Roulette: A Language for Expressive, Exact, and Efficient Discrete Probabilistic Programming

CAMERON MOY, Northeastern University, USA
JACK CZENSZAK, Northeastern University, USA
JOHN M. LI, Northeastern University, USA
BRIANNA MARSHALL, Northeastern University, USA
STEVEN HOLTZEN, Northeastern University, USA

Exact probabilistic inference is a requirement for many applications of probabilistic programming languages (PPLs) such as in high-consequence settings or verification. However, designing and implementing a PPL with scalable high-performance exact inference is difficult: exact inference engines, much like SAT solvers, are intricate low-level programs that are hard to implement. Due to this implementation challenge, PPLs that support scalable exact inference are restrictive and lack many features of general-purpose languages.

This paper presents Roulette, the first discrete probabilistic programming language that combines high-performance exact inference with general-purpose language features. Roulette supports a significant subset of Racket, including data structures, first-class functions, surely-terminating recursion, mutable state, modules, and macros, along with probabilistic features such as finitely supported discrete random variables, conditioning, and top-level inference. The key insight is that there is a close connection between exact probabilistic inference and the symbolic evaluation strategy of Rosette. Building on this connection, Roulette generalizes and extends the Rosette solver-aided programming system to reason about probabilistic rather than symbolic quantities. We prove Roulette sound by generalizing a proof of correctness for Rosette to handle probabilities, and demonstrate its scalability and expressivity on a number of examples.

CCS Concepts: • **Mathematics of computing → Probabilistic inference problems**.

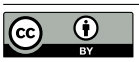
Additional Key Words and Phrases: probabilistic programming, symbolic evaluation

ACM Reference Format:
Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. 2025. Roulette: A Language for Expressive, Exact, and Efficient Discrete Probabilistic Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 231 (June 2025), 25 pages. https://doi.org/10.1145/3729334

1 Efficient Inference with Improved Expressivity

There are many applications of probabilistic programming languages (PPLs) that require or benefit from deterministic exact answers to probabilistic inference queries. In some kinds of verification, such as verified differential privacy [2, 42], it is critical that probabilistic inference be exact because even a small probability of error can be catastrophic. Even when approximations are permissible, exact inference is often useful as a subroutine for scaling approximate inference using Rao-Blackwellization [38].

Authors' Contact Information: Cameron Moy, Northeastern University, Boston, USA, camoy@ccs.neu.edu; Jack Czenszak, Northeastern University, Boston, USA, czenszak.j@northeastern.edu; John M. Li, Northeastern University, Boston, USA, li.john@northeastern.edu; Brianna Marshall, Northeastern University, Boston, USA, marshall.br@northeastern.edu; Steven Holtzen, Northeastern University, Boston, USA, s.holtzen@northeastern.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/6-ART231
https://doi.org/10.1145/3729334

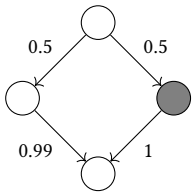
Unfortunately, implementing PPLs that support scalable high-performance exact inference is difficult today and requires great care and expertise. Inference is the task of computing the probability that a probabilistic program evaluates to a particular value, and is #P-hard even for restricted languages (e.g., straight-line programs with only `if`-statements and Boolean random variables [29, 54]). Consequently, state-of-the-art exact inference algorithms walk the fine line of computational intractability. Much like SAT solvers, they rely on careful heuristics and performance tricks to maximize performance at the edge of worst-case hardness. This makes them delicate and error-prone programs—not something one wants to implement twice.

The consequence of this difficulty is that today's languages that support exact probabilistic inference are either (1) small purpose-built languages that support only a limited subset of language features, or (2) expressive but lag behind the performance of state-of-the-art approaches that target more restricted languages. In the first category are languages such as Dice [5, 21, 29], ProbLog [17], and SPPL [56]: these languages are quite restrictive and omit most high-level features (e.g., recursion and mutable state), but in exchange can use high-performance exact inference algorithms. In the second category are languages such as PSI [23, 24] and Hakaru [39]: these languages are more expressive and support many language features (e.g., continuous random variables), but their performance lags behind the more restricted languages when they are evaluated on head-to-head benchmark problems [23, 29].

Our aim is to narrow the exact inference performance gap between expressive PPLs and restricted PPLs. Towards this goal, we present Roulette, the first discrete PPL that supports high-performance exact probabilistic inference and the essential features of a general-purpose language. Specifically, Roulette is a PPL with finitely supported discrete random variables, Bayesian conditioning, and a significant subset of the Racket programming language [18], including its data structures, standard library, and macro system. Roulette brings together two main ideas: *exact inference via knowledge compilation* [7, 14, 17, 29, 57] and *execution via symbolic evaluation* [61, 62].

Knowledge compilation (KC) is the state-of-the-art approach for exact inference of probabilistic graphical models and discrete probabilistic programs [7, 9, 29, 44, 57]. The heart of KC is a reduction from probabilistic inference to *weighted model counting* (WMC), which is a probabilistic analogue of the satisfiability problem: given a Boolean formula φ and weight map w that maps literals to real-valued weights, $\texttt{WMC}(\varphi, w)$ computes the cumulative weight of models of φ. Steady progress has been made on designing and implementing scalable WMC solvers, and today these tools are widely used within the AI and automated reasoning communities for solving discrete probabilistic inference tasks [32, 35, 45, 46].

The key challenge with applying inference-via-KC to probabilistic program inference is the laborious translation from programs to weighted Boolean formulae. Existing PPLs that leverage inference-via-KC, such as Dice [29], meticulously translate each language feature into a weighted Boolean formula and prove each translation correct. Due to the degree of manual work involved in this translation, Dice is an impoverished language. An analogous challenge of encoding programs into logical formulae is present in traditional symbolic execution, and herein lies a solution: the Rosette solver-aided programming model [61, 62]. Rosette is a programming language and symbolic evaluator within the Racket ecosystem. Rosette allows developers to write expressive high-level Racket code enhanced with primitives for introducing and constraining symbolic values. It then compiles programs into SMT constraints and dispatches them to an off-the-shelf SMT solver such as Z3 [15]. The results of queries are then made available from within the language. Rosette has proven effective in practice and has found numerous applications [10, 40, 64]: programmers can embed a language in Rosette and immediately inherit a state-of-the-art symbolic evaluator, without manually encoding each language construct as an SMT constraint.



(a) The Diamond Network Visualized

```
(define SIZE 4)

(define (diamond data)
  (if (flip 0.5) (noisy 0.01 data) (drop 0.99 data)))

(define (drop rate node)
  (if (flip rate) node #false))

(define (noisy rate data)
  (define bit (select (range SIZE)))
  (define mask (rotate-left bit (bv 1 SIZE)))
  (if (and (flip rate) data) (bvxor data mask) data))
```

(b) Definition of `diamond`

Fig. 1. Network Reliability Example

Roulette generalizes Rosette to handle finitely supported discrete random variables rather than symbolic variables, enabling high-performance inference-via-KC for a significant subset of Racket. The rest of this paper proceeds as follows. First, Section 2 gives a quick tour of Roulette and demonstrates its unique expressivity by computing network analyses, performing inference on Bayesian networks, running an interpreter for unreliable hardware, exploring laziness for geometric distributions, and embedding a relational language. Then, Section 3 gives an informal semantics of Roulette, explains how Roulette compiles programs to weighted Boolean formulae, and articulates key points that differentiate Roulette from Rosette. Sections 4 and 5 give a formal semantics for Rosette and Roulette, respectively, and prove correctness theorems for both. Section 6 empirically evaluates the performance of Roulette on a number of benchmarks, including well-known Bayesian networks and tasks that are traditionally challenging for exact inference. The benchmarks establish that, when suitably optimized, Roulette is capable of competitive performance with existing state-of-the-art exact inference strategies while being significantly more expressive. Section 7 discusses related work and Section 8 concludes.

2 Programming in Roulette

This section presents examples that demonstrate some unique features of Roulette, a subset of Racket augmented with discrete random variables, observation, and top-level inference. All the usual constructs of a traditional Scheme-like functional language, including symbols, lists, recursive functions, mutable state, standard library functions, macros, and REPL interactions, are available. Each example is intended to showcase some of these features.

2.1 Data Structures for Network Reliability

Many important problems require exact discrete probabilistic inference. One example is network reliability, which asks for the probability that a packet reaches some node given a network topology and some chance of failure along the way. PPLs are useful for modeling such tasks because one can use a probabilistic program to describe the network's topology and randomized routing protocols [22, 29, 58]. Then, once the network is described, the PPL can answer queries about the network's behavior, such as the probability that a packet successfully traverses the network with its data intact. In this setting, exact inference is critical because failure probabilities can be small and errors catastrophic [58].

Figure 1 gives an example encoding of a network reliability problem in Roulette. Figure 1a shows a visualization of a small network where nodes in the graph represent routers and edges represent

directed links between routers. Edges are annotated with the probability a packet is forwarded along that connection, and the gray router is faulty and may corrupt the payload. Figure 1b shows a program that models this network's behavior, written in Roulette. The `diamond` function, given a bitvector payload representing a packet, returns the payload when it reaches the end of the network or `#false` if the packet is dropped. The probabilistic behavior in this program is introduced via the `(flip p)` syntax, which yields a Boolean random variable that is `#true` with probability p and `#false` with probability $1 - p$. The `diamond` function forwards the packet to one of two routers with 50% probability: the behavior of these two routers is modeled by the `drop` and `noisy` functions. The `drop` function drops a packet with some probability. The `noisy` function forwards a packet and potentially introduces some error in the payload by drawing a bit index uniformly between `0` and `SIZE-1` (using `select`), masking with 1 at that index, and then computing the exclusive-or of the given payload with probability `rate`. In other words, `noisy` randomly flips a bit of the payload with probability `rate`.

Here is an example output from the program, printed using Roulette's REPL:

```
> (diamond (bv #b1111 SIZE))
(pmf | #false        : 0.00500 | (bv #b1111 4) : 0.99
     | (bv #b0111 4) : 0.00125 | (bv #b1011 4) : 0.00125
     | (bv #b1101 4) : 0.00125 | (bv #b1110 4) : 0.00125)
```

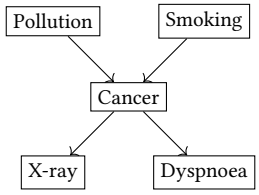
The above probability mass function (`pmf`) maps Racket values to their probabilities. The most likely returned value for the network program is `(bv #b1111 4)`, which denotes a successful traversal. Note that the program is written without contortions, using ordinary language features that would be familiar to Racket or Scheme programmers: conditionals, functions, and lists. Crucially, despite being written in a high-level style, Roulette efficiently and exactly calculates the above PMF—even for large networks. See Section 6 for details.

2.2 Recursion and Interoperability for Bayesian Networks

A Bayesian network is a probabilistic graphical model that represents a set of variables and the causal relationships between them [47]. Bayesian networks are widely used probabilistic models with well-studied inference algorithms, so they make a good baseline for measuring the performance of PPLs. Bayesian network nodes are random variables and directed edges indicate dependencies. Every node is associated with a table of probabilities enumerating the probability of each possible value, conditioned on all dependent variables. Querying a Bayesian network may, for example, amount to computing the marginal probability of some variable.

Consider the Cancer network shown graphically in Figure 2a. There are four variables shown as nodes: Pollution, Smoking, Cancer, X-ray, and Dyspnoea. Each variable has a directed edge to variables that it may influence. So, lung cancer is directly influenced by whether someone smokes and is exposed to pollution. In turn, an individual's X-ray result and the presence of dyspnoea (shortness of breath) are affected by lung cancer. Each variable is associated with a table that lists the probability of every possible value conditioned on some assignment to dependent variables. For example, there are three levels of Pollution: Low (0), Medium (1), and High (2). The table for the Cancer variable is also shown in Figure 2a, where the first row states that the probability of lung cancer is 0.03 conditional on low pollution and smoking.

Roulette can interpret Bayesian networks by generating categorical random variables for each node, conditioned on all causal dependencies. The code for doing so is provided in Figure 2b. Given an s-expression representation of a Bayesian network, the `make-network` function produces a hash table that maps variables to nodes. Each node contains a random variable associated with that



Pollution	Smoking	Cancer (True)
Low	True	0.03
Low	False	0.001
Medium	True	0.03
Medium	False	0.001
High	True	0.05
High	False	0.02

(a) Definition

```
(struct node (value size))

(define (make-network data)
  (for/fold ([acc (hash)])
            ([elem (in-list data)])
    (match-define (list name deps rows ...) elem)
    (hash-set acc name (make-node acc deps rows))))

(define (make-node acc deps rows)
  (match deps
    [(list)
     (match-define (list row) rows)
     (node (categorical row) (length row))]
    [(cons dep deps*)
     (match-define (node var size) (hash-ref acc dep))
     (define rows* (list-ref (slice-at rows size) var))
     (make-node acc deps* rows*)]))
```

(b) Builder

Fig. 2. Bayesian Network Example

node. Exactly how `make-network` works is not so important, just note that the code is short and idiomatic—using features such as recursion, hash tables, pattern matching, and for comprehensions.

The `expectation` function can be used to compute the expected value of a random variable from the Bayesian network:

```
> (define net (make-network (read-bif)))
> (expectation (node-value (hash-ref net 'Pollution)))
0.6
```

First, `net` is defined by parsing a Bayesian network in the Bayesian Interchange Format (BIF) from `STDIN`. The BIF parser is written in Racket and uses its lexing and parsing facilities—Roulette can seamlessly interoperate with Racket so long as concrete values, and not probabilistic quantities, cross the language boundary. For instance, an ordinary Racket function should not be applied to `(flip 0.5)`, but certainly can be applied to the constant value `#true`. Next, the expectation of a variable is computed: given a random variable that always takes on a numeric value, `expectation` computes its weighted average. Expectation is an example of *top-level inference*, where a program can perform inference at any point during execution and use the result in subsequent computations.

Expectation can be combined with other probabilistic operations such as conditioning:

```
> (observe! (equal? (node-value (hash-ref net 'Cancer)) 1))
> (expectation (node-value (hash-ref net 'Pollution)))
0.83233...
```

Observation lies at the heart of Bayesian reasoning. In the standard Bayesian workflow, a programmer: (1) defines a prior probability distribution, (2) conditions on some evidence, and (3) computes a posterior probability distribution. To accommodate this style of reasoning, PPLs often provide an `observe!` construct that conditions on evidence. For a discrete language, `observe!` takes a value and eliminates all worlds in which that value is `#false`. So, the above code calculates the expected level of pollution given that an individual has lung cancer. Intuitively, if a patient is diagnosed with lung cancer, then the expected amount of pollution that patient has been exposed to should increase. This intuition is borne out by the data since 0.83233 is greater than 0.6.

```
(define BIT-WIDTH 16)
(define MEMORY-SIZE 2)

(define (blank-memory)
  (list->vector
    (build-list MEMORY-SIZE
      (λ _ (bv-zero BIT-WIDTH)))))

(define (load mem a)
  (define i (bv->number a))
  (mem-perturb mem i)
  (vector-ref mem i))

(define (store mem a v)
  (define i (bv->number a))
  (mem-perturb mem i)
  (vector-set! mem i v)
  mem)

(define (mem-perturb mem i)
  (define v
    (if (flip 0.0001)
        (bv 0 BIT-WIDTH)
        (vector-ref mem i)))
  (vector-set! mem i v))
```

(a) Memory Model

```
(define ((read e) m) (load m (e m)))
(define ((int i) _) (bv i BIT-WIDTH))
(define ((add e1 e2) m) (bvadd (e1 m) (e2 m)))
(define (skip k m) m)
(define ((seq c1 c2) k m) (c2 k (c1 k m)))
(define ((asgn e1 e2) k m) (store m (e1 m) (e2 m)))
(define ((ite e c1 c2) k m)
  (if (bv-nonzero? (e m)) (c1 k m) (c2 k m)))
(define ((while e c) k m)
  (cond
    [(= k 0) #false]
    [(bv-nonzero? (e m)) ((seq c (while e c)) (- k 1) m)]
    [else m]))
```

(b) Definitional Interpreter

```
(define (tri n)
  (define total (int 0))
  (define i (int 1))
  (seq (asgn i (int n))
       (while (read i)
         (seq (asgn total (add (read total) (read i)))
              (asgn i (add (read i) (int -1)))))))

(define (read-total m)
  (if m (vector-ref m 0) #false))
```

(c) Test Program

Fig. 3. Unreliable Hardware Example

2.3 First-Class Functions and Mutation for Unreliable Hardware

This subsection demonstrates how higher-order functions and mutable state can be used in tandem to analyze the behavior of imperative programs under a low-level execution model where loads and stores from memory are unreliable. The implementation is shown in Figures 3a and 3b. Figure 3a defines the execution model as a machine with 16-bit words and two cells of addressable memory.[1] The memory is represented as a mutable vector of length `MEMORY-SIZE`, and each item in the vector is a bitvector of length `BIT-WIDTH`. The functions `load` and `store` model loads and stores from an unreliable memory. These functions are defined in terms of `vector-ref` and `vector-set!`, which read from and write to entries of a vector, and the `mem-perturb` function. Using `mem-perturb` introduces unreliability: every load and store from memory cell i has a 0.0001 chance of corrupting it, zeroing out its contents.

Figure 3b builds on this model of unreliable memory with a definitional interpreter for a simple imperative language with arithmetic, assignment statements, conditional branching, and while loops. Since Roulette supports higher-order functions and recursion, the interpreter is completely standard, following the usual semantics of WHILE programs. Arithmetic expressions are modelled as functions that take in a memory and produce a bitvector: `(read e)` loads the contents of the address denoted by `e` from memory, `(int i)` produces the two's-complement representation of the integer `i`, and `(add e1 e2)` adds the results produced by `e1` and `e2`. Commands are modelled by functions that receive a fuel parameter `k` and a memory `m`, and produce either an updated memory or the constant `#false` signalling that the interpreter ran out of fuel. Augmenting an interpreter with fuel is a standard technique for ensuring termination [43]. The functions `skip` and

[1] Two cells suffice for the example (Figure 3c); it is easy to add more.

`seq` interpret sequencing of commands, `asgn` interprets assignment, `ite` interprets branching, and `while` interprets while loops.

Finally, Figure 3c uses the interpreter from Figure 3b to define a simple WHILE program `(tri n)` that computes the sum $1 + \cdots + n$ and stores the answer at address 0. Due to the unreliability of memory, there is a chance `(tri n)` produces the wrong result. Nonetheless, since the probability of hardware failure is small, the correct result should be computed with high probability:

```
> (bv->number (read-total ((tri 10) 12 (blank-memory))))
(pmf | 55 : 0.996107 | 45 : 0.000199 | ... | #false : 0.001996)
```

Shown in the output is the distribution over possible values stored at address 0 after running the program `(tri 10)` with twelve units of fuel. The (abbreviated) probability mass function for this distribution looks as expected: the correct result (55) has the highest probability, some erroneous results (e.g., 45) caused by hardware failure have small probabilities, and some execution traces do not terminate within the amount of fuel provided. This example is challenging due to the high-dimensional state space. Section 6 shows that Roulette significantly outperforms naive exhaustive enumeration on these programs.

2.4 Macros and Laziness for Geometric Distributions

The geometric distribution models the number of failed Bernoulli trials before the first success. An example of a geometric distribution involves flipping a coin repeatedly until it lands heads up for the first time, then counting how many times the coin previously landed tails up. This experiment can be expressed as a recursive program:

```
(define (geom p)
  (if (flip p) 0 (+ 1 (geom p))))
```

Unfortunately, any program that applies `geom` will not surely terminate. As mentioned previously, Roulette works only with finitely supported discrete random variables and programs that surely terminate. At first glance, it seems geometric distributions are out of reach.

However, programmers in conventional languages can employ *laziness* to transform some non-terminating programs into terminating ones. That insight is applicable here too, by leveraging the expressiveness of Roulette. If the recursive call is placed inside a thunk, only as many levels of recursion as needed to answer *finite* queries about the distribution have to be computed.

This insight implies that the expression `(+ 1 (geom p))` cannot be eagerly evaluated. Instead, lazy Peano numbers [55] can encode partially evaluated natural numbers.[2] A program can redefine the `+` operator as a macro that thunks the second argument by using the `delay` macro from Racket's promise library and then use Peano addition:

```
(define-syntax-rule (+ n m)
  (peano-add n (delay m)))
```

The `geom` function, without modifying its source code, can now terminate under this new definition of `+` (assuming a suitable implementation of Peano numerals). Well-known properties of the geometric distribution, such as its probability mass and cumulative distribution functions, can be computed using `=` and `<=`, respectively:

```
> (= (geom 0.5) 3)
(pmf | #true : 0.0625 | #false : 0.9375)
> (<= (geom 0.2) 5)
(pmf | #true : 0.7379 | #false : 0.2621)
```

[2]Peano numerals are not an efficient representation, but there are suitable alternatives [28] that may perform better.

```
(define e1 (flip 0.6))
(define e2 (flip 0.1))
(define e3 (flip 0.4))
(define e4 (flip 0.3))

(define-relation (edgeo x y)
  (disj
    (if e1 (== (cons x y) (cons 'A 'B)) fail)
    (if e2 (== (cons x y) (cons 'A 'C)) fail)
    (if e3 (== (cons x y) (cons 'B 'D)) fail)
    (if e4 (== (cons x y) (cons 'C 'D)) fail)))
```

(a) An Example Network

```
> (define-relation (patho x z)
    (disj
      (edgeo x z)
      (fresh (y)
        (conj (edgeo x y)
              (patho y z)))))

> (run* x (patho 'A 'D))
(pmf | (list)           : 0.7372
     | (list '_.0)      : 0.2556
     | (list '_.0 '_.0) : 0.0072)
```

(b) Definition and Example of `patho`

Fig. 4. μKanren Example

The sum of geometric distributions has a closed form, but the product of geometric distributions does not. Exact inference is often more challenging in situations where there is no closed form solution. Since Roulette's inference strategy does not rely on the existence of closed form solutions, it can handle products just fine:

```
> (<= (* (geom 0.5) (geom 0.2)) 8)
(pmf | #true : 0.8572 | #false : 0.1428)
```

In short, non-probabilistic expressiveness can increase probabilistic expressiveness by allowing programmers to describe non-finitely supported random variables via laziness.

2.5 Embedded Relational Programming

As shown in Section 2.3, Roulette is expressive enough to embed interpreters for other languages. Going one step further, Roulette can be turned into a *probabilistic logic programming language* [16, 20, 41] by embedding a traditional logic programming language implementation. Probabilistic logic programming mixes traditional logic programming and probabilistic programming by making rules or atoms uncertain. Querying in this setting involves not only determining whether a goal unifies but also the probability it does so. Traditionally, implementing probabilistic logic programming languages has involved custom solutions for translating programs into a logical formula for inference: for instance, a state-of-the-art implementation of ProbLog compiles programs into weighted Boolean formulae for inference [17]. In Roulette, it is easy to get a probabilistic logic programming language by *embedding* a non-probabilistic interpreter for a relational language.

A good candidate for such an exercise is μKanren [26, 27]. The Scheme community has for many years developed relational programming languages, the most well known being miniKanren [4, 19]. Relational programming differs from traditional logic programming in a few ways. For instance, the interleaving search strategy of miniKanren differs from Prolog's depth-first search. Another difference, and the one most relevant here, is that miniKanren and its various flavors are often embedded—frequently in Scheme. In particular, one small relational language called μKanren falls neatly within the features Roulette supports. Therefore, the implementation of μKanren can quite literally be copy-and-pasted into a Roulette program and run *unmodified*.

Consider a network reliability problem, similar to the one from Section 2.1 but without noisy packets. Determining whether a packet successfully traverses a network requires encoding the network as a relation on edges. Figure 4a defines a relation `edgeo` that succeeds if the given nodes have an edge between them. Given two inputs, `edgeo` unifies if any of the goals inside the disjunction operation `disj` unify. Each goal inside `disj` corresponds to an edge and unifies, via `==`, with the given edge probability.

```
(define (search xs v)
  (cond
    [(empty? xs) (list)]
    [(equal? (first xs) v) xs]
    [else (search (rest xs) v)]))
```

(a) Definition

```
> (search 3 (list 2 3 4 3))
(list 3 4 3)

> (search 1 (list 2 3 4 3))
(list)
```

(b) Example

Fig. 5. The `search` Function

Here is an example μKanren query evaluated using Roulette:

```
> (run* x (edgeo 'A x))
(pmf | (list) : 0.36    | (list 'B) : 0.54
     | (list 'C) : 0.04 | (list 'B 'C) : 0.06)
```

The `run*` form returns all possible substitutions for `x` that unify `(edgeo 'A x)`. Running in Roulette yields a distribution over substitutions. Note how this example freely mixes features of Roulette with μKanren: first, it uses Roulette's built-in `flip` construct to introduce random variables. Then, it uses μKanren's built-in `define-relation` construct to relate these random variables and `run*` to query them as normal. From the perspective of the embedded μKanren implementation, it is as if the edge variables are regular Racket Booleans.

Continuing with the example in Figure 4b, reachability is defined to be the transitive closure of `edgeo`, and then `run*` is used to determine the probability that two nodes are reachable. Here, `(patho x z)` unifies if either there is an edge between `x` and `z` or there is some intermediate node `y` connecting `x` to `z`. Defining this second goal involves two μKanren forms: `fresh`, which generates a new logic variable, and `conj`, which unifies if all its goals do too. The results contain `'_.0` because the logic variable `x` remains unconstrained after the goal unifies. There are three substitutions in the final distribution: the empty list (unreachable), the singleton list (only one path from A to D), and the two-element list (two paths from A to D). So, node A reaches node D with probability $0.2556 + 0.0072 = 0.2628$.

This example is *not* intended to demonstrate that embedding μKanren yields a state-of-the-art probabilistic logic programming language. The μKanren implementation is fewer than 40 lines of code and is intended for pedagogic purposes. There should be no expectation that this implementation competes with a specialized compiler. However, it does showcase Roulette's expressivity and suggests further experimentation with high-performance miniKanren compilers [1].

3 How Roulette Performs Scalable Inference, by Example

Roulette derives its expressivity from Rosette's symbolic semantics and its performance from a weighted model counting backend based on binary decision diagrams (BDDs), similar to Dice [29]. This section informally summarizes (1) how Rosette symbolically evaluates programs into formulae, and then (2) how Roulette adapts Rosette to generate weighted Boolean formulae for inference.

3.1 How Rosette Efficiently Symbolically Evaluates Racket Programs

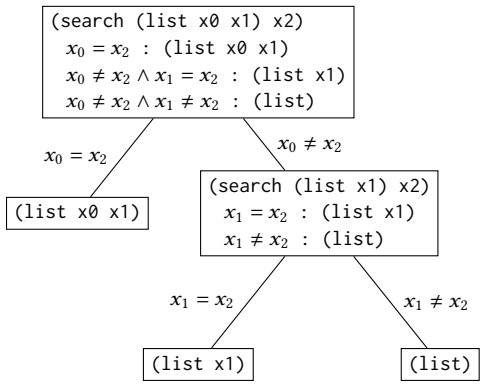
Consider the Racket definition of `search` in Figure 5a. Given a list and an element to look for, `search` returns either a suffix of the input list starting with the desired element or an empty list if the element is not present. An example REPL interaction is given in Figure 5b. Rosette can symbolically evaluate this program to check that it satisfies various correctness properties: for instance, Rosette can verify that for an input list of length n, `search` always returns another list of length at most n.

```
> (define-symbolic x0 integer?)
> (define-symbolic x1 integer?)
> (define-symbolic x2 integer?)
> (define result (search (list x0 x1) x2))
(union
  [x0 = x2 (list x0 x1)]
  [x0 ≠ x2 ∧ x1 = x2 (list x1)]
  [x0 ≠ x2 ∧ x1 ≠ x2 (list)])

> (verify (assert (<= (length result) 2)))
(unsat)

> (verify (assert (< (length result) 2)))
(model [x0 0] [x1 1] [x2 0])
```

(a) REPL Example



(b) Visualizing the Symbolic Union

Fig. 6. Verification of `search` in Rosette

Figure 6a shows how Rosette verifies this fact for lists of length 2. First, three fresh symbolic integers are created using the Rosette form `define-symbolic`. Next, `search` is called using these symbolic integers as arguments. The value returned by `search`, stored in `result`, is a *symbolic union* that represents all possible behaviors of `search` on the given inputs. Symbolic unions are sequences of formula–value pairs representing possible values and the conditions under which each value is produced. The `result` symbolic union in this example is a list where each element captures one of the three possible symbolic return values of the `search` program. For example, for the first element, if the guard $x_0 = x_2$ is true, then `search` evaluates to `(list x0 x1)`.

One can now check that `search` does not return a list that is longer than its input list by calling `verify` with the assertion that the length of the result is less than or equal to 2. Rosette outputs `(unsat)`, meaning that no counterexample to this assertion was found. If <= is changed to < instead, then the program outputs a model describing instantiations of the symbolic variables that produce a counterexample.

Rosette is designed to efficiently generate symbolic unions without suffering from path explosion. This process is visualized in Figure 6b, which gives an abstract view of how the symbolic union placed in `result` is calculated. If the guard of a `cond` depends on a symbolic value, Rosette executes both branches. For example, in `(search (list x0 x1) x2)` the first such guard is `(equal? (first xs) v)` which can be either true or false depending on the instantiation of the symbolic values x_0 and x_2. Thus, both branches are executed, and the results are *merged*. Symbolic unions are merged by conjoining the guard formula to each component of the union. Merging avoids the path explosion that is common in traditional symbolic execution. By exploiting the logical structure present in the symbolic unions, Rosette can scale to programs with many paths. Often, the bottleneck is the SMT solver rather than Rosette itself [10, 61]. For this example, the formula $x_0 = x_2$ is conjoined in the success branch, and $x_0 \neq x_2$ is conjoined in the failure branch.

Rosette's symbolic evaluation strategy is able to directly reuse host language functions, such as `length`, by lifting functions to work over symbolic unions. Roughly speaking, lifting involves mapping the operation over all elements of the union. Only primitive operations on symbolic values, such as `equal?`, must be modified to encode formulae. As such, most language features from the host, i.e. Racket, are automatically available.

3.2 How Roulette Generalizes Rosette to Probabilities

Getting Rosette to perform probabilistic inference requires two additional steps: (1) convert formulae to a representation that is appropriate for an external inference engine based on weighted model counting and (2) query the engine for probabilities. Roulette extends Rosette with an `infer` function that takes a symbolic value and returns a PMF mapping concrete values to probabilities. Rosette's existing compilation pipeline can otherwise stay exactly the same. Hence, Roulette benefits from the significant engineering effort put into Rosette's formula simplification and related optimizations.

The following is an example Roulette program that looks quite similar to the Rosette example in Figure 6a, but makes use of probabilistic rather than symbolic quantities:

```
> (define-symbolic x0 (bern 0.5))
> (define-symbolic x1 (bern 0.5))
> (define result (+ (if x0 2 3) (if x1 3 4)))
(union [x0 ∧ x1 5] [(¬x0 ∧ x1) ∨ (x0 ∧ ¬x1) 6] [¬x0 ∧ ¬x1 7])

> (infer result)
(pmf | 5 : 0.25 | 6 : 0.50 | 7 : 0.25)
```

First, two symbolic Booleans are created and associated with a probability distribution. Here, `x0` and `x1` are fair and independent coin flips. Next, the integer `result` is formed depending on the outcome of these two coin flips. Notice that the `result` symbolic union is exactly the same as it would be in Rosette if `x0` and `x1` were ordinary symbolic Booleans. Finally, `infer` is given `result`, and the PMF is displayed in tabular format. The examples in Section 2 work with more high-level language constructs than `define-symbolic`; these are desugared away using Racket macros, described further in Appendix A. To evaluate `infer`, Roulette computes the probability that the guard of each symbolic union is true. For example, the probability that `result` evaluates to 5 is given by the probability that $x_0 \wedge x_1$ is true, where x_0 and x_1 are independent Bernoulli random variables that are each true with probability 0.5. The correct answer is 0.25, which is indeed the probability printed in the REPL that corresponds to this outcome.

Roulette performs weighted model counting to calculate probabilities. So, to compute the probability that `result` evaluates to 6 in the above example, one must compute the probability that the symbolic union guard $\varphi = (\neg x_0 \wedge x_1) \vee (x_0 \wedge \neg x_1)$ is true. Given a weight map $w(x_0) = w(\neg x_0) = w(x_1) = w(\neg x_1)$, the following holds:

$$\Pr(\varphi) = \mathtt{WMC}(\varphi, w) = w(\neg x_0)w(x_1) + w(x_0)w(\neg x_1) = 0.5,$$

which matches the probability in the above PMF. Efficiently computing this weighted model count relies on knowledge compilation: these Boolean formulae are compiled into binary decision diagrams, which admit a straightforward linear-time weighted model counting procedure using dynamic programming. Figure 7 shows the BDD that represents the formula φ. Each node is annotated with a Boolean variable. Solid edges denote true assignments, and dashed edges denote false assignments. The `WMC` is computed using a single bottom-up pass where each node is annotated with its weighted model count. The `WMC` at each node is computed as the weighted sum of the `WMC` of its children. For instance, the `WMC` at node x_0 is computed as $0.5 \times 0.5 + 0.5 \times 0.5$. It is well known that BDDs exploit program structure and, in practice, can scale to large probabilistic inference tasks [17, 29].

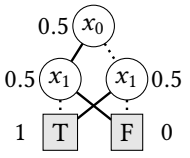


Fig. 7. Example BDD

4 Rosette: Formal Semantics

Section 5 gives a proof that Roulette's probabilistic adaptation of Rosette's symbolic evaluation strategy correctly implements probabilistic inference. Stating this correctness property first requires a formal model of Rosette-style symbolic evaluation to build upon. However, no existing formalism of Rosette contains a description of its symbolic evaluation strategy that is complete enough to serve as a foundation for Roulette's correctness theorem. Torlak and Bodík [62] give only sketches of an operational semantics and correctness proof. Porncharoenwase et al. [52] give a mechanized semantics, but it does not account for programs that dynamically generate symbolic variables or mutable references.

This section presents a model of Rosette that differs from prior work in a few ways. On the one hand, it supports dynamically generated symbolic values and mutable references, and the main theorem establishes correctness of Rosette-style symbolic evaluation in the presence of these features. On the other hand, it has a coarse-grained model of symbolic values and state-merging, and does not account for other features of Rosette such as `assume` statements and queries to the solver. This tradeoff is sensible given that the precise implementation details of symbolic state-merging and other solver-aided features of Rosette are not as relevant for Roulette, while dynamically generated symbolic values and mutable references are crucial for explaining the correctness of Roulette on the examples in Section 2.

The structure of this section parallels prior formalisms of Rosette. Section 4.1 presents a Scheme-like λ-calculus of idealized Rosette programs. We then give two operational semantics for it: a *concrete* semantics that defines standard call-by-value evaluation (Section 4.2) and an *abstract* semantics that defines the behavior of the symbolic evaluator (Section 4.3). Correctness of symbolic evaluation boils down to a *soundness and completeness* theorem linking these two semantics (Theorem 4.1). Soundness states that every concrete outcome of a given program is covered by its abstract semantics, and completeness states that every possible outcome covered by the abstract semantics of a program is reachable by concrete execution.

4.1 Syntax of Idealized Rosette

Figure 8 contains the syntax of idealized Rosette. On the left is the surface syntax, i.e., the syntax a programmer would write down. On the right is the evaluation syntax, i.e., the syntax needed solely to define an evaluation function. Idealized Rosette supports many standard features from functional languages: λ-expressions, conditionals, ML-style mutable references, pairs, arithmetic operations, and errors (`fail`).

There is one feature unique to symbolic evaluation: `sym`, a language form that models Rosette's `define-symbolic*`. This language form is treated differently in the concrete and abstract semantics. In the concrete semantics, programs are run with an infinite stream of Booleans that supplies a concrete Boolean in place of each `sym`. This is similar to the semantics of `random` for a pseudorandom number generator, where the return value comes from some infinite sequence of random numbers determined by a seed. In the abstract semantics, programs are not run with a stream but are instead *lifted* to operate on *symbolic values*, and `sym` generates a fresh symbolic value of Boolean type.

4.2 Concrete Semantics of Idealized Rosette

Figure 9 displays the environment-based natural semantics [31] for concrete execution. In addition to the usual environment ρ and store σ, the evaluation judgment also includes a stream s. The first element of s is used to evaluate `sym` expressions via the rules SymTrue and SymFalse. The remaining rules define a standard call-by-value semantics that additionally threads the stream throughout. Selected rules are shown in Figure 9. When encountering a syntactic form with two subexpressions

Surface Syntax

$$
\begin{aligned}
e \in \text{Expr} &::= x \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \lambda x.e \mid x\ y \\
&\mid b \mid \texttt{if}\ x\ e\ e \mid r \mid x + y \mid x - y \mid x \times y \\
&\mid () \mid (x, y) \mid \texttt{fst}\, x \mid \texttt{snd}\, x \\
&\mid \texttt{ref}\, x \mid \texttt{!}\, y \mid x := y \mid \texttt{sym} \mid \texttt{fail} \\
b \in \text{Bool} &::= \texttt{true} \mid \texttt{false} \\
r \in \text{Num} &::= \mathbb{Q} \\
x, y \in \text{Var}
\end{aligned}
$$

Evaluation Syntax

$$
\begin{aligned}
v, w \in \text{Val} &::= b \mid r \mid () \mid (v, w) \mid c \mid \ell \\
c \in \text{Closure} &::= \texttt{clo}(\lambda x.e, \rho) \\
\rho \in \text{Env} &::= \text{Var} \rightharpoonup_{\text{fin}} \text{Val} \\
\sigma \in \text{Store} &::= \text{Loc} \rightharpoonup_{\text{fin}} \text{Val} \\
s \in \text{Stream} &::= \mathbb{N} \to \mathbb{B} \quad \text{where } \mathbb{B} = \{\mathsf{T}, \mathsf{F}\} \\
\ell \in \text{Loc}
\end{aligned}
$$

Fig. 8. Syntax of Idealized Rosette

$\boxed{\rho, s \vdash (e, \sigma) \Downarrow (v, \sigma')}$

$$\textsc{Let}\quad \frac{s \rightsquigarrow s_1, s_2 \qquad \rho, s_1 \vdash (e_1, \sigma) \Downarrow (v_1, \sigma_1) \qquad \rho[x \mapsto v_1], s_2 \vdash (e_2, \sigma_1) \Downarrow (v_2, \sigma_2)}{\rho, s \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \sigma) \Downarrow (v_2, \sigma_2)}$$

$$\textsc{App}\quad \frac{\rho(x) = \texttt{clo}(\lambda x'.\, e', \rho') \qquad \rho'[x' \mapsto \rho(y)], s \vdash (e', \sigma) \Downarrow (v, \sigma')}{\rho, s \vdash (x\ y, \sigma) \Downarrow (v, \sigma')}$$

$$\textsc{Ref}\quad \frac{\ell \notin \text{locs}(\rho, \sigma)}{\rho, s \vdash (\texttt{ref}\, x, \sigma) \Downarrow (\ell, \sigma[\ell \mapsto \rho(x)])}$$

$$\textsc{Set}\quad \frac{\rho(x) \in \text{dom}(\sigma)}{\rho, s \vdash (x := y, \sigma) \Downarrow ((), \sigma[\rho(x) \mapsto \rho(y)])}$$

$$\textsc{Get}\quad \frac{\rho(x) \in \text{dom}(\sigma)}{\rho, s \vdash (!x, \sigma) \Downarrow (\sigma(\rho(x)), \sigma)} \qquad \textsc{SymTrue}\quad \frac{}{\rho, \mathsf{T} :: s \vdash (\texttt{sym}, \sigma) \Downarrow (\texttt{true}, \sigma)} \qquad \textsc{SymFalse}\quad \frac{}{\rho, \mathsf{F} :: s \vdash (\texttt{sym}, \sigma) \Downarrow (\texttt{false}, \sigma)}$$

Fig. 9. Concrete Semantics of Idealized Rosette (Selected Rules)

(e.g., `let`), the stream is split into two independent pieces via the judgment $s \rightsquigarrow s_1, s_2$, which holds if and only if s_1 contains the even-indexed elements of s and s_2 contains the odd-indexed elements. Note that there is no rule for `fail`, so all traces that encounter `fail` are pruned from the concrete execution. The remaining rules are standard; for details, see Appendix B.

4.3 Abstract Semantics of Idealized Rosette

Defining the abstract semantics of Rosette involves lifting all the evaluation syntax to its symbolic counterpart. For example, values v are lifted to *symbolic values* $\widehat{v}$. Specifying this lifting requires notions of symbolic variables, symbolic propositions, and symbolic unions.

A *symbolic variable* is a Boolean variable that gives meaning to symbolic values generated by `sym`. Unlike ordinary Booleans, a symbolic variable can stand in for either T or F. Assuming the existence of an infinite supply of symbolic variables $\alpha, \beta, \cdots \in \text{SymVar}$, a *model* is an assignment of Boolean values to each symbolic variable. Let $\text{Model} = \text{SymVar} \to \mathbb{B}$ be the set of models. For any set of symbolic variables S, let $\text{Model}_S = S \to \mathbb{B}$ be the set of models over the variables in S. Symbolic execution allocates a finite set of symbolic variables S and produces a *symbolic value*, which is a function $\text{Model}_S \to \text{Val}$. More generally, the *symbolic elements of type* A, written $\widehat{A}$, is the set of functions $\text{Model} \to A_\perp$ where $A_\perp = A \uplus \{\perp\}$. For example, symbolic environments are written $\widehat{\rho} \in \widehat{\text{Env}}$. The *support* of a symbolic element $\widehat{a} \in \widehat{A}$, denoted $\text{symvars}(\widehat{a})$, is the smallest subset S of SymVar such that $\widehat{a}$ restricted to the domain $\text{Model}_{\text{symvars}(\widehat{a})}$ is equivalent to $\widehat{a}$.

$\boxed{\widehat{\rho} \vdash (e, \widehat{\sigma}) \Downarrow (\widehat{v}, \widehat{\sigma}', \psi)}$

Let
$$\frac{\widehat{\rho} \vdash (e_1, \widehat{\sigma}) \Downarrow (\widehat{v}_1, \widehat{\sigma}_1, \psi_1) \quad \widehat{\rho}[x \mapsto \widehat{v}_1] \vdash (e_2, \widehat{\sigma}_1) \Downarrow (\widehat{v}_2, \widehat{\sigma}_2, \psi_2)}{\widehat{\rho} \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \widehat{\sigma}) \Downarrow ([\psi_1 : \widehat{v}_2], [\psi_1 : \widehat{\sigma}_2], \psi_1 \wedge \psi_2)}$$

App
$$\frac{\widehat{\rho}(x) = [\varphi_i : \texttt{clo}(\lambda x_i.e_i, \widehat{\rho}_i)]_{i \in I} \uplus_{\text{Closure}} \widehat{v} \quad \forall i \in I.\ \widehat{\rho}_i[x_i \mapsto \widehat{\rho}(y)] \vdash (e_i, \widehat{\sigma}) \Downarrow (\widehat{v}_i, \widehat{\sigma}_i, \psi_i)}{\widehat{\rho} \vdash (x\ y, \widehat{\sigma}) \Downarrow ([\varphi_i : \widehat{v}_i]_{i \in I}, [\varphi_i : \widehat{\sigma}_i]_{i \in I}, \bigvee_i (\varphi_i \wedge \psi_i))}$$

If
$$\frac{\widehat{\rho}(x) = [\varphi_1 : \texttt{true}, \varphi_2 : \texttt{false}] \uplus_{\text{Bool}} \widehat{v} \quad \widehat{\rho} \vdash (e_1, \widehat{\sigma}) \Downarrow (\widehat{v}_1, \widehat{\sigma}_1, \psi_1) \quad \widehat{\rho} \vdash (e_2, \widehat{\sigma}) \Downarrow (\widehat{v}_2, \widehat{\sigma}_2, \psi_2)}{\widehat{\rho} \vdash (\texttt{if}\ x\ e_1\ e_2, \widehat{\sigma}) \Downarrow ([\varphi_1 : \widehat{v}_1, \varphi_2 : \widehat{v}_2], [\varphi_1 : \widehat{\sigma}_1, \varphi_2 : \widehat{\sigma}_2], (\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2))}$$

Fail
$$\frac{}{\widehat{\rho} \vdash (\texttt{fail}, \widehat{\sigma}) \Downarrow (\varnothing, \varnothing, \mathsf{F})}$$

Ref
$$\frac{\ell \text{ smallest not in } \text{locs}(\widehat{\rho}, \widehat{\sigma})}{\widehat{\rho} \vdash (\texttt{ref}\ x, \widehat{\sigma}) \Downarrow ([\mathsf{T} : \ell], \widehat{\sigma}[\ell \mapsto \widehat{\rho}(x)], \mathsf{T})}$$

Get
$$\frac{\widehat{\rho}(x) = [\varphi_i : \ell_i]_{i \in I} \uplus_{\text{Loc}} \widehat{v}}{\widehat{\rho} \vdash (!x, \widehat{\sigma}) \Downarrow ([\varphi_i : \widehat{\sigma}(\ell_i)]_{i \in I}, \widehat{\sigma}, \bigvee_i \varphi_i)}$$

Set
$$\frac{\widehat{\rho}(x) = [\varphi_i : \ell_i]_{i \in I} \uplus_{\text{Loc}} \widehat{v}}{\widehat{\rho} \vdash (x := y, \widehat{\sigma}) \Downarrow ([(\bigvee_i \varphi_i) : ()], [\varphi_i : \widehat{\sigma}[\ell_i \mapsto \widehat{\rho}(y)]]_{i \in I}, \bigvee_i \varphi_i)}$$

Sym
$$\frac{\alpha \text{ smallest not in } \text{symvars}(\widehat{\rho}, \widehat{\sigma})}{\widehat{\rho} \vdash (\texttt{sym}, \widehat{\sigma}) \Downarrow ([\alpha : \texttt{true}, \neg\alpha : \texttt{false}], \widehat{\sigma}, \mathsf{T})}$$

Fig. 10. Abstract Semantics of Idealized Rosette (Selected Rules)

A *symbolic proposition* is an element of the set $\varphi, \psi \in \text{Model} \to \mathbb{B}$. Two symbolic propositions φ, ψ are *disjoint* if and only if $\varphi(m) \wedge \psi(m) = \mathsf{F}$ for all $m \in \text{Model}$. For a finite family $(\varphi_i)_{i \in I}$ of pairwise-disjoint symbolic propositions and a finite family of symbolic elements $(\widehat{a}_i)_{i \in I}$, their *symbolic union* $[\varphi_i : \widehat{a}_i]_{i \in I} \in \widehat{A}$ is the symbolic proposition defined by $([\varphi_i : \widehat{a}_i]_{i \in I})(m) = \widehat{a}_i(m)$ when $\varphi_i(m) = \mathsf{T}$ for some $i \in I$, and $([\varphi_i : \widehat{a}_i]_{i \in I})(m) = \bot$ when $\varphi_i(m) = \mathsf{F}$ for all $i \in I$.

Symbolic unions are functions out of the space of models, so symbolic quantities are automatically equal up to "nesting" of symbolic unions. This model of symbolic unions has been chosen specifically to facilitate reasoning about Rosette's symbolic evaluation strategy while abstracting over particular details of its implementation. Other Rosette models also abstract over the details of symbolic unions [52].

Figure 10 displays selected rules for defining the abstract semantics. As alluded to earlier, the abstract semantics lifts the input environment ρ and input store σ from the concrete semantics to a symbolic environment $\widehat{\rho}$ and symbolic store $\widehat{\sigma}$; accordingly, it produces symbolic values $\widehat{v}$. The rules are mostly analogous to the concrete ones, but there are a few important differences. Most notably, the $\Downarrow$ relation does not include a stream. This difference is due to the fundamental distinction between concrete and abstract execution: a symbolic evaluator computes *all* branches of execution simultaneously while a concrete evaluator does not.

For instance, consider the rule Sym. Instead of producing a particular Boolean value, the abstract semantics of `sym` generates a fresh symbolic variable and produces a symbolic union whose value is determined by the symbolic variable.

Also note the difference in the semantics of `if`, defined by the If rule. When encountering an `if`, the abstract semantics first evaluates the guard expression and determines the conditions φ_1 and φ_2 under which the guard expression produces the values `true` or `false`, respectively. The operator $\uplus_{\text{Bool}}$ indicates that the remainder of the symbolic union $\widehat{v}$ does not contain any Booleans—formally, $\text{img}(\widehat{v}) \cap \text{Bool} = \varnothing$. Both branches are evaluated, and the result is placed under a symbolic union with the corresponding guards. An additional output component ψ encodes the *assertion proposition*. This formula specifies models under which the output of the abstract semantics is defined. So, in models where the guard is not a Boolean, i.e., those mapped by $\widehat{v}$, the assertion proposition is not satisfied. The remaining rules can be found in Appendix B.

$\boxed{\text{Answers}}$

$$a \in \mathsf{Ans} ::= () \mid b \mid r \mid (a, a) \mid \text{opaque}$$

$\boxed{\lfloor \cdot \rfloor : \mathsf{Value} \to \mathsf{Ans}}$

$$\lfloor v \rfloor = \begin{cases} v & \text{if } v \in \mathsf{Bool} \cup \mathsf{Num} \cup \{()\} \\ (\lfloor v_1 \rfloor, \lfloor v_2 \rfloor) & \text{if } v = (v_1, v_2) \\ \text{opaque} & \text{otherwise} \end{cases}$$

$\boxed{\mathsf{eval} : \mathsf{Expr} \to (\mathsf{Stream} \to \mathsf{Ans}_\perp)}$

$$\mathsf{eval}(e)(s) = \begin{cases} \lfloor v \rfloor & \text{if } \varnothing, s \vdash (e, \varnothing) \Downarrow (v, \sigma) \\ \perp & \text{otherwise} \end{cases}$$

$\boxed{\mathsf{ans} : \widehat{\mathsf{Value}} \times \widehat{\mathsf{Store}} \to \widehat{\mathsf{Ans}}}$

$$\mathsf{ans}(\widehat{v}, \widehat{\sigma})(m) = \begin{cases} \lfloor \widehat{v}(m) \rfloor & \text{if } \widehat{v}(m) \neq \perp, \widehat{\sigma}(m) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Fig. 11. Idealized Rosette Evaluation Functions

4.4 Correctness

The primary correctness property of a precise symbolic evaluator is that it produces a representation of all and only those results that can be produced by concrete execution. In idealized Rosette, *answers* are the result of computation. The left side of Figure 11 explicitly defines the set of answers Ans and the function $\lfloor \cdot \rfloor$ for converting values to answers.

The right side of Figure 11 defines the meaning of concrete and abstract execution in terms of answers. Given a closed term e, the function $\mathsf{eval}(e)$ says which answers can be feasibly produced by e via concrete execution. Given an abstract value $\widehat{v}$ and an abstract store $\widehat{\sigma}$ produced by the abstract execution of e, the function $\mathsf{ans}(\widehat{v}, \widehat{\sigma})$ says which answers abstract execution considers feasible. Theorem 4.1 shows that these two results coincide and additionally that the assertion proposition produced by abstract execution is true exactly when concrete execution produces a well-defined value.

Theorem 4.1 (Correctness of Idealized Rosette)**.** Let e be a closed term where $\varnothing \vdash (e, \varnothing) \Downarrow\!\Downarrow (\widehat{v}, \widehat{\sigma}, \psi)$. Then $\mathsf{img}(\mathsf{eval}(e)) = \mathsf{img}(\mathsf{ans}(\widehat{v}, \widehat{\sigma}))$, and for all $m \in \mathsf{Model}_{\mathsf{symvars}(\widehat{v}, \widehat{\sigma})}$ it holds that $\psi(m) = \mathsf{T}$ if and only if $\widehat{v}(m) \neq \perp$ and $\widehat{\sigma}(m) \neq \perp$.

Proof sketch. The proof is by induction, after generalizing the theorem statement to open terms. To systematically work up to renaming of store locations, the proof contains a nontrivial use of nominal techniques [50]. For details, see Appendix B. □

5 Roulette: Formal Semantics

Given the semantics of Rosette, only a few small changes are needed to model Roulette. Specifically, each symbolic variable in the abstract semantics is additionally associated with a *weight*, which intuitively corresponds to the probability of that variable taking on the value T. Everything else can be adapted in a straightforward way. This section follows the same structure as the previous one, showing only the changes needed to turn Rosette into Roulette.

5.1 Syntax of Roulette

In the syntax of Roulette, the `sym` expression for generating symbolic Booleans is replaced by the expression `flip` x for generating a random Boolean with bias x. For concrete execution, the value of `flip` x is determined by the first element in a stream of real numbers $s \in \mathsf{Stream} \coloneqq \mathbb{N} \to [0, 1]$. This new stream, also known as an entropy source [11], provides a pool of uniform randomness for sampling. For abstract execution, `flip` returns a symbolic variable that is associated with a weight. This association is stored in a *weight map* $w \in \mathsf{WeightMap} \coloneqq \mathsf{SymVar} \to_{\mathsf{fin}} [0, 1]$.

$\boxed{\rho, s \vdash (e, \sigma) \Downarrow (v, \sigma')}$

$$\textsc{FlipTrue}\ \frac{x \in \mathrm{dom}(\rho) \qquad r < \rho(x)}{\rho, r :: s \vdash (\mathtt{flip}\, x, \sigma) \Downarrow (\mathtt{true}, \sigma)} \qquad \textsc{FlipFalse}\ \frac{x \in \mathrm{dom}(\rho) \qquad r \geq \rho(x)}{\rho, r :: s \vdash (\mathtt{flip}\, x, \sigma) \Downarrow (\mathtt{false}, \sigma)}$$

Fig. 12. Concrete Semantics

$\boxed{\widehat{\rho} \vdash (e, \widehat{\sigma}, w) \Downarrow\!\Downarrow (\widehat{v}, \widehat{\sigma}', w', \psi)}$

$$\textsc{Flip}\ \frac{\widehat{\rho}(x) = [\varphi_i : r_i]_{1 \leq i \leq n} \uplus_{\mathrm{Num}} \widehat{v} \qquad \forall 1 \leq i \leq n.\ s_i = \max(\min(r_i, 1), 0) \qquad \alpha_1, \ldots, \alpha_n \text{ smallest not in } \mathrm{dom}(w)}{\widehat{\rho} \vdash (\mathtt{flip}\, x, \widehat{\sigma}, w) \Downarrow\!\Downarrow ([\varphi_i \wedge \alpha_i \mapsto \mathsf{T},\ \varphi_i \wedge \neg\alpha_i \mapsto \mathsf{F}]_{i \in I},\ \widehat{\sigma},\ w \uplus \{\alpha_i \mapsto s_i\}_{i \in I},\ \bigvee_i \varphi_i)}$$

Fig. 13. Abstract Semantics

5.2 Concrete Semantics of Roulette

The concrete semantics of Roulette is given in Figure 12. There are two rules here intended to replace SymTrue and SymFalse. The `flip` form uses the first element of the entropy source to determine if `true` or `false` should be returned, based on the probability given to `flip`. This is exactly how `flip` is implemented in a sampling semantics where the entropy determines the values returned by the random-number generator.

5.3 Abstract Semantics of Roulette

The abstract semantics of Roulette is given in Figure 13. At a high-level, the Flip rule in the abstract semantics returns a fresh symbolic variable and a new weight map that associates the new variable with its probability. If the input probability is a concrete number, then only one symbolic variable is generated. Supporting symbolic arguments to `flip` takes some extra work and requires generating more symbolic variables. For each number in the symbolic union given to `flip`, the semantics generates a fresh symbolic variable and associates it with the given number. Since each r_i can be an arbitrary rational number, not just a probability, every r_i must be clamped to the interval. As in idealized Rosette, this rule is made deterministic by requiring $\alpha_1, \ldots, \alpha_n$ to be the smallest sequence of symbolic variables that is fresh for the variables currently in use.

5.4 Correctness

Theorem 4.1 generalizes smoothly from Rosette to Roulette. For Rosette, correctness amounted to showing that the abstract and concrete semantics compute the same set of possible answers. For Roulette, it is not only the set of possible answers that must be identical, but also the probability of each answer.

More precisely, the correctness theorem for Rosette required $\mathrm{eval}(e) : \mathrm{Stream} \to \mathrm{Ans}_\perp$ and $\mathrm{ans}(\widehat{v}, \widehat{\sigma}) : \mathrm{Model}_V \to \mathrm{Ans}_\perp$ to have the same image. In the analogous theorem for Roulette, these functions become *random variables* out of the sample spaces $\mathrm{Stream} = \mathbb{N} \to [0, 1]$ and $\mathrm{Model}_{\mathrm{dom}(\omega)}$, respectively, where ω is the weight map assigning probabilities to each symbolic variable generated during symbolic evaluation. These sample spaces naturally come with probability measures: Stream comes with the uniform Lebesgue measure following Culpepper and Cobb [11], and $\mathrm{Model}_{\mathrm{dom}(\omega)}$ comes with the measure that assigns a given model $m \in \mathrm{Model}_{\mathrm{dom}(\omega)}$ its *weight*: $\mathrm{weight}_w(m) = \prod_{(x \mapsto b) \in m} (\text{if } b \text{ then } w(x) \text{ else } 1 - w(x))$. For example, under the weight map

$w = [\alpha_0 \mapsto 0.1, \alpha_1 \mapsto 0.2]$, the model $m = [\alpha_0 \mapsto \mathsf{T}, \alpha_1 \mapsto \mathsf{F}]$ has $\mathsf{weight}_w(m) = 0.1 \cdot 0.8 = 0.08$. Correctness of Roulette states that the random variables $\mathsf{eval}(e)$ and $\mathsf{ans}(\widehat{v}, \widehat{\sigma})$ have the same distribution under these probability measures, and that the assertion proposition captures when these random variables are not equal to $\bot$. The proof follows the same structure as Theorem 4.1.

Theorem 5.1 (Correctness of Roulette)**.** Let e be a closed term such that $\varnothing \vdash (e, \varnothing, \varnothing) \Downarrow (\widehat{v}, \widehat{\sigma}, w, \psi)$. Then the random variables $\mathsf{eval}(e) : \mathsf{Stream} \to \mathsf{Ans}_\bot$ and $\mathsf{ans}(\widehat{v}, \widehat{\sigma}) : \mathsf{Model}_{\mathsf{dom}(w)} \to \mathsf{Ans}_\bot$ have the same distribution, where the entropy source Stream is given the uniform Lebesgue measure and $\mathsf{Model}_{\mathsf{dom}(w)}$ is given the measure weight_w. Moreover, for all $m \in \mathsf{Model}_{\mathsf{dom}(w)}$ it holds that $\psi(m) = \mathsf{T}$ if and only if $\widehat{v}(m) \neq \bot$ and $\widehat{\sigma}(m) \neq \bot$.

Proof. The proof is by induction, after generalizing to open terms. For details, see Appendix C. □

6 Evaluation

The examples in Section 2 demonstrate that Roulette is expressive, but to be useful, a probabilistic programming language must support *efficient* inference. This section confirms that Roulette is capable of scaling better than enumeration, and in general its performance is competitive with Dice [29], a state-of-the-art PPL for performing exact discrete inference. Unlike Roulette, Dice is extremely restricted: it has no support for recursion, mutable state, or many of the other features supported by Roulette.

The evaluation consists of three sets of programs. First is a series of small baseline programs commonly used to test PPLs in the literature. These programs are small enough that they finish nearly instantly. Second is a selection of discrete Bayesian networks intended to exercise the limits of efficient inference. These benchmarks come from the Bayesian Network Repository, a collection of Bayesian networks taken from real-world scenarios. All the networks are run on a program similar to the one described in Section 2.2. The final set of programs measures how well Roulette scales on the unreliable hardware example from Section 2.3, hidden Markov model queries, and the scaling experiments from Holtzen et al. [29].

Experimental Setup. We ran the experiments on a 16-core AMD EPYC 7543 CPU with 128GB of RAM using Racket 8.13 and Rosette 4.1. The benchmarks compare against two versions of Dice: a recent build (Git commit `ed86716`), referred to as Dice (2025), and the original version from Holtzen et al. [29], referred to as Dice (2020). All experiments were performed on a best-effort basis where an attempt was made to find the optimal encoding for the program in each language. Additionally, BDD performance is significantly impacted by variable order, and changing the variable order can increase the size of a BDD from linear to exponential [34]. To the extent possible, the experiments control for variable order: both Dice and Roulette use the program order heuristic (i.e., random variables generated earlier during execution come earlier in the variable order) and the programs are written such that the variable order is consistent between each system.

6.1 Results

The main aim in these experiments is to establish that Roulette is competitive with existing high-performance exact inference strategies while gaining expressivity. In general, there should be no expectation that Roulette significantly outperforms existing knowledge-compilation-based PPLs such as Dice. However, Roulette should also not exhibit a significant performance penalty as a price for its expanded expressivity. The remainder of this section presents the experimental results in brief and states their consequences. Then, Section 6.2 gives a higher-level discussion about performance that examines cross-cutting trends across the experiments. Some experimental results are truncated in the main body; see Appendix D for the full experimental results.

Table 1. Simple Baselines

	Roulette		Dice (2025)		Dice (2020)		PSI
Benchmark	BDD Size	Time (ms)	BDD Size	Time (ms)	BDD Size	Time (ms)	Time (ms)
ALARM	**11**	**0**	**11**	19	**11**	16	71
EVIDENCE1	**3**	**0**	5	24	5	17	22
EVIDENCE2	**4**	**0**	6	21	6	18	34
GRASS	**13**	**0**	15	30	15	19	80
MURDER-MYSTERY	**4**	**0**	6	28	6	20	46
NOISY-OR	**33**	**0**	35	23	35	19	328
TWO-COINS	**3**	**0**	5	25	5	22	20

Table 2. Bayesian Networks

	Roulette		Dice (2025)		Dice (2020)	
Benchmark	BDD Size	Time (ms)	BDD Size	Time (ms)	BDD Size	Time (ms)
CANCER	**13**	**2**	15	49	28	19
SURVEY	**46**	**4**	48	29	73	18
ALARM	981	42	**672**	308	1,366	**30**
INSURANCE	75,594	395	**44,846**	643	101,047	**148**
HEPAR2	**1,967**	140	1,969	230	3,936	**32**
HAILFINDER	**33,211**	596	–	–	65,386	**428**
PIGS	**19**	255	25	417	35	**48**
WATER	39,146	**200**	**33,226**	454	51,952	16,083
MUNIN	**10,307**	2,400	3,704	24,839	11,977	**1,605**

6.1.1 Simple Baselines. Table 1 shows the results of the benchmarks introduced by Gehr et al. [23]. The time taken by Roulette and Dice to solve these problems is nearly instant in all cases. PSI [23] is included as a comparison point in this experiment, which lags behind the performance of the knowledge-compilation-based approaches. The table additionally reports the BDD size (in number of nodes) to show the degree to which the different encoding schemes of Roulette, Dice (2020), and Dice (2025) impact performance. Roulette achieves the smallest BDD in all cases.

6.1.2 Bayesian Networks. Table 2 shows the results for single-marginal inference on the nine discrete Bayesian networks evaluated in the original Dice paper [29]. The "–" symbol indicates a timeout after one minute. Single-marginal inference calculates the marginal probability for a single leaf node in a network. These results are for a Bayesian network interpreter that is almost identical to the one in Section 2.2.

Overall, Roulette is competitive with both versions of Dice on these examples, with similar overall BDD size and execution time. In the table, networks are roughly ordered by difficulty:

- *Small networks.* CANCER and SURVEY are small and therefore inference is relatively easy. Roulette has the best performance on these.
- *Medium networks.* ALARM, INSURANCE, HEPAR2, and HAILFINDER are medium-sized Bayesian networks. In general, Dice (2020) has the smallest execution time on these networks, but Roulette is close in performance. Roulette compiles to the smallest final BDD in most of these examples, suggesting that its encoding is quite efficient. There were some performance regressions in Dice (2025), and it failed to complete HAILFINDER.
- *Large networks.* PIGS, WATER, and MUNIN are large and challenging networks with many thousands of parameters. Roulette is significantly faster than both versions of Dice on the WATER network. On the MUNIN network, Roulette generates the smallest BDD, but its performance is marginally inferior to Dice (2020) and lags by about 800 milliseconds.

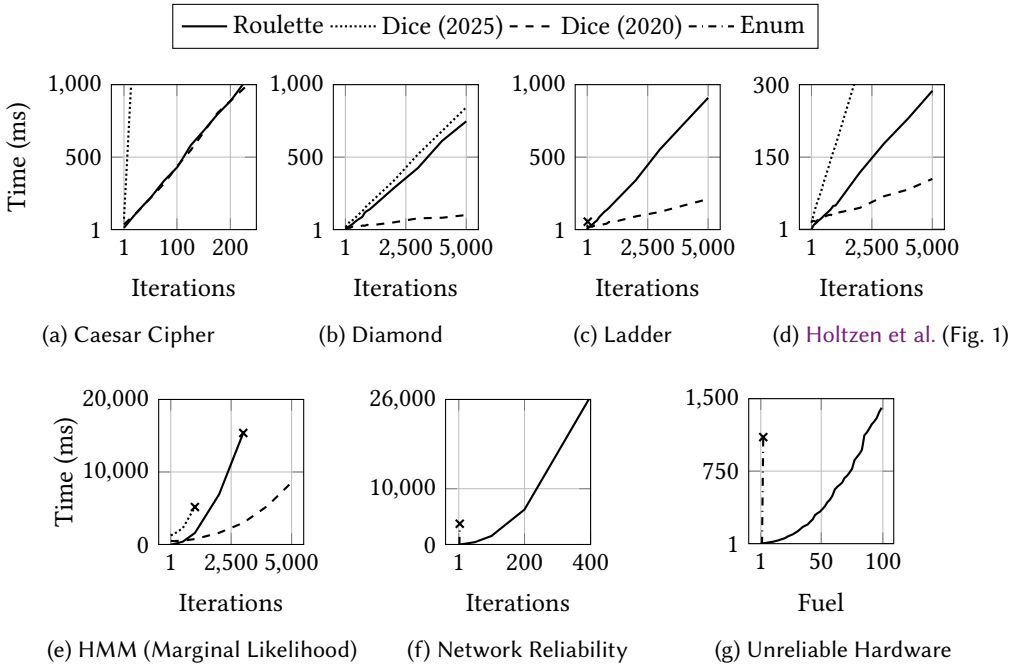


Fig. 14. Plots (a), (b), (c), and (d) parallel Figure 10 from Holtzen et al. [29]. Plot (e) is a hidden Markov model example. These five plots compare against Dice. Plot (f) is the network reliability program from Section 2.1, and Plot (g) is the unreliable hardware program from Section 2.3. These benchmarks compare against enumeration. An × shows the last trial to finish within a one-minute timeout.

Overall, these experiments show that Roulette is not giving up significant performance to attain its expressivity. Roulette can complete all the Bayesian network tasks within a reasonable amount of time and in some cases outperforms Dice on challenging examples such as WATER.

6.1.3 Scaling Benchmarks. Figure 14 shows the results of experiments that examine how Roulette and Dice scale as parameters grow in size. Some of these plots reproduce benchmarks from the original performance evaluation of Dice, while other benchmarks are new. Here is a detailed analysis of these experiments:

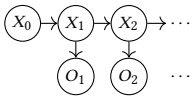
- Figure 14a is the Caesar cipher example from Figure 3 of Holtzen et al. [29]. The program performs frequency analysis to decrypt text that is encrypted using a Caesar cipher. The scaling parameter is the length of the ciphertext. Roulette and Dice (2020) achieve nearly identical performance on this example.
- Figure 14b and Figure 14c examine the probabilistic network reliability benchmarks from Holtzen et al. [29]. The following shows diamond and ladder on the left and right, respectively:



The scaling parameter is the number of repeated subnetworks (boxed). The task is a simplified version of the example from Figure 1: for a particular network topology where each edge has some chance of dropping a packet and each router forwards uniformly at random, determine whether a packet successfully traverses the network. Here, Roulette and Dice (2020) both scale

linearly, with Dice (2020) having overall superior constant factors. Dice (2025) scales surprisingly poorly on ladder despite using the same BDD backend as Roulette. Overall, these experiments demonstrate that Roulette and Dice (2020) have similar asymptotic behavior, with Dice (2020) having superior constant factors. This performance difference is not likely to be an inherent limitation of Roulette but is more likely due to confounding factors, such as Dice (2020) using a different BDD backend.

- Figure 14d is the simple Markov chain example from Holtzen et al. [29, Figure 1]. Again, all tools achieve linear-time performance with Dice (2020) having superior constant factors.
- Figure 14e shows the scaling behavior for querying the following hidden Markov model (HMM):



  The scaling parameter is the length of the HMM. The inference task is, for an HMM of length n, to compute the probability that all the O_i are true (each node in the network is a Boolean random variable). In addition to these automated approaches (Roulette and Dice), we implemented a specialized inference algorithm that performs linear-time exact inference on HMMs [53]. These results are detailed in Appendix D, along with the performance of several other HMM queries. The automated approaches exhibit super-linear scaling behavior, with Dice (2020) having the best performance. Scaling is quadratic for Roulette and Dice due to the way that the BDD is constructed: each additional layer in an HMM requires traversing the entire previously-constructed (linear-size) BDD once, which scales quadratically in n.
- Finally, Figure 14f and Figure 14g examine how Roulette scales on some of the examples from Section 2. The baseline for comparison is enumeration since these programs are not easily representable in Dice. Figure 14f shows how the network reliability example from Section 2.1 scales in the `SIZE` parameter, and Figure 14g shows how the unreliable hardware example from Section 2.3 scales as `(tri (max 0 (- fuel 2)))` is evaluated on varying values of `fuel`. Enumeration times out quickly and exhibits exponential growth, while Roulette is able to scale comfortably. A comparison of least-squares regressions shows that, along all metrics, a quadratic model fits the data better than an exponential model for both the network reliability and unreliable hardware examples.

6.2 Performance Discussion

The experiments demonstrate that Roulette and Dice have similar scaling characteristics but exhibit constant-factor performance differences on some examples. While Roulette, Dice (2025), and Dice (2020) all use knowledge compilation for probabilistic inference, there are differences that impact the performance of these languages on the benchmarks. First, Roulette and Dice (2025) use the RSDD library [30] for managing BDDs, while Dice (2020) uses the CUDD library [59]. Second, Roulette is a DSL hosted in Racket, while Dice has a special-purpose compiler. As such, Dice generally has lower constant-time factors than Roulette. The Rosette runtime also performs formula simplification, caching, and other tasks that increase overhead.

Nonetheless, it is possible to achieve strong performance in Roulette by leveraging the host language's expressivity to control the way in which knowledge compilation is performed. This is similar to programming in Rosette, where changes to the structure of a program can have a significant influence on the resulting encoding and subsequently on the performance of queries to the underlying solver [51]. The expressivity afforded by Roulette allows developers to tune programs in this way. For example, a recursive function may be written in tail form or not, affecting the order in which variables are generated and hence affecting the compactness of BDD encoding.

Roulette's expressivity can help developers write optimized programs in the following ways:

- *Variable order.* The variable order in Roulette is controlled by the order in which random variables are allocated using `flip`. Hence, it is occasionally necessary for the programmer to pre-allocate random variables in order to satisfy some ordering constraint for optimal compilation.
- *Encoding strategy.* The way that data is represented as Boolean formulae is a critical component in the scalability of knowledge compilation strategies [5]. For instance, a number can be represented in binary or with a one-hot encoding. The choice of representation can greatly impact the performance of certain operations. Unlike Dice, Roulette is expressive enough to give this power to users. The Caesar cipher example in Figure 14a uses ordinary functional programming to implement a binary encoding of numbers, which is critical to being competitive with Dice.
- *Compilation order.* Roulette works by generating a Boolean formula that is shipped to a knowledge compiler. The performance of a knowledge compiler is sensitive to the formula it is given. Hence, control over formula structure is critical for achieving linear-time performance on the benchmarks shown in Figures 14b to 14d. In particular, it is necessary to compile the chain "from back to front." This strategy differs from the more naive implementation approach; see Appendix E for a concrete comparison of these two approaches.

These optimizations are available to programmers and can be considered "design patterns" for scalable inference. They follow the general guidance of aiming to (1) have related variables close to each other in the variable order and (2) compile related parts of the program together.

Beyond these guidelines there are additional Roulette-specific low-level optimizations, needed to obtain competitive performance with Dice, that are not available to the programmer for customization. Roulette has to work around some limitations of Rosette that do not occur in Dice. For instance, conditionals in Dice are compiled to `ite` operations on BDDs (which are more efficient), but Rosette does not have a Boolean-valued `ite` node in its formula representation. Therefore, places where conditionals occur have to be inferred, perhaps imperfectly, from the formulae.

Despite these optimizations, it is still the case that Dice outperforms Roulette on some examples. The remaining performance gap is likely due to a combination of (1) inherent overhead of Roulette being hosted in Racket, (2) differing knowledge compilation backends exhibiting different performance on the same examples, and (3) additional optimizations that are available in Dice due to the particularly constrained language.

7 Related Work

Probabilistic Programming Languages. The most closely related probabilistic programming languages to Roulette are those that rely on weighted model counting for exact inference such as Dice [29] and ProbLog [17]. Both Dice and ProbLog rely on a laborious and manual translation of the program structure into weighted Boolean formulae. Roulette vastly simplifies the implementation of these languages and increases expressivity, as demonstrated in Section 2. The exact performance of WMC-based inference is sensitive to how constraints are generated. As shown by Cao et al. [5], subtle choices about how to encode integers can result in different performance characteristics. It is somewhat surprising that Roulette performs as well as it does despite minimal manual optimization beyond what is provided by Rosette; this is a testament to the quality of the formula simplification provided already by the Rosette symbolic evaluator. Nonetheless, it would be interesting future work to integrate the optimizations explored by Cao et al. [5] into Roulette. Garg et al. [21] bit blast continuous random variables in order to use exact WMC-based inference for reasoning about continuous probability distributions; it would also be potentially fruitful to integrate these compilation strategies into Roulette.

While Roulette is expressive for a PPL, it nonetheless omits two useful features: continuous random variables and almost-sure termination. These features are found in many PPLs, but their presence makes exact inference challenging because they require integration. Hakaru [39] and PSI [23] support exact inference in the presence of continuous random variables, and their inference strategies consequently rely on expensive exact integration performed by computer algebra systems. SPPL [56] also supports exact inference in the presence of continuity, but it is a restricted language that lacks mutable state, higher-order functions, and general recursion. Approaches to handling almost-sure termination rely on fix-point solving [8] or generating functions [33]; it would be interesting to adapt these approaches to work with Roulette programs.

Solver-aided Programming. Solver-aided programming was introduced by Rosette [61, 62] and has since been applied in variety of domains. Rosette has been used to verify JIT compilers within the Linux kernel [40], develop a policy specification language for router configuration [64], create an instruction set architecture (ISA) emulator for building correct superoptimizers [49], and even verify safety properties of medical devices [48].

Porncharoenwase et al. [52] present a formally verified semantics of Rosette. Specifically, their $\mathcal{S}_c$ semantics is parameterized by a symbolic factory that abstracts the implementation details of symbolic values. Section 4 does not build on $\mathcal{S}_c$ primarily because it lacks support for dynamically generating symbolic variables and mutable state.

Symbolic Execution for Probabilistic Programming. Applying techniques in symbolic execution to probabilistic programming is not a new idea. Geldenhuys et al. [25] repurpose a symbolic execution framework for verifying Java programs to determine the probability that a function under test encounters a bug. This system relies on model counting, which weights all possible test inputs uniformly to determine the probability of failure and lacks Bayesian conditioning. Susag et al. [60] propose probabilistic symbolic variables for the Plinko language. Plinko is limited to finite loops, variable assignment, conditionals, and drawing symbolic variables from discrete probability distributions. Voogd et al. [63] present a full run-time semantics for probabilistic programming inspired by symbolic execution, called symProb, which performs bounded symbolic execution. However, symProb lacks higher-order functions and higher-order mutable state.

8 Conclusion

Roulette is an expressive discrete probabilistic programming language with exact inference built on top of Rosette. The key idea of Roulette is to adapt the symbolic compilation strategy of Rosette to generate symbolic constraints that can be given to a WMC-based inference backend. This strategy yields both an expressive and efficient probabilistic programming environment. To prove Roulette sound, we first gave a correctness theorem for an idealized version of Rosette that generalizes prior Rosette formalisms by including dynamically introduced symbolic values and mutable state. Idealized Rosette then served as the foundation for an analogous correctness theorem of Roulette. Finally, a suite of experiments showed that Roulette can achieve competitive performance on discrete exact inference for several challenging tasks.

In the future, we hope to make Roulette more usable and scalable by further connecting ideas from knowledge-compilation-based inference and Rosette. There are a variety of high-quality WMC solvers that would be interesting to explore as alternatives to BDDs as backends for Roulette [9, 12, 13, 35, 37]. Since Roulette is as expressive as Rosette, future work may also explore the extent to which the numerous existing Rosette examples benefit from a probabilistic interpretation. And finally, Rosette has benefited from many years of usability improvements, including symbolic profiling [3], performance repair [51], and typed lenient symbolic evaluation [6]. The extent to which these can be generalized and applied to Roulette is also exciting future work.

Acknowledgments

This work was supported by NSF grants SHF 2116372 and FMitF 2220408. The authors would like to thank Ryan Culpepper, Lisa Oakley, Sam Stites, and the anonymous PLDI reviewers, for their comments and suggestions.

Data Availability Statement

An up-to-date version of the software and appendices associated with this paper can be found at: https://doi.org/10.5281/zenodo.15049041. An archived version is also available [36].

References

[1] Michael Ballantyne, Mitch Gamburg, and Jason Hemann. 2024. Compiled, Extensible, Multi-language DSLs (Functional Pearl). In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3674627

[2] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Beguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2492061

[3] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3276519

[4] William E Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph. D. Dissertation. Indiana University.

[5] William X Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling Integer Arithmetic in Probabilistic Programs. In *Uncertainty in Artificial Intelligence (UAI)*. https://doi.org/10.48550/arXiv.2307.13837

[6] Stephen Chang, Alex Knauth, and Emina Torlak. 2018. Symbolic Types for Lenient Symbolic Execution. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3158128

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

[8] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3586050

[9] Arthur Choi, Doga Kisa, and Adnan Darwiche. 2013. Compiling Probabilistic Graphical Models Using Sentential Decision Diagrams. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. https://doi.org/10.1007/978-3-642-39091-3_11

[10] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Conference on Innovative Data Systems Research (CIDR)*.

[11] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-54434-1_14

[12] Adnan Darwiche. 2002. A Compiler for Deterministic, Decomposable Negation Normal Form. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[13] Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-143

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

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-540-78800-3_24

[16] 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)*.

[17] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas. *Theory and Practice of Logic Programming*. https://doi.org/10.1017/S1471068414000076

[18] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

[19] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer*. MIT Press.

[20] Norbert Fuhr. 1995. Probabilistic Datalog — A Logic for Powerful Retrieval Methods. In *Conference on Research and Development in Information Retrieval (SIGIR)*. https://doi.org/10.1145/215206.215372

[21] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2024. Bit Blasting Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3656412

[22] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3296979.3192400

[23] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-41528-4_4

[24] Timon Gehr, Samuel Steffen, and Martin T. Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3386006

[25] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2338965.2336773

[26] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Scheme and Functional Programming Workshop*.

[27] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2017. A Simple Complete Search for Logic Programming. In *International Conference on Logic Programming (ICLP)*. https://doi.org/10.4230/OASICS.ICLP.2017.14

[28] Ralf Hinze and Colin Runciman. 2022. Super-Naturals. *Journal of Functional Programming (JFP)*. https://doi.org/10.1017/s0956796822000028

[29] 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)*. https://doi.org/10.1145/3428208

[30] Steven Holtzen, Matthew Wang, John Gouwar, Sam Stites, and Minsung Cho. 2025. The Rust Decision Diagram Library (RSDD). https://github.com/neuppl/rsdd

[31] Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science (STACS)*. https://doi.org/10.1007/BFB0039592

[32] Rafael Kiesel and Thomas Eiter. 2023. Knowledge Compilation and More with SharpSAT-TD. In *Principles of Knowledge Representation and Reasoning (KR)*. https://doi.org/10.24963/kr.2023/40

[33] 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)*. https://doi.org/10.1145/3649844

[34] Donald E Knuth. 2009. *The Art of Computer Programming*. Vol. 4. Addison-Wesley Professional.

[35] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler. In *International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.24963/ijcai.2017/93

[36] Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. 2025. Artifact: Roulette. https://doi.org/10.5281/zenodo.15253254

[37] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Advances in Artificial Intelligence*. https://doi.org/10.1007/978-3-642-30353-1_36

[38] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics*.

[39] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Functional and Logic Programming (FLOPS)*. https://doi.org/10.1007/978-3-319-29604-3_5

[40] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *Operating Systems Design and Implementation (OSDI)*.

[41] Raymond Ng and Venkatramanan Siva Subrahmanian. 1992. Probabilistic Logic Programming. *Information and Computation*. https://doi.org/10.1016/0890-5401(92)90061-J

[42] Lisa Oakley, Steven Holtzen, and Alina Oprea. 2024. Synthesizing Tight Privacy and Accuracy Bounds via Weighted Model Counting. In *Computer Security Foundations Symposium (CSF)*. https://doi.org/10.1109/CSF61375.2024.00048

[43] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-49498-1_23

[44] Umut Oztok, Arthur Choi, and Adnan Darwiche. 2016. Solving PP^{PP}-Complete Problems using Knowledge Compilation. In *Principles of Knowledge Representation and Reasoning (KR)*.

[45] Umut Oztok and Adnan Darwiche. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[46] Umut Oztok and Adnan Darwiche. 2018. An Exhaustive DPLL Algorithm for Model Counting. *Journal of Artificial Intelligence Research*. https://doi.org/10.1613/jair.1.11201

[47] Judea Pearl. 1985. Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning. In *Conference of the Cognitive Science Society*.

[48] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-41540-6_2

[49] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2872362.2872387

[50] Andrew M Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press.

[51] Sorawee Porncharoenwase, James Bornholt, and Emina Torlak. 2020. Fixing Code That Explodes Under Symbolic Evaluation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-030-39322-9_3

[52] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A Formal Foundation for Symbolic Evaluation with Merging. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3498709

[53] Lawrence R Rabiner. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE*. https://doi.org/10.1109/5.18626

[54] Dan Roth. 1996. On the Hardness of Approximate Reasoning. *Artificial Intelligence*. https://doi.org/10.1016/0004-3702(94)00092-1

[55] Colin Runciman. 1989. What About the Natural Numbers? *Computer Languages*. https://doi.org/10.1016/0096-0551(89)90004-0

[56] 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)*. https://doi.org/10.1145/3453483.3454078

[57] Tian Sang, Paul Beame, and Henry A Kautz. 2005. Performing Bayesian Inference by Weighted Model Counting. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[58] Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314639

[59] Fabio Somenzi. 1998. CUDD: CU Decision Diagram Package Release. https://add-lib.scce.info/assets/documents/cudd-manual.pdf

[60] Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Symbolic Execution for Randomized Programs. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3563344

[61] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Onward!* https://doi.org/10.1145/2509578.2509586

[62] Emina Torlak and Rastislav Bodík. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594340

[63] Erik Voogd, Einar Broch Johnsen, Alexandra Silva, Zachary J Susag, and Andrzej Wąsowski. 2023. Symbolic Semantics for Probabilistic Programs. In *International Conference on Quantitative Evaluation of Systems*. https://doi.org/10.1007/978-3-031-43835-6_23

[64] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2983990.2984012

Received 2024-11-15; accepted 2025-03-06