Ranked Programming

Tjitze Rienstra
Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
rienstra@uni-koblenz.de

Abstract

While probabilistic programming is a powerful tool, uncertainty is not always of a probabilistic kind. Some types of uncertainty are better captured using *ranking theory*, which is an alternative to probability theory where uncertainty is measured using degrees of surprise on the integer scale from 0 to ∞. In this paper we combine probabilistic programming methodology with ranking theory and develop a *ranked programming language*. We use the *Scheme* programming language a basis and extend it with the ability to express both normal and exceptional behaviour of a model, and perform inference on such models. Like probabilistic programming, our approach provides a simple and flexible way to represent and reason with models involving uncertainty, but using a coarser grained and computationally simpler kind of uncertainty.

1 Introduction

Probabilistic programming languages (PPLs) are regular programming languages extended with statements to express probabilistic behaviour and to process evidence via conditionalization. Mixing such statements with regular programming constructs enables one to easily and flexibly build complex probabilistic models and perform inference on them. Examples of PPLs are *Church*, *Venture*, *Figaro*, *Anglican* and *Stan* [Goodman *et al.*, 2008; Mansinghka *et al.*, 2014; Pfeffer, 2009; Carpenter *et al.*, 2017; Wood *et al.*, 2014].

While PPLs are powerful tools, the approach assumes that the uncertainty we are dealing with is probabilistic. However, some situations involve uncertainty of a different nature [Halpern, 2017]. In this paper we focus on situations where we can distinguish *normal* from *exceptional* behaviour but where the probabilistic meaning of these terms is unknown or irrelevant. For example, in fault diagnosis all we may know is that components normally work and only exceptionally fail, or if we process sensor data or user input, each piece of data is normally correct but exceptionally wrong.

Ranking theory [Spohn, 2014] is an alternative to probability theory in which uncertainty is measured on the integer scale from 0 to ∞. These values, called *ranks*, can be understood as degrees of surprise: 0 for not surprising, 1 for surprising, 2 for even more surprising, and so on, with ∞ for impossible. Such a scale is a good fit for the kind of uncertainty outlined above: *normal* means rank 0, while *exceptional* means rank ≥ 1. Even though ranks behave quite differently from probabilities, there are many similarities between the two. For example, ranking functions—like probability functions—are updated by conditionalization, and can be represented compactly by exploiting conditional independencies encoded by graph structures.

In this paper we apply the methodology of PPLs to ranking theory, and develop a *ranked programming language*. It is based on the *Scheme* programming language [Dybvig, 2009], which is a dialect of LISP, and we call it *Ranked Scheme*. Uncertainty in Ranked Scheme is expressed with *ranked choice* expressions, which normally (rank 0) return a value X but exceptionally (rank ≥ 1) another value Y. Expressions in Ranked Scheme return ranking functions over their return values. The rank of a return value can be understood as its degree of surprise or, alternatively, as the number of exceptions that must occur in order to arrive at that return value. Finally, evidence is processed with *observe* expressions, which capture the ranking-theoretic conditionalization operation.

The idea of ranked programming was introduced in [Rienstra, 2017], which discusses a minimal imperative ranked programming language. The current approach has a simpler semantics, is easier to implement, and is significantly more powerful since it permits mixing ranked programming with regular Scheme code. An implementation was developed using the *Racket* Scheme dialect as a basis. For download and instructions see https://pkgd.racket-lang.org/pkgn/package/ranked-programming.

This paper is structured as follows. In section 2 we present the necessary basics of ranking theory, along with a short discussion of how ranks behave compared to probabilities. We then present in section 3 the Ranked Scheme language. It consists of a small set of special expressions whose semantics is presented in a denotational style. We then discuss issues related to the implementation in section 4. Section 5 is dedicated to examples and we conclude in section 6.

2 Ranking Theory

We start with the necessary basics conerning ranking theory. A *ranking function* over a set Ω of possible worlds is a function $\kappa : \Omega \to \mathbb{N} \cup \{\infty\}$ such that $\kappa(w) = 0$ for at least one

$w \in \Omega$. Thus, each possible world w is associated with a non-negative integer or ∞. These values, called *ranks*, represent degrees of surprise: 0 for not surprising, 1 for surprising, 2 for more surprising, and so on, with ∞ indicating impossibility. A ranking function is extended to a function over events (subsets of Ω, which we denote by A or B) as follows.

$$\kappa(A) = \begin{cases} min(\{\kappa(w) \mid w \in A\}) & \text{if } A \neq \emptyset, \\ \infty & \text{if } A = \emptyset. \end{cases} \quad (1)$$

Like conditional probabilities, conditional ranks are ranks given that some event holds. The conditional rank of a possible world w given B (for $\kappa(B) < \infty$) is defined as follows.

$$\kappa(w \mid B) = \begin{cases} \kappa(w) - \kappa(B) & \text{if } w \in B, \\ \infty & \text{if } w \notin B. \end{cases} \quad (2)$$

Combining this with (1) yields the definition for conditional ranks of events:

$$\kappa(A \mid B) = \begin{cases} min(\{\kappa(w \mid B) \mid w \in A\}) & \text{if } A \neq \emptyset, \\ \infty & \text{if } A = \emptyset, \end{cases} \quad (3)$$

which is equivalent to

$$\kappa(A \mid B) = \kappa(B \cap A) - \kappa(B). \quad (4)$$

To see how ranks behave compared to probabilities, the following rule of thumb applies: rank 0 plays the role of probability 1, rank ∞ plays the role of probability 0, and min, $-$ and $+$ play the role, respectively, of $+$, $\div$ and $\times$. Recall, for example, the probabilistic axiom $P(A \cap B) = P(B \mid A)P(A)$. By contrast, rewriting (4) yields $\kappa(A \cap B) = \kappa(B \mid A) + \kappa(A)$ or, in words, the degree of surprise of A and B equals that of B given A plus that of A. Similarly, the axiom $P(A) + P(\Omega \setminus A) = 1$ corresponds in ranking theory to $min(\kappa(A), \kappa(\Omega \setminus A)) = 0$ or, in words, an event and its complement cannot both be surprising. For further details we refer the reader to [Spohn, 2009] or [Spohn, 2014].

3 Ranked Scheme

Ranked Scheme is an extension of the Scheme programming language. We assume in this paper that the reader is familiar with the basics of Scheme (see, e.g., [Dybvig, 2009] for an introduction). The extension consists of a small set of special expressions called *R-expressions*, which generate and manipulate ranking functions over values. Formally, let $\mathbb{V}$ denote the set of all Scheme values (booleans, integers, strings, list structures, procedure objects, etc). While a regular Scheme expression returns a member of $\mathbb{V}$, an R-expressions returns a *ranking function over* $\mathbb{V}$. The core R-expression types are:

- `!x` (Truth)
- `(nrm` κ_1 `exc r` κ_2`)` (Ranked Choice)
- `(observe p` κ`)` (Observation)
- `($` κ_1 `...` κ_n`)` (Ranked Procedure Call)

Here, the symbols `x`, `r`, `p` and κ are parameters. The symbol used furthermore specifies the expected type: `x` for arbitrary members of $\mathbb{V}$; `r` for ranks (non-negative integers or ∞); `p` for predicates (one-argument functions returning true or false); and κ for ranking functions over $\mathbb{V}$. Ranking functions over $\mathbb{V}$ may be represented by lazily-linked list structures (see section 4) and may therefore technically also be members of $\mathbb{V}$. However, for the formal semantics, which is the topic of this section, their representation is irrelevant.

We define the semantics of R-expressions in a denotational style, by defining a function D that maps each R-expression `exp` to a ranking function $D[\![\texttt{exp}]\!]$ over $\mathbb{V}$. Intuitively, this ranking function captures the uncertainty about the return value of `exp`. That is, $D[\![\texttt{exp}]\!](v)$ is the degree of surprise that `exp` returns v or, alternatively, the number of exceptional events that must take place for `exp` to return v.

3.1 Truth

The *truth* expression `!x` captures a value that equals `x` with absolute certainty. It produces a ranking function according to which `x` is ranked 0 and all other values are ranked ∞:

$$D[\![\,!\texttt{x}]\!](v) = \begin{cases} 0 & \text{if x evaluates to } v, \\ \infty & \text{otherwise.} \end{cases}$$

3.2 Ranked Choice

The ranked choice expression `(nrm` κ_1 `exc r` κ_2`)` expresses uncertainty. Intuitively, it *normally* yields the value captured by κ_1, and *exceptionally* (rank `r`) the value captured by κ_2. It evaluates to a ranking function in which the rank of a value v is the minimum among $\kappa_1(v)$ and $\kappa_2(v) + \texttt{r}$:

$$D[\![\texttt{(nrm } \kappa_1 \texttt{ exc r } \kappa_2\texttt{)}]\!](v) = min(\kappa_1(v), \kappa_2(v) + \texttt{r}).$$

Clearly, if κ_1 and κ_2 are ranking functions then so is $D[\![\texttt{(nrm } \kappa_1 \texttt{ exc r } \kappa_2\texttt{)}]\!]$. Setting `r` to 1 means that κ_2 is simply exceptional, while a value exceeding 1 can be understood as saying that κ_2 counts as multiple exceptional events. Setting `r` to 0 means that κ_1 and κ_2 are equally surprising. For this case we define the following syntactic shortcut:

$$D[\![\texttt{(either } \kappa_1 \texttt{ or } \kappa_2\texttt{)}]\!] = D[\![\texttt{(nrm } \kappa_1 \texttt{ exc 0 } \kappa_2\texttt{)}]\!].$$

Below we indicate with => that the preceding expression returns the ranking function that follows, displayed in tabular form from lowest to highest rank, omitting rank ∞.

```
(nrm !"foo" exc 1 !"bar")
=> Rank  Value
   0     "foo"
   1     "bar"
(nrm !"foo" exc 1 (either !"bar" or !"baz"))
=> Rank  Value
   0     "foo"
   1     "bar"
   1     "baz"
```

Below we first define the recursive function `(fun x)` that normally returns `x` and exceptionally `(fun (* x 2))`. We then call `(fun 1)`.

```
(define (fun x) (nrm !x exc 1 (fun (* x 2))))
(fun 1)
=> Rank  Value
   0     1
   1     2
   2     4
   3     8
   ...   ...
```

Note that the ranking returned here assigns finite ranks to an infinite number of values (all powers of two). We use dots to indicate that the displayed ranking function is truncated.

3.3 Observation

The expression `(observe p` κ`)` captures the ranking-theoretic conditionalization operation. Here, κ is the value of interest and `p` is the predicate on which we conditionalize. This predicate is any one-argument function `p` such that `(p` v`)` evaluates to either true or false, represented in Scheme by the symbols `#T` and `#F`. Let `(p` v`)` $\Rightarrow$ `#T` denote the fact that `(p` v`)` evaluates to `#T` and let [p] denote the set $\{v \in \mathbb{V} \mid$ `(p` v`)` $\Rightarrow$ `#T`$\}$. The semantics of `observe` is described by the following rule (note the resemblance with (2)):

$$D[\![\texttt{(observe p } \kappa\texttt{)}]\!](v) = \begin{cases} \kappa(v) - \kappa([\texttt{p}]) & \text{if } v \in [\texttt{p}], \\ \infty & \text{if } v \notin [\texttt{p}]. \end{cases}$$

It can be checked that, if κ is a ranking function, then so is $D[\![\texttt{(observe p } \kappa\texttt{)}]\!]$, provided that $\kappa([p]) < \infty$. Thus, κ must assign a finite rank to at least one value such that `(p` v`)` evaluates to `#T`. For simplicity, we assume that `p` is chosen so that this holds. Alternatively, the semantics can be extended with values denoting *failure of computation*. To keep the definitions simple, we shall not pursue this option.

In the following example, `fun` is as defined in section 3.2 and `(lambda (x) (> x 100))` defines a predicate that returns true only for values greater than 100.

```
(observe (lambda (x) (> x 100)) (fun 1))
=> Rank  Value
   0     128
   1     256
   2     512
   ...   ...
```

3.4 Ranked Procedure Call

The ranked procedure call `($` κ_1 ... κ_n`)` (with $n \geq 1$) generalises the regular procedure call in Scheme. Recall that a regular procedure call is an expression `(`$\texttt{x}_1$ ... $\texttt{x}_n$`)` that, when evaluated, calls the procedure $\texttt{x}_1$ with arguments $\texttt{x}_2, \ldots, \texttt{x}_n$ and returns the result. In a ranked procedure call `(`$ \kappa_1$ ... κ_n`)`, κ_1 is a ranking function over procedures and $\kappa_2, \ldots, \kappa_n$ are ranking functions over arguments for the procedure call. Let `(`v_1 ... v_n`)` $\Rightarrow$ v denote the fact that the regular procedure call `(`v_1 ... v_n`)` evaluates to v. The semantics of the ranked procedure call is defined by:

$$D[\![\texttt{(\$ } \kappa_1 \texttt{ ... } \kappa_n\texttt{)}]\!](v) = min(\{\textstyle\sum_{i=1}^{n} \kappa_i(v_i) \mid (v_1, \ldots, v_n) \in \mathbb{V}^n, \texttt{(}v_1 \texttt{ ... } v_n\texttt{)} \Rightarrow v\}).$$

In words we can describe the semantics of `$` as follows: if there is a sequence $v_1 \ldots v_n$ of values such that the regular procedure call $(v_1 \ldots v_n)$ evaluates to v, then `($` $\kappa_1 \ldots \kappa_n$`)` yields v with rank $\sum_{i=1}^{n} \kappa_i(v_i)$, unless another sequence $v'_1, \ldots, v'_n$ yields a lower rank for v using the same rule.

In the example below we calculate the sum of two values, the first being normally 10 and exceptionally 20, and the second one is simply 5. Recall that the symbol + in Scheme is nothing but a variable bound to a procedure object. Therefore, `!+` evaluates to a ranking function assigning rank 0 to the procedure object +, and rank ∞ to all other values.

```
($ !+ (nrm !10 exc 1 !20) !5)
 => Rank  Value
    0     15
    1     25
```

Let us extend this example by adding uncertainty about the operation: we normally add but exceptionally subtract.

```
($ (nrm !+ exc 1 !-) (nrm !10 exc 1 !20) !5)
 => Rank  Value
    0     15
    1     25
    1     5
```

3.5 Ranked Let

An example: let `b` and `p` be boolean variables standing for *beer* and *peanuts*. We only exceptionally drink beer, and thus `b` becomes `(nrm !#F exc 1 !#T)`. However, our peanut consumption depends on whether we drink beer: if we do, we normally have peanuts, otherwise we don't. Thus, `p` becomes `(if b (nrm !#T exc 1 !#F) !#F)`. We can already express such a dependency using anonymous lambda functions and by relying on Scheme's lexical scoping. The resulting expression is rather cumbersome, though:

```
($ (lambda (b)
 ($ (lambda (p) (list "beer:" b "peanuts:" p))
    (if b (nrm !#T exc 1 !#F) !#F)))
   (nrm !#F exc 1 !#T))
=> Rank Value
   0    (beer: #F peanuts: #F)
   1    (beer: #T peanuts: #T)
   2    (beer: #T peanuts: #F)
```

The *ranked let* expression generalises the `let*` expression in Scheme and provides a simpler syntax.[1] Its general form is

`(rlet* ((`$\texttt{v}_1$ $\texttt{exp}_1$`)`...`(`$\texttt{v}_n$ $\texttt{exp}_n$`)) body)` (5)

where `body` is any Scheme expression, each $\texttt{v}_i$ is a variable, and each $\texttt{exp}_i$ an expression that must return a ranking function. Each variable $\texttt{v}_i$ will take on a value from the ranking function returned by $\texttt{exp}_i$. We can express dependencies because each $\texttt{exp}_i$ may refer to the preceding variables $\texttt{v}_1, \ldots, \texttt{v}_{i-1}$. Finally, `body` may refer to all variables, and its value is returned with a rank that equals the sum of the ranks of each $\texttt{v}_i$. We can express the scenario above as follows.

```
(rlet*
 ((b (nrm !#F exc 1 !#T))
  (p (if b (nrm !#T exc 1 !#F) !#F)))
 (list "beer:" b "peanuts:" p))
=> Rank Value
   0    (beer: #F peanuts: #F)
   1    (beer: #T peanuts: #T)
   2    (beer: #T peanuts: #F)
```

We define the semantics of ranked let with a rule that specifies how (5) is recursively rewritten into another expression. This rule is as follows. Consider the general form (5).

- If $n > 1$ then the general form is rewritten into:

`($ (lambda (`$\texttt{v}_1$`)`
`(rlet* ((`$\texttt{v}_2$ $\texttt{exp}_2$`)`...`(`$\texttt{v}_n$ $\texttt{exp}_n$`)) body))`
$\texttt{exp}_1$`).`

- If $n = 1$ then the general form is rewritten into:

`($ (lambda (`$\texttt{v}_1$`) body)` $\texttt{exp}_1$`).`

Note that this rule is equivalent to the macro definition of `let*` in Scheme, with ranked procedure application substituted for regular procedure application [Abelson *et al.*, 1998].

[1] `let` and `letrec` in Scheme can be generalised similarly.

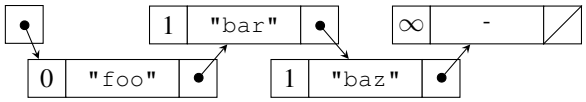


Figure 1: A ranking function as lazily-linked list. Each dot represents a promise returning the triple it points to.

4 Implementation

Semantically, every R-expression maps to a ranking function over $\mathbb{V}$. It is clear that, when computing this ranking function, we only have to compute ranks of finitely-ranked values. The remaining values, which are typically infinite in number, are then assumed to have rank ∞. But the set of finitely-ranked values may also be infinite (see e.g. the expression `(fun 1)` discussed in section 3.2). Moreover, even in the finite case, we are often only interested in values up to a given rank, since these represent the "most normal" behaviour of a program. A solution is to compute return values one by one, in increasing order with respect to rank. We can then stop once we are satisfied, like after having computed all rank 0 return values, and infinite sets of finitely-ranked values pose no problem. We call this a *least-surprising-first* execution strategy.

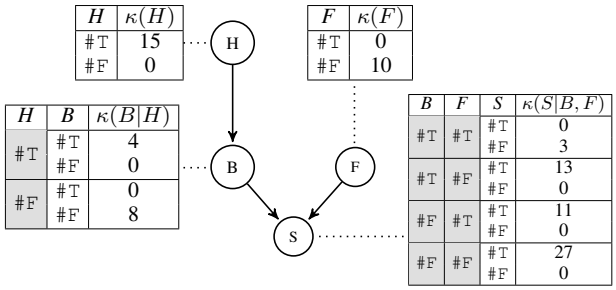
One way to achieve this is by representing ranking functions using *lazily-linked lists* constructed using *promise* objects. A promise in Scheme is constructed with the expression `(delay exp)` and captures the delayed evaluation of `exp`. If `p` is bound to the promise `(delay exp)` then `(force p)` returns whatever `exp` returns, but `exp` is evaluated *only after* `(force p)` is called, and *only once*, i.e., subsequent calls return memoised values. A ranking function can then be represented by a promise that returns a triple `(x r next)`, where `x` is a value, `r` its rank, and `next` is a promise returning another triple with a rank no less than `r`. A triple with rank ∞ marks the end of the list and its value can be ignored. Figure 1 shows how the ranking function in the first example in section 3.2 is represented. Using this approach, which forms the basis for the implementation referred to in the introduction, a return value with rank n is computed only after all rank $n-1$ return values have been computed.

5 Examples

We now demonstrate our approach by discussing a number of example problems implemented using Ranked Scheme.

5.1 A Ranking Network

Bayesian networks are compact representations of probability distributions by means of directed acyclic graphs and conditional probability tables (CPTs) [Pearl, 2014]. The equivalent in the ranking-based setting is called a *ranking network* (or sometimes an *OCF network*) [Goldszmidt and Pearl, 1996; Benferhat and Tabia, 2010]. Figure 2 depicts a ranking network for a car diagnosis scenario (taken from [Eichhorn, 2018]) that consists of four boolean variables H, B, F and S. The tables encode, like CPTs do for conditional probabilities, conditional rankings for each variable. The rank of a complete assignment of variables can be obtained by summing up the matching entries in the tables (as opposed to multiplying them, as done in the probabilistic case). For example, the rank of `H=#F, B=#T, F=#F, S=#F` is 10 since $\kappa(\texttt{H=\#F}) + k(\texttt{B=\#T}|\texttt{H=\#F}) + k(\texttt{F=\#F}) + k(\texttt{S=\#F}|\texttt{B=\#F},\texttt{F=\#F}) = 10$.



H	$\kappa(H)$
#T	15
#F	0

F	$\kappa(F)$
#T	0
#F	10

H	B	$\kappa(B\|H)$
#T	#T	4
	#F	0
#F	#T	0
	#F	8

B	F	S	$\kappa(S\|B,F)$
#T	#T	#T	0
		#F	3
#T	#F	#T	13
		#F	0
#F	#T	#T	11
		#F	0
#F	#F	#T	27
		#F	0

Figure 2: A ranking network with variables H (headlight was left on), B (battery charged), F (full tank) and S (car starts).

The function `network` defined below represents the network from Figure 2. The construction is based on a ranked let expression that sets each variable conditional on its parents and returns a ranking function over lists containing the values of the variables H, B, F and S.

```
(define (network)
 (rlet*
  ((H (nrm !#F exc 15 !#T))
   (B (if H (nrm !#F exc 4 !#T)
            (nrm !#T exc 8 !#F)))
   (F (nrm !#T exc 10 !#F))
   (S (cond [(and B F) (nrm !#T exc 3 !#F)]
            [(and B (not F)) (nrm !#F exc 13 !#T)]
            [(and (not B) F) (nrm !#F exc 11 !#T)]
            [else (nrm !#F exc 27 !#T)])))
  (list H B F S)))
(network)
=> Rank  Value
   0     (#F #T #T #T)
   3     (#F #T #T #F)
   8     (#F #F #T #F)
   10    (#F #T #F #F)
   ...   ...
```

Inference with this network can be modelled by adding observations. For example, the following expression returns the ranking over `S` given that `F` is true. Note that `third` and `fourth` are built-in Scheme functions returning the third and fourth element of a list, i.e., the value of `F` and `S`, respectively.

```
($ !fourth (observe third (network)))
=> Rank  Value
   0     #T
   3     #F
```

5.2 Boolean Circuit Diagnosis

Consider the boolean circuit shown in figure 3. We want to know which malfunctioning gates are most likely responsible when observing faulty behaviour of the circuit. This question can be answered using the function `circuit` defined below.

```
(define (circuit i1 i2 i3 o)
 ($ !cdr
  (observe
   (lambda (x) (eq? (car x) o))
   (rlet* ((N (nrm !#T exc 1 !#F))
           (O1 (nrm !#T exc 1 !#F))
           (O2 (nrm !#T exc 1 !#F))
           (l1 (if N !(not i1) !#F))
           (l2 (if O1 !(or l1 i2) !#F))
           (out (if O2 !(or l2 i3) !#F)))
         (list out N O1 O2)))))
```

The construction is again based on a ranked let expression. Here, the variables `N`, `O1` and `O2` hold the unobservable state of the gates (`#T` for good and `#F` for bad). Each gate is assumed to fail independently and exceptionally. The variables `l1`, `l2` and `out` hold the state of the respective lines (`#F` for low and `#T` for high). Their values depend on `N`, `O1` and `O2`, with a failing gate assumed to be *stuck at low*. The ranked let expression returns a ranking over lists whose `car` is the computed output and whose `cdr` is the corresponding diagnosis. (Note that `car` and `cdr` are functions returning, respectively, the head and tail of a list.) The enclosing observe expression conditionalizes on the observed output, while the enclosing call to `cdr` extracts the diagnosis, which is a list containing the inferred state of the gates `N`, `O1` and `O2` in that order.

Suppose we set `i1` and `i2` to low and `i3` to high. If the circuit functions correctly then `out` should be high. However, we observe that `out` is low. The least surprising explanation is that `O2` fails, as indicated by the rank 0 return value below. Further return values represent more surprising explanations that involve more than one failing gate.

```
(circuit #F #F #T #F)
=> Rank Value
   0    (#T #T #F)
   1    (#T #F #F)
   1    (#F #T #F)
   2    (#F #F #F)
```

5.3 A Ranked Hidden Markov Model

A Hidden Markov Model (HMM) defines a Markov process whose state is observable only indirectly, through an observable random variable whose value is determined by the state [Rabiner and Juang, 1986]. The main inference task is to determine the sequence of states that best explains a sequence of values of the observable variable. They are typically based on probabilities, with *transmission probabilities* used for state transitions, and *emission probabilities* for the values that the observable variable takes on in each state. Here we consider a ranked variant where these probabilities are replaced by ranks, and discuss its implementation.

The function `hmm` defined below implements a generic ranked HMM. It takes as input a sequence of observed values and returns a ranking function over the inferred sequences of states. To simplify the implementation we assume that both input and output are encoded as lists *in reverse temporal order*. It is assumed that the following functions, which encode the structure and parameters of the HMM, are defined: `init` (returns a ranking over initial states); `trans` (takes a state s as input and returns a ranking over the successor states of s); and `emit` (takes a state s as input and returns a ranking over values of the observable variable in state s).

```
(define (hmm obs)
 (if (empty? obs)
  ($ !list (init))
  ($ !cdr
   (observe (lambda (x) (eq? (car x) (car obs)))
            (rlet* ((p (hmm (cdr obs)))
                    (s (trans (car p)))
                    (o (emit s)))
                   (cons o (cons s p)))))))
```

The base case of this recursive function (`obs` is empty) yields a ranking over one-element sequences containing the initial state. The recursive case is based on a ranked let expression. First we determine recursively the sequence `p` of states for the `cdr` of `obs`. Then we determine the new state `s` and corresponding value `o` for the observed variable. The ranked let expression yields a ranking over lists whose `car` is a value for the observed variable and whose `cdr` is the sequence of states leading to this value. The enclosing `observe` expression conditionalizes on the observation, while the ranked procedure call of `cdr` extracts corresponding sequence of states.

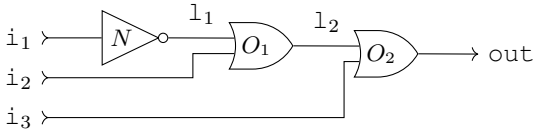


Figure 3: A boolean circuit: one *NOT* gate and two *OR* gates.

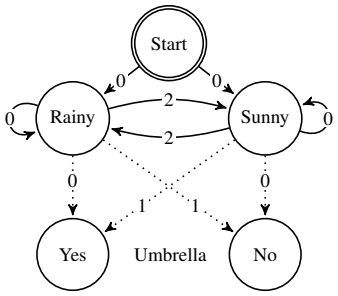


Figure 4: A ranking-based HMM. Solid arrows represent state transitions and dotted arrows represent emission ranks.

A simple toy-example of a ranking-based HMM is shown in Figure 4. The story is as follows: you are a software developer permanently locked in an office without windows. Each day your boss walks in, and you want to infer the weather based on whether he carries an umbrella. It is either rainy or sunny (initially equally likely) and you assume that the weather only exceptionally (to degree 2) changes overnight. Furthermore you assume that, if it's rainy, your boss normally carries an umbrella and exceptionally (to degree 1) not, with the situation reversed when it is sunny. This HMM is implemented by the following definitions for the functions `init`, `trans` and `emit`:

```
(define (init) (either !"rainy" or !"sunny"))
(define (trans s)
 (case s
  (("rainy") (nrm !"rainy" exc 2 !"sunny"))
  (("sunny") (nrm !"sunny" exc 2 !"rainy"))))
(define (emit s)
 (case s
  (("rainy") (nrm !"yes" exc 1 !"no"))
  (("sunny") (nrm !"no" exc 1 !"yes"))))
```

Suppose we make five observations and observe an umbrella only once. Note that a change of weather is more surprising (rank 2) than taking an umbrella on a sunny day (rank 1). Thus, the least surprising sequence is sunny every day:

```
(hmm `("no" "no" "yes" "no" "no"))
=> Rank Value
   0    (sunny sunny sunny sunny sunny)
   ...  ...
```

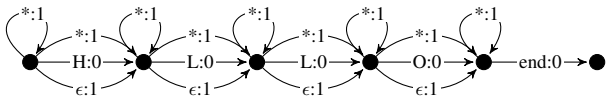


Figure 5: A ranked automaton.

Now suppose we observe an umbrella on the third, fourth and fifth day. Because taking an umbrella on a sunny day three times in a row is more surprising (rank 3) than a change of weather (rank 2) we infer that the weather has changed:

```
(hmm `("yes" "yes" "yes" "no" "no"))
=> Rank Value
   0    (rainy rainy rainy sunny sunny sunny)
   ...  ...
```

5.4 Ranked Automata for Spelling Correction

Spelling correction algorithms often involve searching for words that have a minimal *Levenshtein distance* to a given input string. Here we describe such an algorithm based on the idea of processing each character of an input string under the assumption that the character is normally correct, but exceptionally incorrect, due either to an omission, substitution or the insertion. The algorithm is based on what we call a *ranked automaton*. We just saw that Bayesian networks and HMMs can be adapted to use ranks instead of probabilities. A ranked automaton is a similar adaptation of a *probabilistic automaton* [Stoelinga, 2002]. It is defined like a nondeterministic finite automaton except that a transition between states σ, σ' takes the form $\sigma \xrightarrow{s:r} \sigma'$, where s is the generated symbol and r is the rank of the transition. A ranked automaton generates a string $(s_1, \ldots, s_n)$ with rank r if there is a path $\sigma \xrightarrow{s_1:r_1} \ldots \xrightarrow{s_n:r_n} \sigma'$ between a starting state σ and final state σ' such that $r = \sum_{i=1}^{n} r_i$.

For a given input string we can construct a ranked automaton that generates all potential corrections, ranked according to Levenshtein distance. Figure 5 shows an instance of such an automaton for the string "HLLO". Here, the symbol $*$ denotes an arbitrary character, and ϵ denotes the empty symbol. This automaton generates the string "HLLO" with rank 0, while "H*LLO" and "H*LO", which match the possible corrections "HELLO" and "HALO", are generated with rank 1, which equals their Levenshtein distance to "HLLO". To build a spelling correction algorithm based on this principle we first encode the automaton for a given input string:

```
(define (gen input)
  (if (empty? input)
   !`()
   (nrm ($ !cons !(car input) (gen (cdr input)))
    exc 1 (either (gen (cdr input))
           or ($ !cons !"*" (gen (cdr input)))
           or ($ !cons !"*" (gen input))))))
(gen (string->list "hllo"))
=> Rank Value
   0    (h l l o)
   ...   ...
   1    (h * l l o)
   1    (h * l o)
   ...   ...
```

Now suppose we have a list `dict` at our disposal that contains all possible words, and a function `match` that takes a string S as input and returns a list containing the words from `dict` that match S, with $\star$ acting as a wildcard. The function `correct` defined below returns a list of corrections for an input string, ranked according to Levenshtein distance.

```
(define (correct input)
 ($ !match
  (observe
   (lambda (x) (not (empty? (match x))))
   (gen (string->list input)))))
(correct "hllo")
=> Rank Value
   0    ("hello")
   0    ("halo")
   ...   ...
```

Efficiency can be improved by ruling out intermediate results not matching any prefix of a word in the dictionary. This can be done by adding a conditionalization step to `gen`.

6 Discussion and Conclusion

Probability is arguably the gold standard when it comes to representing uncertainty in AI. There are, however, problems which involve uncertainty of a different nature. We focus on the case where we know what is normal and what is exceptional, but where the probabilistic meaning of these terms is unknown or irrelevant. A suitable formalism to reason under this type of uncertainty is ranking theory. We have applied the methodology of PPLs to ranking theory, and developed a programming language aimed at building models involving uncertainty of this kind and performing inference on them.

The application of ranking theory in AI is not new. It has been used as a theoretical basis for belief revision and nonmonotonic reasoning [Pearl, 1990; Goldszmidt and Pearl, 1996; Darwiche and Pearl, 1996; Kern-Isberner, 2001]. Our work is based on the idea that ranking theory can be used as a general tool—like probability theory—to build arbitrary models involving uncertainty. The ranked programming approach makes it easy to build such models, and thus paves the way for new applications of ranking theory.

Besides ranking theory, there are other alternatives to probability theory. These include Dempster-Shafer functions, possibility measures, and plausibility measures (see [Halpern, 2017] for an overview and comparison, which includes ranking theory). Possibility measures measure uncertainty on an arbitrary totally ordered scale, such as the integer scale from 0 to ∞ (making them equivalent to ranking functions) or real numbers from the interval $[0, 1]$. The latter is often chosen to model vagueness or fuzziness. Possibility theory has been used to equip logic programming and answer set programming with the ability to deal with uncertainty [Nicolas *et al.*, 2006; Bauters *et al.*, 2010; Alsinet and Godo, 2000].

As a final remark we note that ranked programming generalises nondeterministic programming. For example, if we distinguish only finite from infinite ranks, ranked choice and observation in Ranked Scheme work like the `amb` and `require` statements discussed in [Abelson *et al.*, 1985].

Acknowledgements

The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grant KE 1686/3-1).

References

[Abelson *et al.*, 1985] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[Abelson *et al.*, 1998] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, Robert Halstead, et al. Revised 5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[Alsinet and Godo, 2000] Teresa Alsinet and Lluis Godo. A complete calculus for possibilistic logic programming with fuzzy propositional variables. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 1–10. Morgan Kaufmann Publishers Inc., 2000.

[Bauters *et al.*, 2010] Kim Bauters, Steven Schockaert, Martine De Cock, and Dirk Vermeir. Possibilistic answer set programming revisited. In *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 48–55, 2010.

[Benferhat and Tabia, 2010] Salem Benferhat and Karim Tabia. Belief change in ocf-based networks in presence of sequences of observations and interventions: Application to alert correlation. In Byoung-Tak Zhang and Mehmet A. Orgun, editors, *PRICAI 2010: Trends in Artificial Intelligence, 11th Pacific Rim International Conference on Artificial Intelligence, Daegu, Korea, August 30-September 2, 2010. Proceedings*, volume 6230 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2010.

[Carpenter *et al.*, 2017] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A Probabilistic Programming Language. *Journal of Statistical Software*, 76(1), 2017.

[Darwiche and Pearl, 1996] Adnan Darwiche and Judea Pearl. On the logic of iterated belief revision. *Artificial intelligence*, 89(1-2):1–29, 1996.

[Dybvig, 2009] R Kent Dybvig. *The SCHEME programming language*. Mit Press, 2009.

[Eichhorn, 2018] Christian Eichhorn. *Rational Reasoning with Finite Conditional Knowledge Bases: Theoretical and Implementational Aspects*. Springer, 2018.

[Goldszmidt and Pearl, 1996] Moisés Goldszmidt and Judea Pearl. Qualitative probabilities for default reasoning, belief revision, and causal modeling. *Artificial Intelligence*, 84(1):57–112, 1996.

[Goodman *et al.*, 2008] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In David A. McAllester and Petri Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229. AUAI Press, 2008.

[Halpern, 2017] Joseph Y Halpern. *Reasoning about uncertainty*. MIT press, 2017.

[Kern-Isberner, 2001] Gabriele Kern-Isberner. *Conditionals in nonmonotonic reasoning and belief revision: considering conditionals as agents*. Springer-Verlag, 2001.

[Mansinghka *et al.*, 2014] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.

[Nicolas *et al.*, 2006] Pascal Nicolas, Laurent Garcia, Igor Stéphan, and Claire Lefèvre. Possibilistic uncertainty handling for answer set programming. *Ann. Math. Artif. Intell.*, 47(1-2):139–181, 2006.

[Pearl, 1990] Judea Pearl. System Z: A natural ordering of defaults with tractable applications to nonmonotonic reasoning. In Rohit Parikh, editor, *Proceedings of the 3rd Conference on Theoretical Aspects of Reasoning about Knowledge, Pacific Grove, CA, March 1990*, pages 121–135. Morgan Kaufmann, 1990.

[Pearl, 2014] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[Pfeffer, 2009] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

[Rabiner and Juang, 1986] Lawrence R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3:4–16, 1986.

[Rienstra, 2017] Tjitze Rienstra. Rankpl: A qualitative probabilistic programming language. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 470–479. Springer, 2017.

[Spohn, 2009] Wolfgang Spohn. A survey of ranking theory. In *Degrees of belief*, pages 185–228. Springer, 2009.

[Spohn, 2014] Wolfgang Spohn. *The Laws of Belief - Ranking Theory and Its Philosophical Applications*. Oxford University Press, 2014.

[Stoelinga, 2002] Mariëlle Stoelinga. An introduction to probabilistic automata. *Bulletin of the EATCS*, 78(176-198):2, 2002.

[Wood *et al.*, 2014] Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, pages 1024–1032, 2014.