Imprecise Probabilistic Programming, Precisely (Functional Pearl)

Credal Sets via Graded Monads, BDDs, and Semiring-Parametric Inference

JACK LIELL-COCK, University of Oxford, United Kingdom
SAM STATON, University of Oxford, United Kingdom

Imprecise probability generalizes standard probability theory by replacing a single distribution with a convex set of possible distributions. We show that this generalization requires no change to the standard BDD compilation and weighted model counting pipeline used by discrete probabilistic languages. An imprecise coin flip is simply a BDD variable whose weight is left free rather than fixed. We introduce Imp, a Haskell-embedded DSL for imprecise probabilistic programming. A graded monad, indexed by finite sets of named sources of epistemic uncertainty, restores the commutativity that the standard convex powerset monad lacks, and GHC's type system enforces this at compile time. Weighted model counting is parametric in the semiring, so the same compiled BDD supports exact, differentiable, and interval-bounded inference.

CCS Concepts: • **Theory of computation** → **Probabilistic computation**; • **Software and its engineering** → **Functional languages**; *Domain specific languages.*

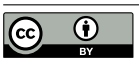
Additional Key Words and Phrases: imprecise probability, credal sets, Knightian uncertainty, graded monads, binary decision diagrams, weighted model counting, probabilistic programming, Haskell

ACM Reference Format:
Jack Liell-Cock and Sam Staton. 2026. Imprecise Probabilistic Programming, Precisely (Functional Pearl): Credal Sets via Graded Monads, BDDs, and Semiring-Parametric Inference. *Proc. ACM Program. Lang.* 10, ICFP, Article 300 (August 2026), 21 pages. https://doi.org/10.1145/3828698

1 Introduction

Ellsberg's paradox [15] reveals that people systematically prefer known risks over unknown ones: given an urn with 30 red balls and 60 that are green or blue in unknown proportion, most people prefer to bet on red. No single probability distribution can justify this preference [37]. The resolution is *imprecise probability*, where uncertainty is modelled not by one distribution but by a *credal set*, a closed, convex set of distributions [1, 37]. Any element of the credal set is considered a plausible probability of the underlying phenomenon, but no single one is privileged. This captures *Knightian uncertainty* [37]: irreducible ambiguity that cannot itself be assigned a probability. In the Ellsberg urn example, the number of red balls is known (so $P(\mathrm{R}) = 1/3$), but the split between green and blue is unknown, so $P(\mathrm{G})$ and $P(\mathrm{B})$ both range over $[0, 2/3]$. The associated credal set is shown in Figure 1. The natural quantities of interest are then *lower* and *upper* probabilities of an event, obtained by minimizing and maximizing over the credal set. Convex closure of the set guarantees that these bounds exist, and standard probability machinery (e.g. Bayes' rule and expectations) generalizes in a canonical way, called the *natural extension* [37]. Throughout this paper, we identify a credal set with its extremal vertices since the convex hull recovers the rest.

Authors' Contact Information: Jack Liell-Cock, University of Oxford, Oxford, United Kingdom, jack.liell-cock@cs.ox.ac.uk; Sam Staton, University of Oxford, Oxford, United Kingdom, sam.staton@cs.ox.ac.uk.



ACM 2475-1421/2026/8-ART300
https://doi.org/10.1145/3828698

This paper presents a Haskell DSL for imprecise probabilistic programming. Programs mix ordinary coin flips with *Knightian* coin flips, whose bias is unknown. A type-level grading of *Knightian names* tracks the sources of epistemic uncertainty and guarantees *commutativity*: independently specified uncertainties compose without interfering [25]. Programs compile to binary decision diagrams (BDDs) [6, 21], and semiring-parametrized weighted model counting (WMC) provides exact, differentiable, and interval-bounded inference [7]. In our DSL, the Ellsberg urn is:

```
data Ball = Red | Green | Blue

ellsberg :: Imp '["split"] Ball
ellsberg = Imp.do
  isRed   ← flip (1/3)
  isGreen ← knight @"split"
  Imp.return $ if isRed then Red else (if isGreen then Green else Blue)
```

In this program, `flip (1/3)` is an ordinary probabilistic coin flip, while `knight @"split"` is a Knightian coin flip. The probability of this coin flip lies somewhere in $[0, 1]$, but it is not known. The type-level string `"split"` is a *name* for this source of Knightian uncertainty, and it appears in the program's type: `Imp '["split"] Ball`. The type constructor `Imp g a` stands for "imprecise probabilistic computations of type `a` whose Knightian uncertainty is indexed by the type-level set of names `g`"; we define it formally in §3. The semantics is a graded monad [22, 32], which tracks the Knightian choices at the type level, following the trace types of Lew et al. [24]. GHC's `QualifiedDo` extension provides ergonomic do-notation despite the non-standard bind signature.

Probabilistic programming languages such as Dice [21] already compile coin flips into BDD variables and perform inference via weighted model counting (WMC). Imprecise probability requires no change to this compilation pipeline. **A Knightian coin flip is simply a BDD variable whose weight is not fixed at compilation time but left free**, ranging over $[0, 1]$ at inference time. Exact enumeration resolves the k Knightian variables by visiting their 2^k extreme weightings; each extremum is a standard WMC pass over the same compiled BDD, although gradient and interval methods (§5) avoid this enumeration.

A key subtlety is that the standard monad for sets of distributions (the convex powerset monad) is not commutative. Its bind permits the resolution of one source of Knightian uncertainty to depend on the outcome of another, so composing two sub-programs in different orders can yield different credal sets. Following [25], we use a graded monad, indexed by finite sets of Knightian names, which restores commutativity when sets of names are disjoint. Our DSL enforces this disjointness at the type level via GHC's type families, so independently specified uncertainties do not interact. Conversely, dropping names and imposing commutativity unconditionally collapses genuinely different models to the same program (§3.3).

WMC is a multilinear polynomial in the variable weights [23], so it is parametric in the semiring, and our implementation exploits this structure. Dual-number weights [2, 26] turn a single WMC pass into a gradient computation over the credal set and interval-valued weights [29] yield sound (but not necessarily tight) bounds in one pass. The result is three inference methods from a single compiled BDD, with no new compilation infrastructure.

In particular, we contribute the following:

(1) A graded monad DSL for imprecise probabilistic programming, where type-level names track Knightian uncertainty and enforce *commutativity* of composition (§3).
(2) The insight that Knightian uncertainty requires no new compilation infrastructure: both probabilistic and Knightian choices compile to BDD variables, differing only at inference time (§4).

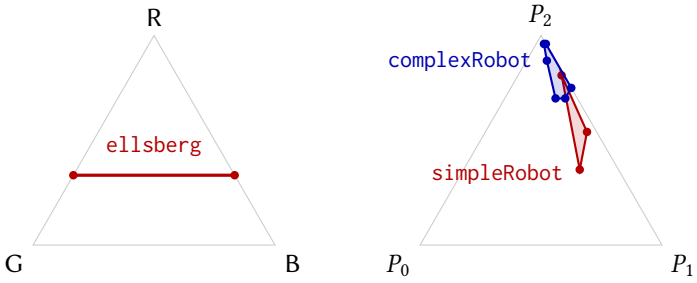


Fig. 1. Credal sets on the probability simplex. *Left:* Ellsberg's urn (`ellsberg`): a line segment at constant $P(\mathrm{R}) = 1/3$, with the green/blue split varying freely. *Right:* Resulting positions of the `simpleRobot` and `complexRobot` after two moves. While the number of credal vertices is 4 and 16, the hulls are only supported by 3 and 6 of these vertices, respectively.

(3) A Haskell implementation, inspired by [23], that obtains three inference methods by instantiating a single weighted-model-counting routine at different semirings: exact enumeration, gradient descent over the credal set (§5.1), and interval-arithmetic outer approximation (§5.2).

The theoretical foundations of compositional imprecise probability appear in [25]. This paper is about the Haskell implementation and the interplay between the graded-monad structure and BDD-based inference. The rest of the paper's layout is as follows. A walkthrough of the language is in §2. The implementation details of the DSL are in §3, and the knowledge compilation to BDDs is in §4. Inbuilt inference methods are in §5. Two further examples appear in §6, and we discuss related work and conclude in §7.

2 An Overview of the Language via Examples

We overview the features of our imprecise probabilistic programming language through an extended example that encodes interval-valued Markov decision processes into our framework, and demonstrate the alternative inference methods of §5 when enumeration over Knightian variables becomes intractable. Along the way, we preview combinators such as `interval` and `tag`, with the inference functions that the following sections define in full. Two further examples, a Monty Hall variant with unknown host bias and an imprecise two-child problem illustrating conditioning, appear in §6.

2.1 Interval-Valued MDPs

Interval-valued Markov decision processes (IMDPs) [18] model sequential decision-making under parameter uncertainty. Consider a robot that is at one of three points: P_0, P_1, or P_2. It starts at point P_0, and the goal is to get to point P_2, which is absorbing. The probability that the robot successfully moves to the successive point is imprecise and lies in the interval $[0.6, 0.9]$.

```
data Position = P0 | P1 | P2

simpleRobot :: Imp '["move1", "move2"] Position
simpleRobot = Imp.do
  move1 ← interval @"move1" 0.6 0.9
  let pos1 = step P0 move1
  move2 ← interval @"move2" 0.6 0.9
  Imp.return $ step pos1 move2
```

The function `step` advances the robot based on its move, and the command `interval` is a derived operator and uses one Knightian name to produce an imprecise coin flip between the provided bounds. The two Knightian sources dictate that there are $2^2 = 4$ valuations to compute to find the credal set of the robot's final state. The distribution for each Knightian valuation, represented as pairs of a value and probability, can be produced using the following method.

```
credalVertices :: Ord a ⇒ Imp g a → [[(a, Double)]]
```

In our example, running `credalVertices simpleRobot` yields the following output.

move1	move2	$P(P_0)$	$P(P_1)$	$P(P_2)$
0.6	0.6	0.16	0.48	0.36
0.6	0.9	0.04	0.42	0.54
0.9	0.6	0.04	0.42	0.54
0.9	0.9	0.01	0.18	0.81

Each row of the table corresponds to one valuation of the two Knightian variables at their extreme biases. The first two columns record the extreme probabilities of each successful move, and the remaining columns give the resulting probability of the robot ending in each position. The intervals are symmetric, so the valuations $\{\texttt{move1} \mapsto 0.6, \texttt{move2} \mapsto 0.9\}$ and $\{\texttt{move1} \mapsto 0.9, \texttt{move2} \mapsto 0.6\}$ produce the same distribution, leaving three distinct vertices. The credal set, shown in Figure 1, is a triangle in the simplex over P_0, P_1 and P_2, the convex hull of the three independent points.

2.1.1 Robot Navigation with Unknown Dynamics. To illustrate the framework's compositionality, we extend the example. The robot may also move backwards, and the imprecise dynamics that govern whether the robot moves forwards, remains stationary, or moves backwards are given by an external function `robotDynamics`.

```
data Move = Backwards | Stationary | Forwards

complexRobot = Imp.do
  move1 ← tag @"move1" robotDynamics
  let pos1 = step P0 move1
  move2 ← tag @"move2" robotDynamics
  Imp.return $ step pos1 move2
```

We don't specify the type of `complexRobot` because it is dependent on the type of `robotDynamics`. This is not an issue, because Haskell's type system is strong enough to infer its type once `robotDynamics` is specified. Additionally, each of the calls to the robot's dynamics is tagged with a symbol, because the names of the Knightian choices dictate whether they are correlated. The `tag` function regrades the names from the robot's dynamics for each move, so they are distinct. One possible implementation of the robot dynamics is given below, and the resulting credal set of the robot's final position is shown in Figure 1.

```
robotDynamics :: Imp '["b", "f"] Move
robotDynamics = Imp.do
  goForwards  ← interval @"f" 0.5 0.8
  goBackwards ← interval @"b" 0.0 0.2
  Imp.return $ if goForwards then Forwards
               else if goBackwards then Backwards
               else Stationary
```

The number of vertices required for inference has increased drastically. Each move introduces two Knightian choices, meaning there are $2^{2\times 2} = 16$ valuations. While 16 valuations are still straightforward to compute, we have additional inference methods that do not require enumerating over all of them, one of which we preview now.

2.1.2 Gradient Descent on the Credal Set. Now consider that the example was an abstraction of a robot crossing a road. That is, P_0 is the starting side of the road, P_1 is the middle of the road, and P_2 is the other side of the road (the goal). The robot should not only reach the goal, but also avoid ending in the middle of the road, where it could be hit by oncoming traffic. We assign a score to each final position, rewarding the robot for reaching the other side of the road, and penalizing it for stopping in the middle.

```
score :: Position → Double
score P0 = 0
score P1 = -10
score P2 = 10
```

Using *dual numbers* for the variable weights in WMC permits the same algorithm to compute forward-mode derivatives over the Knightian variables in the BDD. Therefore, we can perform gradient ascent (or descent) over the credal set to calculate the best (or worst) expected score of our imprecise probabilistic program.

The interface to the credal optimization takes in a maximum number of learning steps (it will early-abort if converged), a learning rate, a scoring function, and an `Imp` program, respectively.

```
credalOptimizeExpectation :: Ord a ⇒ Int → Double → (a → Double)
  → Imp g a → ([(String, Double)], Double)
```

The output is a list of name and probability pairs, encoding the probability of each Knightian choice that optimizes the scoring function, along with the resulting score. In the robot example, we may use this function to search for the worst possible expected score given the four Knightian inputs, `credalOptimizeExpectation 200 (-0.1) score complexRobot`, producing:

`"move1.b"`	`"move1.f"`	`"move2.b"`	`"move2.f"`	Score
0.0	1.0	0.0	0.0	4.5

At each of the robot's moves, increasing the Knightian distribution labelled by `"f"` increases the chance of the robot moving forwards, while increasing the Knightian distribution labelled by `"b"` increases the chance of the robot moving backwards. We can see from the results that the worst score happens when the robot is most likely to move forwards on the first move, and then most likely to move in neither direction on the second move. However, this still results in a positive minimum expected score of 4.5.

3 The Graded Monad DSL

The central data type underlying the DSL is `Imp`, a *generalized algebraic data type* (GADT). GADTs extend ordinary algebraic data types by letting each constructor refine the parameters in its return type, permitting the constructors below to specify different grades. `Imp` is parametrized by an output type `a`, and a type-level set of symbols `g` that represents the grade.

```
data Imp (g :: [Symbol]) a where
  ImpReturn  :: a → Imp '[] a
  ImpBind    :: Imp g1 a → (a → Imp g2 b) → Imp (Merge g1 g2) b
  ImpFlip    :: !Double → Imp '[] Bool
```

```
ImpKnight  :: KnownSymbol n ⇒ Proxy n → Imp '[n] Bool
ImpObserve :: Bool → Imp '[] ()
ImpTag     :: KnownSymbol t ⇒ Proxy t → Imp g a → Imp (TagAll t g) a
```

We represent finite sets of symbols canonically as strictly ordered type-level lists. Grades are introduced only as empty or singletons, and the type families `Merge` and `TagAll` (defined in §3.5) preserve the ordering invariant. In informal terms, each constructor has the following meaning.

- `ImpReturn` injects a value into a pure computation. Its grade is empty `'[]` because a pure computation has no Knightian uncertainty.
- `ImpBind` sequences two computations. The first has grade `g1` and the result is passed to the continuation, which produces a computation at grade `g2`. The resulting computation has grade `Merge g1 g2`, where `Merge` is a type-level function that computes the disjoint union of two sorted name lists (raising an error on overlap).
- `ImpFlip` is a probabilistic coin flip with a known bias. It takes a value in $[0, 1]$ that parametrizes the bias of the coin flip, and returns a `Bool` at grade `'[]`. A coin flip with known bias has no Knightian uncertainty.
- `ImpKnight` is the Knightian counterpart to `ImpFlip`. It represents a binary choice for which we do not know the distribution. It introduces a name `n` that tracks the Knightian input. The name is represented by the type `Proxy n` whose unique value `Proxy` carries no run-time data, but permits passing the type-level name `n` as an argument. The constraint `KnownSymbol n` says that `n` can be reflected back to a `String` at run-time to obtain the Knightian variable's name. Coin flips that share the same name share the same unknown distribution. The resulting computation only depends on a single Knightian choice and thus has singleton grade `'[n]`.
- `ImpObserve` conditions on a boolean (see §6.2), restricting to executions where the predicate holds.
- `ImpTag` renames each of the Knightian names by attaching a tag, using a second type family `TagAll t g` that prepends the symbol `t` to every name in `g`. This functionality is crucial for compositionality. It is required for multiple calls to the same external imprecise probabilistic program, as seen in the `complexRobot` example in §2.1.1.

Binary primitives. The core DSL exposes only binary primitives for probabilistic and Knightian choices. Any finite distribution can be encoded as a sequence of biased coin flips, the standard way to implement categoricals in discrete probabilistic languages such as Dice [21]. A bit-level encoding also works for Knightian choices over finite values, with the credal set recovered as the convex closure of the valuations. We could expose non-binary `flip` and `knight` primitives directly, but the mechanics of inference remain the same since any such choice reduces to binary at the BDD compilation step (§4). With dynamic variable ordering during BDD compilation, the overhead of this encoding is largely absorbed [21, 23].

3.1 Smart Constructors and Intervals

The `ImpKnight` and `ImpTag` constructors require a `Proxy`. To overcome this inconvenience, we provide smart constructors `knight` and `tag` that use a visible type application instead:

```
knight :: forall n. KnownSymbol n ⇒ Imp '[n] Bool
knight = ImpKnight (Proxy @n)

tag :: forall t g a. KnownSymbol t ⇒ Imp g a → Imp (TagAll t g) a
tag = ImpTag (Proxy @t)
```

These enable the `knight @"split"` and `tag @"move1"` patterns from the earlier examples. For the remaining constructors, such as `ImpFlip` and `ImpObserve`, we provide thin wrappers `flip` and `observe`.

Another useful derived combinator is `interval`, which models a coin whose bias is bounded:

```
interval :: forall n. KnownSymbol n ⇒ Double → Double → Imp '[n] Bool
interval lo hi = ImpBind (ImpKnight (Proxy @n)) $ \b →
  ImpFlip (if b then hi else lo)
```

The idea is that `interval @"n" lo hi` returns `True` with some probability in the range $[\texttt{lo}, \texttt{hi}]$. Its implementation makes a Knightian choice between the coin flip biases of `lo` and `hi`, which semantically is the closed interval between them. Then a probabilistic flip with the chosen bias resolves the outcome. The `simpleRobot` example in §2.1 uses `interval @"move1" 0.6 0.9` to model the robot's imprecise dynamics.

3.2 Names and Correlation

At the heart of the DSL is the role played by the names of the Knightian choices. Knightian choices with the *same* name are *correlated*. They must be resolved using the same bias. Choices with *different* names are *independent*, and there is no relation in the way they are resolved.

```
data Three = R | G | B

dependent :: Imp '["a1"] Three
dependent = Imp.do
  x ← flip 0.5
  if x then Imp.do
    y ← knight @"a1"
    Imp.return (if y then R else G)
  else Imp.do
    y ← knight @"a1"
    Imp.return (if y then R else B)
```

```
independent :: Imp '["a1", "a2"] Three
independent = Imp.do
  x ← flip 0.5
  if x then Imp.do
    y ← knight @"a1"
    Imp.return (if y then R else G)
  else Imp.do
    y ← knight @"a2"
    Imp.return (if y then R else B)
```

To illustrate, consider the above two programs adapted from [25], both returning a value in $\{\mathsf{R}, \mathsf{G}, \mathsf{B}\}$. In the first, a Knightian choice is made in each code branch, but they share the same underlying distribution labelled `"a1"`. If the probability of `True` is 1, then the result of the computation will be `R`. On the other extreme, if the probability of `True` is 0, then the result of the computation will be equal odds between `G` and `B`. The hidden distribution `"a1"` can be any convex combination of these two extremes, so the resulting credal set is the line segment between these two points in the probability simplex, as shown in Figure 2. The result cannot be equal odds between `R` and `G`, because the two Knightian choices are correlated.

The program `independent` is almost identical. The only structural difference is that the second branch uses a different underlying distribution with name `"a2"` for its Knightian choice. But the semantics differ because the Knightian choices in each branch are now resolved independently. The resolution of `"a1"` does not constrain `"a2"`, or vice versa. It is now possible for the result to be equal odds between `R` and `G`, because distribution `"a1"` can resolve to certainty of `False` while distribution `"a2"` can resolve to certainty of `True`. There are four possible resolutions of the imprecise probability in total:

$$\left\{\mathsf{R},\ \frac{1}{2}\mathsf{R}+\frac{1}{2}\mathsf{G},\ \frac{1}{2}\mathsf{R}+\frac{1}{2}\mathsf{B},\ \frac{1}{2}\mathsf{G}+\frac{1}{2}\mathsf{B}\right\}$$

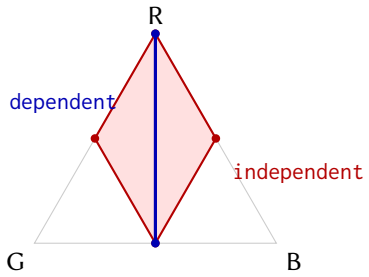


Fig. 2. Credal sets on the probability simplex over {R, G, B}. The semantics of `dependent` (one Knightian name) is a line segment. The semantics of `independent` (two Knightian names) is a quadrilateral. Increasing the number of independent Knightian choices enlarges the credal set.

The credal set is the convex hull of these four vertices, a diamond strictly larger than the line segment, as shown in Figure 2.

This distinction is clear from the type system. The grade of `dependent` is `'["a1"]`, recording one degree of Knightian uncertainty. On the other hand, the grade of `independent` is `'["a1", "a2"]`, recording two degrees of Knightian uncertainty.

When programs are composed via bind, the name sets are combined by disjoint union, reflecting the treatment of each Knightian name as a distinct source of epistemic uncertainty. If two sub-programs reused the same name, the resulting program would silently correlate them. Duplicate names on the same code path, as in the following code, are ambiguous between *reuse the same sample* and *resample with the same underlying bias*.

```
y ← knight @"a1"
z ← knight @"a1"
```

We eliminate this ambiguity by enforcing disjointness and requiring explicit renaming (e.g. with `tag`) when independence is intended.

3.3 Commutativity

The examples above show that sub-programs with disjoint name sets behave independently. The underlying property is *commutativity* [25]: when the set of names of two sub-programs is disjoint, the order of composition does not affect the credal set.

```
Imp.do                          =   Imp.do
  x ← p                               y ← q
  y ← q                               x ← p
  f x y                               f x y
```

where `p :: Imp g1 a` and `q :: Imp g2 b` have disjoint grades.

The standard monad for sets of distributions (the convex powerset monad) is not commutative. Composing unknowns in different orders can yield different credal sets, because the monad permits arbitrary correlations between them. Our graded monad avoids this by using names to separate independent sources of uncertainty. Conversely, if we dropped names and required commutativity unconditionally, `dependent` and `independent` (§3.2) would collapse to the same program. The full derivation appears in [25, Fig. 2]: commutativity and if-then-else hoisting rewrite `dependent` so that each branch of the conditional draws its own `knight`.

A recent extension to Dice, *noDice* [19], takes an alternative route. It adds an unnamed non-deterministic flip command `nflip` and provides state-of-the-art inference algorithms. However,

the lack of names constrains the command to be local, so common program equivalences, such as commutativity above, no longer hold. Our names, by contrast, reinstate these program equivalences and permit a single unified compilation scheme for a range of inference methods (§5).

3.4 Library Combinators

The core constructors of `Imp` are sufficient for expressing any imprecise probabilistic program of interest in this paper. In practice, they are tedious to use for writing large programs with many uncertainties. We therefore expose a layer of derived combinators built on top of this core to promote easy encoding of imprecise systems into our language. Each one elaborates to the `Imp` constructors at compile time with no run-time cost or semantic primitives. They let programs with many uncertainties be written concisely. We give a functional overview of the main combinators and omit their implementation.

- `intervalMap` and `knightMap` populate a type-level list of names with independent interval or Knightian choices. `intervalN` and `knightN` generate a fixed number of choices based on a type-level integer input.

```
intervalMap @'["a1", "a2"] 0.5 0.9 :: Imp '["a1", "a2"] [Bool]
```

- `tagMap` and `tagN` are numbered and iterable versions of `tag`.

```
tagN @2 @"move" robotDynamics
  :: Imp '["move1.b", "move1.f", "move2.b", "move2.f"] [Move]
```

- `foldN`, `scanN`, and `foldmN` are numbered recursive schemes, accumulating the result at each step. The monadic fold, `foldmN`, can only be over programs that do not introduce new Knightian uncertainty. They permit expressing the `complexRobot` example with a single line.

```
complexRobot = foldN @2 @"move" robotDynamics P0 step
```

3.5 Type-Level Machinery

The grading monoid operations are implemented as type families:

```
type family Merge (xs :: [Symbol]) (ys :: [Symbol]) :: [Symbol] where
  Merge '[]       ys        = ys
  Merge xs        '[]       = xs
  Merge (x ': xs) (y ': ys) = MergeH (CmpSymbol x y) x xs y ys

type family MergeH (o :: Ordering) (x :: Symbol) (xs :: [Symbol])
                   (y :: Symbol) (ys :: [Symbol]) :: [Symbol] where
  MergeH 'LT x xs y ys = x ': Merge xs (y ': ys)
  MergeH 'EQ x _  _ _  =
    TypeError ('Text "Duplicate Knightian name: " ':<>: 'ShowType x)
  MergeH 'GT x xs y ys = y ': Merge (x ': xs) ys
```

`Merge` is a type-level, order-preserving merge of two ordered lists, using `MergeH` as a helper. The disjointness check is performed here. We enforce the invariant that the grading list is sorted by only introducing empty lists or singletons in the core DSL, and only combining them with `Merge`. So `Merge` will always eventually compare and catch duplicate entries. If a programmer writes a program that binds two sub-programs both using the name `"a"`, GHC emits a helpful type error:

```
Duplicate Knightian name: "a"
```

This design enforces that grade-disjointness checking happens at compile time. The grades are erased at run-time, leaving `Imp` a free monad-like syntax tree waiting to be interpreted. The names carry no run-time cost, but they still provide a static guarantee about the compositional structure.

3.6 Graded Monad Laws and Partiality

Graded monads [22] are a generalization of monads that permits annotation of computational effects from a monoid or monoidal category. Return produces a computation at the unit annotation and bind tensors annotations together.

The type-level ordered lists of `Imp` encode the monoid of finite sets of symbols, $(P_f(\texttt{Symbol}), \cup, \emptyset)$. Semantically, `Imp g a` is the graded monad $T_g(a) = (2^g \to D(a))$ [25]; a function from valuations of the Knightian names g to distributions over a. Return is the trivial valuation, bind partitions the valuations across the two sub-computations, and tagging is a special type of regrading along the bijective function that prepends a symbol to each name.

At the implementation level, the type family `Merge` agrees with union on disjoint sets and throws a type error otherwise. This restriction deliberately prevents silent correlations between programs that overlap on Knightian names. So `Imp` satisfies the graded monad laws modulo the disjointness side condition.

An alternative design avoiding partiality would be to use a tagged disjoint union for `Merge`. However, identical names on different code paths may receive different tags, silently decoupling them and breaking the deliberate dependence mechanism in §3.2.

3.7 Qualified Do-Notation

The graded monad operations of `Imp` have the following types:

```
return :: a → Imp '[] a
return = ImpReturn

(>>=) :: Imp g1 a → (a → Imp g2 b) → Imp (Merge g1 g2) b
(>>=) = ImpBind
```

These are not the standard types of the Haskell monad class. The grades (and therefore the functors) of the two input computations to `(>>=)` differ. The `return` operation is also constrained to produce a value under the functor `Imp '[]`. GHC's `QualifiedDo` extension resolves this mismatch by permitting us to write `Imp.do` instead of `do`. This instructs GHC to desugar `<-` using the qualified name `Imp.>>=`, rather than the `Prelude` version. The result is that programs in our DSL look like ordinary Haskell, as seen in the previous examples.

GHC extensions. The library is written against `GHC2021`, which has many of the required extensions like `TypeApplications` and `TypeOperators` built in. Beyond this, the DSL requires `DataKinds` and `TypeFamilies` for the type-level name lists and `Merge` and `TagAll` families, `GADTs` for the `Imp` type, and `QualifiedDo` and `RebindableSyntax` for ergonomic do-notation. Internally, the library also uses `UndecidableInstances` (for the recursive type families) and `AllowAmbiguousTypes` (for the smart constructors of §3.1). User code requires only `DataKinds`, `QualifiedDo` and `RebindableSyntax`.

4 Compiling to BDDs

While the `Imp` DSL is convenient for producing readable imprecise probabilistic programs, it is an inefficient representation for inference. We use *knowledge compilation* [4, 13] to reduce it to

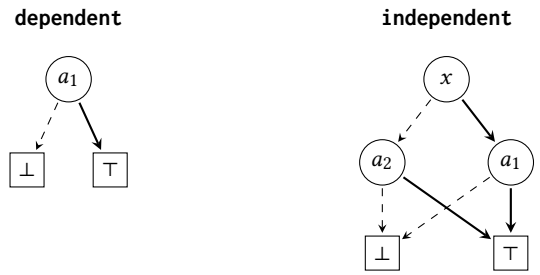


Fig. 3. BDDs for the value R compiled from dependent and independent. Solid arrows go to the variable's *high* child and dashed arrows to the *low* child. In dependent, the two paths producing R share the Knightian variable a_1, collapsing the BDD to a single variable. In independent, the two paths use distinct Knightian variables and thus remain separate.

a computable form. The simple and compositional compilation strategy is identical to that of a precise probabilistic language. An `Imp g a` program is compiled to a list of *worlds*,

```
type World a = (a, BDD)
```

that consists of a return value `a` and a binary decision diagram `BDD`. A BDD is an efficient encoding of a boolean formula, which captures the valuations of the probabilistic coin flips and Knightian choices that return the given value. The results of compiling programs dependent and independent for value R are shown in Figure 3.

The top-level compiler has the signature,

```
compile :: Ord a ⇒ Imp g a → ([World a], CompileState)
```

that returns a list of worlds and a `CompileState`. This holds the BDD manager (the hash-consing table for BDD nodes), the weight assignment for probabilistic variables, and the identities of Knightian variables:

```
data CompileState = CompileState
  { csMgr        :: !BDDManager
  , csWeights    :: !(WMCParams RealS)
  , csKnightVars :: ![(String, VarLabel)]
  , csProbVars   :: ![VarLabel]
  }
```

4.1 The BDD Representation

We use a standard BDD representation with hash-consing and complemented edges [5, 6]. A BDD is either a constant boolean or a reference to a node in the unique table or its complement:

```
data BDD
  = BDDTrue
  | BDDFalse
  | BDDRef  !NodeId
  | BDDComp !NodeId
```

Each node records a variable label and two children:

```
data BDDNode = BDDNode
  { bddVar  :: !VarLabel
```

```
  , bddLow  :: !BDD
  , bddHigh :: !BDD
  }
```

Complemented edges give us $O(1)$ negation because the function merely flips between `BDDRef` and `BDDComp`, and halves the size of the lookup table, since we normalize so that the low edge is never complemented. All BDD operations (conjunction, disjunction, if-then-else) are implemented via the standard recursive ITE algorithm with a computed-table cache [5, 6].

Our BDDs are reduced and ordered. There is a fixed total order on the variables, given by creation order during compilation, used by the ITE operator to decide which variable to branch on next. Variable ordering can affect performance [33, 34], and even correctness when non-determinism and probabilities are combined, because maxing and summing do not commute [8]. Our implementation avoids the variable-order restriction by fixing the Knightian valuations *externally*, reducing each WMC pass to a summative computation. So the choice of variable ordering only affects BDD size, not correctness, and we are free to employ reordering heuristics, such as sifting. As these are primarily performance optimizations, they are beyond the scope of this pearl.

Compilation rules. The compiler is a straightforward structural recursion on the `Imp` syntax tree, running in a state monad that owns the `CompileState`. It employs two helper functions for allocating BDD variables. `flipVar p` allocates a fresh probabilistic BDD variable, records its weight $(1-p, p)$ in `csWeights`, and returns the variable's BDD. `knightVar @n` returns the BDD for the Knightian name `n`, allocating a fresh variable the first time the name is seen and reusing the existing one thereafter. We provide the main rules to give intuition for how compilation proceeds.

`ImpReturn a`: A pure value is always available, guarded by $\top$:

```
compileM (ImpReturn a) =
  return [(a, BDDTrue)]
```

`ImpFlip p`: A probabilistic coin flip creates a fresh weighted BDD variable v and returns two worlds, one where the coin lands heads (guarded by v) and one where it lands tails (guarded by $\neg v$).

```
compileM (ImpFlip p) = do
  v ← flipVar p
  return [(True, v), (False, bddNot v)]
```

`ImpKnight (_ :: Proxy n)`: A Knightian choice compiles similarly, except `knightVar` consults `csKnightVars`. If the name `n` has been seen before, the existing variable is reused. Otherwise a fresh, unweighted BDD variable is allocated for `n`.

```
compileM (ImpKnight (_ :: Proxy n)) = do
  v ← knightVar @n
  return [(True, v), (False, bddNot v)]
```

This is the key insight of the compilation: both `flip` and `knight` produce the same kind of object, a BDD variable that splits the outcomes in two, but they differ in how they are interpreted. Variables from probabilistic flips get weights and contribute to the weighted model count, while Knightian variables are left unweighted. Their interpretation depends on what type of inference is performed later. The BDD is agnostic to the kind of uncertainty the variables represent. The same-name reuse in `knightVar` is also what causes `dependent` and `independent` to compile to structurally different BDDs (Figure 3).

`ImpBind m f`: Monadic bind compiles the computation `m` to its guarded worlds, then for each world (a, g), it compiles the continuation `f a` and conjoins the guards, $g \wedge g'$, expressing "m returned a *and* the continuation returned b".

```
compileM (ImpBind m f) = do
  mWorlds ← compileM m
  fmap concat $ mapM (\(a, g) → do
    fWorlds ← compileM (f a)
    mapM (\(b, g') → do
      gAnd ← bddAnd g g'
      return (b, gAnd)) fWorlds) mWorlds
```

`ImpObserve b`: Conditioning produces a single world guarded by the given boolean.

```
compileM (ImpObserve b) =
  return [((), if b then BDDTrue else BDDFalse)]
```

Subsequent computations are conjoined with this guard via bind. This conditions on the boolean value because any world whose guard is unsatisfiable under the observation will have probability zero. Renormalization happens at inference, as discussed further in §4.2.

4.2 Weighted Model Counting

Given a compiled BDD and a weight assignment, we compute probabilities via *weighted model counting* (WMC) [7, 10]. The WMC of a BDD is defined by recursion,

$$\mathrm{WMC}(\top) = 1, \qquad \mathrm{WMC}(\bot) = 0, \qquad \mathrm{WMC}(v \mapsto (l, h)) = w_v^{\bot} \cdot \mathrm{WMC}(l) + w_v^{\top} \cdot \mathrm{WMC}(h),$$

where $v \mapsto (l, h)$ represents an internal node with variable v, low child l and high child h, and $w_v^{\bot}$ and $w_v^{\top}$ are the weights for v being false and true, respectively. For a flip variable with parameter p, these are $1-p$ and p.

We parametrize this computation over an abstract *semiring*:

```
class Semiring a where
  zero, one :: a
  (.+.), (.*.) :: a → a → a
```

The WMC function is then polymorphic in the semiring:

```
wmc :: Semiring s ⇒ BDDManager → WMCParams s → BDD → s
```

The implementation walks the BDD top-down with memoization, handling complemented edges by negating children. It is a textbook BDD-based WMC algorithm, but the semiring abstraction allows WMC over different parameters, which comes into play when implementing automatic differentiation in §5.1 and efficient conservative inference in §5.2.

Renormalization. Operationally, `observe` introduces an *evidence guard* that assigns a weight of zero to any inconsistent world. During inference, valuations of Knightian choices are processed independently. If the WMC of the valuation is zero, it is infeasible and discarded. Otherwise, posterior quantities are normalized on a per-valuation basis. There is no *global* renormalization, only local probabilistic scaling. This implements the *natural extension* [37] of conditioning to imprecise probabilities. We work through a concrete instance of an imprecise conditioning problem in §6.2.

5 Inference on BDDs

For a purely probabilistic program with no Knightian uncertainty, computing marginals is a single WMC call per world.

```
preciseMarginal :: Ord a ⇒ Imp '[] a → [(a, Double)]
```

The graded monad gives a layer of safety here. The user cannot accidentally input a program containing Knightian uncertainty to calculate its `preciseMarginal`, because the typechecker will reject it.

For programs with Knightian variables `Imp g a`, we compute interval-valued probabilities by enumerating all $2^{|g|}$ boolean valuations of the Knightian variables and running WMC for each, taking the minimum and maximum:

```
intervalProbability :: (Ord a) ⇒ Imp g a → (a → Bool) → (Double, Double)
```

Each valuation is handled by conditioning the BDD, restricting the Knightian variables to a bias of 0 or 1, and then ordinary WMC is applied to the result. The enumeration is exponential in the number of Knightian variables, but there is no way to get around this exponential growth, because computing tight bounds on imprecise probabilities is #P-hard in general [27]. In practice, many models have few Knightian variables, each representing a dimension of epistemic uncertainty, so enumeration is feasible. However, there are cases where this is not sufficient, so we provide alternative approaches for inference when the number of Knightian variables is significant. A comparison of our inference tools is given in §5.3.

5.1 Automatic Differentiation for Credal Optimization

The semiring abstraction introduced in §4.2, following Kimmig et al. [23], also yields exact gradients. Rather than using the semiring over the real numbers, we can use *dual numbers*, which pair a value with a vector of partial derivatives.

```
data DualS = DualS !Double !(V.Vector Double)
```

Dual numbers form a semiring, allowing reuse of the WMC algorithm to obtain the exact gradients of probabilities with respect to model parameters. The semiring instance implements the standard rules of forward-mode automatic differentiation (AD). Addition propagates partials by summation, while multiplication uses the product rule, $\frac{d}{d\theta}(a \cdot b) = a \cdot \frac{db}{d\theta} + \frac{da}{d\theta} \cdot b$. Forward-mode AD is then automatically done through the inference pipeline.

```
instance Semiring DualS where
  zero = DualS 0 V.empty
  one  = DualS 1 V.empty
  DualS a da .+. DualS b db = DualS (a + b) (vzipPlus da db)
  DualS a da .*. DualS b db = DualS (a * b) (vzipPlus (vscale a db) (vscale b da))
```

Differentiating through WMC by computing in a semiring is not new: it is *algebraic model counting* [23], and the dual-number (or *gradient*) semiring was introduced for parameter learning in probabilistic logic programming [20] and later generalized to arbitrary semirings [26]. In the same functional, BDD-based setting, Pluck [4] uses a dual-number semiring over its `rsdd`-based weighted model counting [3], as we do in Haskell. The WMC implementation is polymorphic in the model weights, constrained only to the `Semiring` type class (from §4.2). Hence, using dual numbers `DualS` instead of the real numbers `RealS` is a one-line type change with no run-time dispatch, no configuration flags, and no separate code path. **The four-line `Semiring DualS` instance is the entire AD implementation.**

Formally, the reason why this is possible is that the WMC of a BDD is a *multilinear polynomial* in the variable weights. Each path from the root contributes a product of weights, and WMC sums these products. Since WMC is expressed entirely in terms of semiring operations `(.+.)` and `(.*.)` applied to the weights, the dual-number semiring computes both the polynomial's value and its gradient in a single forward pass.

As we saw from the final robot example, the complexity of finding the credal set representing an imprecise probabilistic program grows exponentially with the number of Knightian choices. Rather than enumerating all valuations of the Knightian choices, we can instead perform gradient descent to find the bounds of imprecise probabilities or expectations.

Seeding the partials. To optimize over the credal set, we seed each of the BDD nodes with a dual number. The probabilistic variables are constant, so we seed them with zero gradient. The Knightian weights are what we optimize over, so each one is initialized to 0.5, with a partial of -1 in the low weight and $+1$ in the high weight in its corresponding vector position. Running WMC with these dual-number weights then yields both the expected score and gradient with respect to the Knightian variables in a single pass.

Learning optimal scores. With dual numbers in hand, we can perform standard gradient ascent (or descent) to find the best-case (or worst-case) expectation of a scoring function. At each step, it computes the expected score and its gradient via WMC with dual numbers, then adjusts in the direction of the gradient. Since WMC is multilinear in the Knightian weights, optimizing over a single Knightian variable with all others fixed is a linear program whose optimum lies at the extremes of $[0, 1]$. For multiple independent variables, the objective is multilinear over the product of simplices. This is a non-convex problem in general, and while our optimizer is deterministic, random restarts are a standard remedy. The same machinery can handle smooth functionals of the distribution (such as variance), where optima need not lie at extremal points.

5.2 Single-Pass Conservative Inference via Interval Arithmetic

Another structure that supports semiring operations is probability intervals.

```
data IntervalS = IntervalS !Double !Double

instance Semiring IntervalS where
  zero = IntervalS 0 0
  one  = IntervalS 1 1
  IntervalS a b .+. IntervalS c d = IntervalS (a + c) (b + d)
  IntervalS a b .*. IntervalS c d = IntervalS (a * c) (b * d)
```

Interval-valued WMC provides a complementary inference method: a single-pass, linear-time sound outer approximation of the credal set bounds, trading tightness for speed. This is useful as an efficient screening before committing to the more expensive enumeration or gradient methods. The approximation is conservative but not tight, for two reasons.

Firstly, for interval values, Knightian variables v are initialized with weights $w_v^\perp = w_v^\top = [0, 1]$, encoding that both the low branch and the high branch have unknown weight in $[0, 1]$. However, when these intervals are propagated through the algorithm, the knowledge that $w_v^\perp + w_v^\top = 1$ is lost. This is the *dependency problem* of interval arithmetic [29]. The second limitation with interval-valued WMC is that each BDD guard is processed separately, so the same Knightian variable can be evaluated differently, contradicting the semantics and giving an over-approximation of bounds.

Even when the Knightian choices are not evaluated consistently, the result can still be useful. It soundly encloses the credal set's true lower and upper probabilities, generally more loosely

than exact enumeration but at a far lower cost. The implementation again reuses the semiring abstraction, exposing the following methods.

```
intervalProbabilityApprox :: Ord a ⇒ Imp g a → (a → Bool) → (Double, Double)
intervalExpectationApprox :: Ord a ⇒ Imp g a → (a → Double) → (Double, Double)
marginalApprox            :: Ord a ⇒ Imp g a → [(a, Double, Double)]
```

For example, running interval-valued WMC on the `dependent` example from §3.2 decouples the two Knightian choices labelled `"a1"`. The resulting expectation bounds are thus resolved over the larger `independent` credal set, rather than the `dependent` one (see Figure 2). This still gives the correct marginals, but if we have a scoring function, `score`, that assigns 0 to `R`, 10 to `G` and −10 to `B`, then running `intervalExpectationApprox dependent score` does not track that `G` and `B` are correlated, giving the wider bounds of $[-5, 5]$ rather than the precise $[0, 0]$.

5.3 Performance of Inference

The goal of this pearl is to demonstrate that a usable DSL can be idiomatically embedded into Haskell. Although performance is not our emphasis, the choice of inference method matters as models grow, which we illustrate on a scalable family of programs.

We generalize the Ellsberg urn of §1 to N colours: colour 0 has known probability $1/N$, and the remaining $(N-1)/N$ mass is split over colours $1, \ldots, N-1$ in unknown proportion. The split is decided by a form of *stick-breaking* (cf. [9]) over $N-2$ Knightian coins: the first coin chooses between colour 1 and the remaining mass, the second between colour 2 and the rest, and so on, with colour $N-1$ reached when no coin fires. This is the `ellsberg` program of §1 (there $N = 3$, with a single Knightian coin) extended to a chain of $N-2$ coins.

Figure 4 shows the time to bound $P(\text{colour} = N-1)$ for $N = 2, \ldots, 15$. The program compiles to a BDD of size $O(N)$. Naive exact enumeration of all 2^{N-2} Knightian valuations is exponential in N; interval WMC is a single weighted model count over the BDD and is polynomial. Gradient descent is likewise polynomial: each query runs a fixed number of ascent/descent steps, each a single weighted model count over the $O(N)$ BDD, so its cost is $O(N)$ with a larger constant than interval WMC. On this family, our deterministic optimizer stays accurate to $N = 9$, so we plot it that far.

We anticipate that state-of-the-art knowledge-compilation techniques, such as variable order heuristics [33, 34] and sentential decision diagrams [12], would lower the absolute constants further, but have no impact on the semantics or DSL interface.

6 Further Examples

We close with two further examples that revisit features introduced above. The first, a Monty Hall variant, shows a Knightian choice that turns out to be vacuous, and the second, an imprecise two-child problem, illustrates conditioning.

6.1 Monty Hall with Knightian Host Bias

In the Monty Hall problem, a contestant on a game show is standing in front of three doors. A car is behind one door, while a goat is behind each of the others. The player chooses a door, call it Door 1. The host, who knows which door the car is behind, opens a different door, revealing a goat, and offers the player the chance to switch their choice. In standard probability theory, switching to the other door wins with probability $2/3$.

A subtlety arises when the car is behind the door the player initially chose, Door 1. The host must choose which of the two remaining doors to open, because behind both are goats. In the

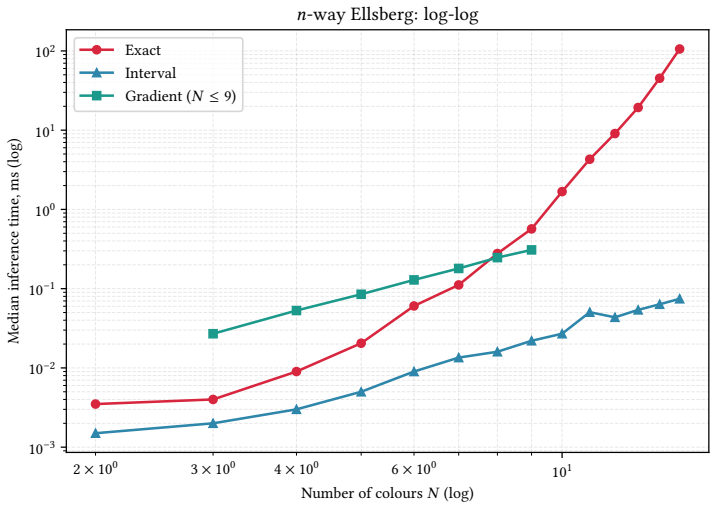


Fig. 4. Inference time vs. number of colours N on the n-way Ellsberg urn (log-log axes). Exact enumeration is exponential (2^{N-2} Knightian valuations); interval WMC is polynomial. Gradient descent is polynomial (a fixed number of WMC passes per query, hence a larger constant); our current optimizer stays accurate to $N = 9$. Medians of 10 runs, GHC 9.4.4 -O2.

original game, the choice is assumed uniform, but in reality, the host's bias is unknown. We can model it as a Knightian choice.

```
data Door = Door1 | Door2 | Door3

montyHall :: Imp '["host_bias"] Bool
montyHall = Imp.do
  c1       ← flip (1/3)
  c2       ← flip (1/2)
  hostBias ← knight @"host_bias"
  let car  = if c1 then Door1 else if c2 then Door2 else Door3
      host = case car of
               Door1 → if hostBias then Door2 else Door3
               Door2 → Door3
               Door3 → Door2
      switchTo = case host of
                   Door2 → Door3
                   Door3 → Door2
  Imp.return (switchTo == car)
```

In this example, running `intervalProbability montyHall id` yields the credal set $[2/3, 2/3]$. Despite the Knightian uncertainty in the program, we get a precise answer. Hence, we can conclude that the host bias doesn't impact the probability of the player winning if they switch. The type signature `Imp '["host_bias"] Bool` records that the program has one degree of Knightian uncertainty, but it is impotent.

6.2 The Imprecise Two-Child Problem

A common paradox in conditional probability is the *two-child problem*: a family has two children, at least one of which we know is a boy. Classic probability tells us that since the outcome of two daughters is impossible, the chance that both are boys is $1/3$.

Now consider that the family has adopted one of the children from an interstellar exchange programme. Human birth statistics are well-known, but we cannot assign a prior to the alien child's status. Knowing at least one is a boy, we may ask for the probability that both are.

```
twoChild :: Imp '["alien"] Bool
twoChild = Imp.do
  humanBoy ← flip 0.5
  alienBoy ← knight @"alien"
  observe (humanBoy || alienBoy)
  Imp.return (humanBoy && alienBoy)
```

The `observe` statement is the DSL's conditioning operator, restricting subsequent inference to executions in which the predicate holds.

Running `intervalProbability twoChild id` yields the credal set $[0, 1/2]$, which contains the classical answer. The extreme points have natural interpretations. When the alien resolves to a boy, the observation is automatically satisfied and provides no information about the human, so the probability is $1/2$. When the alien resolves to a girl, the chance that both are boys must be 0.

7 Concluding Remarks and Related Work

Probabilistic programming and knowledge compilation. Probabilistic programming has a long history in functional languages (e.g. [16]). Modern Haskell probabilistic programming languages, such as monad-bayes [36], target continuous distributions with sampling-based inference; our work instead targets discrete models with exact inference and adds imprecise probability. For discrete models, binary decision diagrams (BDDs) [6] are a well-studied compilation target for exact inference via weighted model counting (WMC) [7, 10, 11]. ProbLog [14] pioneered this approach in probabilistic logic programming, and Kimmig et al. [23] generalized WMC to an algebraic setting parametrized by a semiring, which we exploit in §5.1 and §5.2. Dice [21] compiles discrete probabilistic programs to BDDs for exact inference, using modular compilation to scale to large models. Pluck [4] extends this with lazy knowledge compilation to support higher-order functions and recursion. The underlying Rust library `rsdd` provides semiring-parametrized WMC, which Pluck instantiates with a dual-number semiring for gradients [3]. The primary difference between our system and these BDD-based languages is our treatment of *imprecise probability*: our programs contain both probabilistic and Knightian uncertainty, and inference computes interval-valued bounds rather than point probabilities. The use of standard and plain BDD and WMC machinery is key. Imprecise probability slots into the existing discrete probabilistic programming pipeline with no new compilation infrastructure. The only additions are the graded-monad layer that tracks Knightian names at the type level, and a single semiring-parametric weighted model counting that serves several inference methods.

McIver and Morgan [28] study demonic non-determinism alongside probabilistic choice in a program refinement setting. Our worst-case credal optimization (§5.1) is analogous to their demonic choice, but our motivation is modelling of epistemic uncertainty rather than program verification.

Graded monads and effect tracking. The type-level tracking that distinguishes our system draws on the graded monad for imprecise probability, $T_g(a) = (2^g \to D(a))$ [25], which assigns a distribution over a for each valuation of the finite set of Knightian names g. Graded monads were introduced

by Katsumata [22] and applied to effect tracking by Orchard, Liepelt, and Eades [30] in the Granule language; Orchard and Petricek [31] explored encoding such systems in Haskell using type-level machinery, an approach we follow here. An alternative design would use algebraic effect handlers, but the monoidal grading that tracks and merges Knightian names does not currently arise in the handler-based setting, where effects are characterized by operations and interpretations rather than by a compile-time grade. Our monad `Imp` is graded by finite sets of names with union as the monoid operation, the grade identified in [25] for modelling imprecise probability. The crucial benefit of this grading is *commutativity*: the convex powerset monad (the standard monad for sets of distributions) is not commutative, so the order of composition can affect the credal set; the graded monad of [25], studied string-diagrammatically by Sarkis and Zanasi [35], restores commutativity when name sets are disjoint, and our `Merge` type family enforces this precondition at compile time.

Lew et al. [24] introduced trace types for probabilistic programs, tracking random sampling in the type system via a graded monad in Haskell. Their work inspired our type-level tracking of the grade. However, the grades track different information: Lew et al.'s trace types record which random choices a program makes, with the key property being measurability, whereas our grades record the Knightian choices, with the key property being disjointness (ensuring that names from independent sub-computations do not collide). This paper shows that Haskell's type-level lists, GADTs, and `QualifiedDo` provide a practical encoding of the graded monad from [25].

Future directions. A fourth possible semiring instantiation is convex polyhedra over the Knightian variables, computing the exact credal set in a single pass. The trade-off is representation size: such polyhedra are multilinear polynomials whose monomial count can grow exponentially in the number of Knightian variables, and optimizing over them is NP-hard in general, though BDD structure-sharing should mitigate this in practice.

Our simple prototype uses BDDs, which suffice for the examples in this paper but are sensitive to variable ordering [6]. Sentential decision diagrams [12], as used by the `rsdd` library underlying Dice [21] and Pluck [4], would offer greater robustness and efficient bottom-up compilation. Extending the inference engine to continuous distributions would be a further challenge, although first attempts could build on other techniques (e.g. [17]) or alternative approaches from the MDP literature.

Summary. We have presented a functional pearl showing that Haskell's type system provides a natural home for imprecise probabilistic programming. Probabilistic and Knightian coin flips both compile to BDD variables. The same semiring-polymorphic WMC code serves three different inference methods, and the type-level names ensure that independently specified uncertainties cannot accidentally interfere. The result is a small library[1] (approximately 1,200 lines of Haskell) bridging the mathematical theory of imprecise probability [25] and the practical tools of discrete probabilistic programming, aimed at robust decision-making under model uncertainty.

Acknowledgments

We particularly thank Alex Lew, whose trace types [24] inspired our type-level approach, and who pointed us to the gradient support in Pluck [3] that underlies our gradient-based inference over credal sets. We also thank Todd Millstein, the ARIA Safeguarded AI community, and the Oxford group for helpful discussions. This research was supported by the Clarendon Scholarship, the ERC Consolidator Grant BLAST, AFOSR Project FA9550-21-1-0038, and the ARIA programme on Safeguarded AI.

[1] Available at: https://github.com/jacklc3/imp

References

[1] Thomas Augustin, Frank Coolen, Gert de Cooman, and Matthias Troffaes. 2014. *Introduction to Imprecise Probabilities*. Wiley. doi:10.1002/9781118763117

[2] Atılım Baydin, Barak Pearlmutter, Alexey Radul, and Jeffrey Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* (2017). doi:10.48550/arXiv.1502.05767

[3] Maddy Bowers, Alexander Lew, Joshua Tenenbaum, Armando Solar-Lezama, and Vikash Mansinghka. 2025. Pluck: Gradient Support. https://github.com/pluck-lang/Pluck.jl/tree/c5e3d3b6646d. Automatic differentiation via a dual-number semiring over rsdd-based weighted model counting; `derivatives` branch, commit `c5e3d3b`.

[4] Maddy Bowers, Alexander Lew, Joshua Tenenbaum, Armando Solar-Lezama, and Vikash Mansinghka. 2025. Stochastic Lazy Knowledge Compilation for Inference in Discrete Probabilistic Programs. In *Proc. PLDI '25*. doi:10.1145/3729325

[5] Karl Brace, Richard Rudell, and Randal Bryant. 1990. Efficient Implementation of a BDD Package. In *Proc. DAC '90*. doi:10.1145/123186.123222

[6] Randal Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* (1986). doi:10.1109/TC.1986.1676819

[7] Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *Artificial Intelligence* (2008). doi:10.1016/j.artint.2007.11.002

[8] Minsung Cho, John Gouwar, and Steven Holtzen. 2025. Scaling Optimization over Uncertainty via Compilation. In *Proc. OOPSLA '25*. doi:10.1145/3720500

[9] Robert J. Connor and James E. Mosimann. 1969. Concepts of Independence for Proportions with a Generalization of the Dirichlet Distribution. *J. Amer. Statist. Assoc.* 64, 325 (1969), 194–206. doi:10.2307/2283728

[10] Adnan Darwiche. 2002. A Logical Approach to Factoring Belief Networks. In *Proc. KR '02*.

[11] Adnan Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press. doi:10.1017/CBO9780511811357

[12] Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proc. IJCAI '11*. doi:10.5591/978-1-57735-516-8/IJCAI11-143

[13] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* (2002). doi:10.1613/jair.989

[14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proc. IJCAI '07*.

[15] Daniel Ellsberg. 1961. Risk, Ambiguity, and the Savage Axioms. *The Quarterly Journal of Economics* (1961). doi:10.2307/1884324

[16] Martin Erwig and Steve Kollmansberger. 2006. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming* (2006). doi:10.1017/S0956796805005721

[17] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2024. Bit Blasting Probabilistic Programs. In *Proc. PLDI '24*. doi:10.1145/3656412

[18] Robert Givan, Sonia Leach, and Thomas Dean. 2000. Bounded-Parameter Markov Decision Processes. *Artificial Intelligence* (2000). doi:10.1016/S0004-3702(00)00047-3

[19] Tobias Gürtler and Benjamin Lucien Kaminski. 2026. noDice: Inference for Discrete Probabilistic Programs with Nondeterminism and Conditioning. In *Proc. OOPSLA '26*. doi:10.1145/3798215

[20] Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. 2008. Parameter Learning in Probabilistic Databases: A Least Squares Approach. In *Proc. ECML PKDD '08 (LNCS, Vol. 5211)*. Springer, 473–488. doi:10.1007/978-3-540-87479-9_49

[21] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. In *Proc. OOPSLA '20*. doi:10.1145/3428208

[22] Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proc. POPL '14*. doi:10.1145/2535838.2535846

[23] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. 2017. Algebraic Model Counting. *Journal of Applied Logic* (2017). doi:10.1016/j.jal.2016.11.031

[24] Alexander Lew, Marco Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash Mansinghka. 2020. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. In *Proc. POPL '20*. doi:10.1145/3371087

[25] Jack Liell-Cock and Sam Staton. 2025. Compositional Imprecise Probability: A Solution from Graded Monads and Markov Categories. In *Proc. POPL '25*. doi:10.1145/3704890

[26] Jaron Maene and Luc De Raedt. 2025. The Gradient of Algebraic Model Counting. In *Proc. AAAI/IAAI/EAAI '25*. doi:10.1609/aaai.v39i18.34132

[27] Denis Mauá and Fabio Cozman. 2018. The Complexity of Bayesian Networks Specified by Propositional and Relational Languages. *Artificial Intelligence* (2018). doi:10.1016/j.artint.2018.06.001

[28] Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. doi:10.1007/b138392
[29] Ramon Moore. 1966. *Interval Analysis*. Prentice-Hall.
[30] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. In *Proc. ICFP '19*. doi:10.1145/3341714
[31] Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. In *Proc. Haskell Symposium '14*. doi:10.1145/2633357.2633368
[32] Dominic Orchard, Philip Wadler, and Harley Eades III. 2020. Unifying Graded and Parameterised Monads. In *Proc. MSFP '20*. doi:10.4204/EPTCS.317.2
[33] Shipra Panda and Fabio Somenzi. 1995. Who Are the Variables in Your Neighborhood. In *Proc. ICCAD '95*. doi:10.1109/ICCAD.1995.479994
[34] Richard Rudell. 1993. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proc. ICCAD '93*. doi:10.1109/ICCAD.1993.580029
[35] Ralph Sarkis and Fabio Zanasi. 2025. String Diagrams for Graded Monoidal Theories, with an Application to Imprecise Probability. In *Proc. CALCO '25*. doi:10.4230/LIPIcs.CALCO.2025.5
[36] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. In *Proc. ICFP '18*. doi:10.1145/3236778
[37] Peter Walley. 1991. *Statistical Reasoning with Imprecise Probabilities*. Chapman and Hall.

Received 2026-02-19; accepted 2026-05-13