  discrete(0.18: 0.05, 0.11: 0.24,        6
    0.21: 0.37, 0.09: 0.58,              7
    0.41: 0.89) in                        8

(* Initialize intervals by sampling their   9
respective (possibly overlapping) bounds *) 10
let init =                                11
  (false, within 0.00 (b ())) ::          12
  (false, within (b ()) (b ())) ::        13
  (false, within (b ()) (b ())) ::        14
  (false, within (b ()) 1.00) ::          15
  nil in                                  16

(* Update flags depending on y falling     17
within the respective interval *)           18
let update = fix update y := fun ys ->    19
  match ys with                            20
  | nil -> nil                             21
  | h :: t ->                             22
    ((fst h || ((snd h) y)), snd h)        23
    :: (update y t)                       24
end in                                    25

(* Sample from a gaussian, then use       26
the resulting value to update flags *)     27
let step = fun ys ->                      28
  update (gaussian(0,1)) ys in            29

(* Number of samples determined by        30
unbounded recursion *)                    31
let cover = fix cover st :=               32
  if uniform(0,1) < 0.1 then st           33
  else cover (step st) in                 34

(* Have all intervals been covered at     35
least once? *)                            36
let is_covered = fix is_covered ys :=     37
  match ys with                           38
  | nil -> true                           39
  | h :: t -> (fst h) && (is_covered t)    40
end in                                    41

(* Run entire process on sampled intervals *) 42
let final = cover init in                 43
is_covered final                          44

```

Fig. 4. An infinite stochastic process for random interval covering over $[0, 1]$.

The program models the following process: We start by randomly sampling 4 intervals (l. 6 and 13), stored as a list of pairs, where the first component is a flag and the second component is a function that determines whether a given point falls within that interval. We then employ unbounded recursion (l. 36) to repeat the following a potentially unbounded number of times according to a geometric distribution: sample a point y from a gaussian (l. 31), and update the flag of every interval to true whenever y falls within that interval (l. 22). The program returns true iff *all* intervals have been hit at least once (l. 50 and 42). Exact inference on this program therefore determines the probability that this infinite stochastic process hits every interval at least once.

SLICE automatically produces a *provably equivalent* (cf. §6) discrete program (shown in [51, § A.5]). Even though this program still contains unbounded recursion, our back-end automatically determines that its stochastic execution behavior can be modeled as a *finite-state* Markov chain. Our key insight is that exact inference can then be performed via *probabilistic model checking*: we use STORM [15], which automatically computes that the sought-after probability is 0.034474.... See §7.1 for details. To the best of our knowledge, our pipeline is the first capable of performing exact inference on suitable programs containing continuous sampling, higher-order functions, and unbounded recursion. In §7.1.2, we extend this program by conditioning *inside the recursion*.

3 The Slice Language and Type System

We now describe the language (§3.1), its types (§3.2), and the typing rules for discretization (§3.3). Henceforth, we use the terms *expression*, *subexpression*, and *program* somewhat interchangeably when the intended meaning is clear from context.

$e ::= x \mid c \mid \text{true} \mid \text{false} \mid k_{\#n} \mid ()$	constants
$\mid \text{let } x = e_1 \text{ in } e_2$	let binding
$\mid \text{uniform}(e_1, e_2) \mid \text{gaussian}(e_1, e_2) \mid \text{exponential}(e) \mid \dots$	continuous distributions
$\mid \text{discrete}(p_0, \dots, p_{n-1})$	discrete distribution
$\mid e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 <_{\#n} e_2 \mid e_1 \leq_{\#n} e_2$	comparisons
$\mid e_1 \&\& e_2 \mid e_1 \parallel e_2 \mid \text{not } e$	Boolean operations
$\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$	conditional
$\mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e$	pairs
$\mid \text{fun } x \rightarrow e \mid e_1 \ e_2 \mid \text{fix } f \ x := e$	functions
$\mid \text{observe } e \mid e_1; e_2$	effects
$\mid \text{nil} \mid e_1 :: e_2 \mid (\text{match } e \text{ with } \mid \text{nil} \rightarrow e_{\text{nil}} \mid e_1 :: e_2 \rightarrow e_{\text{cons}} \text{ end})$	lists
$\mid \text{diverge}$	diverge

Fig. 5. Syntax of the SLICE language. SLICE supports a variety of continuous distributions with one and two parameters; for a full list of supported distributions, see the extended version [51, § C].

3.1 The Slice Language

The syntax in Figure 5 extends a standard higher-order functional core with probabilistic choices. Beyond constants, variables, comparisons, Boolean operators, conditionals, pairs, and lists, SLICE offers first-class functions, recursion, and observations. The construct **observe**(e) conditions on e holding. Finite-domain constants $k_{\#n}$ range over $\{0_{\#n}, \dots, (n-1)_{\#n}\}$ as discussed below. The expression **discrete**($p_0, \dots, p_{n-1}$) requires a nonempty list of probabilities: $n \geq 1$, every $p_i \geq 0$, and $\sum_i p_i = 1$. It returns $k_{\#n}$ with probability p_k . For random sampling, we include the usual discrete distributions and any continuous distribution with a tractable cumulative distribution function (one- or two-argument variants such as exponential, laplace, uniform, gaussian, beta, gamma). Adding further distributions is straightforward as long as their CDFs are known; the extended version [51, § C] lists the ones implemented. For brevity, we include only $<$ and $\leq$ (and resp. $<_{\#n}$ and $\leq_{\#n}$) in the language, but we desugar $>$ and $\geq$ (and their finite variants $>_{\#n}$ and $\geq_{\#n}$) by swapping their operands, as well as $==_{\#n}$ by binding each operand once and testing both $\leq_{\#n}$ directions.

3.2 Types

We formalize the types that SLICE supports. A *value* set V is a finite set of reals or $\top$. Elements of $\{<x \mid x \in \mathbb{R}\} \cup \{\leq x \mid x \in \mathbb{R}\}$ are called *cuts*. Cut sets induce intervals partitioning the reals as follows: Given a non-empty cut set $B = \{\sim_1 x_1, \dots, \sim_n x_n\}$ with $x_1 \leq \dots \leq x_n$, ordering $<c$ before $\leq c$ when both occur, we define the set $\text{Intervals}(B)$ of intervals it induces as expected, e.g., $\text{Intervals}(\{<0.1, \leq 0.5\}) = \{(-\infty, 0.1), [0.1, 0.5], (0.5, \infty)\}$. The strict and non-strict cuts differ only in which side contains the boundary point: $\text{Intervals}(\{<c\}) = \{(-\infty, c), [c, \infty)\}$, whereas $\text{Intervals}(\{\leq c\}) = \{(-\infty, c], (c, \infty)\}$. Thus a boundary generated by $e < c$ places c on the right, while a boundary generated by $e \leq c$ places c on the left. Moreover, we let $\text{Intervals}(\emptyset) = \{(-\infty, \infty)\}$ and $\text{Intervals}(\top) = 2^{\mathbb{R}}$. We denote by $\text{Intervals}(B)_i$ the i -th interval of $\text{Intervals}(B)$, assuming $\text{Intervals}(B)$ is 0-indexed, i.e., $i \in \{0, \dots, n\}$. We denote the set of all cut- and value sets, respectively, by $\mathcal{B}$ and $\mathcal{V}$.

Definition 3.1 (Types). For $n \in \mathbb{N}$, $B \in \mathcal{B}$, and $V \in \mathcal{V}$, types τ adhere to the grammar

$$\tau ::= \text{int} \mid \text{bool} \mid \text{unit} \mid \text{fin}(n) \mid \text{float}[B; V] \mid \tau_1 * \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \text{list}(\tau).$$

$$\begin{array}{c}
\text{FLOAT-CONST} \\
\frac{\text{has}(V, c)}{\Gamma \vdash c : \mathbf{float}[B; V]} \\
\\
\text{CONTINUOUS-1} \\
\frac{\Gamma \vdash e : \mathbf{float}[B_1; V_1] \quad \text{recovers}(B_1, V_1) \quad B_1 \equiv_{\top} B}{\Gamma \vdash \mathbf{exponential}(e) : \mathbf{float}[B; \top]} \\
\\
\text{CONTINUOUS-2} \\
\frac{\Gamma \vdash e_1 : \mathbf{float}[B_1; V_1] \quad \text{recovers}(B_1, V_1) \quad B_1 \equiv_{\top} B \quad \Gamma \vdash e_2 : \mathbf{float}[B_2; V_2] \quad \text{recovers}(B_2, V_2) \quad B_2 \equiv_{\top} B}{\Gamma \vdash \mathbf{uniform}(e_1, e_2) : \mathbf{float}[B; \top]} \\
\\
\text{LESS} \\
\frac{\Gamma \vdash e_1 : \mathbf{float}[B; V_1] \quad \Gamma \vdash e_2 : \mathbf{float}[B; V_2] \quad \text{answers}_{<}(B, V_1, V_2)}{\Gamma \vdash e_1 < e_2 : \mathbf{bool}} \\
\\
\text{LESS-EQUAL} \\
\frac{\Gamma \vdash e_1 : \mathbf{float}[B; V_1] \quad \Gamma \vdash e_2 : \mathbf{float}[B; V_2] \quad \text{answers}_{\leq}(B, V_1, V_2)}{\Gamma \vdash e_1 \leq e_2 : \mathbf{bool}}
\end{array}$$

Fig. 6. Typing rules for float types and continuous distributions (**uniform** is a representative for all other two-argument distributions). The notions *recovers* and *answers* are defined in Definitions 3.2 to 3.4. Here Γ is a typing environment of the form $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$, where $\Gamma, x : \tau_1 = (\Gamma \setminus \{x : \tau \mid \tau \text{ type}\}) \cup \{x : \tau_1\}$.

Standard types cover integers, booleans, ordered pairs, functions, and lists. Finite types **fin**(n) contain $\{0_{\#n}, \dots, (n-1)_{\#n}\}$; the $\#n$ -subscript reminds inference backends such as DICE that these values live in a finite domain. The float type **float**[$B; V$] annotates a real-valued expression with its cut set B (an over-approximation of the comparisons it participates in) and value set V (an over-approximation of the values it can take). $B = \top$ means the expression may be compared to arbitrary reals; $B = \emptyset$ means it will never be compared; $V = \top$ indicates an unrestricted result.

3.3 Typing Rules

The typing rules are split into three parts: Rules specific to float types and continuous distributions (Figure 6), standard rules for all other language constructs (Figure 7), and subtyping (Figure 8).

To understand the float typing rules, it is helpful to understand how discretization works. A type **float**[$B; V$] is discretized as follows: If $B \neq \top$ is a finite set of cuts, then the type is discretized to a type **fin**($n+1$) where $n+1 = |\text{Intervals}(B)|$ is the number of intervals induced by B . For instance, if $B = \{<0.5, \leq 0.8\}$, then the type is discretized to **fin**(3), where $0_{\#3}$ represents the interval $(-\infty, 0.5)$, $1_{\#3}$ represents the interval $[0.5, 0.8]$, and $2_{\#3}$ represents the interval $(0.8, \infty)$. If $B = \top$, then no discretization is performed, and the type remains **float**.

To understand the typing rules, keep in mind that the cut sets B need to be sufficiently fine-grained — i.e., contain enough cuts — so that the discretized program has enough information to determine how the original continuous program behaved.

Constants. Rule FLOAT-CONST assigns a constant c the type **float**[$B; V$] whenever $\text{has}(V, c)$ holds (i.e., $c \in V$ or $V = \top$). Constants do not constrain the cut set, but recording them in V ensures later comparisons know which values must remain distinguishable.

Continuous Distributions. Rule CONTINUOUS-1 types a single-argument distribution when its argument e has type **float**[$B; V$] and $\text{recovers}(B, V)$: We write $B_1 \equiv_{\top} B_2$ iff $B_1 = \top \iff B_2 = \top$.

Definition 3.2. Let $B \in \mathcal{B}$ and $V \in \mathcal{V}$. We say that $\text{recovers}(B, V)$ if $B = \top$ or ($V \neq \top$ and every interval I induced by B contains at most one value in V , i.e., $\forall I \in \text{Intervals}(B) : |V \cap I| \leq 1$).

Intuitively, $\text{recovers}(B, V)$ means that the discretization induced by B is lossless on V : from the resulting interval index, the original value in V can be uniquely recovered. For example, **exponential**(e) can be discretized only when e takes finitely many recoverable values. Thus V

must be finite and B must separate those values so the discretized program can determine which one occurred. §5 revisits this invariant. The rule for two-argument distributions is analogous.

Comparisons. For $e < c$ with $e : \mathbf{float}[B_1; \top]$ and $c : \mathbf{float}[B_2; \{c\}]$, the discretized e must still decide the predicate, so B_1 must include $<c$. If c can take any value in V_2 , every $<v$ for $v \in V_2$ must appear. Symmetrically, for $c < e$ with $e : \mathbf{float}[B_2; V_2]$, B_2 must contain $\leq c$ so the discretized intervals know on which side of c they fall. For example, $2.3 < e$ requires cuts separating $(-\infty, 2.3]$ from $(2.3, \infty)$; finer partitions are allowed but not required.

In general, the typing rule for $e_1 < e_2$ requires that the expressions share a cut set B , and that their respective value sets V_1, V_2 are related in a way that allows the comparison to be determined after discretization, via the $\text{answers}_{<}(B, V_1, V_2)$ relation:

Definition 3.3. Let $B \in \mathcal{B}$ and $V_1, V_2 \in \mathcal{V}$. We say that $\text{answers}_{<}(B, V_1, V_2)$ if (1) $B = \top$, or (2) $B \neq \top$ and $V_2 \neq \top$ and $\forall x \in V_2, <x \in B$, or (3) $B \neq \top$ and $V_1 \neq \top$ and $\forall x \in V_1, \leq x \in B$.

To understand this definition, we first want to introduce two different perspectives on the B set. The first perspective is that B specifies a set of intervals, such that values of the expression will be mapped to their respective interval index. The second perspective is that this discretized index *still contains enough information to evaluate all comparisons in B when viewed as predicates*. That is, if B contains $<x$ then the discretized value of $e : \mathbf{float}[B; V]$ still contains enough information to answer $e < x$ precisely: if after discretization, e has an interval index below the interval containing x , then the comparison is true, and otherwise it is false.

Now, if $e_1 : \mathbf{float}[B; V_1]$ and $e_2 : \mathbf{float}[B; V_2]$, then $\text{answers}_{<}(B, V_1, V_2)$ ensures that the comparison result can still be determined after discretization because:

- (1) If $B = \top$, then no discretization is performed, and the comparison can just be performed on the real-valued expression.
- (2) If $B \neq \top$ and $V_2 \neq \top$, then the expression is discretized into intervals as determined by B , but in this case the right-hand side can only take on values in V_2 . In this case, all $<x$ for $x \in V_2$ must be contained in B , which means that the value of e_2 can be exactly recovered from its interval index. Then, given the previous observation, we can determine the comparison result by comparing the interval index of e_1 to the interval index of e_2 .
- (3) If $B \neq \top$ and $V_1 \neq \top$, then the situation is similar to the previous case.

If $V_1 = V_2 = \top$, then $\text{answers}_{<}(B, V_1, V_2)$ (and analogously $\text{answers}_{\leq}(B, V_1, V_2)$) requires $B = \top$, so neither comparand is discretized. Additionally, we insist that $B_1 = B_2$, which ensures that both comparands will be discretized to the same type, or not discretized at all.

Finally, for non-strict comparisons, the rule is analogous to Definition 3.3:

Definition 3.4. Let $B \in \mathcal{B}$ and $V_1, V_2 \in \mathcal{V}$. We say that $\text{answers}_{\leq}(B, V_1, V_2)$ if (1) $B = \top$, or (2) $B \neq \top$ and $V_2 \neq \top$ and $\forall x \in V_2, \leq x \in B$, or (3) $B \neq \top$ and $V_1 \neq \top$ and $\forall x \in V_1, <x \in B$.

Subtyping. The type system includes a subtyping relation that captures when one type can be replaced by another. The subtyping rules are shown in Figure 8. For float types $\mathbf{float}[B; V]$, subtyping follows the lattice ordering on V : an expression known to range over some set of values can also be considered to range over a superset of those values. For B , we require equality to make sure that the discretized representation of the expression is not changed by subtyping.

All other typing rules are standard and depicted in Figure 7 and in [51, § D], respectively.

4 Type Inference

We now describe our type inference algorithm, whose goal is to annotate each subexpression of a program with a sound type. Specifically, we focus on expressions of type $\mathbf{float}[B; V]$ and show

VAR $\frac{}{\Gamma, x : \tau \vdash x : \tau}$	LET $\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$	IF $\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$
DISCRETE		
OBSERVE $\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \text{observe } e : \mathbf{unit}}$	$n \geq 1 \quad \forall i \in \{0, \dots, n-1\}. p_i \geq 0 \quad \sum_{i=0}^{n-1} p_i = 1$ $\frac{}{\Gamma \vdash \text{discrete}(p_0, \dots, p_{n-1}) : \mathbf{fin}(n)}$	DIVERGE $\frac{}{\Gamma \vdash \text{diverge} : \tau}$
FINCONST $\frac{0 \leq k < n}{\Gamma \vdash k_{\#n} : \mathbf{fin}(n)}$	FINLESS $\frac{\Gamma \vdash e_1 : \mathbf{fin}(n) \quad \Gamma \vdash e_2 : \mathbf{fin}(n)}{\Gamma \vdash e_1 <_{\#n} e_2 : \mathbf{bool}}$	FINLESSEQ $\frac{\Gamma \vdash e_1 : \mathbf{fin}(n) \quad \Gamma \vdash e_2 : \mathbf{fin}(n)}{\Gamma \vdash e_1 \leq_{\#n} e_2 : \mathbf{bool}}$

Fig. 7. Typing rules for general language constructs

SUB-REFL $\frac{\tau \in \{\mathbf{int}, \mathbf{bool}, \mathbf{fin}(n), \mathbf{unit}\}}{\tau <: \tau}$	SUB-FLOAT $\frac{B_1 = B_2 \quad V_1 \sqsubseteq V_2}{\mathbf{float}[B_1; V_1] <: \mathbf{float}[B_2; V_2]}$	SUB-PAIR $\frac{\tau_1 <: \tau'_1 \quad \tau_2 <: \tau'_2}{\tau_1 * \tau_2 <: \tau'_1 * \tau'_2}$
SUB-LIST $\frac{\tau <: \tau'}{\mathbf{list}(\tau) <: \mathbf{list}(\tau')}$	SUB-ARROW $\frac{\tau'_1 <: \tau_1 \quad \tau_2 <: \tau'_2}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}$	SUB-TYPE $\frac{\tau_1 <: \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}$

Fig. 8. Subtyping relation. For both cut- and value sets, we let $d \sqsubseteq d'$ iff $\begin{cases} d \subseteq d' & \text{if } d \neq \top \text{ and } d' \neq \top \\ d' = \top & \text{otherwise.} \end{cases}$

how SLICE populates the cut set B and value set V , which later determine how and whether the expression can be discretized. Type inference for all other types is standard and follows the classic Hindley-Milner algorithm. For float-typed expressions, we use a two-phase approach: the first phase traverses the program once to generate constraints over cut and value sets; the second phase solves those constraints to produce a fully annotated program.

The distinction between typing and type inference is important. The correctness argument cleanly separates into two pieces: (1) type inference produces a typing derivation matching our typing rules, and (2) given such a typing derivation, discretization is semantics preserving. Thus the proof of discretization does not have to reason directly about the type inference procedure.

4.1 Constraint Generation

We use the constraint language in Figure 9 to express requirements over the type structure and float refinements. We adopt the framework of [38], introducing a new judgment of the form $\Gamma \vdash e : \tau \mid C$, which reads as *in the environment Γ , e has type τ and produces the set of constraints C* . In this judgment, Γ and e are inputs to the type inference algorithm, while τ and C are outputs: the algorithm infers the type τ for expression e and generates the constraints C that must hold for this typing to be valid. The inference rules formalize which constraints must be generated and later solved for a given expression e to have inferred type τ . Figure 10 gives the constraint generation rules for SLICE. We focus on a small but representative subset of constructs in SLICE, capturing the key ideas behind constraint generation for the *full* language.

The type inference rules closely mirror the typing rules in Figure 6, with two key differences. First, the inference rules generate a constraint set instead of having those constraints appear in the premises. Second, subtyping is handled in a syntax-directed way: rather than using a general subtyping rule, we bake the subtyping relation into each rule.

$$C ::= \tau_1 <: \tau_2 \mid \text{recovers}(B, V) \mid \text{answers}_{<}(B, V_1, V_2) \mid \text{answers}_{\leq}(B, V_1, V_2) \mid \text{has}(V, c) \\ \mid B_1 \sqsubseteq B_2 \mid B_1 = B_2 \mid V_1 \sqsubseteq V_2 \mid V_1 = V_2 \mid B_1 \equiv_{\top} B_2 \mid C_1, C_2$$

Fig. 9. Syntax for our constraint language. Constraints correspond to the semantic requirements in our type system: subtyping ensures type compatibility, $\text{recovers}(B, V)$ ensures that distribution parameters can be recovered after discretization, $\text{answers}_{<}(B, V_1, V_2)$ and $\text{answers}_{\leq}(B, V_1, V_2)$ ensure that comparison results can be determined after discretization, and $\text{has}(V, c)$ ensures that constants are properly tracked. $B_1 \equiv_{\top} B_2$ holds exactly when $B_1 = \top \iff B_2 = \top$.

$$\begin{array}{c} \text{TFloat-CONST} \\ \hline \Gamma \vdash c : \mathbf{float}[B; V] \mid \text{has}(V, c) \\ \\ \text{TIF} \\ \hline \frac{\Gamma \vdash e_1 : \mathbf{bool} \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \Gamma \vdash e_3 : \tau_3 \mid C_3}{\Gamma \vdash \mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 : \tau_4 \mid C_1, C_2, C_3, \tau_2 <: \tau_4, \tau_3 <: \tau_4} \\ \\ \text{TVar} \\ \hline \Gamma, x : \tau \vdash x : \tau \mid \emptyset \\ \\ \text{TLESS} \\ \hline \frac{\Gamma \vdash e_1 : \mathbf{float}[B_1; V_1] \mid C_1 \quad \Gamma \vdash e_2 : \mathbf{float}[B_2; V_2] \mid C_2}{\Gamma \vdash e_1 < e_2 : \mathbf{bool} \mid C_1, C_2, B_1 = B_2, \text{answers}_{<}(B_1, V_1, V_2)} \\ \\ \text{TContinuous-1} \\ \hline \frac{\Gamma \vdash e : \mathbf{float}[B_1; V_1] \mid C}{\Gamma \vdash \mathbf{exponential}(e) : \mathbf{float}[B; V] \mid C, \text{recovers}(B_1, V_1), B_1 \equiv_{\top} B, V = \top} \\ \\ \text{TContinuous-2} \\ \hline \frac{\Gamma \vdash e_1 : \mathbf{float}[B_1; V_1] \mid C_1 \quad \Gamma \vdash e_2 : \mathbf{float}[B_2; V_2] \mid C_2}{\Gamma \vdash \mathbf{uniform}(e_1, e_2) : \mathbf{float}[B; V] \mid C_1, C_2, \text{recovers}(B_1, V_1), B_1 \equiv_{\top} B, \text{recovers}(B_2, V_2), B_2 \equiv_{\top} B, V = \top} \\ \\ \text{TLESSEQ} \\ \hline \frac{\Gamma \vdash e_1 : \mathbf{float}[B_1; V_1] \mid C_1 \quad \Gamma \vdash e_2 : \mathbf{float}[B_2; V_2] \mid C_2}{\Gamma \vdash e_1 \leq e_2 : \mathbf{bool} \mid C_1, C_2, B_1 = B_2, \text{answers}_{\leq}(B_1, V_1, V_2)} \\ \\ \text{TLET} \\ \hline \frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash \mathbf{let } x = e_1 \mathbf{ in } e_2 : \tau_2 \mid C_1, C_2} \end{array}$$

Fig. 10. Constraint generation rules for type inference for float-typed expressions.

Example 4.1. To make the constraint generation process concrete, Figure 11 walks through §2.3. Each subexpression is annotated with type variables for the cut sets (B_i) and value sets (V_i), and the constraints produced at each step are shown explicitly.

4.2 Constraint Solving

Once constraint generation completes, we obtain a system of constraints over type variables, cut sets B , and value sets V . The constraint-solving phase proceeds in two steps:

- (1) **Subtyping reduction:** We first process all subtyping constraints of the form $\tau_1 <: \tau_2$. Since subtyping for float types follows the rule $\mathbf{float}[B_1; V_1] <: \mathbf{float}[B_2; V_2]$ if and only if $B_1 = B_2$ and $V_1 \sqsubseteq V_2$, we reduce each subtyping constraint to corresponding equality and subset constraints on the cut and value sets. Subtyping elimination continues until only constraints among B 's and V 's remain.
- (2) **Lattice constraint solving:** After subtyping reduction, we are left with constraints purely over cut sets and value sets: equalities, subset relations, and the specialized constraints $\text{recovers}(B, V)$, $\text{answers}_{<}(B, V_1, V_2)$, $\text{answers}_{\leq}(B, V_1, V_2)$, and $\text{has}(V, c)$. These constraints are solved using a monotone fixed-point algorithm over the cut and value lattices.

Subtyping Reduction. Subtyping reduction eliminates all remaining type-level subtyping constraints by recursively decomposing them according to type structure. For base float types, a constraint $\mathbf{float}[B_1; V_1] <: \mathbf{float}[B_2; V_2]$ is reduced to the conjunction $B_1 = B_2$ and $V_1 \sqsubseteq V_2$.

Program	Constraints Emitted
let $x = \text{if flip}()$	
then $\text{uniform}(0 : \text{float}[B_1; V_1],$	+ $\text{has}(V_1, 0)$
$2 : \text{float}[B_2; V_2])$	+ $\text{has}(V_2, 2)$
$: \text{float}[B_3; V_3]$	+ $\text{recovers}(B_1, V_1), B_1 \sqsubseteq_{\top} B_3, \text{recovers}(B_2, V_2), B_2 \sqsubseteq_{\top} B_3, V_3 = \top$
else $\text{gaussian}(0 : \text{float}[B_4; V_4],$	+ $\text{has}(V_4, 0)$
$1 : \text{float}[B_5; V_5])$	+ $\text{has}(V_5, 1)$
$: \text{float}[B_6; V_6]$	+ $\text{recovers}(B_4, V_4), B_4 \sqsubseteq_{\top} B_6, \text{recovers}(B_5, V_5), B_5 \sqsubseteq_{\top} B_6, V_6 = \top$
$: \text{float}[B_7; V_7]$	+ $\text{float}[B_3; V_3] <: \text{float}[B_7; V_7], \text{float}[B_6; V_6] <: \text{float}[B_7; V_7]$
in	
let $y = \text{if } x : \text{float}[B_7; V_7]$	+ $\emptyset$
$<$	
$1.5 : \text{float}[B_8; V_8]$	+ $\text{has}(V_8, 1.5)$
$: \text{bool}$	+ $\text{answers}_{<}(B_7, V_7, V_8), B_7 = B_8$
then $1.8 : \text{float}[B_9; V_9]$	+ $\text{has}(V_9, 1.8)$
else $0.3 : \text{float}[B_{10}; V_{10}]$	+ $\text{has}(V_{10}, 0.3)$
$: \text{float}[B_{11}; V_{11}]$	+ $\text{float}[B_9; V_9] <: \text{float}[B_{11}; V_{11}], \text{float}[B_{10}; V_{10}] <: \text{float}[B_{11}; V_{11}]$
in	
$x : \text{float}[B_7; V_7]$	+ $\emptyset$
$<=$	
$y : \text{float}[B_{11}; V_{11}]$	+ $\emptyset$
$: \text{bool}$	+ $\text{answers}_{\leq}(B_7, V_7, V_{11}), B_7 = B_{11}$

Fig. 11. Program from §2.3 (left) and its emitted constraints from each line (right) from **TFLOAT-CONST**, **TCONTINUOUS-2**, **TIF**, **TVAR**, and **TLESS** and **TLESSEq** following the constraint generation rules in Figure 10. Lines annotated with $\emptyset$ emit no constraints.

For compound types, subtyping is handled structurally before reaching floats. For example, a constraint $\text{float}[B_1; V_1] * \text{float}[B_2; V_2] <: \text{float}[B_5; V_5] * \text{float}[B_6; V_6]$ is decomposed into $\text{float}[B_1; V_1] <: \text{float}[B_5; V_5]$ and $\text{float}[B_2; V_2] <: \text{float}[B_6; V_6]$, each of which is then reduced to equality and subset constraints on the corresponding cut and value sets. This process is syntactic and continues until no subtyping constraints remain, leaving only constraints over B and V .

Lattice Solving. The lattice solver keeps track of the current contents of all B and V variables. Initially, all B and V variables are set to $\emptyset$. During the solving process, the variables may go up in the lattice to a finite set, or to $\top$. Each of the constraints is handled as follows:

- **Value containment** ($\text{has}(V, c)$): We update V to include the constant c , i.e., $V := V \sqcup \{c\}$. If V is already $\top$, the constraint is trivially satisfied. ①
- **Top assignment** ($V = \top$): We immediately set V to $\top$, representing that the value set contains all possible float values. ②
- **Subset constraints** ($B_1 \sqsubseteq B_2$ or $V_1 \sqsubseteq V_2$): We ensure that the contents of the first variable are always a subset of the second. When B_1 or V_1 is updated, we propagate the changes to B_2 or V_2 by taking the least upper bound. ③
- **Recoverability** ($\text{recovers}(B, V)$): This constraint ensures that the cut set B can distinguish all values in V . If $B = \top$, the constraint is already satisfied. If $V = \top$ and $B \neq \top$, we set $B := \top$, since no finite cut set recovers an unrestricted real value. Otherwise V is finite; we use a heuristic of inserting $\leq x$ for each value $x \in V$. These cuts place distinct elements of V in distinct intervals and therefore satisfy recoverability. This choice can add redundant cuts, but remains sound. ④
- **Less-than answerability** ($\text{answers}_{<}(B, V_1, V_2)$): Intuitively, this constraint ensures that B contains enough cuts to answer all possible comparisons of the form “ $v_1 < v_2$ ” where $v_1 \in V_1$ and $v_2 \in V_2$. If $B = \top$, the constraint is trivially satisfied. Otherwise, we again have a heuristic choice:

Subtyping Constraint	After Subtyping Reduction
$\text{float}[B_3; V_3] <: \text{float}[B_7; V_7]$	$B_3 = B_7, V_3 \sqsubseteq V_7$
$\text{float}[B_6; V_6] <: \text{float}[B_7; V_7]$	$B_6 = B_7, V_6 \sqsubseteq V_7$
$\text{float}[B_9; V_9] <: \text{float}[B_{11}; V_{11}]$	$B_9 = B_{11}, V_9 \sqsubseteq V_{11}$
$\text{float}[B_{10}; V_{10}] <: \text{float}[B_{11}; V_{11}]$	$B_{10} = B_{11}, V_{10} \sqsubseteq V_{11}$

(a) *Subtyping reduction*. Each subtyping constraint over float types is decomposed into an equality constraint on cut sets and a subset constraint on value sets.

Constraints Processed	Solver Updates	Cut Sets
	$\forall i: B_i = \emptyset, V_i = \emptyset$	
① $\left\{ \begin{array}{l} \text{has}(V_1, 0) \\ \text{has}(V_2, 2) \\ \text{has}(V_4, 0) \\ \text{has}(V_5, 1) \\ \text{has}(V_8, 1.5) \\ \text{has}(V_9, 1.8) \\ \text{has}(V_{10}, 0.3) \end{array} \right.$	$V_1 := V_1 \sqcup \{0\} = \{0\}$ $V_2 := V_2 \sqcup \{2\} = \{2\}$ $V_4 := V_4 \sqcup \{0\} = \{0\}$ $V_5 := V_5 \sqcup \{1\} = \{1\}$ $V_8 := V_8 \sqcup \{1.5\} = \{1.5\}$ $V_9 := V_9 \sqcup \{1.8\} = \{1.8\}$ $V_{10} := V_{10} \sqcup \{0.3\} = \{0.3\}$	$B_1 = \emptyset$ $B_2 = \emptyset$ $B_4 = \emptyset$ $B_5 = \emptyset$ $B_3 = B_6$ $= B_7 = B_8$ $= B_9 = B_{10}$ $= B_{11}$ $= \{<1.5, \leq 0.3, \leq 1.8\}$
② $\left\{ \begin{array}{l} V_3 = \top \\ V_6 = \top \end{array} \right.$	$V_3 := \top$ $V_6 := \top$	
③ $\left\{ \begin{array}{l} V_3 \sqsubseteq V_7 \\ V_6 \sqsubseteq V_7 \\ V_9 \sqsubseteq V_{11} \\ V_{10} \sqsubseteq V_{11} \end{array} \right.$	$V_7 := V_7 \sqcup \top = \top$ $\emptyset$ $V_{11} := V_{11} \sqcup \{1.8\} = \{1.8\}$ $V_{11} := V_{11} \sqcup \{0.3\} = \{0.3, 1.8\}$	Value Sets $V_1 = \{0\}$ $V_2 = \{2\}$ $V_3 = \top$ $V_4 = \{0\}$ $V_5 = \{1\}$ $V_6 = \top$ $V_7 = \top$ $V_8 = \{1.5\}$ $V_9 = \{1.8\}$ $V_{10} = \{0.3\}$ $V_{11} = \{0.3, 1.8\}$
④ $\left\{ \begin{array}{l} \text{recovers}(B_1, V_1) \\ \text{recovers}(B_2, V_2) \\ \text{recovers}(B_4, V_4) \\ \text{recovers}(B_5, V_5) \end{array} \right.$	$\emptyset$ $\emptyset$ $\emptyset$ $\emptyset$	
⑤ $\left\{ \begin{array}{l} \text{answers}_{<}(B_7, V_7, V_8) \\ \text{answers}_{\leq}(B_7, V_7, V_{11}) \end{array} \right.$	$B_7 := B_7 \sqcup \{<1.5\} = \{<1.5\}$ $B_7 := B_7 \sqcup \{\leq 0.3, \leq 1.8\} = \{<1.5, \leq 0.3, \leq 1.8\}$	
⑥ $\left\{ \begin{array}{l} B_7 = B_8, B_7 = B_{11} \\ B_3 = B_7, B_6 = B_7 \\ B_9 = B_{10}, B_{10} = B_{11} \end{array} \right.$	$B_7 = B_8, B_7 = B_{11}$ $B_3 = B_7, B_6 = B_7$ $B_9 = B_{11}, B_{10} = B_{11}$	
⑦ $\left\{ \begin{array}{l} B_1 \equiv_{\top} B_3, B_2 \equiv_{\top} B_3 \\ B_4 \equiv_{\top} B_6, B_5 \equiv_{\top} B_6 \end{array} \right.$	$\emptyset$ $\emptyset$	(c) Final cut sets and value sets obtained.

(b) *Lattice solving*. Each pass performs monotone updates to B_i or V_i . Lines annotated with $\emptyset$ denote constraint is trivially satisfied.

Fig. 12. Following Figure 11, the constraint solving process for §2.3.

we add a cut $<v$ for each value $v \in V_2$ (if $V_2 \neq \top$) and $\leq v$ for each value $v \in V_1$ (if $V_1 \neq \top$). If V_1 and V_2 are both $\top$, we set B to $\top$. ⑤

- **Less-equal answerability** ($\text{answers}_{\leq}(B, V_1, V_2)$): Analogous to less-than. ⑤
- **Equality constraints** ($B_1 = B_2$ or $V_1 = V_2$): We unify the variables, treating them as the same variable throughout the solving process. Any update to one variable is immediately reflected in all variables unified with it. ⑥
- **Top consistency** ($B_1 \equiv_{\top} B_2$): If either variable becomes $\top$, set both to $\top$. ⑦

The solver repeatedly processes constraints until reaching a fixed point, where no constraint causes any further updates to the variables. The monotonicity of the lattice operations ensures that

this process terminates. This procedure may add more cuts to B s than strictly necessary, but it never sets $B = \top$ unless that is the only way to satisfy all constraints. In other words, the discretization itself may be suboptimal in terms of the number of cuts.

Lattice constraint solving takes time polynomial in the size of the program: at worst, *every* constant in the program gets added to the cut set or value set of every subexpression of type **float**[$B; V$].

Additional Constraint for Contextual Equivalence. When the return type of the program involves a float type, either directly or as part of a larger compound type (e.g., **float**[$B; V$] $\rightarrow$ **float**[$B'; V'$]), we add an additional constraint to ensure that these float types are not discretized. We recursively traverse the return type and set all nested B sets to $\top$. This ensures that the observable behavior of the program is preserved after discretization, regardless of the context in which the program is used. When the return type is basic, such as **bool**, nothing needs to be done.

Example 4.2. To make the constraint solving process concrete, Figure 12 walks through §2.3. Each constraint obtained from the generation phase is processed by the solver, then collected to obtain the final cut and value sets.

5 Discretization

This section presents local transformation rules for our type-directed discretization algorithm, with examples. Our goal is to compute, from a typed expression $e : \mathbf{bool}$ and its type annotations, a *discretized* expression $\mathcal{D}\llbracket e : \mathbf{bool} \rrbracket : \mathbf{bool}$ such that e and $\mathcal{D}\llbracket e : \mathbf{bool} \rrbracket$ are *semantically equivalent*, i.e., they induce the same distribution. We refer to this semantics-preserving transformation as *exact discretization*, to distinguish it from approaches [19, 27] that approximate continuous programs via externally chosen precision parameters. Exactness here is the equality proved by Theorem 6.2: the transformed program preserves the relevant probabilities of the source program, not merely an approximation with an error bound. Intuitively, when a program uses continuous quantities only through finitely many comparison outcomes, exact inference should reduce to a discrete problem, and exact discretization makes that reduction explicit.

As shown in Figure 13, $\mathcal{D}\llbracket e : \tau \rrbracket$ is defined recursively on $e : \tau$ and its type annotations; we focus on continuous uniform distributions, with others handled analogously. The only sub-expressions replaced by $\mathcal{D}\llbracket \cdot \rrbracket$ are float constants and sampling instructions. The rule schemata are local, but their selection and parameters come from globally inferred cut/value information; §6 proves that this non-local, type-mediated construction preserves semantics. Before detailing the rules, we outline how type annotations guide discretization. The type system is designed so that every typed sub-expression of the form $e = \mathbf{uniform}(e_1, e_2)$ of type **float**[$B; V$] with $B = \{\sim_1 b_1, \dots, \sim_n b_n\} \neq \top$ can be *discretized* to an expression $e' : \mathbf{fin}(n+1)$ such that the probability that e evaluates to a value in $\text{Intervals}(B)_i$ *coincides* with the probability that e' evaluates to $i_{\#n+1}$. In particular, if *all* **float**[$B; V$]-typed subexpressions are finite-cut, then the given expression can be *fully discretized*.

Example 5.1. Consider a uniform distribution whose outcomes are compared to constants:

```
let x = uniform(0, 1) : float[B; T] in
  (x : float[B; T] < (0.3 : float[B; {0.3}])) ||
  (x : float[B; T] >= (0.5 : float[B; {0.5}]))
```

where $B = \{<0.3, <0.5\}$. The cut set B informs us that the probability of the comparisons evaluating to true is determined by the probability of x evaluating to one of the intervals

$$\text{Intervals}(B)_0 = (-\infty, 0.3) \quad \text{Intervals}(B)_1 = [0.3, 0.5) \quad \text{Intervals}(B)_2 = [0.5, \infty)$$

induced by B . Hence, all expressions of type **float**[$B; \cdot$] can be discretized to expressions of the finite type **fin**(3), where $k_{\#3}$ represents the k -th interval $\text{Intervals}(B)_k$ induced by B for $k \in \{0, 1, 2\}$.

$e : \tau$	Condition	$\mathcal{D}[\![e : \tau]\!]$
$x : \tau$		x
$c : \text{float}[B; V]$	$B = \top$	c
	$B \neq \top$	$k_{\#(B +1)}$ such that $c \in \text{Intervals}(B)_k$
$\text{let } x = e_1 : \tau_1 \text{ in } e_2 : \tau_2$		$\text{let } x = \mathcal{D}[\![e_1 : \tau_1]\!] \text{ in } \mathcal{D}[\![e_2 : \tau_2]\!]$
$(\text{if } e_1 : \text{bool} \text{ then } e_2 : \tau \text{ else } e_3 : \tau) : \tau$		$\text{if } \mathcal{D}[\![e_1 : \text{bool}]\!] \text{ then } \mathcal{D}[\![e_2 : \tau]\!] \text{ else } \mathcal{D}[\![e_3 : \tau]\!]$
$(e_1 : \text{float}[B; V_1] < e_2 : \text{float}[B; V_2]) : \text{bool}$	$B \neq \top$	$\mathcal{D}[\![e_1 : \text{float}[B; V_1]\!]\!] <_{\#(B +1)} \mathcal{D}[\![e_2 : \text{float}[B; V_2]\!]\!]$
	$B = \top$	$\mathcal{D}[\![e_1 : \text{float}[B; V_1]\!]\!] < \mathcal{D}[\![e_2 : \text{float}[B; V_2]\!]\!]$
$(e_1 : \text{float}[B; V_1] \leq e_2 : \text{float}[B; V_2]) : \text{bool}$	$B \neq \top$	$\mathcal{D}[\![e_1 : \text{float}[B; V_1]\!]\!] \leq_{\#(B +1)} \mathcal{D}[\![e_2 : \text{float}[B; V_2]\!]\!]$
	$B = \top$	$\mathcal{D}[\![e_1 : \text{float}[B; V_1]\!]\!] \leq \mathcal{D}[\![e_2 : \text{float}[B; V_2]\!]\!]$
	$e_1, e_2 \in \mathbb{R}, e_1 \geq e_2$	diverge
	$B = \top$	uniform ($\mathcal{D}[\![e_1 : \text{float}[B_1; V_1]\!]\!]$, $\mathcal{D}[\![e_2 : \text{float}[B_2; V_2]\!]\!]$)
	$e_1, e_2 \in \mathbb{R}_+$ $B = \{\sim_1 b_1, \dots, \sim_n b_n\}$	discrete ($p_0, \dots, p_n$) where $p_k = \text{CDF}(b_{k+1}) - \text{CDF}(b_k)$
$\text{uniform}(e_1 : \text{float}[B_1; V_1], e_2 : \text{float}[B_2; V_2]) : \text{float}[B; V]$	$B \neq \top,$ $B_1 \neq \top,$ $B_2 \neq \top,$ $V_1 = \{v_1, \dots, v_n\},$ $V_2 = \{w_1, \dots, w_m\}$	$\text{let } v'_1 = \mathcal{D}[\![e_1 : \text{float}[B_1; V_1]\!]\!] \text{ in}$ $\text{let } v'_2 = \mathcal{D}[\![e_2 : \text{float}[B_2; V_2]\!]\!] \text{ in}$ $\text{if } v'_1 ==_{\#(B_1 +1)} \mathcal{D}[\![v_1 : \text{float}[B_1; V_1]\!]\!] \text{ then}$ $\quad \&\& v'_2 ==_{\#(B_2 +1)} \mathcal{D}[\![w_1 : \text{float}[B_2; V_2]\!]\!] \text{ then}$ $\quad \quad \mathcal{D}[\![\text{uniform}(v_1, w_1) : \text{float}[B; V]]\!]$ $\quad \dots \text{ (enumerate over } V_1 \times V_2) \dots$ $\text{else if } v'_1 ==_{\#(B_1 +1)} \mathcal{D}[\![v_n : \text{float}[B_1; V_1]\!]\!] \text{ then}$ $\quad \&\& v'_2 ==_{\#(B_2 +1)} \mathcal{D}[\![w_m : \text{float}[B_2; V_2]\!]\!] \text{ then}$ $\quad \quad \mathcal{D}[\![\text{uniform}(v_n, w_m) : \text{float}[B; V]]\!]$ else diverge

Fig. 13. Discretization function, where $\text{CDF}(b_0) = 0$ and $\text{CDF}(b_{n+1}) = 1$. For brevity, the table writes $\text{CDF}(b_i)$ for each cut b_i , by which we mean either $\text{CDF}(b_i^-)$ or $\text{CDF}(b_i)$ depending on if the cut is strict or non-strict. Here, $\text{CDF}(b_i^-) = \Pr[X < b_i]$ denotes the left limit at b_i , while $\text{CDF}(b_i) = \Pr[X \leq b_i]$. These coincide for continuous distributions, but may differ for discrete or mixed distributions when $\Pr[X = b_i] > 0$. For the **uniform** distribution, $V = \top$ is implicit, and all remaining cut-set and value-set cases are ruled out by type inference. For the final **uniform** case, if $V_1 = \emptyset$ and/or $V_2 = \emptyset$, the case enumeration is empty and thus yields **diverge**. For all other language constructs, $\mathcal{D}[\![\cdot]\!]$ recurses structurally over them.

Now consider the discretized expression obtained from computing $\mathcal{D}[\![e : \text{bool}]\!]$, given by

```
let x = discrete(0.3, 0.2, 0.5) in (x <#3 1#3) || (x >=#3 2#3).
```

The probabilities appearing in **discrete**(...) are determined by the cumulative distribution function (CDF) of the uniform distribution over $[0, 1]$, ensuring that the probability of x evaluating to $k_{\#3}$ in $\mathcal{D}[\![e : \text{bool}]\!]$ coincides with the probability of x evaluating to a value in $\text{Intervals}(B)_k$ in e . Hence, the constants 0.3 and 0.5 in e can be replaced by the constants $1_{\#3}$ and $2_{\#3}$ in $\mathcal{D}[\![e : \text{bool}]\!]$.

Crucially, the *answers* constraint in the premise of the typing rules for comparisons (cf. Figure 6) ensures that B is a sufficiently fine-grained partition of $\mathbb{R}$, i.e., there is an injective function $\text{ToInts} : \{0.3, 0.5\} \rightarrow \text{Intervals}(B)$ that maps each constant $c \in \{0.3, 0.5\}$ to the unique interval in $\text{Intervals}(B)$ containing c . This interval is determined by the cut convention: a cut $<c$ puts c in the interval to the right of the boundary, while a cut $\leq c$ puts c in the interval to the left.

Example 5.2. If the arguments to a uniform sampling are non-constant, the situation becomes more challenging. Consider the following typed expression $e : \text{bool}$ (again only showing relevant types):

```

let x = (
  if uniform(0, 1) : float[{<0.5}; T] < (0.5 : float[{<0.5}; {0.5}]) then
    10 : float[{<=10,<=20}; {10}]
  else
    20 : float[{<=10,<=20}; {20}]) : float[{<=10,<=20}; {10,20}] in
uniform(0 : float[{<=0}; {0}], x : float[{<=10,<=20}; {10,20}]) : float[{<5}; T]
  < (5 : float[{<5}; {5}])

```

Why can this expression be discretized even though the argument to the second uniform sampling instruction is non-constant? The typing information ($x : \text{float}[\{<=10, <=20\}; \{10, 20\}]$) correctly tells us that x can only take the values 10 and 20. Consequently, x (and the **if**-expression generating it) can be discretized to a finite type **fin**(3) such that the probability of the resulting discrete expression evaluating to $0_{\#3}$ (resp. $1_{\#3}$) coincides with the probability of the continuous expression evaluating to 10 (resp. 20). Let us (partially) apply $\mathcal{D}\llbracket e : \text{bool} \rrbracket$, resulting in:

```

let x = if discrete(0.5, 0.5) <#2 1#2 then 0#3 else 1#3 in
( let v1' =  $\mathcal{D}\llbracket 0 : \text{float}[\{<=0\}; \{0\}] \rrbracket$  in
  let v2' =  $\mathcal{D}\llbracket x : \text{float}[\{<=10, <=20\}; \{10, 20\}] \rrbracket$  in
    if v1' ==#2  $\mathcal{D}\llbracket 0 : \text{float}[\{<=0\}; \{0\}] \rrbracket$  &&
      v2' ==#3  $\mathcal{D}\llbracket 10 : \text{float}[\{<=10, <=20\}; \{10, 20\}] \rrbracket$ 
    then  $\mathcal{D}\llbracket \text{uniform}(0, 10) : \text{float}[\{<5\}; T] \rrbracket$ 
    else  $\mathcal{D}\llbracket \text{uniform}(0, 20) : \text{float}[\{<5\}; T] \rrbracket$  ) <#2  $\mathcal{D}\llbracket 5 : \text{float}[\{<5\}; \{5\}] \rrbracket$ 

```

The continuous **uniform**(0, x)-expression is replaced by a case distinction on the discretizations of **uniform**(0, 10) and **uniform**(0, 20), with each case executed with the correct probability 0.5. Since this reduces the remaining discretization process to discretizing expressions with *constant parameters only*, the remaining steps are completely analogous to Theorem 5.1.

Crucially, the *recoverability* constraints in the CONTINUOUS-rules from Figure 6 ensure that the parameter expressions of the respective sampling expressions can take only finitely many values, and that the cut sets are fine-grained enough to recover the values the parameter expressions evaluate to, as demonstrated in the preceding example.

A similar intuition applies to discretizable comparisons $e_1 < e_2$, where *both* expressions are non-constant but finite-cut. The *answers* constraints in the rule LESS (resp. LESS-EQUAL) ensure that at least one of e_1 and e_2 evaluates to only finitely many values, which enables discretization:

Example 5.3. Consider the following typed expression ($e_1 \leq e_2$) : **bool**, given by

```

(if discrete(0.5, 0.5) <#2 1#2 then
  uniform(0, 10) : float[B; T] else 2.5 : float[B; {2.5}])
<=
(if discrete(0.5, 0.5) <#2 1#2
  then 2.5 : float[B; {2.5}] else 3.5 : float[B; {3.5}])

```

where $B = \{\leq 2.5, \leq 3.5\}$, and where $e_1 : \text{float}[B; T]$ and $e_2 : \text{float}[B; \{2.5, 3.5\}]$. Intuitively, e_1 samples from a mixture of a uniform and a dirac distribution, and e_2 samples from a finite distribution. Why can $e_1 \leq e_2$ be discretized even though neither e_1 nor e_2 is a constant? As suggested by its type, e_2 can take only finitely many values, namely $\{2.5, 3.5\}$. We apply $\mathcal{D}\llbracket \cdot \rrbracket$:

```

(if discrete(0.5, 0.5) <#2 1#2 then discrete(0.25, 0.1, 0.65) else 0#3)
<=
(if discrete(0.5, 0.5) <#2 1#2 then 0#3 else 1#3)

```

$$\mathcal{D}[\tau] = \begin{cases} \mathbf{fin}(|B| + 1) & \text{if } \tau = \mathbf{float}[B; V] \text{ and } B \neq \top \\ \mathbf{float}[B; V] & \text{if } \tau = \mathbf{float}[B; V] \text{ and } B = \top \\ \text{(straightforward recursion on type structure)} & \text{otherwise} \end{cases}$$

Fig. 14. Transformation of types under $\mathcal{D}$, where $\mathcal{D}$ acts on types.

Notice that the probability of e_1 evaluating to some value in $(-\infty, 2.5]$ indeed *coincides* with the probability of the discretization of e_1 evaluating to $0_{\#3}$. For e_2 , on the other hand, the probability of it evaluating to 2.5 (resp. 3.5) *coincides* with the probability of its discretization evaluating to $0_{\#3}$ (resp. $1_{\#3}$). For this reason, we obtain a semantically equivalent discretization of $e_1 \leq e_2$.

Discretization of Types. We conclude by extending $\mathcal{D}[\cdot]$ to types. Figure 14 shows how type τ is discretized to type $\mathcal{D}[\tau]$. We then obtain the following natural result, proved in [51, § E.1]:

THEOREM 5.4 (DISCRETIZATION IS WELL-TYPED). *Let $\Gamma \vdash e : \tau$. We have $\mathcal{D}[\Gamma] \vdash \mathcal{D}[e : \tau] : \mathcal{D}[\tau]$, where $\mathcal{D}[\Gamma]$ transforms all types in Γ , i.e., $\mathcal{D}[\Gamma](x) = \mathcal{D}[\Gamma(x)]$ for all $x \in \text{dom}(\Gamma)$.*

6 Operational Semantics and Soundness

We now establish the soundness of our approach in the following sense: For *every* expression $e : \mathbf{bool}$, possibly including *continuous sampling, higher-order functions, recursion, and conditioning*,

$e : \mathbf{bool}$ and $\mathcal{D}[e : \mathbf{bool}] : \mathbf{bool}$ (obtained from Figure 13) are semantically equivalent.

The first step is to make formal sense of semantic equivalence. For that, we will define a monadic operational semantics for our language in §6.1. We will then proceed by proving the above claim. This is non-trivial: discretization decisions are driven by non-local, type-propagated information, so we cannot proceed by a local rule-by-rule argument. To prove the claim rigorously, we relate the executions of e and $\mathcal{D}[e]$ through a coupling-style logical relation in §6.1.

6.1 Operational Semantics

The type-directedness of our discretization function $\mathcal{D}[\cdot]$ requires our soundness argument to depend on the types of the given (sub-)expressions as well. We thus start by defining a *type-dependent semantic functional*, i.e., we define for every type τ a function $\langle \cdot \rangle : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Vals}_\tau + \perp)$, mapping each *fully type-annotated* expression $e : \tau$ to a *sub-probability measure* $\langle e : \tau \rangle$ over values of type τ or $\perp$. In the presence of possibly unbounded recursion *and* conditioning, our semantics distinguishes between non-termination and observation failure, which we realize as follows: The “missing” probability mass in $\langle e : \tau \rangle$ is the probability that e diverges. The probability of $\perp$ is the probability of encountering an observation failure. With this model, semantic equivalence indeed captures our sought-after correctness claim: If $\langle e : \mathbf{bool} \rangle = \langle \mathcal{D}[e : \mathbf{bool}] : \mathbf{bool} \rangle$, then both expressions have the same probability of producing true, false, or an observation failure.

Remark 6.1. Distinguishing between non-termination and observation failure yields a more fine-grained semantics which does, e.g., *not* semantically equate the two expressions

let $x = \mathbf{uniform}(0, 1)$ **in** **observe** $(x > 1)$ **and** **diverge**.

In particular, our soundness theorem does not deem the latter a valid discretization of the former.

6.1.1 Small Step Semantics. The type-dependent semantic functional $\langle \cdot \rangle$ is defined via a *small-step* semantic functional $\llbracket \cdot \rrbracket : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$. Defining the latter requires us to carefully design a suitable measurable space over the set of type-annotated expressions of type τ and $\perp$. All details

on these measure-theoretic constructions are presented in the extended version [51, § F.2]. Let us consider the key idea here.

Let $\tau = \mathbf{bool}$ for the sake of concreteness. Care must be taken since, e.g., sub-expressions of type $\mathbf{float}[B; \{3.5, 4.5\}]$ cannot evaluate to *arbitrary* reals but are restricted to the values $\{3.5, 4.5\}$. We tackle this by extracting from each type-annotated expression a suitable *skeleton*, where float constants are replaced by “holes” (hence the term “skeleton”). For instance, the expression

$$e = (x : \mathbf{float}[\{<3.5, <4.5\}; \top] < y : \mathbf{float}[\{<3.5, <4.5\}; \{3.5, 4.5\}]) : \mathbf{bool}$$

gives rise to the skeleton

$$s = (\Box : \mathbf{float}[\{<3.5, <4.5\}; \top] < \Box : \mathbf{float}[\{<3.5, <4.5\}; \{3.5, 4.5\}]) : \mathbf{bool}$$

and the type annotations tell us that the possible values these holes can be filled with are in $\mathbb{R} \times \{3.5, 4.5\}$. We can thus inherit a measurable space over all well-typed expressions arising from this particular skeleton s from the standard Borel measurable space over $\mathbb{R} \times \{3.5, 4.5\}$. The measurable space over *all* well-typed expressions of type $\mathbf{bool}$ (i.e., over $\text{Expr}_{\mathbf{bool}}$) is then obtained from the measurable spaces of *all* skeletons that give rise to expressions of type $\mathbf{bool}$.

Equipped with this family of measurable spaces, we define the small-step functional $\llbracket \cdot \rrbracket$ in the expected, type-preserving manner (see [51, Tables 1 and 2]). For instance, if e_1 is not a value, then¹

$$\llbracket (\mathbf{let} \ x = e_1 : \tau_1 \ \mathbf{in} \ e_2 : \tau_2) : \tau_2 \rrbracket = \underbrace{\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \begin{cases} \delta_{\perp} & \text{if } g = \perp \\ \delta_{(\mathbf{let} \ x = g \ \mathbf{in} \ e_2 : \tau_2) : \tau_2} & \text{otherwise} \end{cases}}_{\text{standard probabilistic monadic bind over suitable sub-Markov kernels}},$$

i.e., $\llbracket e_1 : \tau_1 \rrbracket$ yields the sub-distribution over expressions obtained from executing e_1 for *one step*, and the monadic bind $\ggg$ “folds” this distribution into the surrounding **let**-statement. The intuition for the remaining statements is analogous. Crucially, $\ggg$ propagates the probability of encountering $\perp$ — an observation failure — through executions. For instance, assume that e_1 from above is given by **observe** ($\text{false} : \mathbf{bool}$) : **unit**. Since e_1 encounters an observation failure with probability 1, i.e., $\llbracket \mathbf{observe} \ (\text{false} : \mathbf{bool}) : \mathbf{unit} \rrbracket (\{\perp\}) = 1$, the same holds for the surrounding **let**-statement, i.e., also $\llbracket (\mathbf{let} \ x = e_1 : \tau_1 \ \mathbf{in} \ e_2 : \tau_2) : \tau_2 \rrbracket (\{\perp\}) = 1$. All details — including proofs that all monadic binds are well-behaved since they operate on suitable sub-Markov kernels — are provided in the extended version [51, § F.4].

6.1.2 From Small-Step to Big-Step Semantics. We now define $\langle \cdot \rangle$ via $\llbracket \cdot \rrbracket$. For that, we first lift the one-step semantics $\llbracket \cdot \rrbracket$ to an n -step semantics $\langle \cdot \rangle_n$ in the expected way:

$$\langle e : \tau \rangle_0 = \delta_{e:\tau} \quad \text{and} \quad \langle e : \tau \rangle_{n+1} = \langle e : \tau \rangle_n \ggg \lambda e'. \begin{cases} \delta_{\perp} & \text{if } e' = \perp \\ \llbracket e' : \tau \rrbracket & \text{otherwise} \end{cases}$$

This construction is well-defined since the functional $\llbracket \cdot \rrbracket$ itself gives rise to a suitable sub-Markov kernel, as proved in the extended version [51, Lemma F.39]. We then define for every type τ the functional $\langle \cdot \rangle : \text{Expr}_{\tau} \rightarrow \mathcal{D}(\text{Vals}_{\tau} + \perp)$ as $\langle e : \tau \rangle = \lambda A. \lim_{n \rightarrow \infty} \langle e : \tau \rangle_n(A)$, which is well-defined by the monotone convergence theorem: the sequence $(\langle e : \tau \rangle_n)_{n \in \mathbb{N}}$ is monotonically increasing for every measurable set A over $\text{Vals}_{\tau} + \perp$ (see the extended version [51, Lemma F.49]) and upper-bounded by 1: executing a program for more and more steps can only increase the probability of terminating in a given set of values or an observation failure.

¹Put formally, $(\mu \ggg f)(A) = \int f(x)(A) \cdot \mu(dx)$; see the extended version [51, Definition F.28] for details. δ_e denotes the Dirac distribution of e .

6.2 Soundness via Probabilistic Couplings

We first state the central theorem, then outline our coupling-style correctness argument. Throughout, $e : \tau$ denotes a closed, fully annotated expression satisfying $\emptyset \vdash e : \tau$ under the declarative typing rules.

THEOREM 6.2. *Let $e : \mathbf{bool}$. Then $\langle e : \mathbf{bool} \rangle = \langle \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle$.*

Intuitively, we prove that the two programs can execute in a synchronized manner such that their probabilities of terminating in value v coincide for each $v \in \{\text{true}, \text{false}, \perp\}$.

Towards this end, define for every type τ the set $P_\tau = \{(e : \tau, \mathcal{D}[\![e : \tau]\!] : \mathcal{D}[\![\tau]\!]) \mid e : \tau\}$. We define a *coupled small-step semantics* $\llbracket \cdot, \cdot \rrbracket : P_\tau \rightarrow \mathcal{D}(P_\tau + \perp)$ (see [51, Tables 3 and 4]), which models the simultaneous execution of e and $\mathcal{D}[\![e : \tau]\!]$. The key property of this coupled semantics is that the semantics of both e and $\mathcal{D}[\![e : \tau]\!]$ can be recovered in a sense we formalize next.

Given $\mu \in \mathcal{D}(P_\tau + \perp)$, we define the *projection of μ onto the i -th component* for $i \in \{1, 2\}$ as

$$\pi_i(\mu) = \lambda A. \int_{x \in P_\tau + \perp} \begin{cases} \delta_{e_i}(A) & x = (e_1, e_2) \\ \delta_\perp(A) & x = \perp \end{cases} \mu(dx),$$

With projections, we recover the semantics of e and $\mathcal{D}[\![e : \tau]\!]$ as follows:

LEMMA 6.3. *Let $e : \tau$. We have:*

- (1) $\pi_1(\llbracket e : \tau, \mathcal{D}[\![e : \tau]\!] : \mathcal{D}[\![\tau]\!]\rrbracket) = \llbracket e : \tau \rrbracket$.
- (2) $\pi_2(\llbracket e : \tau, \mathcal{D}[\![e : \tau]\!] : \mathcal{D}[\![\tau]\!]\rrbracket) = \llbracket \mathcal{D}[\![e : \tau]\!] : \mathcal{D}[\![\tau]\!] \rrbracket$.

Now, using a limit construction analogous to §6.1.2 for obtaining a coupled *big-step* semantics $\langle \cdot, \cdot \rangle : P_\tau \rightarrow \mathcal{D}(V_\tau + \perp)$, where $V_\tau = \{(v : \tau, \mathcal{D}[\![v]\!] : \mathcal{D}[\![\tau]\!]) \mid v : \tau \text{ is a value}\}$, we prove a suitable continuity property of projection (see [51, Lemma F.55]) to conclude that Lemma 6.3 transfers to $\langle \cdot, \cdot \rangle$. Finally, observing that $V_{\mathbf{bool}} + \perp = \{(\text{true}, \text{true}), (\text{false}, \text{false}), \perp\}$ yields the desired claim:

Proof of Theorem 6.2. Let $v \in \{\text{true}, \text{false}, \perp\}$. We have

$$\begin{aligned} & \langle e : \mathbf{bool} \rangle(v) \\ &= \pi_1(\langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle)(v) && \text{(Lemma 6.3.1 for } \langle \cdot, \cdot \rangle \text{)} \\ &= \pi_1(\langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle)(v) && (P_{\mathbf{bool}} \text{ is discrete)} \\ &= \langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle((\text{true}, \text{true})) \cdot \delta_{\text{true}}(v) \\ &\quad + \langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle((\text{false}, \text{false})) \cdot \delta_{\text{false}}(v) \\ &\quad + \langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle(\perp) \cdot \delta_\perp(v) && \text{(definition of } \pi_1, \text{ argument is discrete)} \\ &= \pi_2(\langle e : \mathbf{bool}, \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle)(v) && \text{(definition of } \pi_2, \text{ argument is discrete)} \\ &= \langle \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle(v) && \text{(Lemma 6.3.2 for } \langle \cdot, \cdot \rangle \text{)} \end{aligned}$$

7 Implementation, Case Studies, and Evaluation

We have implemented the SLICE type inference and discretization procedures in OCaml. To compute the interval masses of continuous distributions, we use the OCaml GSL package [18] for cumulative distribution function calculations. All experiments were run on an Apple M3 with 16 GB of RAM.

This section is driven by two research questions:

- (1) Does SLICE enable exact inference on programs combining continuous sampling, higher-order functions, and unbounded recursion?
- (2) On programs that are within the scope of existing tools for exact inference, is the combination of SLICE and discrete inference backends competitive?

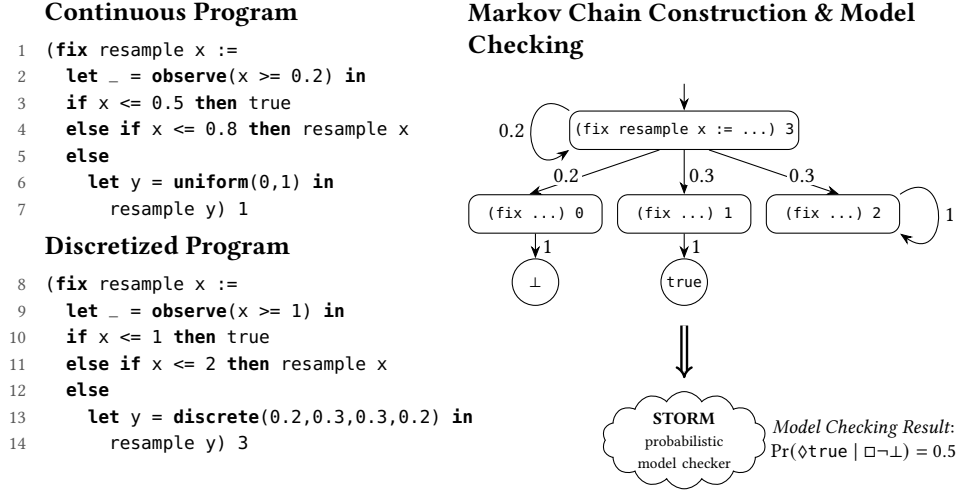


Fig. 15. Conditional resampling loop and its discretization. The continuous program is transformed into a finite-state discrete program (SLICE alone takes less than 0.001s), which induces a finite Markov chain (generated in 0.001s) and can then be analyzed by STORM (with a model checking time of 0.024s). The $\perp$ -state in that Markov chain is entered when encountering an observation failure.

For the first question, we provide a collection of case studies that involve continuous distributions together with recursion, higher-order functions, lists, or conditioning in §7.1. For these case studies, we use SLICE to transform the original programs to finite discrete models and analyze the resulting program with STORM [15] — a probabilistic model checker.

For the second question, we evaluate SLICE+DICE and SLICE+ROULETTE on two classes of benchmarks in Sections 7.2 and 7.3: (i) synthetic scaling benchmarks that let us vary program size and dependency structure in a controlled way, and (ii) benchmark suites from prior work [21, 40]. We compare against SPPL [40], an exact inference engine supporting continuous probabilistic programs, on benchmarks that *both* SLICE and SPPL support, and we additionally report PSI [21] and HAKARU [37] timings for the representative prior-work benchmarks. We remark, however, that SPPL supports arithmetic, as well as (soft) conditioning on measure zero events, which SLICE does not support. Our evaluation merely serves the purpose of investigating (i) the overhead of discretization and (ii) the benefits of being able to use efficient *BDD-based* inference back-ends (i.e., DICE and ROULETTE) for exact inference on *continuous programs*.

7.1 SLICE+STORM: Case Studies

There is no exact inference system that, to our knowledge, can cover programs combining all three features: continuous sampling, higher-order functions, and unbounded recursion. As a consequence, there is no exact inference engine for the programs in Sections 2.6, 7.1.2 and 7.1.3. Instead, we use *probabilistic model checking*: The small-step semantics of our language (given in the extended version [51, Tables 1 and 2]) for discrete programs gives rise to a *Markov chain* modeling the stochastic process of program execution, inspired by [29] on imperative programs. If that Markov chain is finite, we can use the probabilistic model checker STORM [15] for exact inference. Let us start with an illustrative example.

7.1.1 Conditional Resampling Loop. Consider the continuous program depicted in Figure 15, containing continuous sampling, unbounded recursion, and conditioning inside of the recursion. SLICE

automatically discretizes this program, also shown in Figure 15. We generate the corresponding Markov chain by unfolding the small-step semantics in a breadth-first manner. In the presence of unbounded recursion, this Markov chain *might* be infinite and this process might thus diverge. If, however, it terminates, the resulting Markov chain is finite and precisely captures the execution behavior of the discretized program. For this example, this is the case: Even though the discretized program contains unbounded recursion, its execution behavior can be modelled as a finite-state Markov chain², where $\perp$ is a special state entered when encountered an observation failure.

Now, using the probabilistic model checker STORM, we can perform exact inference: It is immediate that the Markov chain's probability $\Pr(\diamond \text{true} \mid \Box \neg \perp)$ of eventually reaching true (in LTL: $\diamond \text{true}$) *given* that we never enter the observation-failure state $\perp$ (in LTL: $\Box \neg \perp$) equals the measure of true in the program's normalized posterior distribution.

7.1.2 Random Interval Covering of $[0, 1]$ with Conditioning. Our second case study extends the random interval covering example from §2.6 by conditioning inside of the recursion: We condition on *every* sample falling within *at most* one interval. The difference to the program in §2.6 is:

```

1  let non_overlap = fix non_overlap y :=      10 let cover = fix cover st :=
2  fun seen -> fun ys ->                      11  if uniform(0,1) < 0.1 then st
3  match ys with                               12  else
4  | nil -> true                               13  let y = gaussian(0,1) in
5  | h :: t ->                                 14  let _ = observe(non_overlap y false st) in
6      let hit = (snd h) y in                  15  cover (step y st)
7      if hit && seen then false                16  in
8      else non_overlap y (seen || hit) t      17 let final = cover init in
9  end in                                       18 is_covered final

```

The key new component is the recursive function `non_overlap` (l. 1), which checks whether a sampled point y hits more than one interval of the initially sampled intervals. The loop `cover` (l. 10) repeatedly samples a Gaussian point and immediately conditions on this predicate via `observe` (l. 14), so only executions in which the point lies in at most one interval are retained. The measure of true in the conditional posterior distribution is thus the probability that this unbounded process hits every interval at least once *given* that each sampled point falls in at most one interval.

The program remains discretizable: each sampled point is still used only through comparisons against finitely many sampled interval boundaries, and `non_overlap` depends only on the resulting hit pattern, not on the exact real value of y . SLICE replaces each Gaussian draw by its interval index, yielding a discrete program (shown in [51, Figure 22]) with a finite exact state space. SLICE takes 0.001s, and we generate a Markov chain consisting of 268878 states in 80 seconds. STORM takes 4.7 seconds for model checking, and reports that the sought-after probability is 0.000227....

Currently, our Markov chain construction process is rather naive: extending the number of sampling points or intervals yields the Markov chain to be infeasibly large. Developing efficient symbolic discrete inference engines for higher-order programs with unbounded recursion is not in the scope of this paper. However, our approach is an excellent motivation for this future direction.

7.1.3 Bloom Filter. We model a probabilistically populated variant of a Bloom filter [7], which is a data structure using hash functions for storing data space efficiently: one can insert data into a Bloom filter and subsequently query whether a given data point is stored in it. Due to aggressive compression via hashing, this process can be faulty: while it can never produce false-negatives, it can produce false-positives. Now consider the following program (full program in [51, Figure 23]):

²We apply a straightforward optimization: Intermediate execution steps that do not branch probabilistically can be collapsed into a single transition. This is the reason why all states in Figure 15 correspond to the header of the recursive function.

```

1  let insert = fun q -> fun h1 ->
2    fun h2 -> fun bf ->
3    let index1 = h1 q in
4    let index2 = h2 q in
5    let bf1 = set_true bf index1 in
6    set_true bf1 index2 in
7
8  let search = fun q -> fun h1 ->
9    fun h2 -> fun bf ->
10   let index1 = h1 q in
11   let index2 = h2 q in
12   (get bf index1) && (get bf index2) in
13
14  let h1 = fun q ->
15    if q <= -2 then 0
16    else if q <= 0 then 1
17    else if q <= 2 then 2
18    else 3 in
19
19  let h2 = fun q ->
20    if q <= -2 then 3
21    else if q <= 0 then 2
22    else if q <= 2 then 1
23    else 0 in
24
25  let init = (false, (false, (false, false))) in
26  let key = gaussian(0,1) in
27
28  let search_before_full = fix loop bf :=
29    if (full bf) then bf
30    else if (search key h1 h2 bf) then bf
31    else
32      let q = gaussian(0,1) in
33      loop (insert q h1 h2 bf) in
34
35  let bf_final = (search_before_full init) in
36  not (full bf_final)

```

The program samples a Gaussian key (l. 26) and starts from an empty four-bit Bloom filter (l. 25). It then builds the filter recursively by drawing fresh Gaussian samples (l. 32) and inserting each one into the filter by hashing the sampled real twice using the comparison-based hash functions $h1$ and $h2$ (l. 14 and 19), following a locality-sensitive hashing technique that hashes similar input items into the same buckets. Each insertion sets the corresponding bits in the filter using `insert` (l. 1). The unbounded recursion continues until either the filter becomes full or the key is reported present by `search` (l. 8). The final Boolean result, `not (full bf_final)` (l. 36), therefore asks whether the key was found strictly before the filter became full.

The key simplification is that the hash functions never inspect the exact value of a sample. They only ask which region the sample falls into, using the cut points -2 , 0 , and 2 . These cuts partition the real line into four regions, which already coincide with the Bloom filter's index space. SLICE therefore replaces each Gaussian draw by its bucket index while preserving exactly the probability of every collision pattern relevant to insertion and search. The fully discretized program is shown in the extended version [51, Figure 24]. SLICE takes less than 0.001s, and we generate a Markov chain consisting of 14 states in 0.023 seconds. STORM takes 0.001 seconds for model checking, and reports the probability 0.91314....

7.2 SLICE+DICE and SLICE+ROULETTE: New Benchmarks for Scaling

In the absence of unbounded recursion, we can use DICE or ROULETTE as inference backends. We designed scalable benchmarks of conditionally independent continuous programs with varying variable dependencies. Formally, all programs in our benchmark can be described by the structure: $X_i \perp\!\!\!\perp \{X_j : j < i, j \neq \pi(i)\} \mid X_{\pi(i)}$, where $\pi(i) \in \{1, \dots, i-1\}$ is the parent index of X_i . Each variable X_i depends directly on only one earlier variable, namely its parent $X_{\pi(i)}$; conditioned on $X_{\pi(i)}$, it is independent of all other preceding variables. Detailed listings and graphs for variants of these benchmarks appear in the extended version [51, § G.1.1]. Results are depicted in Figure 16.

We report runtime (seconds) for (i) SPPL, (ii) SLICE (discretization only), (iii) SLICE+DICE (discretization plus inference via DICE), and (iv) SLICE+ROULETTE (discretization plus inference via ROULETTE). The timeout is 120 seconds. Program sizes are evaluated in increments of 5, from 0 to 50. For each size, we generate 10 random instances and report average runtime. We observe that discretization remains small across sizes, and takes less than 0.01 seconds for each program. End-to-end runtimes for SLICE with discrete backends grow more mildly on these families; SPPL

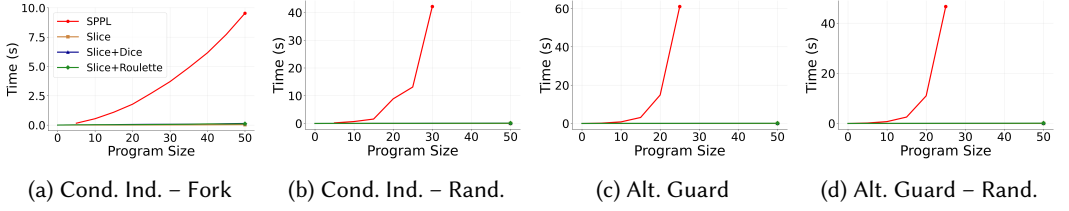


Fig. 16. Scaling behavior on representative benchmark families (additional families are provided in the extended version [51, § G.1.1]). The plots emphasize asymptotic growth-rate differences and the resulting timeout frontier as program size increases. Time-out=120s; a disappearing line indicates timeout.

reaches a timeout frontier at larger instances. We conclude that using efficient BDD-based back-ends for exact inference on continuous programs via discretization can pay off for programs whose control flow induces strong conditional independence structure, allowing the resulting discrete models to remain compact even as program size grows.

7.3 SLICE+DICE and SLICE+ROULETTE: Existing Benchmarks

We next use benchmark suites from prior work, detailed in the extended version [51, § G, Tables 5–7].

The decision tree benchmarks from [1] assess fairness properties under different population models. In the extended version [51, Tables 5 and 6], DT_n denotes a tree with n conditionals, the second column specifies the population model, and the remaining columns report runtimes. We split SPPL execution into translation and fairness-judgment phases to make phase costs explicit. We additionally report runtimes for PSI and HAKARU. For HAKARU, we separate Disintegrate, Simplify, and Total runtimes. Hakaru’s disintegration phase transforms the original probabilistic program into a new Hakaru program representing a conditional distribution, which is then simplified by Maple to perform exact inference through computer algebra. Thus, the Total column reports the timing for the full exact inference pipeline. We observe that discretization itself (SLICE) becomes expensive for larger n , eventually yielding a timeout. Regarding end-to-end runtimes, SLICE+DICE mostly outperforms SPPL, whereas SLICE+ROULETTE is mostly inferior to SPPL for larger n .

The extended version [51, Table 7] reports on benchmarks drawn from the PSI and SPPL benchmark sets. For benchmarks adapted by SPPL from PSI, we use the modified versions provided in the SPPL Github repo, which replace certain continuous choices with finite ones. Translating these versions to SLICE enables a direct comparison between SLICE and SPPL. For completeness, we also report PSI/HAKARU results on the original, unmodified programs, which remain outside the class supported by both SLICE and SPPL. Moreover, programs requiring numeric transformations or arithmetic *remain outside the language features of SLICE*. Like in the decision tree benchmarks, we report runtimes for (i) SLICE+DICE, (ii) SLICE+ROULETTE, (iii) SPPL, (iv) PSI, (v) HAKARU (Disintegrate), and (vi) HAKARU (Simplify) in seconds. We observe that SLICE+DICE and SLICE+ROULETTE are competitive with SPPL, PSI, and HAKARU (Disintegrate + Simplify).

8 Related and Future Work

We start with related work and conclude with future work.

Discretization approaches. Holtzen et al. [25] are the closest prior work on exact discretization. They develop a probabilistic predicate abstraction framework that, given predicates of interest, constructs an abstract probabilistic program and proves soundness for events expressible in the predicate domain. For a first-order, loop-free fragment of our language, SLICE can be viewed similarly: inferred cuts such as $x < c$ and $x \leq c$ induce predicates over which the program is discretized.

The key difference is that *SLICE automatically infers and propagates* the relevant cut/value sets through higher-order functions, recursion, and observations, then compiles to a discrete program for existing exact backends. By contrast, Holtzen et al. [25] assume predicates are supplied a priori, explicitly identify useful predicate discovery as a hard problem, and restrict their semantics to measurable functions, excluding arbitrary unbounded loops and higher-order functions. Beyond this, other discretization methods use approximate discretization [11, 19, 27]. Recent work also uses approximate discretization to compute posterior bounds for recursive higher-order programs [4]. Vice versa, Leios [32] approximates a fully continuous program from a discrete one.

Discrete-only probabilistic languages. Roulette [35] 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 inference on discrete programs. Dice [26] follows a similar philosophy in an OCaml DSL, compiling discrete programs to weighted model counting. Pluck [8] enables exact and approximate inference in discrete probabilistic programs based on lazy evaluation. Earlier work in probabilistic logic programming, most prominently ProbLog [14], annotates Prolog facts with probabilities and reduces inference to weighted Boolean formulas. PERPL is a system that performs exact inference on recursive calls over recursive data [10]. Caesar [42] is a quantitative program verification infrastructure for discrete probabilistic programs that together with the approach from [2] enables the verification of continuous programs.

Exact inference on continuous programs. Many existing systems perform exact inference directly in continuous/hybrid languages. Psi [21] and Hakaru [37] support symbolic exact inference capabilities with different algebraic strengths, though symbolic generality can make some tractable instances challenging. Symbo [17] leverages weighted model integration [3] for exact inference in hybrid domains, using sentinel decision diagrams as the representation and the PSI solver to symbolically integrate over continuous variables. SPPL [40] emphasizes efficient exact symbolic inference for its fragment and supports features we do not currently automate, notably numeric transformations and conditioning on measure-zero events. λ PSI [22] supports exact inference for higher-order continuous programs with nested inference and bounded recursion. Generating-function systems such as Genfer [53], Geni [33], and Prodigy [31] collectively support exact inference for infinite-support models and restricted unbounded-control models, but not general higher-order programs. PERPL [10] supports higher-order functions and unbounded recursion, but not continuous random variables. GuBPI [4] supports recursive higher-order programs with continuous distributions, but computes guaranteed converging bounds rather than exact posteriors. *SLICE* targets the combination needed for our case studies: exact inference for programs that combine continuous sampling, higher-order functions, and unbounded recursion.

Relational proof techniques. Our soundness argument uses a coupling-style semantic relation between source and discretized executions. This connects directly to logical-relations and relational-semantics techniques for probabilistic programs [6, 12, 48], adapted here to a global, non-local, type-guided transformation. Notably, [6] considers only discrete probabilistic programs without conditioning, while [12, 48] support continuous distributions and scoring. More recently, [54] considered probabilistic programs with nested queries and recursion using a step-indexed model.

Future Work. We would like to extend our work to support tractable symbolic integration for richer classes of continuous distributions (i.e. conjugate priors or other distributions admitting closed-form expressions), thus broadening the class of programs that *SLICE* can discretize exactly. We also plan to investigate symbolic techniques à la Wang et al. [49] to speed-up inference in the presence of recursion.

Data-Availability Statement

A reproducible artifact for the evaluation in §7 is available on Zenodo [50]. The extended version of this paper (e.g. with appendices) is also available [51].

Acknowledgments

We are grateful to our OOPSLA reviewers who helped us improve the paper. We also thank Cheng Zhang for providing feedback on an initial draft of our paper, as well as the Silva Lab retreat guests for helpful discussions. This work was supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-2139899. Additionally, this research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) and the Office of Naval Research (ONR). The views, opinions, and/or findings contained in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of the Navy, DARPA, or the U.S. Government.

References

- [1] Aws Albarghouthi, Loris D’Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3133904
- [2] Kevin Batz, Joost-Pieter Katoen, Francesca Randone, and Tobias Winkler. 2025. Foundations for Deductive Verification of Continuous Probabilistic Programs: From Lebesgue to Riemann and Back. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3720429
- [3] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *International Joint Conference on Artificial Intelligence (IJCAI)*. <https://dl.acm.org/doi/10.5555/2832581.2832636>
- [4] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programming. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3519939.3523721
- [5] 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).
- [6] Aleš Bizjak and Lars Birkedal. 2015. Step-Indexed Logical Relations for Probability. In *Foundations of Software Science and Computation Structures (FoSSaCS)*. doi:10.1007/978-3-662-46678-0_18
- [7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. doi:10.1145/362686.362692
- [8] Maddy Bowers, Alexander K. Lew, Joshua B. Tenenbaum, Armando Solar-Lezama, and Vikash K. Mansinghka. 2025. Stochastic Lazy Knowledge Compilation for Inference in Discrete Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3729325
- [9] 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
- [10] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3586050
- [11] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. doi:10.1145/2491411.2491423
- [12] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-662-54434-1_14
- [13] 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 *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3314221.3314642
- [14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *International Joint Conference on Artificial Intelligence (IJCAI)*. <https://dl.acm.org/doi/10.5555/1625275.1625673>
- [15] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verification (CAV)*. doi:10.1007/978-3-319-63390-9_31

- [16] 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. arXiv:1711.10604 doi:10.48550/arXiv.1711.10604
- [17] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. 2019. Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation. In *AAAI Conference on Artificial Intelligence (AAAI)*. doi:10.1609/aaai.v33i01.33017825
- [18] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. 2009. *GNU Scientific Library Reference Manual* (3rd ed.). <http://www.gnu.org/software/gsl/>
- [19] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2024. Bit Blasting Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3656412
- [20] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research, Vol. 84)*. PMLR, 1682–1690. <https://proceedings.mlr.press/v84/ge18b.html>
- [21] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV)*. doi:10.1007/978-3-319-41528-4_4
- [22] Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386006
- [23] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. <https://dl.acm.org/doi/10.5555/3023476.3023503>
- [24] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. <http://dippl.org>
- [25] 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 (ICML) (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 1999–2008. <https://proceedings.mlr.press/v80/holtzen18a.html>
- [26] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3428208
- [27] Zixin Huang, Saikat Dutta, and Sasa Misailovic. 2021. AQUA: Automated Quantized Inference for Probabilistic Programs. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. doi:10.1007/978-3-030-88885-5_16
- [28] Jules Jacobs. 2021. Paradoxes of Probabilistic Programming: And How to Condition on Events of Measure Zero with Infinitesimal Probabilities. In *Principles of Programming Languages (POPL)*. doi:10.1145/3434339
- [29] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. doi:10.1007/978-3-319-46520-3_5
- [30] Olav Kallenberg. 1997. *Foundations of Modern Probability*. Springer-Verlag, New York.
- [31] Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian Inference for Loopy Probabilistic Programs Using Generating Functions. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3649844
- [32] Jacob Laurel and Sasa Misailovic. 2020. Continualization of Probabilistic Programs With Correction. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-030-44914-8_14
- [33] Jianlin Li and Yizhou Zhang. 2025. Compiling with Generating Functions. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/3747534
- [34] Tom Minka, John M. Winn, John Guiver, Yordan Zaykov, David Fabian, and Jim Bronskill. 2018. Infer.NET 0.3. <http://dotnet.github.io/infer>
- [35] 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 *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3729334
- [36] James R. Munkres. 2000. *Topology* (2nd ed.). Prentice Hall, Upper Saddle River, NJ.
- [37] Praveen Narayanan, Jacques Carette, Wren Romano, Chung chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming (FLOPS)*. doi:10.1007/978-3-319-29604-3_5
- [38] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems* (1999). doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4
- [39] Avi Pfeffer. 2009. *Figaro: An Object-Oriented Probabilistic Programming Language*. Technical Report. Charles River Analytics. <https://www.cs.tufts.edu/comp/150FP/archive/avi-pfeffer/figaro.pdf>

- [40] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3453483.3454078
- [41] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic Programming in Python Using PyMC3. *PeerJ CS* (2016). doi:10.7717/peerj-cs.55
- [42] Philipp Schroer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. A Deductive Verification Infrastructure for Probabilistic Programs (Extended Version). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3622870
- [43] Elias M. Stein and Rami Shakarchi. 2005. *Real Analysis: Measure Theory, Integration, and Hilbert Spaces* (1st ed.). Princeton Lectures in Analysis, Vol. 3. Princeton University Press, Princeton, NJ.
- [44] Terence Tao. 2011. *An Introduction to Measure Theory* (1st ed.). Graduate Studies in Mathematics, Vol. 126. American Mathematical Society, Providence, RI.
- [45] Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Sepehr Masouleh, Eric Lippert, and Erik Meijer. 2020. Bean Machine: A Declarative Probabilistic Programming Language For Efficient Programmable Inference. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models (PGM) (Proceedings of Machine Learning Research, Vol. 138)*. PMLR, 485–496. <https://proceedings.mlr.press/v138/tehrani20a.html>
- [46] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Implementation and Application of Functional Languages (IFL)*. doi:10.1145/3064899.3064910
- [47] 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:1610.09787 doi:10.48550/arXiv.1610.09787
- [48] Mitchell Wand, Theophilos Giannakopoulos, Andrew Cobb, and Ryan Culpepper. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/3236782
- [49] Peixin Wang, Tengshun Yang, Hongfei Fu, Guanyan Li, and C.-H. Luke Ong. 2024. Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3656432
- [50] Katherine Wu, Jules Jacobs, Kevin Batz, and Alexandra Silva. 2026. Artifact: Type-Directed Discretization of Probabilistic Programs. Zenodo. doi:10.5281/zenodo.21780754
- [51] Katherine Wu, Jules Jacobs, Kevin Batz, and Alexandra Silva. 2026. Type-Directed Discretization of Probabilistic Programs (Extended Version). arXiv:2608.16093 doi:10.48550/arXiv.2608.16093
- [52] Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *Proceedings of the 35th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 5343–5352. <https://proceedings.mlr.press/v80/wu18f.html>
- [53] Fabian Zaiser, Andrzej S. Murawski, and Luke Ong. 2023. Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach. In *Conference on Neural Information Processing Systems (NeurIPS)*. doi:10.52202/075280-0113
- [54] Yizhou Zhang and Nada Amin. 2022. Reasoning about “Reasoning about Reasoning”: Semantics and Contextual Equivalence for Probabilistic Programs with Nested Queries and Recursion. In *Principles of Programming Languages (POPL)*. doi:10.1145/3498677

CONTENTS

Contents	29
A SLICE Examples (Continued)	29
A.1 Control Flow	29
A.2 Discrete Latent Variable	30
A.3 Indian GPA Problem	32
A.4 Point and Measure-Zero Observations	33
A.5 Random Interval Covering Over $[0, 1]$	34
B SLICE Case Studies (continued)	34
B.1 Random Interval Covering Over $[0, 1]$ with Conditioning	34
B.2 Bloom Filter	35
C List of Supported SLICE Distributions	39
D Typing Rules for Full SLICE Language	39
E Discretization Proof	41
E.1 Well-Typedness of Discretization	41
F Soundness Proof	42
F.1 Measure Theory Preliminaries	42
F.2 Measure Space on Expressions	45
F.3 Distribution Monad with Error	52
F.4 Small-Step Semantics	55
F.5 Small-Step Coupling Semantics	62
F.6 Big-Step Semantics and Correctness of Discretization	67
G Benchmarks	73
G.1 Asymptotic Scaling	73
G.2 Decision Trees	73
G.3 Baselines	78

A **SLICE** Examples (Continued)

This section is a continuation of §2, describing more in detail the following examples: Sections 2.3 to 2.6. Additionally, we include an extra example based on the Indian GPA problem, a canonical example in the probabilistic programming literature [52].

A.1 Control Flow

We revisit the control-flow example from §2.3:

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

The main feature of this program is that the comparison $x \leq y$ is governed by control flow: the right-hand side is not a single constant, but may evaluate to either 0.3 or 1.8 depending on the outcome of the earlier test $1.5 > x$. Consequently, the discretization of x must account for both these values.

We show, at a high-level, how the float type information for this program is propagated. First, Figure 17 makes these dependencies explicit using a constraint graph over float-typed subexpressions. Solid edges denote data flow through conditionals, while dashed edges denote comparison

```

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

```

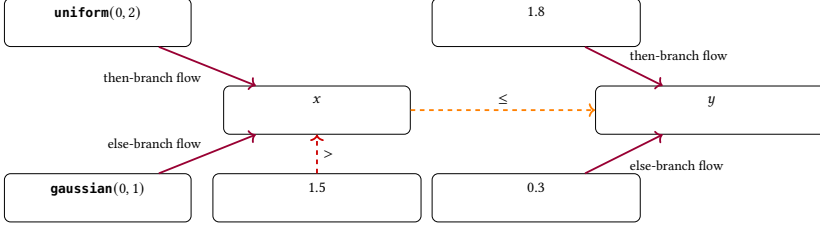


Fig. 17. Constraint graph for float types. Solid arrows represent data flow, and dashed arrows constraints.

constraints. In this example, the comparisons $1.5 > x$ and $x \leq y$ jointly determine the cut structure relevant to x . The data flow edges record how continuous samples from **uniform**(0, 2) and **gaussian**(0, 1) flow into x , while the constants 1.8 and 0.3 flow into y through its two branches.

The propagation process of float type information proceeds in four stages (Figure 18). First, literal constants contribute singleton value sets, while continuous samples are assigned value set $\top$ (Figure 18a). Second, value sets propagate forward along data-flow edges, yielding in particular that y has value set $\{0.3, 1.8\}$ (Figure 18b). Third, comparison sites generate cut information: $1.5 > x$ contributes the threshold 1.5, and $x \leq y$ contributes the possible values of y , namely 0.3 and 1.8 (Figure 18c). Since both operands of a comparison must be discretized against compatible interval boundaries, these cut sets are unified across the compared expressions. Finally, the resulting cut set propagates backward along data flow edges to all contributing subexpressions (Figure 18d).

The final cut set is

$$\{\leq 0.3, < 1.5, \leq 1.8\},$$

inducing the intervals $(-\infty, 0.3]$, $(0.3, 1.5)$, $[1.5, 1.8]$, and $(1.8, \infty)$.

A.2 Discrete Latent Variable

We revisit the discrete latent variable example from §2.4:

```

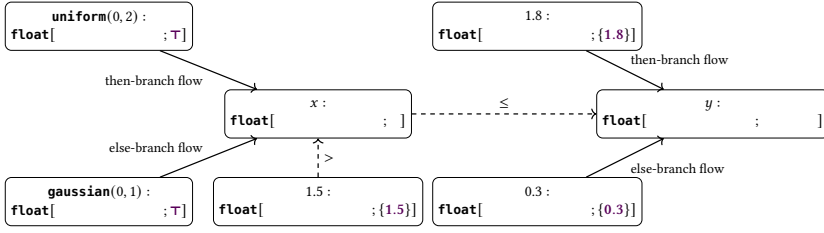
let x = if flip() then 0.5 else 1.5 in gaussian(0, x) < 0.5

```

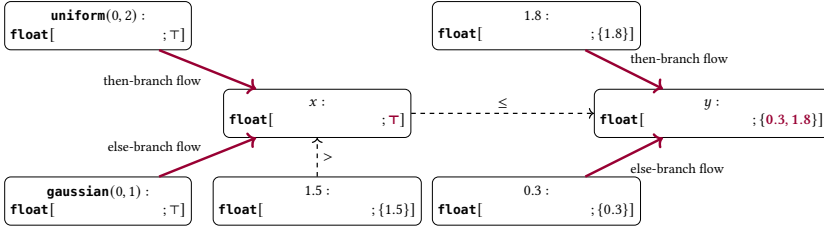
The essential feature here is that the parameter of the continuous distribution is itself determined by a discrete choice. Thus, the discretization of the program must also account for the fact that the Gaussian is instantiated at two distinct parameter values.

Figure 19 presents the corresponding constraint graph. The variable x is formed by control flow from the two constants 0.5 and 1.5, and therefore has value set $\{0.5, 1.5\}$. Because x appears as a parameter to **gaussian**(0, x), SLICE must recover which concrete value of x was selected in order to instantiate the appropriate Gaussian distribution. The figure records this dependency by the *recovers* constraint on x .

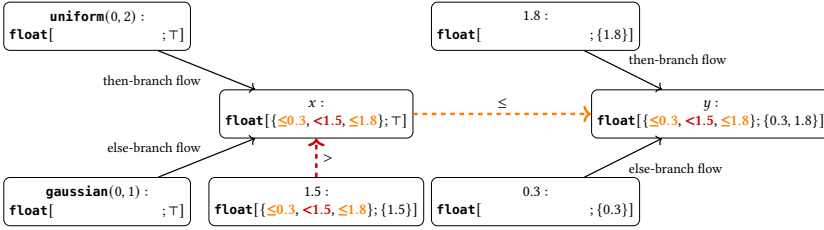
Operationally, the *recovers* constraint requires that the values in the value set of x be represented by distinct discrete indices. This is why the type system uses non-strict cuts for such values: the cut set $\{\leq 0.5, \leq 1.5\}$ induces a partition in which 0.5 and 1.5 occupy distinguishable regions, allowing them to be mapped to separate cases in the discretized program. The final comparison **gaussian**(0, x) < 0.5 contributes the cut <0.5 for the sampled Gaussian result, while the value-set information on x determines which discretized Gaussian is selected.



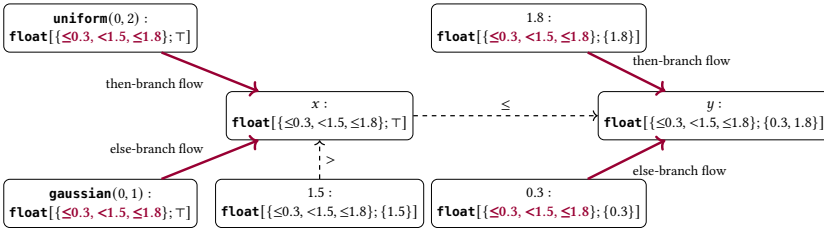
(a) *Initialize value sets.* Literal floats are annotated with their numeric value, and continuous distributions with $\mathcal{T}$.



(b) *Propagate value sets (forward along data flow edges).* Any expression formed by an if-expression inherits the union of the value sets of its branches.



(c) *Generate cut sets.* Comparisons contribute threshold values (e.g., 1.5, 0.3, 1.8) to the cut sets of participating expressions. Cut sets of expressions on both sides of a comparison (e.g. $1.5 > x$ and $x \leq y$) are unified to ensure consistent discretization.



(d) *Propagate cut sets (backward along data flow edges).* Cuts introduced at comparison sites are pushed backward along the edges in Figure 18b so that all contributing subexpressions share the same interval boundaries for discretization.

Fig. 18. Constraint graphs showing value-set flow and cut-set propagation over float-typed subexpressions for example §2.3. Colored elements indicate the components of the float types activated at each step.

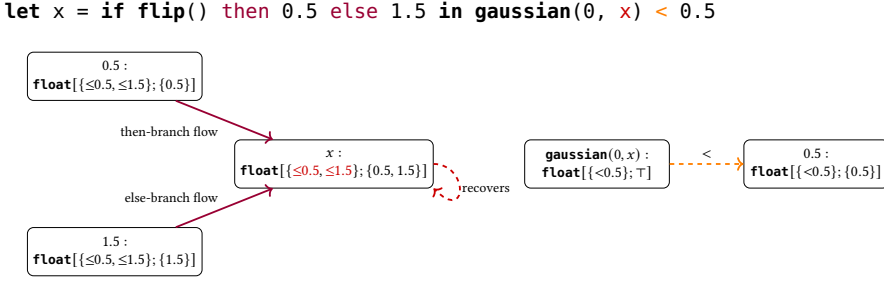


Fig. 19. Constraint graph populated with float type information. As before, solid arrows represent data flow, and dashed arrows represent comparison constraints. Using x as a Gaussian parameter induces a *recovers* constraint, the red dashed self-loop, which ensures a unique discrete index for each value of x .

A.3 Indian GPA Problem

This 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. SLICE correctly handles conditioning on a positive-measure atom at 10:

```

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 [{<10, ≤10}; {10}])
  else
    (uniform(0, 10) : float [{<10, ≤10}; T])
  else
    if perfect ≤ 0 then
      (4.0 : float [{<10, ≤10}; {4}])
    else
      (uniform(0, 4) : float [{<10, ≤10}; T])
  : float [{<10, ≤10}; T] in
let _ = observe(gpa : float [{<10, ≤10}; T] ≤ 10 : float [{<10, ≤10}; {10}]
  && gpa : float [{<10, ≤10}; T] ≥ 10 : float [{<10, ≤10}; {10}]) in
nationality > 0

```

The type system propagates the cuts <10 and ≤ 10 for all GPA expressions. These cuts induce the three regions:

$$(-\infty, 10), \quad \{10\}, \quad (10, \infty).$$

Under this partition, both continuous branches, $\text{uniform}(0, 10)$ and $\text{uniform}(0, 4)$, place all of their probability mass in the first region. The constant 10.0 is mapped to the singleton region $\{10\}$, while the constant 4.0 is mapped to the region $(-\infty, 10)$. The resulting discretized program is therefore:

```

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(1.0, 0.0, 0.0)

```

```

else
  if perfect <= 0 then 0
  else
    discrete(1.0, 0.0, 0.0) in
  let _ = observe(gpa <= 1 && gpa >= 1)
  nationality > 0

```

After discretization, the equality test `gpa = 10` becomes the discrete test that the GPA index is exactly the one corresponding to the singleton region $\{10\}$.

A.4 Point and Measure-Zero Observations

SLICE can condition on an exact value when that event has strictly positive probability mass, as in the Indian GPA example in §A.3. In that case, conditioning is well-defined and SLICE returns the correct answer. The exact set of programs supported by SLICE is therefore somewhat larger than one would initially expect, while not covering full point observations on continuous samples.

The mechanism is the same as in §A.3: the cuts $<c$ and $\leq c$ induce the partition

$$(-\infty, c), \quad \{c\}, \quad (c, \infty),$$

so any probability mass assigned exactly to c is tracked as its own discrete outcome.

More precisely:

- (1) SLICE cannot handle programs where every control-flow path conditions on an event of measure zero, and therefore programs with exact observations on continuous samples are in general out of scope, i.e. the programs one typically writes in e.g. Stan are out of scope unless one widens exact conditioning to a small but finite interval.
- (2) SLICE can handle programs where at least one control-flow path conditions only on strictly-positive-probability events, even if that is a point observation, and even if other paths condition only on measure-zero events.
- (3) It does not matter whether an event is measure-zero because it is point-conditioning on a continuous sample, or whether it is measure-zero for another reason, e.g. `observe(uniform(0, 1) > 3)` or simply `observe(false)`. The critical distinction for `observe(E)` is whether $\Pr(E) = 0$, not whether E is of the form “ $x = 3.2$ ”.
- (4) SLICE will never give the wrong answer, even when all paths condition on measure-zero events. It will give the empty subdistribution, indicating that under a rejection-sampling interpretation, all samples are rejected with probability 1. Thus the user is notified, and SLICE never silently gives a wrong answer.

For instance, this program:

```

let x = uniform(0,1) in
if flip() then
  let _ = observe(x >= 0.5 && x <= 0.5) in true
else
  let _ = observe(x <= 0.3 && x >= 0.3) in false

```

results in:

$$\Pr(\text{true}) = 0 \quad \Pr(\text{false}) = 0.$$

Whereas this program:

```

let x = uniform(0,1) in
if flip() then
  let _ = observe(x >= 0.5 && x <= 0.5) in true

```

```

(* Returns true if y is in [lo,hi] *)
let within = fun lo -> fun hi -> fun y ->
  (y >= lo) && (y < hi) in

(* Function for sampling interval bounds *)
let b = fun _ ->
  discrete(0.18: 0.05, 0.11: 0.24,
    0.21: 0.37, 0.09: 0.58,
    0.41: 0.89) in

(* Initialize intervals by sampling their
   respective (possibly overlapping) bounds *)
let init =
  (false, within 1 (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) 8) ::
  nil in

(* Update flags depending on y falling
   within the respective interval *)
let update = fix update y := fun ys ->
  match ys with
  | nil -> nil
  | h :: t ->
    ((fst h || ((snd h) y)), snd h)
    :: (update y t)
  end in

(* Sample from a discretized gaussian,
   then use the resulting value to update
   flags *)
let step = fun ys ->
  update (discrete(0.5, 0.0199388,
    0.0748961, 0.0494739,
    0.0747339, 0.0632619,
    0.0309625, 0.0280777,
    0.158655)) ys in

(* Number of samples determined by
   unbounded recursion *)
let cover = fix cover st :=
  if discrete(0.1, 0.9) < 1 then st
  else cover (step st) in

(* Have all intervals been covered at
   least once? *)
let is_covered = fix is_covered ys :=
  match ys with
  | nil -> true
  | h :: t -> (fst h) && (is_covered t)
  end in

(* Run entire process on sampled intervals *)
let final = cover init in
  is_covered final

```

Fig. 20. A *discretized* infinite stochastic process for random interval covering over $[0, 1]$, involving continuous sampling, higher-order recursion, and lists.

```

else
  let _ = observe(x <= 0.3001 && x >= 0.3) in false

```

results in:

$$\Pr(\text{true}) = 0 \quad \Pr(\text{false}) = 1.$$

This is the correct behavior: consider repeatedly running the program under a rejection-sampling interpretation until an observation succeeds. With probability 1, the first successful test will be **observe**($x \leq 0.3001$ && $x \geq 0.3$) and the program will therefore return false with probability 1.

A.5 Random Interval Covering Over $[0, 1]$

The discretized program for the case study from §2.6 is shown in Figure 20. SLICE takes 0.000s, and we generate a Markov chain consisting of 192365 states in 38.69 seconds. STORM takes 0.69 seconds for model checking, and reports that the sought-after probability is 0.034474....

B SLICE Case Studies (continued)

B.1 Random Interval Covering Over $[0, 1]$ with Conditioning

The full program for the case study from §7.1.2 is shown in Figure 21. The discretized program is shown in Figure 22.

```

(* Returns true if y is in [lo,hi] *)
let within = fun lo -> fun hi -> fun y ->
  (y >= lo) && (y < hi) in

(* Function for sampling interval bounds *)
let b = fun _ ->
  discrete(0.18: 0.05, 0.11: 0.24,
    0.21: 0.37, 0.09: 0.58,
    0.41: 0.89) in

(* Initialize intervals by sampling their
   respective (possibly overlapping) bounds *)
let init =
  (false, within 0.00 (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) 1.00) ::
  nil in

(* Update flags depending on y falling
   within the respective interval *)
let update = fix update y := fun ys ->
  match ys with
  | nil -> nil
  | h :: t ->
    ((fst h || ((snd h) y)), snd h)
    :: (update y t)
  end in

(* Ensure sampled y hits at most one
   interval *)
let non_overlap =
  fix non_overlap y := fun seen -> fun ys ->
    match ys with
    | nil -> true
    | h :: t ->
      let hit = (snd h) y in
      if hit && seen then false
      else non_overlap y (seen || hit) t
    end in

(* Use the sampled value to update flags *)
let step = fun y -> fun ys ->
  update y ys in

(* Number of samples determined by
   unbounded recursion *)
let cover = fix cover st :=
  if uniform(0,1) < 0.1 then st
  else
    let y = gaussian(0,1) in
    let _ = observe(non_overlap y false st) in
    cover (step y st) in

(* Have all intervals been covered
   at least once? *)
let is_covered = fix is_covered ys :=
  match ys with
  | nil -> true
  | h :: t -> (fst h) && (is_covered t)
  end in

(* Run the process and query whether
   the whole interval was covered *)
let final = cover init in
is_covered final

```

Fig. 21. An infinite stochastic process for random interval covering over $[0, 1]$, involving conditioning, continuous sampling, higher-order recursion, and lists.

B.2 Bloom Filter

The full program for the case study from §7.1.3 is shown in Figure 23. The discretized program is shown in Figure 24.

```

(* Returns true if y is in [lo,hi] *)
let within = fun lo -> fun hi -> fun y ->
  (y >= lo) && (y < hi) in

(* Function for sampling interval bounds *)
let b = fun _ ->
  discrete(0.18: 0.05, 0.11: 0.24,
    0.21: 0.37, 0.09: 0.58,
    0.41: 0.89) in

(* Initialize intervals by sampling their
   respective (possibly overlapping) bounds *)
let init =
  (false, within 1 (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) (b ())) ::
  (false, within (b ()) 8) ::
  nil in

(* Update flags depending on y falling
   within the respective interval *)
let update = fix update y := fun ys ->
  match ys with
  | nil -> nil
  | h :: t ->
    ((fst h || (snd h) y), snd h)
    :: (update y t)
  end in

(* Ensure sampled y hits at most one
   interval *)
let non_overlap =
  fix non_overlap y := fun seen -> fun ys ->
    match ys with
    | nil -> true
    | h :: t ->
      let hit = (snd h) y in
      if hit && seen then false
      else non_overlap y (seen || hit) t
    end in

(* Use the sampled value to update flags *)
let step = fun y -> fun ys ->
  update y ys in

(* Number of samples determined by
   unbounded recursion *)
let cover = fix cover st :=
  if discrete(0.1, 0.9) < 1 then st
  else
    let y = discrete(0.5, 0.0199388,
      0.0748961, 0.0494739,
      0.0747339, 0.0632619,
      0.0309625, 0.0280777,
      0.158655) in
    let _ = observe(non_overlap y false st) in
    cover (step y st) in

(* Have all intervals been covered
   at least once? *)
let is_covered = fix is_covered ys :=
  match ys with
  | nil -> true
  | h :: t -> (fst h) && (is_covered t)
  end in

(* Run the process and query whether
   the whole interval was covered *)
let final = cover init in
is_covered final

```

Fig. 22. A discretized infinite stochastic process for random interval covering over $[0, 1]$, involving conditioning, continuous sampling, higher-order recursion, and lists.

```

1  (* Set bf[idx] := true, where idx is 0..3 and
2  bf = (b0, (b1, (b2, b3))) *)
3  let set_true = fun bf -> fun idx ->
4    let b0 = fst bf in
5    let tail1 = snd bf in
6    let b1 = fst tail1 in
7    let tail2 = snd tail1 in
8    let b2 = fst tail2 in
9    let b3 = snd tail2 in
10   if idx <= 0 then
11     (true, (b1, (b2, b3)))
12   else if idx <= 1 then
13     (b0, (true, (b2, b3)))
14   else if idx <= 2 then
15     (b0, (b1, (true, b3)))
16   else
17     (b0, (b1, (b2, true))) in
18
19  (* Get bf[idx], where idx is 0..3 *)
20  let get = fun bf -> fun idx ->
21    let b0 = fst bf in
22    let tail1 = snd bf in
23    let b1 = fst tail1 in
24    let tail2 = snd tail1 in
25    let b2 = fst tail2 in
26    let b3 = snd tail2 in
27    if idx <= 0 then b0
28    else if idx <= 1 then b1
29    else if idx <= 2 then b2
30    else b3 in
31
32  (* Insert element q into bf using hash
33  functions h1 and h2 *)
34  let insert = fun q -> fun h1
35    -> fun h2 -> fun bf ->
36    let index1 = h1 q in
37    let index2 = h2 q in
38    let bf1 = set_true bf index1 in
39    let bf2 = set_true bf1 index2 in
40    bf2 in
41
42  (* Search for q in bf using hash
43  functions h1 and h2 *)
44  let search = fun q -> fun h1
45    -> fun h2 -> fun bf ->
46    let index1 = h1 q in
47    let index2 = h2 q in
48    let b01 = get bf index1 in
49    let b02 = get bf index2 in
50    b01 && b02 in
51
52  (* Hash functions *)
53  let h1 = fun q ->
54    if q <= -2 then 0
55    else if -2 < q && q <= 0 then 1
56    else if 0 < q && q <= 2 then 2
57    else 3 in
58
59  let h2 = fun q ->
60    if q <= -2 then 3
61    else if -2 < q && q <= 0 then 2
62    else if 0 < q && q <= 2 then 1
63    else 0 in
64
65  (* Test whether the Bloom filter is full *)
66  let full = fun bf ->
67    let b0 = fst bf in
68    let t1 = snd bf in
69    let b1 = fst t1 in
70    let t2 = snd t1 in
71    let b2 = fst t2 in
72    let b3 = snd t2 in
73    b0 && b1 && b2 && b3 in
74
75  (* Initial, empty Bloom filter *)
76  let init = (false, (false, (false, false))) in
77
78  (* Query key *)
79  let key = gaussian(0,1) in
80
81  (* Insert gaussian samples until bf is full
82  or the key is reported present *)
83  let search_before_full = fix loop bf :=
84    if full bf then bf
85    else if search key h1 h2 bf then bf
86    else
87      let q = gaussian(0,1) in
88      loop (insert q h1 h2 bf) in
89
90  (* Query whether search terminated before
91  the filter became full *)
92  let bf_final = search_before_full init in
93  not (full bf_final)

```

Fig. 23. A 4-bit Bloom filter program with continuous samples, locality-sensitive hash functions, and search and insert functions.

```

(* Set bf[idx] := true, where idx is 0..3 and
   bf = (b0, (b1, (b2, b3))) *)
let set_true = fun bf -> fun idx ->
  let b0 = fst bf in
  let tail1 = snd bf in
  let b1 = fst tail1 in
  let tail2 = snd tail1 in
  let b2 = fst tail2 in
  let b3 = snd tail2 in
  if idx <= 0 then
    (true, (b1, (b2, b3)))
  else if idx <= 1 then
    (b0, (true, (b2, b3)))
  else if idx <= 2 then
    (b0, (b1, (true, b3)))
  else
    (b0, (b1, (b2, true))) in

(* Get bf[idx], where idx is 0..3 *)
let get = fun bf -> fun idx ->
  let b0 = fst bf in
  let tail1 = snd bf in
  let b1 = fst tail1 in
  let tail2 = snd tail1 in
  let b2 = fst tail2 in
  let b3 = snd tail2 in
  if idx <= 0 then b0
  else if idx <= 1 then b1
  else if idx <= 2 then b2
  else b3 in

(* Insert element q into bf using hash
   functions h1 and h2 *)
let insert = fun q -> fun h1
  -> fun h2 -> fun bf ->
  let index1 = h1 q in
  let index2 = h2 q in
  let bf1 = set_true bf index1 in
  let bf2 = set_true bf1 index2 in
  bf2 in

(* Search for q in bf using hash
   functions h1 and h2 *)
let search = fun q -> fun h1
  -> fun h2 -> fun bf ->
  let index1 = h1 q in
  let index2 = h2 q in
  let b01 = get bf index1 in
  let b02 = get bf index2 in
  b01 && b02 in

(* Hash functions *)
let h1 = fun q ->
  if q <= 0 then 0
  else if 0 < q && q <= 1 then 1
  else if 1 < q && q <= 2 then 2
  else 3 in
let h2 = fun q ->
  if q <= 0 then 3
  else if 0 < q && q <= 1 then 2
  else if 1 < q && q <= 2 then 1
  else 0 in

(* Test whether the Bloom filter is full *)
let full = fun bf ->
  let b0 = fst bf in
  let t1 = snd bf in
  let b1 = fst t1 in
  let t2 = snd t1 in
  let b2 = fst t2 in
  let b3 = snd t2 in
  b0 && b1 && b2 && b3 in

(* Initial, empty Bloom filter *)
let init = (false, (false, (false, false))) in

(* Query key *)
let key = discrete(0.0227501,
  0.47725,
  0.47725,
  0.0227501) in

(* Insert discretized gaussian samples until
   bf is full or the key is reported
   present *)
let search_before_full = fix loop bf :=
  if full bf then bf
  else if search key h1 h2 bf then bf
  else
    let q = discrete(0.0227501, 0.47725,
      0.47725, 0.0227501) in
    loop (insert q h1 h2 bf) in

(* Query whether search terminated before
   the filter became full *)
let bf_final = search_before_full init in
not (full bf_final)

```

Fig. 24. A discretized 4-bit Bloom filter program with continuous samples, locality-sensitive hash functions, and search and insert functions.

Source constructor	Density or PMF	Support	Parameter domain and convention
<i>Continuous distributions</i>			
exponential(λ)	$\lambda e^{-\lambda x}$	$[0, \infty)$	Rate $\lambda > 0$
uniform(a, b)	$\frac{1}{b-a} \mathbf{1}_{[a,b)}(x)$	$[a, b]$	Endpoints $a < b$
laplace(b)	$\frac{1}{2b} e^{- x /b}$	$\mathbb{R}$	Centered; scale $b > 0$
gaussian(μ, σ)	$\frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)}$	$\mathbb{R}$	Location $\mu \in \mathbb{R}$; standard deviation $\sigma > 0$
cauchy(γ)	$\frac{\gamma}{\pi(x^2 + \gamma^2)}$	$\mathbb{R}$	Centered; scale $\gamma > 0$
beta(α, β)	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$	$[0, 1]$	Shapes $\alpha, \beta > 0$
tdist(ν)	$\frac{\Gamma((\nu+1)/2)}{\sqrt{\nu\pi} \Gamma(\nu/2)} (1 + x^2/\nu)^{-(\nu+1)/2}$	$\mathbb{R}$	Centered; degrees of freedom $\nu > 0$
lognormal(μ, σ)	$\frac{1}{x\sigma\sqrt{2\pi}} e^{-(\log x - \mu)^2/(2\sigma^2)}$	$[0, \infty)$	Log-location $\mu \in \mathbb{R}$; log-scale $\sigma > 0$
chi2(ν)	$\frac{x^{\nu/2-1} e^{-x/2}}{2^{\nu/2} \Gamma(\nu/2)}$	$[0, \infty)$	Real degrees of freedom $\nu > 0$
gamma(α, θ)	$\frac{x^{\alpha-1} e^{-x/\theta}}{\Gamma(\alpha) \theta^\alpha}$	$[0, \infty)$	Shape $\alpha > 0$; scale $\theta > 0$
logistic(s)	$\frac{e^{-x/s}}{s(1+e^{-x/s})^2}$	$\mathbb{R}$	Centered; scale $s > 0$
pareto(a, b)	$ab^a x^{-a-1}$	$[b, \infty)$	GSL shape $a > 0$; threshold/scale $b > 0$
rayleigh(σ)	$\frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)}$	$[0, \infty)$	Scale $\sigma > 0$
weibulll(a, b)	$\frac{b}{a} (x/a)^{b-1} e^{-(x/a)^b}$	$[0, \infty)$	GSL scale $a > 0$; exponent $b > 0$
gumbell1(a, b)	$ab e^{-(be^{-ax} + ax)}$	$\mathbb{R}$	GSL parameters $a, b > 0$
gumbell2(a, b)	$ab x^{-a-1} e^{-bx^{-a}}$	$[0, \infty)$	GSL shape $a > 0$; coefficient $b > 0$
exppow(a, b)	$\frac{1}{2a\Gamma(1+1/b)} e^{-(x /a)^b}$	$\mathbb{R}$	Scale $a > 0$; exponent $b > 0$
<i>Discrete distributions</i>			
poisson(μ)	$e^{-\mu} \mu^k / k!$	$\mathbb{N}_0$ if $\mu > 0$; $\{0\}$ if $\mu = 0$	Mean $\mu \geq 0$
binomial(p, n)	$\binom{n}{k} p^k (1-p)^{n-k}$	$\{0, \dots, n\}$, with endpoint degeneracies	$0 \leq p \leq 1$; integer $0 \leq n \leq \min(2^{32} - 1, \text{max_int})$

Fig. 25. List of supported distributions. The binomial support is $\{0\}$ when $n = 0$ or $p = 0$, and $\{n\}$ when $p = 1$; otherwise it is $\{0, \dots, n\}$.

C List of Supported SLICE Distributions

The list of distributions supported by SLICE, along with their probability density / mass functions, is given in Figure 25. SLICE supports discrete distributions by applying the same cut-set and value-set analysis used for continuous distributions, *provided that the discrete distribution admits a well-defined CDF*. The extension to discrete distributions does not introduce any fundamental changes to the discretization algorithm.

D Typing Rules for Full SLICE Language

The typing rules for general language constructs is given in Figure 26.

VAR	LET	IF
$\frac{}{\Gamma, x : \tau \vdash x : \tau}$	$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2}$	$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 : \tau}$
DISCRETE	LESS EQ	
$\frac{n \geq 1 \quad \forall i \in \{0, \dots, n-1\}. p_i \geq 0 \quad \sum_{i=0}^{n-1} p_i = 1}{\Gamma \vdash \mathbf{discrete}(p_0, \dots, p_{n-1}) : \mathbf{fin}(n)}$	$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash e \leq i : \mathbf{bool}}$	
PAIR	FST	SND
$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$	$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{fst} \ e : \tau_1}$	$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{snd} \ e : \tau_2}$
		FUN
		$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun} \ x \rightarrow e : \tau_1 \rightarrow \tau_2}$
APP	TRUE	FALSE
$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}$	$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}$	$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}$
AND	OR	NOT
$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \mathbf{bool}}$	$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool}}{\Gamma \vdash e_1 \ \ e_2 : \mathbf{bool}}$	$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \mathbf{not} \ e : \mathbf{bool}}$
UNIT	FINCONST	FINLESS
$\frac{}{\Gamma \vdash () : \mathbf{unit}}$	$\frac{0 \leq k < n}{\Gamma \vdash k_{\#n} : \mathbf{fin}(n)}$	$\frac{\Gamma \vdash e_1 : \mathbf{fin}(n) \quad \Gamma \vdash e_2 : \mathbf{fin}(n)}{\Gamma \vdash e_1 <_{\#n} e_2 : \mathbf{bool}}$
FINLESS EQ	OBSERVE	SEQ
$\frac{\Gamma \vdash e_1 : \mathbf{fin}(n) \quad \Gamma \vdash e_2 : \mathbf{fin}(n)}{\Gamma \vdash e_1 \leq_{\#n} e_2 : \mathbf{bool}}$	$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash \mathbf{observe} \ e : \mathbf{unit}}$	$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 ; e_2 : \tau_2}$
FIX	NIL	CONS
$\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fix} \ f \ x := e : \tau_1 \rightarrow \tau_2}$	$\frac{}{\Gamma \vdash \mathbf{nil} : \mathbf{list}(\tau)}$	$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathbf{list}(\tau)}{\Gamma \vdash e_1 :: e_2 : \mathbf{list}(\tau)}$
MATCH		DIVERGE
$\frac{\Gamma \vdash e : \mathbf{list}(\tau_1) \quad \Gamma \vdash e_1 : \tau_2 \quad \Gamma, h : \tau_1, t : \mathbf{list}(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{match} \ e \ \mathbf{with} \ \mathbf{nil} \rightarrow e_1 \mid h :: t \rightarrow e_2 \ \mathbf{end} : \tau_2}$		$\frac{}{\Gamma \vdash \mathbf{diverge} : \tau}$

Fig. 26. Typing rules for general language constructs

E Discretization Proof

E.1 Well-Typedness of Discretization

PROOF OF THEOREM 5.4. We proceed by induction on the structure of the typing derivation $\Gamma \vdash e : \tau$, matching each case of the discretization function $\mathcal{D}[\![\cdot]\!]$ defined in Figure 13.

Case $e = x$.

Since $\Gamma(x) = \tau$, we have $\mathcal{D}[\![\Gamma]\!](x) = \mathcal{D}[\![\tau]\!]$. Moreover, $\mathcal{D}[\![x : \tau]\!] = x$, so $\mathcal{D}[\![\Gamma]\!] \vdash x : \mathcal{D}[\![\tau]\!]$.

Case $e = (c : \mathbf{float}[B; V])$.

The FLOAT-CONST rule was applied, so $\text{has}(V, c)$ holds.

- If $B = \top$: then $\mathcal{D}[\![c : \mathbf{float}[B; V]]\!] = c$ and $\mathcal{D}[\![\mathbf{float}[B; V]]\!] = \mathbf{float}[B; V]$. Since $\text{has}(V, c)$ still holds, c is well-typed at $\mathbf{float}[B; V]$ by FLOAT-CONST.
- If $B \neq \top$: then $\mathcal{D}[\![c : \mathbf{float}[B; V]]\!] = k_{\#|B|+1}$ for the unique k such that $c \in \text{Intervals}(B)_k$, and $\mathcal{D}[\![\mathbf{float}[B; V]]\!] = \mathbf{fin}(|B| + 1)$. Since $0 \leq k < |B| + 1$, the finite constant $k_{\#|B|+1}$ is well-typed at $\mathbf{fin}(|B| + 1)$ by FINCONST.

Case $e = (\mathbf{let} \ x = e_1 : \tau_1 \ \mathbf{in} \ e_2 : \tau_2)$.

By the inductive hypothesis,

$$\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![e_1 : \tau_1]\!] : \mathcal{D}[\![\tau_1]\!] \quad \text{and} \quad \mathcal{D}[\![\Gamma, x : \tau_1]\!] \vdash \mathcal{D}[\![e_2 : \tau_2]\!] : \mathcal{D}[\![\tau_2]\!].$$

Note that $\mathcal{D}[\![\Gamma, x : \tau_1]\!] = \mathcal{D}[\![\Gamma]\!], x : \mathcal{D}[\![\tau_1]\!]$. Applying LET gives

$$\mathcal{D}[\![\Gamma]\!] \vdash \mathbf{let} \ x = \mathcal{D}[\![e_1 : \tau_1]\!] \ \mathbf{in} \ \mathcal{D}[\![e_2 : \tau_2]\!] : \mathcal{D}[\![\tau_2]\!].$$

Case $e = (\mathbf{if} \ e_1 : \mathbf{bool} \ \mathbf{then} \ e_2 : \tau \ \mathbf{else} \ e_3 : \tau)$.

By the inductive hypothesis, $\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![e_i : \tau_i]\!] : \mathcal{D}[\![\tau_i]\!]$ for $i = 1, 2, 3$, where $\tau_1 = \mathbf{bool}$ and $\tau_2 = \tau_3 = \tau$. Since $\mathcal{D}[\![\mathbf{bool}]\!] = \mathbf{bool}$, applying IF gives

$$\mathcal{D}[\![\Gamma]\!] \vdash \mathbf{if} \ \mathcal{D}[\![e_1 : \mathbf{bool}]\!] \ \mathbf{then} \ \mathcal{D}[\![e_2 : \tau]\!] \ \mathbf{else} \ \mathcal{D}[\![e_3 : \tau]\!] : \mathcal{D}[\![\tau]\!].$$

Case $e = (e_1 : \mathbf{float}[B; V_1] < e_2 : \mathbf{float}[B; V_2])$.

The LESS rule was applied, so both subexpressions share the same cut set B and $\text{answers}_{<}(B, V_1, V_2)$ holds. By the inductive hypothesis, $\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![e_i : \mathbf{float}[B; V_i]]\!] : \mathcal{D}[\![\mathbf{float}[B; V_i]]\!]$ for $i = 1, 2$.

- If $B = \top$: then $\mathcal{D}[\![\mathbf{float}[B; V_i]]\!] = \mathbf{float}[B; V_i]$ for both i , and the standard float comparison gives type $\mathbf{bool} = \mathcal{D}[\![\mathbf{bool}]\!]$.
- If $B \neq \top$: then both subexpressions are discretized to type $\mathbf{fin}(|B| + 1)$, and the finite comparison $<_{\#(|B|+1)}$ is well-typed at $\mathbf{bool} = \mathcal{D}[\![\mathbf{bool}]\!]$ by FINLESS.

Case $e = (e_1 : \mathbf{float}[B; V_1] \leq e_2 : \mathbf{float}[B; V_2])$.

Analogous to the $<$ case, using $\text{answers}_{\leq}(B, V_1, V_2)$ and FINLESSEQ.

Case $e = (\mathbf{uniform}(e_1 : \mathbf{float}[B_1; V_1], e_2 : \mathbf{float}[B_2; V_2]) : \mathbf{float}[B; \top])$.

The CONTINUOUS-2 rule was applied, so $\text{recovers}(B_1, V_1)$ and $\text{recovers}(B_2, V_2)$ hold. By the inductive hypothesis, $\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![e_i : \mathbf{float}[B_i; V_i]]\!] : \mathcal{D}[\![\mathbf{float}[B_i; V_i]]\!]$ for $i = 1, 2$. Also, $B_i \equiv_{\top} B$, so $B_i = \top$ iff $B = \top$. We consider each sub-case of the discretization function:

- $e_1, e_2 \in \mathbb{R}$ and $e_1 \geq e_2$: then $\mathcal{D}[\llbracket \text{uniform}(\dots) : \text{float}[B; \top] \rrbracket] = \text{diverge}$, which is well-typed at any type by DIVERGE.

- $B = \top$: then $\mathcal{D}[\llbracket \text{float}[B; \top] \rrbracket] = \text{float}[B; \top]$, and

$$\mathcal{D}[\llbracket \text{uniform}(e_1, e_2) : \text{float}[B; \top] \rrbracket] = \text{uniform}(\mathcal{D}[\llbracket e_1 : \text{float}[B_1; V_1] \rrbracket], \mathcal{D}[\llbracket e_2 : \text{float}[B_2; V_2] \rrbracket]).$$

Here $B_1 = B_2 = \top$, so the inductive hypotheses give float-typed arguments; CONTINUOUS-2 yields $\text{float}[\top; \top]$.

- $B \neq \top$ and $e_1, e_2 \in \mathbb{R}$: then $\mathcal{D}[\llbracket \text{float}[B; \top] \rrbracket] = \text{fin}(|B| + 1)$, and

$$\mathcal{D}[\llbracket \text{uniform}(e_1, e_2) : \text{float}[B; \top] \rrbracket] = \text{discrete}(p_0, \dots, p_{|B|})$$

where $p_k = \text{CDF}(b_{k+1}) - \text{CDF}(b_k)$. Each $p_k \in [0, 1]$ and $\sum_k p_k = 1$, so this is well-typed at $\text{fin}(|B| + 1)$ by DISCRETE.

- If $B, B_1, B_2 \neq \top$ and $V_1 = \emptyset$ or $V_2 = \emptyset$, then the enumeration over $V_1 \times V_2$ is empty, so the generated expression evaluates $\mathcal{D}[\llbracket e_1 \rrbracket]$ and $\mathcal{D}[\llbracket e_2 \rrbracket]$ through its two outer **let**-bindings and then yields **diverge**; the bindings are well-typed by the inductive hypotheses and weakening, and their body is well-typed at $\text{fin}(|B| + 1)$ by DIVERGE.
- $B, B_1, B_2 \neq \top$, $V_1, V_2 \neq \emptyset$, and $e_1 \notin \mathbb{R}$ or $e_2 \notin \mathbb{R}$: the algorithm generates conditionals over all pairs (v_i, w_j) , each branch dispatching to a constant parameter in **uniform**. The inductive hypotheses and weakening type the two outer **let**-bindings. In each guard, both operands of each finite equality have the same finite type; since finite equality is desugared using **FINLESSEQ**, each equality has type **bool**, as does their conjunction. By a previous subcase, each branch $\mathcal{D}[\llbracket \text{uniform}(v_i, w_j) : \text{float}[B; \top] \rrbracket]$ is well-typed at $\text{fin}(|B| + 1) = \mathcal{D}[\llbracket \text{float}[B; \top] \rrbracket]$. The final **else diverge** branch has the same type by DIVERGE; hence the conditional structure is well-typed by **If** (as in the **if** case above), and the full expression is well-typed at $\mathcal{D}[\llbracket \text{float}[B; \top] \rrbracket]$.

Case a one-argument continuous distribution. Analogous, using $B_1 \equiv_{\top} B$.

Case e is any other language construct (functions, pairs, lists, *etc.*).

By definition of $\mathcal{D}[\llbracket \cdot \rrbracket]$, the discretization recurses structurally over e and τ . That is, $\mathcal{D}[\llbracket e : \tau \rrbracket]$ is formed by applying $\mathcal{D}[\llbracket \cdot \rrbracket]$ to each subexpression and subtype of $e : \tau$, and $\mathcal{D}[\llbracket \tau \rrbracket]$ is formed by applying $\mathcal{D}[\llbracket \cdot \rrbracket]$ to each component type of τ . The result then follows immediately by applying the inductive hypothesis to each subterm, analogous to the above cases.

In every case, the typing derivation is preserved under $\mathcal{D}[\llbracket \cdot \rrbracket]$. Therefore,

$$\mathcal{D}[\llbracket \Gamma \rrbracket] \vdash \mathcal{D}[\llbracket e : \tau \rrbracket] : \mathcal{D}[\llbracket \tau \rrbracket].$$

□

F Soundness Proof

F.1 Measure Theory Preliminaries

We first give a brief overview of basic measure-theoretic terms and lemmas, and refer the reader to standard textbooks for further background [43, 44].

Topological space. A topological space is a pair $(X, \mathcal{T})$ where $\mathcal{T}$ is a collection of subsets of X , called open sets, such that $\emptyset, X \in \mathcal{T}$, and $\mathcal{T}$ is closed under arbitrary unions and finite intersections.

σ -algebra. A σ -algebra on a set X is a collection Σ of subsets of X that includes X itself and is closed under complement and under countable unions. By implication, a σ -algebra includes the empty set $\emptyset$ and is closed under countable intersections. Given a topological space $(X, \mathcal{T})$, the Borel σ -algebra $\mathcal{B}(X)$ is the smallest σ -algebra containing all open sets in $\mathcal{T}$.

Measurable space. A measurable space is a pair (X, Σ) consisting of a set X and a σ -algebra Σ on X . When Σ is the Borel σ -algebra on a topological space X , we call (X, Σ) a Borel measurable space.

Measure. Let (X, Σ) be a measurable space. A function $\mu : \Sigma \rightarrow [0, \infty]$ is called a measure if it satisfies the following properties: nonnegativity, $\mu(\emptyset) = 0$, and countable additivity. When Σ is a Borel σ -algebra, μ is called a Borel measure.

Measure space. The triple (X, Σ, μ) is called a measure space. When Σ is the Borel σ -algebra and μ is a Borel measure, (X, Σ, μ) is called a Borel measure space.

Subprobability measure. Let (X, Σ, μ) be a measure space. When $\mu(X) \leq 1$, μ is called a (Borel) subprobability measure. When $\mu(X) = 1$, μ is called a (Borel) probability measure and (X, Σ, μ) a (Borel) probability space.

Product measurable space. The product of two measurable spaces (X_1, Σ_1) and (X_2, Σ_2) is also a measurable space, namely $(X_1 \times X_2, \Sigma_1 \times \Sigma_2)$, where $\Sigma_1 \times \Sigma_2$ is the σ -algebra generated by sets of the form $A_1 \times A_2$ with $A_1 \in \Sigma_1$ and $A_2 \in \Sigma_2$.

Generated σ -algebra. Let X be a set and let $\mathcal{G} \subseteq \mathcal{P}(X)$ be a collection of subsets of X . The σ -algebra generated by $\mathcal{G}$, denoted $\sigma(\mathcal{G})$, is the smallest σ -algebra on X that contains $\mathcal{G}$. Equivalently, it is the intersection of all σ -algebras on X that contain $\mathcal{G}$.

Subprobability distribution. Let (X, Σ) be a measurable space. We write $\mathcal{D}(X)$ for the set of all subprobability measures on (X, Σ) .

Dirac measure. Let (X, Σ) be a measurable space and let $x \in X$. The *Dirac measure* at x is the probability measure

$$\delta_x : \Sigma \rightarrow [0, 1]$$

defined by

$$\delta_x(A) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A. \end{cases}$$

Measurable function. A function $f : X \rightarrow Y$ between measurable spaces $(X, \mathcal{F}_X)$ and $(Y, \mathcal{F}_Y)$ is called measurable if the preimage of any measurable set in Y is measurable in X , i.e. if

$$f^{-1}(A) = \{x \in X \mid f(x) \in A\} \in \mathcal{F}_X$$

for all $A \in \mathcal{F}_Y$.

Definition F.1 (Subspace σ -algebra). Let (X, Σ) be a Borel measurable space and let $Y \subseteq X$. The subspace (or trace) σ -algebra on Y is defined as

$$\Sigma_Y = \{Y \cap A \mid A \in \Sigma\}.$$

Then (Y, Σ_Y) is a Borel measurable space.

Definition F.2 (σ -algebra generated by a function). Let $f : X \rightarrow Y$ be a function and let (Y, Σ_Y) be a measurable space. The σ -algebra generated by f is defined by:

$$f^{-1}(\Sigma_Y) := \{f^{-1}(A) \mid A \in \Sigma_Y\}.$$

Then $(X, f^{-1}(\Sigma_Y))$ is a measurable space.

LEMMA F.3 (BIJECTION INDUCES ISOMORPHISM). *Let $f : X \rightarrow Y$ be a bijection and let (Y, Σ_Y) be a measurable space. Equip X with the σ -algebra generated by f :*

$$\Sigma_X := f^{-1}(\Sigma_Y) = \{f^{-1}(A) \mid A \in \Sigma_Y\}.$$

Then,

- (1) (X, Σ_X) is a measurable space, and
- (2) f induces an isomorphism of measurable spaces:

$$(X, \Sigma_X) \cong (Y, \Sigma_Y).$$

PROOF. We first verify that i) (X, Σ_X) is a measurable space, then show that ii) f induces an isomorphism of measurable spaces $(X, \Sigma_X) \cong (Y, \Sigma_Y)$.

Proof of i) We show that $\Sigma_X = f^{-1}(\Sigma_Y)$ is a σ -algebra on X .

- **(Contains X):** $X = f^{-1}(Y) \in \Sigma_X$ since $Y \in \Sigma_Y$.
- **(Complements):** If $f^{-1}(A) \in \Sigma_X$, then $X \setminus f^{-1}(A) = f^{-1}(Y \setminus A) \in \Sigma_X$ since $Y \setminus A \in \Sigma_Y$.
- **(Countable unions):** If $f^{-1}(A_i) \in \Sigma_X$ for $i \in \mathbb{N}$, then $\bigcup_i f^{-1}(A_i) = f^{-1}(\bigcup_i A_i) \in \Sigma_X$ since $\bigcup_i A_i \in \Sigma_Y$.

Hence (X, Σ_X) is a measurable space.

Proof of ii) Two measurable spaces (X, Σ_X) and (Y, Σ_Y) are isomorphic iff there exists a bijection $f : X \rightarrow Y$ such that f and f^{-1} are measurable.

We first show that f is measurable. Measurability of f means that $f^{-1}(A) \in \Sigma_X$ for every $A \in \Sigma_Y$. Let $A \in \Sigma_Y$ be arbitrary. By definition of Σ_X ,

$$f^{-1}(A) \in f^{-1}(\Sigma_Y) = \Sigma_X.$$

It follows that f is measurable.

Next we show that f^{-1} is measurable. Measurability of f^{-1} means that $(f^{-1})^{-1}(B) \in \Sigma_Y$ for every $B \in \Sigma_X$. Let $B \in \Sigma_X$ be arbitrary. By definition of Σ_X , there exists some $A \in \Sigma_Y$ such that $B = f^{-1}(A)$.

Observe that:

$$\begin{aligned} (f^{-1})^{-1}(B) &= f(B) && \text{(preimage of } f^{-1} \text{ is } f) \\ &= f(f^{-1}(A)) && \text{(substituting } B = f^{-1}(A)) \\ &= A && \text{(} f \text{ is a bijection)} \end{aligned}$$

Since $A \in \Sigma_Y$, it follows that f^{-1} is measurable.

Therefore f is a measurable bijection with measurable inverse, so it is an isomorphism of measurable spaces. $\square$

LEMMA F.4 (MINIMALITY OF GENERATED σ -ALGEBRAS). *Let X be a set and let $\mathcal{G} \subseteq \mathcal{P}(X)$. Suppose C is a σ -algebra on X such that $\mathcal{G} \subseteq C$. Then*

$$\sigma(\mathcal{G}) \subseteq C.$$

PROOF. Recall that $\sigma(\mathcal{G})$ is defined as the smallest σ -algebra on X containing $\mathcal{G}$, given concretely by the intersection of all σ -algebras on X containing $\mathcal{G}$:

$$\sigma(\mathcal{G}) = \bigcap_{\substack{\mathcal{F} \text{ is a } \sigma\text{-algebra on } X \\ \mathcal{G} \subseteq \mathcal{F}}} \mathcal{F}.$$

Since C is a σ -algebra on X containing $\mathcal{G}$, it appears in this intersection. Therefore,

$$\sigma(\mathcal{G}) \subseteq C.$$

□

Definition F.5 (Sub-Markov kernels). Given measurable spaces (X, Σ_X) and (Y, Σ_Y) , a *sub-Markov kernel* from X to Y is a function

$$P : X \times \Sigma_Y \rightarrow [0, 1]$$

such that:

- (1) for every $x \in X$, $P(x, -)$ is a subprobability measure on Y ;
- (2) for every $A \in \Sigma_Y$, the map $x \mapsto P(x, A)$ is measurable.

A Markov kernel is a sub-Markov kernel satisfying $P(x, Y) = 1$ for every $x \in X$.

Definition F.6 (Projection with error). Let $(X + \perp, \Sigma_{X+\perp})$ and $(Y + \perp, \Sigma_{Y+\perp})$ be measurable spaces, with (X, Σ_X) and (Y, Σ_Y) measurable spaces as well. Let $\mu \in \mathcal{D}((X \times Y) + \perp)$. We define the first and second projection measures as:

- (1) $\pi_1(\mu)(A) := \mu(A \times Y)$ for every measurable $A \in \Sigma_X$, and $\pi_1(\mu)(\perp) := \mu(\perp)$
- (2) $\pi_2(\mu)(B) := \mu(X \times B)$ for every measurable $B \in \Sigma_Y$, and $\pi_2(\mu)(\perp) := \mu(\perp)$

Then, $\pi_1(\mu)$ is a measure on $(X + \perp, \Sigma_{X+\perp})$ and $\pi_2(\mu)$ is a measure on $(Y + \perp, \Sigma_{Y+\perp})$.

F.2 Measure Space on Expressions

To make the probabilistic semantics precise, we construct an appropriate measure space structure on the set of expressions by factoring out real numbers using the notion of what we call *expression skeletons*.

Definition F.7 (Expression Skeleton). An expression skeleton s is a syntactic expression obtained from a program by replacing each occurrence of a float constant with a hole of the form $\square$. The set of skeletons is generated by the grammar:

$$\begin{aligned} s ::= & \square \mid x \mid \text{true} \mid \text{false} \mid k_{\#n} \mid \text{diverge} \mid () \\ & \mid \text{let } x = s_1 \text{ in } s_2 \mid \text{if } s_1 \text{ then } s_2 \text{ else } s_3 \\ & \mid s_1 < s_2 \mid s_1 \leq s_2 \mid s_1 <_{\#n} s_2 \mid s_1 \leq_{\#n} s_2 \\ & \mid \text{uniform}(s_1, s_2) \mid \text{gaussian}(s_1, s_2) \mid \text{exponential}(s) \mid \text{beta}(s_1, s_2) \mid \text{discrete}(p_0, \dots, p_{n-1}) \\ & \mid s_1 \ \&\& \ s_2 \mid s_1 \ \parallel \ s_2 \mid \text{not } s \mid \text{observe}(s) \mid s_1; s_2 \\ & \mid (s_1, s_2) \mid \text{fst } s \mid \text{snd } s \\ & \mid \text{fun } x \rightarrow s \mid s_1 \ s_2 \mid \text{fix } f \ x := s \\ & \mid \text{nil} \mid s_1 :: s_2 \\ & \mid (\text{match } s \text{ with } \mid \text{nil} \rightarrow s_{\text{nil}} \mid h :: t \rightarrow s_{\text{cons}} \text{ end}) \end{aligned}$$

We denote by Skel the set of all expression skeletons, and by $\text{holes}(s)$ the number of distinct holes in skeleton s .

Example F.8. The expression $\text{uniform}(0.0, 1.0) < 0.5$ has the skeleton $\text{uniform}(\square, \square) < \square$ with three holes.

Definition F.9 (Filling a skeleton). Let $s \in \text{Skel}$ and let $\vec{r} = (r_1, \dots, r_{\text{holes}(s)}) \in \mathbb{R}^{\text{holes}(s)}$. Define the expression $s[\vec{r}] \in \text{Expr}$ to be the result of replacing each hole $\square_i$ in s with the corresponding real constant r_i , for $1 \leq i \leq \text{holes}(s)$.

To recover a *typed* notion of skeletons, we extend the typing judgment to skeletons by adding a distinguished typing rule for holes.

Definition F.10 (Well-typed expression skeleton). Define a typing judgment for skeletons, written $\Gamma \vdash s : \tau$, with the understanding that all typing rules from expressions are preserved for skeletons, and that holes are typed as $\mathbf{float}[B; V]$ according to the rule below:

$$\frac{}{\square : \mathbf{float}[B; V]}$$

We denote by $Skel_\tau$ the set of all well-typed expression skeletons, where:

$$Skel_\tau := \{s \in Skel \mid s : \tau\}.$$

Remark F.11. Both $Skel_\tau$ and $Expr_\tau$ are to be understood as the set of all well-typed skeletons and expressions, respectively, *equipped with their full subtype information*.

For example,

$$\square : \mathbf{float}[B; \{0.5\}] < \square : \mathbf{float}[B; \{0.5\}] \quad \text{and} \quad \square : \mathbf{float}[B; \{0.2\}] < \square : \mathbf{float}[B; \{0.8\}]$$

are distinct elements of $Skel_{\mathbf{bool}}$, while

$$0.5 : \mathbf{float}[B; \{0.5\}] < 0.5 : \mathbf{float}[B; \{0.5\}] \quad \text{and} \quad 0.2 : \mathbf{float}[B; \{0.2\}] < 0.8 : \mathbf{float}[B; \{0.8\}]$$

are distinct elements of $Expr_{\mathbf{bool}}$.

In particular, both $Skel_\tau$ and $Expr_\tau$ are in general uncountable, since float subtype annotations of the form $\mathbf{float}[B; V]$ may vary over uncountably many real values.

Definition F.12 (Decomposing an expression). The decomposition function

$$\text{dec} : Expr_\tau \longrightarrow \bigcup_{s \in Skel_\tau} (\{s\} \times \mathbb{R}^{\text{holes}(s)})$$

is defined as the inverse of filling a skeleton, i.e., for every expression $e : \tau$,

$$\text{dec}(e) = (s, \vec{r}) \quad \text{such that} \quad e = s[\vec{r}].$$

The explicit recursive definition of dec is shown in Figures 27 and 28.

Definition F.13 (Hole assignments of a skeleton). For a given $s : \tau \in Skel_\tau$, define the set of hole assignments of s as:

$$A_{s;\tau} := \bigcup_{e:\tau \in Expr_\tau} \{\text{dec}(e : \tau)_2 \mid \text{dec}(e : \tau)_1 = s : \tau\} \subseteq \mathbb{R}^{\text{holes}(s)}$$

Intuitively, $A_{s;\tau}$ is the set of all real tuples that can be substituted into the holes of s to yield a well-typed expression of type τ .

Example F.14. Consider the skeleton

$$s : \tau = (\square : \mathbf{float}[\{<3.5, <4.5\}; \top] < \square : \mathbf{float}[\{<3.5, <4.5\}; \{3.5, 4.5\}]) : \mathbf{bool}.$$

This skeleton has two holes:

- The left hole has type $\mathbf{float}[\{<3.5, <4.5\}; \top]$, so it may be instantiated by any real number.
- The right hole has type $\mathbf{float}[\{<3.5, <4.5\}; \{3.5, 4.5\}]$, so it may only be instantiated by one of the values 3.5 or 4.5.

An expression $e : \mathbf{bool}$ has skeleton s iff it is of the form

$$e = c_1 < c_2$$

where $c_1 \in \mathbb{R}$ and $c_2 \in \{3.5, 4.5\}$.

$(x : \tau, ())$	if $e = x : \tau$
$(\text{true} : \mathbf{bool}, ())$	if $e = \text{true} : \mathbf{bool}$
$(\text{false} : \mathbf{bool}, ())$	if $e = \text{false} : \mathbf{bool}$
$(k_{\#n} : \mathbf{fin}(n), ())$	if $e = k_{\#n} : \mathbf{fin}(n)$
$(\mathbf{diverge} : \tau, ())$	if $e = \mathbf{diverge} : \tau$
$(() : \mathbf{unit}, ())$	if $e = () : \mathbf{unit}$
$((\square, \mathbf{float}[B; V]) : \mathbf{float}[B; V], c)$	if $e = c : \mathbf{float}[B; V]$
$(\mathbf{discrete}(p_0, \dots, p_{n-1}) : \mathbf{fin}(n), ())$	if $e = \mathbf{discrete}(p_0, \dots, p_{n-1}) : \mathbf{fin}(n)$
$(\mathbf{uniform}(\text{dec}(e_1)_1, \text{dec}(e_2)_1) : \mathbf{float}[B; V],$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2)$	if $e = \mathbf{uniform}(e_1, e_2) : \mathbf{float}[B; V]$
$(\mathbf{gaussian}(\text{dec}(e_1)_1, \text{dec}(e_2)_1) : \mathbf{float}[B; V],$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2)$	if $e = \mathbf{gaussian}(e_1, e_2) : \mathbf{float}[B; V]$
$(\mathbf{exponential}(\text{dec}(e_1)_1) : \mathbf{float}[B; V],$ $\text{dec}(e_1)_2)$	if $e = \mathbf{exponential}(e_1) : \mathbf{float}[B; V]$
$(\mathbf{beta}(\text{dec}(e_1)_1, \text{dec}(e_2)_1) : \mathbf{float}[B; V],$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2)$	if $e = \mathbf{beta}(e_1, e_2) : \mathbf{float}[B; V]$
$(\text{dec}(e_1)_1 \sim \text{dec}(e_2)_1 : \mathbf{bool},$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2)$	if $e = e_1 \sim e_2 : \mathbf{bool}, \sim, <, \leq, <_{\#n}, \leq_{\#n}$
$(\mathbf{if} \text{dec}(e_1)_1 \text{ then } \text{dec}(e_2)_1 \text{ else } \text{dec}(e_3)_1 : \tau,$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2, \text{dec}(e_3)_2)$	if $e = \mathbf{if} e_1 \text{ then } e_2 \text{ else } e_3 : \tau$
$(\mathbf{let} x = \text{dec}(e_1)_1 \text{ in } \text{dec}(e_2)_1 : \tau,$ $\text{dec}(e_1)_2, \text{dec}(e_2)_2)$	if $e = \mathbf{let} x = e_1 \text{ in } e_2 : \tau$
$(\mathbf{observe}(\text{dec}(e_1)_1 : \mathbf{bool}) : \mathbf{unit}, \text{dec}(e_1)_2)$	if $e = \mathbf{observe}(e_1 : \mathbf{bool}) : \mathbf{unit}$

Fig. 27. Recursive definition of dec , a decomposition function, over typed expressions. $\text{dec}(\cdot)_1$ and $\text{dec}(\cdot)_2$ denote first and second projections of dec , respectively. $\text{dec}(\cdot)_2$ is canonically bijective to a flat vector in $\mathbb{R}^n$ via coordinate concatenation over holes.

Therefore, the admissible parameter tuples are precisely

$$A_{S;\tau} = \{(c_1, c_2) \mid c_1 \in \mathbb{R}, c_2 \in \{3.5, 4.5\}\}.$$

Equivalently,

$$A_{S;\tau} = \mathbb{R} \times \{3.5, 4.5\} \subseteq \mathbb{R}^2.$$

$$\text{dec}(e : \tau) = \left\{ \begin{array}{ll}
(\text{dec}(e_1)_1 \ \&\& \text{dec}(e_2)_1 : \mathbf{bool}, & \text{if } e = e_1 \ \&\& \ e_2 : \mathbf{bool} \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
(\text{dec}(e_1)_1 \ || \ \text{dec}(e_2)_1 : \mathbf{bool}, & \text{if } e = e_1 \ || \ e_2 : \mathbf{bool} \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
(\text{not } \text{dec}(e_1)_1 : \mathbf{bool}, \text{dec}(e_1)_2) & \text{if } e = \text{not } e_1 : \mathbf{bool} \\
\\
(\text{dec}(e_1)_1 ; \text{dec}(e_2)_1 : \tau_2, & \text{if } e = e_1 ; e_2 : \tau_2 \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
((\text{dec}(e_1)_1, \text{dec}(e_2)_1) : \tau_1 * \tau_2, & \text{if } e = (e_1, e_2) : \tau_1 * \tau_2 \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
(\mathbf{fst} \ \text{dec}(e_1)_1 : \tau_1, \text{dec}(e_1)_2) & \text{if } e = \mathbf{fst} \ e_1 : \tau_1 \\
\\
(\mathbf{snd} \ \text{dec}(e_1)_1 : \tau_2, \text{dec}(e_1)_2) & \text{if } e = \mathbf{snd} \ e_1 : \tau_2 \\
\\
(\mathbf{fun} \ x \rightarrow \text{dec}(e_1)_1 : \tau_1 \rightarrow \tau_2, \text{dec}(e_1)_2) & \text{if } e = \mathbf{fun} \ x \rightarrow e_1 : \tau_1 \rightarrow \tau_2 \\
\\
(\mathbf{fix} \ f \ x := \text{dec}(e_1)_1 : \tau_1 \rightarrow \tau_2, \text{dec}(e_1)_2) & \text{if } e = \mathbf{fix} \ f \ x := e_1 : \tau_1 \rightarrow \tau_2 \\
\\
(\text{dec}(e_1)_1 \ \text{dec}(e_2)_1 : \tau_2, & \text{if } e = (e_1 : \tau_1 \rightarrow \tau_2 \ e_2 : \tau_1) : \tau_2 \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
(\mathbf{nil} : \mathbf{list}(\tau_1), ()) & \text{if } e = \mathbf{nil} : \mathbf{list}(\tau_1) \\
\\
(\text{dec}(e_1)_1 :: \text{dec}(e_2)_1 : \mathbf{list}(\tau_1), & \text{if } e = e_1 :: e_2 : \mathbf{list}(\tau_1) \\
\text{dec}(e_1)_2, \text{dec}(e_2)_2) & \\
\\
(\mathbf{match} \ \text{dec}(e_0)_1 \ \mathbf{with} \ | \ \mathbf{nil} \rightarrow \text{dec}(e_{\mathbf{nil}})_1 & \text{if } e = (\mathbf{match} \ e_0 : \mathbf{list}(\tau_1) \ \mathbf{with} \ | \ \mathbf{nil} \rightarrow e_{\mathbf{nil}} : \tau_2 \\
| \ h :: t \rightarrow \text{dec}(e_{\mathbf{cons}})_1 \ \mathbf{end} : \tau_2, & | \ h :: t \rightarrow e_{\mathbf{cons}} : \tau_2 \ \mathbf{end}) : \tau_2 \\
(\text{dec}(e_0)_2, \text{dec}(e_{\mathbf{nil}})_2, \text{dec}(e_{\mathbf{cons}})_2)) &
\end{array} \right.$$

Fig. 28. Recursive definition of dec over typed expressions (continued).

Thus, expressions with this syntactic shape are parameterized by a two-dimensional space whose first coordinate ranges over $\mathbb{R}$ and whose second coordinate ranges over a finite set.

Definition F.15 (σ -algebra construction on $A_{s;\tau}$). Let $s : \tau \in \text{Skel}_\tau$. Endow $A_{s;\tau}$ with the subspace σ -algebra inherited from the standard Borel σ -algebra on $\mathbb{R}^{\text{holes}(s)}$:

$$\mathcal{F}_{s;\tau} := \{A_{s;\tau} \cap M \mid M \in \mathcal{B}(\mathbb{R}^{\text{holes}(s)})\}.$$

LEMMA F.16. $(A_{s;\tau}, \mathcal{F}_{s;\tau})$ is a Borel measurable space.

PROOF. For a fixed $s : \tau \in \text{Skel}_\tau$, $(\mathbb{R}^{\text{holes}(s)}, \mathcal{B}(\mathbb{R}^{\text{holes}(s)}))$ is a Borel measurable space and $A_{s;\tau} \subseteq \mathbb{R}^{\text{holes}(s)}$. Thus, by Definition F.1, $(A_{s;\tau}, \{A_{s;\tau} \cap M \mid M \in \mathcal{B}(\mathbb{R}^{\text{holes}(s)})\}) = (A_{s;\tau}, \mathcal{F}_{s;\tau})$ is a Borel measurable space. $\square$

Now, we map this measurable space to the space of actual expressions that share that skeleton.

Definition F.17 (Expressions of a skeleton). For a given $s : \tau \in \text{Skel}_\tau$, define the set of well-typed expressions having skeleton s as:

$$\text{Expr}_{s;\tau} := \{ e : \tau \in \text{Expr}_\tau \mid \text{dec}(e : \tau)_1 = s \} \subseteq \text{Expr}_\tau.$$

Intuitively, $\text{Expr}_{s;\tau}$ consists exactly of those expressions whose syntactic shape is fixed by s , and whose real constants vary according to the admissible hole assignments in $A_{s;\tau}$.

Example F.18. Consider the skeleton

$$s : \tau = (\square : \mathbf{float}[\{<3.5, <4.5\}; \top] < \square : \mathbf{float}[\{<3.5, <4.5\}; \{3.5, 4.5\}]) : \mathbf{bool}.$$

An expression $e : \mathbf{bool}$ belongs to $\text{Expr}_{s;\tau}$ iff it is of the form

$$e = c_1 < c_2$$

where $c_1 \in \mathbb{R}$ and $c_2 \in \{3.5, 4.5\}$.

Therefore,

$$\text{Expr}_{s;\tau} = \{ c_1 < c_2 \mid c_1 \in \mathbb{R}, c_2 \in \{3.5, 4.5\} \}.$$

Definition F.19 (σ -algebra construction on $\text{Expr}_{s;\tau}$). Let $s : \tau \in \text{Skel}_\tau$. Endow $\text{Expr}_{s;\tau}$ with the following σ -algebra:

$$\mathcal{F}_{\text{Expr}_{s;\tau}} := \{ \{ e : \tau \in \text{Expr}_{s;\tau} \mid \text{dec}(e : \tau)_2 \in B \} \mid B \in \mathcal{F}_{s;\tau} \}.$$

LEMMA F.20. For a fixed $s : \tau \in \text{Skel}_\tau$, the map

$$\text{dec}_{s;\tau} : \text{Expr}_{s;\tau} \rightarrow A_{s;\tau}, \quad e \mapsto \text{dec}(e)_2.$$

is a bijection.

PROOF. Let $s : \tau \in \text{Skel}_\tau$.

(i) *Injectivity:* Suppose $e_1, e_2 \in \text{Expr}_{s;\tau}$ satisfy $\text{dec}_{s;\tau}(e_1) = \text{dec}_{s;\tau}(e_2)$. By definition of $\text{dec}_{s;\tau}$, $\text{dec}_{s;\tau}(e_1) = \text{dec}_{s;\tau}(e_2) = \vec{r}$ for some $\vec{r} \in A_{s;\tau}$. Since both e_1 and e_2 have skeleton s by assumption, we have $e_1 = s[\vec{r}]$ and $e_2 = s[\vec{r}]$, hence $e_1 = e_2$.

(ii) *Surjectivity:* Let $\vec{r} \in A_{s;\tau}$. By definition of $A_{s;\tau}$, there exists some expression $e \in \text{Expr}_\tau$ such that $\text{dec}(e) = (s, \vec{r})$. Then $e \in \text{Expr}_{s;\tau}$ by construction of $\text{Expr}_{s;\tau}$, and $\text{dec}_{s;\tau}(e) = \vec{r}$.

$\text{dec}_{s;\tau}$ is injective and surjective, hence a bijection. $\square$

LEMMA F.21 (ISOMORPHISM PROPERTY). For $s \in \text{Skel}_\tau$,

$$(\text{Expr}_{s;\tau}, \mathcal{F}_{\text{Expr}_{s;\tau}}) \cong (A_{s;\tau}, \mathcal{F}_{s;\tau}).$$

PROOF. We know $(A_{s;\tau}, \mathcal{F}_{s;\tau})$ is a Borel measurable space by Lemma F.16. First, verify that $\mathcal{F}_{\text{Expr}_{s;\tau}}$ coincides with the σ -algebra generated by $\text{dec}_{s;\tau}$, i.e. $\mathcal{F}_{\text{Expr}_{s;\tau}} = \text{dec}_{s;\tau}^{-1}(\mathcal{F}_{s;\tau})$:

$$\begin{aligned} \mathcal{F}_{\text{Expr}_{s;\tau}} &:= \{ \{ e : \tau \in \text{Expr}_{s;\tau} \mid \text{dec}(e : \tau)_2 \in B \} \mid B \in \mathcal{F}_{s;\tau} \} && \text{(Definition F.19)} \\ &:= \{ \text{dec}_{s;\tau}^{-1}(B) \mid B \in \mathcal{F}_{s;\tau} \} && \text{(unfolding the preimage of } \text{dec}_{s;\tau} \text{)} \\ &:= \text{dec}_{s;\tau}^{-1}(\mathcal{F}_{s;\tau}) && \text{(Definition F.2)} \end{aligned}$$

By Lemma F.20, $\text{dec}_{s;\tau} : \text{Expr}_{s;\tau} \rightarrow A_{s;\tau}$ is a bijection. By Lemma F.3, $\text{dec}_{s;\tau}$ induces an isomorphism $(\text{Expr}_{s;\tau}, \mathcal{F}_{\text{Expr}_{s;\tau}}) \cong (A_{s;\tau}, \mathcal{F}_{s;\tau})$. $\square$

LEMMA F.22. $(\text{Expr}_{s;\tau}, \mathcal{F}_{\text{Expr}_{s;\tau}})$ is a Borel measurable space.

PROOF. By Lemma F.21. $\square$

Now we combine the measurable spaces for each distinct skeleton into a single measurable space of all well-typed expressions of type τ . The set of all well-typed expressions of type τ is thus the disjoint union:

$$Expr_\tau = \bigsqcup_{s:\tau \in Skel_\tau} Expr_{s;\tau}.$$

This union is disjoint because every typed expression $e : \tau$ has a unique skeleton, namely $\text{dec}(e)_1$, so e belongs to exactly one set $Expr_{s;\tau}$.

Definition F.23 (σ -algebra construction on $Expr_\tau$). Endow $Expr_\tau$ with the following σ -algebra:

$$\mathcal{F}_{Expr_\tau} := \{A \subseteq Expr_\tau \mid \forall s \in Skel_\tau. A \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}\}$$

LEMMA F.24. $(Expr_\tau, \mathcal{F}_{Expr_\tau})$ is a measurable space.

PROOF. We verify that $\mathcal{F}_{Expr_\tau}$ is indeed a σ -algebra on $Expr_\tau$.

(i) *Contains $Expr_\tau$:* We show $Expr_\tau \in \mathcal{F}_{Expr_\tau}$. By Definition F.23, it suffices to show that $\forall s \in Skel_\tau, Expr_\tau \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$.

Observe that:

$$\begin{aligned} Expr_\tau \cap Expr_{s;\tau} &= Expr_{s;\tau} \quad (\text{since } Expr_{s;\tau} \subseteq Expr_\tau) \end{aligned}$$

Because $\mathcal{F}_{Expr_{s;\tau}}$ is a σ -algebra, it contains the set $Expr_{s;\tau}$. Hence,

$$Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}.$$

Therefore, $Expr_\tau \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$ for all $s \in Skel_\tau$. Therefore, $Expr_\tau \in \mathcal{F}_{Expr_\tau}$.

(ii) *Closure under complements:* Let $A \in \mathcal{F}_{Expr_\tau}$. We show that $Expr_\tau \setminus A \in \mathcal{F}_{Expr_\tau}$. By Definition F.23, it suffices to show that $\forall s \in Skel_\tau, (Expr_\tau \setminus A) \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$.

Fix $s \in Skel_\tau$. Observe that:

$$\begin{aligned} (Expr_\tau \setminus A) \cap Expr_{s;\tau} &= (Expr_\tau \cap Expr_{s;\tau}) \setminus (A \cap Expr_{s;\tau}) \quad (\cap \text{ distributes over } \setminus) \\ &= Expr_{s;\tau} \setminus (A \cap Expr_{s;\tau}). \quad (\text{since } Expr_{s;\tau} \subseteq Expr_\tau, \text{ so } Expr_\tau \cap Expr_{s;\tau} = Expr_{s;\tau}) \end{aligned}$$

Since $A \in \mathcal{F}_{Expr_\tau}$ by assumption, by Definition F.23, we have $A \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$, and because $\mathcal{F}_{Expr_{s;\tau}}$ is a σ -algebra, it is closed under complements relative to $Expr_{s;\tau}$. Hence

$$Expr_{s;\tau} \setminus (A \cap Expr_{s;\tau}) \in \mathcal{F}_{Expr_{s;\tau}}.$$

Therefore $(Expr_\tau \setminus A) \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$ for all $s \in Skel_\tau$. Therefore, $Expr_\tau \setminus A \in \mathcal{F}_{Expr_\tau}$.

(iii) *Closure under countable unions:* Let $(A_n)_{n \in \mathbb{N}}$ be a sequence with $A_n \in \mathcal{F}_{Expr_\tau}$ for all n . We show that $\bigcup_{n \in \mathbb{N}} A_n \in \mathcal{F}_{Expr_\tau}$. By Definition F.23, it suffices to show that $\forall s \in Skel_\tau, (\bigcup_{n \in \mathbb{N}} A_n) \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$.

Fix $s \in Skel_\tau$. Using distributivity of $\cap$ over $\cup$, we have

$$\left(\bigcup_{n \in \mathbb{N}} A_n \right) \cap Expr_{s;\tau} = \bigcup_{n \in \mathbb{N}} (A_n \cap Expr_{s;\tau}).$$

Since each $A_n \in \mathcal{F}_{Expr_\tau}$ by assumption, by Definition F.23, we have $A_n \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$ for all n . Because $\mathcal{F}_{Expr_{s;\tau}}$ is a σ -algebra, it is closed under countable unions, hence

$$\bigcup_{n \in \mathbb{N}} (A_n \cap Expr_{s;\tau}) \in \mathcal{F}_{Expr_{s;\tau}}.$$

Therefore $(\bigcup_{n \in \mathbb{N}} A_n) \cap \text{Expr}_{s;\tau} \in \mathcal{F}_{\text{Expr}_{s;\tau}}$ for all s , and thus $\bigcup_{n \in \mathbb{N}} A_n \in \mathcal{F}_{\text{Expr}_\tau}$.

Since $\mathcal{F}_{\text{Expr}_\tau}$ is a σ -algebra on Expr_τ , $(\text{Expr}_\tau, \mathcal{F}_{\text{Expr}_\tau})$ is a measurable space. $\square$

Definition F.25 (Extension with error element). Let $(\text{Expr}_\tau, \mathcal{F}_{\text{Expr}_\tau})$ be the measurable space of well-typed expressions of type τ . Define

$$\text{Expr}_\tau + \perp := \text{Expr}_\tau \cup \{\perp\},$$

equipped with the σ -algebra

$$\mathcal{F}_{\text{Expr}_\tau + \perp} := \{A, A \cup \{\perp\} \mid A \in \mathcal{F}_{\text{Expr}_\tau}\}.$$

LEMMA F.26. $(\text{Expr}_\tau + \perp, \mathcal{F}_{\text{Expr}_\tau + \perp})$ is a measurable space.

PROOF. We verify that $\mathcal{F}_{\text{Expr}_\tau + \perp}$ is indeed a σ -algebra on $\text{Expr}_\tau + \perp$.

By construction, every element of $\mathcal{F}_{\text{Expr}_\tau + \perp}$ is of the form A or $A \cup \{\perp\}$ for some $A \in \mathcal{F}_{\text{Expr}_\tau}$.

(i) *Contains $\text{Expr}_\tau + \perp$:* Take $A = \text{Expr}_\tau \in \mathcal{F}_{\text{Expr}_\tau}$. Then,

$$A \cup \{\perp\} = \text{Expr}_\tau \cup \{\perp\} = \text{Expr}_\tau + \perp \in \mathcal{F}_{\text{Expr}_\tau + \perp}.$$

(ii) *Closure under complements:* Let $C \in \mathcal{F}_{\text{Expr}_\tau + \perp}$. There are two cases:

- If $C = A$: Then, $(\text{Expr}_\tau + \perp) \setminus A = (\text{Expr}_\tau \setminus A) \cup \{\perp\}$. Since $\mathcal{F}_{\text{Expr}_\tau}$ is a σ -algebra, $\text{Expr}_\tau \setminus A \in \mathcal{F}_{\text{Expr}_\tau}$, so $(\text{Expr}_\tau \setminus A) \cup \{\perp\}$ is of the form $A \cup \{\perp\}$ for some $A \in \mathcal{F}_{\text{Expr}_\tau}$ and hence lies in $\mathcal{F}_{\text{Expr}_\tau + \perp}$.
- If $C = A \cup \{\perp\}$: Then, $(\text{Expr}_\tau + \perp) \setminus (A \cup \{\perp\}) = \text{Expr}_\tau \setminus A$. Since $\mathcal{F}_{\text{Expr}_\tau}$ is a σ -algebra, $\text{Expr}_\tau \setminus A \in \mathcal{F}_{\text{Expr}_\tau}$, so $\text{Expr}_\tau \setminus A$ is of the form A for some $A \in \mathcal{F}_{\text{Expr}_\tau}$ and hence lies in $\mathcal{F}_{\text{Expr}_\tau + \perp}$.

In both cases the complement lies in $\mathcal{F}_{\text{Expr}_\tau + \perp}$.

(iii) *Closure under countable unions:* Let $(C_n)_{n \in \mathbb{N}}$ be a sequence with $C_n \in \mathcal{F}_{\text{Expr}_\tau + \perp}$ for all n . For each n , define $A_n := C_n \cap \text{Expr}_\tau \in \mathcal{F}_{\text{Expr}_\tau}$, where the latter holds since C_n is of the form A or $A \cup \{\perp\}$ for some $A \in \mathcal{F}_{\text{Expr}_\tau}$, and in either case $C_n \cap \text{Expr}_\tau = A$.

Let $A := \bigcup_{n \in \mathbb{N}} A_n$. Since $\mathcal{F}_{\text{Expr}_\tau}$ is a σ -algebra, $A \in \mathcal{F}_{\text{Expr}_\tau}$. Now observe:

$$\bigcup_{n \in \mathbb{N}} C_n = A \cup \left(\bigcup_{n \in \mathbb{N}} (C_n \setminus \text{Expr}_\tau) \right).$$

The second factor is either $\emptyset$ (if every $C_n = A_n$) or $\{\perp\}$ (if some $C_n = A_n \cup \{\perp\}$). In the first sub-case, $\bigcup_n C_n = A$ is of the form A for some $A \in \mathcal{F}_{\text{Expr}_\tau}$; in the second, $\bigcup_n C_n = A \cup \{\perp\}$ is of the form $A \cup \{\perp\}$ for some $A \in \mathcal{F}_{\text{Expr}_\tau}$. In both cases $\bigcup_n C_n \in \mathcal{F}_{\text{Expr}_\tau + \perp}$. $\square$

Finally, we can equip this measurable space $(\text{Expr}_\tau + \perp, \mathcal{F}_{\text{Expr}_\tau + \perp})$ with a subprobability measure such that $\mu(\text{Expr}_\tau + \perp) \leq 1$.

We give some basic lemmas:

LEMMA F.27. Let $(\text{Expr}_\tau + \perp, \mathcal{F}_{\text{Expr}_\tau + \perp})$ be the Borel measurable space of well-typed expressions with error from Definition F.25. Let $\text{Vals}_\tau \subset \text{Expr}_\tau$ be the set of all well-typed value expressions. Let $N_\tau \subset \text{Expr}_\tau$, defined as $N = \text{Expr}_\tau \setminus \text{Vals}_\tau$, be the set of all well-typed non-value expressions. Let $\{\perp\}$ be the singleton set containing the error element $\perp$. The following are true:

- (1) The set $\text{Vals}_\tau \in \mathcal{F}_{\text{Expr}_\tau + \perp}$.
- (2) The set $\{\perp\} \in \mathcal{F}_{\text{Expr}_\tau + \perp}$.
- (3) The set $N_\tau \in \mathcal{F}_{\text{Expr}_\tau + \perp}$.

PROOF. We prove the measurability of $Vals_\tau$, $\{\perp\}$, and N_τ in separate parts. For each, we show that it belongs to the σ -algebra $\mathcal{F}_{Expr_\tau + \perp}$ on the extended expression space $Expr_\tau + \perp$.

- (1) **(Measurability of $Vals_\tau$):** Every expression skeleton $s \in Skel_\tau$ is either a value skeleton or a non-value skeleton, depending on whether the syntactic form of s is a value. An expression $e \in Expr_\tau$ is a value iff its skeleton is a value skeleton.

To show $Vals_\tau \in \mathcal{F}_{Expr_\tau}$, by Definition F.23, it suffices to show that $\forall s \in Skel_\tau, Vals_\tau \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$.

Fix $s \in Skel_\tau$:

- If s is a value skeleton, then every expression in $Expr_{s;\tau}$ is a value, so $Vals_\tau \cap Expr_{s;\tau} = Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$.
- If s is a non-value skeleton, then no expression in $Expr_{s;\tau}$ is a value, so $Vals_\tau \cap Expr_{s;\tau} = \emptyset \in \mathcal{F}_{Expr_{s;\tau}}$.

Thus, $Vals_\tau \cap Expr_{s;\tau} \in \mathcal{F}_{Expr_{s;\tau}}$ for any $s \in Skel_\tau$, so $Vals_\tau \in \mathcal{F}_{Expr_\tau} \subseteq \mathcal{F}_{Expr_\tau + \perp}$.

- (2) **(Measurability of $\{\perp\}$):** Taking $A = \emptyset$ in Definition F.25 gives $\{\perp\} \in \mathcal{F}_{Expr_\tau + \perp}$.
- (3) **(Measurability of N_τ):** $N_\tau = Expr_\tau \setminus Vals_\tau$ is the complement of $Vals_\tau$ in $Expr_\tau$. Since $Vals_\tau \in \mathcal{F}_{Expr_\tau}$ and $\mathcal{F}_{Expr_\tau}$ is a σ -algebra, $N_\tau \in \mathcal{F}_{Expr_\tau} \subseteq \mathcal{F}_{Expr_\tau + \perp}$.

□

Note that in general, given (X, Σ_X) and (Y, Σ_Y) to be measurable spaces, $((X \times Y) + \perp)$ forms a measurable space, since the product $(X \times Y)$ is measurable and for any measurable space (Z, Σ_Z) , the space $(Z + \perp, \Sigma_{Z+\perp})$ with $\Sigma_{Z+\perp}$ generated by $\Sigma_Z \cup \perp$ is measurable.

F.3 Distribution Monad with Error

Let X be a set equipped with a suitable measurable space so that $\mathcal{D}(X + \perp)$ denotes the set of subprobability measures over $X + \perp$. We will make heavy use of the following generic monad construction, for which one easily verifies the monad laws:

Definition F.28 (Distribution Monad with Error). We define

$$\begin{aligned} \text{return} &: X \rightarrow \mathcal{D}(X + \perp), \quad \text{return}(x) = \delta_x \\ \text{bind} &: \mathcal{D}(X + \perp) \rightarrow ((X + \perp) \rightarrow \mathcal{D}(Y + \perp)) \rightarrow \mathcal{D}(Y + \perp), \\ \text{bind}(\mu)(f)(A) &= \int_{X+\perp} f(x)(A) \cdot \mu(dx), \end{aligned}$$

where δ_x denotes the Dirac measure for x , and f is a sub-Markov kernel.

Definition F.29 ($\perp$ -extension). Define the $\perp$ -extension of $g : X \rightarrow \mathcal{D}(Y + \perp)$ to be $g_\perp : (X + \perp) \rightarrow \mathcal{D}(Y + \perp)$, obtained from g by:

$$g_\perp(e) = \begin{cases} \delta_\perp & \text{if } e = \perp \\ g(e) & \text{otherwise} \end{cases}.$$

LEMMA F.30 ($\perp$ -EXTENSION IS A SUB-MARKOV KERNEL). *Given a sub-Markov kernel $f : X \rightarrow \mathcal{D}(Y + \perp)$, its $\perp$ extension $f_\perp : (X + \perp) \rightarrow \mathcal{D}(Y + \perp)$ is also a sub-Markov kernel.*

PROOF. We verify the two conditions for $f_\perp$ to be a sub-Markov kernel.

- (1) *Subprobability measure:* We show for each $e \in X + \perp$, the map

$$A \mapsto f_\perp(e)(A)$$

is a subprobability measure.

Fix $e \in X + \perp$.

- If $e = \perp$, then $f_{\perp}(\perp) = \delta_{\perp}$, the Dirac measure at $\perp$, which is a subprobability measure on $Y + \perp$.
- If $e \in X$, then $f_{\perp}(e) = f(e)$, and since f is a sub-Markov kernel by assumption, $f(e)$ is a subprobability measure on $Y + \perp$.

Thus, for all $e \in X + \perp$, the map $A \mapsto f_{\perp}(e)(A)$ is a subprobability measure.

(2) *Measurable function:* We show for each $A \in \mathcal{F}_{Y+\perp}$, the map $e \mapsto f_{\perp}(e)(A)$ is a measurable function.

Fix $A \in \mathcal{F}_{Y+\perp}$. Define

$$g: (X + \perp) \rightarrow [0, 1], \quad g(e) := f_{\perp}(e)(A).$$

We prove g is measurable by showing $g^{-1}(B) \in \mathcal{F}_{X+\perp}$ for every $B \in \mathcal{B}([0, 1])$. It suffices to prove that for every $a \in \mathbb{R}$,

$$g^{-1}((a, \infty) \cap [0, 1]) \in \mathcal{F}_{X+\perp}$$

because the family of sets $\{(a, \infty) \cap [0, 1] : a \in \mathbb{R}\}$ generates $\mathcal{B}([0, 1])$.

Because $g(e) \in [0, 1]$ for all $e \in X + \perp$ by definition, we have for every $a \in \mathbb{R}$,

$$\begin{aligned} g^{-1}((a, \infty) \cap [0, 1]) &= \{e \in X + \perp \mid g(e) \in (a, \infty) \cap [0, 1]\} \\ &= \{e \in X + \perp \mid (g(e) \in (a, \infty)) \wedge (g(e) \in [0, 1])\} \\ &= \{e \in X + \perp \mid (g(e) > a) \wedge (g(e) \in [0, 1])\} \\ &= \{e \in X + \perp \mid g(e) > a\}, \end{aligned}$$

where the last step uses that $g(e) \in [0, 1]$ always holds. Hence it suffices to show that $\{e \in X + \perp \mid g(e) > a\} \in \mathcal{F}_{X+\perp}$ for every $a \in \mathbb{R}$.

- If $a \geq 1$: $\{e \in X + \perp \mid g(e) > a\} = \emptyset \in \mathcal{F}_{X+\perp}$.
- If $a < 0$: $\{e \in X + \perp \mid g(e) > a\} = X + \perp \in \mathcal{F}_{X+\perp}$.
- If $0 \leq a < 1$: $\{e \in X + \perp \mid g(e) > a\}$.

The only nontrivial case is the last. It splits into two sets:

$$\begin{aligned} \{e \in X + \perp \mid g(e) > a\} &= \{e \in X + \perp \mid f_{\perp}(e)(A) > a\} \\ &= \{e \in X \mid f_{\perp}(e)(A) > a\} \cup \{\perp \mid f_{\perp}(e)(A) > a\} \\ &= \{e \in X \mid f(e)(A) > a\} \cup \{\perp \mid \delta_{\perp}(A) > a\} \end{aligned}$$

The first set is measurable in X (and hence $X + \perp$) since f is a sub-Markov kernel by assumption. The second set is either $\emptyset \in \mathcal{F}_{X+\perp}$ or $\{\perp\} \in \mathcal{F}_{X+\perp}$. Since $\mathcal{F}_{X+\perp}$ is closed under countable unions, then $g^{-1}(B) \in \mathcal{F}_{X+\perp}$ for every $B \in \mathcal{B}([0, 1])$. Thus, for all $A \in \mathcal{F}_{Y+\perp}$, the map $e \mapsto f_{\perp}(e)(A)$ is a measurable function.

Since both conditions hold, $f_{\perp}$ is a sub-Markov kernel. □

LEMMA F.31 (DIRAC RETURN IS A SUB-MARKOV KERNEL). *For any measurable space $(X, \mathcal{F}_X)$, the map*

$$\text{return} : X \rightarrow \mathcal{D}(X + \perp), \quad x \mapsto \delta_x.$$

is a sub-Markov kernel.

PROOF. We verify the two conditions for return to be a sub-Markov kernel.

(1) *Subprobability measure*: We show for each $x \in X$, the map

$$A \mapsto \text{return}(x)(A) = \delta_x(A)$$

is a subprobability measure.

Fix $x \in X$. δ_x is the Dirac measure at x , hence a probability measure and therefore a subprobability measure on $X + \perp$. Thus, for every $x \in X$, the map $A \mapsto \delta_x(A)$ is a subprobability measure.

(2) *Measurable function*: We show that for each $A \in \mathcal{F}_{X+\perp}$, the map

$$x \mapsto \text{return}(x)(A) = \delta_x(A)$$

is a measurable function.

Fix $A \in \mathcal{F}_{X+\perp}$. Define

$$g : X \rightarrow [0, 1], \quad g(x) := \delta_x(A).$$

We prove g is measurable by showing $g^{-1}(B) \in \mathcal{F}_X$ for every $B \in \mathcal{B}([0, 1])$. It suffices to prove that for every $a \in \mathbb{R}$,

$$g^{-1}((a, \infty) \cap [0, 1]) \in \mathcal{F}_X,$$

because the family $\{(a, \infty) \cap [0, 1] : a \in \mathbb{R}\}$ generates $\mathcal{B}([0, 1])$.

Because $g(e) \in [0, 1]$ for all $e \in X + \perp$, we have for every $a \in \mathbb{R}$,

$$\begin{aligned} g^{-1}((a, \infty) \cap [0, 1]) &= \{e \in X \mid g(e) \in (a, \infty) \cap [0, 1]\} \\ &= \{e \in X \mid (g(e) \in (a, \infty)) \wedge (g(e) \in [0, 1])\} \\ &= \{e \in X \mid (g(e) > a) \wedge (g(e) \in [0, 1])\} \\ &= \{e \in X \mid g(e) > a\}, \end{aligned}$$

where the last step uses that $g(e) \in [0, 1]$ always holds. Hence it suffices to show that $\{e \in X \mid g(e) > a\} \in \mathcal{F}_X$ for every $a \in \mathbb{R}$.

- If $a \geq 1$: $\{x \in X \mid g(x) > a\} = \emptyset \in \mathcal{F}_X$.
- If $a < 0$: $\{x \in X \mid g(x) > a\} = X \in \mathcal{F}_X$.
- If $0 \leq a < 1$: $\{x \in X \mid g(x) > a\}$.

The only nontrivial case is the last. Using $g(x) = \delta_x(A)$ and the fact that $\delta_x(A) \in \{0, 1\}$ for all $x \in X$, we have for $0 \leq a < 1$:

$$\begin{aligned} \{x \in X \mid g(x) > a\} &= \{x \in X \mid \delta_x(A) > a\} \\ &= \{x \in X \mid \delta_x(A) = 1\} \\ &= \{x \in X \mid x \in A\} \\ &= A \cap X. \end{aligned}$$

Since $A \in \mathcal{F}_{X+\perp}$ and $\mathcal{F}_X = \{X \cap A \mid A \in \mathcal{F}_{X+\perp}\}$ is the subspace σ -algebra on $X \subseteq X + \perp$, it follows that $A \cap X \in \mathcal{F}_X$.

Therefore $\{x \in X \mid g(x) > a\} \in \mathcal{F}_X$ for all $a \in \mathbb{R}$, so $g^{-1}((a, \infty) \cap [0, 1]) \in \mathcal{F}_X$ for all a . Hence $g^{-1}(B) \in \mathcal{F}_X$ for every $B \in \mathcal{B}([0, 1])$, and g is measurable. $\square$

LEMMA F.32 (ZERO KERNEL IS A SUB-MARKOV KERNEL). *Let $(X, \mathcal{F}_X)$ and $(Y, \mathcal{F}_Y)$ be measurable spaces. The constant-zero map*

$$f : X \rightarrow \mathcal{D}(Y), \quad f(x)(A) := 0$$

is a sub-Markov kernel.

PROOF. We verify the two kernel conditions.

(1) *Subprobability measure.* We show for each $x \in X$, the map

$$A \mapsto f(x)(A)$$

is a subprobability measure.

Fix any $x \in X$. Then $f(x)$ is the zero measure: it satisfies σ -additivity since

$$f(x)\left(\bigcup_i A_i\right) = 0 = \sum_i 0 = \sum_i f(x)(A_i)$$

for every pairwise disjoint family $(A_i)_i$, and total mass $f(x)(Y) = 0 \leq 1$.

(2) *Measurable function.* For any $A \in \mathcal{F}_Y$, the map $x \mapsto f(x)(A) = 0$ is the constant-zero function on X , hence measurable.

Thus both conditions hold, so f is a sub-Markov kernel. $\square$

LEMMA F.33 (COMPOSITION OF SUB-MARKOV KERNELS). *Let (X, Σ_X) , (Y, Σ_Y) , and (Z, Σ_Z) be measurable spaces. If*

$$P : X \rightarrow \mathcal{D}(Y) \quad \text{and} \quad Q : Y \rightarrow \mathcal{D}(Z)$$

are sub-Markov kernels, then their composition

$$(P \gg Q)(x, A) := \int_Y Q(y, A) P(x, dy)$$

defines a sub-Markov kernel

$$P \gg Q : X \rightarrow \mathcal{D}(Z).$$

PROOF. For each $x \in X$, the map $A \mapsto (P \gg Q)(x, A)$ is a measure by the standard integral construction. Its total mass is bounded by

$$(P \gg Q)(x, Z) = \int_Y Q(y, Z) P(x, dy) \leq \int_Y 1 P(x, dy) = P(x, Y) \leq 1.$$

For each $A \in \Sigma_Z$, measurability of $y \mapsto Q(y, A)$ and the standard measurability theorem for integration against a kernel imply that $x \mapsto \int_Y Q(y, A) P(x, dy)$ is measurable; see [30, Lemma 1.38]. Hence the composition is a sub-Markov kernel. $\square$

F.4 Small-Step Semantics

Throughout this section, we denote by $e : \tau$ a well-typed expression of type τ and implicitly assume that all sub-expressions are annotated with types, as well. Moreover, for the typed expression $e : \tau = \mathbf{uniform}(e_1 : \mathbf{float}[B_1; V_1], e_2 : \mathbf{float}[B_2; V_2]) : \mathbf{float}[B, V]$ with $B, B_1, B_2 \neq \top$ and $V_1, V_2 \neq \top$, we introduce an auxiliary statement $\mathbf{DC}_{B, B_1, B_2, V_1, V_2}$ and let

$$\mathcal{D}\llbracket e : \tau \rrbracket = \mathbf{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}\llbracket e_1 : \mathbf{float}[B_1; V_1] \rrbracket, \mathcal{D}\llbracket e_2 : \mathbf{float}[B_2; V_2] \rrbracket).$$

We define the semantics of $\mathbf{DC}_{B, B_1, B_2, V_1, V_2}$ in such a way that it behaves like the discretized program in the last case of Figure 13 but resolves the let-bindings and the conditional choices in a single step, which will simplify our proof. We equip this statement with the following typing rule :

$$\frac{\text{TDISCRETECASE} \quad e_1 : \mathbf{fin}(|B_1| + 1) \quad e_2 : \mathbf{fin}(|B_2| + 1) \quad \text{recovers}(B_1, V_1) \quad \text{recovers}(B_2, V_2)}{\mathbf{DC}_{B, B_1, B_2, V_1, V_2}(e_1, e_2) : \mathbf{fin}(|B| + 1)}$$

Equipped with these notions, we define the small-step semantics of our language:

Definition F.34. The small-step semantics $\llbracket e : \tau \rrbracket \in \mathcal{D}(\text{Expr}_\tau + \perp)$ is defined recursively on the structure of $e : \tau$ by the rules in Table 1 and Table 2.

Table 1. Small-step operational semantics. In the last case, the existence of suitable v_i and w_j is guaranteed since $\text{recovers}(B_i, V_i)$ and $e_i \in V_i$ for $i \in \{1, 2\}$. Here, every function of the form $f(g) = \lambda g. \delta_{\dots}$ is extended to $f_{\perp}$ in the sense of Definition F.29.

$e : \tau$	constraints	$\llbracket e : \tau \rrbracket$
diverge : τ		$\lambda A. 0$
$v : \tau$		$\delta_{v:\tau}$
let $x = e_1 : \tau_1$ in $e_2 : \tau_2 : \tau_2$	$e_1 = v$ e_1 no val.	$\delta_{e_2[v/x]:\tau_2}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{\text{let } x=g \text{ in } e_2:\tau_2:\tau_2}$
if $e_1 : \text{bool}$ then $e_2 : \tau$ else $e_3 : \tau : \tau$	$e_1 = \text{true}$ $e_1 = \text{false}$ o.w.	$\delta_{e_2:\tau}$ $\delta_{e_3:\tau}$ $\llbracket e_1 : \text{bool} \rrbracket \ggg \lambda g. \delta_{\text{if } g \text{ then } e_2:\tau \text{ else } e_3:\tau:\tau}$
observe ($e : \text{bool}$) : unit	$e = \text{true}$ $e = \text{false}$ o.w.	δ_{unit} $\delta_{\perp}$ $\llbracket e : \text{bool} \rrbracket \ggg \lambda g. \delta_{\text{observe } (g:\text{bool}):\text{unit}}$
$e_1 : \tau_1 < e_2 : \tau_2 : \text{bool}$ for $\tau_1 = \text{float}[B; V_1]$, $\tau_2 = \text{float}[B; V_2]$ or $\tau_1 = \tau_2 = \text{fin}(n)$	$e_1 = v_1, e_2 = v_2, v_1 < v_2$ $e_1 = v_1, e_2 = v_2, v_1 \geq v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{\text{true:bool}}$ $\delta_{\text{false:bool}}$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g. \delta_{v_1:\tau_1 < g:\text{bool}}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{g < e_2:\tau_2:\text{bool}}$
uniform ($e_1 : \tau_1, e_2 : \tau_2$) : float [$B; V$]	$e_1 = v_1, e_2 = v_2, v_1 < v_2$ $e_1 = v_1, e_2 = v_2, v_1 \geq v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\text{Uniform}(v_1, v_2) \ggg \lambda v. \delta_{(v:\text{float}[B;V])}$ $\llbracket \text{diverge} : \tau \rrbracket$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g. \delta_{\text{uniform}(v_1:\tau_1,g):\tau}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{\text{uniform}(g,e_2:\tau_2):\tau}$
discrete ($v_0, \dots, v_{n-1}$) : fin (n)		$\text{Discrete}(v_0, \dots, v_{n-1})$
$\text{DC}_{B,B_1,B_2,V_1,V_2}(e_1 : \tau_1, e_2 : \tau_2) : \text{fin}(B + 1)$ $\tau = \text{fin}(B + 1)$, $\tau_1 = \text{fin}(B_1 + 1)$, $\tau_2 = \text{fin}(B_2 + 1)$, $V_1 = \{v_1, \dots, v_n\}$, $V_2 = \{w_1, \dots, w_m\}$	$e_1 = v', e_2 = w'$, $V_1 = \emptyset \vee V_2 = \emptyset$ $e_1 = \mathcal{D}\llbracket v_i : \text{float}[B_1; V_1] \rrbracket$, $e_2 = \mathcal{D}\llbracket w_j : \text{float}[B_2; V_2] \rrbracket$ $e_1 = v', e_2 \neq w'$ o.w.	$\llbracket \text{diverge} : \tau \rrbracket$ $\llbracket \mathcal{D}\llbracket \text{uniform}(v_i, w_j) : \text{float}[B; \top] \rrbracket : \tau \rrbracket$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g. \delta_{\text{DC}_{B,B_1,B_2,V_1,V_2}(v':\tau_1,g)}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{\text{DC}_{B,B_1,B_2,V_1,V_2}(g,e_2:\tau_2)}$

The missing probability mass in $\llbracket e : \tau \rrbracket$ is the probability of encountering an error due to ill-formed parameters fed into distributions (e.g., $\text{uniform}(0.8, 0.1)$). $\llbracket e : \tau \rrbracket(\perp)$ is the probability of encountering an observation failure.

LEMMA F.35 (MEASURABILITY OF CONSTANT EXTENSION MAP). *Let $A \subseteq \mathbb{R}^n$ be equipped with the subspace σ -algebra*

$$\mathcal{F}_A := \{A \cap M \mid M \in \mathcal{B}(\mathbb{R}^n)\},$$

and let $B \subseteq \mathbb{R}^d$ be equipped with the subspace σ -algebra

$$\mathcal{F}_B := \{B \cap N \mid N \in \mathcal{B}(\mathbb{R}^d)\}.$$

Fix constant vectors $c_1 \in \mathbb{R}^{m_1}, \dots, c_k \in \mathbb{R}^{m_k}$. Define

$$g : \mathbb{R}^n \rightarrow \mathbb{R}^d$$

to be any map obtained by concatenating blocks consisting of

- *copies of the input vector $\vec{r} \in \mathbb{R}^n$, and*
- *the constant vectors $c_1, \dots, c_k$,*

Table 2. Small-step operational semantics (continued). Here, every function of the form $f(g) = \lambda g.\delta \dots$ is extended to $f_\perp$ in the sense of Definition F.29.

$e : \tau$	constraints	$\llbracket e : \tau \rrbracket$
$e_1 : \tau_1 \ \&\& \ e_2 : \tau_2 : \mathbf{bool}$ $\tau_1 = \mathbf{bool}, \tau_2 = \mathbf{bool}$	$e_1 = b_1 = \text{true}, e_2 = b_2 = \text{true}$ $e_1 = b_1, e_2 = b_2$ $e_1 = b_1, e_2 \neq b_2$ o.w.	$\delta_{\text{true}:\mathbf{bool}}$ $\delta_{\text{false}:\mathbf{bool}}$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g.\delta_{b_1:\tau_1 \ \&\& g:\mathbf{bool}}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g.\delta_{g \ \&\& e_2:\tau_2:\mathbf{bool}}$
$e_1 : \tau_1 \ \ e_2 : \tau_2 : \mathbf{bool}$ $\tau_1 = \mathbf{bool}, \tau_2 = \mathbf{bool}$	$e_1 = b_1 = \text{false}, e_2 = b_2 = \text{false}$ $e_1 = b_1, e_2 = b_2$ $e_1 = b_1, e_2 \neq b_2$ o.w.	$\delta_{\text{false}:\mathbf{bool}}$ $\delta_{\text{true}:\mathbf{bool}}$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g.\delta_{b_1:\tau_1 \ g:\mathbf{bool}}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g.\delta_{g \ e_2:\tau_2:\mathbf{bool}}$
$\text{not } e : \tau : \mathbf{bool}$ $\tau = \mathbf{bool}$	$e = \text{true}$ $e = \text{false}$ o.w.	$\delta_{\text{false}:\mathbf{bool}}$ $\delta_{\text{true}:\mathbf{bool}}$ $\llbracket e : \tau \rrbracket \ggg \lambda g.\delta_{\text{not } g:\mathbf{bool}}$
$(e_1 : \tau_1, e_2 : \tau_2) : \tau_1 * \tau_2$	$e_1 = v_1, e_2 = v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{(v_1,v_2):\tau_1 * \tau_2}$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g.\delta_{(v_1:\tau_1, f):\tau_1 * \tau_2}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g.\delta_{(f, e_2:\tau_2):\tau_1 * \tau_2}$
$\mathbf{fst} (e : \tau_1 * \tau_2) : \tau_1$	$e = (v_1, v_2)$ o.w.	$\delta_{v_1:\tau_1}$ $\llbracket e : \tau_1 * \tau_2 \rrbracket \ggg \lambda g.\delta_{\mathbf{fst } e:\tau_1}$
$\mathbf{snd} (e : \tau_1 * \tau_2) : \tau_2$	$e = (v_1, v_2)$ o.w.	$\delta_{v_2:\tau_2}$ $\llbracket e : \tau_1 * \tau_2 \rrbracket \ggg \lambda g.\delta_{\mathbf{snd } e:\tau_2}$
$(e_1 : \tau_1 \rightarrow \tau_2 \ e_2 : \tau_1) : \tau_2$	$e_1 = \mathbf{fun } x \rightarrow e \text{ or } \mathbf{fix } g \ x := e, e_2 = v_2$ $e_1 = \mathbf{fun } x \rightarrow e \text{ or } \mathbf{fix } g \ x := e, e_2 \neq v_2$ o.w.	$\delta_{e[v_2/x]:\tau_2}$ $\llbracket e_2 : \tau_2 \rrbracket \ggg \lambda g.\delta_{e_1:\tau_1 \rightarrow \tau_2 \ g:\tau_2}$ $\llbracket e_1 : \tau_1 \rightarrow \tau_2 \rrbracket \ggg \lambda g.\delta_{f \ e_2:\tau_1:\tau_2}$
$\mathbf{nil} : \mathbf{list}(\tau)$		$\delta_{\mathbf{nil}:\mathbf{list}(\tau)}$
$(e_1 : \tau_1 :: e_2 : \mathbf{list}(\tau)) : \mathbf{list}(\tau)$	$e_1 = v_1, e_2 = v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{v_1::v_2:\mathbf{list}(\tau)}$ $\llbracket e_2 : \mathbf{list}(\tau) \rrbracket \ggg \lambda g.\delta_{v_1:\tau_1::g:\mathbf{list}(\tau)}$ $\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g.\delta_{f::e_2:\mathbf{list}(\tau):\mathbf{list}(\tau)}$
$(\mathbf{match } e : \mathbf{list}(\tau_1) \text{ with}$ $ \ \mathbf{nil} \rightarrow e_1 : \tau_2$ $ \ h :: t \rightarrow e_2 : \tau_2$ $\mathbf{end}) : \tau_2$	$e = \mathbf{nil}$ $e = v_h :: v_t$ o.w.	$\delta_{e_1:\tau_2}$ $\delta_{e_2[v_h/h, v_t/t]:\tau_2}$ $\llbracket e : \mathbf{list}(\tau_1) \rrbracket \ggg \lambda g.\delta_{\mathbf{match } g \text{ with } \dots:\tau_2}$

in some fixed order. That is, $g(\vec{r})$ is formed by inserting the vectors $\vec{r}$ and c_j into a fixed sequence of blocks.

Assume $g(A) \subseteq B$ and define $\phi := g|_A : A \rightarrow B$. Then ϕ is measurable.

Example F.36 (Constant extension maps). The constant extension map allows constants to be inserted and the input vector $\vec{r} \in \mathbb{R}^n$ to appear multiple times. The following are examples of such maps:

- $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $g(\vec{r}) = \vec{r}$
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^{n+m_1}$, $g(\vec{r}) = (\vec{r}, c_1)$, $c_1 \in \mathbb{R}^{m_1}$
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^{m_1+n+m_2}$, $g(\vec{r}) = (c_1, \vec{r}, c_2)$, $c_1 \in \mathbb{R}^{m_1}$, $c_2 \in \mathbb{R}^{m_2}$
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^{n+m_2}$, $g(\vec{r}) = (\vec{r}, c_2)$, $c_2 \in \mathbb{R}^{m_2}$
- $g : \mathbb{R}^n \rightarrow \mathbb{R}^{m_1+n+m_2+n+m_3+n}$, $g(\vec{r}) = (c_1, \vec{r}, c_2, \vec{r}, c_3, \vec{r})$, $c_i \in \mathbb{R}^{m_i}$

PROOF. We show that $\phi^{-1}(S) \in \mathcal{F}_A$ for each $S \in \mathcal{F}_B$.

Fix $S \in \mathcal{F}_B$. By definition of $\mathcal{F}_B$, there exists $N \in \mathcal{B}(\mathbb{R}^d)$ such that $S = B \cap N$. We compute:

$$\begin{aligned}
 \phi^{-1}(S) &= \{\vec{r} \in A \mid \phi(\vec{r}) \in B \cap N\} \\
 &= \{\vec{r} \in A \mid \phi(\vec{r}) \in B \cap \phi(\vec{r}) \in N\} && \text{(definition of } \cap \text{)} \\
 &= \{\vec{r} \in A \mid \phi(\vec{r}) \in N\} && (\phi(\vec{r}) \in B \text{ for all } \vec{r} \in A) \\
 &= \{\vec{r} \in A \mid g(\vec{r}) \in N\} && \text{(definition of } \phi \text{)} \\
 &= \{\vec{r} \in A \mid g(\vec{r}) \in N\} \cap \mathbb{R}^n && (\vec{r} \text{ ranges over } A \subseteq \mathbb{R}^n) \\
 &= A \cap \{\vec{r} \in \mathbb{R}^n \mid g(\vec{r}) \in N\}. && \text{(rewrite as } \cap \text{ with } A \text{)}
 \end{aligned}$$

It remains to show that $\{\vec{r} \in \mathbb{R}^n \mid (\vec{r}, \vec{c}) \in N\} \in \mathcal{B}(\mathbb{R}^n)$. Define the total constant-extension map

$$g: \mathbb{R}^n \rightarrow \mathbb{R}^d.$$

We claim g is continuous. A map into a product space is continuous if and only if each coordinate function is continuous [36, Theorem 19.6]. The coordinate functions of g are:

- the identity $\text{id}: \mathbb{R}^n \rightarrow \mathbb{R}^n, \vec{r} \mapsto \vec{r}$, and
- the constant map $\kappa_{\vec{c}_i}: \mathbb{R}^n \rightarrow \mathbb{R}^{m_i}, \vec{r} \mapsto \vec{c}_i$.

Both are continuous: for any open $U \subseteq \mathbb{R}^n$, $\text{id}^{-1}(U) = U$ is open; and for any open $V \subseteq \mathbb{R}^{m_i}$,

$$\kappa_{\vec{c}_i}^{-1}(V) = \begin{cases} \mathbb{R}^n & \text{if } \vec{c}_i \in V, \\ \emptyset & \text{if } \vec{c}_i \notin V, \end{cases}$$

both of which are open. Hence g is continuous, and therefore Borel measurable. Applying this to $N \in \mathcal{B}(\mathbb{R}^d)$:

$$\{\vec{r} \in \mathbb{R}^n \mid g(\vec{r}) \in N\} = g^{-1}(N) \in \mathcal{B}(\mathbb{R}^n).$$

Hence $\phi_{\vec{c}}^{-1}(S) = A \cap g^{-1}(N) \in \mathcal{F}_A$ by definition, so $\phi_{\vec{c}}$ is measurable. $\square$

LEMMA F.37 (SUBSTITUTION IS MEASURABLE). *Fix a typed expression $e : \tau_2$ and a variable $x : \tau_1$. The substitution map*

$$\text{Sub}_{e,x}: \text{Val}_{\tau_1} \rightarrow \text{Expr}_{\tau_2}, \quad \text{Sub}_{e,x}(v) := e[v/x],$$

is measurable, where $\text{Val}_{\tau_1} \subseteq \text{Expr}_{\tau_1}$ carries the subspace σ -algebra inherited from $(\text{Expr}_{\tau_1}, \mathcal{F}_{\text{Expr}_{\tau_1}})$.

PROOF. We show that $\text{Sub}_{e,x}^{-1}(A) \in \mathcal{F}_{\text{Val}_{\tau_1}}$ for all $A \in \mathcal{F}_{\text{Expr}_{\tau_2}}$.

Fix $A \in \mathcal{F}_{\text{Expr}_{\tau_2}}$. We know:

$$\begin{aligned}
 \text{Sub}_{e,x}^{-1}(A) &= \{v \in \text{Val}_{\tau_1} \mid e[v/x] \in A\} \\
 &= \{v \in \text{Val}_{\tau_1} \mid e[v/x] \in A\} \cap \text{Expr}_{\tau_1} && (v \text{ ranges over } \text{Val}_{\tau_1} \subseteq \text{Expr}_{\tau_1}) \\
 &= \text{Val}_{\tau_1} \cap \{u \in \text{Expr}_{\tau_1} \mid e[u/x] \in A\} && \text{(rewrite as } \cap \text{ with } \text{Val}_{\tau_1} \text{)}
 \end{aligned}$$

Now, define:

$$C := \{u \in \text{Expr}_{\tau_1} \mid e[u/x] \in A\}.$$

We claim proving $C \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ is sufficient because by Definition F.1, this gives $\text{Sub}_{e,x}^{-1}(A) = \text{Val}_{\tau_1} \cap C \in \mathcal{F}_{\text{Val}_{\tau_1}}$. To show $C \in \mathcal{F}_{\text{Expr}_{\tau_1}}$, by Definition F.23, it suffices to show that $\forall s \in \text{Skel}_{\tau_1}, C \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}}$.

Fix $s \in \text{Skel}_{\tau_1}$. We must prove:

$$\begin{aligned}
 & C \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} \\
 \iff & \{u \in \text{Expr}_{\tau_1} \mid e[u/x] \in A\} \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && \text{(definition of } C) \\
 \iff & \{u \in \text{Expr}_{s;\tau_1} \mid e[u/x] \in A\} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && (\text{Expr}_{s;\tau_1} \subseteq \text{Expr}_{\tau_1}) \\
 \iff & \{s[\vec{r}] \in \text{Expr}_{s;\tau_1} \mid e[s[\vec{r}]/x] \in A\} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && \text{(rewrite } u) \\
 \iff & \{\vec{r} \in A_{s;\tau_1} \mid e[s[\vec{r}]/x] \in A\} \in \mathcal{F}_{s;\tau_1} && \text{(Lemma F.21)}
 \end{aligned}$$

Define

$$R_s := \{\vec{r} \in A_{s;\tau_1} \mid e[s[\vec{r}]/x] \in A\}.$$

Let $t \in \text{Skel}_{\tau_2}$ be the skeleton obtained from e by replacing each occurrence of x with s . Define a constant extension map:

$$\psi_s : A_{s;\tau_1} \rightarrow A_{t;\tau_2}, \quad \psi_s(\vec{r}) = \phi(\vec{r}, \vec{c}_1, \dots, \vec{c}_k)$$

where $\vec{c}_1, \dots, \vec{c}_k \in \mathbb{R}$ are the float constants appearing in e (other than at occurrences of x), and ϕ permutes/interleaves the blocks into the order dictated by a left-to-right traversal of the holes of t .

More precisely, since t is obtained from e by replacing each free occurrence of x with s , the holes of t consist exactly of:

- the holes of s , repeated once per occurrence of x in e , and
- the holes coming from the float constants of e itself.

By Lemma F.35, ψ_s is measurable.

By construction, for all $\vec{r} \in A_{s;\tau_1}$:

$$e[s[\vec{r}]/x] = t[\psi_s(\vec{r})].$$

Define the measurable set of hole-assignments corresponding to A at skeleton t :

$$\begin{aligned}
 A_t &:= \{\vec{u} \in A_{t;\tau_2} \mid t[\vec{u}] \in A\} \\
 &:= \{\vec{u} \in A_{t;\tau_2} \mid t[\vec{u}] \in A \cap \text{Expr}_{t;\tau_2}\} && (t[\vec{u}] \text{ ranges over } \text{Expr}_{t;\tau_2})
 \end{aligned}$$

By Definition F.23, $A \cap \text{Expr}_{t;\tau_2} \in \mathcal{F}_{\text{Expr}_{t;\tau_2}}$, and by Lemma F.3, $A_t \in \mathcal{F}_{t;\tau_2}$. We compute:

$$\begin{aligned}
 R_s &= \{\vec{r} \in A_{s;\tau_1} \mid e[s[\vec{r}]/x] \in A\} \\
 &= \{\vec{r} \in A_{s;\tau_1} \mid t[\psi_s(\vec{r})] \in A\} \\
 &= \{\vec{r} \in A_{s;\tau_1} \mid \psi_s(\vec{r}) \in A_t\} \\
 &= \psi_s^{-1}(A_t).
 \end{aligned}$$

Since ψ_s is measurable and $A_t \in \mathcal{F}_{t;\tau_2}$, we have $R_s = \psi_s^{-1}(A_t) \in \mathcal{F}_{s;\tau_1}$.

Since $R_s \in \mathcal{F}_{s;\tau_1}$ for every $s \in \text{Skel}_{\tau_1}$, we get $C \in \mathcal{F}_{\text{Expr}_{\tau_1}}$, and therefore

$$\text{Sub}_{e,x}^{-1}(A) = C \cap \text{Val}_{\tau_1} \in \mathcal{F}_{\text{Val}_{\tau_1}}.$$

Hence $\text{Sub}_{e,x}$ is measurable. □

LEMMA F.38 (MONADIC BIND IS A SUB-MARKOV KERNEL). *For every occurrence of a monadic bind $\mu \gg f$ given in the small-step operational semantics in Tables 1 and 2, $f : (\text{Expr}_{\tau_1} + \perp) \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is a sub-Markov kernel.*

PROOF. We first prove that $f : \text{Expr}_{\tau_1} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is a sub-Markov kernel. Let e be an expression whose evaluation rule contains a monadic bind $\mu \gg f$. We proceed by case analysis on e :

- Consider $e = (\text{let } y = x : \tau_1 \text{ in } e_2 : \tau_2) : \tau_2$.

This means proving that $\lambda x. \delta_{\text{let } y=x \text{ in } e_2}$ is a sub-Markov kernel. Formally, define

$$f: \text{Expr}_{\tau_1} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp), \quad f(x) = \delta_{\text{let } y=x \text{ in } e_2}.$$

We verify the two conditions for f to be a sub-Markov kernel.

(i) *Subprobability measure*: We show for each $x \in \text{Expr}_{\tau_1}$, the map $A \mapsto f(x)(A)$ is a subprobability measure.

Fix $x \in \text{Expr}_{\tau_1}$. The map $A \mapsto f(x)(A)$ equals $A \mapsto \delta_{\text{let } y=x \text{ in } e_2}(A)$, which satisfies:

$$f(x)(A) = \begin{cases} 1 & \text{if } \text{let } y = x \text{ in } e_2 \in A, \\ 0 & \text{otherwise.} \end{cases}$$

A Dirac measure is a subprobability measure, so this condition holds.

(ii) *Measurable function*: We show for each $A \in \mathcal{F}_{\text{Expr}_{\tau_2} + \perp}$, the map $x \mapsto f(x)(A)$ is a measurable function.

Fix $A \in \mathcal{F}_{\text{Expr}_{\tau_2} + \perp}$. Define

$$g: \text{Expr}_{\tau_1} \rightarrow [0, 1], \quad g(x) := f(x)(A) = \delta_{\text{let } y=x \text{ in } e_2}(A).$$

We prove g is measurable by showing $g^{-1}(B) \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ for every $B \in \mathcal{B}([0, 1])$. It suffices to prove that for every $a \in \mathbb{R}$,

$$g^{-1}((a, \infty) \cap [0, 1]) \in \mathcal{F}_{\text{Expr}_{\tau_1}},$$

because the family $\{(a, \infty) \cap [0, 1] : a \in \mathbb{R}\}$ generates $\mathcal{B}([0, 1])$.

Because $g(x) \in [0, 1]$ for all $x \in \text{Expr}_{\tau_1}$, we have for every $a \in \mathbb{R}$,

$$\begin{aligned} g^{-1}((a, \infty) \cap [0, 1]) &= \{x \in \text{Expr}_{\tau_1} \mid g(x) \in (a, \infty) \cap [0, 1]\} \\ &= \{x \in \text{Expr}_{\tau_1} \mid (g(x) \in (a, \infty)) \wedge (g(x) \in [0, 1])\} \\ &= \{x \in \text{Expr}_{\tau_1} \mid (g(x) > a) \wedge (g(x) \in [0, 1])\} \\ &= \{x \in \text{Expr}_{\tau_1} \mid g(x) > a\}, \end{aligned}$$

where the last step uses that $g(x) \in [0, 1]$ always holds. Hence it suffices to show that $\{x \in X \mid g(x) > a\} \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ for every $a \in \mathbb{R}$.

- If $a \geq 1$: $\{x \in \text{Expr}_{\tau_1} \mid g(x) > a\} = \emptyset \in \mathcal{F}_{\text{Expr}_{\tau_1}}$.
- If $a < 0$: $\{x \in \text{Expr}_{\tau_1} \mid g(x) > a\} = \text{Expr}_{\tau_1} \in \mathcal{F}_{\text{Expr}_{\tau_1}}$.
- If $0 \leq a < 1$: $\{x \in \text{Expr}_{\tau_1} \mid g(x) > a\}$.

The only nontrivial case is the last. Using $g(x) = \delta_{\text{let } y=x \text{ in } e_2}$ and the fact that $g(x) \in \{0, 1\}$ for all $x \in \text{Expr}_{\tau_1}$, we have for $0 \leq a < 1$:

$$\{x \in \text{Expr}_{\tau_1} \mid g(x) > a\} = \{x \in \text{Expr}_{\tau_1} \mid g(x) = 1\}$$

Define

$$B := \{x \in \text{Expr}_{\tau_1} \mid \text{let } y = x \text{ in } e_2 \in A\}.$$

We must show $B \in \mathcal{F}_{\text{Expr}_{\tau_1}}$. By Definition F.23, it suffices to show that $\forall s \in \text{Skel}_{\tau_1}, B \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}}$.

Fix $s \in \text{Skel}_{\tau_1}$. We must prove:

$$\begin{aligned}
 & B \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} \\
 \iff & \{x \in \text{Expr}_{\tau_1} \mid \text{let } y = x \text{ in } e_2 \in A\} \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && (\text{Definition of } B) \\
 \iff & \{x \in \text{Expr}_{s;\tau_1} \mid \text{let } y = x \text{ in } e_2 \in A\} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && (\text{Expr}_{s;\tau_1} \subseteq \text{Expr}_{\tau_1}) \\
 \iff & \{s[\vec{r}] \in \text{Expr}_{s;\tau_1} \mid \text{let } y = s[\vec{r}] \text{ in } e_2 \in A\} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}} && (\text{rewrite } x) \\
 \iff & \{\vec{r} \in A_{s;\tau_1} \mid \text{let } y = s[\vec{r}] \text{ in } e_2 \in A\} \in \mathcal{F}_{s;\tau_1} && (\text{Lemma F.21})
 \end{aligned}$$

Define

$$R_s := \{\vec{r} \in A_{s;\tau_1} \mid \text{let } y = s[\vec{r}] \text{ in } e_2 \in A\}.$$

Let s_2 be the skeleton of e_2 , with $m := \text{holes}(s_2)$, and let $\vec{c} \in A_{s_2;\tau_2} \subseteq \mathbb{R}^m$ be the unique tuple such that $e_2 = s_2[\vec{c}]$. Define the combined skeleton

$$s' := \text{let } y = s \text{ in } s_2 \in \text{Skel}_{\tau_2},$$

whose holes consist of those from s (filled by $\vec{r} \in \mathbb{R}^n$) followed by those from s_2 (filled by $\vec{c} \in \mathbb{R}^m$), so that $A_{s';\tau_2} \subseteq \mathbb{R}^{n+m}$. Thus, for every $\vec{r} \in A_{s;\tau_1}$:

$$\text{let } y = s[\vec{r}] \text{ in } e_2 = s'[(\vec{r}, \vec{c})].$$

Define a constant-extension map

$$\phi_{\vec{c}}: A_{s;\tau_1} \rightarrow A_{s';\tau_2}, \quad \phi_{\vec{c}}(\vec{r}) := (\vec{r}, \vec{c}).$$

By Lemma F.35, $\phi_{\vec{c}}$ is measurable.

Define the measurable set of hole-assignments corresponding to A at skeleton s' :

$$\begin{aligned}
 A_{s'} &:= \{(\vec{r}, \vec{r}') \in A_{s';\tau_2} \mid s'[\vec{r}, \vec{r}'] \in A\} \\
 &:= \{(\vec{r}, \vec{r}') \in A_{s';\tau_2} \mid s'[\vec{r}, \vec{r}'] \in A \cap \text{Expr}_{\tau_2}\} && (\text{Definition F.1}) \\
 &:= \{(\vec{r}, \vec{r}') \in A_{s';\tau_2} \mid s'[\vec{r}, \vec{r}'] \in A \cap \text{Expr}_{s';\tau_2}\} && (s'[\vec{r}, \vec{r}'] \text{ ranges over } \text{Expr}_{s';\tau_2})
 \end{aligned}$$

Since $A \cap \text{Expr}_{s';\tau_2} \in \mathcal{F}_{\text{Expr}_{s';\tau_2}}$, by Lemma F.21, $A_{s'} \in \mathcal{F}_{s';\tau_2}$.

Then:

$$\begin{aligned}
 R_s &= \{\vec{r} \in A_{s;\tau_1} \mid \text{let } y = s[\vec{r}] \text{ in } e_2 \in A\} \\
 &= \{\vec{r} \in A_{s;\tau_1} \mid s'[(\vec{r}, \vec{c})] \in A\} \\
 &= \{\vec{r} \in A_{s;\tau_1} \mid s'[\phi_{\vec{c}}(\vec{r})] \in A\} \\
 &= \{\vec{r} \in A_{s;\tau_1} \mid \phi_{\vec{c}}(\vec{r}) \in A_{s'}\} \\
 &= \phi_{\vec{c}}^{-1}(A_{s'}).
 \end{aligned}$$

Since $A_{s'} \in \mathcal{F}_{s';\tau_2}$ and $\phi_{\vec{c}}$ is measurable, we conclude

$$\phi_{\vec{c}}^{-1}(A_{s'}) \in \mathcal{F}_{s;\tau_1},$$

and therefore $B \cap \text{Expr}_{s;\tau_1} \in \mathcal{F}_{\text{Expr}_{s;\tau_1}}$ for all $s \in \text{Skel}_{\tau_1}$, giving $B \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ as required.

Hence, $\{x \in \text{Expr}_{\tau_1} \mid g(x) > a\} \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ for all $a \in \mathbb{R}$, so $g^{-1}((a, \infty) \cap [0, 1]) \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ for all $a \in \mathbb{R}$. Hence $g^{-1}(B) \in \mathcal{F}_{\text{Expr}_{\tau_1}}$ for every $B \in \mathcal{B}([0, 1])$, and g is measurable.

Thus, $f: \text{Expr}_{\tau_1} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is a sub-Markov kernel.

- All other cases analogous.

By Lemma F.30, the $\perp$ -extension $f_{\perp}: (\text{Expr}_{\tau_1} + \perp) \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is also a sub-Markov kernel, as required. $\square$

LEMMA F.39 (SMALL-STEP SEMANTICS IS A SUB-MARKOV KERNEL). *For each type τ , the small-step operational semantics $\llbracket \cdot \rrbracket : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$ defines a sub-Markov kernel.*

PROOF. By induction on the typing derivation of $e : \tau$. We show in each case that the map $e \mapsto \llbracket e : \tau \rrbracket$ is a sub-Markov kernel.

- Consider $e = (\text{let } y = x : \tau_1 \text{ in } e_2 : \tau_2) : \tau_2$. The small-step semantics is:

$$\llbracket \text{let } x = e_1 \text{ in } e_2 : \tau_2 \rrbracket = \begin{cases} \delta_{e_2[v/x]} & \text{if } e_1 = v \\ \llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{\text{let } x=g \text{ in } e_2} & \text{if } e_1 \neq v \end{cases}$$

- If $e_1 = v$, By Lemma F.31 and Lemma F.37, $e \mapsto \delta_{e_2[v/y]:\tau_2}$ is a sub-Markov kernel. Hence, $\llbracket \text{let } x = e_1 \text{ in } e_2 : \tau_2 \rrbracket : \text{Expr}_{\tau_2} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is a sub-Markov kernel.
- By the induction hypothesis,

$$\llbracket e_1 : \tau_1 \rrbracket : \text{Expr}_{\tau_1} \rightarrow \mathcal{D}(\text{Expr}_{\tau_1} + \perp)$$

is a sub-Markov kernel. By Lemma F.38,

$$\lambda g. \delta_{\text{let } x=g \text{ in } e_2} : (\text{Expr}_{\tau_1} + \perp) \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$$

is a sub-Markov kernel. Hence by Lemma F.33, their composition

$$\llbracket e_1 : \tau_1 \rrbracket \ggg \lambda g. \delta_{\text{let } x=g \text{ in } e_2} : \text{Expr}_{\tau_1} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$$

is a sub-Markov kernel.

Thus, $\llbracket \text{let } x = e_1 \text{ in } e_2 : \tau_2 \rrbracket : \text{Expr}_{\tau_2} \rightarrow \mathcal{D}(\text{Expr}_{\tau_2} + \perp)$ is a sub-Markov kernel.

- Consider $e = \text{diverge} : \tau$.

The small-step semantics is:

$$\llbracket \text{diverge} : \tau \rrbracket = \lambda A. 0.$$

By Lemma F.32,

$$\lambda A. 0$$

is a sub-Markov kernel.

Thus, $\llbracket \text{diverge} : \tau \rrbracket : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$ is a sub-Markov kernel.

- All other cases are analogous.

□

F.5 Small-Step Coupling Semantics

Define $P_\tau = \{(e : \tau, \mathcal{D}[\llbracket e : \tau \rrbracket] : \mathcal{D}[\llbracket \tau \rrbracket]) \mid e : \tau\}$. To define subprobability measures and kernels on the coupling space, we must first equip it with a σ -algebra. Since

$$P_\tau \subseteq \text{Expr}_\tau \times \text{Expr}_{\mathcal{D}[\llbracket \tau \rrbracket]},$$

we use the subspace σ -algebra from Definition F.1, inherited from the product measurable space. We then extend this space with the isolated error element $\perp$, as for $\text{Expr}_\tau + \perp$.

Definition F.40 (σ -algebra construction on P_τ). Let

$$P_\tau = \{(e : \tau, \mathcal{D}[\llbracket e \rrbracket] : \mathcal{D}[\llbracket \tau \rrbracket]) \mid e : \tau\} \subseteq \text{Expr}_\tau \times \text{Expr}_{\mathcal{D}[\llbracket \tau \rrbracket]}.$$

We equip P_τ with the subspace σ -algebra

$$\mathcal{F}_{P_\tau} := \left\{ P_\tau \cap A \mid A \in \mathcal{F}_{\text{Expr}_\tau} \otimes \mathcal{F}_{\text{Expr}_{\mathcal{D}[\llbracket \tau \rrbracket]}} \right\}.$$

We equip $P_\tau + \perp$ with

$$\mathcal{F}_{P_\tau + \perp} := \{A, A \cup \{\perp\} \mid A \in \mathcal{F}_{P_\tau}\}.$$

Table 3. Small-step coupling semantics. Notice that $\mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!]$ is uniquely determined by $e : \tau$ so that it suffices to indicate the latter in the left-hand column. Here, every function of the form $f(g, g') = \lambda(g, g').\delta_{\dots}$ is extended to $f_{\perp}$ in the sense of Definition F.29.

$e : \tau$	constraints	$\llbracket e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rrbracket$
diverge : τ		$\lambda A.0$
$v : \tau$		$\delta_{(v:\tau, \mathcal{D}[\![v:\tau]\!]:\mathcal{D}[\![\tau]\!])}$
let $x = e_1 : \tau_1$ in $e_2 : \tau_2 : \tau_2$	$e_1 = v$ e_1 no val.	$\delta_{(e_2[v/x]:\tau_2, \mathcal{D}[\![e_2:v]\!]:\mathcal{D}[\![v:\tau_1]\!]/x):\mathcal{D}[\![\tau_2]\!]}$ $\llbracket e_1 : \tau_1, \mathcal{D}[\![e_1 : \tau_1]\!] : \mathcal{D}[\![\tau_1]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{let } x=g \text{ in } e_2:\tau_2:\tau_2, \text{let } x=g' \text{ in } \mathcal{D}[\![e_2:\tau_2]\!]:\mathcal{D}[\![\tau_2]\!]:\mathcal{D}[\![\tau]\!])}$
if $e_1 : \text{bool}$ then $e_2 : \tau$ else $e_3 : \tau : \tau$	$e_1 = \text{true}$ $e_1 = \text{false}$ o.w.	$\delta_{(e_2:\tau, \mathcal{D}[\![e_2:\tau]\!]:\mathcal{D}[\![\tau]\!])}$ $\delta_{(e_3:\tau, \mathcal{D}[\![e_3:\tau]\!]:\mathcal{D}[\![\tau]\!])}$ $\llbracket e_1 : \text{bool}, \mathcal{D}[\![e_1 : \text{bool}]\!] : \mathcal{D}[\![\text{bool}]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{if } g \text{ then } e_2:\tau \text{ else } e_3:\tau:\tau, \text{if } g' \text{ then } \mathcal{D}[\![e_2:\tau]\!]:\mathcal{D}[\![\tau]\!] \text{ else } \mathcal{D}[\![e_3:\tau]\!]:\mathcal{D}[\![\tau]\!]:\mathcal{D}[\![\tau]\!]})}$
observe ($e : \text{bool}$) : unit	$e = \text{true}$ $e = \text{false}$ o.w.	$\delta_{(\text{unit}, \text{unit})}$ $\delta_{\perp}$ $\llbracket e : \text{bool}, \mathcal{D}[\![e]\!] : \text{bool} \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{observe } (g:\text{bool}):\text{unit}, \text{observe } (g':\text{bool}):\text{unit})}$
$e_1 : \tau_1 < e_2 : \tau_2 : \text{bool}$ for $\tau_1 = \text{float}[B; V_1], \tau_2 = \text{float}[B; V_2]$ $B = \top$	$e_1 = v_1, e_2 = v_2, v_1 < v_2$ $e_1 = v_1, e_2 = v_2, v_1 \geq v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{(\text{true}:\text{bool}, \text{true}:\text{bool})}$ $\delta_{(\text{false}:\text{bool}, \text{false}:\text{bool})}$ $\llbracket e_2 : \tau_2, \mathcal{D}[\![e_2 : \tau_2]\!] : \mathcal{D}[\![\tau_2]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(v_1:\tau_1 < g:\text{bool}, \mathcal{D}[\![v_1:\tau_1]\!]:\mathcal{D}[\![\tau_1]\!] < g':\mathcal{D}[\![\text{bool}]\!]}$ $\llbracket e_1 : \tau_1, \mathcal{D}[\![e_1 : \tau_1]\!] : \mathcal{D}[\![\tau_1]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(g < e_2:\tau_2:\text{bool}, g' < \mathcal{D}[\![e_2:\tau_2]\!]:\mathcal{D}[\![\tau_2]\!]:\mathcal{D}[\![\text{bool}]\!]})}$
$e_1 : \tau_1 < e_2 : \tau_2 : \text{bool}$ for $\tau_1 = \text{float}[B; V_1], \tau_2 = \text{float}[B; V_2]$ $B \neq \top$	$e_1 = v_1, e_2 = v_2, v_1 < v_2$ $e_1 = v_1, e_2 = v_2, v_1 \geq v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{(\text{true}:\text{bool}, \text{true}:\text{bool})}$ $\delta_{(\text{false}:\text{bool}, \text{false}:\text{bool})}$ $\llbracket e_2 : \tau_2, \mathcal{D}[\![e_2 : \tau_2]\!] : \mathcal{D}[\![\tau_2]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(v_1:\tau_1 < g:\text{bool}, \mathcal{D}[\![v_1:\tau_1]\!]:\mathcal{D}[\![\tau_1]\!] <_{\#(B +1)} g':\mathcal{D}[\![\text{bool}]\!]}$ $\llbracket e_1 : \tau_1, \mathcal{D}[\![e_1 : \tau_1]\!] : \mathcal{D}[\![\tau_1]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(g < e_2:\tau_2:\text{bool}, g' <_{\#(B +1)} \mathcal{D}[\![e_2:\tau_2]\!]:\mathcal{D}[\![\tau_2]\!]:\mathcal{D}[\![\text{bool}]\!]})}$
uniform ($e_1 : \tau_1, e_2 : \tau_2$) : float [$B; V$] $B = \top$	$e_1 = v_1 < v_2 = e_2$ $e_1 = v_1 \geq v_2 = e_2$ $e_1 = v_1, e_2$ no val. e_1 no val.	$\text{Uniform}(v_1, v_2) \gg \lambda v. \delta_{(v:\text{float}[B; V], v:\text{float}[B; V])}$ $\llbracket \text{diverge} : \tau, \mathcal{D}[\![\text{uniform}(v_1 : \tau_1, v_2 : \tau_2)]\!] : \mathcal{D}[\![\tau]\!] \rrbracket$ $\llbracket e_2 : \tau_2, \mathcal{D}[\![e_2]\!] : \mathcal{D}[\![\tau_2]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{uniform}(v_1:\tau_1, g:\tau_2):\tau, \text{uniform}(\mathcal{D}[\![v_1]\!]:\mathcal{D}[\![\tau_1]\!], g':\mathcal{D}[\![\tau_2]\!]):\mathcal{D}[\![\tau]\!]}$ $\llbracket e_1 : \tau_1, \mathcal{D}[\![e_1]\!] : \mathcal{D}[\![\tau_1]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{uniform}(g:\tau_1, e_2:\tau_2):\tau, \text{uniform}(g':\mathcal{D}[\![\tau_1]\!], \mathcal{D}[\![e_2]\!]:\mathcal{D}[\![\tau_2]\!]):\mathcal{D}[\![\tau]\!]})}$
uniform ($e_1 : \tau_1, e_2 : \tau_2$) : float [$B; V$] $B \neq \top$ $\tau_1 = \text{float}[B_1 \neq \top, \{v_1, \dots, v_n\}]$ $\tau_2 = \text{float}[B_2 \neq \top, \{w_1, \dots, w_m\}]$	$e_1 = c_1 < c_2 = e_2$ values $e_1 = v_1 \geq v_2 = e_2$ $e_1 = c_1, e_2$ no val. e_1 no val.	$\text{Uniform}(c_1, c_2) \gg \lambda v. \delta_{(v:\text{float}[B; V], v_i:\text{fin}(B +1))}$ where v_i is unique number s.t. $v \in \text{Intervals}(B)_{v_i}$ $\llbracket \text{diverge} : \tau, \mathcal{D}[\![\text{uniform}(v_1 : \tau_1, v_2 : \tau_2)]\!] : \mathcal{D}[\![\tau]\!] \rrbracket$ $\llbracket e_2 : \tau_2, \mathcal{D}[\![e_2]\!] : \mathcal{D}[\![\tau_2]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{uniform}(v_1:\tau_1, g:\tau_2):\tau, \text{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}[\![v_1]\!]:\mathcal{D}[\![\tau_1]\!], g':\mathcal{D}[\![\tau_2]\!]):\mathcal{D}[\![\tau]\!]}$ $\llbracket e_1 : \tau_1, \mathcal{D}[\![e_1]\!] : \mathcal{D}[\![\tau_1]\!] \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{uniform}(g:\tau_1, e_2:\tau_2):\tau, \text{DC}_{B, B_1, B_2, V_1, V_2}(g':\mathcal{D}[\![\tau_1]\!], \mathcal{D}[\![e_2]\!]:\mathcal{D}[\![\tau_2]\!]):\mathcal{D}[\![\tau]\!]})}$

LEMMA F.41. $(P_{\tau}, \mathcal{F}_{P_{\tau}})$ and $(P_{\tau} + \perp, \mathcal{F}_{P_{\tau} + \perp})$ are measurable spaces.

PROOF. The first claim follows from Definition F.1. The second follows from the same error-extension construction used for $\text{Expr}_{\tau} + \perp$. $\square$

Now, exploiting that $e : \tau$ is a value iff $\mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!]$ is a value, we define:

Definition F.42. The small-step coupling $\llbracket e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rrbracket \in \mathcal{D}(P_{\tau} + \perp)$ is defined recursively on the structure of $e : \tau$ by the rules in Tables 3 and 4.

LEMMA F.43 (MONADIC BIND IS A SUB-MARKOV KERNEL). For every occurrence of a monadic bind $\mu \gg (f, f')$ given in the small-step operational coupling semantics in Tables 3 and 4, $(f, f') : (P_{\tau_1} + \perp) \rightarrow \mathcal{D}(P_{\tau_2} + \perp)$ is a sub-Markov kernel.

Table 4. Small-step coupling semantics (continued). Here, every function of the form $f(g, g') = \lambda(g, g').\delta \dots$ is extended to $f_\perp$ in the sense of Definition F.29.

$e : \tau$	constraints	$\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket$
$e_1 : \tau_1 \ \&\& \ e_2 : \tau_2 : \mathbf{bool}$ $\tau_1 = \mathbf{bool}, \tau_2 = \mathbf{bool}$	$e_1 = b_1 = \text{true}, e_2 = b_2 = \text{true}$ $e_1 = b_1, e_2 = b_2$ $e_1 = b_1, e_2 \neq b_2$ o.w.	$\delta_{(\text{true}:\mathbf{bool}, \text{true}:\mathbf{bool})}$ $\delta_{(\text{false}:\mathbf{bool}, \text{false}:\mathbf{bool})}$ $\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 : \tau_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(b_1:\tau_1 \ \&\&g:\mathbf{bool}, b_1:\mathcal{D}\llbracket \tau_1 \rrbracket \ \&\&g':\mathcal{D}\llbracket \mathbf{bool} \rrbracket)}$ $\llbracket e_1 : \tau_1, \mathcal{D}\llbracket e_1 : \tau_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(g\ \&\&e_2:\tau_2:\mathbf{bool}, g' \ \&\&\mathcal{D}\llbracket e_2:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket:\mathcal{D}\llbracket \mathbf{bool} \rrbracket)}$
$e_1 : \tau_1 \ \ e_2 : \tau_2 : \mathbf{bool}$ $\tau_1 = \mathbf{bool}, \tau_2 = \mathbf{bool}$	$e_1 = b_1 = \text{false}, e_2 = b_2 = \text{false}$ $e_1 = b_1, e_2 = b_2$ $e_1 = b_1, e_2 \neq b_2$ o.w.	$\delta_{(\text{false}:\mathbf{bool}, \text{false}:\mathbf{bool})}$ $\delta_{(\text{true}:\mathbf{bool}, \text{true}:\mathbf{bool})}$ $\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 : \tau_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(b_1:\tau_1 \ g:\mathbf{bool}, b_1:\mathcal{D}\llbracket \tau_1 \rrbracket \ g':\mathcal{D}\llbracket \mathbf{bool} \rrbracket)}$ $\llbracket e_1 : \tau_1, \mathcal{D}\llbracket e_1 : \tau_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(g \ e_2:\tau_2:\mathbf{bool}, g' \ \mathcal{D}\llbracket e_2:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket:\mathcal{D}\llbracket \mathbf{bool} \rrbracket)}$
$\text{not } e : \tau : \mathbf{bool}$ $\tau = \mathbf{bool}$	$e = \text{true}$ $e = \text{false}$ o.w.	$\delta_{(\text{false}:\mathbf{bool}, \text{false}:\mathbf{bool})}$ $\delta_{(\text{true}:\mathbf{bool}, \text{true}:\mathbf{bool})}$ $\llbracket e : \tau, \mathcal{D}\llbracket e : \tau \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(\text{not } g:\mathbf{bool}, \text{not } g':\mathcal{D}\llbracket \mathbf{bool} \rrbracket)}$
$(e_1 : \tau_1, e_2 : \tau_2) : \tau_1 * \tau_2$	$e_1 = v_1, e_2 = v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{((v_1:\tau_1, v_2:\tau_2), (\mathcal{D}\llbracket v_1 \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket, \mathcal{D}\llbracket v_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket):\mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket)}$ $\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 : \tau_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{((v_1:\tau_1, g:\tau_2):v_1 * v_2, (\mathcal{D}\llbracket v_1 \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket, g':\mathcal{D}\llbracket \tau_2 \rrbracket):\mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket)}$ $\llbracket e_1 : \tau_1, \mathcal{D}\llbracket e_1 : \tau_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{((g, e_2:\tau_2):v_1 * v_2, (g', \mathcal{D}\llbracket e_2:\tau_2 \rrbracket):\mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket)}$
$\mathbf{fst} (e : \tau_1 * \tau_2) : \tau_1$	$e = (v_1, v_2)$ o.w.	$\delta_{(v_1:\tau_1, \mathcal{D}\llbracket v_1:\tau_1 \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket)}$ $\llbracket e : \tau_1 * \tau_2, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(\mathbf{fst} (g:\tau_1 * \tau_2):\tau_1, \mathbf{fst} (g':\mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket):\mathcal{D}\llbracket \tau_1 \rrbracket)}$
$\mathbf{snd} (e : \tau_1 * \tau_2) : \tau_2$	$e = (v_1, v_2)$ o.w.	$\delta_{(v_2:\tau_2, \mathcal{D}\llbracket v_2:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket)}$ $\llbracket e : \tau_1 * \tau_2, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(\mathbf{snd} (g:\tau_1 * \tau_2):\tau_2, \mathbf{snd} (g':\mathcal{D}\llbracket \tau_1 * \tau_2 \rrbracket):\mathcal{D}\llbracket \tau_2 \rrbracket)}$
$\mathbf{fun} x \rightarrow e : \tau_1 \rightarrow \tau_2$		$\delta_{(\mathbf{fun} x \rightarrow e:\tau_1 \rightarrow \tau_2, \mathbf{fun} x \rightarrow \mathcal{D}\llbracket e \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket \rightarrow \mathcal{D}\llbracket \tau_2 \rrbracket)}$
$\mathbf{fix} g x := e : \tau_1 \rightarrow \tau_2$		$\delta_{(\mathbf{fix} g x := e:\tau_1 \rightarrow \tau_2, \mathbf{fix} g x := \mathcal{D}\llbracket e \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket \rightarrow \mathcal{D}\llbracket \tau_2 \rrbracket)}$
$(e_1 : \tau_1 \rightarrow \tau_2 \ e_2 : \tau_1) : \tau_2$	$e_1 = \mathbf{fun} x \rightarrow e \text{ or } \mathbf{fix} f x := e,$ $e_2 = v_2$ $e_1 = \mathbf{fun} x \rightarrow e \text{ or } \mathbf{fix} f x := e,$ $e_2 \neq v_2$ o.w.	$\delta_{(e[v_2/x]:\tau_2, \mathcal{D}\llbracket e[v_2/x] \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket)}$ $\llbracket e_2 : \tau_1, \mathcal{D}\llbracket e_2 : \tau_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(e_1:\tau_1 \rightarrow \tau_2, g:\tau_1:\tau_2, \mathcal{D}\llbracket e_1 \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket \rightarrow \mathcal{D}\llbracket \tau_2 \rrbracket, g':\mathcal{D}\llbracket \tau_1 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket)}$ $\llbracket e_1 : \tau_1 \rightarrow \tau_2, \mathcal{D}\llbracket e_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rightarrow \tau_2 \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(g \ e_2:\tau_1:\tau_2, g' \ \mathcal{D}\llbracket e_2 \rrbracket:\mathcal{D}\llbracket \tau_1 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket)}$
$\mathbf{nil} : \mathbf{list}(\tau)$		$\delta_{(\mathbf{nil}:\mathbf{list}(\tau), \mathbf{nil}:\mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket))}$
$(e_1 : \tau :: e_2 : \mathbf{list}(\tau)) : \mathbf{list}(\tau)$	$e_1 = v_1, e_2 = v_2$ $e_1 = v_1, e_2 \neq v_2$ o.w.	$\delta_{(v_1::v_2:\mathbf{list}(\tau), \mathcal{D}\llbracket v_1 \rrbracket::\mathcal{D}\llbracket v_2 \rrbracket:\mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket))}$ $\llbracket e_2 : \mathbf{list}(\tau), \mathcal{D}\llbracket e_2 \rrbracket : \mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket) \rrbracket$ $\gg \lambda(g, g').\delta_{(v_1::g:\mathbf{list}(\tau), \mathcal{D}\llbracket v_1 \rrbracket::g':\mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket))}$ $\llbracket e_1 : \tau, \mathcal{D}\llbracket e_1 \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket$ $\gg \lambda(g, g').\delta_{(g::e_2:\mathbf{list}(\tau):\mathbf{list}(\tau), g'::\mathcal{D}\llbracket e_2 \rrbracket:\mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket):\mathbf{list}(\mathcal{D}\llbracket \tau \rrbracket))}$
$(\mathbf{match} e : \mathbf{list}(\tau_1) \text{ with}$ $ \ \mathbf{nil} \rightarrow e_1 : \tau_2$ $ \ h :: t \rightarrow e_2 : \tau_2$ $\mathbf{end}) : \tau_2$	$e = \mathbf{nil}$ $e = v_h :: v_t$ o.w.	$\delta_{(e_1:\tau_2, \mathcal{D}\llbracket e_1:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket)}$ $\delta_{(e_2[v_h/h, v_t/t]:\tau_2, \mathcal{D}\llbracket e_2[v_h/h, v_t/t] \rrbracket:\mathcal{D}\llbracket v_h \rrbracket/h.\mathcal{D}\llbracket v_t \rrbracket/t:\mathcal{D}\llbracket \tau_2 \rrbracket)}$ $\llbracket e : \mathbf{list}(\tau_1), \mathcal{D}\llbracket e \rrbracket : \mathbf{list}(\mathcal{D}\llbracket \tau_1 \rrbracket) \rrbracket$ $\gg \lambda(g, g').\delta_{(\mathbf{match} g \text{ with } [\mathbf{nil} \rightarrow e_1:\tau_2 \ \ h:t \rightarrow e_2:\tau_2 \ \mathbf{end}:\tau_2, \mathbf{match} g' \text{ with } [\mathbf{nil} \rightarrow \mathcal{D}\llbracket e_1:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket \ \ h:t \rightarrow \mathcal{D}\llbracket e_2:\tau_2 \rrbracket:\mathcal{D}\llbracket \tau_2 \rrbracket \ \mathbf{end}:\mathcal{D}\llbracket \tau_2 \rrbracket])}$

PROOF. Analogous to proof of Lemma F.38. □

LEMMA F.44 (SMALL-STEP COUPLING SEMANTICS IS A SUB-MARKOV KERNEL). *For each type τ , the small-step operational coupling semantics $\llbracket \cdot, \cdot \rrbracket : P_\tau \rightarrow \mathcal{D}(P_\tau + \perp)$ defines a sub-Markov kernel.*

PROOF. Analogous to proof of Lemma F.39. □

We now define suitable projection functions on measures $\mu \in \mathcal{D}(P_\tau + \perp)$ by

$$\pi_1(\mu) = \mu \gg \lambda(e_1, e_2). \delta_{e_1} \quad (1)$$

$$\pi_2(\mu) = \mu \gg \lambda(e_1, e_2). \delta_{e_2} . \quad (2)$$

We have the following key property:

LEMMA F.45 (SMALL-STEP COUPLING PROJECTION). *Let $e : \tau$. We have:*

- (1) $\pi_1(\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket) = \llbracket e : \tau \rrbracket$.
- (2) $\pi_2(\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket) = \llbracket \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket$.

PROOF. By induction on the type derivation of $e : \tau$. The most interesting case is

$$e : \tau = \mathbf{uniform}(c_1 : \mathbf{float}[B_1; V_1], c_2 : \mathbf{float}[B_2; V_2]) : \mathbf{float}[B; V] ,$$

where

- (1) $c_1 < c_2$,
- (2) $B = \{\sim_1 b_1, \dots, \sim_n b_n\}$,
- (3) $V = \top$,
- (4) $V_1 = \{v_1, \dots, v_{n_1}\}$ with $\text{recovers}(B_1, V_1)$ and $c_1 \in V_1$ due to $\text{has}(V_1, c_1)$ by I.H.
- (5) $V_2 = \{w_1, \dots, w_{n_2}\}$ with $\text{recovers}(B_2, V_2)$ and $c_2 \in V_2$ due to $\text{has}(V_2, c_2)$ by I.H.

First observe that we have

$$\begin{aligned} & \mathcal{D}\llbracket e : \tau \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \\ &= \mathbf{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}\llbracket c_1 : \mathbf{float}[B_1; V_1] \rrbracket, \mathcal{D}\llbracket c_2 : \mathbf{float}[B_2; V_2] \rrbracket) : \mathbf{fin}(|B| + 1) \\ &= \mathbf{DC}_{B, B_1, B_2, V_1, V_2}(k_1 : \mathbf{fin}(|B_1| + 1), k_2 : \mathbf{fin}(|B_2| + 1)) : \mathbf{fin}(|B| + 1) , \end{aligned}$$

where k_i is the unique number such that $c_i \in \text{Intervals}(B_i)_{k_i}$. By Table 1 and Figure 13, this yields

$$\begin{aligned} & \llbracket \mathcal{D}\llbracket e : \tau \rrbracket \rrbracket \\ &= \llbracket \mathbf{discrete}(p_0, \dots, p_n) \rrbracket \text{ where } p_k = \text{CDF}(b_{k+1}) - \text{CDF}(b_k) = \frac{b_{k+1} - c_1}{c_2 - c_1} - \frac{b_k - c_1}{c_2 - c_1} = \frac{b_{k+1} - b_k}{c_2 - c_1} . \end{aligned}$$

For Lemma F.45.2, consider the following:

$$\begin{aligned} & \pi_1(\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket) \\ &= \llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket \gg \lambda(e_1, e_2). \delta_{e_1} \quad (\text{definition}) \\ &= (\mathbf{Uniform}(c_1, c_2) \gg \lambda v. \delta_{(v : \mathbf{float}[B; V], v_i : \mathbf{fin}(|B| + 1))}) \gg \lambda(e_1, e_2). \delta_{e_1} \\ & \quad (\text{Table 3, } v_i \text{ is unique number s.t. } v \in \text{Intervals}(B)_{v_i}) \\ &= \mathbf{Uniform}(c_1, c_2) \gg \lambda v. (\delta_{(v : \mathbf{float}[B; V], v_i : \mathbf{fin}(|B| + 1))} \gg \lambda(e_1, e_2). \delta_{e_1}) \\ & \quad (\text{monad assoc., } v_i \text{ is unique number s.t. } v \in \text{Intervals}(B)_{v_i}) \\ &= \mathbf{Uniform}(c_1, c_2) \gg \lambda v. \delta_{v : \mathbf{float}[B; V]} \\ &= \mathbf{Uniform}(c_1, c_2) \\ &= \llbracket e : \tau \rrbracket . \quad (\text{Table 1}) \end{aligned}$$

For Lemma F.45.2, consider the following:

$$\begin{aligned}
& \pi_2(\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket) \\
&= \llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket \ggg \lambda(e_1, e_2). \delta_{e_2} \quad (\text{definition}) \\
&= (\text{Uniform}(c_1, c_2) \ggg \lambda v. \delta_{(v : \text{float}[B; V], v_i : \text{fin}(|B|+1))}) \ggg \lambda(e_1, e_2). \delta_{e_2} \\
&\quad (\text{Table 3, } v_i \text{ is unique number s.t. } v \in \text{Intervals}(B)_{v_i}) \\
&= \text{Uniform}(c_1, c_2) \ggg \lambda v. (\delta_{(v : \text{float}[B; V], v_i : \text{fin}(|B|+1))}) \ggg \lambda(e_1, e_2). \delta_{e_2} \\
&\quad (\text{monad assoc., } v_i \text{ is unique number s.t. } v \in \text{Intervals}(B)_{v_i}) \\
&= \text{Uniform}(c_1, c_2) \ggg \lambda v. \delta_{v_i : \text{fin}(|B|+1)} \\
&\quad (\text{monad return, } v_i \text{ is unique number s.t. } v \in \text{Intervals}(B)_{v_i}) \\
&= \llbracket \text{discrete}(p_0, \dots, p_n) \rrbracket \text{ where } p_k = \text{CDF}(b_{k+1}) - \text{CDF}(b_k) = \frac{b_{k+1} - c_1}{c_2 - c_1} - \frac{b_k - c_1}{c_2 - c_1} = \frac{b_{k+1} - b_k}{c_2 - c_1} \\
&\quad (\text{by construction}) \\
&= \llbracket \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket. \quad (\text{Table 1})
\end{aligned}$$

Now, the remaining cases are either (i) trivial base cases or (ii) immediately follow from the I.H. and Lemma F.46 further below. We demonstrate this for the case

$$e : \tau = \text{uniform}(e_1 : \text{float}[B_1; V_1], e_2 : \text{float}[B_2; V_2]) : \text{float}[B; V],$$

where

- (1) $e_1 = c_1$ is a value but e_2 is no value,
- (2) $B = \{\sim_1 b_1, \dots, \sim_n b_n\}$,
- (3) $V = \top$,
- (4) $V_1 = \{v_1, \dots, v_{n_1}\}$ with $\text{recovers}(B_1, V_1)$ and $c_1 \in V_1$ due to $\text{has}(V_1, c_1)$ by I.H.
- (5) $V_2 = \{w_1, \dots, w_{n_2}\}$ with $\text{recovers}(B_2, V_2)$ and $c_2 \in V_2$ due to $\text{has}(V_2, c_2)$ by I.H.

For Lemma F.45.2, consider the following:

$$\begin{aligned}
& \pi_1(\llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket) \\
&= \llbracket e : \tau, \mathcal{D}\llbracket e \rrbracket : \mathcal{D}\llbracket \tau \rrbracket \rrbracket \ggg \lambda(e_1, e_2). \delta_{e_1} \quad (\text{definition}) \\
&= (\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket \\
&\quad \ggg \lambda(f, f'). \delta_{(\text{uniform}(v_1 : \tau_1, f : \tau_2) : \tau, \text{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}\llbracket v_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket, f' : \mathcal{D}\llbracket \tau_2 \rrbracket) : \mathcal{D}\llbracket \tau \rrbracket)}) \ggg \lambda(e_1, e_2). \delta_{e_1} \\
&= \llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket \\
&\quad \ggg \lambda(f, f'). (\delta_{(\text{uniform}(v_1 : \tau_1, f : \tau_2) : \tau, \text{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}\llbracket v_1 \rrbracket : \mathcal{D}\llbracket \tau_1 \rrbracket, f' : \mathcal{D}\llbracket \tau_2 \rrbracket) : \mathcal{D}\llbracket \tau \rrbracket)}) \ggg \lambda(e_1, e_2). \delta_{e_1} \\
&\quad (\text{monad assoc.}) \\
&= \llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket \ggg \lambda(f, f'). \delta_{\text{uniform}(v_1 : \tau_1, f : \tau_2) : \tau} \quad (\text{monad return}) \\
&= \pi_1(\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket) \ggg \lambda f. \delta_{\text{uniform}(v_1 : \tau_1, f : \tau_2) : \tau} \quad (\text{Lemma F.46.1}) \\
&= \llbracket e_2 : \tau_2 \rrbracket \ggg \lambda f. \delta_{\text{uniform}(v_1 : \tau_1, f : \tau_2) : \tau} \cdot (\llbracket e_2 : \tau_2, \mathcal{D}\llbracket e_2 \rrbracket : \mathcal{D}\llbracket \tau_2 \rrbracket \rrbracket = \llbracket e_2 : \tau_2 \rrbracket \text{ by I.H.}) \\
&= \llbracket e : \tau \rrbracket \quad (\text{Table 1})
\end{aligned}$$

□

LEMMA F.46. Let $F : P_\tau \rightarrow \mathcal{D}(P_\tau + \perp)$, let $G : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$, and let $G' : \text{Expr}_{\mathcal{D}\llbracket \tau \rrbracket} \rightarrow \mathcal{D}(\text{Expr}_{\mathcal{D}\llbracket \tau \rrbracket} + \perp)$. We have:

- (1) If $\forall (f, f') \in P_\tau : F(f, f') = G(f)$, then for all $\mu \in \mathcal{D}(P_\tau + \perp)$,

$$\pi_1(\mu) \ggg \lambda f. G(f) = \mu \ggg \lambda (f, f'). F(f, f') .$$

(2) If $\forall (f, f') \in P_\tau : F(f, f') = G'(f)$, then for all $\mu \in \mathcal{D}(P_\tau + \perp)$,

$$\pi_2(\mu) \ggg \lambda f.G'(f) = \mu \ggg \lambda(f, f').F(f, f') .$$

PROOF. We only prove Lemma F.46.1 because the reasoning for Lemma F.46.2 is completely analogous. We have

$$\begin{aligned} \pi_1(\mu) &\ggg \lambda f.G(f) \\ &= (\mu \ggg \lambda(f, f').\delta_f) \ggg \lambda f.G(f) && \text{(definition)} \\ &= \mu \ggg \lambda(f, f').(\delta_f \ggg \lambda f.G(f)) && \text{(monad assoc.)} \\ &= \mu \ggg \lambda(f, f').G(f) && \text{(monad neutral)} \\ &= \mu \ggg \lambda(f, f').F(f, f') . && \text{(assumption)} \end{aligned}$$

□

F.6 Big-Step Semantics and Correctness of Discretization

We extend both of the small-step semantics canonically to an n -step semantics by

$$\begin{aligned} \langle e : \tau \rangle_n &= \begin{cases} \delta_{e:\tau} & \text{if } n = 0 \\ \langle e : \tau \rangle_{n-1} \ggg \lambda e'. \llbracket e' : \tau \rrbracket & \text{if } n > 0 . \end{cases} \\ \langle e : \tau, \mathcal{D} \llbracket e \rrbracket : \mathcal{D} \llbracket \tau \rrbracket \rangle_n &= \begin{cases} \delta_{(e:\tau, \mathcal{D} \llbracket e \rrbracket : \mathcal{D} \llbracket \tau \rrbracket)} & \text{if } n = 0 \\ \langle e : \tau, \mathcal{D} \llbracket e \rrbracket : \mathcal{D} \llbracket \tau \rrbracket \rangle_{n-1} \ggg \lambda(e', e''). \llbracket e' : \tau, e'' : \mathcal{D} \llbracket \tau \rrbracket \rrbracket & \text{if } n > 0 . \end{cases} \end{aligned}$$

As a sanity check, we show that the n -step semantics defines a sub-Markov kernel.

LEMMA F.47 (n -STEP SEMANTICS IS A SUB-MARKOV KERNEL). *For all $n \in \mathbb{N}$, $\langle \cdot \rangle_n : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$ is a sub-Markov kernel.*

PROOF. By induction on n .

Base case: $n = 0$. By definition,

$$\langle e : \tau \rangle_n = \delta_{e:\tau}$$

i.e. the map $e \mapsto \delta_e$. By Lemma F.31, this is a sub-Markov kernel.

Inductive step: $n > 0$. By definition,

$$\langle e : \tau \rangle_n = \langle e : \tau \rangle_{n-1} \ggg \lambda e'. \llbracket e' : \tau \rrbracket$$

By the I.H., $\langle \cdot \rangle_{n-1} : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$ is a sub-Markov kernel. By Lemma F.39, $\lambda e'. \llbracket e' : \tau \rrbracket : \text{Expr}_\tau \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$ is a sub-Markov kernel. By Lemma F.30, its $\perp$ -extension from $(\text{Expr}_\tau + \perp)$ to $\mathcal{D}(\text{Expr}_\tau + \perp)$ is also a sub-Markov kernel. By Lemma F.33, $\langle e : \tau \rangle_{n-1} \ggg \lambda e'. \llbracket e' : \tau \rrbracket$ is a sub-Markov kernel. □

LEMMA F.48 (n -STEP COUPLING SEMANTICS IS A SUB-MARKOV KERNEL). *For all $n \in \mathbb{N}$, $\langle \cdot, \cdot \rangle_n : P_\tau \rightarrow \mathcal{D}(P_\tau + \perp)$ is a sub-Markov kernel.*

PROOF. Analogous to proof of Lemma F.47. □

For every type τ , we denote by Vals_τ the set of values of type τ . By Lemma F.27 and closure under finite unions, $\text{Vals}_\tau + \perp$ and $\text{Expr}_\tau \setminus \text{Vals}_\tau$ are measurable subsets of $\text{Expr}_\tau + \perp$, which we equip with their subspace σ -algebras.

Furthermore, let

$$K_\tau : (\text{Expr}_\tau + \perp) \rightarrow \mathcal{D}(\text{Expr}_\tau + \perp)$$

be the $\perp$ -extension of the small-step kernel:

$$K_\tau(x, A) = \begin{cases} \llbracket x : \tau \rrbracket(A) & x \in \text{Expr}_\tau, \\ \delta_\perp(A) & x = \perp, \end{cases} \quad A \in \mathcal{F}_{\text{Expr}_\tau + \perp}.$$

Thus, for every $e : \tau$,

$$\langle e : \tau \rangle_0 = \delta_e$$

and

$$\langle e : \tau \rangle_{n+1}(A) = \int_{\text{Expr}_\tau + \perp} K_\tau(x, A) \langle e : \tau \rangle_n(dx).$$

LEMMA F.49 (MONOTONICITY OF N-STEP SEMANTICS). *For every $e : \tau$, $n \in \mathbb{N}$, and $A \in \mathcal{F}_{\text{Vals}_\tau + \perp}$,*

$$\langle e : \tau \rangle_n(A) \leq \langle e : \tau \rangle_{n+1}(A).$$

PROOF. Fix $e : \tau$, $n \in \mathbb{N}$, and $A \in \mathcal{F}_{\text{Vals}_\tau + \perp}$. Write

$$\mu_n := \langle e : \tau \rangle_n.$$

Every $x \in \text{Vals}_\tau + \perp$ is absorbing:

$$K_\tau(x, -) = \delta_x.$$

Indeed, $\llbracket v : \tau \rrbracket = \delta_v$ for every $v \in \text{Vals}_\tau$, while the $\perp$ -extension maps $\perp$ to $\delta_\perp$. Therefore,

$$\begin{aligned} \langle e : \tau \rangle_{n+1}(A) &= \int_{\text{Expr}_\tau + \perp} K_\tau(x, A) \mu_n(dx) && \text{(definition of the } n\text{-step semantics)} \\ &= \int_{\text{Vals}_\tau + \perp} K_\tau(x, A) \mu_n(dx) + \int_{\text{Expr}_\tau \setminus \text{Vals}_\tau} K_\tau(x, A) \mu_n(dx) && \text{(partition of } \text{Expr}_\tau + \perp) \\ &= \int_{\text{Vals}_\tau + \perp} \delta_x(A) \mu_n(dx) + \int_{\text{Expr}_\tau \setminus \text{Vals}_\tau} K_\tau(x, A) \mu_n(dx) && \text{(terminal states are absorbing)} \\ &= \int_{\text{Vals}_\tau + \perp} \mathbf{1}_A(x) \mu_n(dx) + \int_{\text{Expr}_\tau \setminus \text{Vals}_\tau} K_\tau(x, A) \mu_n(dx) && (\delta_x(A) = \mathbf{1}_A(x)) \\ &= \mu_n(A) + \int_{\text{Expr}_\tau \setminus \text{Vals}_\tau} K_\tau(x, A) \mu_n(dx) && (A \subseteq \text{Vals}_\tau + \perp) \\ &\geq \mu_n(A) && \text{(nonnegativity)} \\ &= \langle e : \tau \rangle_n(A). && \text{(definition of } \mu_n) \end{aligned}$$

□

LEMMA F.50. *For every $e : \tau$,*

$$\langle e : \tau \rangle \in \mathcal{D}(\text{Vals}_\tau + \perp).$$

That is, $\langle e : \tau \rangle$ is a subprobability measure on $\text{Vals}_\tau + \perp$.

PROOF. Fix $e : \tau$, and write

$$\mu_n := \langle e : \tau \rangle_n.$$

Each μ_n is a subprobability measure on $\text{Expr}_\tau + \perp$. Since $\text{Vals}_\tau + \perp$ is measurable, every $A \in \mathcal{F}_{\text{Vals}_\tau + \perp}$ is also measurable in $\text{Expr}_\tau + \perp$.

First,

$$\begin{aligned} \langle e : \tau \rangle(\emptyset) &= \lim_{n \rightarrow \infty} \mu_n(\emptyset) && \text{(definition of } \langle e : \tau \rangle) \\ &= \lim_{n \rightarrow \infty} 0 && (\mu_n \text{ is a measure}) \\ &= 0. \end{aligned}$$

Now let

$$(A_i)_{i \in \mathbb{N}} \subseteq \mathcal{F}_{Vals_\tau + \perp}$$

be pairwise disjoint. Then

$$\begin{aligned} \langle e : \tau \rangle\left(\bigcup_{i \in \mathbb{N}} A_i\right) &= \lim_{n \rightarrow \infty} \mu_n\left(\bigcup_{i \in \mathbb{N}} A_i\right) && \text{(definition of } \langle e : \tau \rangle) \\ &= \lim_{n \rightarrow \infty} \sum_{i \in \mathbb{N}} \mu_n(A_i) && \text{(countable additivity of } \mu_n) \\ &= \sum_{i \in \mathbb{N}} \lim_{n \rightarrow \infty} \mu_n(A_i) && \text{(monotone convergence on } \mathbb{N}) \\ &= \sum_{i \in \mathbb{N}} \langle e : \tau \rangle(A_i). && \text{(definition of } \langle e : \tau \rangle) \end{aligned}$$

The monotone-convergence step applies because

$$\mu_n(A_i) \leq \mu_{n+1}(A_i)$$

for every $i, n \in \mathbb{N}$, by Lemma F.49. Thus $\langle e : \tau \rangle$ is countably additive and therefore a measure on $Vals_\tau + \perp$.

Finally,

$$\begin{aligned} \langle e : \tau \rangle(Vals_\tau + \perp) &= \lim_{n \rightarrow \infty} \mu_n(Vals_\tau + \perp) && \text{(definition of } \langle e : \tau \rangle) \\ &\leq 1. && (\mu_n \text{ is a subprobability measure}) \end{aligned}$$

Therefore $\langle e : \tau \rangle$ is a subprobability measure on $Vals_\tau + \perp$. $\square$

Moreover, let

$$V_\tau = \{(v : \tau, \mathcal{D}[\![v]\!] : \mathcal{D}[\![\tau]\!]) \mid v : \tau\}.$$

We equip $V_\tau + \perp$ with the subspace σ -algebra inherited from $P_\tau + \perp$.

LEMMA F.51 (MONOTONICITY OF N-STEP COUPLING SEMANTICS). *For every $(e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!]) \in P_\tau$, $n \in \mathbb{N}$, and $A \in \mathcal{F}_{V_\tau + \perp}$,*

$$\langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_n(A) \leq \langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_{n+1}(A).$$

PROOF. The coupled small-step semantics maps every element of V_τ to its Dirac measure, while the $\perp$ -extension maps $\perp$ to $\delta_\perp$. Thus every element of $V_\tau + \perp$ is absorbing. The remainder is analogous to the proof of Lemma F.49, replacing $Expr_\tau + \perp$ by $P_\tau + \perp$ and $Vals_\tau + \perp$ by $V_\tau + \perp$. $\square$

LEMMA F.52. *For every $(e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!]) \in P_\tau$,*

$$\langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle \in \mathcal{D}(V_\tau + \perp).$$

That is, the coupled big-step semantics is a subprobability measure on $V_\tau + \perp$.

PROOF. Analogous to the proof of Lemma F.50, replacing $Expr_\tau + \perp$ by $P_\tau + \perp$, $Vals_\tau + \perp$ by $V_\tau + \perp$, and using Lemma F.51 for monotonicity. $\square$

Definition F.53 (Big-step semantics). Let $e : \tau$. For measurable events in the respective terminal spaces, the source and coupled (*big-step*) semantics are defined by

$$\begin{aligned} \langle e : \tau \rangle(A) &:= \lim_{n \rightarrow \infty} \langle e : \tau \rangle_n(A), & A \in \mathcal{F}_{V_{\text{als}_\tau + \perp}}, \\ \langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle(A) &:= \lim_{n \rightarrow \infty} \langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_n(A), & A \in \mathcal{F}_{V_\tau + \perp}. \end{aligned}$$

By Lemmas F.49 and F.51, the source and coupled n -step semantics are monotonically increasing on every measurable event in $V_{\text{als}_\tau + \perp}$ and $V_\tau + \perp$, respectively. By Lemmas F.50 and F.52, their pointwise limits are subprobability measures.

LEMMA F.54 (CONTINUITY OF PROJECTIONS). *For all $n \in \mathbb{N}$,*

- (1) $\pi_1(\lim_{n \rightarrow \infty} \langle e, e' \rangle_n) = \lim_{n \rightarrow \infty} \pi_1(\langle e, e' \rangle_n)$
- (2) $\pi_2(\lim_{n \rightarrow \infty} \langle e, e' \rangle_n) = \lim_{n \rightarrow \infty} \pi_2(\langle e, e' \rangle_n)$

PROOF. We prove part i) in detail; the proof of part ii) is analogous. Fix an arbitrary measurable $A \in \Sigma_1$:

$$\begin{aligned} &\pi_1(\lim_{n \rightarrow \infty} \langle e, e' \rangle_n)(A) \\ &= \left(\lim_{n \rightarrow \infty} \langle e, e' \rangle_n \right) (A \times X_2) && \text{(by definition of } \pi_1) \\ &= \sup_{n \in \mathbb{N}} \langle e, e' \rangle_n (A \times X_2) && \text{(since } \langle e, e' \rangle_n \text{ is monotone)} \\ &= \sup_{n \in \mathbb{N}} \pi_1(\langle e, e' \rangle_n)(A) && \text{(by definition of } \pi_1) \\ &= \lim_{n \rightarrow \infty} \pi_1(\langle e, e' \rangle_n)(A) && \text{(since } \pi_1(\langle e, e' \rangle_n) \text{ is monotone)} \end{aligned}$$

Since A was arbitrary, this implies:

$$\pi_1(\lim_{n \rightarrow \infty} \langle e, e' \rangle_n) = \lim_{n \rightarrow \infty} \pi_1(\langle e, e' \rangle_n),$$

as desired. □

LEMMA F.55 (n -STEP COUPLING PROJECTION). *For all $n \in \mathbb{N}$,*

- (1) $\pi_1(\langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_n) = \langle e : \tau \rangle_n$
- (2) $\pi_2(\langle e : \tau, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_n) = \langle \mathcal{D}[\![e]\!] : \mathcal{D}[\![\tau]\!] \rangle_n$

PROOF. By induction on n . We prove i), as the proof for ii) is analogous.

Base case: $n = 0$

$$\begin{aligned} &\pi_1(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle_0) \\ &= \pi_1(\delta_{(e:\tau, \mathcal{D}[\![e']]\!:\mathcal{D}[\![\tau']]\!)}) && \text{(definition of } n\text{-step coupling semantics)} \\ &= \delta_{e:\tau} && \text{(projection of Dirac)} \\ &= \langle e : \tau \rangle_0. && \text{(definition of } n\text{-step semantics)} \end{aligned}$$

Inductive step: $n > 0$

$$\begin{aligned}
& \pi_1(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle_n) \\
= & \pi_1(\langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).[\![f : \tau, g : \mathcal{D}[\![\tau]]]\!]\rangle) \\
& \text{(definition of } n\text{-step coupling semantics)} \\
= & (\langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).[\![f : \tau, g : \mathcal{D}[\![\tau]]]\!]\rangle \gg \lambda(f, g).\delta_f) \quad \text{(by (2))} \\
= & \langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).([\![f : \tau, g : \mathcal{D}[\![\tau]]]\!] \gg \lambda(f, g).\delta_f) \quad \text{(monad assoc.)} \\
= & \langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).\pi_1([\![f, g]\!]) \quad \text{(by (2))} \\
= & \langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).[\![f]\!] \quad \text{(Lemma F.45.1)} \\
= & \langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).(\delta_f \gg \lambda f.[\![f]\!]) \quad \text{(monad unit)} \\
= & (\langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1} \gg \lambda(f, g).\delta_f) \gg \lambda f.[\![f]\!] \quad \text{(monad assoc.)} \\
= & \pi_1(\langle e : \tau, \mathcal{D}[\![e]] : \mathcal{D}[\![\tau]] \rangle_{n-1}) \gg \lambda f.[\![f]\!] \quad \text{(by (2))} \\
= & \langle e : \tau \rangle_{n-1} \gg \lambda f.[\![f]\!] \quad \text{(by I.H.)} \\
= & \langle e : \tau \rangle_n \quad \text{(definition of } n\text{-step semantics)}
\end{aligned}$$

□

Before concluding the soundness proof, we relate the auxiliary statement $\text{DC}_{B, B_1, B_2, V_1, V_2}$ used in the coupling semantics to the nested target expression generated by the discretization function in Figure 13.

LEMMA F.56. *Suppose an expression $F(e_1, e_2)$ evaluates e_1 first, then e_2 , and, when both arguments are values v_1, v_2 , continues as $q(v_1, v_2)$. Then*

$$\langle F(e_1, e_2) \rangle = \langle e_1 \rangle \gg \lambda v_1. \langle e_2 \rangle \gg \lambda v_2. \langle q(v_1, v_2) \rangle.$$

All functions bound by $\gg$ are extended to $\perp$ as in Definition F.29.

PROOF. By induction on the n -step semantics, using the small-step rules for evaluating the first and second arguments and associativity of $\gg$. Taking the limit as $n \rightarrow \infty$ gives the result. □

THEOREM F.57 ($\text{DC}_{B, B_1, B_2, V_1, V_2}$ IS A VALID CONSTRUCT). *Let*

$$e = \text{uniform}(e_1 : \text{float}[B_1; V_1], e_2 : \text{float}[B_2; V_2]) : \text{float}[B; \top]$$

*be well typed, where $B, B_1, B_2 \neq \top$ and $V_1, V_2 \neq \top$. Let e_{case} be the nested **let**- and **if**-expression in the last case of Figure 13. Then*

$$\langle \text{DC}_{B, B_1, B_2, V_1, V_2}(\mathcal{D}[\![e_1 : \text{float}[B_1; V_1]]\!], \mathcal{D}[\![e_2 : \text{float}[B_2; V_2]]\!]) : \text{fin}(|B|+1) \rangle = \langle e_{\text{case}} : \text{fin}(|B|+1) \rangle.$$

PROOF. Write

$$d_1 = \mathcal{D}[\![e_1 : \text{float}[B_1; V_1]]\!] \quad \text{and} \quad d_2 = \mathcal{D}[\![e_2 : \text{float}[B_2; V_2]]\!].$$

For arbitrary values $k_1 : \text{fin}(|B_1| + 1)$ and $k_2 : \text{fin}(|B_2| + 1)$, define

$$q(k_1, k_2) = \begin{cases} \mathcal{D}[\![\text{uniform}(v_i, w_j) : \text{float}[B; \top]]\!] & \begin{array}{l} \text{if there exists } (v_i, w_j) \in V_1 \times V_2 \text{ such that} \\ k_1 = \mathcal{D}[\![v_i : \text{float}[B_1; V_1]]\!] \text{ and} \\ k_2 = \mathcal{D}[\![w_j : \text{float}[B_2; V_2]]\!], \end{array} \\ \text{diverge} & \text{if no such pair exists.} \end{cases}$$

When such a pair exists, it is unique because $\text{recovers}(B_1, V_1)$ and $\text{recovers}(B_2, V_2)$; if $V_1 = \emptyset$ or $V_2 = \emptyset$, no such pair exists and hence $q(k_1, k_2) = \text{diverge}$ for all k_1, k_2 .

By the operational rule for $\text{DC}_{B,B_1,B_2,V_1,V_2}$ and Lemma F.56,

$$\langle \text{DC}_{B,B_1,B_2,V_1,V_2}(d_1, d_2) \rangle = \langle d_1 \rangle \gg \lambda k_1. \langle d_2 \rangle \gg \lambda k_2. \langle q(k_1, k_2) \rangle. \quad (1)$$

Now consider e_{case} . Its two outer **let**-bindings evaluate d_1 and d_2 in the same order. Once they evaluate to k_1 and k_2 , finite equality and conjunction select the unique branch indexed by (v_i, w_j) when a matching pair exists; otherwise, including whenever $V_1 = \emptyset$ or $V_2 = \emptyset$, the final branch yields **diverge**. Thus the remaining nested conditional evaluates to $q(k_1, k_2)$. Applying Lemma F.56 again gives

$$\langle e_{\text{case}} \rangle = \langle d_1 \rangle \gg \lambda k_1. \langle d_2 \rangle \gg \lambda k_2. \langle q(k_1, k_2) \rangle. \quad (2)$$

The right-hand sides of (1) and (2) are identical. Therefore,

$$\langle \text{DC}_{B,B_1,B_2,V_1,V_2}(d_1, d_2) \rangle = \langle e_{\text{case}} \rangle. \quad \square$$

The following will imply our desired equivalence claim:

LEMMA F.58 (BIG-STEP COUPLING PROJECTION). *Let $\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle \in \mathcal{D}(P_\tau + \perp)$. Then,*

- (1) $\pi_1(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle = \langle e : \tau \rangle$
- (2) $\pi_2(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle = \langle \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle$

PROOF.

$$\begin{aligned} & \pi_1(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!] \rangle \\ &= \pi_1(\lim_{n \rightarrow \infty} \langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!]_n \rangle && \text{(by definition of big-step coupling semantics)} \\ &= \lim_{n \rightarrow \infty} \pi_1(\langle e : \tau, \mathcal{D}[\![e']]\!] : \mathcal{D}[\![\tau']]\!]_n \rangle && \text{(by commutativity lemma)} \\ &= \lim_{n \rightarrow \infty} \langle e : \tau \rangle_n && \text{(by definition of n-step projection lemma)} \\ &= \langle e : \tau \rangle && \text{(by definition of big-step semantics)} \end{aligned} \quad \square$$

THEOREM F.59. *Let $e : \mathbf{bool}$. Then $\langle e : \mathbf{bool} \rangle = \langle \mathcal{D}[\![e : \mathbf{bool}]\!] : \mathbf{bool} \rangle$.*

PROOF. Since $\mathcal{D}(\text{Vals}_\tau + \perp)$ contains only discrete distributions, we proceed by case distinction on the probability of the final outcome $v \in \{\text{true}, \text{false}, \perp\}$.

The case $v = \text{true}$. We have

$$\begin{aligned} & \langle e : \mathbf{bool} \rangle(\text{true}) \\ &= \pi_1(\langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle)(\text{true}) && \text{(Lemma F.58.1)} \\ &= (\langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle \gg \lambda(e_1, e_2). \delta_{e_1})(\text{true}) && \text{(definition)} \\ &= \langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle((\text{true}, \text{true})) \cdot \delta_{\text{true}}(\text{true}) \\ &\quad + \langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle((\text{false}, \text{false})) \cdot \delta_{\text{false}}(\text{true}) \\ &\quad + \langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle(\perp) \cdot \delta_\perp(\text{true}) && \text{(definition of } \gg \text{)} \\ &= \langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle((\text{true}, \text{true})) && \text{(definition of } \delta \text{)} \\ &= \pi_2(\langle e : \mathbf{bool}, \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle)(\text{true}) && \text{(reasoning completely analogous)} \\ &= \langle \mathcal{D}[\![e]\!] : \mathcal{D}[\![\mathbf{bool}]\!] \rangle(\text{true}). && \text{(Lemma F.58.2)} \end{aligned}$$

The cases $v = \text{false}$ and $v = \perp$ are completely analogous. $\square$

G Benchmarks

G.1 Asymptotic Scaling

G.1.1 Code Sketches for Scaling Benchmarks. This section contains code snippets of the programs (Figures 29a to 29d and 30a to 30d) used in the scaling experiments in §7.2. We implemented a Python script that automatically generates these programs based on configurable parameters such as program size, guard structure, and dependency pattern. For reproducibility, we used a fixed random seed to generate each program. All r_i values in the generated programs, where $0 \leq r_i \leq 1$ is randomly chosen, are rounded to two decimal places.

All of these programs follow the same structural pattern: they define a sequence of random variables $X_1, \dots, X_n$, where each X_i is drawn from a uniform distribution whose parameters depend on the value of a single predecessor variable through a conditional branch of the form:

$$X_i = \begin{cases} \text{Unif}(a_i, b_i), & X_{\pi(i)} < r_i, \\ \text{Unif}(a'_i, b'_i), & \text{otherwise,} \end{cases}$$

for some parent selection function $\pi : \{2, \dots, n\} \rightarrow \{1, \dots, i-1\}$. Thus each program is a Bayesian network of conditionally independent nodes, differing only in the choice of $\pi(i)$ that determines the dependency pattern.

The code snippets show representative size 5 instances from each benchmark family. The scaling experiments generate larger instances of the same form by varying the program size and, for the alternating benchmarks, the guard span. The programs in Figure 29a follow the scheme depicted above where the program size is n , i.e., the number of comparisons performed is n . In Figure 29b, each x_i does not necessarily depend on x_{i-1} but on some randomly chosen predecessor³. In Figures 29c and 29d, we additionally choose at random whether the guards of the conditionals and the branches are either some predecessor variable x_i or a **uniform**-sampling instruction. The programs in Figure 30a also follow the scheme depicted above but where the variables used in the guards cycle with a fixed span size of m . For instance, for $m = 2$, the pattern is $x_1, x_2, x_1, \dots$. The programs in Figures 30b to 30d follow a similar pattern.

We report runtime (seconds) for (i) SPPL, (ii) SLICE (discretization only), (iii) SLICE+DICE (discretization + inference via DICE), and (iv) SLICE+ROULETTE (discretization + inference via ROULETTE). The timeout is 120 seconds.

Results. Discretization remains inexpensive across these benchmark families, and the resulting SLICE+DICE/ROULETTE pipelines scale well with increasing program size. In contrast, exact inference with SPPL exhibits substantially steeper runtime growth and reaches the timeout on several benchmark variants.

G.2 Decision Trees

Tables 5 and 6 report the decision tree benchmarks for varying tree sizes n . The first table contains the original decision tree benchmarks from [1], while the second extends this evaluation with additional, larger instances that we generated to more systematically study scaling as the tree size increases.

Results. We observe that SLICE+DICE achieves the lowest total runtime in practically all *DT* programs. SLICE+ROULETTE is competitive only on very small trees, and becomes substantially slower by DT_{100} – DT_{500} and times out from DT_{800} onward. Additionally, the discretization cost grows substantially by DT_{1000} – DT_{5000} . PSI completes the three DT_4 variants and the independent DT_{14}

³We generate these programs in a randomized manner with a fixed seed for reproducibility

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.56 then uniform(0,2) else uniform(0,3) in
let x3 = if x2 < 0.92 then uniform(0,4) else uniform(0,5) in
let x4 = if x3 < 0.47 then uniform(0,6) else uniform(0,7) in
let x5 = if x4 < 0.51 then uniform(0,8) else uniform(0,9) in
x5 < 0.5

```

(a) Conditional Independent (Figure 31a): Size 5 chain in which each variable depends on its immediate predecessor; all uniform distributions are distinct.

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.56 then uniform(0,2) else uniform(0,3) in
let x3 = if x1 < 0.45 then uniform(0,4) else uniform(0,5) in
let x4 = if x1 < 0.18 then uniform(0,6) else uniform(0,7) in
let x5 = if x1 < 0.48 then uniform(0,8) else uniform(0,9) in
x5 < 0.5

```

(b) Conditional Independent Fork (Figure 31b): Size 5 fork in which x_1 is the sole parent of all subsequent variables; all uniform distributions are distinct.

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.18 then uniform(0,2) else uniform(0,3) in
let x3 = if x1 < 0.63 then uniform(0,4) else uniform(0,5) in
let x4 = if x2 < 0.09 then uniform(0,6) else uniform(0,7) in
let x5 = if x1 < 0.14 then uniform(0,8) else uniform(0,9) in
x5 < 0.5

```

(c) Conditional Random 1 (Figure 31c): Size 5 program with randomly selected predecessors. Some variables may be unused; all uniform distributions are distinct.

```

let x1 = uniform(0, 10) in
let x2 = if x1 < 0.06 then x1 else x1 in
let x3 = if uniform(0, 4) < 0.64 then x2 else x1 in
let x4 = if x3 < 0.71 then uniform(0, 5) else x1 in
let x5 = if uniform(0, 4) < 0.4 then x1 else uniform(0, 2) in
x4 < 0.5

```

(d) Conditional Random 2 (Figure 31d): Size 5 program with randomized guards and branches. Uniform upper bounds are random integers from 0 to 10 and may repeat.

Fig. 29. Representative programs from the scaling benchmarks.

variant, but times out on the remaining programs. Hakaru's disintegration phase remains fast, but its subsequent simplification is less robust: it produces errors on several smaller independent variants, approaches the timeout at DT_{500} , and times out on all variants from DT_{800} onward.

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.02 then uniform(0,2) else uniform(0,3) in
let x3 = if x2 < 0.68 then uniform(0,4) else uniform(0,5) in
let x4 = if x1 < 0.21 then uniform(0,6) else uniform(0,7) in
let x5 = if x2 < 0.93 then uniform(0,8) else uniform(0,9) in
x5 < 0.5

```

(a) Alternating Guard 1 (Figure 32a): Size 5 program with guard span 2. Guard variables cycle over the span, so variables beyond it may be unused.

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.38 then uniform(0,2) else x1 in
let x3 = if x2 < 0.71 then uniform(0,4) else x2 in
let x4 = if x1 < 0.42 then uniform(0,6) else x3 in
let x5 = if x2 < 0.57 then uniform(0,8) else x4 in
x5 < 0.5

```

(b) Alternating Guard 2 (Figure 32b): Size 5 program with guard span 2. Every variable is later used as a guard or else-branch expression; all uniform distributions are distinct.

```

let x1 = uniform(0,1) in
let x2 = if x1 < 0.67 then uniform(0,2) else uniform(0,3) in
let x3 = if x2 < 0.5 then x2 else uniform(0,4) in
let x4 = if x1 < 0.5 then x3 else uniform(0,6) in
let x5 = if x2 < 0.5 then x4 else uniform(0,8) in
x5 < 0.5

```

(c) Alternating Guard 3 (Figure 32c): Size 5 program with guard span 2. Every variable is later used as a guard or then-branch expression; all uniform distributions are distinct.

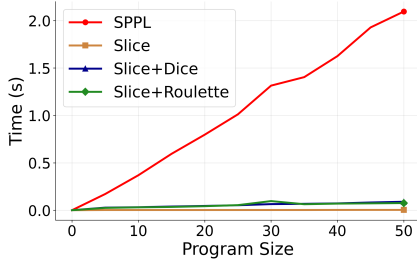
```

let x1 = uniform(0,2) in
let x2 = if x1 < 0.13 then uniform(0,2) else uniform(0,1) in
let x3 = if x2 < 0.06 then uniform(0,8) else uniform(0,3) in
let x4 = if x1 < 0.68 then uniform(0,4) else uniform(0,8) in
let x5 = if x2 < 0.51 then uniform(0,3) else uniform(0,7) in
x5 < 0.5

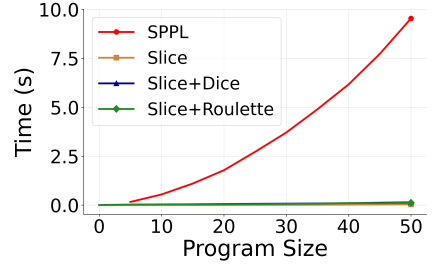
```

(d) Random Alternating (Figure 32d): Size 5 program with guard span 2. Uniform upper bounds are random integers from 0 to 10 and may repeat.

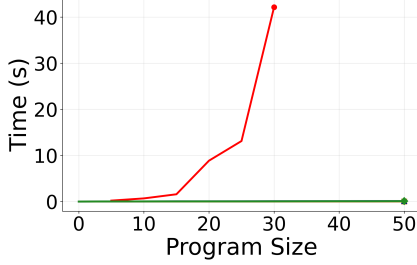
Fig. 30. Representative programs from the scaling benchmarks.



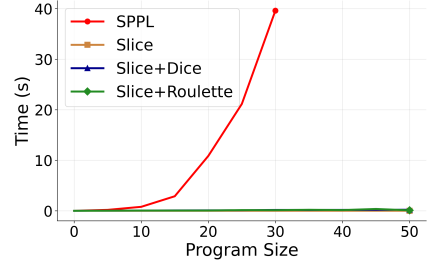
(a) Conditional Independent



(b) Conditional Independent – Fork

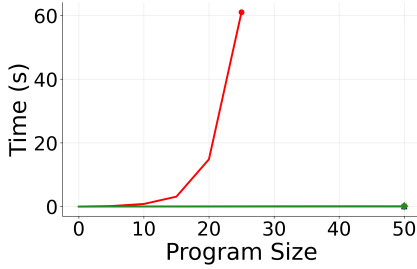


(c) Conditional Independent – Rand. 1

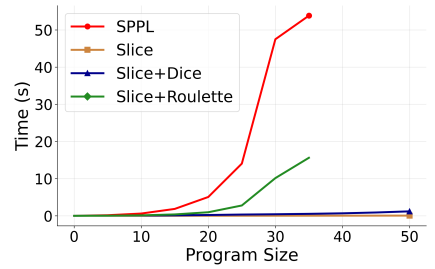


(d) Conditional Independent – Rand. 2

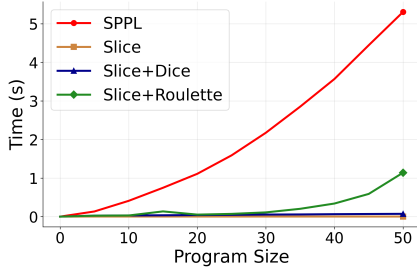
Fig. 31. Scaling graphs for conditionally independent programs. Time-out=120s; a disappearing line indicates timeout. Graphs (a)–(d) correspond, respectively, to the representative programs in Figures 29a to 29d.



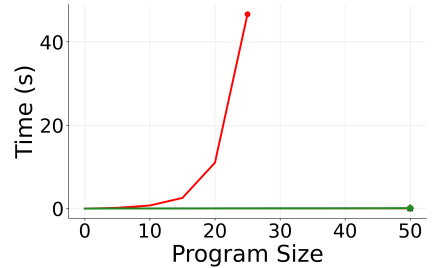
(a) Alt. Guard 1



(b) Alt. Guard 2



(c) Alt. Guard 3



(d) Alt. Guard – Rand.

Fig. 32. Scaling graphs for conditionally independent programs with alternating guards. Time-out=120s; a disappearing line indicates timeout. Graphs (a)–(d) correspond, respectively, to the representative programs in Figures 30a to 30d.

Table 5. Execution times (seconds) for the benchmarks from [1], with a timeout of 120s. The err. 's from Hakaru's simplification step represent a crash, caused by an internal symbolic processing exception.

Decision Program	Population Model	Slice+Dice (s)			Slice+Roulette (s)			SPPL (s)			PSI (s)	Hakaru (s)		
		Slice	Dice	Total	Slice	Roulette	Total	Trans.	Judg.	Total		Disint.	Simplify	Total
DT ₄	Ind	0.002	0.031	0.033	0.002	0.026	0.028	0.074	0.011	0.085	3.477	0.104	1.806	1.910
	BN1	0.002	0.033	0.035	0.002	0.031	0.033	0.245	0.028	0.273	19.890	0.031	0.426	0.457
	BN2	0.002	0.035	0.037	0.003	0.034	0.037	0.268	0.030	0.298	51.107	0.030	0.528	0.558
DT ₁₄	Ind	0.002	0.047	0.049	0.002	0.049	0.051	0.153	0.024	0.177	26.150	0.038	err.	err.
	BN1	0.002	0.056	0.058	0.002	0.074	0.076	0.597	0.076	0.673	t.o.	0.029	1.067	1.096
	BN2	0.002	0.058	0.060	0.002	0.073	0.075	0.535	0.072	0.607	t.o.	0.028	2.619	2.647
DT ₁₆	Ind	0.002	0.045	0.047	0.002	0.054	0.056	0.176	0.026	0.202	t.o.	0.031	err.	err.
	BN1	0.002	0.053	0.055	0.002	0.079	0.081	0.535	0.088	0.623	t.o.	0.036	1.125	1.161
	BN2	0.003	0.053	0.056	0.003	0.079	0.082	0.554	0.090	0.644	t.o.	0.035	1.594	1.629
DT ₁₆ ^z	Ind	0.002	0.034	0.036	0.002	0.052	0.054	0.201	0.029	0.230	t.o.	0.031	err.	err.
	BN1	0.002	0.042	0.044	0.003	0.191	0.194	0.597	0.080	0.677	t.o.	0.036	err.	err.
	BN2	0.003	0.049	0.052	0.003	0.224	0.227	0.642	0.075	0.717	t.o.	0.040	2.142	2.182
DT ₄₄	Ind	0.002	0.099	0.101	0.002	0.276	0.278	0.402	0.073	0.475	t.o.	0.041	t.o.	t.o.
	BN1	0.003	0.120	0.123	0.003	0.550	0.553	1.043	0.309	1.352	t.o.	0.042	15.972	16.014
	BN2	0.003	0.122	0.125	0.003	0.546	0.549	1.135	0.346	1.481	t.o.	0.053	44.456	44.509

Table 6. Execution times (seconds) for the benchmarks from [1], with a timeout of 120s. For DT₅₀₀₀ under our setup, SPPL returned a maximum recursion depth exceeded error (reported as err.).

Decision Program	Population Model	Slice+Dice (s)			Slice+Roulette (s)			SPPL (s)			PSI (s)	Hakaru (s)		
		Slice	Dice	Total	Slice	Roulette	Total	Trans.	Judg.	Total		Disint.	Simplify	Total
DT ₅₀	Ind	0.006	0.094	0.100	0.004	0.156	0.160	0.765	0.029	0.793	t.o.	0.045	3.712	3.757
	BN1	0.007	0.079	0.086	0.006	0.451	0.457	1.646	0.085	1.731	t.o.	0.039	3.548	3.587
	BN2	0.007	0.083	0.090	0.006	0.403	0.409	1.580	0.080	1.660	t.o.	0.037	3.635	3.672
DT ₁₀₀	Ind	0.011	0.138	0.149	0.011	0.654	0.665	1.455	0.036	1.490	t.o.	0.057	8.490	8.547
	BN1	0.015	0.148	0.163	0.016	1.775	1.791	2.867	0.150	3.017	t.o.	0.077	8.596	8.673
	BN2	0.017	0.153	0.170	0.016	1.775	1.791	2.914	0.152	3.066	t.o.	0.069	8.816	8.885
DT ₂₀₀	Ind	0.045	0.289	0.334	0.045	4.089	4.134	2.710	0.059	2.770	t.o.	0.099	24.551	24.650
	BN1	0.064	0.322	0.386	0.064	9.496	9.560	5.836	0.318	6.154	t.o.	0.111	23.741	23.852
	BN2	0.068	0.305	0.373	0.068	9.556	9.624	5.797	0.302	6.099	t.o.	0.115	23.988	24.103
DT ₅₀₀	Ind	0.479	0.879	1.358	0.515	55.677	56.192	7.908	0.141	8.049	t.o.	0.295	118.438	118.733
	BN1	0.599	0.961	1.560	0.636	113.426	114.062	15.017	0.787	15.804	t.o.	0.293	117.507	117.800
	BN2	0.616	0.999	1.615	0.610	112.976	113.586	15.330	0.889	16.219	t.o.	0.301	113.667	113.968
DT ₈₀₀	Ind	1.738	1.592	3.330	1.738	t.o.	t.o.	14.698	0.238	14.937	t.o.	0.488	t.o.	t.o.
	BN1	2.184	1.690	3.874	2.184	t.o.	t.o.	26.297	1.265	27.561	t.o.	0.597	t.o.	t.o.
	BN2	2.151	1.704	3.855	2.151	t.o.	t.o.	26.542	1.298	27.840	t.o.	0.512	t.o.	t.o.
DT ₁₀₀₀	Ind	3.584	2.096	5.680	3.584	t.o.	t.o.	17.626	0.279	17.905	t.o.	0.774	t.o.	t.o.
	BN1	4.125	2.192	6.317	4.125	t.o.	t.o.	33.417	1.537	34.954	t.o.	0.784	t.o.	t.o.
	BN2	4.052	2.152	6.204	4.052	t.o.	t.o.	34.225	1.612	35.837	t.o.	0.803	t.o.	t.o.
DT ₂₀₀₀	Ind	26.791	4.607	31.398	26.791	t.o.	t.o.	44.957	0.786	45.742	t.o.	2.868	t.o.	t.o.
	BN1	29.148	4.726	33.874	29.148	t.o.	t.o.	84.744	3.227	87.972	t.o.	2.775	t.o.	t.o.
	BN2	30.353	4.715	35.068	30.353	t.o.	t.o.	76.398	3.152	79.550	t.o.	2.944	t.o.	t.o.
DT ₅₀₀₀	Ind	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	err.	err.	err.	t.o.	16.030	t.o.	t.o.
	BN1	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	err.	err.	err.	t.o.	14.422	t.o.	t.o.
	BN2	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	err.	err.	err.	t.o.	14.933	t.o.	t.o.

G.3 Baselines

We finally evaluate SLICE on benchmarks drawn from prior work [21, 40]. Most benchmark programs are taken directly from the SPPL Github repository and translated into SLICE. In some cases, the SPPL versions differ from the original PSI benchmarks, for example by replacing continuous choices with finite ones so that the programs fall within the class supported by SPPL. CoinBias and SurveyUnbias are not included in the SPPL repository, so we adapted these benchmarks in the same style. Consequently, the SLICE and SPPL columns in Table 7 evaluate the same benchmark programs and provide an apples-to-apples comparison. The full benchmark programs are provided in §G.3.1. The first column in the table gives the benchmark name, the second shows the number of datasets on which to condition the program, and the remaining columns display execution times in seconds. For benchmarks with both modified and unmodified versions, the PSI and HAKARU cells report the modified timing marked by • above the unmodified timing marked by ◦. The • entries therefore provide the apples-to-apples comparison with SLICE and SPPL. For Hakaru, we report disintegration, subsequent simplification, and their total. t.o. denotes the same 120s timeout used elsewhere, and err. denotes a tool error.

Table 7. Execution times (in seconds) for benchmarks from PSI [21] and SPPL [40], with modified and unmodified Psi and Hakaru exact inference baselines. Timeout=120s. For benchmarks with two outputs for Psi and Hakaru, • marks the modified program and ◦ marks the unmodified program. err. represents a crash, caused by out of memory.

Benchmark	Datasets	Slice+Dice (s)	Slice+Roulette (s)	SPPL (s)	PSI (s)	Hakaru (s)		
						Disintegrate	Simplify	Total
ClickGraph [21]	10	1.469	t.o.	t.o.	t.o. • t.o. ◦	0.046 • 0.035 ◦	t.o. • 11.287 ◦	t.o. • 11.322 ◦
ClinicalTrial [21]	20	1.582	5.020	17.895	t.o. • 1.916 ◦	t.o. • t.o. ◦	t.o. • t.o. ◦	t.o. • t.o. ◦
CoinBias [21]	5	0.466	0.590	2.382	20.138 • 0.369 ◦	0.058 • 0.025 ◦	4.789 • 0.528 ◦	4.847 • 0.553 ◦
Election [40]	–	1.587	t.o.	19.410	t.o.	0.019	t.o.	t.o.
MarkovSwitching [40]	–	0.423	0.240	19.998	t.o.	0.020	0.275	0.295
StudentInterviews [40]	–	3.331	t.o.	53.209	t.o.	0.025	0.443	0.468
SurveyUnbias [21]	5	75.221	0.651	3.519	t.o. • t.o. ◦	0.027 • 0.029 ◦	t.o. • err. ◦	t.o. • err. ◦
TrueSkill [21]	3	0.750	68.669	32.881	t.o. • 0.068 ◦	0.027 • 0.018 ◦	err. • 0.215 ◦	err. • 0.233 ◦

Results. Across the modified baseline programs in Table 7, the SLICE pipelines generally provide the fastest successful end-to-end inference: SLICE+DICE performs best on most benchmarks, while SLICE+ROULETTE is faster on some instances. SPPL is generally slower, and PSI frequently times out, although both PSI and Hakaru perform better on some unmodified programs. Hakaru’s disintegration phase is usually fast, but subsequent Maple simplification often times out or returns an error, limiting its end-to-end reliability.

G.3.1 Baseline Programs. The exact SLICE programs used for these baseline benchmarks are reproduced below.

ClickGraph

```

1  let p_similar = beta(1, 1) in
2
3  let similar_0 =
4    if p_similar <= 0.05 then discrete(0.95, 0.05)
5    else if p_similar <= 0.1 then discrete(0.9, 0.1)
6    else if p_similar <= 0.15 then discrete(0.85, 0.15)
7    else if p_similar <= 0.2 then discrete(0.8, 0.2)
8    else if p_similar <= 0.25 then discrete(0.75, 0.25)
9    else if p_similar <= 0.3 then discrete(0.7, 0.3)
10   else if p_similar <= 0.35 then discrete(0.65, 0.35)
11   else if p_similar <= 0.4 then discrete(0.6, 0.4)
12   else if p_similar <= 0.45 then discrete(0.55, 0.45)
13   else if p_similar <= 0.5 then discrete(0.5, 0.5)
14   else if p_similar <= 0.55 then discrete(0.45, 0.55)
15   else if p_similar <= 0.6 then discrete(0.4, 0.6)
16   else if p_similar <= 0.65 then discrete(0.35, 0.65)
17   else if p_similar <= 0.7 then discrete(0.3, 0.7)
18   else if p_similar <= 0.75 then discrete(0.25, 0.75)
19   else if p_similar <= 0.8 then discrete(0.2, 0.8)
20   else if p_similar <= 0.85 then discrete(0.15, 0.85)
21   else if p_similar <= 0.9 then discrete(0.1, 0.9)
22   else if p_similar <= 0.95 then discrete(0.05, 0.95)
23   else discrete(0.0, 1.0)
24 in
25
26 let p_click_url_A_0 =
27   if similar_0 ==#2 1#2 then uniform(0,1) else uniform(0, 1)
28 in
29
30 let p_click_url_B_0 =
31   if similar_0 ==#2 1#2 then p_click_url_A_0 else uniform(0, 1)
32 in
33
34 let click_url_A_0 =
35   if p_click_url_A_0 <= 0.1 then discrete(0.9, 0.1)
36   else if p_click_url_A_0 <= 0.2 then discrete(0.8, 0.2)
37   else if p_click_url_A_0 <= 0.3 then discrete(0.7, 0.3)
38   else if p_click_url_A_0 <= 0.4 then discrete(0.6, 0.4)
39   else if p_click_url_A_0 <= 0.5 then discrete(0.5, 0.5)
40   else if p_click_url_A_0 <= 0.6 then discrete(0.4, 0.6)
41   else if p_click_url_A_0 <= 0.7 then discrete(0.3, 0.7)
42   else if p_click_url_A_0 <= 0.8 then discrete(0.2, 0.8)
43   else if p_click_url_A_0 <= 0.9 then discrete(0.1, 0.9)
44   else discrete(0.0, 1.0)
45 in
46
47 let click_url_B_0 =
48   if p_click_url_B_0 <= 0.1 then discrete(0.9, 0.1)
49   else if p_click_url_B_0 <= 0.2 then discrete(0.8, 0.2)
50   else if p_click_url_B_0 <= 0.3 then discrete(0.7, 0.3)
51   else if p_click_url_B_0 <= 0.4 then discrete(0.6, 0.4)
52   else if p_click_url_B_0 <= 0.5 then discrete(0.5, 0.5)
53   else if p_click_url_B_0 <= 0.6 then discrete(0.4, 0.6)
54   else if p_click_url_B_0 <= 0.7 then discrete(0.3, 0.7)
55   else if p_click_url_B_0 <= 0.8 then discrete(0.2, 0.8)
56   else if p_click_url_B_0 <= 0.9 then discrete(0.1, 0.9)
57   else discrete(0.0, 1.0)
58 in
59
60 let similar_1 =
61   if p_similar <= 0.05 then discrete(0.95, 0.05)
62   else if p_similar <= 0.1 then discrete(0.9, 0.1)
63   else if p_similar <= 0.15 then discrete(0.85, 0.15)
64   else if p_similar <= 0.2 then discrete(0.8, 0.2)
65   else if p_similar <= 0.25 then discrete(0.75, 0.25)
66   else if p_similar <= 0.3 then discrete(0.7, 0.3)

```

```

67   else if p_similar <= 0.35 then discrete(0.65, 0.35)
68   else if p_similar <= 0.4 then discrete(0.6, 0.4)
69   else if p_similar <= 0.45 then discrete(0.55, 0.45)
70   else if p_similar <= 0.5 then discrete(0.5, 0.5)
71   else if p_similar <= 0.55 then discrete(0.45, 0.55)
72   else if p_similar <= 0.6 then discrete(0.4, 0.6)
73   else if p_similar <= 0.65 then discrete(0.35, 0.65)
74   else if p_similar <= 0.7 then discrete(0.3, 0.7)
75   else if p_similar <= 0.75 then discrete(0.25, 0.75)
76   else if p_similar <= 0.8 then discrete(0.2, 0.8)
77   else if p_similar <= 0.85 then discrete(0.15, 0.85)
78   else if p_similar <= 0.9 then discrete(0.1, 0.9)
79   else if p_similar <= 0.95 then discrete(0.05, 0.95)
80   else discrete(0.0, 1.0)
81 in
82
83 let p_click_url_A_1 =
84   if similar_1 ==#2 1#2 then uniform(0,1) else uniform(0, 1)
85 in
86
87 let p_click_url_B_1 =
88   if similar_1 ==#2 1#2 then p_click_url_A_1 else uniform(0, 1)
89 in
90
91 let click_url_A_1 =
92   if p_click_url_A_1 <= 0.1 then discrete(0.9, 0.1)
93   else if p_click_url_A_1 <= 0.2 then discrete(0.8, 0.2)
94   else if p_click_url_A_1 <= 0.3 then discrete(0.7, 0.3)
95   else if p_click_url_A_1 <= 0.4 then discrete(0.6, 0.4)
96   else if p_click_url_A_1 <= 0.5 then discrete(0.5, 0.5)
97   else if p_click_url_A_1 <= 0.6 then discrete(0.4, 0.6)
98   else if p_click_url_A_1 <= 0.7 then discrete(0.3, 0.7)
99   else if p_click_url_A_1 <= 0.8 then discrete(0.2, 0.8)
100  else if p_click_url_A_1 <= 0.9 then discrete(0.1, 0.9)
101  else discrete(0.0, 1.0)
102 in
103
104 let click_url_B_1 =
105   if p_click_url_B_1 <= 0.1 then discrete(0.9, 0.1)
106   else if p_click_url_B_1 <= 0.2 then discrete(0.8, 0.2)
107   else if p_click_url_B_1 <= 0.3 then discrete(0.7, 0.3)
108   else if p_click_url_B_1 <= 0.4 then discrete(0.6, 0.4)
109   else if p_click_url_B_1 <= 0.5 then discrete(0.5, 0.5)
110   else if p_click_url_B_1 <= 0.6 then discrete(0.4, 0.6)
111   else if p_click_url_B_1 <= 0.7 then discrete(0.3, 0.7)
112   else if p_click_url_B_1 <= 0.8 then discrete(0.2, 0.8)
113   else if p_click_url_B_1 <= 0.9 then discrete(0.1, 0.9)
114   else discrete(0.0, 1.0)
115 in
116
117 let similar_2 =
118   if p_similar <= 0.05 then discrete(0.95, 0.05)
119   else if p_similar <= 0.1 then discrete(0.9, 0.1)
120   else if p_similar <= 0.15 then discrete(0.85, 0.15)
121   else if p_similar <= 0.2 then discrete(0.8, 0.2)
122   else if p_similar <= 0.25 then discrete(0.75, 0.25)
123   else if p_similar <= 0.3 then discrete(0.7, 0.3)
124   else if p_similar <= 0.35 then discrete(0.65, 0.35)
125   else if p_similar <= 0.4 then discrete(0.6, 0.4)
126   else if p_similar <= 0.45 then discrete(0.55, 0.45)
127   else if p_similar <= 0.5 then discrete(0.5, 0.5)
128   else if p_similar <= 0.55 then discrete(0.45, 0.55)
129   else if p_similar <= 0.6 then discrete(0.4, 0.6)
130   else if p_similar <= 0.65 then discrete(0.35, 0.65)
131   else if p_similar <= 0.7 then discrete(0.3, 0.7)
132   else if p_similar <= 0.75 then discrete(0.25, 0.75)
133   else if p_similar <= 0.8 then discrete(0.2, 0.8)
134   else if p_similar <= 0.85 then discrete(0.15, 0.85)
135   else if p_similar <= 0.9 then discrete(0.1, 0.9)
136   else if p_similar <= 0.95 then discrete(0.05, 0.95)
137   else discrete(0.0, 1.0)

```

```

138 in
139
140 let p_click_url_A_2 =
141   if similar_2 ==#2 1#2 then uniform(0,1) else uniform(0, 1)
142 in
143
144 let p_click_url_B_2 =
145   if similar_2 ==#2 1#2 then p_click_url_A_2 else uniform(0, 1)
146 in
147
148 let click_url_A_2 =
149   if p_click_url_A_2 <= 0.1 then discrete(0.9, 0.1)
150   else if p_click_url_A_2 <= 0.2 then discrete(0.8, 0.2)
151   else if p_click_url_A_2 <= 0.3 then discrete(0.7, 0.3)
152   else if p_click_url_A_2 <= 0.4 then discrete(0.6, 0.4)
153   else if p_click_url_A_2 <= 0.5 then discrete(0.5, 0.5)
154   else if p_click_url_A_2 <= 0.6 then discrete(0.4, 0.6)
155   else if p_click_url_A_2 <= 0.7 then discrete(0.3, 0.7)
156   else if p_click_url_A_2 <= 0.8 then discrete(0.2, 0.8)
157   else if p_click_url_A_2 <= 0.9 then discrete(0.1, 0.9)
158   else discrete(0.0, 1.0)
159 in
160
161 let click_url_B_2 =
162   if p_click_url_B_2 <= 0.1 then discrete(0.9, 0.1)
163   else if p_click_url_B_2 <= 0.2 then discrete(0.8, 0.2)
164   else if p_click_url_B_2 <= 0.3 then discrete(0.7, 0.3)
165   else if p_click_url_B_2 <= 0.4 then discrete(0.6, 0.4)
166   else if p_click_url_B_2 <= 0.5 then discrete(0.5, 0.5)
167   else if p_click_url_B_2 <= 0.6 then discrete(0.4, 0.6)
168   else if p_click_url_B_2 <= 0.7 then discrete(0.3, 0.7)
169   else if p_click_url_B_2 <= 0.8 then discrete(0.2, 0.8)
170   else if p_click_url_B_2 <= 0.9 then discrete(0.1, 0.9)
171   else discrete(0.0, 1.0)
172 in
173
174 let similar_3 =
175   if p_similar <= 0.05 then discrete(0.95, 0.05)
176   else if p_similar <= 0.1 then discrete(0.9, 0.1)
177   else if p_similar <= 0.15 then discrete(0.85, 0.15)
178   else if p_similar <= 0.2 then discrete(0.8, 0.2)
179   else if p_similar <= 0.25 then discrete(0.75, 0.25)
180   else if p_similar <= 0.3 then discrete(0.7, 0.3)
181   else if p_similar <= 0.35 then discrete(0.65, 0.35)
182   else if p_similar <= 0.4 then discrete(0.6, 0.4)
183   else if p_similar <= 0.45 then discrete(0.55, 0.45)
184   else if p_similar <= 0.5 then discrete(0.5, 0.5)
185   else if p_similar <= 0.55 then discrete(0.45, 0.55)
186   else if p_similar <= 0.6 then discrete(0.4, 0.6)
187   else if p_similar <= 0.65 then discrete(0.35, 0.65)
188   else if p_similar <= 0.7 then discrete(0.3, 0.7)
189   else if p_similar <= 0.75 then discrete(0.25, 0.75)
190   else if p_similar <= 0.8 then discrete(0.2, 0.8)
191   else if p_similar <= 0.85 then discrete(0.15, 0.85)
192   else if p_similar <= 0.9 then discrete(0.1, 0.9)
193   else if p_similar <= 0.95 then discrete(0.05, 0.95)
194   else discrete(0.0, 1.0)
195 in
196
197 let p_click_url_A_3 =
198   if similar_3 ==#2 1#2 then uniform(0,1) else uniform(0, 1)
199 in
200
201 let p_click_url_B_3 =
202   if similar_3 ==#2 1#2 then p_click_url_A_3 else uniform(0, 1)
203 in
204
205 let click_url_A_3 =
206   if p_click_url_A_3 <= 0.1 then discrete(0.9, 0.1)
207   else if p_click_url_A_3 <= 0.2 then discrete(0.8, 0.2)
208   else if p_click_url_A_3 <= 0.3 then discrete(0.7, 0.3)

```

```

209     else if p_click_url_A.3 <= 0.4 then discrete(0.6, 0.4)
210     else if p_click_url_A.3 <= 0.5 then discrete(0.5, 0.5)
211     else if p_click_url_A.3 <= 0.6 then discrete(0.4, 0.6)
212     else if p_click_url_A.3 <= 0.7 then discrete(0.3, 0.7)
213     else if p_click_url_A.3 <= 0.8 then discrete(0.2, 0.8)
214     else if p_click_url_A.3 <= 0.9 then discrete(0.1, 0.9)
215     else discrete(0.0, 1.0)
216 in
217
218 let click_url_B.3 =
219   if p_click_url_B.3 <= 0.1 then discrete(0.9, 0.1)
220   else if p_click_url_B.3 <= 0.2 then discrete(0.8, 0.2)
221   else if p_click_url_B.3 <= 0.3 then discrete(0.7, 0.3)
222   else if p_click_url_B.3 <= 0.4 then discrete(0.6, 0.4)
223   else if p_click_url_B.3 <= 0.5 then discrete(0.5, 0.5)
224   else if p_click_url_B.3 <= 0.6 then discrete(0.4, 0.6)
225   else if p_click_url_B.3 <= 0.7 then discrete(0.3, 0.7)
226   else if p_click_url_B.3 <= 0.8 then discrete(0.2, 0.8)
227   else if p_click_url_B.3 <= 0.9 then discrete(0.1, 0.9)
228   else discrete(0.0, 1.0)
229 in
230
231 let similar_4 =
232   if p_similar <= 0.05 then discrete(0.95, 0.05)
233   else if p_similar <= 0.1 then discrete(0.9, 0.1)
234   else if p_similar <= 0.15 then discrete(0.85, 0.15)
235   else if p_similar <= 0.2 then discrete(0.8, 0.2)
236   else if p_similar <= 0.25 then discrete(0.75, 0.25)
237   else if p_similar <= 0.3 then discrete(0.7, 0.3)
238   else if p_similar <= 0.35 then discrete(0.65, 0.35)
239   else if p_similar <= 0.4 then discrete(0.6, 0.4)
240   else if p_similar <= 0.45 then discrete(0.55, 0.45)
241   else if p_similar <= 0.5 then discrete(0.5, 0.5)
242   else if p_similar <= 0.55 then discrete(0.45, 0.55)
243   else if p_similar <= 0.6 then discrete(0.4, 0.6)
244   else if p_similar <= 0.65 then discrete(0.35, 0.65)
245   else if p_similar <= 0.7 then discrete(0.3, 0.7)
246   else if p_similar <= 0.75 then discrete(0.25, 0.75)
247   else if p_similar <= 0.8 then discrete(0.2, 0.8)
248   else if p_similar <= 0.85 then discrete(0.15, 0.85)
249   else if p_similar <= 0.9 then discrete(0.1, 0.9)
250   else if p_similar <= 0.95 then discrete(0.05, 0.95)
251   else discrete(0.0, 1.0)
252 in
253
254 let p_click_url_A.4 =
255   if similar_4 ==#2 1#2 then uniform(0,1) else uniform(0, 1)
256 in
257
258 let p_click_url_B.4 =
259   if similar_4 ==#2 1#2 then p_click_url_A.4 else uniform(0, 1)
260 in
261
262 let click_url_A.4 =
263   if p_click_url_A.4 <= 0.1 then discrete(0.9, 0.1)
264   else if p_click_url_A.4 <= 0.2 then discrete(0.8, 0.2)
265   else if p_click_url_A.4 <= 0.3 then discrete(0.7, 0.3)
266   else if p_click_url_A.4 <= 0.4 then discrete(0.6, 0.4)
267   else if p_click_url_A.4 <= 0.5 then discrete(0.5, 0.5)
268   else if p_click_url_A.4 <= 0.6 then discrete(0.4, 0.6)
269   else if p_click_url_A.4 <= 0.7 then discrete(0.3, 0.7)
270   else if p_click_url_A.4 <= 0.8 then discrete(0.2, 0.8)
271   else if p_click_url_A.4 <= 0.9 then discrete(0.1, 0.9)
272   else discrete(0.0, 1.0)
273 in
274
275 let click_url_B.4 =
276   if p_click_url_B.4 <= 0.1 then discrete(0.9, 0.1)
277   else if p_click_url_B.4 <= 0.2 then discrete(0.8, 0.2)
278   else if p_click_url_B.4 <= 0.3 then discrete(0.7, 0.3)
279   else if p_click_url_B.4 <= 0.4 then discrete(0.6, 0.4)

```

```

280     else if p_click_url_B_4 <= 0.5 then discrete(0.5, 0.5)
281     else if p_click_url_B_4 <= 0.6 then discrete(0.4, 0.6)
282     else if p_click_url_B_4 <= 0.7 then discrete(0.3, 0.7)
283     else if p_click_url_B_4 <= 0.8 then discrete(0.2, 0.8)
284     else if p_click_url_B_4 <= 0.9 then discrete(0.1, 0.9)
285     else discrete(0.0, 1.0)
286   in
287
288   let _0 = observe(click_url_A_0 ==#2 1#2) in
289   let _1 = observe(click_url_B_0 ==#2 1#2) in
290   let _2 = observe(click_url_A_1 ==#2 1#2) in
291   let _3 = observe(click_url_B_1 ==#2 1#2) in
292   let _4 = observe(click_url_A_2 ==#2 0#2) in
293   let _5 = observe(click_url_B_2 ==#2 0#2) in
294   let _6 = observe(click_url_A_3 ==#2 1#2) in
295   let _7 = observe(click_url_B_3 ==#2 1#2) in
296   let _8 = observe(click_url_A_4 ==#2 0#2) in
297   let _9 = observe(click_url_B_4 ==#2 0#2) in
298
299   p_similar < 0.3

```

ClinicalTrial

```

1  let isEffective = discrete(0.5, 0.5) in
2  let probControl = discrete(0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
    0.05, 0.05, 0.05, 0.05, 0.05, 0.05) in
3  let probTreated = discrete(0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
    0.05, 0.05, 0.05, 0.05, 0.05, 0.05) in
4  let probAll = discrete(0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
    0.05, 0.05, 0.05, 0.05, 0.05) in
5
6  let controlGroup0 =
7    if isEffective ==#2 1#2 then
8      if probControl ==#20 0#20 then discrete(1.0, 0.0)
9      else if probControl ==#20 1#20 then discrete(0.95, 0.05)
10     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
11     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
12     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
13     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
14     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
15     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
16     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
17     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
18     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
19     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
20     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
21     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
22     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
23     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
24     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
25     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
26     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
27     else discrete(0.05, 0.95)
28   else
29     if probAll ==#20 0#20 then discrete(1.0, 0.0)
30     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
31     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
32     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
33     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
34     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
35     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
36     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
37     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
38     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
39     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
40     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
41     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
42     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
43     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
44     else if probAll ==#20 15#20 then discrete(0.25, 0.75)

```

```

45     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
46     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
47     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
48     else discrete(0.05, 0.95) in
49
50 let treatedGroup0 =
51   if isEffective ==#2 1#2 then
52     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
53     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
54     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
55     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
56     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
57     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
58     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
59     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
60     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
61     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
62     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
63     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
64     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
65     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
66     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
67     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
68     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
69     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
70     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
71     else discrete(0.05, 0.95)
72   else
73     if probAll ==#20 0#20 then discrete(1.0, 0.0)
74     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
75     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
76     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
77     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
78     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
79     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
80     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
81     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
82     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
83     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
84     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
85     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
86     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
87     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
88     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
89     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
90     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
91     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
92     else discrete(0.05, 0.95) in
93
94 let controlGroup1 =
95   if isEffective ==#2 1#2 then
96     if probControl ==#20 0#20 then discrete(1.0, 0.0)
97     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
98     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
99     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
100    else if probControl ==#20 4#20 then discrete(0.8, 0.2)
101    else if probControl ==#20 5#20 then discrete(0.75, 0.25)
102    else if probControl ==#20 6#20 then discrete(0.7, 0.3)
103    else if probControl ==#20 7#20 then discrete(0.65, 0.35)
104    else if probControl ==#20 8#20 then discrete(0.6, 0.4)
105    else if probControl ==#20 9#20 then discrete(0.55, 0.45)
106    else if probControl ==#20 10#20 then discrete(0.5, 0.5)
107    else if probControl ==#20 11#20 then discrete(0.45, 0.55)
108    else if probControl ==#20 12#20 then discrete(0.4, 0.6)
109    else if probControl ==#20 13#20 then discrete(0.35, 0.65)
110    else if probControl ==#20 14#20 then discrete(0.3, 0.7)
111    else if probControl ==#20 15#20 then discrete(0.25, 0.75)
112    else if probControl ==#20 16#20 then discrete(0.2, 0.8)
113    else if probControl ==#20 17#20 then discrete(0.15, 0.85)
114    else if probControl ==#20 18#20 then discrete(0.1, 0.9)
115    else discrete(0.05, 0.95)

```

```

116     else
117         if probAll ==#20 0#20 then discrete(1.0, 0.0)
118         else if probAll ==#20 1#20 then discrete(0.95, 0.05)
119         else if probAll ==#20 2#20 then discrete(0.9, 0.1)
120         else if probAll ==#20 3#20 then discrete(0.85, 0.15)
121         else if probAll ==#20 4#20 then discrete(0.8, 0.2)
122         else if probAll ==#20 5#20 then discrete(0.75, 0.25)
123         else if probAll ==#20 6#20 then discrete(0.7, 0.3)
124         else if probAll ==#20 7#20 then discrete(0.65, 0.35)
125         else if probAll ==#20 8#20 then discrete(0.6, 0.4)
126         else if probAll ==#20 9#20 then discrete(0.55, 0.45)
127         else if probAll ==#20 10#20 then discrete(0.5, 0.5)
128         else if probAll ==#20 11#20 then discrete(0.45, 0.55)
129         else if probAll ==#20 12#20 then discrete(0.4, 0.6)
130         else if probAll ==#20 13#20 then discrete(0.35, 0.65)
131         else if probAll ==#20 14#20 then discrete(0.3, 0.7)
132         else if probAll ==#20 15#20 then discrete(0.25, 0.75)
133         else if probAll ==#20 16#20 then discrete(0.2, 0.8)
134         else if probAll ==#20 17#20 then discrete(0.15, 0.85)
135         else if probAll ==#20 18#20 then discrete(0.1, 0.9)
136         else discrete(0.05, 0.95) in
137
138     let treatedGroup1 =
139         if isEffective ==#2 1#2 then
140             if probTreated ==#20 0#20 then discrete(1.0, 0.0)
141             else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
142             else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
143             else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
144             else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
145             else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
146             else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
147             else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
148             else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
149             else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
150             else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
151             else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
152             else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
153             else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
154             else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
155             else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
156             else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
157             else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
158             else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
159             else discrete(0.05, 0.95)
160         else
161             if probAll ==#20 0#20 then discrete(1.0, 0.0)
162             else if probAll ==#20 1#20 then discrete(0.95, 0.05)
163             else if probAll ==#20 2#20 then discrete(0.9, 0.1)
164             else if probAll ==#20 3#20 then discrete(0.85, 0.15)
165             else if probAll ==#20 4#20 then discrete(0.8, 0.2)
166             else if probAll ==#20 5#20 then discrete(0.75, 0.25)
167             else if probAll ==#20 6#20 then discrete(0.7, 0.3)
168             else if probAll ==#20 7#20 then discrete(0.65, 0.35)
169             else if probAll ==#20 8#20 then discrete(0.6, 0.4)
170             else if probAll ==#20 9#20 then discrete(0.55, 0.45)
171             else if probAll ==#20 10#20 then discrete(0.5, 0.5)
172             else if probAll ==#20 11#20 then discrete(0.45, 0.55)
173             else if probAll ==#20 12#20 then discrete(0.4, 0.6)
174             else if probAll ==#20 13#20 then discrete(0.35, 0.65)
175             else if probAll ==#20 14#20 then discrete(0.3, 0.7)
176             else if probAll ==#20 15#20 then discrete(0.25, 0.75)
177             else if probAll ==#20 16#20 then discrete(0.2, 0.8)
178             else if probAll ==#20 17#20 then discrete(0.15, 0.85)
179             else if probAll ==#20 18#20 then discrete(0.1, 0.9)
180             else discrete(0.05, 0.95) in
181
182     let controlGroup2 =
183         if isEffective ==#2 1#2 then
184             if probControl ==#20 0#20 then discrete(1.0, 0.0)
185             else if probControl ==#20 1#20 then discrete(0.95, 0.05)
186             else if probControl ==#20 2#20 then discrete(0.9, 0.1)

```

```

187     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
188     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
189     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
190     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
191     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
192     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
193     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
194     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
195     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
196     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
197     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
198     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
199     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
200     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
201     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
202     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
203     else discrete(0.05, 0.95)
204   else
205     if probAll ==#20 0#20 then discrete(1.0, 0.0)
206     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
207     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
208     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
209     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
210     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
211     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
212     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
213     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
214     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
215     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
216     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
217     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
218     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
219     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
220     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
221     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
222     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
223     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
224     else discrete(0.05, 0.95) in
225
226   let treatedGroup2 =
227     if isEffective ==#2 1#2 then
228       if probTreated ==#20 0#20 then discrete(1.0, 0.0)
229       else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
230       else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
231       else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
232       else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
233       else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
234       else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
235       else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
236       else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
237       else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
238       else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
239       else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
240       else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
241       else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
242       else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
243       else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
244       else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
245       else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
246       else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
247       else discrete(0.05, 0.95)
248     else
249       if probAll ==#20 0#20 then discrete(1.0, 0.0)
250       else if probAll ==#20 1#20 then discrete(0.95, 0.05)
251       else if probAll ==#20 2#20 then discrete(0.9, 0.1)
252       else if probAll ==#20 3#20 then discrete(0.85, 0.15)
253       else if probAll ==#20 4#20 then discrete(0.8, 0.2)
254       else if probAll ==#20 5#20 then discrete(0.75, 0.25)
255       else if probAll ==#20 6#20 then discrete(0.7, 0.3)
256       else if probAll ==#20 7#20 then discrete(0.65, 0.35)
257       else if probAll ==#20 8#20 then discrete(0.6, 0.4)

```

```

258     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
259     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
260     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
261     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
262     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
263     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
264     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
265     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
266     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
267     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
268     else discrete(0.05, 0.95) in
269
270 let controlGroup3 =
271   if isEffective ==#2 1#2 then
272     if probControl ==#20 0#20 then discrete(1.0, 0.0)
273     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
274     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
275     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
276     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
277     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
278     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
279     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
280     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
281     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
282     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
283     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
284     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
285     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
286     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
287     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
288     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
289     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
290     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
291     else discrete(0.05, 0.95)
292   else
293     if probAll ==#20 0#20 then discrete(1.0, 0.0)
294     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
295     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
296     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
297     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
298     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
299     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
300     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
301     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
302     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
303     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
304     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
305     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
306     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
307     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
308     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
309     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
310     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
311     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
312     else discrete(0.05, 0.95) in
313
314 let treatedGroup3 =
315   if isEffective ==#2 1#2 then
316     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
317     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
318     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
319     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
320     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
321     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
322     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
323     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
324     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
325     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
326     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
327     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
328     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)

```

```

329     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
330     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
331     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
332     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
333     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
334     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
335     else discrete(0.05, 0.95)
336   else
337     if probAll ==#20 0#20 then discrete(1.0, 0.0)
338     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
339     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
340     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
341     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
342     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
343     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
344     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
345     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
346     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
347     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
348     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
349     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
350     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
351     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
352     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
353     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
354     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
355     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
356     else discrete(0.05, 0.95) in
357
358   let controlGroup4 =
359     if isEffective ==#2 1#2 then
360       if probControl ==#20 0#20 then discrete(1.0, 0.0)
361       else if probControl ==#20 1#20 then discrete(0.95, 0.05)
362       else if probControl ==#20 2#20 then discrete(0.9, 0.1)
363       else if probControl ==#20 3#20 then discrete(0.85, 0.15)
364       else if probControl ==#20 4#20 then discrete(0.8, 0.2)
365       else if probControl ==#20 5#20 then discrete(0.75, 0.25)
366       else if probControl ==#20 6#20 then discrete(0.7, 0.3)
367       else if probControl ==#20 7#20 then discrete(0.65, 0.35)
368       else if probControl ==#20 8#20 then discrete(0.6, 0.4)
369       else if probControl ==#20 9#20 then discrete(0.55, 0.45)
370       else if probControl ==#20 10#20 then discrete(0.5, 0.5)
371       else if probControl ==#20 11#20 then discrete(0.45, 0.55)
372       else if probControl ==#20 12#20 then discrete(0.4, 0.6)
373       else if probControl ==#20 13#20 then discrete(0.35, 0.65)
374       else if probControl ==#20 14#20 then discrete(0.3, 0.7)
375       else if probControl ==#20 15#20 then discrete(0.25, 0.75)
376       else if probControl ==#20 16#20 then discrete(0.2, 0.8)
377       else if probControl ==#20 17#20 then discrete(0.15, 0.85)
378       else if probControl ==#20 18#20 then discrete(0.1, 0.9)
379       else discrete(0.05, 0.95)
380     else
381       if probAll ==#20 0#20 then discrete(1.0, 0.0)
382       else if probAll ==#20 1#20 then discrete(0.95, 0.05)
383       else if probAll ==#20 2#20 then discrete(0.9, 0.1)
384       else if probAll ==#20 3#20 then discrete(0.85, 0.15)
385       else if probAll ==#20 4#20 then discrete(0.8, 0.2)
386       else if probAll ==#20 5#20 then discrete(0.75, 0.25)
387       else if probAll ==#20 6#20 then discrete(0.7, 0.3)
388       else if probAll ==#20 7#20 then discrete(0.65, 0.35)
389       else if probAll ==#20 8#20 then discrete(0.6, 0.4)
390       else if probAll ==#20 9#20 then discrete(0.55, 0.45)
391       else if probAll ==#20 10#20 then discrete(0.5, 0.5)
392       else if probAll ==#20 11#20 then discrete(0.45, 0.55)
393       else if probAll ==#20 12#20 then discrete(0.4, 0.6)
394       else if probAll ==#20 13#20 then discrete(0.35, 0.65)
395       else if probAll ==#20 14#20 then discrete(0.3, 0.7)
396       else if probAll ==#20 15#20 then discrete(0.25, 0.75)
397       else if probAll ==#20 16#20 then discrete(0.2, 0.8)
398       else if probAll ==#20 17#20 then discrete(0.15, 0.85)
399       else if probAll ==#20 18#20 then discrete(0.1, 0.9)

```

```

400     else discrete(0.05, 0.95) in
401
402 let treatedGroup4 =
403   if isEffective ==#2 1#2 then
404     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
405     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
406     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
407     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
408     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
409     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
410     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
411     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
412     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
413     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
414     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
415     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
416     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
417     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
418     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
419     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
420     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
421     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
422     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
423     else discrete(0.05, 0.95)
424   else
425     if probAll ==#20 0#20 then discrete(1.0, 0.0)
426     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
427     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
428     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
429     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
430     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
431     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
432     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
433     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
434     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
435     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
436     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
437     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
438     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
439     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
440     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
441     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
442     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
443     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
444     else discrete(0.05, 0.95) in
445
446 let controlGroup5 =
447   if isEffective ==#2 1#2 then
448     if probControl ==#20 0#20 then discrete(1.0, 0.0)
449     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
450     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
451     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
452     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
453     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
454     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
455     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
456     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
457     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
458     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
459     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
460     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
461     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
462     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
463     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
464     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
465     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
466     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
467     else discrete(0.05, 0.95)
468   else
469     if probAll ==#20 0#20 then discrete(1.0, 0.0)
470     else if probAll ==#20 1#20 then discrete(0.95, 0.05)

```

```

471     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
472     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
473     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
474     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
475     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
476     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
477     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
478     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
479     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
480     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
481     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
482     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
483     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
484     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
485     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
486     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
487     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
488     else discrete(0.05, 0.95) in
489
490 let treatedGroup5 =
491   if isEffective ==#2 1#2 then
492     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
493     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
494     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
495     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
496     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
497     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
498     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
499     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
500     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
501     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
502     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
503     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
504     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
505     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
506     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
507     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
508     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
509     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
510     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
511     else discrete(0.05, 0.95)
512   else
513     if probAll ==#20 0#20 then discrete(1.0, 0.0)
514     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
515     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
516     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
517     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
518     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
519     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
520     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
521     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
522     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
523     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
524     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
525     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
526     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
527     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
528     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
529     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
530     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
531     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
532     else discrete(0.05, 0.95) in
533
534 let controlGroup6 =
535   if isEffective ==#2 1#2 then
536     if probControl ==#20 0#20 then discrete(1.0, 0.0)
537     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
538     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
539     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
540     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
541     else if probControl ==#20 5#20 then discrete(0.75, 0.25)

```

```

542     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
543     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
544     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
545     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
546     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
547     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
548     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
549     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
550     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
551     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
552     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
553     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
554     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
555     else discrete(0.05, 0.95)
556   else
557     if probAll ==#20 0#20 then discrete(1.0, 0.0)
558     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
559     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
560     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
561     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
562     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
563     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
564     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
565     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
566     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
567     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
568     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
569     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
570     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
571     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
572     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
573     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
574     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
575     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
576     else discrete(0.05, 0.95) in
577
578   let treatedGroup6 =
579     if isEffective ==#2 1#2 then
580       if probTreated ==#20 0#20 then discrete(1.0, 0.0)
581       else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
582       else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
583       else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
584       else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
585       else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
586       else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
587       else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
588       else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
589       else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
590       else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
591       else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
592       else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
593       else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
594       else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
595       else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
596       else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
597       else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
598       else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
599       else discrete(0.05, 0.95)
600     else
601       if probAll ==#20 0#20 then discrete(1.0, 0.0)
602       else if probAll ==#20 1#20 then discrete(0.95, 0.05)
603       else if probAll ==#20 2#20 then discrete(0.9, 0.1)
604       else if probAll ==#20 3#20 then discrete(0.85, 0.15)
605       else if probAll ==#20 4#20 then discrete(0.8, 0.2)
606       else if probAll ==#20 5#20 then discrete(0.75, 0.25)
607       else if probAll ==#20 6#20 then discrete(0.7, 0.3)
608       else if probAll ==#20 7#20 then discrete(0.65, 0.35)
609       else if probAll ==#20 8#20 then discrete(0.6, 0.4)
610       else if probAll ==#20 9#20 then discrete(0.55, 0.45)
611       else if probAll ==#20 10#20 then discrete(0.5, 0.5)
612       else if probAll ==#20 11#20 then discrete(0.45, 0.55)

```

```

613     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
614     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
615     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
616     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
617     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
618     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
619     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
620     else discrete(0.05, 0.95) in
621
622 let controlGroup7 =
623   if isEffective ==#2 1#2 then
624     if probControl ==#20 0#20 then discrete(1.0, 0.0)
625     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
626     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
627     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
628     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
629     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
630     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
631     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
632     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
633     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
634     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
635     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
636     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
637     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
638     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
639     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
640     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
641     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
642     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
643     else discrete(0.05, 0.95)
644   else
645     if probAll ==#20 0#20 then discrete(1.0, 0.0)
646     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
647     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
648     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
649     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
650     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
651     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
652     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
653     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
654     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
655     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
656     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
657     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
658     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
659     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
660     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
661     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
662     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
663     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
664     else discrete(0.05, 0.95) in
665
666 let treatedGroup7 =
667   if isEffective ==#2 1#2 then
668     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
669     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
670     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
671     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
672     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
673     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
674     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
675     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
676     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
677     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
678     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
679     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
680     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
681     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
682     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
683     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)

```

```

684     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
685     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
686     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
687     else discrete(0.05, 0.95)
688   else
689     if probAll ==#20 0#20 then discrete(1.0, 0.0)
690     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
691     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
692     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
693     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
694     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
695     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
696     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
697     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
698     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
699     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
700     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
701     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
702     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
703     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
704     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
705     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
706     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
707     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
708     else discrete(0.05, 0.95) in
709
710   let controlGroup8 =
711     if isEffective ==#2 1#2 then
712       if probControl ==#20 0#20 then discrete(1.0, 0.0)
713       else if probControl ==#20 1#20 then discrete(0.95, 0.05)
714       else if probControl ==#20 2#20 then discrete(0.9, 0.1)
715       else if probControl ==#20 3#20 then discrete(0.85, 0.15)
716       else if probControl ==#20 4#20 then discrete(0.8, 0.2)
717       else if probControl ==#20 5#20 then discrete(0.75, 0.25)
718       else if probControl ==#20 6#20 then discrete(0.7, 0.3)
719       else if probControl ==#20 7#20 then discrete(0.65, 0.35)
720       else if probControl ==#20 8#20 then discrete(0.6, 0.4)
721       else if probControl ==#20 9#20 then discrete(0.55, 0.45)
722       else if probControl ==#20 10#20 then discrete(0.5, 0.5)
723       else if probControl ==#20 11#20 then discrete(0.45, 0.55)
724       else if probControl ==#20 12#20 then discrete(0.4, 0.6)
725       else if probControl ==#20 13#20 then discrete(0.35, 0.65)
726       else if probControl ==#20 14#20 then discrete(0.3, 0.7)
727       else if probControl ==#20 15#20 then discrete(0.25, 0.75)
728       else if probControl ==#20 16#20 then discrete(0.2, 0.8)
729       else if probControl ==#20 17#20 then discrete(0.15, 0.85)
730       else if probControl ==#20 18#20 then discrete(0.1, 0.9)
731       else discrete(0.05, 0.95)
732     else
733       if probAll ==#20 0#20 then discrete(1.0, 0.0)
734       else if probAll ==#20 1#20 then discrete(0.95, 0.05)
735       else if probAll ==#20 2#20 then discrete(0.9, 0.1)
736       else if probAll ==#20 3#20 then discrete(0.85, 0.15)
737       else if probAll ==#20 4#20 then discrete(0.8, 0.2)
738       else if probAll ==#20 5#20 then discrete(0.75, 0.25)
739       else if probAll ==#20 6#20 then discrete(0.7, 0.3)
740       else if probAll ==#20 7#20 then discrete(0.65, 0.35)
741       else if probAll ==#20 8#20 then discrete(0.6, 0.4)
742       else if probAll ==#20 9#20 then discrete(0.55, 0.45)
743       else if probAll ==#20 10#20 then discrete(0.5, 0.5)
744       else if probAll ==#20 11#20 then discrete(0.45, 0.55)
745       else if probAll ==#20 12#20 then discrete(0.4, 0.6)
746       else if probAll ==#20 13#20 then discrete(0.35, 0.65)
747       else if probAll ==#20 14#20 then discrete(0.3, 0.7)
748       else if probAll ==#20 15#20 then discrete(0.25, 0.75)
749       else if probAll ==#20 16#20 then discrete(0.2, 0.8)
750       else if probAll ==#20 17#20 then discrete(0.15, 0.85)
751       else if probAll ==#20 18#20 then discrete(0.1, 0.9)
752       else discrete(0.05, 0.95) in
753
754   let treatedGroup8 =

```

```

755 if isEffective ==#2 1#2 then
756   if probTreated ==#20 0#20 then discrete(1.0, 0.0)
757   else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
758   else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
759   else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
760   else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
761   else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
762   else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
763   else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
764   else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
765   else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
766   else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
767   else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
768   else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
769   else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
770   else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
771   else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
772   else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
773   else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
774   else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
775   else discrete(0.05, 0.95)
776 else
777   if probAll ==#20 0#20 then discrete(1.0, 0.0)
778   else if probAll ==#20 1#20 then discrete(0.95, 0.05)
779   else if probAll ==#20 2#20 then discrete(0.9, 0.1)
780   else if probAll ==#20 3#20 then discrete(0.85, 0.15)
781   else if probAll ==#20 4#20 then discrete(0.8, 0.2)
782   else if probAll ==#20 5#20 then discrete(0.75, 0.25)
783   else if probAll ==#20 6#20 then discrete(0.7, 0.3)
784   else if probAll ==#20 7#20 then discrete(0.65, 0.35)
785   else if probAll ==#20 8#20 then discrete(0.6, 0.4)
786   else if probAll ==#20 9#20 then discrete(0.55, 0.45)
787   else if probAll ==#20 10#20 then discrete(0.5, 0.5)
788   else if probAll ==#20 11#20 then discrete(0.45, 0.55)
789   else if probAll ==#20 12#20 then discrete(0.4, 0.6)
790   else if probAll ==#20 13#20 then discrete(0.35, 0.65)
791   else if probAll ==#20 14#20 then discrete(0.3, 0.7)
792   else if probAll ==#20 15#20 then discrete(0.25, 0.75)
793   else if probAll ==#20 16#20 then discrete(0.2, 0.8)
794   else if probAll ==#20 17#20 then discrete(0.15, 0.85)
795   else if probAll ==#20 18#20 then discrete(0.1, 0.9)
796   else discrete(0.05, 0.95) in
797
798 let controlGroup9 =
799   if isEffective ==#2 1#2 then
800     if probControl ==#20 0#20 then discrete(1.0, 0.0)
801     else if probControl ==#20 1#20 then discrete(0.95, 0.05)
802     else if probControl ==#20 2#20 then discrete(0.9, 0.1)
803     else if probControl ==#20 3#20 then discrete(0.85, 0.15)
804     else if probControl ==#20 4#20 then discrete(0.8, 0.2)
805     else if probControl ==#20 5#20 then discrete(0.75, 0.25)
806     else if probControl ==#20 6#20 then discrete(0.7, 0.3)
807     else if probControl ==#20 7#20 then discrete(0.65, 0.35)
808     else if probControl ==#20 8#20 then discrete(0.6, 0.4)
809     else if probControl ==#20 9#20 then discrete(0.55, 0.45)
810     else if probControl ==#20 10#20 then discrete(0.5, 0.5)
811     else if probControl ==#20 11#20 then discrete(0.45, 0.55)
812     else if probControl ==#20 12#20 then discrete(0.4, 0.6)
813     else if probControl ==#20 13#20 then discrete(0.35, 0.65)
814     else if probControl ==#20 14#20 then discrete(0.3, 0.7)
815     else if probControl ==#20 15#20 then discrete(0.25, 0.75)
816     else if probControl ==#20 16#20 then discrete(0.2, 0.8)
817     else if probControl ==#20 17#20 then discrete(0.15, 0.85)
818     else if probControl ==#20 18#20 then discrete(0.1, 0.9)
819     else discrete(0.05, 0.95)
820   else
821     if probAll ==#20 0#20 then discrete(1.0, 0.0)
822     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
823     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
824     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
825     else if probAll ==#20 4#20 then discrete(0.8, 0.2)

```

```

826     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
827     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
828     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
829     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
830     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
831     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
832     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
833     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
834     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
835     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
836     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
837     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
838     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
839     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
840     else discrete(0.05, 0.95) in
841
842 let treatedGroup9 =
843   if isEffective ==#2 1#2 then
844     if probTreated ==#20 0#20 then discrete(1.0, 0.0)
845     else if probTreated ==#20 1#20 then discrete(0.95, 0.05)
846     else if probTreated ==#20 2#20 then discrete(0.9, 0.1)
847     else if probTreated ==#20 3#20 then discrete(0.85, 0.15)
848     else if probTreated ==#20 4#20 then discrete(0.8, 0.2)
849     else if probTreated ==#20 5#20 then discrete(0.75, 0.25)
850     else if probTreated ==#20 6#20 then discrete(0.7, 0.3)
851     else if probTreated ==#20 7#20 then discrete(0.65, 0.35)
852     else if probTreated ==#20 8#20 then discrete(0.6, 0.4)
853     else if probTreated ==#20 9#20 then discrete(0.55, 0.45)
854     else if probTreated ==#20 10#20 then discrete(0.5, 0.5)
855     else if probTreated ==#20 11#20 then discrete(0.45, 0.55)
856     else if probTreated ==#20 12#20 then discrete(0.4, 0.6)
857     else if probTreated ==#20 13#20 then discrete(0.35, 0.65)
858     else if probTreated ==#20 14#20 then discrete(0.3, 0.7)
859     else if probTreated ==#20 15#20 then discrete(0.25, 0.75)
860     else if probTreated ==#20 16#20 then discrete(0.2, 0.8)
861     else if probTreated ==#20 17#20 then discrete(0.15, 0.85)
862     else if probTreated ==#20 18#20 then discrete(0.1, 0.9)
863     else discrete(0.05, 0.95)
864   else
865     if probAll ==#20 0#20 then discrete(1.0, 0.0)
866     else if probAll ==#20 1#20 then discrete(0.95, 0.05)
867     else if probAll ==#20 2#20 then discrete(0.9, 0.1)
868     else if probAll ==#20 3#20 then discrete(0.85, 0.15)
869     else if probAll ==#20 4#20 then discrete(0.8, 0.2)
870     else if probAll ==#20 5#20 then discrete(0.75, 0.25)
871     else if probAll ==#20 6#20 then discrete(0.7, 0.3)
872     else if probAll ==#20 7#20 then discrete(0.65, 0.35)
873     else if probAll ==#20 8#20 then discrete(0.6, 0.4)
874     else if probAll ==#20 9#20 then discrete(0.55, 0.45)
875     else if probAll ==#20 10#20 then discrete(0.5, 0.5)
876     else if probAll ==#20 11#20 then discrete(0.45, 0.55)
877     else if probAll ==#20 12#20 then discrete(0.4, 0.6)
878     else if probAll ==#20 13#20 then discrete(0.35, 0.65)
879     else if probAll ==#20 14#20 then discrete(0.3, 0.7)
880     else if probAll ==#20 15#20 then discrete(0.25, 0.75)
881     else if probAll ==#20 16#20 then discrete(0.2, 0.8)
882     else if probAll ==#20 17#20 then discrete(0.15, 0.85)
883     else if probAll ==#20 18#20 then discrete(0.1, 0.9)
884     else discrete(0.05, 0.95) in
885
886 let _0 = observe(controlGroup0 ==#2 0#2) in
887 let _1 = observe(treatedGroup0 ==#2 0#2) in
888 let _2 = observe(controlGroup1 ==#2 0#2) in
889 let _3 = observe(treatedGroup1 ==#2 1#2) in
890 let _4 = observe(controlGroup2 ==#2 1#2) in
891 let _5 = observe(treatedGroup2 ==#2 0#2) in
892 let _6 = observe(controlGroup3 ==#2 1#2) in
893 let _7 = observe(treatedGroup3 ==#2 0#2) in
894 let _8 = observe(controlGroup4 ==#2 1#2) in
895 let _9 = observe(treatedGroup4 ==#2 1#2) in
896 let _10 = observe(controlGroup5 ==#2 0#2) in

```

```

897 let _11 = observe(treatedGroup5 ==#2 1#2) in
898 let _12 = observe(controlGroup6 ==#2 1#2) in
899 let _13 = observe(treatedGroup6 ==#2 1#2) in
900 let _14 = observe(controlGroup7 ==#2 1#2) in
901 let _15 = observe(treatedGroup7 ==#2 1#2) in
902 let _16 = observe(controlGroup8 ==#2 0#2) in
903 let _17 = observe(treatedGroup8 ==#2 1#2) in
904 let _18 = observe(controlGroup9 ==#2 0#2) in
905 let _19 = observe(treatedGroup9 ==#2 1#2) in
906
907 isEffective ==#2 1#2

```

CoinBias

```

1 let bias = beta(2, 5) in
2
3 let tossResults_0 =
4   if bias <= 0.05 then discrete(0.95, 0.05)
5   else if bias <= 0.1 then discrete(0.9, 0.1)
6   else if bias <= 0.15 then discrete(0.85, 0.15)
7   else if bias <= 0.2 then discrete(0.8, 0.2)
8   else if bias <= 0.25 then discrete(0.75, 0.25)
9   else if bias <= 0.3 then discrete(0.7, 0.3)
10  else if bias <= 0.35 then discrete(0.65, 0.35)
11  else if bias <= 0.4 then discrete(0.6, 0.4)
12  else if bias <= 0.45 then discrete(0.55, 0.45)
13  else if bias <= 0.5 then discrete(0.5, 0.5)
14  else if bias <= 0.55 then discrete(0.45, 0.55)
15  else if bias <= 0.6 then discrete(0.4, 0.6)
16  else if bias <= 0.65 then discrete(0.35, 0.65)
17  else if bias <= 0.7 then discrete(0.3, 0.7)
18  else if bias <= 0.75 then discrete(0.25, 0.75)
19  else if bias <= 0.8 then discrete(0.2, 0.8)
20  else if bias <= 0.85 then discrete(0.15, 0.85)
21  else if bias <= 0.9 then discrete(0.1, 0.9)
22  else if bias <= 0.95 then discrete(0.05, 0.95)
23  else discrete(0.0, 1.0)
24 in
25
26 let tossResults_1 =
27   if bias <= 0.05 then discrete(0.95, 0.05)
28   else if bias <= 0.1 then discrete(0.9, 0.1)
29   else if bias <= 0.15 then discrete(0.85, 0.15)
30   else if bias <= 0.2 then discrete(0.8, 0.2)
31   else if bias <= 0.25 then discrete(0.75, 0.25)
32   else if bias <= 0.3 then discrete(0.7, 0.3)
33   else if bias <= 0.35 then discrete(0.65, 0.35)
34   else if bias <= 0.4 then discrete(0.6, 0.4)
35   else if bias <= 0.45 then discrete(0.55, 0.45)
36   else if bias <= 0.5 then discrete(0.5, 0.5)
37   else if bias <= 0.55 then discrete(0.45, 0.55)
38   else if bias <= 0.6 then discrete(0.4, 0.6)
39   else if bias <= 0.65 then discrete(0.35, 0.65)
40   else if bias <= 0.7 then discrete(0.3, 0.7)
41   else if bias <= 0.75 then discrete(0.25, 0.75)
42   else if bias <= 0.8 then discrete(0.2, 0.8)
43   else if bias <= 0.85 then discrete(0.15, 0.85)
44   else if bias <= 0.9 then discrete(0.1, 0.9)
45   else if bias <= 0.95 then discrete(0.05, 0.95)
46   else discrete(0.0, 1.0)
47 in
48
49 let tossResults_2 =
50   if bias <= 0.05 then discrete(0.95, 0.05)
51   else if bias <= 0.1 then discrete(0.9, 0.1)
52   else if bias <= 0.15 then discrete(0.85, 0.15)
53   else if bias <= 0.2 then discrete(0.8, 0.2)
54   else if bias <= 0.25 then discrete(0.75, 0.25)
55   else if bias <= 0.3 then discrete(0.7, 0.3)
56   else if bias <= 0.35 then discrete(0.65, 0.35)

```

```

57   else if bias <= 0.4 then discrete(0.6, 0.4)
58   else if bias <= 0.45 then discrete(0.55, 0.45)
59   else if bias <= 0.5 then discrete(0.5, 0.5)
60   else if bias <= 0.55 then discrete(0.45, 0.55)
61   else if bias <= 0.6 then discrete(0.4, 0.6)
62   else if bias <= 0.65 then discrete(0.35, 0.65)
63   else if bias <= 0.7 then discrete(0.3, 0.7)
64   else if bias <= 0.75 then discrete(0.25, 0.75)
65   else if bias <= 0.8 then discrete(0.2, 0.8)
66   else if bias <= 0.85 then discrete(0.15, 0.85)
67   else if bias <= 0.9 then discrete(0.1, 0.9)
68   else if bias <= 0.95 then discrete(0.05, 0.95)
69   else discrete(0.0, 1.0)
70 in
71
72 let tossResults_3 =
73   if bias <= 0.05 then discrete(0.95, 0.05)
74   else if bias <= 0.1 then discrete(0.9, 0.1)
75   else if bias <= 0.15 then discrete(0.85, 0.15)
76   else if bias <= 0.2 then discrete(0.8, 0.2)
77   else if bias <= 0.25 then discrete(0.75, 0.25)
78   else if bias <= 0.3 then discrete(0.7, 0.3)
79   else if bias <= 0.35 then discrete(0.65, 0.35)
80   else if bias <= 0.4 then discrete(0.6, 0.4)
81   else if bias <= 0.45 then discrete(0.55, 0.45)
82   else if bias <= 0.5 then discrete(0.5, 0.5)
83   else if bias <= 0.55 then discrete(0.45, 0.55)
84   else if bias <= 0.6 then discrete(0.4, 0.6)
85   else if bias <= 0.65 then discrete(0.35, 0.65)
86   else if bias <= 0.7 then discrete(0.3, 0.7)
87   else if bias <= 0.75 then discrete(0.25, 0.75)
88   else if bias <= 0.8 then discrete(0.2, 0.8)
89   else if bias <= 0.85 then discrete(0.15, 0.85)
90   else if bias <= 0.9 then discrete(0.1, 0.9)
91   else if bias <= 0.95 then discrete(0.05, 0.95)
92   else discrete(0.0, 1.0)
93 in
94
95 let tossResults_4 =
96   if bias <= 0.05 then discrete(0.95, 0.05)
97   else if bias <= 0.1 then discrete(0.9, 0.1)
98   else if bias <= 0.15 then discrete(0.85, 0.15)
99   else if bias <= 0.2 then discrete(0.8, 0.2)
100  else if bias <= 0.25 then discrete(0.75, 0.25)
101  else if bias <= 0.3 then discrete(0.7, 0.3)
102  else if bias <= 0.35 then discrete(0.65, 0.35)
103  else if bias <= 0.4 then discrete(0.6, 0.4)
104  else if bias <= 0.45 then discrete(0.55, 0.45)
105  else if bias <= 0.5 then discrete(0.5, 0.5)
106  else if bias <= 0.55 then discrete(0.45, 0.55)
107  else if bias <= 0.6 then discrete(0.4, 0.6)
108  else if bias <= 0.65 then discrete(0.35, 0.65)
109  else if bias <= 0.7 then discrete(0.3, 0.7)
110  else if bias <= 0.75 then discrete(0.25, 0.75)
111  else if bias <= 0.8 then discrete(0.2, 0.8)
112  else if bias <= 0.85 then discrete(0.15, 0.85)
113  else if bias <= 0.9 then discrete(0.1, 0.9)
114  else if bias <= 0.95 then discrete(0.05, 0.95)
115  else discrete(0.0, 1.0)
116 in
117
118 let _0 = observe(tossResults_0 ==#2 1#2) in
119 let _1 = observe(tossResults_1 ==#2 1#2) in
120 let _2 = observe(tossResults_2 ==#2 0#2) in
121 let _3 = observe(tossResults_3 ==#2 1#2) in
122 let _4 = observe(tossResults_4 ==#2 0#2) in
123
124 bias < 0.3

```

Election

```
1  let n = 4000 in
2  let param = discrete(
3    0.01: 250,
4    0.01: 251,
5    0.01: 252,
6    0.01: 253,
7    0.01: 254,
8    0.01: 255,
9    0.01: 256,
10   0.01: 257,
11   0.01: 258,
12   0.01: 259,
13   0.01: 260,
14   0.01: 261,
15   0.01: 262,
16   0.01: 263,
17   0.01: 264,
18   0.01: 265,
19   0.01: 266,
20   0.01: 267,
21   0.01: 268,
22   0.01: 269,
23   0.01: 270,
24   0.01: 271,
25   0.01: 272,
26   0.01: 273,
27   0.01: 274,
28   0.01: 275,
29   0.01: 276,
30   0.01: 277,
31   0.01: 278,
32   0.01: 279,
33   0.01: 280,
34   0.01: 281,
35   0.01: 282,
36   0.01: 283,
37   0.01: 284,
38   0.01: 285,
39   0.01: 286,
40   0.01: 287,
41   0.01: 288,
42   0.01: 289,
43   0.01: 290,
44   0.01: 291,
45   0.01: 292,
46   0.01: 293,
47   0.01: 294,
48   0.01: 295,
49   0.01: 296,
50   0.01: 297,
51   0.01: 298,
52   0.01: 299,
53   0.01: 300,
54   0.01: 301,
55   0.01: 302,
56   0.01: 303,
57   0.01: 304,
58   0.01: 305,
59   0.01: 306,
60   0.01: 307,
61   0.01: 308,
62   0.01: 309,
63   0.01: 310,
64   0.01: 311,
65   0.01: 312,
66   0.01: 313,
67   0.01: 314,
68   0.01: 315,
69   0.01: 316,
70   0.01: 317,
71   0.01: 318,
```

```

72  0.01: 319,
73  0.01: 320,
74  0.01: 321,
75  0.01: 322,
76  0.01: 323,
77  0.01: 324,
78  0.01: 325,
79  0.01: 326,
80  0.01: 327,
81  0.01: 328,
82  0.01: 329,
83  0.01: 330,
84  0.01: 331,
85  0.01: 332,
86  0.01: 333,
87  0.01: 334,
88  0.01: 335,
89  0.01: 336,
90  0.01: 337,
91  0.01: 338,
92  0.01: 339,
93  0.01: 340,
94  0.01: 341,
95  0.01: 342,
96  0.01: 343,
97  0.01: 344,
98  0.01: 345,
99  0.01: 346,
100 0.01: 347,
101 0.01: 348,
102 0.01: 349
103 ) in
104
105 let p =
106   if param <= 250 then beta(277, 250)
107   else if param <= 251 then beta(277, 251)
108   else if param <= 252 then beta(277, 252)
109   else if param <= 253 then beta(277, 253)
110   else if param <= 254 then beta(277, 254)
111   else if param <= 255 then beta(277, 255)
112   else if param <= 256 then beta(277, 256)
113   else if param <= 257 then beta(277, 257)
114   else if param <= 258 then beta(277, 258)
115   else if param <= 259 then beta(277, 259)
116   else if param <= 260 then beta(277, 260)
117   else if param <= 261 then beta(277, 261)
118   else if param <= 262 then beta(277, 262)
119   else if param <= 263 then beta(277, 263)
120   else if param <= 264 then beta(277, 264)
121   else if param <= 265 then beta(277, 265)
122   else if param <= 266 then beta(277, 266)
123   else if param <= 267 then beta(277, 267)
124   else if param <= 268 then beta(277, 268)
125   else if param <= 269 then beta(277, 269)
126   else if param <= 270 then beta(277, 270)
127   else if param <= 271 then beta(277, 271)
128   else if param <= 272 then beta(277, 272)
129   else if param <= 273 then beta(277, 273)
130   else if param <= 274 then beta(277, 274)
131   else if param <= 275 then beta(277, 275)
132   else if param <= 276 then beta(277, 276)
133   else if param <= 277 then beta(277, 277)
134   else if param <= 278 then beta(277, 278)
135   else if param <= 279 then beta(277, 279)
136   else if param <= 280 then beta(277, 280)
137   else if param <= 281 then beta(277, 281)
138   else if param <= 282 then beta(277, 282)
139   else if param <= 283 then beta(277, 283)
140   else if param <= 284 then beta(277, 284)
141   else if param <= 285 then beta(277, 285)
142   else if param <= 286 then beta(277, 286)

```

```

143  else if param <= 287 then beta(277, 287)
144  else if param <= 288 then beta(277, 288)
145  else if param <= 289 then beta(277, 289)
146  else if param <= 290 then beta(277, 290)
147  else if param <= 291 then beta(277, 291)
148  else if param <= 292 then beta(277, 292)
149  else if param <= 293 then beta(277, 293)
150  else if param <= 294 then beta(277, 294)
151  else if param <= 295 then beta(277, 295)
152  else if param <= 296 then beta(277, 296)
153  else if param <= 297 then beta(277, 297)
154  else if param <= 298 then beta(277, 298)
155  else if param <= 299 then beta(277, 299)
156  else if param <= 300 then beta(277, 300)
157  else if param <= 301 then beta(277, 301)
158  else if param <= 302 then beta(277, 302)
159  else if param <= 303 then beta(277, 303)
160  else if param <= 304 then beta(277, 304)
161  else if param <= 305 then beta(277, 305)
162  else if param <= 306 then beta(277, 306)
163  else if param <= 307 then beta(277, 307)
164  else if param <= 308 then beta(277, 308)
165  else if param <= 309 then beta(277, 309)
166  else if param <= 310 then beta(277, 310)
167  else if param <= 311 then beta(277, 311)
168  else if param <= 312 then beta(277, 312)
169  else if param <= 313 then beta(277, 313)
170  else if param <= 314 then beta(277, 314)
171  else if param <= 315 then beta(277, 315)
172  else if param <= 316 then beta(277, 316)
173  else if param <= 317 then beta(277, 317)
174  else if param <= 318 then beta(277, 318)
175  else if param <= 319 then beta(277, 319)
176  else if param <= 320 then beta(277, 320)
177  else if param <= 321 then beta(277, 321)
178  else if param <= 322 then beta(277, 322)
179  else if param <= 323 then beta(277, 323)
180  else if param <= 324 then beta(277, 324)
181  else if param <= 325 then beta(277, 325)
182  else if param <= 326 then beta(277, 326)
183  else if param <= 327 then beta(277, 327)
184  else if param <= 328 then beta(277, 328)
185  else if param <= 329 then beta(277, 329)
186  else if param <= 330 then beta(277, 330)
187  else if param <= 331 then beta(277, 331)
188  else if param <= 332 then beta(277, 332)
189  else if param <= 333 then beta(277, 333)
190  else if param <= 334 then beta(277, 334)
191  else if param <= 335 then beta(277, 335)
192  else if param <= 336 then beta(277, 336)
193  else if param <= 337 then beta(277, 337)
194  else if param <= 338 then beta(277, 338)
195  else if param <= 339 then beta(277, 339)
196  else if param <= 340 then beta(277, 340)
197  else if param <= 341 then beta(277, 341)
198  else if param <= 342 then beta(277, 342)
199  else if param <= 343 then beta(277, 343)
200  else if param <= 344 then beta(277, 344)
201  else if param <= 345 then beta(277, 345)
202  else if param <= 346 then beta(277, 346)
203  else if param <= 347 then beta(277, 347)
204  else if param <= 348 then beta(277, 348)
205  else beta(277, 349)
206  in
207
208  let votes =
209    if p <= 0.05 then binomial(0.025, n)
210    else if p <= 0.1 then binomial(0.075, n)
211    else if p <= 0.15 then binomial(0.125, n)
212    else if p <= 0.2 then binomial(0.175, n)
213    else if p <= 0.25 then binomial(0.225, n)

```

```

214   else if p <= 0.3 then binomial(0.275, n)
215   else if p <= 0.35 then binomial(0.325, n)
216   else if p <= 0.4 then binomial(0.375, n)
217   else if p <= 0.45 then binomial(0.425, n)
218   else if p <= 0.5 then binomial(0.475, n)
219   else if p <= 0.55 then binomial(0.525, n)
220   else if p <= 0.6 then binomial(0.575, n)
221   else if p <= 0.65 then binomial(0.625, n)
222   else if p <= 0.7 then binomial(0.675, n)
223   else if p <= 0.75 then binomial(0.725, n)
224   else if p <= 0.8 then binomial(0.775, n)
225   else if p <= 0.85 then binomial(0.825, n)
226   else if p <= 0.9 then binomial(0.875, n)
227   else if p <= 0.95 then binomial(0.925, n)
228   else binomial(0.975, n)
229 in
230
231 let win = votes > 2000 in
232 let _win = observe(win) in
233 param <= 250

```

MarkovSwitching (HMM)

```

1  let seperated = discrete(0.6, 0.4) in
2
3  let Z0 =
4    if seperated ==#2 0#2 then
5      discrete(0.5, 0.5)
6    else
7      discrete(0.5, 0.5)
8  in
9
10 let X0 =
11   if seperated ==#2 0#2 then
12     (if Z0 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
13   else
14     (if Z0 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
15 in
16
17 let Y0 =
18   if seperated ==#2 0#2 then
19     (if Z0 ==#2 0#2 then poisson(5) else poisson(8))
20   else
21     (if Z0 ==#2 0#2 then poisson(3) else poisson(8))
22 in
23
24 let Z1 =
25   if Z0 ==#2 0#2 then
26     discrete(0.8, 0.2)
27   else
28     discrete(0.2, 0.8)
29 in
30
31 let X1 =
32   if seperated ==#2 0#2 then
33     (if Z1 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
34   else
35     (if Z1 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
36 in
37
38 let Y1 =
39   if seperated ==#2 0#2 then
40     (if Z1 ==#2 0#2 then poisson(5) else poisson(8))
41   else
42     (if Z1 ==#2 0#2 then poisson(3) else poisson(8))
43 in
44
45 let Z2 =
46   if Z1 ==#2 0#2 then
47     discrete(0.8, 0.2)

```

```

48   else
49     discrete(0.2, 0.8)
50 in
51
52 let X2 =
53   if seperated ==#2 0#2 then
54     (if Z2 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
55   else
56     (if Z2 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
57 in
58
59 let Y2 =
60   if seperated ==#2 0#2 then
61     (if Z2 ==#2 0#2 then poisson(5) else poisson(8))
62   else
63     (if Z2 ==#2 0#2 then poisson(3) else poisson(8))
64 in
65
66 let Z3 =
67   if Z2 ==#2 0#2 then
68     discrete(0.8, 0.2)
69   else
70     discrete(0.2, 0.8)
71 in
72
73 let X3 =
74   if seperated ==#2 0#2 then
75     (if Z3 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
76   else
77     (if Z3 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
78 in
79
80 let Y3 =
81   if seperated ==#2 0#2 then
82     (if Z3 ==#2 0#2 then poisson(5) else poisson(8))
83   else
84     (if Z3 ==#2 0#2 then poisson(3) else poisson(8))
85 in
86
87 let Z4 =
88   if Z3 ==#2 0#2 then
89     discrete(0.8, 0.2)
90   else
91     discrete(0.2, 0.8)
92 in
93
94 let X4 =
95   if seperated ==#2 0#2 then
96     (if Z4 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
97   else
98     (if Z4 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
99 in
100
101 let Y4 =
102   if seperated ==#2 0#2 then
103     (if Z4 ==#2 0#2 then poisson(5) else poisson(8))
104   else
105     (if Z4 ==#2 0#2 then poisson(3) else poisson(8))
106 in
107
108 let Z5 =
109   if Z4 ==#2 0#2 then
110     discrete(0.8, 0.2)
111   else
112     discrete(0.2, 0.8)
113 in
114
115 let X5 =
116   if seperated ==#2 0#2 then
117     (if Z5 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
118   else

```

```

119     (if Z5 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
120 in
121
122 let Y5 =
123   if seperated ==#2 0#2 then
124     (if Z5 ==#2 0#2 then poisson(5) else poisson(8))
125   else
126     (if Z5 ==#2 0#2 then poisson(3) else poisson(8))
127 in
128
129 let Z6 =
130   if Z5 ==#2 0#2 then
131     discrete(0.8, 0.2)
132   else
133     discrete(0.2, 0.8)
134 in
135
136 let X6 =
137   if seperated ==#2 0#2 then
138     (if Z6 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
139   else
140     (if Z6 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
141 in
142
143 let Y6 =
144   if seperated ==#2 0#2 then
145     (if Z6 ==#2 0#2 then poisson(5) else poisson(8))
146   else
147     (if Z6 ==#2 0#2 then poisson(3) else poisson(8))
148 in
149
150 let Z7 =
151   if Z6 ==#2 0#2 then
152     discrete(0.8, 0.2)
153   else
154     discrete(0.2, 0.8)
155 in
156
157 let X7 =
158   if seperated ==#2 0#2 then
159     (if Z7 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
160   else
161     (if Z7 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
162 in
163
164 let Y7 =
165   if seperated ==#2 0#2 then
166     (if Z7 ==#2 0#2 then poisson(5) else poisson(8))
167   else
168     (if Z7 ==#2 0#2 then poisson(3) else poisson(8))
169 in
170
171 let Z8 =
172   if Z7 ==#2 0#2 then
173     discrete(0.8, 0.2)
174   else
175     discrete(0.2, 0.8)
176 in
177
178 let X8 =
179   if seperated ==#2 0#2 then
180     (if Z8 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
181   else
182     (if Z8 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
183 in
184
185 let Y8 =
186   if seperated ==#2 0#2 then
187     (if Z8 ==#2 0#2 then poisson(5) else poisson(8))
188   else
189     (if Z8 ==#2 0#2 then poisson(3) else poisson(8))

```

```

190 in
191
192 let Z9 =
193   if Z8 ==#2 0#2 then
194     discrete(0.8, 0.2)
195   else
196     discrete(0.2, 0.8)
197 in
198
199 let X9 =
200   if seperated ==#2 0#2 then
201     (if Z9 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
202   else
203     (if Z9 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
204 in
205
206 let Y9 =
207   if seperated ==#2 0#2 then
208     (if Z9 ==#2 0#2 then poisson(5) else poisson(8))
209   else
210     (if Z9 ==#2 0#2 then poisson(3) else poisson(8))
211 in
212
213 let Z10 =
214   if Z9 ==#2 0#2 then
215     discrete(0.8, 0.2)
216   else
217     discrete(0.2, 0.8)
218 in
219
220 let X10 =
221   if seperated ==#2 0#2 then
222     (if Z10 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
223   else
224     (if Z10 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
225 in
226
227 let Y10 =
228   if seperated ==#2 0#2 then
229     (if Z10 ==#2 0#2 then poisson(5) else poisson(8))
230   else
231     (if Z10 ==#2 0#2 then poisson(3) else poisson(8))
232 in
233
234 let Z11 =
235   if Z10 ==#2 0#2 then
236     discrete(0.8, 0.2)
237   else
238     discrete(0.2, 0.8)
239 in
240
241 let X11 =
242   if seperated ==#2 0#2 then
243     (if Z11 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
244   else
245     (if Z11 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
246 in
247
248 let Y11 =
249   if seperated ==#2 0#2 then
250     (if Z11 ==#2 0#2 then poisson(5) else poisson(8))
251   else
252     (if Z11 ==#2 0#2 then poisson(3) else poisson(8))
253 in
254
255 let Z12 =
256   if Z11 ==#2 0#2 then
257     discrete(0.8, 0.2)
258   else
259     discrete(0.2, 0.8)
260 in

```

```

261
262 let X12 =
263   if seperated ==#2 0#2 then
264     (if Z12 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
265   else
266     (if Z12 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
267 in
268
269 let Y12 =
270   if seperated ==#2 0#2 then
271     (if Z12 ==#2 0#2 then poisson(5) else poisson(8))
272   else
273     (if Z12 ==#2 0#2 then poisson(3) else poisson(8))
274 in
275
276 let Z13 =
277   if Z12 ==#2 0#2 then
278     discrete(0.8, 0.2)
279   else
280     discrete(0.2, 0.8)
281 in
282
283 let X13 =
284   if seperated ==#2 0#2 then
285     (if Z13 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
286   else
287     (if Z13 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
288 in
289
290 let Y13 =
291   if seperated ==#2 0#2 then
292     (if Z13 ==#2 0#2 then poisson(5) else poisson(8))
293   else
294     (if Z13 ==#2 0#2 then poisson(3) else poisson(8))
295 in
296
297 let Z14 =
298   if Z13 ==#2 0#2 then
299     discrete(0.8, 0.2)
300   else
301     discrete(0.2, 0.8)
302 in
303
304 let X14 =
305   if seperated ==#2 0#2 then
306     (if Z14 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
307   else
308     (if Z14 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
309 in
310
311 let Y14 =
312   if seperated ==#2 0#2 then
313     (if Z14 ==#2 0#2 then poisson(5) else poisson(8))
314   else
315     (if Z14 ==#2 0#2 then poisson(3) else poisson(8))
316 in
317
318 let Z15 =
319   if Z14 ==#2 0#2 then
320     discrete(0.8, 0.2)
321   else
322     discrete(0.2, 0.8)
323 in
324
325 let X15 =
326   if seperated ==#2 0#2 then
327     (if Z15 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
328   else
329     (if Z15 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
330 in
331

```

```

332 let Y15 =
333   if seperated ==#2 0#2 then
334     (if Z15 ==#2 0#2 then poisson(5) else poisson(8))
335   else
336     (if Z15 ==#2 0#2 then poisson(3) else poisson(8))
337 in
338
339 let Z16 =
340   if Z15 ==#2 0#2 then
341     discrete(0.8, 0.2)
342   else
343     discrete(0.2, 0.8)
344 in
345
346 let X16 =
347   if seperated ==#2 0#2 then
348     (if Z16 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
349   else
350     (if Z16 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
351 in
352
353 let Y16 =
354   if seperated ==#2 0#2 then
355     (if Z16 ==#2 0#2 then poisson(5) else poisson(8))
356   else
357     (if Z16 ==#2 0#2 then poisson(3) else poisson(8))
358 in
359
360 let Z17 =
361   if Z16 ==#2 0#2 then
362     discrete(0.8, 0.2)
363   else
364     discrete(0.2, 0.8)
365 in
366
367 let X17 =
368   if seperated ==#2 0#2 then
369     (if Z17 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
370   else
371     (if Z17 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
372 in
373
374 let Y17 =
375   if seperated ==#2 0#2 then
376     (if Z17 ==#2 0#2 then poisson(5) else poisson(8))
377   else
378     (if Z17 ==#2 0#2 then poisson(3) else poisson(8))
379 in
380
381 let Z18 =
382   if Z17 ==#2 0#2 then
383     discrete(0.8, 0.2)
384   else
385     discrete(0.2, 0.8)
386 in
387
388 let X18 =
389   if seperated ==#2 0#2 then
390     (if Z18 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
391   else
392     (if Z18 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
393 in
394
395 let Y18 =
396   if seperated ==#2 0#2 then
397     (if Z18 ==#2 0#2 then poisson(5) else poisson(8))
398   else
399     (if Z18 ==#2 0#2 then poisson(3) else poisson(8))
400 in
401
402 let Z19 =

```

```

403   if Z18 ==#2 0#2 then
404     discrete(0.8, 0.2)
405   else
406     discrete(0.2, 0.8)
407 in
408
409 let X19 =
410   if seperated ==#2 0#2 then
411     (if Z19 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
412   else
413     (if Z19 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
414 in
415
416 let Y19 =
417   if seperated ==#2 0#2 then
418     (if Z19 ==#2 0#2 then poisson(5) else poisson(8))
419   else
420     (if Z19 ==#2 0#2 then poisson(3) else poisson(8))
421 in
422
423 let Z20 =
424   if Z19 ==#2 0#2 then
425     discrete(0.8, 0.2)
426   else
427     discrete(0.2, 0.8)
428 in
429
430 let X20 =
431   if seperated ==#2 0#2 then
432     (if Z20 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
433   else
434     (if Z20 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
435 in
436
437 let Y20 =
438   if seperated ==#2 0#2 then
439     (if Z20 ==#2 0#2 then poisson(5) else poisson(8))
440   else
441     (if Z20 ==#2 0#2 then poisson(3) else poisson(8))
442 in
443
444 let Z21 =
445   if Z20 ==#2 0#2 then
446     discrete(0.8, 0.2)
447   else
448     discrete(0.2, 0.8)
449 in
450
451 let X21 =
452   if seperated ==#2 0#2 then
453     (if Z21 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
454   else
455     (if Z21 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
456 in
457
458 let Y21 =
459   if seperated ==#2 0#2 then
460     (if Z21 ==#2 0#2 then poisson(5) else poisson(8))
461   else
462     (if Z21 ==#2 0#2 then poisson(3) else poisson(8))
463 in
464
465 let Z22 =
466   if Z21 ==#2 0#2 then
467     discrete(0.8, 0.2)
468   else
469     discrete(0.2, 0.8)
470 in
471
472 let X22 =
473   if seperated ==#2 0#2 then

```

```

474     (if Z22 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
475   else
476     (if Z22 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
477 in
478
479 let Y22 =
480   if seperated ==#2 0#2 then
481     (if Z22 ==#2 0#2 then poisson(5) else poisson(8))
482   else
483     (if Z22 ==#2 0#2 then poisson(3) else poisson(8))
484 in
485
486 let Z23 =
487   if Z22 ==#2 0#2 then
488     discrete(0.8, 0.2)
489   else
490     discrete(0.2, 0.8)
491 in
492
493 let X23 =
494   if seperated ==#2 0#2 then
495     (if Z23 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
496   else
497     (if Z23 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
498 in
499
500 let Y23 =
501   if seperated ==#2 0#2 then
502     (if Z23 ==#2 0#2 then poisson(5) else poisson(8))
503   else
504     (if Z23 ==#2 0#2 then poisson(3) else poisson(8))
505 in
506
507 let Z24 =
508   if Z23 ==#2 0#2 then
509     discrete(0.8, 0.2)
510   else
511     discrete(0.2, 0.8)
512 in
513
514 let X24 =
515   if seperated ==#2 0#2 then
516     (if Z24 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
517   else
518     (if Z24 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
519 in
520
521 let Y24 =
522   if seperated ==#2 0#2 then
523     (if Z24 ==#2 0#2 then poisson(5) else poisson(8))
524   else
525     (if Z24 ==#2 0#2 then poisson(3) else poisson(8))
526 in
527
528 let Z25 =
529   if Z24 ==#2 0#2 then
530     discrete(0.8, 0.2)
531   else
532     discrete(0.2, 0.8)
533 in
534
535 let X25 =
536   if seperated ==#2 0#2 then
537     (if Z25 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
538   else
539     (if Z25 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
540 in
541
542 let Y25 =
543   if seperated ==#2 0#2 then
544     (if Z25 ==#2 0#2 then poisson(5) else poisson(8))

```

```

545     else
546       (if Z25 ==#2 0#2 then poisson(3) else poisson(8))
547   in
548
549   let Z26 =
550     if Z25 ==#2 0#2 then
551       discrete(0.8, 0.2)
552     else
553       discrete(0.2, 0.8)
554   in
555
556   let X26 =
557     if seperated ==#2 0#2 then
558       (if Z26 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
559     else
560       (if Z26 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
561   in
562
563   let Y26 =
564     if seperated ==#2 0#2 then
565       (if Z26 ==#2 0#2 then poisson(5) else poisson(8))
566     else
567       (if Z26 ==#2 0#2 then poisson(3) else poisson(8))
568   in
569
570   let Z27 =
571     if Z26 ==#2 0#2 then
572       discrete(0.8, 0.2)
573     else
574       discrete(0.2, 0.8)
575   in
576
577   let X27 =
578     if seperated ==#2 0#2 then
579       (if Z27 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
580     else
581       (if Z27 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
582   in
583
584   let Y27 =
585     if seperated ==#2 0#2 then
586       (if Z27 ==#2 0#2 then poisson(5) else poisson(8))
587     else
588       (if Z27 ==#2 0#2 then poisson(3) else poisson(8))
589   in
590
591   let Z28 =
592     if Z27 ==#2 0#2 then
593       discrete(0.8, 0.2)
594     else
595       discrete(0.2, 0.8)
596   in
597
598   let X28 =
599     if seperated ==#2 0#2 then
600       (if Z28 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
601     else
602       (if Z28 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
603   in
604
605   let Y28 =
606     if seperated ==#2 0#2 then
607       (if Z28 ==#2 0#2 then poisson(5) else poisson(8))
608     else
609       (if Z28 ==#2 0#2 then poisson(3) else poisson(8))
610   in
611
612   let Z29 =
613     if Z28 ==#2 0#2 then
614       discrete(0.8, 0.2)
615     else

```

```

616     discrete(0.2, 0.8)
617 in
618
619 let X29 =
620   if seperated ==#2 0#2 then
621     (if Z29 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
622   else
623     (if Z29 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
624 in
625
626 let Y29 =
627   if seperated ==#2 0#2 then
628     (if Z29 ==#2 0#2 then poisson(5) else poisson(8))
629   else
630     (if Z29 ==#2 0#2 then poisson(3) else poisson(8))
631 in
632
633 let Z30 =
634   if Z29 ==#2 0#2 then
635     discrete(0.8, 0.2)
636   else
637     discrete(0.2, 0.8)
638 in
639
640 let X30 =
641   if seperated ==#2 0#2 then
642     (if Z30 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
643   else
644     (if Z30 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
645 in
646
647 let Y30 =
648   if seperated ==#2 0#2 then
649     (if Z30 ==#2 0#2 then poisson(5) else poisson(8))
650   else
651     (if Z30 ==#2 0#2 then poisson(3) else poisson(8))
652 in
653
654 let Z31 =
655   if Z30 ==#2 0#2 then
656     discrete(0.8, 0.2)
657   else
658     discrete(0.2, 0.8)
659 in
660
661 let X31 =
662   if seperated ==#2 0#2 then
663     (if Z31 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
664   else
665     (if Z31 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
666 in
667
668 let Y31 =
669   if seperated ==#2 0#2 then
670     (if Z31 ==#2 0#2 then poisson(5) else poisson(8))
671   else
672     (if Z31 ==#2 0#2 then poisson(3) else poisson(8))
673 in
674
675 let Z32 =
676   if Z31 ==#2 0#2 then
677     discrete(0.8, 0.2)
678   else
679     discrete(0.2, 0.8)
680 in
681
682 let X32 =
683   if seperated ==#2 0#2 then
684     (if Z32 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
685   else
686     (if Z32 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))

```

```

687 in
688
689 let Y32 =
690   if seperated ==#2 0#2 then
691     (if Z32 ==#2 0#2 then poisson(5) else poisson(8))
692   else
693     (if Z32 ==#2 0#2 then poisson(3) else poisson(8))
694 in
695
696 let Z33 =
697   if Z32 ==#2 0#2 then
698     discrete(0.8, 0.2)
699   else
700     discrete(0.2, 0.8)
701 in
702
703 let X33 =
704   if seperated ==#2 0#2 then
705     (if Z33 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
706   else
707     (if Z33 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
708 in
709
710 let Y33 =
711   if seperated ==#2 0#2 then
712     (if Z33 ==#2 0#2 then poisson(5) else poisson(8))
713   else
714     (if Z33 ==#2 0#2 then poisson(3) else poisson(8))
715 in
716
717 let Z34 =
718   if Z33 ==#2 0#2 then
719     discrete(0.8, 0.2)
720   else
721     discrete(0.2, 0.8)
722 in
723
724 let X34 =
725   if seperated ==#2 0#2 then
726     (if Z34 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
727   else
728     (if Z34 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
729 in
730
731 let Y34 =
732   if seperated ==#2 0#2 then
733     (if Z34 ==#2 0#2 then poisson(5) else poisson(8))
734   else
735     (if Z34 ==#2 0#2 then poisson(3) else poisson(8))
736 in
737
738 let Z35 =
739   if Z34 ==#2 0#2 then
740     discrete(0.8, 0.2)
741   else
742     discrete(0.2, 0.8)
743 in
744
745 let X35 =
746   if seperated ==#2 0#2 then
747     (if Z35 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
748   else
749     (if Z35 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
750 in
751
752 let Y35 =
753   if seperated ==#2 0#2 then
754     (if Z35 ==#2 0#2 then poisson(5) else poisson(8))
755   else
756     (if Z35 ==#2 0#2 then poisson(3) else poisson(8))
757 in

```

```

758
759 let Z36 =
760   if Z35 ==#2 0#2 then
761     discrete(0.8, 0.2)
762   else
763     discrete(0.2, 0.8)
764 in
765
766 let X36 =
767   if seperated ==#2 0#2 then
768     (if Z36 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
769   else
770     (if Z36 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
771 in
772
773 let Y36 =
774   if seperated ==#2 0#2 then
775     (if Z36 ==#2 0#2 then poisson(5) else poisson(8))
776   else
777     (if Z36 ==#2 0#2 then poisson(3) else poisson(8))
778 in
779
780 let Z37 =
781   if Z36 ==#2 0#2 then
782     discrete(0.8, 0.2)
783   else
784     discrete(0.2, 0.8)
785 in
786
787 let X37 =
788   if seperated ==#2 0#2 then
789     (if Z37 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
790   else
791     (if Z37 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
792 in
793
794 let Y37 =
795   if seperated ==#2 0#2 then
796     (if Z37 ==#2 0#2 then poisson(5) else poisson(8))
797   else
798     (if Z37 ==#2 0#2 then poisson(3) else poisson(8))
799 in
800
801 let Z38 =
802   if Z37 ==#2 0#2 then
803     discrete(0.8, 0.2)
804   else
805     discrete(0.2, 0.8)
806 in
807
808 let X38 =
809   if seperated ==#2 0#2 then
810     (if Z38 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
811   else
812     (if Z38 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
813 in
814
815 let Y38 =
816   if seperated ==#2 0#2 then
817     (if Z38 ==#2 0#2 then poisson(5) else poisson(8))
818   else
819     (if Z38 ==#2 0#2 then poisson(3) else poisson(8))
820 in
821
822 let Z39 =
823   if Z38 ==#2 0#2 then
824     discrete(0.8, 0.2)
825   else
826     discrete(0.2, 0.8)
827 in
828

```

```

829 let X39 =
830   if seperated ==#2 0#2 then
831     (if Z39 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
832   else
833     (if Z39 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
834 in
835
836 let Y39 =
837   if seperated ==#2 0#2 then
838     (if Z39 ==#2 0#2 then poisson(5) else poisson(8))
839   else
840     (if Z39 ==#2 0#2 then poisson(3) else poisson(8))
841 in
842
843 let Z40 =
844   if Z39 ==#2 0#2 then
845     discrete(0.8, 0.2)
846   else
847     discrete(0.2, 0.8)
848 in
849
850 let X40 =
851   if seperated ==#2 0#2 then
852     (if Z40 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
853   else
854     (if Z40 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
855 in
856
857 let Y40 =
858   if seperated ==#2 0#2 then
859     (if Z40 ==#2 0#2 then poisson(5) else poisson(8))
860   else
861     (if Z40 ==#2 0#2 then poisson(3) else poisson(8))
862 in
863
864 let Z41 =
865   if Z40 ==#2 0#2 then
866     discrete(0.8, 0.2)
867   else
868     discrete(0.2, 0.8)
869 in
870
871 let X41 =
872   if seperated ==#2 0#2 then
873     (if Z41 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
874   else
875     (if Z41 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
876 in
877
878 let Y41 =
879   if seperated ==#2 0#2 then
880     (if Z41 ==#2 0#2 then poisson(5) else poisson(8))
881   else
882     (if Z41 ==#2 0#2 then poisson(3) else poisson(8))
883 in
884
885 let Z42 =
886   if Z41 ==#2 0#2 then
887     discrete(0.8, 0.2)
888   else
889     discrete(0.2, 0.8)
890 in
891
892 let X42 =
893   if seperated ==#2 0#2 then
894     (if Z42 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
895   else
896     (if Z42 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
897 in
898
899 let Y42 =

```

```

900  if seperated ==#2 0#2 then
901    (if Z42 ==#2 0#2 then poisson(5) else poisson(8))
902  else
903    (if Z42 ==#2 0#2 then poisson(3) else poisson(8))
904  in
905
906  let Z43 =
907    if Z42 ==#2 0#2 then
908      discrete(0.8, 0.2)
909    else
910      discrete(0.2, 0.8)
911  in
912
913  let X43 =
914    if seperated ==#2 0#2 then
915      (if Z43 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
916    else
917      (if Z43 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
918  in
919
920  let Y43 =
921    if seperated ==#2 0#2 then
922      (if Z43 ==#2 0#2 then poisson(5) else poisson(8))
923    else
924      (if Z43 ==#2 0#2 then poisson(3) else poisson(8))
925  in
926
927  let Z44 =
928    if Z43 ==#2 0#2 then
929      discrete(0.8, 0.2)
930    else
931      discrete(0.2, 0.8)
932  in
933
934  let X44 =
935    if seperated ==#2 0#2 then
936      (if Z44 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
937    else
938      (if Z44 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
939  in
940
941  let Y44 =
942    if seperated ==#2 0#2 then
943      (if Z44 ==#2 0#2 then poisson(5) else poisson(8))
944    else
945      (if Z44 ==#2 0#2 then poisson(3) else poisson(8))
946  in
947
948  let Z45 =
949    if Z44 ==#2 0#2 then
950      discrete(0.8, 0.2)
951    else
952      discrete(0.2, 0.8)
953  in
954
955  let X45 =
956    if seperated ==#2 0#2 then
957      (if Z45 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
958    else
959      (if Z45 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
960  in
961
962  let Y45 =
963    if seperated ==#2 0#2 then
964      (if Z45 ==#2 0#2 then poisson(5) else poisson(8))
965    else
966      (if Z45 ==#2 0#2 then poisson(3) else poisson(8))
967  in
968
969  let Z46 =
970    if Z45 ==#2 0#2 then

```

```

971     discrete(0.8, 0.2)
972   else
973     discrete(0.2, 0.8)
974   in
975
976   let X46 =
977     if seperated ==#2 0#2 then
978       (if Z46 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
979     else
980       (if Z46 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
981   in
982
983   let Y46 =
984     if seperated ==#2 0#2 then
985       (if Z46 ==#2 0#2 then poisson(5) else poisson(8))
986     else
987       (if Z46 ==#2 0#2 then poisson(3) else poisson(8))
988   in
989
990   let Z47 =
991     if Z46 ==#2 0#2 then
992       discrete(0.8, 0.2)
993     else
994       discrete(0.2, 0.8)
995   in
996
997   let X47 =
998     if seperated ==#2 0#2 then
999       (if Z47 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1000    else
1001      (if Z47 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1002  in
1003
1004  let Y47 =
1005    if seperated ==#2 0#2 then
1006      (if Z47 ==#2 0#2 then poisson(5) else poisson(8))
1007    else
1008      (if Z47 ==#2 0#2 then poisson(3) else poisson(8))
1009  in
1010
1011  let Z48 =
1012    if Z47 ==#2 0#2 then
1013      discrete(0.8, 0.2)
1014    else
1015      discrete(0.2, 0.8)
1016  in
1017
1018  let X48 =
1019    if seperated ==#2 0#2 then
1020      (if Z48 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1021    else
1022      (if Z48 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1023  in
1024
1025  let Y48 =
1026    if seperated ==#2 0#2 then
1027      (if Z48 ==#2 0#2 then poisson(5) else poisson(8))
1028    else
1029      (if Z48 ==#2 0#2 then poisson(3) else poisson(8))
1030  in
1031
1032  let Z49 =
1033    if Z48 ==#2 0#2 then
1034      discrete(0.8, 0.2)
1035    else
1036      discrete(0.2, 0.8)
1037  in
1038
1039  let X49 =
1040    if seperated ==#2 0#2 then
1041      (if Z49 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))

```

```

1042   else
1043     (if Z49 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1044   in
1045
1046   let Y49 =
1047     if seperated ==#2 0#2 then
1048       (if Z49 ==#2 0#2 then poisson(5) else poisson(8))
1049     else
1050       (if Z49 ==#2 0#2 then poisson(3) else poisson(8))
1051   in
1052
1053   let Z50 =
1054     if Z49 ==#2 0#2 then
1055       discrete(0.8, 0.2)
1056     else
1057       discrete(0.2, 0.8)
1058   in
1059
1060   let X50 =
1061     if seperated ==#2 0#2 then
1062       (if Z50 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1063     else
1064       (if Z50 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1065   in
1066
1067   let Y50 =
1068     if seperated ==#2 0#2 then
1069       (if Z50 ==#2 0#2 then poisson(5) else poisson(8))
1070     else
1071       (if Z50 ==#2 0#2 then poisson(3) else poisson(8))
1072   in
1073
1074   let Z51 =
1075     if Z50 ==#2 0#2 then
1076       discrete(0.8, 0.2)
1077     else
1078       discrete(0.2, 0.8)
1079   in
1080
1081   let X51 =
1082     if seperated ==#2 0#2 then
1083       (if Z51 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1084     else
1085       (if Z51 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1086   in
1087
1088   let Y51 =
1089     if seperated ==#2 0#2 then
1090       (if Z51 ==#2 0#2 then poisson(5) else poisson(8))
1091     else
1092       (if Z51 ==#2 0#2 then poisson(3) else poisson(8))
1093   in
1094
1095   let Z52 =
1096     if Z51 ==#2 0#2 then
1097       discrete(0.8, 0.2)
1098     else
1099       discrete(0.2, 0.8)
1100   in
1101
1102   let X52 =
1103     if seperated ==#2 0#2 then
1104       (if Z52 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1105     else
1106       (if Z52 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1107   in
1108
1109   let Y52 =
1110     if seperated ==#2 0#2 then
1111       (if Z52 ==#2 0#2 then poisson(5) else poisson(8))
1112     else

```

```

1113     (if Z52 ==#2 0#2 then poisson(3) else poisson(8))
1114 in
1115
1116 let Z53 =
1117   if Z52 ==#2 0#2 then
1118     discrete(0.8, 0.2)
1119   else
1120     discrete(0.2, 0.8)
1121 in
1122
1123 let X53 =
1124   if seperated ==#2 0#2 then
1125     (if Z53 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1126   else
1127     (if Z53 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1128 in
1129
1130 let Y53 =
1131   if seperated ==#2 0#2 then
1132     (if Z53 ==#2 0#2 then poisson(5) else poisson(8))
1133   else
1134     (if Z53 ==#2 0#2 then poisson(3) else poisson(8))
1135 in
1136
1137 let Z54 =
1138   if Z53 ==#2 0#2 then
1139     discrete(0.8, 0.2)
1140   else
1141     discrete(0.2, 0.8)
1142 in
1143
1144 let X54 =
1145   if seperated ==#2 0#2 then
1146     (if Z54 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1147   else
1148     (if Z54 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1149 in
1150
1151 let Y54 =
1152   if seperated ==#2 0#2 then
1153     (if Z54 ==#2 0#2 then poisson(5) else poisson(8))
1154   else
1155     (if Z54 ==#2 0#2 then poisson(3) else poisson(8))
1156 in
1157
1158 let Z55 =
1159   if Z54 ==#2 0#2 then
1160     discrete(0.8, 0.2)
1161   else
1162     discrete(0.2, 0.8)
1163 in
1164
1165 let X55 =
1166   if seperated ==#2 0#2 then
1167     (if Z55 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1168   else
1169     (if Z55 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1170 in
1171
1172 let Y55 =
1173   if seperated ==#2 0#2 then
1174     (if Z55 ==#2 0#2 then poisson(5) else poisson(8))
1175   else
1176     (if Z55 ==#2 0#2 then poisson(3) else poisson(8))
1177 in
1178
1179 let Z56 =
1180   if Z55 ==#2 0#2 then
1181     discrete(0.8, 0.2)
1182   else
1183     discrete(0.2, 0.8)

```

```

1184 in
1185
1186 let X56 =
1187   if seperated ==#2 0#2 then
1188     (if Z56 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1189   else
1190     (if Z56 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1191 in
1192
1193 let Y56 =
1194   if seperated ==#2 0#2 then
1195     (if Z56 ==#2 0#2 then poisson(5) else poisson(8))
1196   else
1197     (if Z56 ==#2 0#2 then poisson(3) else poisson(8))
1198 in
1199
1200 let Z57 =
1201   if Z56 ==#2 0#2 then
1202     discrete(0.8, 0.2)
1203   else
1204     discrete(0.2, 0.8)
1205 in
1206
1207 let X57 =
1208   if seperated ==#2 0#2 then
1209     (if Z57 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1210   else
1211     (if Z57 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1212 in
1213
1214 let Y57 =
1215   if seperated ==#2 0#2 then
1216     (if Z57 ==#2 0#2 then poisson(5) else poisson(8))
1217   else
1218     (if Z57 ==#2 0#2 then poisson(3) else poisson(8))
1219 in
1220
1221 let Z58 =
1222   if Z57 ==#2 0#2 then
1223     discrete(0.8, 0.2)
1224   else
1225     discrete(0.2, 0.8)
1226 in
1227
1228 let X58 =
1229   if seperated ==#2 0#2 then
1230     (if Z58 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1231   else
1232     (if Z58 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1233 in
1234
1235 let Y58 =
1236   if seperated ==#2 0#2 then
1237     (if Z58 ==#2 0#2 then poisson(5) else poisson(8))
1238   else
1239     (if Z58 ==#2 0#2 then poisson(3) else poisson(8))
1240 in
1241
1242 let Z59 =
1243   if Z58 ==#2 0#2 then
1244     discrete(0.8, 0.2)
1245   else
1246     discrete(0.2, 0.8)
1247 in
1248
1249 let X59 =
1250   if seperated ==#2 0#2 then
1251     (if Z59 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1252   else
1253     (if Z59 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1254 in

```

```

1255
1256 let Y59 =
1257   if seperated ==#2 0#2 then
1258     (if Z59 ==#2 0#2 then poisson(5) else poisson(8))
1259   else
1260     (if Z59 ==#2 0#2 then poisson(3) else poisson(8))
1261 in
1262
1263 let Z60 =
1264   if Z59 ==#2 0#2 then
1265     discrete(0.8, 0.2)
1266   else
1267     discrete(0.2, 0.8)
1268 in
1269
1270 let X60 =
1271   if seperated ==#2 0#2 then
1272     (if Z60 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1273   else
1274     (if Z60 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1275 in
1276
1277 let Y60 =
1278   if seperated ==#2 0#2 then
1279     (if Z60 ==#2 0#2 then poisson(5) else poisson(8))
1280   else
1281     (if Z60 ==#2 0#2 then poisson(3) else poisson(8))
1282 in
1283
1284 let Z61 =
1285   if Z60 ==#2 0#2 then
1286     discrete(0.8, 0.2)
1287   else
1288     discrete(0.2, 0.8)
1289 in
1290
1291 let X61 =
1292   if seperated ==#2 0#2 then
1293     (if Z61 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1294   else
1295     (if Z61 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1296 in
1297
1298 let Y61 =
1299   if seperated ==#2 0#2 then
1300     (if Z61 ==#2 0#2 then poisson(5) else poisson(8))
1301   else
1302     (if Z61 ==#2 0#2 then poisson(3) else poisson(8))
1303 in
1304
1305 let Z62 =
1306   if Z61 ==#2 0#2 then
1307     discrete(0.8, 0.2)
1308   else
1309     discrete(0.2, 0.8)
1310 in
1311
1312 let X62 =
1313   if seperated ==#2 0#2 then
1314     (if Z62 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1315   else
1316     (if Z62 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1317 in
1318
1319 let Y62 =
1320   if seperated ==#2 0#2 then
1321     (if Z62 ==#2 0#2 then poisson(5) else poisson(8))
1322   else
1323     (if Z62 ==#2 0#2 then poisson(3) else poisson(8))
1324 in
1325

```

```

1326 let Z63 =
1327   if Z62 ==#2 0#2 then
1328     discrete(0.8, 0.2)
1329   else
1330     discrete(0.2, 0.8)
1331 in
1332
1333 let X63 =
1334   if seperated ==#2 0#2 then
1335     (if Z63 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1336   else
1337     (if Z63 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1338 in
1339
1340 let Y63 =
1341   if seperated ==#2 0#2 then
1342     (if Z63 ==#2 0#2 then poisson(5) else poisson(8))
1343   else
1344     (if Z63 ==#2 0#2 then poisson(3) else poisson(8))
1345 in
1346
1347 let Z64 =
1348   if Z63 ==#2 0#2 then
1349     discrete(0.8, 0.2)
1350   else
1351     discrete(0.2, 0.8)
1352 in
1353
1354 let X64 =
1355   if seperated ==#2 0#2 then
1356     (if Z64 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1357   else
1358     (if Z64 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1359 in
1360
1361 let Y64 =
1362   if seperated ==#2 0#2 then
1363     (if Z64 ==#2 0#2 then poisson(5) else poisson(8))
1364   else
1365     (if Z64 ==#2 0#2 then poisson(3) else poisson(8))
1366 in
1367
1368 let Z65 =
1369   if Z64 ==#2 0#2 then
1370     discrete(0.8, 0.2)
1371   else
1372     discrete(0.2, 0.8)
1373 in
1374
1375 let X65 =
1376   if seperated ==#2 0#2 then
1377     (if Z65 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1378   else
1379     (if Z65 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1380 in
1381
1382 let Y65 =
1383   if seperated ==#2 0#2 then
1384     (if Z65 ==#2 0#2 then poisson(5) else poisson(8))
1385   else
1386     (if Z65 ==#2 0#2 then poisson(3) else poisson(8))
1387 in
1388
1389 let Z66 =
1390   if Z65 ==#2 0#2 then
1391     discrete(0.8, 0.2)
1392   else
1393     discrete(0.2, 0.8)
1394 in
1395
1396 let X66 =

```

```

1397   if seperated ==#2 0#2 then
1398     (if Z66 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1399   else
1400     (if Z66 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1401   in
1402
1403   let Y66 =
1404     if seperated ==#2 0#2 then
1405       (if Z66 ==#2 0#2 then poisson(5) else poisson(8))
1406     else
1407       (if Z66 ==#2 0#2 then poisson(3) else poisson(8))
1408   in
1409
1410   let Z67 =
1411     if Z66 ==#2 0#2 then
1412       discrete(0.8, 0.2)
1413     else
1414       discrete(0.2, 0.8)
1415   in
1416
1417   let X67 =
1418     if seperated ==#2 0#2 then
1419       (if Z67 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1420     else
1421       (if Z67 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1422   in
1423
1424   let Y67 =
1425     if seperated ==#2 0#2 then
1426       (if Z67 ==#2 0#2 then poisson(5) else poisson(8))
1427     else
1428       (if Z67 ==#2 0#2 then poisson(3) else poisson(8))
1429   in
1430
1431   let Z68 =
1432     if Z67 ==#2 0#2 then
1433       discrete(0.8, 0.2)
1434     else
1435       discrete(0.2, 0.8)
1436   in
1437
1438   let X68 =
1439     if seperated ==#2 0#2 then
1440       (if Z68 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1441     else
1442       (if Z68 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1443   in
1444
1445   let Y68 =
1446     if seperated ==#2 0#2 then
1447       (if Z68 ==#2 0#2 then poisson(5) else poisson(8))
1448     else
1449       (if Z68 ==#2 0#2 then poisson(3) else poisson(8))
1450   in
1451
1452   let Z69 =
1453     if Z68 ==#2 0#2 then
1454       discrete(0.8, 0.2)
1455     else
1456       discrete(0.2, 0.8)
1457   in
1458
1459   let X69 =
1460     if seperated ==#2 0#2 then
1461       (if Z69 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1462     else
1463       (if Z69 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1464   in
1465
1466   let Y69 =
1467     if seperated ==#2 0#2 then

```

```

1468     (if Z69 ==#2 0#2 then poisson(5) else poisson(8))
1469   else
1470     (if Z69 ==#2 0#2 then poisson(3) else poisson(8))
1471 in
1472
1473 let Z70 =
1474   if Z69 ==#2 0#2 then
1475     discrete(0.8, 0.2)
1476   else
1477     discrete(0.2, 0.8)
1478 in
1479
1480 let X70 =
1481   if seperated ==#2 0#2 then
1482     (if Z70 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1483   else
1484     (if Z70 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1485 in
1486
1487 let Y70 =
1488   if seperated ==#2 0#2 then
1489     (if Z70 ==#2 0#2 then poisson(5) else poisson(8))
1490   else
1491     (if Z70 ==#2 0#2 then poisson(3) else poisson(8))
1492 in
1493
1494 let Z71 =
1495   if Z70 ==#2 0#2 then
1496     discrete(0.8, 0.2)
1497   else
1498     discrete(0.2, 0.8)
1499 in
1500
1501 let X71 =
1502   if seperated ==#2 0#2 then
1503     (if Z71 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1504   else
1505     (if Z71 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1506 in
1507
1508 let Y71 =
1509   if seperated ==#2 0#2 then
1510     (if Z71 ==#2 0#2 then poisson(5) else poisson(8))
1511   else
1512     (if Z71 ==#2 0#2 then poisson(3) else poisson(8))
1513 in
1514
1515 let Z72 =
1516   if Z71 ==#2 0#2 then
1517     discrete(0.8, 0.2)
1518   else
1519     discrete(0.2, 0.8)
1520 in
1521
1522 let X72 =
1523   if seperated ==#2 0#2 then
1524     (if Z72 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1525   else
1526     (if Z72 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1527 in
1528
1529 let Y72 =
1530   if seperated ==#2 0#2 then
1531     (if Z72 ==#2 0#2 then poisson(5) else poisson(8))
1532   else
1533     (if Z72 ==#2 0#2 then poisson(3) else poisson(8))
1534 in
1535
1536 let Z73 =
1537   if Z72 ==#2 0#2 then
1538     discrete(0.8, 0.2)

```

```

1539     else
1540         discrete(0.2, 0.8)
1541 in
1542
1543 let X73 =
1544     if seperated ==#2 0#2 then
1545         (if Z73 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1546     else
1547         (if Z73 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1548 in
1549
1550 let Y73 =
1551     if seperated ==#2 0#2 then
1552         (if Z73 ==#2 0#2 then poisson(5) else poisson(8))
1553     else
1554         (if Z73 ==#2 0#2 then poisson(3) else poisson(8))
1555 in
1556
1557 let Z74 =
1558     if Z73 ==#2 0#2 then
1559         discrete(0.8, 0.2)
1560     else
1561         discrete(0.2, 0.8)
1562 in
1563
1564 let X74 =
1565     if seperated ==#2 0#2 then
1566         (if Z74 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1567     else
1568         (if Z74 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1569 in
1570
1571 let Y74 =
1572     if seperated ==#2 0#2 then
1573         (if Z74 ==#2 0#2 then poisson(5) else poisson(8))
1574     else
1575         (if Z74 ==#2 0#2 then poisson(3) else poisson(8))
1576 in
1577
1578 let Z75 =
1579     if Z74 ==#2 0#2 then
1580         discrete(0.8, 0.2)
1581     else
1582         discrete(0.2, 0.8)
1583 in
1584
1585 let X75 =
1586     if seperated ==#2 0#2 then
1587         (if Z75 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1588     else
1589         (if Z75 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1590 in
1591
1592 let Y75 =
1593     if seperated ==#2 0#2 then
1594         (if Z75 ==#2 0#2 then poisson(5) else poisson(8))
1595     else
1596         (if Z75 ==#2 0#2 then poisson(3) else poisson(8))
1597 in
1598
1599 let Z76 =
1600     if Z75 ==#2 0#2 then
1601         discrete(0.8, 0.2)
1602     else
1603         discrete(0.2, 0.8)
1604 in
1605
1606 let X76 =
1607     if seperated ==#2 0#2 then
1608         (if Z76 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1609     else

```

```

1610     (if Z76 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1611 in
1612
1613 let Y76 =
1614   if seperated ==#2 0#2 then
1615     (if Z76 ==#2 0#2 then poisson(5) else poisson(8))
1616   else
1617     (if Z76 ==#2 0#2 then poisson(3) else poisson(8))
1618 in
1619
1620 let Z77 =
1621   if Z76 ==#2 0#2 then
1622     discrete(0.8, 0.2)
1623   else
1624     discrete(0.2, 0.8)
1625 in
1626
1627 let X77 =
1628   if seperated ==#2 0#2 then
1629     (if Z77 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1630   else
1631     (if Z77 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1632 in
1633
1634 let Y77 =
1635   if seperated ==#2 0#2 then
1636     (if Z77 ==#2 0#2 then poisson(5) else poisson(8))
1637   else
1638     (if Z77 ==#2 0#2 then poisson(3) else poisson(8))
1639 in
1640
1641 let Z78 =
1642   if Z77 ==#2 0#2 then
1643     discrete(0.8, 0.2)
1644   else
1645     discrete(0.2, 0.8)
1646 in
1647
1648 let X78 =
1649   if seperated ==#2 0#2 then
1650     (if Z78 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1651   else
1652     (if Z78 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1653 in
1654
1655 let Y78 =
1656   if seperated ==#2 0#2 then
1657     (if Z78 ==#2 0#2 then poisson(5) else poisson(8))
1658   else
1659     (if Z78 ==#2 0#2 then poisson(3) else poisson(8))
1660 in
1661
1662 let Z79 =
1663   if Z78 ==#2 0#2 then
1664     discrete(0.8, 0.2)
1665   else
1666     discrete(0.2, 0.8)
1667 in
1668
1669 let X79 =
1670   if seperated ==#2 0#2 then
1671     (if Z79 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1672   else
1673     (if Z79 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1674 in
1675
1676 let Y79 =
1677   if seperated ==#2 0#2 then
1678     (if Z79 ==#2 0#2 then poisson(5) else poisson(8))
1679   else
1679     (if Z79 ==#2 0#2 then poisson(3) else poisson(8))
1680

```

```

1681 in
1682
1683 let Z80 =
1684   if Z79 ==#2 0#2 then
1685     discrete(0.8, 0.2)
1686   else
1687     discrete(0.2, 0.8)
1688 in
1689
1690 let X80 =
1691   if seperated ==#2 0#2 then
1692     (if Z80 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1693   else
1694     (if Z80 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1695 in
1696
1697 let Y80 =
1698   if seperated ==#2 0#2 then
1699     (if Z80 ==#2 0#2 then poisson(5) else poisson(8))
1700   else
1701     (if Z80 ==#2 0#2 then poisson(3) else poisson(8))
1702 in
1703
1704 let Z81 =
1705   if Z80 ==#2 0#2 then
1706     discrete(0.8, 0.2)
1707   else
1708     discrete(0.2, 0.8)
1709 in
1710
1711 let X81 =
1712   if seperated ==#2 0#2 then
1713     (if Z81 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1714   else
1715     (if Z81 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1716 in
1717
1718 let Y81 =
1719   if seperated ==#2 0#2 then
1720     (if Z81 ==#2 0#2 then poisson(5) else poisson(8))
1721   else
1722     (if Z81 ==#2 0#2 then poisson(3) else poisson(8))
1723 in
1724
1725 let Z82 =
1726   if Z81 ==#2 0#2 then
1727     discrete(0.8, 0.2)
1728   else
1729     discrete(0.2, 0.8)
1730 in
1731
1732 let X82 =
1733   if seperated ==#2 0#2 then
1734     (if Z82 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1735   else
1736     (if Z82 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1737 in
1738
1739 let Y82 =
1740   if seperated ==#2 0#2 then
1741     (if Z82 ==#2 0#2 then poisson(5) else poisson(8))
1742   else
1743     (if Z82 ==#2 0#2 then poisson(3) else poisson(8))
1744 in
1745
1746 let Z83 =
1747   if Z82 ==#2 0#2 then
1748     discrete(0.8, 0.2)
1749   else
1750     discrete(0.2, 0.8)
1751 in

```

```

1752
1753 let X83 =
1754   if seperated ==#2 0#2 then
1755     (if Z83 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1756   else
1757     (if Z83 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1758 in
1759
1760 let Y83 =
1761   if seperated ==#2 0#2 then
1762     (if Z83 ==#2 0#2 then poisson(5) else poisson(8))
1763   else
1764     (if Z83 ==#2 0#2 then poisson(3) else poisson(8))
1765 in
1766
1767 let Z84 =
1768   if Z83 ==#2 0#2 then
1769     discrete(0.8, 0.2)
1770   else
1771     discrete(0.2, 0.8)
1772 in
1773
1774 let X84 =
1775   if seperated ==#2 0#2 then
1776     (if Z84 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1777   else
1778     (if Z84 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1779 in
1780
1781 let Y84 =
1782   if seperated ==#2 0#2 then
1783     (if Z84 ==#2 0#2 then poisson(5) else poisson(8))
1784   else
1785     (if Z84 ==#2 0#2 then poisson(3) else poisson(8))
1786 in
1787
1788 let Z85 =
1789   if Z84 ==#2 0#2 then
1790     discrete(0.8, 0.2)
1791   else
1792     discrete(0.2, 0.8)
1793 in
1794
1795 let X85 =
1796   if seperated ==#2 0#2 then
1797     (if Z85 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1798   else
1799     (if Z85 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1800 in
1801
1802 let Y85 =
1803   if seperated ==#2 0#2 then
1804     (if Z85 ==#2 0#2 then poisson(5) else poisson(8))
1805   else
1806     (if Z85 ==#2 0#2 then poisson(3) else poisson(8))
1807 in
1808
1809 let Z86 =
1810   if Z85 ==#2 0#2 then
1811     discrete(0.8, 0.2)
1812   else
1813     discrete(0.2, 0.8)
1814 in
1815
1816 let X86 =
1817   if seperated ==#2 0#2 then
1818     (if Z86 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1819   else
1820     (if Z86 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1821 in
1822

```

```

1823 let Y86 =
1824   if seperated ==#2 0#2 then
1825     (if Z86 ==#2 0#2 then poisson(5) else poisson(8))
1826   else
1827     (if Z86 ==#2 0#2 then poisson(3) else poisson(8))
1828 in
1829
1830 let Z87 =
1831   if Z86 ==#2 0#2 then
1832     discrete(0.8, 0.2)
1833   else
1834     discrete(0.2, 0.8)
1835 in
1836
1837 let X87 =
1838   if seperated ==#2 0#2 then
1839     (if Z87 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1840   else
1841     (if Z87 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1842 in
1843
1844 let Y87 =
1845   if seperated ==#2 0#2 then
1846     (if Z87 ==#2 0#2 then poisson(5) else poisson(8))
1847   else
1848     (if Z87 ==#2 0#2 then poisson(3) else poisson(8))
1849 in
1850
1851 let Z88 =
1852   if Z87 ==#2 0#2 then
1853     discrete(0.8, 0.2)
1854   else
1855     discrete(0.2, 0.8)
1856 in
1857
1858 let X88 =
1859   if seperated ==#2 0#2 then
1860     (if Z88 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1861   else
1862     (if Z88 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1863 in
1864
1865 let Y88 =
1866   if seperated ==#2 0#2 then
1867     (if Z88 ==#2 0#2 then poisson(5) else poisson(8))
1868   else
1869     (if Z88 ==#2 0#2 then poisson(3) else poisson(8))
1870 in
1871
1872 let Z89 =
1873   if Z88 ==#2 0#2 then
1874     discrete(0.8, 0.2)
1875   else
1876     discrete(0.2, 0.8)
1877 in
1878
1879 let X89 =
1880   if seperated ==#2 0#2 then
1881     (if Z89 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1882   else
1883     (if Z89 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1884 in
1885
1886 let Y89 =
1887   if seperated ==#2 0#2 then
1888     (if Z89 ==#2 0#2 then poisson(5) else poisson(8))
1889   else
1890     (if Z89 ==#2 0#2 then poisson(3) else poisson(8))
1891 in
1892
1893 let Z90 =

```

```

1894   if Z89 ==#2 0#2 then
1895       discrete(0.8, 0.2)
1896   else
1897       discrete(0.2, 0.8)
1898   in
1899
1900   let X90 =
1901       if seperated ==#2 0#2 then
1902           (if Z90 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1903       else
1904           (if Z90 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1905   in
1906
1907   let Y90 =
1908       if seperated ==#2 0#2 then
1909           (if Z90 ==#2 0#2 then poisson(5) else poisson(8))
1910       else
1911           (if Z90 ==#2 0#2 then poisson(3) else poisson(8))
1912   in
1913
1914   let Z91 =
1915       if Z90 ==#2 0#2 then
1916           discrete(0.8, 0.2)
1917       else
1918           discrete(0.2, 0.8)
1919   in
1920
1921   let X91 =
1922       if seperated ==#2 0#2 then
1923           (if Z91 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1924       else
1925           (if Z91 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1926   in
1927
1928   let Y91 =
1929       if seperated ==#2 0#2 then
1930           (if Z91 ==#2 0#2 then poisson(5) else poisson(8))
1931       else
1932           (if Z91 ==#2 0#2 then poisson(3) else poisson(8))
1933   in
1934
1935   let Z92 =
1936       if Z91 ==#2 0#2 then
1937           discrete(0.8, 0.2)
1938       else
1939           discrete(0.2, 0.8)
1940   in
1941
1942   let X92 =
1943       if seperated ==#2 0#2 then
1944           (if Z92 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1945       else
1946           (if Z92 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1947   in
1948
1949   let Y92 =
1950       if seperated ==#2 0#2 then
1951           (if Z92 ==#2 0#2 then poisson(5) else poisson(8))
1952       else
1953           (if Z92 ==#2 0#2 then poisson(3) else poisson(8))
1954   in
1955
1956   let Z93 =
1957       if Z92 ==#2 0#2 then
1958           discrete(0.8, 0.2)
1959       else
1960           discrete(0.2, 0.8)
1961   in
1962
1963   let X93 =
1964       if seperated ==#2 0#2 then

```

```

1965     (if Z93 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1966   else
1967     (if Z93 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1968   in
1969
1970   let Y93 =
1971     if seperated ==#2 0#2 then
1972       (if Z93 ==#2 0#2 then poisson(5) else poisson(8))
1973     else
1974       (if Z93 ==#2 0#2 then poisson(3) else poisson(8))
1975   in
1976
1977   let Z94 =
1978     if Z93 ==#2 0#2 then
1979       discrete(0.8, 0.2)
1980     else
1981       discrete(0.2, 0.8)
1982   in
1983
1984   let X94 =
1985     if seperated ==#2 0#2 then
1986       (if Z94 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
1987     else
1988       (if Z94 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
1989   in
1990
1991   let Y94 =
1992     if seperated ==#2 0#2 then
1993       (if Z94 ==#2 0#2 then poisson(5) else poisson(8))
1994     else
1995       (if Z94 ==#2 0#2 then poisson(3) else poisson(8))
1996   in
1997
1998   let Z95 =
1999     if Z94 ==#2 0#2 then
2000       discrete(0.8, 0.2)
2001     else
2002       discrete(0.2, 0.8)
2003   in
2004
2005   let X95 =
2006     if seperated ==#2 0#2 then
2007       (if Z95 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
2008     else
2009       (if Z95 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
2010   in
2011
2012   let Y95 =
2013     if seperated ==#2 0#2 then
2014       (if Z95 ==#2 0#2 then poisson(5) else poisson(8))
2015     else
2016       (if Z95 ==#2 0#2 then poisson(3) else poisson(8))
2017   in
2018
2019   let Z96 =
2020     if Z95 ==#2 0#2 then
2021       discrete(0.8, 0.2)
2022     else
2023       discrete(0.2, 0.8)
2024   in
2025
2026   let X96 =
2027     if seperated ==#2 0#2 then
2028       (if Z96 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
2029     else
2030       (if Z96 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
2031   in
2032
2033   let Y96 =
2034     if seperated ==#2 0#2 then
2035       (if Z96 ==#2 0#2 then poisson(5) else poisson(8))

```

```

2036     else
2037         (if Z96 ==#2 0#2 then poisson(3) else poisson(8))
2038 in
2039
2040 let Z97 =
2041     if Z96 ==#2 0#2 then
2042         discrete(0.8, 0.2)
2043     else
2044         discrete(0.2, 0.8)
2045 in
2046
2047 let X97 =
2048     if seperated ==#2 0#2 then
2049         (if Z97 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
2050     else
2051         (if Z97 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
2052 in
2053
2054 let Y97 =
2055     if seperated ==#2 0#2 then
2056         (if Z97 ==#2 0#2 then poisson(5) else poisson(8))
2057     else
2058         (if Z97 ==#2 0#2 then poisson(3) else poisson(8))
2059 in
2060
2061 let Z98 =
2062     if Z97 ==#2 0#2 then
2063         discrete(0.8, 0.2)
2064     else
2065         discrete(0.2, 0.8)
2066 in
2067
2068 let X98 =
2069     if seperated ==#2 0#2 then
2070         (if Z98 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
2071     else
2072         (if Z98 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
2073 in
2074
2075 let Y98 =
2076     if seperated ==#2 0#2 then
2077         (if Z98 ==#2 0#2 then poisson(5) else poisson(8))
2078     else
2079         (if Z98 ==#2 0#2 then poisson(3) else poisson(8))
2080 in
2081
2082 let Z99 =
2083     if Z98 ==#2 0#2 then
2084         discrete(0.8, 0.2)
2085     else
2086         discrete(0.2, 0.8)
2087 in
2088
2089 let X99 =
2090     if seperated ==#2 0#2 then
2091         (if Z99 ==#2 0#2 then gaussian(5, 1) else gaussian(7, 1))
2092     else
2093         (if Z99 ==#2 0#2 then gaussian(5, 1) else gaussian(15, 1))
2094 in
2095
2096 let Y99 =
2097     if seperated ==#2 0#2 then
2098         (if Z99 ==#2 0#2 then poisson(5) else poisson(8))
2099     else
2100         (if Z99 ==#2 0#2 then poisson(3) else poisson(8))
2101 in
2102
2103 Z0 <#2 1#2

```

StudentInterviews

```

1  let student_perfect_0 = discrete(0.8, 0.2) in
2  let student_gpa_0 = if student_perfect_0 ==#2 0#2 then beta(7, 3) else 1.0 in
3
4  let student_perfect_1 = discrete(0.8, 0.2) in
5  let student_gpa_1 = if student_perfect_1 ==#2 0#2 then beta(7, 3) else 1.0 in
6
7  let num_recruiters = poisson(25) in
8  let _recruiters_range = observe(10 <= num_recruiters && num_recruiters <= 40) in
9
10 let student_interviews_0 =
11   if num_recruiters <= 10 then
12     if student_perfect_0 ==#2 1#2 then binomial(0.9, 10)
13     else if student_gpa_0 > 0.875 then binomial(0.6, 10)
14     else binomial(0.5, 10)
15   else if num_recruiters <= 11 then
16     if student_perfect_0 ==#2 1#2 then binomial(0.9, 11)
17     else if student_gpa_0 > 0.875 then binomial(0.6, 11)
18     else binomial(0.5, 11)
19   else if num_recruiters <= 12 then
20     if student_perfect_0 ==#2 1#2 then binomial(0.9, 12)
21     else if student_gpa_0 > 0.875 then binomial(0.6, 12)
22     else binomial(0.5, 12)
23   else if num_recruiters <= 13 then
24     if student_perfect_0 ==#2 1#2 then binomial(0.9, 13)
25     else if student_gpa_0 > 0.875 then binomial(0.6, 13)
26     else binomial(0.5, 13)
27   else if num_recruiters <= 14 then
28     if student_perfect_0 ==#2 1#2 then binomial(0.9, 14)
29     else if student_gpa_0 > 0.875 then binomial(0.6, 14)
30     else binomial(0.5, 14)
31   else if num_recruiters <= 15 then
32     if student_perfect_0 ==#2 1#2 then binomial(0.9, 15)
33     else if student_gpa_0 > 0.875 then binomial(0.6, 15)
34     else binomial(0.5, 15)
35   else if num_recruiters <= 16 then
36     if student_perfect_0 ==#2 1#2 then binomial(0.9, 16)
37     else if student_gpa_0 > 0.875 then binomial(0.6, 16)
38     else binomial(0.5, 16)
39   else if num_recruiters <= 17 then
40     if student_perfect_0 ==#2 1#2 then binomial(0.9, 17)
41     else if student_gpa_0 > 0.875 then binomial(0.6, 17)
42     else binomial(0.5, 17)
43   else if num_recruiters <= 18 then
44     if student_perfect_0 ==#2 1#2 then binomial(0.9, 18)
45     else if student_gpa_0 > 0.875 then binomial(0.6, 18)
46     else binomial(0.5, 18)
47   else if num_recruiters <= 19 then
48     if student_perfect_0 ==#2 1#2 then binomial(0.9, 19)
49     else if student_gpa_0 > 0.875 then binomial(0.6, 19)
50     else binomial(0.5, 19)
51   else if num_recruiters <= 20 then
52     if student_perfect_0 ==#2 1#2 then binomial(0.9, 20)
53     else if student_gpa_0 > 0.875 then binomial(0.6, 20)
54     else binomial(0.5, 20)
55   else if num_recruiters <= 21 then
56     if student_perfect_0 ==#2 1#2 then binomial(0.9, 21)
57     else if student_gpa_0 > 0.875 then binomial(0.6, 21)
58     else binomial(0.5, 21)
59   else if num_recruiters <= 22 then
60     if student_perfect_0 ==#2 1#2 then binomial(0.9, 22)
61     else if student_gpa_0 > 0.875 then binomial(0.6, 22)
62     else binomial(0.5, 22)
63   else if num_recruiters <= 23 then
64     if student_perfect_0 ==#2 1#2 then binomial(0.9, 23)
65     else if student_gpa_0 > 0.875 then binomial(0.6, 23)
66     else binomial(0.5, 23)
67   else if num_recruiters <= 24 then
68     if student_perfect_0 ==#2 1#2 then binomial(0.9, 24)
69     else if student_gpa_0 > 0.875 then binomial(0.6, 24)
70     else binomial(0.5, 24)
71   else if num_recruiters <= 25 then

```

```

72     if student_perfect_0 ==#2 1#2 then binomial(0.9, 25)
73     else if student_gpa_0 > 0.875 then binomial(0.6, 25)
74     else binomial(0.5, 25)
75   else if num_recruiters <= 26 then
76     if student_perfect_0 ==#2 1#2 then binomial(0.9, 26)
77     else if student_gpa_0 > 0.875 then binomial(0.6, 26)
78     else binomial(0.5, 26)
79   else if num_recruiters <= 27 then
80     if student_perfect_0 ==#2 1#2 then binomial(0.9, 27)
81     else if student_gpa_0 > 0.875 then binomial(0.6, 27)
82     else binomial(0.5, 27)
83   else if num_recruiters <= 28 then
84     if student_perfect_0 ==#2 1#2 then binomial(0.9, 28)
85     else if student_gpa_0 > 0.875 then binomial(0.6, 28)
86     else binomial(0.5, 28)
87   else if num_recruiters <= 29 then
88     if student_perfect_0 ==#2 1#2 then binomial(0.9, 29)
89     else if student_gpa_0 > 0.875 then binomial(0.6, 29)
90     else binomial(0.5, 29)
91   else if num_recruiters <= 30 then
92     if student_perfect_0 ==#2 1#2 then binomial(0.9, 30)
93     else if student_gpa_0 > 0.875 then binomial(0.6, 30)
94     else binomial(0.5, 30)
95   else if num_recruiters <= 31 then
96     if student_perfect_0 ==#2 1#2 then binomial(0.9, 31)
97     else if student_gpa_0 > 0.875 then binomial(0.6, 31)
98     else binomial(0.5, 31)
99   else if num_recruiters <= 32 then
100    if student_perfect_0 ==#2 1#2 then binomial(0.9, 32)
101    else if student_gpa_0 > 0.875 then binomial(0.6, 32)
102    else binomial(0.5, 32)
103  else if num_recruiters <= 33 then
104    if student_perfect_0 ==#2 1#2 then binomial(0.9, 33)
105    else if student_gpa_0 > 0.875 then binomial(0.6, 33)
106    else binomial(0.5, 33)
107  else if num_recruiters <= 34 then
108    if student_perfect_0 ==#2 1#2 then binomial(0.9, 34)
109    else if student_gpa_0 > 0.875 then binomial(0.6, 34)
110    else binomial(0.5, 34)
111  else if num_recruiters <= 35 then
112    if student_perfect_0 ==#2 1#2 then binomial(0.9, 35)
113    else if student_gpa_0 > 0.875 then binomial(0.6, 35)
114    else binomial(0.5, 35)
115  else if num_recruiters <= 36 then
116    if student_perfect_0 ==#2 1#2 then binomial(0.9, 36)
117    else if student_gpa_0 > 0.875 then binomial(0.6, 36)
118    else binomial(0.5, 36)
119  else if num_recruiters <= 37 then
120    if student_perfect_0 ==#2 1#2 then binomial(0.9, 37)
121    else if student_gpa_0 > 0.875 then binomial(0.6, 37)
122    else binomial(0.5, 37)
123  else if num_recruiters <= 38 then
124    if student_perfect_0 ==#2 1#2 then binomial(0.9, 38)
125    else if student_gpa_0 > 0.875 then binomial(0.6, 38)
126    else binomial(0.5, 38)
127  else if num_recruiters <= 39 then
128    if student_perfect_0 ==#2 1#2 then binomial(0.9, 39)
129    else if student_gpa_0 > 0.875 then binomial(0.6, 39)
130    else binomial(0.5, 39)
131  else
132    if student_perfect_0 ==#2 1#2 then binomial(0.9, 40)
133    else if student_gpa_0 > 0.875 then binomial(0.6, 40)
134    else binomial(0.5, 40)
135  in
136
137  let student_interviews_1 =
138    if num_recruiters <= 10 then
139      if student_perfect_1 ==#2 1#2 then binomial(0.9, 10)
140      else if student_gpa_1 > 0.875 then binomial(0.6, 10)
141      else binomial(0.5, 10)
142    else if num_recruiters <= 11 then

```

```

143     if student_perfect_1 ==#2 1#2 then binomial(0.9, 11)
144     else if student_gpa_1 > 0.875 then binomial(0.6, 11)
145     else binomial(0.5, 11)
146   else if num_recruiters <= 12 then
147     if student_perfect_1 ==#2 1#2 then binomial(0.9, 12)
148     else if student_gpa_1 > 0.875 then binomial(0.6, 12)
149     else binomial(0.5, 12)
150   else if num_recruiters <= 13 then
151     if student_perfect_1 ==#2 1#2 then binomial(0.9, 13)
152     else if student_gpa_1 > 0.875 then binomial(0.6, 13)
153     else binomial(0.5, 13)
154   else if num_recruiters <= 14 then
155     if student_perfect_1 ==#2 1#2 then binomial(0.9, 14)
156     else if student_gpa_1 > 0.875 then binomial(0.6, 14)
157     else binomial(0.5, 14)
158   else if num_recruiters <= 15 then
159     if student_perfect_1 ==#2 1#2 then binomial(0.9, 15)
160     else if student_gpa_1 > 0.875 then binomial(0.6, 15)
161     else binomial(0.5, 15)
162   else if num_recruiters <= 16 then
163     if student_perfect_1 ==#2 1#2 then binomial(0.9, 16)
164     else if student_gpa_1 > 0.875 then binomial(0.6, 16)
165     else binomial(0.5, 16)
166   else if num_recruiters <= 17 then
167     if student_perfect_1 ==#2 1#2 then binomial(0.9, 17)
168     else if student_gpa_1 > 0.875 then binomial(0.6, 17)
169     else binomial(0.5, 17)
170   else if num_recruiters <= 18 then
171     if student_perfect_1 ==#2 1#2 then binomial(0.9, 18)
172     else if student_gpa_1 > 0.875 then binomial(0.6, 18)
173     else binomial(0.5, 18)
174   else if num_recruiters <= 19 then
175     if student_perfect_1 ==#2 1#2 then binomial(0.9, 19)
176     else if student_gpa_1 > 0.875 then binomial(0.6, 19)
177     else binomial(0.5, 19)
178   else if num_recruiters <= 20 then
179     if student_perfect_1 ==#2 1#2 then binomial(0.9, 20)
180     else if student_gpa_1 > 0.875 then binomial(0.6, 20)
181     else binomial(0.5, 20)
182   else if num_recruiters <= 21 then
183     if student_perfect_1 ==#2 1#2 then binomial(0.9, 21)
184     else if student_gpa_1 > 0.875 then binomial(0.6, 21)
185     else binomial(0.5, 21)
186   else if num_recruiters <= 22 then
187     if student_perfect_1 ==#2 1#2 then binomial(0.9, 22)
188     else if student_gpa_1 > 0.875 then binomial(0.6, 22)
189     else binomial(0.5, 22)
190   else if num_recruiters <= 23 then
191     if student_perfect_1 ==#2 1#2 then binomial(0.9, 23)
192     else if student_gpa_1 > 0.875 then binomial(0.6, 23)
193     else binomial(0.5, 23)
194   else if num_recruiters <= 24 then
195     if student_perfect_1 ==#2 1#2 then binomial(0.9, 24)
196     else if student_gpa_1 > 0.875 then binomial(0.6, 24)
197     else binomial(0.5, 24)
198   else if num_recruiters <= 25 then
199     if student_perfect_1 ==#2 1#2 then binomial(0.9, 25)
200     else if student_gpa_1 > 0.875 then binomial(0.6, 25)
201     else binomial(0.5, 25)
202   else if num_recruiters <= 26 then
203     if student_perfect_1 ==#2 1#2 then binomial(0.9, 26)
204     else if student_gpa_1 > 0.875 then binomial(0.6, 26)
205     else binomial(0.5, 26)
206   else if num_recruiters <= 27 then
207     if student_perfect_1 ==#2 1#2 then binomial(0.9, 27)
208     else if student_gpa_1 > 0.875 then binomial(0.6, 27)
209     else binomial(0.5, 27)
210   else if num_recruiters <= 28 then
211     if student_perfect_1 ==#2 1#2 then binomial(0.9, 28)
212     else if student_gpa_1 > 0.875 then binomial(0.6, 28)
213     else binomial(0.5, 28)

```

```

214     else if num_recruiters <= 29 then
215         if student_perfect_1 ==#2 1#2 then binomial(0.9, 29)
216         else if student_gpa_1 > 0.875 then binomial(0.6, 29)
217         else binomial(0.5, 29)
218     else if num_recruiters <= 30 then
219         if student_perfect_1 ==#2 1#2 then binomial(0.9, 30)
220         else if student_gpa_1 > 0.875 then binomial(0.6, 30)
221         else binomial(0.5, 30)
222     else if num_recruiters <= 31 then
223         if student_perfect_1 ==#2 1#2 then binomial(0.9, 31)
224         else if student_gpa_1 > 0.875 then binomial(0.6, 31)
225         else binomial(0.5, 31)
226     else if num_recruiters <= 32 then
227         if student_perfect_1 ==#2 1#2 then binomial(0.9, 32)
228         else if student_gpa_1 > 0.875 then binomial(0.6, 32)
229         else binomial(0.5, 32)
230     else if num_recruiters <= 33 then
231         if student_perfect_1 ==#2 1#2 then binomial(0.9, 33)
232         else if student_gpa_1 > 0.875 then binomial(0.6, 33)
233         else binomial(0.5, 33)
234     else if num_recruiters <= 34 then
235         if student_perfect_1 ==#2 1#2 then binomial(0.9, 34)
236         else if student_gpa_1 > 0.875 then binomial(0.6, 34)
237         else binomial(0.5, 34)
238     else if num_recruiters <= 35 then
239         if student_perfect_1 ==#2 1#2 then binomial(0.9, 35)
240         else if student_gpa_1 > 0.875 then binomial(0.6, 35)
241         else binomial(0.5, 35)
242     else if num_recruiters <= 36 then
243         if student_perfect_1 ==#2 1#2 then binomial(0.9, 36)
244         else if student_gpa_1 > 0.875 then binomial(0.6, 36)
245         else binomial(0.5, 36)
246     else if num_recruiters <= 37 then
247         if student_perfect_1 ==#2 1#2 then binomial(0.9, 37)
248         else if student_gpa_1 > 0.875 then binomial(0.6, 37)
249         else binomial(0.5, 37)
250     else if num_recruiters <= 38 then
251         if student_perfect_1 ==#2 1#2 then binomial(0.9, 38)
252         else if student_gpa_1 > 0.875 then binomial(0.6, 38)
253         else binomial(0.5, 38)
254     else if num_recruiters <= 39 then
255         if student_perfect_1 ==#2 1#2 then binomial(0.9, 39)
256         else if student_gpa_1 > 0.875 then binomial(0.6, 39)
257         else binomial(0.5, 39)
258     else
259         if student_perfect_1 ==#2 1#2 then binomial(0.9, 40)
260         else if student_gpa_1 > 0.875 then binomial(0.6, 40)
261         else binomial(0.5, 40)
262 in
263
264 let student_offers_0 =
265     if student_interviews_0 <= 0 then binomial(0.4, 0)
266     else if student_interviews_0 <= 1 then binomial(0.4, 1)
267     else if student_interviews_0 <= 2 then binomial(0.4, 2)
268     else if student_interviews_0 <= 3 then binomial(0.4, 3)
269     else if student_interviews_0 <= 4 then binomial(0.4, 4)
270     else if student_interviews_0 <= 5 then binomial(0.4, 5)
271     else if student_interviews_0 <= 6 then binomial(0.4, 6)
272     else if student_interviews_0 <= 7 then binomial(0.4, 7)
273     else if student_interviews_0 <= 8 then binomial(0.4, 8)
274     else if student_interviews_0 <= 9 then binomial(0.4, 9)
275     else if student_interviews_0 <= 10 then binomial(0.4, 10)
276     else if student_interviews_0 <= 11 then binomial(0.4, 11)
277     else if student_interviews_0 <= 12 then binomial(0.4, 12)
278     else if student_interviews_0 <= 13 then binomial(0.4, 13)
279     else if student_interviews_0 <= 14 then binomial(0.4, 14)
280     else if student_interviews_0 <= 15 then binomial(0.4, 15)
281     else if student_interviews_0 <= 16 then binomial(0.4, 16)
282     else if student_interviews_0 <= 17 then binomial(0.4, 17)
283     else if student_interviews_0 <= 18 then binomial(0.4, 18)
284     else if student_interviews_0 <= 19 then binomial(0.4, 19)

```

```

285   else if student_interviews_0 <= 20 then binomial(0.4, 20)
286   else if student_interviews_0 <= 21 then binomial(0.4, 21)
287   else if student_interviews_0 <= 22 then binomial(0.4, 22)
288   else if student_interviews_0 <= 23 then binomial(0.4, 23)
289   else if student_interviews_0 <= 24 then binomial(0.4, 24)
290   else if student_interviews_0 <= 25 then binomial(0.4, 25)
291   else if student_interviews_0 <= 26 then binomial(0.4, 26)
292   else if student_interviews_0 <= 27 then binomial(0.4, 27)
293   else if student_interviews_0 <= 28 then binomial(0.4, 28)
294   else if student_interviews_0 <= 29 then binomial(0.4, 29)
295   else if student_interviews_0 <= 30 then binomial(0.4, 30)
296   else if student_interviews_0 <= 31 then binomial(0.4, 31)
297   else if student_interviews_0 <= 32 then binomial(0.4, 32)
298   else if student_interviews_0 <= 33 then binomial(0.4, 33)
299   else if student_interviews_0 <= 34 then binomial(0.4, 34)
300   else if student_interviews_0 <= 35 then binomial(0.4, 35)
301   else if student_interviews_0 <= 36 then binomial(0.4, 36)
302   else if student_interviews_0 <= 37 then binomial(0.4, 37)
303   else if student_interviews_0 <= 38 then binomial(0.4, 38)
304   else if student_interviews_0 <= 39 then binomial(0.4, 39)
305   else binomial(0.4, 40)
306 in
307
308 let student_offers_1 =
309   if student_interviews_1 <= 0 then binomial(0.4, 0)
310   else if student_interviews_1 <= 1 then binomial(0.4, 1)
311   else if student_interviews_1 <= 2 then binomial(0.4, 2)
312   else if student_interviews_1 <= 3 then binomial(0.4, 3)
313   else if student_interviews_1 <= 4 then binomial(0.4, 4)
314   else if student_interviews_1 <= 5 then binomial(0.4, 5)
315   else if student_interviews_1 <= 6 then binomial(0.4, 6)
316   else if student_interviews_1 <= 7 then binomial(0.4, 7)
317   else if student_interviews_1 <= 8 then binomial(0.4, 8)
318   else if student_interviews_1 <= 9 then binomial(0.4, 9)
319   else if student_interviews_1 <= 10 then binomial(0.4, 10)
320   else if student_interviews_1 <= 11 then binomial(0.4, 11)
321   else if student_interviews_1 <= 12 then binomial(0.4, 12)
322   else if student_interviews_1 <= 13 then binomial(0.4, 13)
323   else if student_interviews_1 <= 14 then binomial(0.4, 14)
324   else if student_interviews_1 <= 15 then binomial(0.4, 15)
325   else if student_interviews_1 <= 16 then binomial(0.4, 16)
326   else if student_interviews_1 <= 17 then binomial(0.4, 17)
327   else if student_interviews_1 <= 18 then binomial(0.4, 18)
328   else if student_interviews_1 <= 19 then binomial(0.4, 19)
329   else if student_interviews_1 <= 20 then binomial(0.4, 20)
330   else if student_interviews_1 <= 21 then binomial(0.4, 21)
331   else if student_interviews_1 <= 22 then binomial(0.4, 22)
332   else if student_interviews_1 <= 23 then binomial(0.4, 23)
333   else if student_interviews_1 <= 24 then binomial(0.4, 24)
334   else if student_interviews_1 <= 25 then binomial(0.4, 25)
335   else if student_interviews_1 <= 26 then binomial(0.4, 26)
336   else if student_interviews_1 <= 27 then binomial(0.4, 27)
337   else if student_interviews_1 <= 28 then binomial(0.4, 28)
338   else if student_interviews_1 <= 29 then binomial(0.4, 29)
339   else if student_interviews_1 <= 30 then binomial(0.4, 30)
340   else if student_interviews_1 <= 31 then binomial(0.4, 31)
341   else if student_interviews_1 <= 32 then binomial(0.4, 32)
342   else if student_interviews_1 <= 33 then binomial(0.4, 33)
343   else if student_interviews_1 <= 34 then binomial(0.4, 34)
344   else if student_interviews_1 <= 35 then binomial(0.4, 35)
345   else if student_interviews_1 <= 36 then binomial(0.4, 36)
346   else if student_interviews_1 <= 37 then binomial(0.4, 37)
347   else if student_interviews_1 <= 38 then binomial(0.4, 38)
348   else if student_interviews_1 <= 39 then binomial(0.4, 39)
349   else binomial(0.4, 40)
350 in
351
352 let _offer = observe(1 <= student_offers_0 && student_offers_0 <= 1) in
353 let _recruiters_positive = observe(num_recruiters > 10) in
354
355 student_gpa_0 < 0.125

```

SurveyUnbias

```

1  let bias_0 = beta(1, 1) in
2  let bias_1 = beta(1, 1) in
3
4  let votes_0 =
5    if bias_0 <= 0.05 then binomial(0.05, 1000)
6    else if bias_0 <= 0.1 then binomial(0.1, 1000)
7    else if bias_0 <= 0.15 then binomial(0.15, 1000)
8    else if bias_0 <= 0.2 then binomial(0.2, 1000)
9    else if bias_0 <= 0.25 then binomial(0.25, 1000)
10   else if bias_0 <= 0.3 then binomial(0.3, 1000)
11   else if bias_0 <= 0.35 then binomial(0.35, 1000)
12   else if bias_0 <= 0.4 then binomial(0.4, 1000)
13   else if bias_0 <= 0.45 then binomial(0.45, 1000)
14   else if bias_0 <= 0.5 then binomial(0.5, 1000)
15   else if bias_0 <= 0.55 then binomial(0.55, 1000)
16   else if bias_0 <= 0.6 then binomial(0.6, 1000)
17   else if bias_0 <= 0.65 then binomial(0.65, 1000)
18   else if bias_0 <= 0.7 then binomial(0.7, 1000)
19   else if bias_0 <= 0.75 then binomial(0.75, 1000)
20   else if bias_0 <= 0.8 then binomial(0.8, 1000)
21   else if bias_0 <= 0.85 then binomial(0.85, 1000)
22   else if bias_0 <= 0.9 then binomial(0.9, 1000)
23   else if bias_0 <= 0.95 then binomial(0.95, 1000)
24   else binomial(1.0, 1000)
25  in
26
27  let votes_1 =
28    if bias_1 <= 0.05 then binomial(0.05, 2000)
29    else if bias_1 <= 0.1 then binomial(0.1, 2000)
30    else if bias_1 <= 0.15 then binomial(0.15, 2000)
31    else if bias_1 <= 0.2 then binomial(0.2, 2000)
32    else if bias_1 <= 0.25 then binomial(0.25, 2000)
33    else if bias_1 <= 0.3 then binomial(0.3, 2000)
34    else if bias_1 <= 0.35 then binomial(0.35, 2000)
35    else if bias_1 <= 0.4 then binomial(0.4, 2000)
36    else if bias_1 <= 0.45 then binomial(0.45, 2000)
37    else if bias_1 <= 0.5 then binomial(0.5, 2000)
38    else if bias_1 <= 0.55 then binomial(0.55, 2000)
39    else if bias_1 <= 0.6 then binomial(0.6, 2000)
40    else if bias_1 <= 0.65 then binomial(0.65, 2000)
41    else if bias_1 <= 0.7 then binomial(0.7, 2000)
42    else if bias_1 <= 0.75 then binomial(0.75, 2000)
43    else if bias_1 <= 0.8 then binomial(0.8, 2000)
44    else if bias_1 <= 0.85 then binomial(0.85, 2000)
45    else if bias_1 <= 0.9 then binomial(0.9, 2000)
46    else if bias_1 <= 0.95 then binomial(0.95, 2000)
47    else binomial(1.0, 2000)
48  in
49
50  let ansBias_0 = bias_0 in
51  let ansBias_1 = bias_1 in
52  let ansBias_2 = bias_0 in
53  let ansBias_3 = bias_1 in
54  let ansBias_4 = bias_0 in
55
56  let answer_0 =
57    if ansBias_0 <= 0.05 then discrete(0.95, 0.05)
58    else if ansBias_0 <= 0.1 then discrete(0.9, 0.1)
59    else if ansBias_0 <= 0.15 then discrete(0.85, 0.15)
60    else if ansBias_0 <= 0.2 then discrete(0.8, 0.2)
61    else if ansBias_0 <= 0.25 then discrete(0.75, 0.25)
62    else if ansBias_0 <= 0.3 then discrete(0.7, 0.3)
63    else if ansBias_0 <= 0.35 then discrete(0.65, 0.35)
64    else if ansBias_0 <= 0.4 then discrete(0.6, 0.4)
65    else if ansBias_0 <= 0.45 then discrete(0.55, 0.45)
66    else if ansBias_0 <= 0.5 then discrete(0.5, 0.5)
67    else if ansBias_0 <= 0.55 then discrete(0.45, 0.55)
68    else if ansBias_0 <= 0.6 then discrete(0.4, 0.6)
69    else if ansBias_0 <= 0.65 then discrete(0.35, 0.65)
70    else if ansBias_0 <= 0.7 then discrete(0.3, 0.7)

```

```

71     else if ansBias_0 <= 0.75 then discrete(0.25, 0.75)
72     else if ansBias_0 <= 0.8 then discrete(0.2, 0.8)
73     else if ansBias_0 <= 0.85 then discrete(0.15, 0.85)
74     else if ansBias_0 <= 0.9 then discrete(0.1, 0.9)
75     else if ansBias_0 <= 0.95 then discrete(0.05, 0.95)
76     else discrete(0.0, 1.0)
77 in
78
79 let answer_1 =
80     if ansBias_1 <= 0.05 then discrete(0.95, 0.05)
81     else if ansBias_1 <= 0.1 then discrete(0.9, 0.1)
82     else if ansBias_1 <= 0.15 then discrete(0.85, 0.15)
83     else if ansBias_1 <= 0.2 then discrete(0.8, 0.2)
84     else if ansBias_1 <= 0.25 then discrete(0.75, 0.25)
85     else if ansBias_1 <= 0.3 then discrete(0.7, 0.3)
86     else if ansBias_1 <= 0.35 then discrete(0.65, 0.35)
87     else if ansBias_1 <= 0.4 then discrete(0.6, 0.4)
88     else if ansBias_1 <= 0.45 then discrete(0.55, 0.45)
89     else if ansBias_1 <= 0.5 then discrete(0.5, 0.5)
90     else if ansBias_1 <= 0.55 then discrete(0.45, 0.55)
91     else if ansBias_1 <= 0.6 then discrete(0.4, 0.6)
92     else if ansBias_1 <= 0.65 then discrete(0.35, 0.65)
93     else if ansBias_1 <= 0.7 then discrete(0.3, 0.7)
94     else if ansBias_1 <= 0.75 then discrete(0.25, 0.75)
95     else if ansBias_1 <= 0.8 then discrete(0.2, 0.8)
96     else if ansBias_1 <= 0.85 then discrete(0.15, 0.85)
97     else if ansBias_1 <= 0.9 then discrete(0.1, 0.9)
98     else if ansBias_1 <= 0.95 then discrete(0.05, 0.95)
99     else discrete(0.0, 1.0)
100 in
101
102 let answer_2 =
103     if ansBias_2 <= 0.05 then discrete(0.95, 0.05)
104     else if ansBias_2 <= 0.1 then discrete(0.9, 0.1)
105     else if ansBias_2 <= 0.15 then discrete(0.85, 0.15)
106     else if ansBias_2 <= 0.2 then discrete(0.8, 0.2)
107     else if ansBias_2 <= 0.25 then discrete(0.75, 0.25)
108     else if ansBias_2 <= 0.3 then discrete(0.7, 0.3)
109     else if ansBias_2 <= 0.35 then discrete(0.65, 0.35)
110     else if ansBias_2 <= 0.4 then discrete(0.6, 0.4)
111     else if ansBias_2 <= 0.45 then discrete(0.55, 0.45)
112     else if ansBias_2 <= 0.5 then discrete(0.5, 0.5)
113     else if ansBias_2 <= 0.55 then discrete(0.45, 0.55)
114     else if ansBias_2 <= 0.6 then discrete(0.4, 0.6)
115     else if ansBias_2 <= 0.65 then discrete(0.35, 0.65)
116     else if ansBias_2 <= 0.7 then discrete(0.3, 0.7)
117     else if ansBias_2 <= 0.75 then discrete(0.25, 0.75)
118     else if ansBias_2 <= 0.8 then discrete(0.2, 0.8)
119     else if ansBias_2 <= 0.85 then discrete(0.15, 0.85)
120     else if ansBias_2 <= 0.9 then discrete(0.1, 0.9)
121     else if ansBias_2 <= 0.95 then discrete(0.05, 0.95)
122     else discrete(0.0, 1.0)
123 in
124
125 let answer_3 =
126     if ansBias_3 <= 0.05 then discrete(0.95, 0.05)
127     else if ansBias_3 <= 0.1 then discrete(0.9, 0.1)
128     else if ansBias_3 <= 0.15 then discrete(0.85, 0.15)
129     else if ansBias_3 <= 0.2 then discrete(0.8, 0.2)
130     else if ansBias_3 <= 0.25 then discrete(0.75, 0.25)
131     else if ansBias_3 <= 0.3 then discrete(0.7, 0.3)
132     else if ansBias_3 <= 0.35 then discrete(0.65, 0.35)
133     else if ansBias_3 <= 0.4 then discrete(0.6, 0.4)
134     else if ansBias_3 <= 0.45 then discrete(0.55, 0.45)
135     else if ansBias_3 <= 0.5 then discrete(0.5, 0.5)
136     else if ansBias_3 <= 0.55 then discrete(0.45, 0.55)
137     else if ansBias_3 <= 0.6 then discrete(0.4, 0.6)
138     else if ansBias_3 <= 0.65 then discrete(0.35, 0.65)
139     else if ansBias_3 <= 0.7 then discrete(0.3, 0.7)
140     else if ansBias_3 <= 0.75 then discrete(0.25, 0.75)
141     else if ansBias_3 <= 0.8 then discrete(0.2, 0.8)

```

```

142     else if ansBias_3 <= 0.85 then discrete(0.15, 0.85)
143     else if ansBias_3 <= 0.9 then discrete(0.1, 0.9)
144     else if ansBias_3 <= 0.95 then discrete(0.05, 0.95)
145     else discrete(0.0, 1.0)
146 in
147
148 let answer_4 =
149   if ansBias_4 <= 0.05 then discrete(0.95, 0.05)
150   else if ansBias_4 <= 0.1 then discrete(0.9, 0.1)
151   else if ansBias_4 <= 0.15 then discrete(0.85, 0.15)
152   else if ansBias_4 <= 0.2 then discrete(0.8, 0.2)
153   else if ansBias_4 <= 0.25 then discrete(0.75, 0.25)
154   else if ansBias_4 <= 0.3 then discrete(0.7, 0.3)
155   else if ansBias_4 <= 0.35 then discrete(0.65, 0.35)
156   else if ansBias_4 <= 0.4 then discrete(0.6, 0.4)
157   else if ansBias_4 <= 0.45 then discrete(0.55, 0.45)
158   else if ansBias_4 <= 0.5 then discrete(0.5, 0.5)
159   else if ansBias_4 <= 0.55 then discrete(0.45, 0.55)
160   else if ansBias_4 <= 0.6 then discrete(0.4, 0.6)
161   else if ansBias_4 <= 0.65 then discrete(0.35, 0.65)
162   else if ansBias_4 <= 0.7 then discrete(0.3, 0.7)
163   else if ansBias_4 <= 0.75 then discrete(0.25, 0.75)
164   else if ansBias_4 <= 0.8 then discrete(0.2, 0.8)
165   else if ansBias_4 <= 0.85 then discrete(0.15, 0.85)
166   else if ansBias_4 <= 0.9 then discrete(0.1, 0.9)
167   else if ansBias_4 <= 0.95 then discrete(0.05, 0.95)
168   else discrete(0.0, 1.0)
169 in
170
171 let _0 = observe(answer_0 ==#2 1#2) in
172 let _1 = observe(answer_1 ==#2 0#2) in
173 let _2 = observe(answer_2 ==#2 1#2) in
174 let _3 = observe(answer_3 ==#2 1#2) in
175 let _4 = observe(answer_4 ==#2 0#2) in
176
177 bias_0 < 0.5

```

TrueSkill

```

1 let perfB1 = binomial(0.9, 100) in
2 let _perfB1_observed = observe(80 <= perfB1) in
3
4 let skillA = poisson(100) in
5 let _skillA_observed = observe(77 <= skillA && skillA < 125) in
6
7 let perfA1 =
8   if skillA <= 77 then binomial(0.9, 77)
9   else if skillA <= 78 then binomial(0.9, 78)
10  else if skillA <= 79 then binomial(0.9, 79)
11  else if skillA <= 80 then binomial(0.9, 80)
12  else if skillA <= 81 then binomial(0.9, 81)
13  else if skillA <= 82 then binomial(0.9, 82)
14  else if skillA <= 83 then binomial(0.9, 83)
15  else if skillA <= 84 then binomial(0.9, 84)
16  else if skillA <= 85 then binomial(0.9, 85)
17  else if skillA <= 86 then binomial(0.9, 86)
18  else if skillA <= 87 then binomial(0.9, 87)
19  else if skillA <= 88 then binomial(0.9, 88)
20  else if skillA <= 89 then binomial(0.9, 89)
21  else if skillA <= 90 then binomial(0.9, 90)
22  else if skillA <= 91 then binomial(0.9, 91)
23  else if skillA <= 92 then binomial(0.9, 92)
24  else if skillA <= 93 then binomial(0.9, 93)
25  else if skillA <= 94 then binomial(0.9, 94)
26  else if skillA <= 95 then binomial(0.9, 95)
27  else if skillA <= 96 then binomial(0.9, 96)
28  else if skillA <= 97 then binomial(0.9, 97)
29  else if skillA <= 98 then binomial(0.9, 98)
30  else if skillA <= 99 then binomial(0.9, 99)
31  else if skillA <= 100 then binomial(0.9, 100)

```

```

32  else if skillA <= 101 then binomial(0.9, 101)
33  else if skillA <= 102 then binomial(0.9, 102)
34  else if skillA <= 103 then binomial(0.9, 103)
35  else if skillA <= 104 then binomial(0.9, 104)
36  else if skillA <= 105 then binomial(0.9, 105)
37  else if skillA <= 106 then binomial(0.9, 106)
38  else if skillA <= 107 then binomial(0.9, 107)
39  else if skillA <= 108 then binomial(0.9, 108)
40  else if skillA <= 109 then binomial(0.9, 109)
41  else if skillA <= 110 then binomial(0.9, 110)
42  else if skillA <= 111 then binomial(0.9, 111)
43  else if skillA <= 112 then binomial(0.9, 112)
44  else if skillA <= 113 then binomial(0.9, 113)
45  else if skillA <= 114 then binomial(0.9, 114)
46  else if skillA <= 115 then binomial(0.9, 115)
47  else if skillA <= 116 then binomial(0.9, 116)
48  else if skillA <= 117 then binomial(0.9, 117)
49  else if skillA <= 118 then binomial(0.9, 118)
50  else if skillA <= 119 then binomial(0.9, 119)
51  else if skillA <= 120 then binomial(0.9, 120)
52  else if skillA <= 121 then binomial(0.9, 121)
53  else if skillA <= 122 then binomial(0.9, 122)
54  else if skillA <= 123 then binomial(0.9, 123)
55  else binomial(0.9, 124)
56  in
57
58  let result =
59    if perfB1 <= 80 then
60      if perfA1 > 80 then 1.0 else 0.0
61    else if perfB1 <= 81 then
62      if perfA1 > 81 then 1.0 else 0.0
63    else if perfB1 <= 82 then
64      if perfA1 > 82 then 1.0 else 0.0
65    else if perfB1 <= 83 then
66      if perfA1 > 83 then 1.0 else 0.0
67    else if perfB1 <= 84 then
68      if perfA1 > 84 then 1.0 else 0.0
69    else if perfB1 <= 85 then
70      if perfA1 > 85 then 1.0 else 0.0
71    else if perfB1 <= 86 then
72      if perfA1 > 86 then 1.0 else 0.0
73    else if perfB1 <= 87 then
74      if perfA1 > 87 then 1.0 else 0.0
75    else if perfB1 <= 88 then
76      if perfA1 > 88 then 1.0 else 0.0
77    else if perfB1 <= 89 then
78      if perfA1 > 89 then 1.0 else 0.0
79    else if perfB1 <= 90 then
80      if perfA1 > 90 then 1.0 else 0.0
81    else if perfB1 <= 91 then
82      if perfA1 > 91 then 1.0 else 0.0
83    else if perfB1 <= 92 then
84      if perfA1 > 92 then 1.0 else 0.0
85    else if perfB1 <= 93 then
86      if perfA1 > 93 then 1.0 else 0.0
87    else if perfB1 <= 94 then
88      if perfA1 > 94 then 1.0 else 0.0
89    else if perfB1 <= 95 then
90      if perfA1 > 95 then 1.0 else 0.0
91    else if perfB1 <= 96 then
92      if perfA1 > 96 then 1.0 else 0.0
93    else if perfB1 <= 97 then
94      if perfA1 > 97 then 1.0 else 0.0
95    else if perfB1 <= 98 then
96      if perfA1 > 98 then 1.0 else 0.0
97    else if perfB1 <= 99 then
98      if perfA1 > 99 then 1.0 else 0.0
99    else
100     if perfA1 > 100 then 1.0 else 0.0
101  in
102

```

```
103 let _result_observed = observe(result <= 0 && result >= 0) in  
104 50 <= perfA1 && perfA1 <= 50
```

Received 2026-03-17; accepted 2026-06-10