Languages and Algorithms for Artificial Intelligence

Module 2

Matteo Donati

March 16, 2022

Contents

1 Introduction

One reason to study languages comes from the **linguistic relativity hypothesis**, the language shapes our view of the world, i.e. how we approach the world. When it comes to programming languages, Kenneth Iverson wrote a paper which is called “Notation as a tool of thought”, in which he argues that the programming language we use influences the way we solve problems.

1.1 Programming Paradigms

Programming languages are categorized based on their features (e.g. imperative, declarative, functional, object-oriented, logic-based, reactive, etc.). A programming paradigm specifies the following:

- **entities**, i.e. what one computes (e.g. function, expression);
- **execution model**, i.e. how one computes entities (e.g. message passing, substitution model, logical inference).

1.2 Artificial Intelligence Paradigms

In the “The master algorithm” book, the main tribes of artificial intelligence are:

- **connectionism**, neural networks, deep learning, gradient descent are examples of such concept;
- **bayesianism**, probabilistic models, probabilistic inference, Bayes theorem are examples of such concept;
- **symbolism**, logical theorems, symbolic manipulation, resolution are examples of such concept.

These paradigms explain the “what” and the “how” reasoning and learning happen. For example, in connectionism, the model is a neural network and such network works by using gradient descent, in bayesianism one has a probabilistic model and such a model is learned using the Bayes theorem, while in symbolism one has logical theories which are learned by applying symbolic manipulation (or resolution, etc.).

1.3 Duality Between Paradigms

There exist analogies between programs (programming) and models (artificial intelligence), and between computation (programming) and learning (artificial intelligence), so one can notice that these two types of paradigms are equivalent. For example, let $\mathcal{L}+$ be a programming language which is extended in order to be able to define an artificial intelligence model as well as a way to implement learning (e.g. the interpreter must be able to perform differentiation, bayesian inference, resolution). In particular, each artificial intelligence paradigm induces a new programming paradigm ($\mathcal{L}+$), i.e. the new programming language is rich enough to define both the “what” and the “how” of the considered artificial intelligence paradigm.

2 The Racket Language

The Racket language is a modern dialect of Lisp and a descendant of Scheme. It is designed as a platform for programming language design and implementation. In particular, in Racket everything is a symbolic expression (i.e. a list). At interpretation time, such lists are converted into trees. Moreover:

- `(define x 3)`, defines a variable `x` and assigns to it the value `3`. In particular, `x` is simply a placeholder for the given value, not a memory pointer. For example, one cannot run `(define x (+ x x))` (but one can run `(define y (+ x x))`);
- in general, if one wants to express $f(e_0, \ldots, e_N)$, where $e_0, \ldots, e_N$ are expressions, one can run `(f e0, ..., eN)`. For example, $-(x, y)$ is expressed as `(- x y)`. Moreover, the general `(f e0, ..., eN)` is a function call;
- if one considers `(define x 3)` and `(define y (+ x 3))`, at interpretation time `(+ x 3)` gets computed so to obtain `(define y 6)`. One will then reach a state in which $\{x \mapsto 3, y \mapsto 6\}$;
- anonymous functions (lambda functions) are defined as `(lambda (x0, ..., xN) e)` $(\lambda(\boldsymbol{x}).e)$ (this implements the concept $x \mapsto f(x)$, namely of $\lambda x.f$). Informally, an expression of the form `(lambda (x) e)` is a function with formal parameter `x` and body `e`. These functions come from the lambda calculus. For example, `(define cube (lambda (x) (* x x x)))` (or, equivalently, `(define (cube x) (* x x x))`);
- for example, the program which takes in input a function, f, and returns the composition $f \circ f$ is defined as follows:

  ```
  (define apply_twice (lambda (f) (lambda (x) (f (f x)))))
  ```

  or as:

  ```
  (define apply_twice f (lambda (x) (f (f x))))
  ```

  or as:

  ```
  (define (apply_twice f x (f (f x)))
  ```

- recursive functions, such as the following `power` function, are defined as follows:

  ```
  (define (power x y)
    (if (= y 0)
      1
      (* x (power x (- y 1)))))
  ```

  Moreover, the general form of the conditional operator is `(if cond e1 e2)`.

2.1 Syntax

Racket expressions are built upon a collection of atomic expressions (e.g. `lambda`, `if`, `3`, `define`) and a disjoint collection of symbols/names used as variables (e.g. `x`, `y`, `age`, `cube`). In particular:

- every atomic expression is an expression;
- every variable is an expression;
- if `e0, ..., eN` are expressions, then also `(e0, ..., eN)` is an expression;
- nothing else is an expression.

2.2 Semantics

In order to introduce the semantics of Racket, one needs to introduce the concept of value. In particular, a value is either an atomic expression, a variable, or a lambda expression. Therefore, values are expressions that cannot be evaluated any further. In order to evaluate an expression e one needs also to consider and environment Γ containing bindings of variables and functors (e.g. $\Gamma = \{x \mapsto 2, \dots\}$). Considering an expression of the form `(e0 e1 ... eN)` in a given environment Γ, if such evaluation succeeds, one obtains a value as a result, and the environment can be possibly modified. In particular, one starts evaluating `e0`, and if the process succeeds one obtains a symbol ϕ:

- in case ϕ is an atomic expression `a`, then one applies the predefined semantics `a` on `(a e1 ... eN)`;
- in case ϕ is a variable `x`, then one looks for the value associated to `x` in Γ, namely `v`. Lastly, one evaluates `(v e1 ... eN)`;
- in case ϕ is a lambda-abstraction `(lambda (x1 ... xN) e)`, then one evaluates operands `e1`, `...`, `eN`. If the evaluation succeeds, one obtains values `v1, ... vN` and such values are used to replace each variable `xI` so to evaluate the result thus obtained.

For example, given the following lambda function:

```
(define (double x)
  (+ x x))
```

and the query `(+ (double 6) (double 6))`, the evaluator recursively evaluates expressions by exploiting the knowledge stored in Γ:

- it creates an environment (Γ) in which `name` $\mapsto$ `expression`. The previously defined function is equivalent to `(define double (lambda (x) (+ x x)))`. This leads to have $\Gamma = \{$`double` $\mapsto$ `lambda (x) (+ x x)`$\}$;
- it reads the query expression, which is of the form `(e0, e1, ..., eN)`, and substitutes `e0` with `+`;

- it reads `(double 6)` and the definition of `double` from Γ, and finally computes `(lambda (x) (+ x x) 6)` substituting `x` with `6` in the body of the function, obtaining `(+ 6 6)`;
- it reads `(+ 6 6)` and computes the result `12`.
- it reads the second `(double 6)` and evaluates the expression as before;
- it lastly computes the final result `(+ 12 12)`, which results in `24`.

Moreover:

- there exist two types of substitutions. Considering the query expression `(lambda (x) (+ x x) (+ 3 2))`:
  - by using the call-by-name (also known as **memoization**) substitution type one obtaines `(+ (+ 3 2) (+ 3 2))`. This allows one to do not evalute something that may not ever be called, such as a function. It is also useful to deal with infinitary data structures, or similar structures;
  - by using the call-by-value substitution type one obtaines `(+ 5 5)`. This allows one to do evaluate `(+ 3 2)` only once.
- parenthesis also have semantic meaning. In particular, there exists a different between `e` and `(e)`. Indeed:

  $$\frac{\texttt{e0, e1, ..., eN} \in \mathit{Exp}}{\texttt{(e0, e1, ..., eN)} \in \mathit{Exp}}$$

  In this case, `e` is not evaluated, since it is considered to be a value, while `(e)` is considered to be a function call with zero parameters and it is, thus, evaluated. By adding parenthesis one modifies the semantics of the expression.

3 Probabilistic Programming

Probabilistic programming is an artificial intelligence paradigm which could be either randomized (randomized algorithm, generative models) or Bayesian (inference). The world of interest is a probabilistic one, in which one assigns probabilities to facts. Most of the times one is unable to describe such a world, so one can use a declarative program to let the machine decide how to model the world.

3.1 Randomized Probabilistic Programming

Generative models (artificial intelligence) can be related to probabilistic computation (computation). In particular, in order to modify the considered language, one can add primitives to the language to sample from probability distributions (i.e. to implement random computation) (`flip`, `normal`, `bernoulli`). For example:

- the function `(define test-flip (repeat flip 1000))` allows one to toss a coin for a thousand times. In particular, a function call to `test-flip` will produce a list of a thousand elements `(list ...)`, it will execute `flip` a thousand times, and finally it will store the results into the already created list (e.g. `(list #t #t #f ...)`). The returned list implements a sub-probability distribution (for same cases the program can diverge, $\sum_{\omega} P(\omega) \leq 1$);
- to visualize results (i.e. the output distribution) one can use `(hist test-flip)`;
- the functions `(define unfair-coin (bernoulli-dist 0.3))`, `(define my-gamma (gamma-dist 0.9 0.5))`, `(define my-gauss (normal-dist 0 1))` implement, respectively, a Bernoulli distribution, a gamma distribution, and a normal distribution. However, `bernoulli-dist`, etc. and `flip` are two different types of object (`flip` is a procedure, while the former objects are distribution objects). In order to execute such functions, one need to sample from such distributions. To do so, one can evaluate `(sample (bernoulli-dist 0.5))`, etc., using the primitive procedure `sample`;
- for continuous distribution, one can use, for example, `(density (repeat (lambda () (normal 0 1)) 600))`, which will produce a probability density function instead of a histogram.
- examples of probabilistic functions are the following:

  ```
  (define fair-coin (lambda () (if (flip 0.5) 'h 't)))
  (define (make-coin-wgt) (lambda () (if flip wgt) 'h 't))
  ```

- `map` is a higher-order function which takes in input a list and returns the list obtained by applying the argument function `f` to the elements of the input list. A simple implementation is the following:

  ```
  (define (my-map f xs)
    (if (null? xs) nil (cons (f (first xs)) map f (rest xs))))
  ```

  This allows one to implement parallel execution, given that the applications of `f` onto the elements of the input list do not threat each other.

- a simple probabilistic experiment is the following:

  ```
  (define my-coin (lambda () (flip 0.5)))

  (define sum (lambda (xs) (foldl + 0 xs)))

  (define (experiment n)
    (repeat (lambda () (sum (map (lambda (x) (if x 1 0)) (repeat my-coin 10)))) n))
  ```

  The semantics of the `foldl` procedure is `(foldl + 0 (x0 x1 ... xN))` is `+ x1 (+ x2 + (... + (xN)))`, where `0` is the output result whenever `xs` is empty. In this experiment one will toss a coin ten times and produce a list. Then by using the function `map` is applied, producing a new list. Finally, the `sum` function is called, which sums all the elements of the `map`'s output list. In theory, the output of such experiment should output `5` (50% of the tosses);

- divergence can happen. For example, the stochastic recursive function that randomly decide whether to stop or not is the following:

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

  (hist (repeat (lambda () (geometric 0.5)) 300))
  ```

  Namely, one can generate a distribution as a function that flips a coin, returning the number of times one get false before the first true. However, this program could not terminate. As a matter of fact, if a coin always output false this program would not terminate;

- the construct for building discrete distributions is the following:

  ```
  (define colors (discrete-dist ['blue 1/3] ['green 1/3] ['brown 1/3]))
  ```

  One can then define the following procedure:

  ```
  (define (eye-color person) (sample colors))
  ```

  If one now evaluates `(eye-color 'Alice)` multiple times, one will obtain different results. This is usually not a wanted result. To achieve persistency, one can do the following:

  ```
  (define eye-color-mem (mem (lambda (person) (sample colors))))
  ```

  i.e. one can exploit memoization (no evaluation if not needed) and store the first evaluation's output in order to output the same result when the same procedure is called again. Now, `(eye-color-mem 'Alice)` will always produce the same result.

3.2 Bayesian Probabilistic Programming

This kind of probabilistic programming allows one to do inference and to update the model based on new information. In particular:

- considering the following program:

  ```
  (define model1 (rejection-sampler
    (define A (bernoulli 0.5))
    (define B (bernoulli 0.5))
    (define C (bernoulli 0.5))
    (define D (+ A B C))
    D))
  ```

  and supposing that at some moment `A` becomes equal to 1, then this changes one's knowledge (this can be thought as some learning mechanism). This is reflected onto one's probabilistic model. In particular, the `observe/fail` function allows one to observe a boolean fact:

  ```
  (define model1 (rejection-sampler
    (define A (bernoulli 0.5))
    (define B (bernoulli 0.5))
    (define C (bernoulli 0.5))
    (define D (+ A B C))
    (observe/fail (= A 1))
    D))
  ```

  This is an example of **conditional inference**. Conditional inference is a type of inference that follows a specific probabilistic pattern based on the Bayes theorem. Reasoning and learning are examples of conditional inference;

- the Bayes theorem states that:

  $$P(H|E) = \frac{P(E|H)P(H)}{P(E)} \quad \Rightarrow \quad P(H|E) \propto P(E|H)P(H) \tag{1}$$

  where $P(H|E)$ is the posterior distribution (posterior knowledge), $P(E|H)$ is the likelihood distribution (observations), and $P(H)$ is the prior distribution (prior knowledge);

- for example, the following program first builds a generative model, than compute the posterior distribution (how likely it is to be weekend given that four busses has passed, $P(weekend = 1|nbusses = 4)$) by using the given knowledge (`observe-sample`):

  ```
  (define model3
    (rejection-sampler
      (define is-weekend (bernoulli (/ 2 7)))
      (define rate (if is-weekend 3 10))
  ```

```
    (observe-sample (poisson-dist rate) 4)
    is-weekend))
```

In particular, $P(weekend = 1) = \frac{2}{7}$, $P(nbusses = 4|weekend = 1) = \frac{1}{4!}3^4e^{-3} = 0.168$ (computed using the Poisson distribution). Lastly, by using the Bayes rule, and then by normalizing the result (this is achieved by finding a constant α which allows one to produce a correct probability distribution, namely to obtain $\sum_\omega P(\omega) = 1$), one can compute the posterior distribution;

- however, finding α is usually not feasible. The posterior distribution is, indeed, approximated, and not exactly computed, by using different methods. A particular method is **rejection** (`rejection-sampler`). The afore-mentioned program works as if the program definition was the following:

```
(define (model3-rejection)
  (define is-weekend (bernoulli (/ 2 7)))
  (define rate (if is-weekend 3 10))
  (define observed-value (sample (poisson-dist rate)))
  (if (= observed-value 4)
    is-weekend
    (model3-rejection)))
```

  In this case one is using stochastic recursion (recursive call to `model3-rejection`) and only uses randomized programming approach (not a Bayesian one), and can also not terminate (for example, when `observed-value N`, and `N` is not in the considered support);

- learning can also be viewd as a sort of conditional inference. In particular, learning is modelled by considering a first hyphotesis sampled from some prior knowledge, $H \sim Prior$, and by considering some observations one tries to build a replica of such observations to be compared with the true ones, and in doing so one changes the initial hyphotesis H;

- for example, considering the following program:

```
(define observed-data '(h h h h h))
(define make-coin (lambda (weight) (lambda () (if (flip weight) 'h t))))
```

  if one wants to learn the coin is fair (unfair), one could use the following Bayesian program:

```
(define (learning1 data)
  (mh-sampler)
    (define fair-coin? (flip 0.999))
    (define coin (make-coin (if fair-coin 0.5 0.95)))
    (observe/fail (equal? data (repeat coin (length data))))
    fair-coin?)
```

  by doing so, one is able to find if the given observations lead one to have a fair (unfair) coin. The higher is the number of `'h` (compared to the number of `'t`) in the observed data, the higher will be the probability of having an unfair coin.

- the **Markov chain Monte Carlo** is the main family technique used to simulate sampling from an unknown distribution (e.g. the posterior distribution). In particular, a **Markov chain** is a probabilistic and non-deterministic automata (i.e. a probabilistic transition system), where each of the edges of the considered automata is labelled with a probability. Moreover, in these automata, the transitions history does not matter, i.e. $P(x_t|x_{t-1}, \dots, x_N) = P(x_t|x_{t-1})$. More formally, Markov chains can be defined as:
  - a function, $c : \mathcal{X} \to D(\mathcal{X})$, which returns a probability distribution for each state $x \in \mathcal{X}$. In particular, $c(x_1)(x_2) = \epsilon \in [0, 1]$ is the probability of passing from x_1 to x_2. For example, if $c(x_1) = \{0.3 \cdot x_2, 0.7 \cdot x_3\}$, then $c(x_1)(x_2) = 0.3$. The application of c on every state of $\mathcal{X}$ will produce a **transition matrix**, P, containing the probability distribution of each state:
$$c\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}\right) = \underbrace{\begin{bmatrix} \epsilon_2 & \epsilon_3 & \dots \\ \epsilon_1 & \epsilon_3 & \dots \\ & \vdots & \end{bmatrix}}_{P} \tag{2}$$
    The transition matrix P completely deinfes the generic Markov chain;
  - a relation, $c : \mathcal{X} \times \mathcal{X} \to [0, 1]$, between states.

- given a transition matrix P, and a starting state $\bar{x}$, one can apply matrix multiplication, between P and $\bar{x}$, in order to take steps (i.e. to compute transitions) in the considered automata:
$$\bar{x} \cdot P \cdot P \cdot P \cdot \dots \tag{3}$$
  the resulting vector determines the final distribution and implements all the taken steps. After some iterations one can eventually reach a **steady state** (i.e. the fixed point, stationary distribution) φ, for which $\varphi \cdot P = \varphi$;

- given a target distribution φ, and a Markov chain P for which φ is a steady point, once φ has been reached, then one can use φ to produce samples. For example, once φ has been reached, one can query $\varphi(x_1), \varphi(x_2), \dots$;

- given a steady state φ of a general Markov chain P, the **detailed balanced condition** is a technique used to find such a stationary distribution. In particular, for φ to be a steady point of such chain, the following property must hold:
$$\forall x_1, x_2.(\varphi(x_1) \cdot P(x_1, x_2) = \varphi(x_2) \cdot P(x_2, x_1)) \tag{4}$$

- **Metropolis Hastings** (`mh-sampler`) is an instanciation of the Monte Carlo algorithm which allows one to simulate sampling from $\varphi = k \cdot f(x)$ (equivalent to the Bayes theorem), where $f(x)$ is the un-normalized version of the unknown distribution φ, and k is a normalization factor (f can be seen as a prior distribution and φ and as a posterior distribution). In particular, to simulate sampling from φ, one needs to build a Markov chain such that φ is a fixed point of P. To do so:

1. consider a known distribution from which it is easy to sample, $g(x_{t+1}|x_t) = \mathcal{N}(x_t, \sigma^2)$;
2. accept the next state, x_{t+1}, with a given probability $A(x_t, x_{t+1})$, namely $g(x_2|x_1) \cdot A(x_1, x_2)$. The acceptance rate A is computed as follows:
   (a) being $\forall x_1, x_2.(k \cdot f(x_1) \cdot P(x_1, x_2) = k \cdot f(x_2) \cdot P(x_2, x_1))$, and considering $P(x_1, x_2) = g(x_2|x_1) \cdot A(x_1, x_2)$ and $P(x_2, x_1) = g(x_1|x_2) \cdot A(x_1, x_2)$, one obtains:

   $$\forall x_1, x_2.(k \cdot f(x_1) \cdot g(x_2|x_1) \cdot A(x_1, x_2) = k \cdot f(x_2) \cdot g(x_1|x_2) \cdot A(x_1, x_2)) \tag{5}$$

   (b) by re-arranging terms:

   $$\frac{A(x_1, x_2)}{A(x_2, x_1)} = \frac{k \cdot f(x_2)}{k \cdot f(x_1)} \frac{g(x_1|x_2)}{g(x_2|x_1)} \tag{6}$$

   and thus:

   $$\frac{A(x_1, x_2)}{A(x_2, x_1)} = \underbrace{\frac{f(x_2)}{f(x_1)}}_{R_f} \underbrace{\frac{g(x_1|x_2)}{g(x_2|x_1)}}_{R_g} \tag{7}$$

   (c) the acceptance rate A is defined as follows:

   $$\begin{cases} A(x_1, x_2) = R_f \cdot R_g \text{ and } A(x_2, x_1) = 1 & \text{if } R_f \cdot R_g < 1 \\ A(x_1, x_2) = 1 \text{ and } A(x_2, x_1) = \frac{1}{R_f \cdot R_g} & \text{if } R_f \cdot R_g \geq 1 \end{cases} \tag{8}$$

   Therefore $A(x_1, x_2) = \min(1, R_f \cdot R_g)$. This allows one to have, by definition of detailed balanced condition, φ as steady state of P.

Lastly, in order to build the Markov chain one can proceed as follows:

1. start from an initial guess, x_0, and transition until x_t;
2. guess a candidate successor y using g, namely $g(y|x_t)$;
3. compute the acceptance rate $A(x_t, y) = \min(1, R_f \cdot R_g)$;
4. choose a randomly a threshold μ;
5. if $\mu \leq A(x_t, y)$ then $x_{t+1} = y$, otherwise $x_{t+1} = x_t$;
6. $t = t + 1$.

4 Interpretation of a Program

The **operational semantics** of a program is a description of how a program executes ($P \rightarrow Q$) on a general machine (abstract type of semantics). In particular, the considered language upon which the semantics is applied is defined as follows:

$$\begin{array}{rcl} e & ::= & c \\ & | & x \\ & | & (\lambda(x_1, \ldots, x_n)e) \\ & | & (e_1, \ldots, e_n) \\ & | & (if\ e_0\ e_1\ e_2) \end{array}$$

where $c ::= true|false|\underline{n}$ (where $\underline{n}$ represents a general positive integer number). In particular, the operational semantics is defined as a binary relation between expressions, $e \rightarrow e^*$. To define this relation, the concept of **value** and **redex** should be defined:

- a **value**, v, is an expression which cannot be simplified any further, namely:

$$\begin{array}{rcl} v & ::= & c \\ & | & x \\ & | & (\lambda(x_1, \ldots, x_n)e) \end{array}$$

- a **redex**, r, is an expression which needs to be simplified, namely:

$$\begin{array}{rcl} r & ::= & (v_0, v_1, \ldots, v_n) \\ & | & if\ v\ e_0\ e_1 \end{array}$$

Moreover, an **evaluation context**, E, is defined as:

$$\begin{array}{rcl} E & ::= & [-] \\ & | & (v_0, \ldots, v_{k-1}\ E\ e_{k+1}, \ldots, e_n) \\ & | & if\ E\ e_0\ e_1 \end{array}$$

For any $e \in closed$, where $closed$ is the set of variables under λ scope, then either e is a value or $e = E[r]$. At this point, the operational semantics of such language can be defined by a general inference rule:

$$\frac{premise \quad premise}{conclusion} \tag{9}$$

in particular:

- the relation $e \rightarrow e^*$ is defined by the following inference rule:

$$\frac{r \mapsto e}{E[r] \rightarrow E[e]} \tag{10}$$

Practical implementations are:

– considering $if\ v\ e_0\ e_1$:

$$\frac{}{(if\ true\ e_0\ e_1) \mapsto e_0} \qquad \frac{v \neq true}{(if\ v\ e_0\ e_1) \mapsto e_1}$$

– considering $(v_0, v_1, \ldots, v_n)$:

$$\frac{}{((\lambda(x_1, \ldots, x_n)e)v_1, \ldots, v_n) \mapsto e[x_1 = v_1, \ldots x_n = v_n]} \qquad \frac{[\![c]\!](v_1, \ldots, v_n) = v}{(c, v_1, \ldots, v_n) \mapsto v}$$

For all the other cases an error is produced.

- the $Eval : Exp \to (Val \text{ or } Error)$ operator is the operator which allows one to compute $e \to^* v$.

In order to introduce probabilistic computation, the considered language is extended as follows:

$$\begin{array}{rcl} e & ::= & c \\ & | & x \\ & | & (\lambda(x_1, \ldots, x_n)e) \\ & | & (e_1, \ldots, e_n) \\ & | & (if\ e_0\ e_1\ e_2) \\ & | & sample(e_0, \ldots, e_n) \end{array}$$

and

$$\begin{array}{rcl} v & ::= & c\ (c \text{ can also be } bernoulli\text{, etc.}) \\ & | & x \\ & | & (\lambda(x_1, \ldots, x_n)e) \\ & | & (\mu, v_i, \ldots, v_n)\ (\mu \text{ is a known distribution}) \end{array}$$

The engine of randomness is given by the redex $r :- (sample(\mu v))$. The operational semantics is modified as follows:

$$\frac{[\![c]\!](v_1, \ldots, v_n) = v \quad c \text{ not } \sim \text{ some distribution}}{(c, v_1, \ldots, v_n) \overset{1}{\mapsto} v}$$

$$\frac{[\![n]\!](v_1, \ldots, v_n)(c) = P}{(sample(\mu, v_1, v_n)) \overset{P}{\mapsto} c}$$

where P denotes the probability of obtaining c by sampling from some distribution.