RankPL: A Qualitative Probabilistic Programming Language

Tjitze Rienstra

University of Luxembourg
tjitze@gmail.com

Abstract. In this paper we introduce *RankPL*, a modeling language that can be thought of as a qualitative variant of a probabilistic programming language with a semantics based on Spohn's ranking theory. Broadly speaking, RankPL can be used to represent and reason about processes that exhibit uncertainty expressible by distinguishing "normal" from "surprising" events. RankPL allows (iterated) revision of rankings over alternative program states and supports various types of reasoning, including abduction and causal inference. We present the language, its denotational semantics, and a number of practical examples. We also discuss an implementation of RankPL that is available for download.

1 Introduction

Probabilistic programming languages (PPLs) are programming languages extended with statements to (1) draw values at random from a given probability distribution, and (2) perform conditioning due to observation. Probabilistic programs yield, instead of a deterministic outcome, a probability distribution over possible outcomes. PPLs greatly simplify representation of, and reasoning with rich probabilistic models. Interest in PPLs has increased in recent years, mainly in the context of Bayesian machine learning. Examples of modern PPLs include *Church*, *Venture* and *Figaro* [4,8,10], while early work goes back to Kozen [6].

Ranking theory is a qualitative abstraction of probability theory in which events receive discrete degrees of surprise called *ranks* [11]. That is, events are ranked 0 (not surprising), 1 (surprising), 2 (very surprising), and so on, or ∞ if impossible. Apart from being computationally simpler, ranking theory permits meaningful inference without requiring precise probabilities. Still, it provides analogues to powerful notions known from probability theory, like conditioning and independence. Ranking theory has been applied in logic-based AI (e.g. belief revision and non-monotonic reasoning [1,3]) as well as formal epistemology [11].

In this paper we develop a language called *RankPL*. Semantically, the language draws a parallel with probabilistic programming in terms of ranking theory. We start with a minimal imperative programming language (**if-then-else**, **while**, etc.) and extend it with statements to (1) draw choices at random from a given ranking function and (2) perform ranking-theoretic conditioning due to observation. Analogous to probabilistic programs, a RankPL programs yields, instead of a deterministic outcome, a ranking function over possible outcomes.

Broadly speaking, RankPL can be used to represent and reason about processes whose input or behavior exhibits uncertainty expressible by distinguishing normal (rank 0) from surprising (rank > 0) events. Conditioning in RankPL amounts to the (iterated) revision of rankings over alternative program states. This is a form of revision consistent with the well-known AGM and DP postulates for (iterated) revision [1,2]. Various types of reasoning can be modeled, including abduction and causal inference. Like with PPLs, these reasoning tasks can be modeled without having to write inference-specific code.

The overview of this paper is as follows. Section 2 deals with the basics of ranking theory. In section 3 we introduce RankPL and present its syntax and formal semantics. In section 4 we discuss two generalized conditioning schemes (L-conditioning and J-conditioning) and show how they can be implemented in RankPL. All the above will be demonstrated by practical examples. In section 5 we discuss our RankPL implementation. We conclude in section 6.

2 Ranking Theory

Here we present the necessary basics of ranking theory, all of which is due to Spohn [11]. The definition of a *ranking function* presupposes a finite set Ω of possibilities and a boolean algebra $\mathcal{A}$ over subsets of Ω, which we call *events*.

Definition 1. *A ranking function is a function $\kappa : \Omega \to \mathbb{N} \cup \{\infty\}$ that associates every possibiltiy with a* rank*. κ is extended to a function over events by defining $\kappa(\emptyset) = \infty$ and $\kappa(A) = min(\{\kappa(w) \mid w \in A\})$ for each $A \in \mathcal{A} \setminus \emptyset$. A ranking function must satisfy $\kappa(\Omega) = 0$.*

As mentioned in the introduction, ranks can be understood as degrees of surprise or, alternatively, as inverse degrees of plausibility. The requirement that $\kappa(\Omega) = 0$ is equivalent to the condition that at least one $w \in \Omega$ receives a rank of 0. We sometimes work with functions $\lambda : \Omega \to \mathbb{N} \cup \{\infty\}$ that violate this condition. The *normalization* of λ is a ranking function denoted by $||\lambda||$ and defined by $||\lambda||(w) = \lambda(w) - \lambda(\Omega)$. Conditional ranks are defined as follows.

Definition 2. *Given a ranking function κ, the rank of A conditional on B (denoted $\kappa(A \mid B)$) is defined by*

$$\kappa(A \mid B) = \begin{cases} \kappa(A \cap B) - \kappa(B) & \text{if } \kappa(B) \neq \infty, \\ \infty & \text{otherwise.} \end{cases}$$

We denote by κ_B the ranking function defined by $\kappa_B(A) = \kappa(A \mid B)$.

In words, the effect of conditioning on B is that the rank of B is shifted down to zero (keeping the relative ranks of the possibilities in B constant) while the rank of its complement is shifted up to ∞.

How do ranks compare to probabilities? An important difference is that ranks of events do not add up as probabilities do. That is, if A and B are disjoint, then $\kappa(A \cup B) = min(\kappa(A), \kappa(B))$, while $P(A \cup B) = P(A) + P(B)$. This is, however,

consistent with the interpretation of ranks as degrees of surprise (i.e., $A \cup B$ is no less surprising than A or B). Furthermore, ranks provide deductively closed beliefs, whereas probabilities do not. More precisely, if we say that A is believed with firmness x (for some $x > 0$) with respect to κ iff $\kappa(\overline{A}) > x$, then if A and B are believed with firmness x then so is $A \cap B$. A similar notion of belief does not exist for probabilities, as is demonstrated by the Lottery paradox [7].

Finally, note that ∞ and 0 in ranking theory can be thought of as playing the role of 0 and 1 in probability, while min, $-$ and $+$ play the role, respectively, of $+$, $\div$ and $\times$. Recall, for example, the definition of conditional probability, and compare it with definition 2. This correspondence also underlies notions such as (conditional) independence and ranking nets (the ranking-based counterpart of Bayesian networks) that have been defined in terms of rankings [11].

3 RankPL

We start with a brief overview of the features of RankPL. The basis is a minimal imperative language consisting of integer-typed variables, an **if-then-else** statement and a **while-do** construct. We extend it with the two special statements mentioned in the introduction. We call the first one *ranked choice*. It has the form $\{s_1\}\ \langle e \rangle\ \{s_2\}$. Intuitively, it states that either s_1 or s_2 is executed, where the former is a normal (rank 0) event and the latter a typically surprising event whose rank is the value of the expression e. Put differently, it represents a draw of a statement to be executed, at random, from a ranking function over two choices. Note that we can set e to zero to represent a draw from two equally likely choices, and that larger sets of choices can be represented through nesting.

The second special statement is called the *observe* statement **observe** b. It states that the condition b is observed to hold. Its semantics corresponds to ranking-theoretic conditioning. To illustrate, consider the program

$$\mathtt{x} := 10;\ \{\mathtt{y} := 1\}\ \langle 1 \rangle\ \{\{\mathtt{y} := 2\}\ \langle 1 \rangle\ \{\mathtt{y} := 3\}\};\ \mathtt{x} := \mathtt{x} \times \mathtt{y};$$

This program has three possible outcomes: $\mathtt{x} = 10$, $\mathtt{x} = 20$ and $\mathtt{x} = 30$, ranked 0, 1 and 2, respectively. Now suppose we extend the program as follows:

$$\mathtt{x} := 10;\ \{\mathtt{y} := 1\}\ \langle 1 \rangle\ \{\{\mathtt{y} := 2\}\ \langle 1 \rangle\ \{\mathtt{y} := 3\}\};\ \textbf{observe}\ \mathtt{y} > 1;\ \mathtt{x} := \mathtt{x} \times \mathtt{y};$$

Here, the observation rules out the event $\mathtt{y} = 1$, and the ranks of the remaining possibilities are shifted down, resulting in two outcomes $\mathtt{x} = 20$ and $\mathtt{x} = 30$, ranked 0 and 1, respectively.

A third special construct is the *rank expression* **rank** b., which evaluates to the rank of the boolean expression b. Its use will be demonstrated later.

3.1 Syntax

We fix a set *Vars* of variables (ranged over by x) and denote by *Val* the set of integers including ∞ (ranged over by n). We use e, b and s to range over the numerical expressions, boolean expressions, and statements. They are defined by the following BNF rules:

$$
\begin{aligned}
e &:= n \mid x \mid \textbf{rank } b \mid (e_1 \lozenge e_2) \text{ (for } \lozenge \in \{-, +, \times, \div\}) \\
b &:= \neg b \mid (b_1 \vee b_2) \mid (e_1 \blacklozenge e_2) \text{ (for } \blacklozenge \in \{=, <\}) \\
s &:= \{s_0; s_1\} \mid x := e \mid \textbf{if } b \textbf{ then } \{s_1\} \textbf{ else } \{s_2\} \mid \\
&\quad \textbf{while } b \textbf{ do } \{s\} \mid \{s_1\} \langle e \rangle \{s_2\} \mid \textbf{observe } b \mid \textbf{skip}
\end{aligned}
$$

We omit parentheses and curly brackets when possible and define $\wedge$ in terms of $\vee$ and $\neg$. We write **if** b **then** $\{s\}$ instead of **if** b **then** $\{s\}$ **else** $\{\textbf{skip}\}$, and abbreviate statements of the form $\{x := e_1\} \langle e \rangle \{x := e_2\}$ to $x := e_1 \langle e \rangle e_2$. Note that the **skip** statement does nothing and is added for technical convenience.

3.2 Semantics

The denotational semantics of RankPL defines the meaning of a statement s as a function $D[\![s]\!]$ that maps prior rankings into posterior rankings. The subjects of these rankings are program states represented by *valuations*, i.e., functions that assign values to all variables. The *initial valuation*, denoted by σ_0, sets all variables to 0. The *initial ranking*, denoted by κ_0, assigns 0 to σ_0 and ∞ to others. We denote by $\sigma[x \to n]$ the valuation equivalent to σ except for assigning n to x.

From now on we associate Ω with the set of valuations and denote the set of rankings over Ω by K. Intuitively, if $\kappa(\sigma)$ is the degree of surprise that σ is the actual valuation *before* executing s, then $D[\![s]\!](\kappa)(\sigma)$ is the degree of surprise that σ is the actual valuation *after* executing s. If we refer to the result of running the *program* s, we refer to the ranking $D[\![s]\!](\kappa_0)$. Because s might not execute successfully, $D[\![s]\!]$ is not a total function over K. There are two issues to deal with. First of all, non-termination of a loop leads to an undefined outcome. Therefore $D[\![s]\!]$ is a partial function whose value $D[\![s]\!](\kappa)$ is defined only if s terminates given κ. Secondly, observe statements may rule out all possibilities. A program whose outcome is empty because of this is said to *fail*. We denote failure with a special ranking κ_∞ that assigns ∞ to all valuations. Since $\kappa_\infty \notin K$, we define the range of $D[\![s]\!]$ by $K^* = K \cup \{\kappa_\infty\}$. Thus, the semantics of a statement s is defined by a partial function $D[\![s]\!]$ from K^* to K^*.

But first, we define the semantics of expressions. A numerical expression is evaluated w.r.t. both a ranking function (to determine values of **rank** expressions) and a valuation (to determine values of variables). Boolean expressions may also contain **rank** expressions and therefore also depend on a ranking function. Given a valuation σ and ranking κ, we denote by $\sigma_\kappa(e)$ the value of the numerical expression e w.r.t. σ and κ, and by $[b]_\kappa$ the set of valuations satisfying the boolean expression b w.r.t. κ. These functions are defined as follows.[1]

$$
\begin{aligned}
\sigma_\kappa(n) &= n \\
\sigma_\kappa(x) &= \sigma(x) \\
\sigma_\kappa(\textbf{rank } b) &= \kappa([b]_\kappa) \\
\sigma_\kappa(a_1 \lozenge a_2) &= \sigma_\kappa(a_1) \lozenge \sigma_\kappa(a_2)
\end{aligned}
\qquad\qquad
\begin{aligned}
[\neg b]_\kappa &= \Omega \setminus [b]_\kappa \\
[b_1 \vee b_2]_\kappa &= [b_1]_\kappa \cup [b_2]_\kappa \\
[a_1 \blacklozenge a_2]_\kappa &= \{\sigma \in \Omega \mid \sigma_\kappa(a_1) \blacklozenge \sigma_\kappa(a_2)\}
\end{aligned}
$$

[1] We omit explicit treatment of undefined operations (i.e. division by zero and some operations involving ∞). They lead to program termination.

Given a boolean expression b we will write $\kappa(b)$ as shorthand for $\kappa([b]_\kappa)$ and κ_b as shorthand for $\kappa_{[b]_\kappa}$. We are now ready to define the semantics of statements. It is captured by seven rules, numbered **D1** to **D7**. The first deals with the **skip** statement, which does nothing and therefore maps to the identity function.

$$D[\![\textbf{skip}]\!](\kappa) = \kappa. \tag{\textbf{D1}}$$

The meaning of $s_1; s_2$ is the composition of $D[\![s_1]\!]$ and $D[\![s_2]\!]$.

$$D[\![s_1; s_2]\!](\kappa) = D[\![s_2]\!](D[\![s_1]\!](\kappa)) \tag{\textbf{D2}}$$

The rank of a valuation σ after executing an assignment $x := e$ is the minimum of all ranks of valuations that equal σ after assigning the value of e to x.

$$D[\![x := e]\!](\kappa)(\sigma) = \kappa(\{\sigma' \in \Omega \mid \sigma = \sigma'[x \to \sigma'_\kappa(e)]\}) \tag{\textbf{D3}}$$

To execute **if** b **then** $\{s_1\}$ **else** $\{s_2\}$ we first execute s_1 and s_2 conditional on b and $\neg b$, yielding the rankings $D[\![s_1]\!](\kappa_b)$ and $D[\![s_2]\!](\kappa_{\neg b})$. These are adjusted by adding the prior ranks of b and $\neg b$ and combined by taking the minimum of the two. The result is normalized to account for the case where one branch fails.

$$D[\![\textbf{if } e \textbf{ then } \{s_1\} \textbf{ else } \{s_2\}]\!](\kappa) = ||\lambda||, \quad \text{where } \lambda(\sigma) = min \begin{pmatrix} D[\![s_1]\!](\kappa_b)(\sigma) + \kappa(b), \\ D[\![s_2]\!](\kappa_{\neg b})(\sigma) + \kappa(\neg b) \end{pmatrix} \tag{\textbf{D4}}$$

Given a prior κ, the rank of a valuation after executing $s_1 \langle e \rangle s_2$ is the minimum of the ranks assigned by $D[\![s_1]\!](\kappa)$ and $D[\![s_2]\!](\kappa)$, where the latter is increased by e. The result is normalized to account for the case where one branch fails.

$$D[\![\{s_1\} \langle e \rangle \{s_2\}]\!](\kappa) = ||\lambda||, \text{ where } \lambda(\sigma) = min \begin{pmatrix} D[\![s_1]\!](\kappa)(\sigma), \\ D[\![s_2]\!](\kappa)(\sigma) + \sigma_\kappa(e) \end{pmatrix} \tag{\textbf{D5}}$$

The semantics of **observe** b corresponds to conditioning on the set of valuations satisfying b, unless the rank of this set is ∞ or the prior ranking equals κ_∞.

$$D[\![\textbf{observe } b]\!](\kappa) = \begin{cases} \kappa_\infty \text{ if } \kappa = \kappa_\infty \text{ or } \kappa(b) = \infty, \text{ or} \\ \kappa_b \quad \text{otherwise.} \end{cases} \tag{\textbf{D6}}$$

We define the semantics of **while** b **do** $\{s\}$ as the iterative execution of **if** b **then** $\{s\}$ **else** $\{\textbf{skip}\}$ until the rank of b is ∞ (the loop terminates normally) or the result is undefined (s does not terminate). If neither of these conditions is ever met (i.e., if the **while** statement loops endlessly) then the result is undefined.

$$D[\![\textbf{while } b \textbf{ do } \{s\}]\!](\kappa) = \begin{cases} F^n_{b,s}(\kappa) \text{ for the first } n \text{ s.t. } F^n_{b,s}(\kappa)(b) = \infty, \text{ or} \\ \text{undef. if there is no such } n, \end{cases} \tag{\textbf{D7}}$$

where $F_{b,s} : K^* \to K^*$ is defined by $F_{b,s}(\kappa) = D[\![\textbf{if } b \textbf{ then } \{s\} \textbf{ else } \{\textbf{skip}\}]\!](\kappa)$.

Some remarks. Firstly, the semantics of RankPL can be thought of as a ranking-based variation of the Kozen's semantics of probabistic programs [6] (i.e.,

replacing $\times$ with $+$ and $+$ with min). Secondly, a RankPL implementation does not need to compute complete rankings. Our implementation discussed in section 5 follows a *most-plausible-first* execution strategy: different alternatives are explored in ascending order w.r.t. rank, and higher-ranked alternatives need not be explored if knowing the lowest-ranked outcomes is enough, as is often the case.

Example Consider the *two-bit full adder* circuit shown in figure 1. It contains two *XOR* gates X_1, X_2, two *AND* gates A_1, A_2 and an *OR* gate O_1. The function of this circuit is to generate a binary representation $(\mathtt{b}_1, \mathtt{b}_2)$ of the number of inputs among $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$ that are high. The *circuit diagnosis problem* is about explaining observed incorrect behavior by finding minimal sets of gates that, if faulty, cause this behavior.[2]

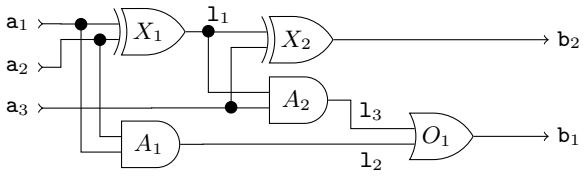


Fig. 1: A two-bit full adder

The listing below shows a RankPL solution. On line 1 we set the constants L and H (representing a *low* and *high* signal); and OK and FAIL (to represent the state of a gate). Line 2 encodes the space of possible inputs (L or H, equally likely). The failure variables $\mathtt{fa}_1$, $\mathtt{fa}_2$, $\mathtt{fo}_1$, $\mathtt{fx}_2$ and $\mathtt{fx}_2$ represent the events of individual gates failing and are set on line 3. Here, we assume that failure is surprising to degree 1. The circuit's logic is encoded on lines 4-8, where the output of a failing gate is arbitrarily set to L or H. Note that $\oplus$ stands for XOR. At the end we observe ϕ.

```
1 L := 0; H := 1; OK := 0; FAIL := 1;
2 a1 := (L ⟨0⟩ H); a2 := (L ⟨0⟩ H); a3 := (L ⟨0⟩ H);
3 fx1 := (OK ⟨1⟩ FAIL); fx2 := (OK ⟨1⟩ FAIL); fa1 := (OK ⟨1⟩ FAIL);
      fa2 := (OK ⟨1⟩ FAIL); fo1 := (OK ⟨1⟩ FAIL);
4 if fx1 = OK then l1 := a1 ⊕ a2 else l1 := (L ⟨0⟩ H);
5 if fa1 = OK then l2 := a1 ∧ a2 else l2 := (L ⟨0⟩ H);
6 if fa2 = OK then l3 := l1 ∧ a3 else l3 := (L ⟨0⟩ H);
7 if fx2 = OK then b2 := l1 ⊕ a3 else b2 := (L ⟨0⟩ H);
8 if fo1 = OK then b1 := l3 ∨ l2 else b1 := (L ⟨0⟩ H);
9 observe φ;
```

The different valuations of the failure variables produced by this program represent explanations for the observation ϕ, ranked according to plausibility. Suppose we observe that the input $(\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3)$ is valued (L,L,H) while the

[2] See Halpern [5, Chapter 9] for a similar treatment of this example.

output $(\mathtt{b}_1, \mathtt{b}_2)$ is incorrectly valued (H,L) instead of (L,H). Thus, we set ϕ to $\mathtt{a}_1 = \mathtt{L} \wedge \mathtt{a}_2 = \mathtt{L} \wedge \mathtt{a}_3 = \mathtt{H} \wedge \mathtt{b}_1 = \mathtt{H} \wedge \mathtt{b}_2 = \mathtt{L}$. The program then produces one outcome ranked 0, namely $(\mathtt{fa}_1, \mathtt{fa}_2, \mathtt{fo}_1, \mathtt{fx}_2, \mathtt{fx}_2) = (\mathtt{OK}, \mathtt{OK}, \mathtt{OK}, \mathtt{FAIL}, \mathtt{OK})$. That is, ϕ is most plausibly explained by failure of X_1. Other outcomes are ranked higher than 0 and represent explanations involving more than one faulty gate.

4 Noisy Observation and Iterated Revision

Conditioning by A means that A becomes believed with infinite firmness. This is undesirable if we have to deal with iterated belief change or noisy or untrustworthy observations, since we cannot, after conditioning on A, condition on events inconsistent with A. *J-conditioning* [3] is a more general form of conditioning that addresses this problem. It is parametrized by a rank x that indicates the firmness with which the evidence must be believed.

Definition 3. *Let $A \subseteq \Omega$, κ a ranking function over Ω such that $\kappa(A), \kappa(\overline{A}) < \infty$, and x a rank. The* J-conditioning *of κ by A with strength x, denoted by $\kappa_{A \to x}$, is defined by $\kappa_{A \to x}(B) = min(\kappa(B|A), \kappa(B|\overline{A}) + x)$.*

In words, the effect of J-conditioning by A with strength x is that A becomes believed with firmness x. This permits iterated belief change, because the rank of $\overline{A}$ is shifted up only by a finite number of ranks and hence can be shifted down again afterwards. Instead of introducing a special statement representing J-conditioning, we show that we can already express it in RankPL, using ranked choice and observation as basic building blocks. Below we write $\kappa_{b \to x}$ as shorthand for $\kappa_{[b]_\kappa \to x}$. Proofs are omitted due to space constraints.

Theorem 1. *Let b be a boolean expression and κ a ranking function s.t. $\kappa(b) < \infty$ and $\kappa(\neg b) < \infty$. We then have $\kappa_{b \to x} = D[\![\{\textbf{observe } b\} \langle x \rangle \{\textbf{observe } \neg b\}]\!](\kappa)\}$.*

L-conditioning [3] is another kind of generalized conditioning. Here, the parameter x characterizes the 'impact' of the evidence.

Definition 4. *Let $A \subseteq \Omega$, κ a ranking function over Ω such that $\kappa(A), \kappa(\overline{A}) < \infty$, and x a rank. The* L-conditioning *of κ by A with strength x is denoted by $\kappa_{A \uparrow x}$ and is defined by $\kappa_{A \uparrow x}(B) = min(\kappa(A \cap B) - y, \kappa(\neg A \cap B) + x - y)$, where $y = min(\kappa(A), x)$.*

Thus, L-conditioning by A with strength x means that A improves by x ranks w.r.t. the rank of $\neg A$. Unlike J-conditioning, L-conditioning satisfies two properties that are desirable for modeling noisy observation: *reversibility* ($(\kappa_{A \uparrow x})_{\overline{A} \uparrow x} = \kappa$) and *commutativity* ($(\kappa_{A \uparrow x})_{B \uparrow x} = (\kappa_{B \uparrow x})_{A \uparrow x}$) [11]. We can expression L-conditioning in RankPL using ranked choice, observation and the rank expression as basic building blocks. Like before, we write $\kappa_{b \uparrow x}$ to denote $\kappa_{[b]_\kappa \uparrow x}$.

Theorem 2. *Let b be a boolean expression, κ a ranking function over Ω such that $\kappa(b), \kappa(\neg b) < \infty$, and x a rank. We then have:*

$$\kappa_{b\uparrow x} = D \left[\!\!\left[\begin{array}{c} \textbf{if } (\textbf{rank}(b) \leq x) \textbf{ then} \\ \{\textbf{observe } b\} \ \langle x - \textbf{rank}(b) + \textbf{rank}(\neg b)\rangle \ \{\textbf{observe } \neg b\} \\ \textbf{else} \\ \{\textbf{observe } \neg b\} \ \langle \textbf{rank}(b) - x\rangle \ \{\textbf{observe } b\} \end{array} \right]\!\!\right] (\kappa)$$

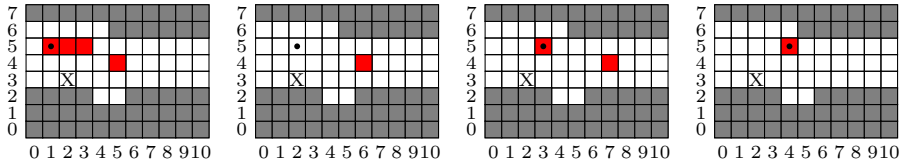
In what follows we use the statement $\textbf{observe}_{\mathbf{L}}(x, b)$ as shorthand for the statement that represents L-conditioning as defined in theorem 2.

Example This example involves both iterated revision and noisy observation. A robot navigates a grid world and has to determine its location using a map and two distance sensors. Figure 2 depicts the map that we use. Gray cells represent walls and other cells are empty (ignoring, for now, the red cells and dots). The sensors (oriented north and south) measure the distance to the nearest wall or obstacle. To complicate matters, the sensor readings might not match the map. For example, the X in figure 2 marks an obstacle that affects sensor readings, but as far as the robot knows, this cell is empty.

The listing below shows a RankPL solution. The program takes as input: (1) A map, held by an array `map` of size `m` × `n`, storing the content of each cell (0 = empty, 1 = wall); (2) An array `mv` (length `k`) of movements (`N`/`E`/`S`/`W` for north/east/south/west) at given time points; and (3) Two arrays `ns` and `ss` (length `k`) with distances read by the north and south sensor at given time points. Note that, semantically, arrays are just indexed variables.

```
Input: k: number of steps to simulate
Input: mv: array (size ≥ k) of movements (N/E/S/W)
Input: ns and ss: arrays (size ≥ k) of north and south sensor readings
Input: map: 2D array (size m × n) encoding the map
 1 t := 0;  x := 0 ⟨0⟩ {1 ⟨0⟩ {2 ⟨0⟩ ...m}};  y := 0 ⟨0⟩ {1 ⟨0⟩ {2 ⟨0⟩ ...n}};
 2 while (t < k) do {
 3       if (mv[t] = N) then y := y + 1
 4       else if (mv[t] = S) then y := y − 1
 5       else if (mv[t] = W) then x := x − 1
 6       else if (mv[t] = E) then x := x + 1 else skip;
 7       nd := 0;  while map[x][y + nd + 1] = 0 do nd := nd + 1;
 8       observe_L(1, nd = ns[t]);
 9       sd := 0;  while map[x][y − sd − 1] = 0 do sd := sd + 1;
10       observe_L(1, sd = ss[t]);
11       t := t + 1;
12 }
```

The program works as follows. On line 1 the time point `t` is set to 0 and the robot's location (`x`, `y`) is set to a randomly chosen coordinate (all equally likely) using nested ranked choice statements. Inside the **while** loop, which iterates over `t`, we first process the movement `mv`[`t`] (lines 3-6). We then process (lines 7-8) the north sensor reading, by counting empty cells between the robot and nearest wall, the result of which is observed to equal `ns`[`t`]—and likewise for the



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$ (d) $k = 4$

Fig. 2: Most plausible inferred locations during four iterations

south sensor (lines 9-10). We use L-conditioning with strength 1 to account for possible incorrect observations. On line 11 we update `t`.

Suppose we want to simulate a movement from $(0, 5)$ to $(4, 5)$. Thus, we use the inputs $\texttt{mv} = \{E, E, E, E\}$, $\texttt{ns} = \{1, 1, 1, 1\}$, $\texttt{ss} = \{2, 1, 2, 3\}$, while `map` is set as shown in figure 2 (i.e., 1 for every cell containing a wall, 0 for every other cell). Note that the observed distances stored in `ns` and `ss` are consistent with the distances observed along this path, where, at $t = 1$, the south sensor reads a distance of 1 instead of 2, due to the obstacle X.

The different values of `x, y` generated by this program encode possible locations of the robot, ranked according to plausibility. The dots in figure 2 show the actual locations, while the red cells represent the inferred most plausible (i.e., rank zero) locations generated by the program. Terminating after $t = 0$ (i.e., setting k to 1) yields four locations, all consistent with the observed distances 1 (north) and 2 (south). If we terminate after $t = 1$, the robot wrongly believes to be at (6, 4), due to having observed the obstacle. However, if we terminate after $t = 3$, the program produces the actual location.

Note that using L-conditioning here is essential. Regular conditioning would cause failure after the third iteration. We could also have used J-conditioning, which gives different rankings of intermediate results.

5 Implementation

A RankPL interpreter written in Java can be found at `http://github.com/tjitze/RankPL`. It runs programs written using the syntax described in this paper, or constructed using Java classes that map to this syntax. The latter makes it possible to embed RankPL programs inside Java code and to make it interact with and use classes and methods written Java. The interpreter is faithful to the semantics described in section 3 and implements the *most-plausible-first* execution strategy discussed in section 3.2. All examples discussed in this paper are included, as well as a number of additional examples.

6 Conclusion and Future Work

We have introduced RankPL, a language semantically similar to probabilistic programming, but based on Spohn's ranking theory, and demonstrated its utility

using examples involving abduction and iterated revision. We believe that the approach has great potential for applications where PPLs are too demanding due to their computational complexity and dependence on precise probability values. Moreover, we hope that our approach will generate a broader and more practical scope for the topics of ranking theory and belief revision which, in the past, have been studied mostly from purely theoretical perspectives.

A number of aspects were not touched upon and will be addressed in future work. This includes a more fine grained characterization of termination and a discussion of the relationship with nondeterministic programming, which is a special case of RankPL. Furthermore, we have omitted examples to show that RankPL subsumes ranking networks and can be used to reason about causal rules and actions [3]. We also did not contrast our approach with default reasoning formalisms that use ranking theory as a semantic foundation (see, e.g., [9]).

Even though we demonstrated that RankPL is expressive enough to solve fairly complex tasks in a compact manner, it is a very basic language that is best regarded as proof of concept. In principle, the approach can be applied to any programming language, whether object-oriented, functional, or LISP-like. Doing so would make it possible to reason about ranking-based models expressed using, for example, recursion and complex data structures. These features are also supported by PPLs such as Church [4], Venture [8] and Figaro [10].

References

1. Darwiche, A., Pearl, J.: On the logic of iterated belief revision. Artificial intelligence 89(1-2), 1–29 (1996)
2. Gärdenfors, P., Rott, H.: Handbook of logic in artificial intelligence and logic programming (vol. 4). chap. Belief Revision, pp. 35–132. Oxford University Press, Oxford, UK (1995)
3. Goldszmidt, M., Pearl, J.: Qualitative probabilities for default reasoning, belief revision, and causal modeling. Artificial Intelligence 84(1), 57–112 (1996)
4. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: McAllester, D.A., Myllymäki, P. (eds.) UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008. pp. 220–229. AUAI Press (2008)
5. Halpern, J.Y.: Reasoning about uncertainty. MIT Press (2005)
6. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
7. Kyburg Jr, H.E.: Probability and the logic of rational belief (1961)
8. Mansinghka, V.K., Selsam, D., Perov, Y.N.: Venture: a higher-order probabilistic programming platform with programmable inference. CoRR abs/1404.0099 (2014)
9. Pearl, J.: System Z: A natural ordering of defaults with tractable applications to nonmonotonic reasoning. In: Parikh, R. (ed.) Proceedings of the 3rd Conference on Theoretical Aspects of Reasoning about Knowledge, Pacific Grove, CA, March 1990. pp. 121–135. Morgan Kaufmann (1990)
10. Pfeffer, A.: Figaro: An object-oriented probabilistic programming language. Charles River Analytics Technical Report 137 (2009)
11. Spohn, W.: The Laws of Belief - Ranking Theory and Its Philosophical Applications. Oxford University Press (2014)