KU LEUVEN

Implementing Cognitive Models in ProbLog

Koen Boeckx

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Thesis supervisors:
Prof. dr. Luc De Raedt
Dr. Angelika Kimmig

Assessors:
Dr. Jonas Vlasselaer
Dr. ir. Tinne De Laet

Academic year 2016 – 2017

Preface

Writing a thesis is always very intensive, especially for a part-time student. But nevertheless, it was a very rewarding experience. I would like to thank my two co-promotors, Prof. Dr. Luc De Raedt and Dr. Angelika Kimmig. Special thanks go out to Angelika for all her help, her availability and the many useful tips she gave me.

Furthermore, I would also like to thank my family for their patience and support.

Koen Boeckx

Contents

Abstract

Probabilistic programming languages have become a very powerful and popular tool for statistical modeling and inference. One of these languages, Church, has been used to model a number of cognitive models in the online book *Probabilistic Models of Cognition* by Noah Goodman and Joshua Tenenbaum. Another probabilistic programming language, ProbLog, is based on Prolog and is being developed at the department of Computer Science at the Katholieke Universiteit Leuven. The goal of this master thesis is to take the cognitive models as they are programmed in Church, and port them to ProbLog. It is therefore necessary to establish a set of general rules applicable for the transition from Church to ProbLog. Some of these models however, are based on hierarchical structures that pass continuous probability distributions from one layer to the next. Since ProbLog cannot handle continuous distributions, Distributional Clauses, another probabilistic logic programming language of the Prolog family was used to model these programs. Finally, the performance of these probabilistic languages was compared. Both speed of execution and precision were taken into account.

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this thesis is to port the different programs from the online book *Probabilistic Models of Cognition* [1] by Noah Goodman and Josh Tenenbaum, to ProbLog, a probabilistic programming language developped at the KU Leuven. In order to do this, several design patterns were identified and translated in a suitable ProbLog expression.

The goal of this thesis is NOT to interpret the results obtained from the programs, since that's already (and very thoroughly) done by the authors of [1]. However, when an obtained result is interesting, or when the result differs from the one obtained in [1], it will be looked into.

1.1 The Probabilistic Approach to Cognitive Science

Cognitive Science is the interdisciplinary, scientific study of the human mind and its processes. It sudies how knowledge is acquired, and how it is understood, through thought, experience and the senses [2]. Almost every second of every day, our mind is busy making intelligent inferences for both reasoning and learning. One subfield in cognitive science is the *computational theory of mind*, which tries to understand the mind by putting forward that the mind is like a computer. In this analogy, mental representations are computer programs, and thinking itself is like running a computer program [1].

According to the generative approach, mental representations are like models of the world: they capture general descriptions of how the world works, and running them leads to predictions of observations, which can later be used to make inferences.

Inference in its largest sense is then to reason about the assumptions in the mental model, based on what our senses observe. In this way, we can adjust our prior beliefs.

For these models to work, they have to be probabilistic: just like the world is uncertain (be it inherently, or because of a lack of knowledge about it), so must be our mental models. Probability theory provides us with a system for reasoning under uncertainty.

From this point of view, a probabilistic programming language is a formal method to put this theory to use. The universality of the language should enable us to express any computable process. An inference algorithm allows us then to reason about the model, based on observations. Since it is a probabilistic program, we are not concerned with absolute truths, but rather with probability distributions on the parameters of the model.

In [1], the authors made abundantly clear that probabilistic programming can be a very efficient tool to model cognitive processes and to validate and explain real-life psychological experiments. That, however, is not the goal of this thesis. The focus here is on translating probabilistic models from Church to ProbLog. That being said, the cognitive aspect of all these models remains very important, and wherever necessary the needed context is provided to be able to interpret the models.

1.2 Probabilistic Programming Languages

Probabilistic programming is the next step in probabilistic modeling, extending and improving on the Probabilistic Graphical Models (PGM) framework that originated in the late 1980s [3]. The goal of a probabilistic programming language is to unify general purpose programming with probabilistic modeling [4], and methods for inference on these models. The programmer specifies an entire probabilistic model by writing code that is capable of generating a sample from the joint probability distribution (the generative model), and efficient inference algorithms then allow to draw conclusions based on the observation of some variables.

The general setup of a probabilistic program follows three steps:

1. Write the probabilistic programming, that puts into code the (causal) model of the world, linking together the probabilistic variables through more or less complex interactions.

2. Describe which are the variables that are observed, and what is their observed value. This is called the *evidence*.

3. Define which variables we are interested in; thus, what is the result of the inference process? These are the *queried variables*.

A popular probabilistic programming language (PPL) is *Church*, named after computing pioneer Alonzo Church and based on the *Scheme* programming language. Developed by Josh Tenenbaum and Noah Goodman, it is the language in which the original version of [1] was written, although it now has been ported to WEBPPL, a PPL based on javascript. Nevertheless, this work will be based on the Church version of [1]. Since Church is based on Scheme, it too is a functional programming language. It is also sampling-based, a major difference with ProbLog.

ProbLog, another PPL, was developed at the Katholieke Universiteit Leuven, and is currently in its second version. It is based on the logical programming language *Prolog*, and adheres to the basic rules of logic programming, although it has some specificities of its own. ProbLog uses another inference engine, based on *weighted* *model counting*, which, in contrast with the sampling approach of Church, gives exact results. On the other hand, ProbLog cannot handle continuous distributions.

Chapter 2 will provide a more detailed study of both Church and ProbLog, as well as of Distributional Clauses, an approach to probabilistic programming based on Prolog, but with sampling based inference.

1.3 Thesis Overview

Chapter 2 gives an overview of the syntax, semantics and inference mechanism of Church, ProbLog and Distributional Clauses. Following that, chapter 3 offers a detailed study on a concrete example of how a Church program can be ported to ProbLog. Once that is done, chapter 4 walks through the early chapters of *Probabilistic Modeling of Cognition* [1]. These chapters are not directly related to aspects of cognitive science, but explain the basic functionality and possibilities of Church. Transforming these programs into ProbLog serves as a proof-of-concept of the feasibility of the goal of this thesis, and as a prelude for the programs of chapter 5. This chapter is more oriented to aspects of cognitive science, and requires more effort to both understand and translate the different programs. Sometimes, the use of a continuous distribution can not be averted, and *Distributional Clauses* instead of ProbLog will be used. Chapter 6 studies the differences in performance between Church, ProbLog and Distributional Clauses. It focuses on two aspects: precision and speed. Finally, chapter 7 offers a general conclusion and ideas for further work.

Chapter 2

Probabilistic Programming Languages

This chapter will briefly introduce the three probabilistic programming languages used in this thesis: Church [10], ProbLog [6] and Distributional Clauses [17].

2.1 Church

Church, named after computer pioneer Alonzo Church, is based on the functional language Scheme, a dialect of Lisp. It is a dynamically typed language in which procedures are first-class and expressions are values.

This section gives a brief overview of the workings of Church. It is based on [10].

Expressions in Church describe generative processes: their meaning is specified through a primitive procedure `eval`, which samples from the process. Another primitive procedure, `query`, generalizes `eval` to sample conditionally.

2.1.1 Syntax

As in Scheme, Church expressions can take one of a few forms:

```
 expression ::= c | x | (e1 e2 ...) | (lambda (x ...) e) |
(if e1 e2 e3) | (define x e) | (quote e)
```

where `x` is a variable, `e` an expression and `c` a constant.

Constants include primitive data types (nil, Boolean, char, ...) and standard functions to build data structures (`pair`, `first`, ...)

Procedures come in two types:

1. *Ordinary procedures* are triples `(body, args, env)` where a Church expression in the body is applied to the arguments `args`, taking into account the values of other variables in the environment.

2. *Elementary random procedures* are elementary procedures that also have a *distribution function.* This is a function that reports the the probability P(value

| env, args) of a return value from evaluating the body, given the environment and the values of the parameters.

Several elementary random functions are provided, like `flip`, which flips a fair coin (or a weighted coin, when called with a weight argument). More complex random functions are formed by combining elementary random functions.

The basic way to define a function is the `lambda` primitive. The following expression assigns, through the `define` keyword, a function to the variable `func`, which has one argument `x`, and returns the square of this argument:

```
(define func (lambda (x) (* x x)))
```

Since this is such a common programming construct, Church (and Scheme, for that matter) provide some *syntactic sugar* to allow easier programming. The following construct is identical to the previous one:

```
(define (func x) (* x x))
```

2.1.2 Semantics

An *expression* defines a generative process via recursive application of the `eval` primitive. It takes an expression and an environment, and returns a value. This is very similar to the workings of scheme, except that the use of random procedures introduces a set of random choices in the program. The probability of a finite evaluation history is the product of the probabilities for each elementary random evaluation procedure in this history.

2.1.3 Inference

`eval` only allows the modeling of generative processes. For useful probabilistic inference, however, we also need a way to sample from a distribution conditioned on observations. For this purpose, we use the `query` primitive. A query in Church normally has the following form:

```
1  (query
2     (define ...)
3     (define ...)
4     ...
5     (list return-value-1 ... return-value-n)
6
7     (condition (equal? observation1 value1))
8     (condition (equal? observation2 value2))
9     ...
10 )
```

where the first `define` statements define the generative model - including the random primitives. Line 5 collects all the (random) variables we want to observe. If there is more than one, it is common practice to return them in `list`-form. Finally, the lines from line 7 onwards condition the hitherto generative probability distribution, by requiring that observations (variables inside the generative process) are equal to certain values.

Church provides a number of concrete implementations of the `query` primitive:

- `enumerate-query`, which enumerates all possible worlds and their (conditional) probability
- `rejection-query`, which is probably conceptually the simplest query method, but which can be very inefficient - this type of query is treated in more detail in section 4.2.
- `mh-query`, which is based on Metropolis-Hastings sampling, a particular form of the Markov-Chain Monte-Carlo (MCMC) methods. This type of query requires two additional parameters: the number of samples wanted, and the *burn-in period*, the number of samples to discard before the method samples from a distribution that is close the the real one. Since this method is the primary sampling method of Church, it is explained in the next section.

2.1.4 Metropolis-Hasting Sampling

The Metropolis-Hasting (MH) sampling method [25] is a member of the Markov Chain Monte-Carlo (MCMC) family. Just as importance sampling (see section 2.3.2), this method uses a proposal distribution to sample from. The goal is than that the distribution of the samples drawn from this proposal distribution resembles the true distribution. Sampling from the proposal distribution should be easy.

In MH, the proposal distribution $Q(x|x^{(t)})$ depends on the current state $x^{(t)}$. The the procedure to generate a sample is as follows:

- A tentative new state x' is sampled from $Q(x|x^{(t)})$
- For this tentative state x', the acceptance ratio $\alpha = f(x')/f(x^{(t)})$ is calculated. The function $f(x)$ is a function proportional to the real distribution $p(x)$.
- If $\alpha > 1$, the new state is accepted, and the algorithm continues with $x' = x^{(t+1)}$ as the new current state.
- If $\alpha < 1$, the new state is accepted **with probability** α. If it is rejected, the algorithm remains in state $x^{(t)}$.

How is all this implemented in Church? This is explained in chapter 7 of [1]. The states x are all the possible executions of the code inside a query. The function $f(x)$ is than the product of all the probabilities of the different probabilistic choices encoutered during execution. To sample from the proposal distribution, a random choice during executed is changed and in doing so, a new state x' is created. This new state is than accepted or rejected as described in the procedure above.

Metropolis-Hastings starts with a random initial state x^0. Typically, the first samples generated are not very good; the chain of samples must evolve to represent the real distribution sufficiently well. therefore, the first samples are rejected. This is the *burn-in period*, and is a parameter of the `mh-query` function.

2.1.5 Stochastic Memoization

Memoization is a technique for efficient execution of a program that makes many identical calls to a function. When the function is evaluated for the first time with given arguments, the return value is stored. The next time a call to the relevant function with the same arguments is encountered, the function is not evaluated, but the stored value is simply retrieved from memory. Since memory retrieval is done in constant time, this method leads to a significant speed-up when many identical function calls are made.

While ordinary memoization doesn't alter the behavior of the program, stochastic memoization can have a big impact. Random functions return samples from a probability distribution, and these return values are thus different for every call, even if the parameters are the same. However, stochastic memoization allows the programmer to store the return value of a random function. Every time this function is called from that point on, the same value is returned. Church does this by providing the higher-order function `mem`, which takes a procedure as input and returns another function. As an example:

```
(define same (mem (lambda (arg) (flip))))
```

A call like `(same arg1)` will always return the same value, namely `'t` (true) or `'f` (false), depending on the result of the first evaluation. The argument in this case has no other purpose than to reference the stored value.

While stochastic memoization has to be handled explicitly in Church, the opposite is true in ProbLog: memoization is implicit and special precautions have to be taken to avoid it.

2.2 ProbLog

ProbLog is an extension of the Prolog programming language. It is part of a whole set of *probabilistic logic programming Languages*, like PRISM[11], ICL[12] and LPAD[13]. The concepts behind logic programming, as in Prolog, are explained in [9]. Appendix A gives an overview of these concepts. This section will focus on the concepts specific to ProbLog2, and is based on [6].

2.2.1 Syntax

ProbLog programs have two parts: a set of ground probabilistic facts, and a set of rules and (non-probabilistic) facts. A probabilistic fact is written as `p::f`, which represent a ground fact `f` that is true with probability `p`. A syntactic sugar construct allows for the specification of an entire set of probabilistic facts with a single statement. A statement of the form `p::f(X1,X2,...,Xn) :- body`, where `body` is a conjunction of calls to non-probabilistic facts, is called an *intensional probabilistic fact*. The clauses in the body define the domain for the variables X_i. When performing inference, these probabilistic facts should be replaced by their corresponding set of ground probabilistic facts.

Consider program 2.1 that models the Alarm Bayesian Network:

```
1 person(mary). person(john).
2 0.1::burglary.
3 0.2::earthquake.
4 0.7::hears_alarm(X) :- person(X).
5 alarm :- burglary.
6 alarm :- earthquake.
7 calls(X) :- alarm, hears_alarm(X).
```

Program 2.1: Alarm Bayesian Network

Line 1 are two normal Prolog facts: they state that both mary and john are persons. Lines 2 and 3 are probabilistic facts: line 2 says that there is a 10% chance of a burglary, while line 3 says there is a 20% chance of an earthquake. Line 4 is an intensional probabilistic fact: it says that if X is a person, there is a 70% chance that that person will hear the alarm. There are two ground probabilistic facts corresponding with this statement, one for each person:

```
0.7::hears_alarm(mary).
0.7::hears_alarm(john).
```

Lines 5 to 7 form a set of normal Prolog rules, describing the interactions between the different actors.

ProbLog provides an additional construct for ease of modeling, the *annotated disjunction* [5]. This construct has the form:

```
p1::h1; p2::h2; ...; pn::hn :- body.
```

with $\sum_{i=1}^{n} p_i \leq 1$, meaning that if `body` is true, one of the `hi` will be true with corresponding probability `pi`. If the probabilities don't sum to one, it might be that none of the `hi` is true.

When querying in ProbLog, we use a similar template as in Church, as explained in section 2.1.3. The `query` keyword, however, has another role: while in Church, it is a function that returns the sample(s) that correspond(s) with the conditions, in ProbLog it is a clause that determines which probabilistic fact should be queried. Since ProbLog performs exact inference, it returns the probability distribution over the entire domain of each atom. The generic structure of a ProbLog query is given below, where the `evidence` term plays a similar role as the `condition` keyword in Church:

```
generative_model :- ...
query(what_we_want_to_know). % query variable(s)
evidence(what_we_know).
```

2.2.2 Semantics

Each probabilistic fact `p::f` gives rise to two possible worlds: a world in which `f` is true with probability `p`, and another world where `f` is false, with probability `1-p`. This called an *atomic choice*. An atomic choice can be made for each ground probabilistic fact in the program. This is called a *total choice*. Hence, if there are n ground probabilistic facts in the program, there are 2^n total choices. The probability of a total choice is the product of the probabilities of all the atomic choices that were made. A ProbLog program defines thus a probability distribution over all total choices.

In the program 2.1, there are 4 ground probabilistic facts:

```
burglary, earthquake, hears_alarm(mary), hears_alarm(john)
```

and thus there are $2^4 = 16$ possible worlds, each with its own probability.

When we extend the total choice C with all non-probabilistic facts and rules R of the logic program, we obtain the *well-founded model*. This is denoted as $\mathrm{WFM}(C \cup R)$, and a given world ω is a model of the program if there exists a choice C such that $\mathrm{WFM}(C \cup R) = \omega$. The probability of a world that is a model of the program is the probability of the corresponding total choice; if there is no such total choice, the probability of the world is zero. This defines the distribution over possible worlds implied by the program.

2.2.3 Inference

The two most common inference tasks in probabilistic logical programming are:

1. Computing the marginal probability of a set of random variables given some observations or evidence (MARG task)
2. Finding the most likely joint state of the random variables given the evidence (Most Probable Explanation, or MPE)

In order to formally define these tasks, we use the definitions from [6]:

At is the *Herbrand base* of the probabilistic logic program. This is the set of all ground atoms in a given ProbLog program. A given set $\mathbf{E} \subset \mathbf{At}$ atoms is observed; the vector $\mathbf{e}$ contains the observed values. This is the *evidence* and we write $\mathbf{E} = \mathbf{e}$.

- in the **MARG** task, we are given a set $\mathbf{Q} \subset \mathbf{At}$ of atoms in which we are interested. These are the *query* atoms. For each of these atoms $\mathrm{q} \in \mathbf{Q}$, the goal is to compute the marginal probability distribution given the evidence: $\mathrm{P}(\mathrm{q}|\ \mathbf{E} = \mathbf{e})$.

- The **EVID** task is to compute the marginal probability of the evidence: P(**E** = **e**). Computing this marginal probability will be needed for the MARG task.
- The **MPE** is to find the most likely joint state of all non-evidence atoms, given the evidence: argmax_u P($\mathbf{U} = \mathbf{u} \mid \mathbf{E} = \mathbf{e}$), where **U** are the atoms not part of the evidence: $\mathbf{U} = \mathbf{At} \setminus \mathbf{E}$.

In this thesis, only the MARG task (and by extension the EVID task) will be performed. Hence, I'll focus on inference for these two tasks, and not for MPE.

The approach ProbLog takes to solving these inference problems consists of two steps:

1. Convert the program to a *weighted boolean formula*, and
2. Perform inference on the resulting weighted formula

Conversion to a Weighted Boolean Formula

The goal of this step is to transform the ProbLog program, together with the evidence $\mathbf{E} = \mathbf{e}$ and the set of query atoms **Q**, into a boolean formula φ that contains all the necessary information for inference.

The outline of this procedure is as follows:

1. Ground the ProbLog program L into a new program L_g, taking evidence and query atoms into account. We only need the part relevant to the query atoms.
2. Convert L_g into an equivalent Boolean formula φ_r. This formula is in *Conjunctive normal form* form.
3. Integrate the evidence $E = e$ into φ_r. This yields a new boolean formula φ.
4. Assign a weight to each probabilistic literal in φ. For each probabilistic fact `p::f`, set the weight of f to p and of $1 - p$ for $\neg f$.

How these steps are performed is explained in [6]. It can be shown that the resulting weighted boolean formula contains all the necessary information to perform inference.

Inference on the Weighted Formula

An advantage of transforming the logic program into a weighted boolean formula is that inference can be performed with existing, state-of-the art algorithms.

For the EVID task, the inference is done with *weighted model counting* (WMC). A *model* of a propositional formula is a truth-value assignment to all variables such that the formula is true. The weighted model count is the sum of all the weights of these models:

$$P(E = e) = \sum_{\omega \in MOD_{E=e}} P_L(\omega) = \sum_{\omega \in SAT_\varphi} w(\omega) \tag{2.1}$$

where $MOD_{E=e}$ the collection is of all models consistent with $E = e$, and SAT_φ all the models that satisfy the formula φ. As stated previously, these two sets are equivalent for inference.

In practice, this is done by converting the boolean formula into a arithmetic circuit, and then evaluating this circuit. Details can be found in [6].

The MARG tasks asks us to find

$$P(Q|E = e) = \frac{P(Q, E = e)}{P(E = e)} \tag{2.2}$$

The denominator is the result of the EVID task. The numerator can easily be found by exploiting the arithmetic circuit for each query variable Q, as encountered in the EVID task.

2.3 Distributional Clauses

2.3.1 Semantics and Syntax

Distributional Clauses (DC)[17] is, just as ProbLog, an extension of Prolog to allow for modeling of and inference with probabilistic programs. However, just as Church, inference in DC is done by generating samples from the posterior distribution.

To this end, the concept of a distributional clause is introduced. It is a clause of the form

```
h ~ D := b1, ..., bn.
```

where the different `bi` form the body of the clause, `~` is a predefined binary predicate in infix notation, and `D` is a probability distribution. For every substitution θ that grounds clause, $h\theta$ is a random variable distributed according to $D\theta$. Sampling is done with the `~=` predicate:

```
h ~ = H.
```

instantiates `H` to the value of the distributional clause `h`.

2.3.2 Inference

Just as MCMC methods, importance sampling [19] is a way to draw samples from a (possibly very complex) probability distribution. These samples can then be used to estimate certain properties of the distribution in question, like the mean. A posterior distribution often concentrates a significant portion of its probability mass in a limited region of its domain. To achieve accurate sampling from the true distribution p, importance sampling uses a proposal distribution q, from which samples can relatively easily be drawn.

Assume we want to find the the mean of a function f with respect to the distribution p:

$$\mu = E(f(X)) = \int_D f(x)p(x)dx \tag{2.3}$$

where D is the domain of distribution p. When sampling from p is hard, we draw samples from q and apply the following formula:

$$\mu = E(f(X)) = \int_D f(x)p(x)dx = \int_D \frac{f(x)p(x)}{q(x)}q(x)dx = E_q\left[\frac{f(X)p(X)}{q(X)}\right] \tag{2.4}$$

E_q is the expectation with respect to distribution q. introducing an adjustment factor $p(x)/q(x)$ thus compensates for sampling from the *importance distribution* $q(x)$ instead of the target distribution $p(x)$. This factor is also called the *likelihood ratio*. The choice of $q(x)$ is important: first of all, $q(x)$ should only be zero when also $p(x)$ is zero; otherwise the likelihood factor goes to infinity. Even when $q(x)$ is very small but not quite zero, difficulties can arise. Secondly, when $q(x) > 0$ where $p(x) = 0$, the efficiency of the sampling algorithm goes down, since we will draw samples for whom the likelihood ratio is zero, and therefore they won't count in the calculation of $E_p(f(X))$.

The result we obtain with this method is the *importance sampling estimate* of $\mu = E_p(f(X))$:

$$\hat{\mu_q} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)p(x_i)}{q(x_i)} \tag{2.5}$$

where x_i are samples drawn from $q(x)$.

In Distributional Clauses, this method is implemented with the `EvalSampleQuery` algorithm, that returns as a sample a partial world $x_q^{P(i)}$ of the probabilistic logic program, together with a weight $w_q^{(i)}$. The partial world $x_q^{P(i)}$ is sampled by combining likelihood weighting with an adapted version of the SLD resolution mechanism for logic programs. More details about this method can be found in [16].

This algorithm has a number of practical instantiations:

- `generate_backward(Query, L)`, which generates one sample as requested by the `Query`
- `query(PosEvidence, NegEvidence, Query, N, P)`, which returns the probability that `Query` is true, given the evidence. This can be both positive or negative. `N` is the number of (attempted) samples to evaluate `P`.
- `eval_query_distribution(X, PosEvidence, NegEvidence, Query ~= X, N, LP, _, _)` which returns the probability distribution all the values that the `Query` generates during the different sampling trials.

2.4 Comparison

	Church	**ProbLog**	**DC**
Based on	Scheme	Prolog	Prolog
Distributions	Hybrid	Discrete only	Hybrid
Inference	Metropolis-Hastings Sampling	Weighted Model Counting	Importance Sampling with Likelihood Weighting
Precision	Approximate	Exact	Approximate

Table 2.1: Comparing Church, ProbLog and DC

Chapter 3

From Church to ProbLog

.

Before beginning the work of interpreting and translating the different cognitive models from *Probabilistic Models of Cognition*, this chapter will explore how a Church program can be transformed into a ProbLog program. This will be done with a program of chapter 4 of [1], that studies a children's game named *Blickets and Blocking*. This example is relatively easy though not trivial.

Figure 3.1 indicates the different steps taken for translating the original program on the left to the problog program on the right.

1. `samples` is a collection of samples generated by the `mh-query` function. Apart from the number of samples and the burn-in period (see 2.1.4), this function takes as arguments:

   - An undefined number of `define` statements, who form the model. Most of the attention during translation will go to these statements
   - A statement about the query variables (line 13)
   - A list of condition (or observation) statements (line 15)

   It is crucial to identify these parts of the program from the start, since they will form the skeleton of the ProbLog program.

2. `(blicket 'A)` is the function of which the result will be returned to `samples`. In this case, it is a true-or-false statement whether or not block `A` is blicket (more on this later). This is translated in the problog `query` clause (line 20).

3. Line 15 of the Church program, `(machine (list 'A 'B))`, is shorthand for the complete expression `(condition (equal? (machine (list 'A 'B)) true))`. The `condition` expression identifies the evidence or observations in Church. The ProbLog analogue is the `evidence` clause. This is directly reflected in line 21 of the ProbLog program.

4. We now come to the model. Line 4 of the Church program defines the `blicket` function, which takes one argument, `block`, and returns the result of

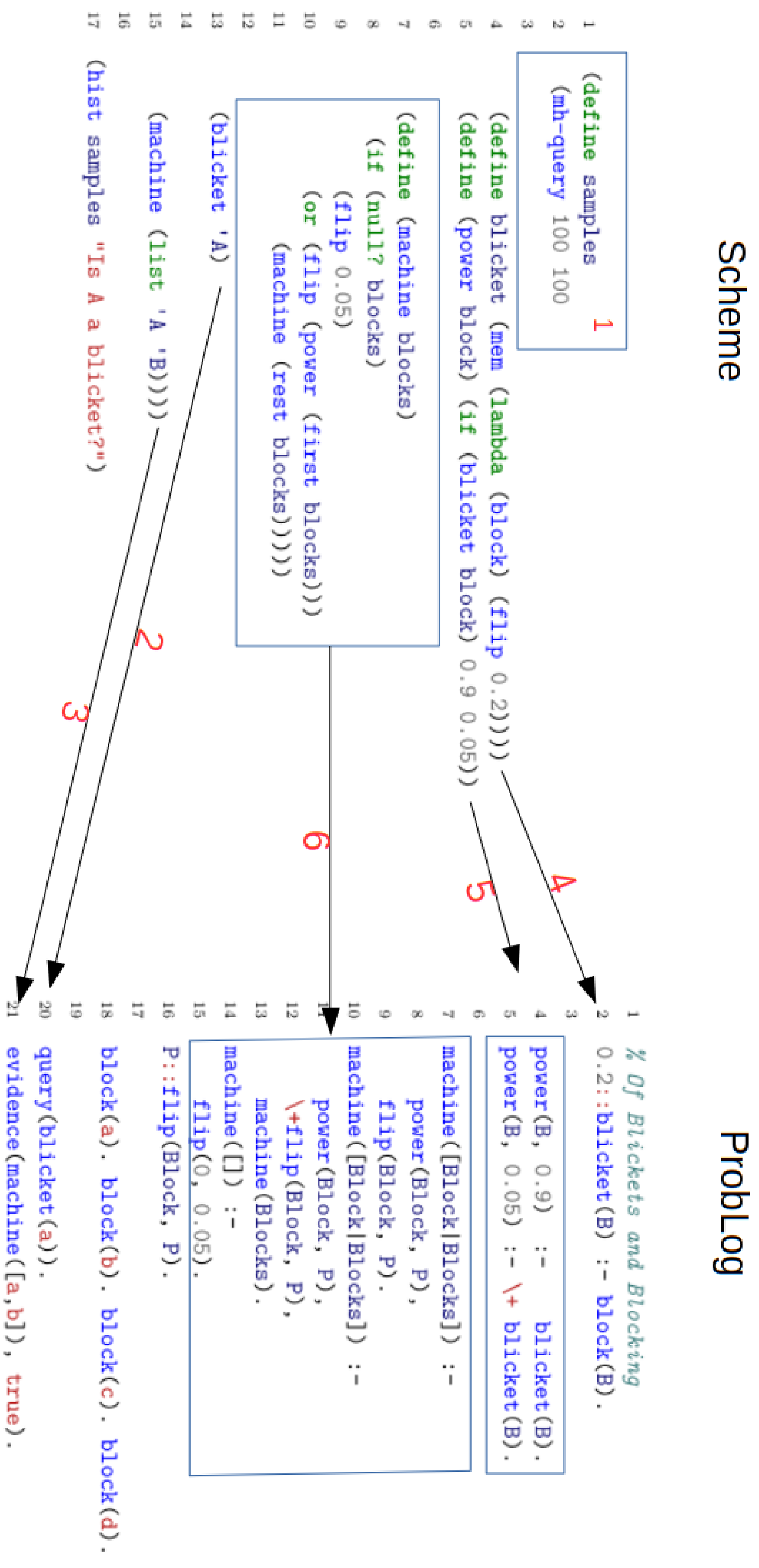
`(flip 0.2)`. Any function that returns true or false, is modeled in ProbLog as a probabilistic fact: `0.2::blicket(Block)`. In this case however, a body term has been added to the fact (making it effectively a rule), saying that the `Block` variable must be a block. Particularly noteworthy is the use of the `mem` function in Scheme, which *memoizes* the `blicket` function. This means that when this function is called for the first time with a certain argument, this value is stored and re-used the next this function is called with the same argument. In ProbLog however, memoization is implicit, and sometimes (see step 6) measures must be taken to circumvent it.

5. Line 5 of the Church program contains no stochastic primitive, and can thus be modeled as an ordinary Prolog clause. `power` is a function that takes an argument (`block`), and returns a value. This is typically modeled with a predicate with two arguments: `power(?Block, ?Value)` This function also contains a conditional statement: if `blicket(block)` is true, the function will return 0.9, else it returns 0.05. This is typically modeled by two clauses about the same term: one time with a body where the condition is true, and one where it is not. Thus, line 5 of the Church program is modeled by two rules in ProbLog: lines 4 and 5.

6. Lines 7 to 11 define a function `machine` that takes as argument a list `blocks`, and returns true or false. Because of this latter fact, this function can be modeled as a term with one argument: `machine(+Blocks)`, which will be true of false. In Church this function is implemented as a recursive function following the typical pattern of recursion on a list: the base case is the empty list (`(null? blocks)`), the first element of the list is treated (`(flip (power (first blocks)))`), and if this is negative, a recursive call is executed on the rest of the list (`(machine (rest blocks))`). This is similar in Prolog: the base case is identified: `machine([])` and is processed if matched; otherwise the first element is isolated and processed (line 7-9), and possibly the recursion goes on (lines 10-13).

   Line 16 defines an auxiliary probabilistic term, `flip(Block, P)`, that is true with probability P. The term thus contains in its head its own probability, which is a powerful modeling template. The variable `Block` is added here to circumvent memoization; without this additional variable, `flip` would always have the same truth value. This is clearly unacceptable, since this term is potentially called for each member of `Blocks`, and its truth value must be independent of the precious value. Thus `Block` serves as a unique identifier[1].

[1]This might still lead to problems, when the same block appears several times in the list. A potentially better unique identifier would be the position of the list that is currently being processed



```scheme
(define samples
 (mh-query 100 100
   (define blicket (mem (lambda (block) (flip 0.2))))
   (define (power block) (if (blicket block) 0.9 0.05))

   (define (machine blocks)
     (if (null? blocks)
       (flip 0.05)
       (or (flip (power (first blocks)))
         (machine (rest blocks)))))

   (blicket 'A)

   (machine (list 'A 'B))))

(hist samples "Is A a blicket?")
```

```prolog
% Of Blickets and Blocking
0.2::blicket(B) :- block(B).

power(B, 0.9)  :-    blicket(B).
power(B, 0.05) :- \+ blicket(B).

machine([Block|Blocks]) :-
    power(Block, P),
    flip(Block, P).
machine([Block|Blocks]) :-
    power(Block, P),
    \+flip(Block, P),
    machine(Blocks).
machine([]) :-
    flip(0, 0.05).

P::flip(Block, P).

block(a). block(b). block(c). block(d).

query(blicket(a)).
evidence(machine([a,b]), true).
```

Figure 3.1: Transforming a Church program in a ProbLog program

It is hard to identify all the possible translation templates that can be encountered. However, this chapter exemplifies a few of the most important ones:

- Identifying query and evidence variables, together with the model. These will give the skeleton of the target program.
- Modeling stochastic functions as probabilistic facts or rules.
- Identifying whether functions return a boolean value or something else, and modeling the translation appropriately.
- Translating conditional statements as multiple clauses where the body corresponds to a particular condition.
- Identifying memoization, and certainly situations where the implicit memoization of ProbLog can lead to problems and finding identifiers to circumvent this.
- The use of recursion. This is very similar in a functional language as scheme as in a logic programming language as Prolog.

Chapter 4

Introductory Chapters

Chapters 4 and 5 explain how the programs from [1] are translated to ProbLog. All code is available on my github page: `https://github.com/koenboeckx/ProbLog`.

This chapter focuses on the early chapters of [1]. These chapters don't really try to implement cognitive science principles, but are rather introductory chapters, whose goal is to implement elementary probabilistic programming concepts in Church.

4.1 Generative Models

This chapter examines models that are purely 'generative', meaning there is no conditioning being done. In ProbLog, these models are represented by an ordinary query, without imposing any evidence.

Since this is an introductory chapter, many of the examples are trivial. One of the more interesting examples is the implementation of *stochastic recursion*. In this example, the geometric distribution is defined recursively. The stop condition is based on a stochastic primitive, namely whether or not a flip with weight `p` is true. This Church program is shown below:

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

It is not straightforward to implement this in ProbLog, since the number of generated worlds must be finite. A recursion with a random stop condition cannot achieve this. To solve this, we add an additional stop condition: if the maximum recursion depth is exceeded, the recursion stops and the current value is returned. This is the condition in line 6.

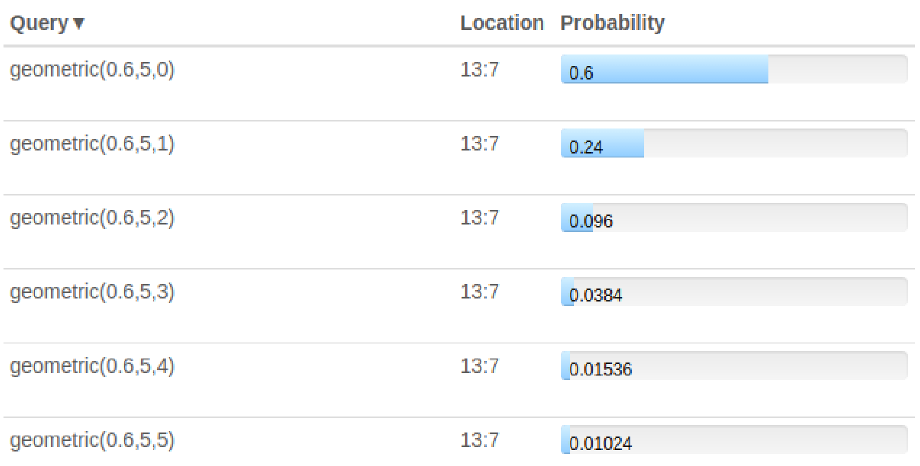
```
1  % Parameters
2  max_depth(5). % maximum recursion depth
3
4  P::flip(P,_).
```

```

geometric(_, 0, 0).    % stop condition 1: max depth reached
geometric(P, N, 0) :- % stop condition 2: flip is true
    N > 0,
    flip(P, N).
geometric(P, N, Result) :-
    N > 0,
    \+flip(P, N), N1 is N-1,
    geometric(P, N1, Res),
    Result is Res+1.

query(geometric(0.6, M, _)) :- max_depth(M).
```

Program 4.1: ProbLog: Geometric Distribution

Also mind the definition of the `flip` fact, where we include an additional argument to avoid memoization. If we don't do this, the coin flip will have the same result at every level of the recursion. This will be a programming construct that we will see throughout this thesis.

The result is shown in figure 4.1.



Query ▾	Location	Probability
geometric(0.6,5,0)	13:7	0.6
geometric(0.6,5,1)	13:7	0.24
geometric(0.6,5,2)	13:7	0.096
geometric(0.6,5,3)	13:7	0.0384
geometric(0.6,5,4)	13:7	0.01536
geometric(0.6,5,5)	13:7	0.01024

Figure 4.1: Result of Stochastic Recursion (geometric distribution)

4.2 Conditioning

This chapter explores a basic feature of both cognition and probabilistic programming: conditioning. More concretely, it explores how our a priori beliefs (as defined by the original probabilistic facts and the model) change when we observe results from the model. This is expressed in the famous Bayes' Rule:

$$P(h|d) = \frac{P(d|h)P(h)}{P(d)} \tag{4.1}$$

where the prior $P(h)$ of hypothesis h is transformed in the posterior $P(h|d)$ through the likelihood $P(d|h)$ of the observation d.

The initial examples show the use of the `query` primitive in Church. This is expressed in the general scheme:

```
 (query
   generative-model           ; model (likelihood)
   what-we-want-to-know       ; posterior of what we are looking for
   (condition what-we-know)) ; observation(s)
```

This is the same scheme as explained in 1.2. In ProbLog, this is implemented in a similar manner:

```
generative_model :- ...
query(what_we_want_to_know).
evidence(what_we_know).
```

This will be the basic layout of most programs in this thesis, and will also determine to basic procedure to approach the problem of porting probabilistic programs from Church to ProbLog. The questions we have to answer are:

1. What do we want to know?
2. What have we observed / what do we know?
3. How do we link the different elements of the program together?

To offer a concrete example, consider one of the first, and simplest examples from this chapter, where the model consists of 3 randomly picked binary values, and a fourth value that is the sum of the previous three. On observing that this latter value is larger or equal than 2, what can we say about some other value? In Church, this becomes:

```
(define (take-sample)
  (rejection-query

   (define A (if (flip) 1 0))
   (define B (if (flip) 1 0))
   (define C (if (flip) 1 0))
   (define D (+ A B C))

   A

   (condition (>= D 2))))
```

Here, the authors use a specific instantiation of `query`, namely `rejection-query`. This will be discussed later in this section.

In this simple program, the answers to the question are:

- We are looking for the value of variable A,
- knowing that the value of D must be larger than or equal to two,
- and we must link, through programming, the probabilistic facts A, B and C to form D

The ProbLog version of this program is shown below:

```
% probabilistic facts
0.5::a(0); 0.5::a(1).
0.5::b(0); 0.5::b(1).
0.5::c(0); 0.5::c(1).

% logic rules
d(D) :-
    a(A), b(B), c(C),
    D is A+B+C.

greater_than_or_equal(G) :-
    d(D), D>=G.

evidence(greater_than_or_equal(2)).
query(a(A)).
```

The similarities between both programs are obvious. A few things are noteworthy:

- Random variables that can take multiple values are best written as terms in an annotated disjunction, with the value as its variable: `a(A)`.
- Other, derived random variables are modeled as terms in which both the variables on which they depend are instantiated (`d(D)`).
- Compositional evidence is modeled by means of an additional term: (`greater_than_or_equal(G)`).

4.2.1 Rejection Query

Church offers several methods for querying, all of which are based on sampling. Later chapters of [1] will use a more advanced version based on *Metropolis-Hasting* inference (see 2.1.4). The aforementioned `rejection-query` is conceptually easier but less efficient. It is basically a recursive sampling method, that returns a sample if it agrees with the observation and keeps sampling if it doesn't. In Church, this is implemented as follows:

```
(define (take-sample)
   (define A (if (flip) 1 0))
   (define B (if (flip) 1 0))
   (define C (if (flip) 1 0))
   (define D (+ A B C))
   (if (>= D 2) A (take-sample)))
```

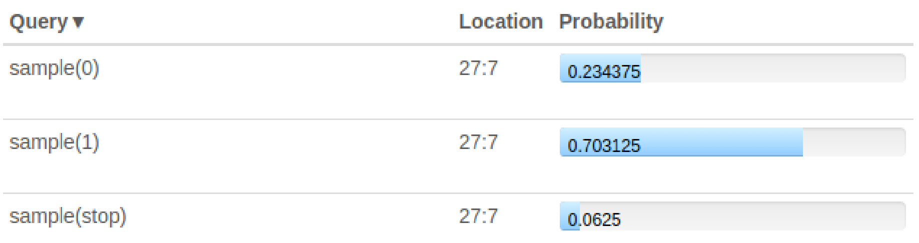
ProbLog is not based on sampling methods; it performs inference by considering all possible worlds and through compilation to a suitable data structure does weighted model counting (see section 2.1.3). This implies that inference is exact, but also that time complexity is related to the number of possible worlds created by the logic program. Unbounded recursion is thus impossible. Nevertheless, it is possible to simulate the `rejection-query` of above in ProbLog. One possible way is shown below:

```
% parameters: max(N) = max length of sequence
max(5). % if > 7 => time requirements blow up

% probabilistic facts
0.5::a(0, ID); 0.5::a(1, ID).
0.5::b(0, ID); 0.5::b(1, ID).
0.5::c(0, ID); 0.5::c(1, ID).

d(D,ID) :-
    a(A,ID), b(B,ID), c(C,ID),
    D is A+B+C.

sample(stop, M) :- max(M). % A sampling sequence that tries
                           % to go beyond pre-defined limit
sample(Sample, ID) :-
    max(M), ID < M,
    a(A,ID), b(B,ID), c(C,ID),
    D is A+B+C,
    (D >= 2, Sample=A;
     D < 2, ID1 is ID+1,
     sample(Sample, ID1)).

sample(S) :-
    sample(S, 1).

query(sample(S)).
```

Program 4.2: Rejection query in ProbLog

This program mirrors its Church counterpart, including the decision point: if the condition is met, stop sampling; otherwise, try to get another sample. Of importance is the use of the ID variable, which is used to circumvent the implicit stochastic memoization of ProbLog.

Since ProbLog creates all possible worlds, we need some mechanism to limit this number of worlds, just as in program 4.1. In program 4.2, this is done with the `max(M), ID<M` term in the recursive method, which explicitly limits how far the sequence of failed sampling can go. This is a nuisance, since it creates an explicit stopping point (represented by the `stop` sample), but also because if the observation points to a rare occurrence, rejection sampling can go quite deep, which means that many possible worlds can be generated. Figure 4.2 shows the distribution of `sample(_)`. Normally, values 0 and 1 should have probability 0.25 and 0.75, respectively. The distribution tends to this real distribution, and this becomes better if we allow for longer sampling sequences. However, a significant portion of the probability mass is still taken up by the `stop` symbol, representing all sequences for which the sampling process was not terminated.



Query ▾	Location	Probability
sample(0)	27:7	0.234375
sample(1)	27:7	0.703125
sample(stop)	27:7	0.0625

Figure 4.2: The result of program 4.2

4.2.2 Tug of War

As a (slightly) more complex example of conditioning, we consider the *Tug-of-War* example. In this example, multiple teams of players engage in a tug-of-war match. The Church code can be found here. First, the query and evidence variables are identified:

- The evidence consists of a sequence of games, for which we know which team won
- We query for the strength of one particular player

The ProbLog implementation (link) can be found below.

```
0.2::strength(Person, 0); 0.2::strength(Person, 1); 0.2::strength(Person, 2);
0.2::strength(Person, 3);0.2::strength(Person, 4) :- person(Person).

1/3::lazy(Person, Game) :- person(Person), game(Game).

pulling([], _, 0).
```

```
7  pulling([Member|Team], Game, P) :-
8      strength(Member, S), \+ lazy(Member, Game), % Team member is NOT lazy
9      pulling(Team, Game, P1),
10     P is P1 + S.
11
12 pulling([Member|Team], Game, P) :-
13     strength(Member, S), lazy(Member, Game),    % Team member is lazy
14     pulling(Team, Game, P1),
15     P is P1 + S/2.
16
17 win(Team1, Team2, Game) :-
18     pulling(Team1, Game, P1),
19     pulling(Team2, Game, P2),
20     P1 >= P2.
21
22 game(game1). game(game2).
23 person(bob). person(mary). person(tom).
24 person(sue). person(jim).
25
26 evidence(win([bob, mary], [tom, sue], game1)).
27 evidence(win([bob, sue],  [tom, jim], game2)).
28
29 query(strength(bob, S)).
```

Program 4.3: ProbLog: Tug-of-War

Each `person` has a randomly selected `strength` (lines 1-2). This annotated disjunction is the translation of the Church expression

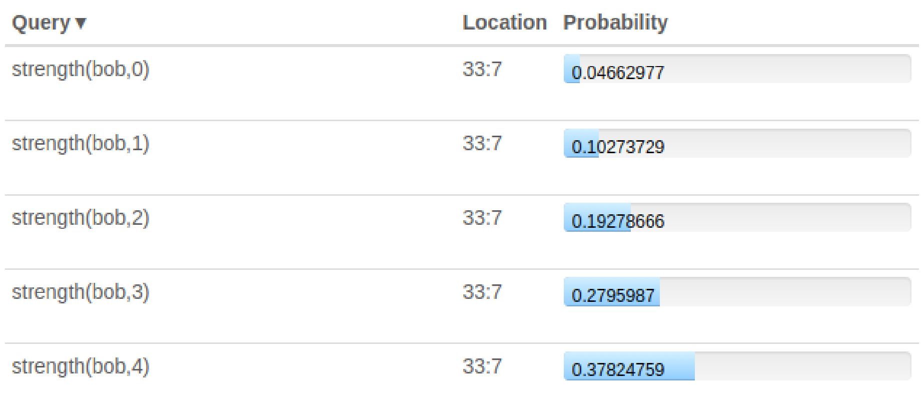
```
(define strength (mem (lambda (person) (gaussian 0 1))))
```

We use a discrete distribution of a person's strength, instead of a Gaussian distribution as in Church. Since ProbLog cannot handle continuous distributions, there is no way around this. An alternative would be to use *Distributional Clauses*. Another difference is the explicit memoization in Church.

Lines 6 to 15 define the pulling strength of a team (a list of persons). In each game, a person can be lazy or not (`lazy(Person, Game)`). If he is lazy, his pulling strength is divided by two. These lines correspond with this Church expression:

```
 (define (total-pulling team)
      (sum
         (map
          (lambda (person) (if (lazy person) (/ (strength person) 2)
                                             (strength person)))
          team)))
```

The result of the query `query(strength(bob, S))` of program 4.3 is shown in figure 4.3.



Query ▾	Location	Probability
strength(bob,0)	33:7	0.04662977
strength(bob,1)	33:7	0.10273729
strength(bob,2)	33:7	0.19278666
strength(bob,3)	33:7	0.2795987
strength(bob,4)	33:7	0.37824759

Figure 4.3: Tug-of-War: what is Bob's strength?

4.3 Patterns of Inference

This chapter explores common inference patterns, and as such, serves a s a basis for later chapters. These patterns include simple statistical dependence, conditional dependence and associated phenomena, like *screening off* and *explaining away*. After a simple example in the area of medical diagnosis, three larger examples are studied: trait attribution, a game called blickets and blocking, and visual perception of a surface lightness and color.

4.3.1 Medical Diagnosis

The initial program is the traditional graphical model for medical diagnosis. It describes a list of symptoms, and their potential causes. It also implements a so-called *noisy-or* model, where a variable can also be activated even if none of its parents are true. This can be used to model unknown causes. The original Church code can be found below.

```
(define samples
 (mh-query
  200 100

  (define smokes (flip 0.2))

  (define lung-disease (or (flip 0.001) (and smokes (flip 0.1))))
  (define cold (flip 0.02))

  (define cough (or (and cold (flip 0.5))
                    (and lung-disease (flip 0.5)) (flip 0.01)))
  (define fever (or (and cold (flip 0.3)) (flip 0.01)))
  (define chest-pain (or (and lung-disease (flip 0.2)) (flip 0.01)))
  (define shortness-of-breath (or (and lung-disease (flip 0.2))
                                        (flip 0.01)))

   (list cold lung-disease)

   cough))

(hist (map first samples) "cold")
(hist (map second samples) "lung-disease")
(hist samples "cold, lung-disease")
```

Program 4.4: Church: Medical Diagnosis

This program is the implementation of a probabilistic graphical model, where potential causes (`smokes`) can induce a number of illnesses (`lung-disease`, `cold`), but we can only observe their symptoms (`cough`, `fever`, `chest-pain`). Since each of these variables is binary, they can easily be modeled by simple probabilistic facts and rules in ProbLog.

This ProbLog code (link) is shown in listing 4.5. It is very similar is spirit and structure as the Church version.

```
% independent variables
0.2::smokes.
0.02::cold.

% dependent variables
0.1::lung_disease :- smokes.
0.001::lung_disease.

% symptoms
0.5::cough :- cold.
0.5::cough :- lung_disease.
0.01::cough.

0.3::fever :- cold.
0.01::fever.

0.2::chest_pain :- lung_disease.
0.01::chest_pain.

0.2::shortness_of_breath :- lung_disease.
0.01::shortness_of_breath.

% for query purposes
both(f,f) :- \+ cold, \+lung_disease.
both(t,f) :- cold, \+lung_disease.
both(f,t) :- \+ cold, lung_disease.
both(t,t) :- cold, lung_disease.

query(cold).
query(lung_disease).
query(both(_,_)).

evidence(cough, true).
```

Program 4.5: ProbLog: Medical Diagnosis

Since it is the direct translation of a probabilistic graphical model, the program only uses grounded facts and rules.

4.3.2 Trait Attribution

The following example (link) examines the issue of trait attribution. This concept explains how humans assign traits, both good and bad, to their fellow humans, and how these assumptions can change, based on additional information. In this specific example, we assume that if somebody fails an exam, she hasn't done her homework. But if we learn that several others also failed that same exam, we might change our opinion and assume that the exam itself wasn't fair.

```
0.8::exam_fair(E) :- exam(E).
0.8::does_homework(S) :- student(S).

0.9::pass(S, E) :- exam_fair(E), does_homework(S).
0.4::pass(S, E) :- exam_fair(E), \+does_homework(S).
0.6::pass(S, E) :- \+exam_fair(E), does_homework(S).
0.2::pass(S, E) :- \+exam_fair(E), \+does_homework(S).

exam(exam1).
student(bill).
student(mary).
student(tim).

% For query purposes:
joint(t, t) :- exam_fair(exam1), does_homework(bill).
joint(f, t) :- \+exam_fair(exam1), does_homework(bill).
joint(t, f) :- exam_fair(exam1), \+does_homework(bill).
joint(f, f) :- \+exam_fair(exam1), \+does_homework(bill).

query(does_homework(bill)).
query(exam_fair(exam1)).
query(joint(_,_)).

evidence(pass(bill, exam1), false).
```

```
%evidence(pass(mary, exam1), false).
%evidence(pass(tim, exam1), false).
```

Program 4.6: ProbLog: Trait Attribution

Without the additional evidence about the result of Mary and Tim, we assume that there's a 50% chance that Bill did do his homework if he failed the exam. After learning that Mary and Tim also failed the same exam, we adjust our opinion and belief that there's a 61% chance Bill did his homework (which is still less than the 80% prior). The probability that the exam was fair, drops from 62,5% when just Bill failed to 22% when all three failed.

4.3.3 Of Blickets and Blocking

Blickets and Blocking is a psychological experiment whereby children's causal learning abilities are assessed by using a 'blicket detector', a toy box that lights up when certain blocks are put on top of it. The children are then asked to infer which blocks are blickets - the blocks that light up the detector - and which are not. The experiment is described in [15].

This example (link) was studied in detail in chapter 3.

4.3.4 Visual Perception of Surface Lightness and Color

This example is about how an observer's perception of the reflectance of a surface changes when she sees that the surface actually lies in the shadow. The Church program is not complicated, but poses a particular problem for ProbLog: it draws samples from a Gaussian distribution. Since ProbLog cannot handle continuous probability distributions, direct translation is not possible. This can be handled however, by discretizing the concerned variables. This was already done in section 4.2.2, but is more complicated and cumbersome in this example.

The line `(define reflectance (gaussian 1 1))` of the Church code is translated in this ProbLog snippet:

```
reflectance(R) :-
    n(N), % how many points?
    gaussian_probs(N, 1.0, 0.5, Vals, Ws),
    select_weighted(reflectance, Ws, Vals, R, _).
```

- The predicate `n(N)` determines how many possible values for the reflectance are considered.
- `gaussian_probs/5` is a clause that takes as input the number of wanted points, the mean and the standard deviation of a Gaussian distribution, and returns two lists:
    1. `Vals`, a list of `N` values, situated within 2 standard deviations from the mean, and

2. Ws, a list with the value of a Gaussian PDF for each of the values in Vals.

This predicate was written in Python for this application, based on the template as from the ProbLog tutorial website (link). This python file, gaussian.py, can be found here.

- The instantiated variables Vals and Ws are consequently used in the select_weighted predicate. This predicate is part of the lists library, and picks randomly a value from Vals, according to the weights Ws, and assigns this value to R. This is an alternative to an annotated disjunction, that can become cumbersome if there are too many values.

The entire code listing is shown in program 4.7.

```
:-use_module('gaussian.py').
:-use_module(library(lists)).

n(11). % the number of points when discretizing

reflectance(R) :-
  n(N),
  gaussian_probs(N, 1.0, 0.5, Vals, Ws),
  select_weighted(reflectance, Ws, Vals, R, _).

illumination(I) :-
  n(N),
  gaussian_probs(N, 3.0, 0.5, Vals, Ws),
  select_weighted(illumination, Ws, Vals, I, _).

luminance(L) :-
  reflectance(R), illumination(I),
  L is R*I.

observed_luminance(L) :-
  n(N),
  gaussian_probs(N, 3.0, 0.1, Vals, Ws),
  select_weighted(observed_luminance, Ws, Vals, L, _).

observation1 :-
  luminance(L),
  observed_luminance(OL),
  L = OL.

lower_illumination(I) :- % in shadow of cylinder
  n(N),
  gaussian_probs(N, 2.0, 0.5, Vals, Ws),
  select_weighted(lower_illumination, Ws, Vals, I, _).

observation2 :-
  illumination(I),
  lower_illumination(LI),
  I = LI.

evidence(observation1).
evidence(observation2).
query(reflectance(_)).
```

Program 4.7: Visual Perception of Surface Lightness and Color

`observation1` is the original observation, namely that the true luminance is noisy-equal to the observed one. `observation2` reflects the knowledge that the illumination is lower in presence of a shadow.

To obtain a result however, a few modifications had to be made:

1. The spread of the reflectance was reduced
2. The loss in illumination was reduced

Unless these changes were made, an `InconsistentEvidence` error was generated, meaning that no world was generated where the evidence was true. The cause was the limited domain and resolution of the different discrete variables who simulate the Gaussian.

Figure 4.4 shows how the distribution of the reflectance evolves when the shadow evidence is added. As expected, when the shadow is present, we belief that the reflectance is higher for the same observed luminance.

4.4 Models for Sequence of Observations

This chapter explores the use of dynamical models, namely models where the state of the different variables evolves as time goes on. This poses a particular problem for ProbLog, because the implicit memoization means that we always have to be mindful of the fact that a variable can be instantiated just once, or many times.

4.4.1 Independent and Exchangeable Sequences

The first examples explore the issues of independence and the related concept of exchangeability. Two random variables A & B are independent if learning the value of A does not change our belief about B.

This first example shows how this works. The Church code is shown below.

```
(define (sequences first-val)
  (mh-query
```

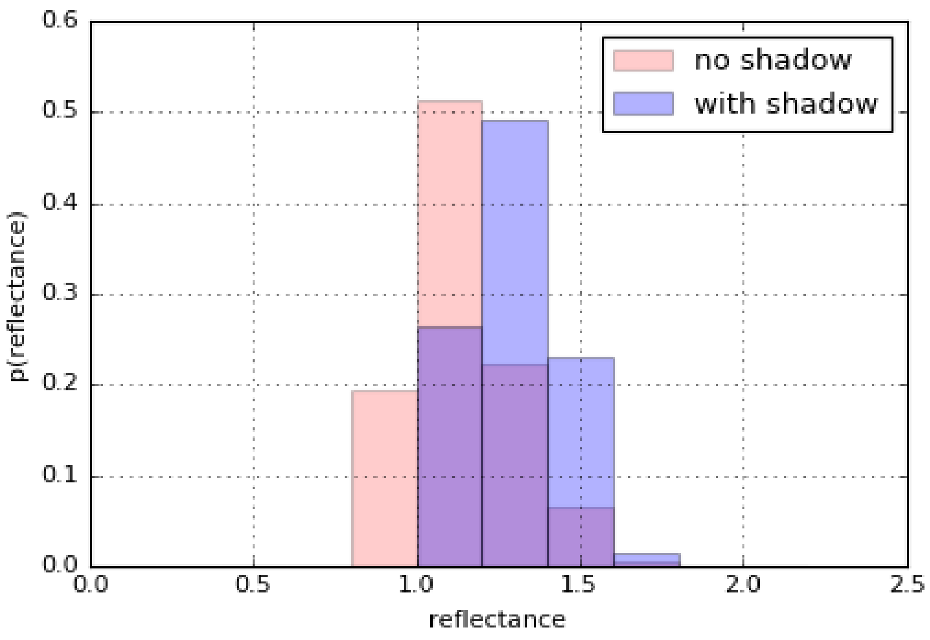


Figure 4.4: Visual Perception of Surface Lightness and Color

```
    1000 10
       (define prob (if (flip) 0.2 0.7))
       (define (myflip) (flip prob))
       (define s (repeat 10 myflip))
       (second s)
       (equal? (first s) first-val)))
```

This program relies on the higher-order `(repeat N foo)` primitive, which generates a list with the result of N calls to the function `foo`. A random function without parameters is called a *thunk*. The code below (link) emulates this with a recursive clause `repeat/3`, where the first argument is the goal we want to execute. The result of this goal should be its first argument. The second is an identifier (the position in the sequence) to independently generate the sample. This is shown in lines 11 to 17.

```
1  % Prior
2  0.5::prob(0.2); 0.5::prob(0.7).  % IMPORTANT: no identifier!!!
3  
4  % Thunk
5  PT::myflip(t, N); PF::myflip(f, N) :- % IMPORTANT: with identifier!!!
6      prob(P),
7      PT is P, PF is 1.0-P.
8  
9  % repeat(Thunk, Times, L).
10 repeat(_, 0, []).
```

```
11 repeat(Thunk, N, [Result|T]) :-
12     N > 0,
13     call(Thunk, Result, N),
14     N1 is N-1,
15     repeat(Thunk, N1, T).
16
17 % Aux functions
18 first(H) :-
19     sequence([H|_]).
20 second(H) :-
21     sequence([_,H|_]).
22
23 sequence(S) :-
24     repeat(myflip, 3, S).
25
26 %evidence(first(f)). % uncomment to see impact
27 query(second(_)).
```

The program then proceeds by applying this `repeat` clause to the `myflip` thunk.

The goal of this example is to show how evidence impacts the belief about the variables. With evidence about the first element of the sequence, the distribution of the second element is:

- P(second = false) = 0.55
- P(second = true) = 0.45

When we add evidence about the first element of the sequence (by uncommenting line 26), the distribution of the second element changes:

- P(second = false) = 0.66
- P(second = true) = 0.33

Thus the second element is not independent from the first element. However, this model has a weaker property: *exchangeability* ([1]): the probability of a sequence is the same if its elements are rearranged.

4.4.2 Polya Urn Model

The second problem examines the Polya Urn model. The first implementation is the classical model, while the second implementation is an implementation based on the *de Finetti* representation.

In the Polya urn problem, there is an urn containing white and black balls. At each turn, we pick a ball from the urn and put a fixed number of balls of the same color as the picked ball back in the urn. This model can be directly described as in the code below (link).

```
P1::pick(Nsample, w, Urn); P2::pick(Nsample, b, Urn) :-
    Urn = urn(Nw, Nb),
    P1 is Nw/(Nw + Nb),
    P2 is Nb/(Nw + Nb).

% samples(Urn, Nreplace, Nsamples, Samples)
samples(_,_,0,[]).
samples(urn(Nw,Nb), Nreplace, Nsamples, [Sample|OtherSamples]) :-
    pick(Nsamples, Sample, urn(Nw, Nb)),
    (Sample = w,
        Add_white is Nreplace-1, Add_black is 0;
    Sample = b,
        Add_white is 0, Add_black is Nreplace-1
    ),
    NewWhite is Nw + Add_white,
    NewBlack is Nb + Add_black,
    NewSamples is Nsamples-1,
    samples(urn(NewWhite,NewBlack), Nreplace, NewSamples, OtherSamples).

query(samples(urn(1,2), 4, 3, [_,_,_])).
```

Program 4.8: ProbLog: classical Polya urn model

The result of this query, where we pick 3 balls from an urn with originally 1 white and 2 black balls, is:

```
samples(urn(1,2),4,3,[b, b, b]): 0.49382716
samples(urn(1,2),4,3,[b, b, w]): 0.061728395
samples(urn(1,2),4,3,[b, w, b]): 0.061728395
samples(urn(1,2),4,3,[b, w, w]): 0.049382716
samples(urn(1,2),4,3,[w, b, b]): 0.061728395
samples(urn(1,2),4,3,[w, b, w]): 0.049382716
samples(urn(1,2),4,3,[w, w, b]): 0.049382716
samples(urn(1,2),4,3,[w, w, w]): 0.17283951
```

This result shows the exchangeability of the samples: the three sequences containing 2 black and 1 white ball, for example, all have the same probability.

Since the resulting distribution over sequences is exchangeable, we should also be able to model it with the de Finetti method. The de Finetti theorem states that each exchangeable sequence can be represented as follows:

```
 P::latent_prior(...) :- ...
 P1::thunk(V1, N); ...; Pk::thunk(Vk, N) :-
  latent_prior(...),
  probs(P1, ..., Pk, P).
repeat(Thunk, L, Result).
```

whereby `Result` is an exchangeable sequence of length L. This method is used to implement the Polya urn model in the example below (link). Since this program needs a Beta distribution, a method to use this distribution was implemented, based on the `select_weighted` method from the `lists` library [20] (lines 4-14). The `repeat/3` is re-written as a tail-recursive clause.

```
1   :-use_module(library(lists)).
2
3  %beta_weight(+X,+A,+B,?W).
4  beta_weight(X,A,B,W) :-
5      P1 is X**(A-1),
6      P2 is (1.0-X)**(B-1),
7      W is P1*P2.
8
9  %apply_beta(+List,+A,+B,?ListOut).
10 apply_beta([],_,_,[]).
11 apply_beta([In|InRest],A,B,[Out|OutRest]) :-
12     beta_weight(In,A,B,Out),
13     apply_beta(InRest,A,B,OutRest).
14
15 values([0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]).
16 %select_weighted(ID, Weights, Values, Value, Rest)
17 b(X,A,B) :-
18     values(Values),
19     apply_beta(Values,A,B,Weights),
20     select_weighted(b(A,B), Weights, Values, X, _).
21
22 latent_prior(Prob, A, B) :- b(Prob, A, B).
23
24 % urn(NWhite, NBlack, NReplace)
25 urn(1,2,4).
26
27 PB::thunk(b,N); PW::thunk(w,N) :-
28     urn(NWhite, NBlack, NReplace),
29     A is NBlack/NReplace,
30     B is NWhite/NReplace,
31     latent_prior(P,A,B),
32     PB is P, PW is 1.0-P.
33
34 % repeat(Thunk, N, Result). -> with Tail Recursion
35 repeat(Thunk,N,Result) :-
36     repeat(Thunk,N,[],Result).
37 repeat(_,0,Acc,Acc).
38 repeat(Thunk, N, Acc, Result) :-
39     N > 0,
    call(Thunk, Sample, N),
    N1 is N-1,
    repeat(Thunk, N1, [Sample|Acc], Result).

query(repeat(thunk, 3, [_,_,_])).
```

Program 4.9: ProbLog: Polya urn model via de Finetti

4.4.3 Markov Models

Markov models are dynamical statistical models where an observation only depends on the previous observation. A simple example is given below, where the transition matrix from $State_{N-1}$ to $State_N$ is given explicitly. Traversing the sequence is done with recursion.

```
1   % Markov chain
2
3  % flip(New, Prev).
4  0.9::flip(t, t, N); 0.1::flip(f, t, N).
5  0.1::flip(t, f, N); 0.9::flip(f, f, N).
6
7  markov(0, _, []).
8  markov(N, Prev, [New|Chain]) :-
9      N > 0,
10     N1 is N-1,
11     flip(New, Prev, N),
12     markov(N1, New, Chain).
13
14 query(markov(3, t, M)).
```

The transition matrix is modeled with the clauses in lines 4 and 5. The Markov chain is constructed with the markov predicate, that recursively builds up a chain until the maximum length is reached.

Another example of a Markov model is a language model. The example below shows a simple language model, where the state transition matrix is given explicitly:

```
 % Markov chain

% next_word(CurrWord, NextWord, T)
0.0032::next_word(start, chef, T); 0.4863::next_word(start, omelet, T);
0.0789::next_word(start, soup, T); 0.0675::next_word(start, eat, T);
0.1974::next_word(start, work, T); 0.1387::next_word(start, bake, T);
0.0277::next_word(start, stop, T).

...

```

```
markov(_, stop, []).
markov(T, Curr, [New|Chain]) :-
    T > 0,
    T1 is T-1,
    next_word(Curr, New, T),
    markov(T1, New, Chain).

query(markov(3, start, M)).
```

The important part is thus the definition of the transition matrix, since the code for generating the sequence is exactly the same.

This model can be used for sampling (something similar will be done with a *Context-Free Grammar* later on), but it can also be used to determine the entire probability distribution over sequence of a certain length. The following list gives this distribution over all sequences of length 2, knowing that we added a `stop` symbol to indicate the end of a sentence:

```
markov(3,start,[bake, bake, stop]):     0.00053288401
markov(3,start,[bake, chef, stop]):     1.2294368e-05
markov(3,start,[bake, eat, stop]):      0.00025933433
markov(3,start,[bake, omelet, stop]):   0.0018683597
markov(3,start,[bake, soup, stop]):     0.00030313301
     ...
markov(3,start,[work, soup, stop]):     0.00043142362
markov(3,start,[work, stop]):           0.00546798
markov(3,start,[work, work, stop]):     0.0010793793
```

4.4.4 Subjective randomness

This example (link) explores which sequences we consider to be truly random, and which not. We model two type of sequences: those generated by a fair coin, and the ones generated by a biased coin. As evidence, we introduce a few sequences of coin throws, and then we ask whether or not these throws were generated by a fair or by a biased coin.

The Church code is fairly straight-forward (see listing 4.11). A model is constructed with a coin that is fair or unfair (line 3) and this coin is then used to construct a sequence (implicitly in line 5). The query looks at the prior `isfair`.

```
1  (query
2     (define isfair (flip))
3     (define (coin) (flip (if isfair 0.5 0.2)))
4     isfair
5     (condition (equal? sequence (repeat 5 coin))))
```

Program 4.10: Church: Subjective Randomness

The ProbLog implementation follows the same template: construction of a prior on coin fairness, generation of the coin and subsequently the generation of the sequence of flips.

```
0.5::is_fair(C).
0.5::flip(N, C) :-
    is_fair(C).
0.2::flip(N, C) :-
    \+ is_fair(C).

sequence(0, _, []).
sequence(N, Coin, [true|Rest]) :-
    N > 0,
    flip(N, Coin),
    N1 is N-1,
    sequence(N1, Coin, Rest).

sequence(N, Coin, [false|Rest]) :-
    N > 0,
    \+ flip(N, Coin),
    N1 is N-1,
    sequence(N1, Coin, Rest).

% is 10010 fair?
evidence(sequence(5, coin, [true, false, false, true, false]), true).
query(is_fair(coin)).
```

Program 4.11: Church: Subjective Randomness

The queries were repeated 4 times, for 4 different sequences:

1. Query 1 - Sequence 10010: `is_fair(c1)`: 0.6040982
2. Query 2 - Sequence 00000: `is_fair(c2)`: 0.0870643
3. Query 3 - Sequence 01010: `is_fair(c3)`: 0.6040982
4. Query 4 - Sequence 01100: `is_fair(c4)`: 0.6040982

This model correctly believes that ’00000’ is not the result of a fair coin. However, this model also implies that ’01010’ is produced by a coin that is as fair as the one that produced ’01100’: the model is exchangeable: sequence with the same number of 0’s and 1’s are equally likely. Most people, however, would say that that ’01010’ is less random than ’01100’ or ’10010’, because the former has a repetitive structure. This can be resolved by integrating a transition model that believes that alternating sequences are less random:

```
0.5::is_fair(C).

0.5::transition(N, Coin, true) :-
    is_fair(Coin).
0.5::transition(N, Coin, false) :-
    is_fair(Coin).
0.1::transition(N, Coin, true) :-
    \+is_fair(Coin).
0.9::transition(N, Coin, false) :-
    \+is_fair(Coin).


sequence(0, _, _, []).
sequence(N, Coin, Prev, [true|Rest]) :-
    N > 0,
    transition(N, Coin, Prev),
    N1 is N-1,
    sequence(N1, Coin, true, Rest).

sequence(N, Coin, Prev, [false|Rest]) :-
    N > 0,
    \+ transition(N, Coin, Prev),
    N1 is N-1,
    sequence(N1, Coin, false, Rest).

% is 10010 fair?
evidence(sequence(5, coin, [true, false, false, true, false]), true).
query(is_fair(coin)).
```

The results of these queries are:

1. Query 1 - Sequence 10010: `is_fair(c1)`: 0.70427296
2. Query 2 - Sequence 00000: `is_fair(c2)`: 0.99942433
3. Query 3 - Sequence 01010: `is_fair(c3)`: 0.20924284
4. Query 4 - Sequence 01100: `is_fair(c4)`: 0.95542375

This model considers '01010' to be a lot less fair than '01100'. However, it believes that '00000' is random.

4.4.5 Probabilistic Context-Free Grammars

Probabilistic Context-Free Grammars (PCFG) are context-free grammars where the choice to apply a production rule is done stochastically. Development of a CFG in Prolog is relatively straightforward (see [9]). The extension to ProbLog is also obvious: each production rule is modeled as a probabilistic rule. The difficulty is setting an identifier. This is absolutely necessary. Otherwise, when certain production rules, like generating a noun, are called multiple times in sentence they will always produce the same result. That is clearly not wanted.

We cannot simply rely on the position in the sequence of a word as the identifier, since the construction of a sentence is not linear, but happens in a tree. Therefore, an `id(List)` functor was created and propagated through the production tree. Every leaf of the tree, where the non-terminal symbols are generated, will thus get a unique ID. The complete ProbLog program is shown below (link).

```
% 1. Terminal
0.5::d([the], ID); 0.5::d([a], ID).
1/3::n([chef], ID); 1/3::n([soup], ID);1/3::n([omelet], ID).
0.5::v([cooks], ID); 0.5::v([works], ID).
a([diligently], ID).

% 2. Non-Terminal
ap(AP, id(ID)) :-
    a(AP, id([left|ID])).
np(NP, id(ID)) :-
    d(D, id([left|ID])), n(N, id([left|ID])),
    append(D,N,NP).
0.5::vp1; 0.5::vp2. % choose between vp -> v ap or vp -> v np
vp(VP, id(ID)) :-
    vp1,
    v(V, id([left|ID])), ap(AP, id([right|ID])),
    append(V, AP,VP).
vp(VP, id(ID)) :-
    vp2,
    v(V,id([left|ID])), np(NP, id([right|ID])),
    append(V,NP,VP).
s(S) :-
    np(NP, id([left])), vp(VP, id([right])),
    append(NP,VP,S).

% auxiliary
append([],L,L).
append([H|T], L1, [H|L2]) :-
    append(T, L1, L2).

query(s(S)).
```

Program 4.12: ProbLog: Probabilistic Context-Free Grammars
This program is best used with the sampling functionality of ProbLog:

```
problog sample -N 3 PCFG.pl
```

A few examples of this command:

```
s([a, chef, works, diligently]).
s([the, soup, works, a, chef]).
s([a, omelet, cooks, the, soup]).
```

Chapter 5

Specific Chapters

This chapter explores the later chapters of *Probabilistic Models of Cognition*. These are chapters more tailored to the cognitive processes themselves, instead of explanations how certain common patterns of probabilistic programming are implemented in Church. As a consequence, these chapters also contain harder programming problems.

An important addition in these chapters, and certainly from chapter 9 (Hierarchical Models) onwards, is the use of continuous distributions as distributions on the parameters of other probability distributions. Because of the hierarchical structure, it was no longer evident to model this in ProbLog, as was done previously when continuous distributions were encountered. Hence, Distributional Clauses were used to deal with the later chapters.

Chapter 7 of [1] is not considered in this thesis, since it concerns the implementation of Metropolis-Hastings in Scheme and is thus not about modeling. However, a few examples have been translated to ProbLog and can be found in the github repository.

5.1 Inference about Inference

This is chapter deals with the nesting of the Church `query` operator. Using this operator inside another `query` represents hypothetical inference about a hypothetical inference ([1], chapter 6). This can for example be used to reason about an external agent, who herself is a rational reasoner. This process is also known as *social cognition*.

ProbLog does not allow to nest the `query` predicate. Nevertheless, an alternative in ProbLog to handle this type of problems is the `subquery` predicate, which allows to ask queries inside a program. The usage is:

```
subquery(+query(QueryVar), ?P) % or
subquery(+query(QueryVar), ?P, +[Evidence1, ...])
```

The term instantiates the variable `P` with the probability of `query(QueryVar)` being true, potentially given evidence `[Evidence1, Evidence2, ...]`.

This section is based on chapter 7 of [1].

5.1.1 Prelude: Thinking about Assembly Lines

This example is set up to introduce the nested query, specifically the use of a nested rejection query. It deals with a widget factory, that produces unnamed widgets. The inner query only allows widgets that exceed a certain threshold. The outer query seeks to find this unknown threshold, based on a sequence of observations of widgets that passed the test. The Church code is this:

```
 1  (define (sample)
 2  (rejection-query
 3
 4    ;;this machine makes a widget, which we'll just represent with a real number:
 5    (define (widget-maker)  (multinomial '(.2 .3 .4 .5 .6 .7 .8)
 6                                         '(.05 .1 .2 .3 .2 .1 .05)))
 7
 8    ;;this machine tests widgets as they come out of the widget-maker, letting
 9    ;; through only those that pass threshold:
10    (define (next-good-widget)
11      (rejection-query
12       (define widget (widget-maker))
13       widget
14       (> widget threshold)))
15
16    ;;but we don't know what threshold the widget tester is set to:
17
18    (define threshold  (multinomial '(.3 .4 .5 .6 .7) '(.1 .2 .4 .2 .1)))
19
20    ;;what is the threshold?
21    threshold
22
23    ;;if we see this sequence of good widgets:
24    (equal? (repeat 3 next-good-widget)
25            '(0.6 0.7 0.8))))
```

The outer query is actually the entire definition of the `sample` function, but more specifically the query on line 21, together with the observation of 3 widgets (lines 24 and 25). The inner query is the rejection query on lines 10-14, whereby widgets are produced and only those that exceed the threshold are returned.

The code below (link) shows the ProbLog code:

```
:-use_module(library(lists)).

widget(Widget, ID) :-
    select_weighted(widget(ID), [.05,.1,.2,.3,.2,.1,.05],
                    [.2,.3,.4,.5,.6,.7,.8], Widget, _).

```

```
 7 threshold(Threshold) :-
 8     select_weighted(threshold, [.1,.2,.4,.2,.1],
 9                     [.3,.4,.5,.6,.7], Threshold, _).
10
11 next_good_widget(Widget, ID) :-
12     threshold(Threshold),
13     subquery(widget(Widget, ID), P, [good(Threshold, ID)]),
14     pr(Widget, Threshold, ID, P). % IMPORTANT!
15
16 P::pr(_,_,_,P).
17
18 good(Threshold, ID) :-
19     widget(Widget, ID),
20     Widget > Threshold.
21
22 query(threshold(_)).
23
24 evidence(next_good_widget(.6, 1)).
25 evidence(next_good_widget(.7, 2)).
26 evidence(next_good_widget(.8, 3)).
```

A few comments on the similarities and differences between both programs:

- `widget-maker` and `threshold` are readily reproduced in ProbLog: respectively the predicates in lines 3-5 and 7-9 are the translations of these Church functions, which in turn are only encapsulations of multinomial distributions. It is important that the `widget/2` predicate in ProbLog uses an identifier to circumvent memoization, since we will be observing multiple widgets.

- The inner Church query is translated to the `next_good_widget(+Widget, +ID)` predicate. This predicate does three important things:

  1. It instantiates the threshold to the value that it has in the world : `threshold(Threshold)`
  2. It uses the `subquery` predicate to query the probability of the widget, given the evidence. The evidence says that for the current threshold, the widget is *good* (`Widget > Threshold`).
  3. The `subquery` predicate returns the probability P of the query. With help of the `pr(_,_,_,P)` term, this probability is associated with the `next_good_widget` predicate. It is important that this probability is linked to the widget, the threshold and the ID, to avoid any unwanted memoization issues.

  In this form, the `next_good_widget` predicate can NOT be used to generate the widget, but can only be used when the widget is observed. Thus we must use the predicate with all its variables instantiated. For our purposes though, this is enough, because we only use it as part of the evidence.

- We query for the distribution of the `threshold`, which corresponds directly to line 21 of the Church code.
- The evidence is a sequence of three widgets that are 'good'. It is important to emphasize that, for example, observing two instead of 1 widget with value 0.6 has a significant impact on the posterior distribution. Observing a second widget of value 0.6 namely makes it more likely that the threshold is 0.5 and not lower, since otherwise, we might have observed a widget of 0.5, for example. Furthermore, since we observed a widget of 0.6, the threshold can never be 0.6 or higher.

The result of the Church program (with 200 samples) is shown in figure 5.1. There is a strong preference for a threshold value of 0.5, although 0.4 and 0.3 cannot be excluded.

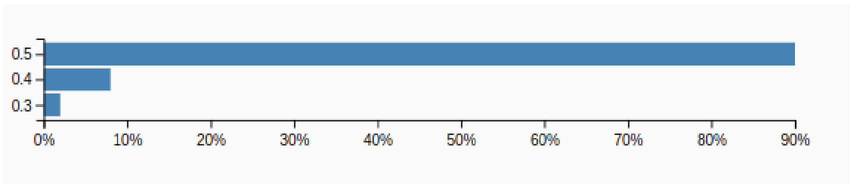


Figure 5.1: The widget factory in Church

ProbLog gives the exact values:

- P(Threshold = 0.3) = 1.6%
- P(Threshold = 0.4) = 7.1%
- P(Threshold = 0.5) = 91.3%

These values correspond very closely with the approximate values of Church.

5.1.2 Goal Inference

In this section, the aim is to infer the goal of someone, based on the actions they take. The running example is a vending machine: it has two buttons (a and b), and gives out food items (a cookie or a bagel) in a stochastic manner. In this scenario, it is assumed that the transition matrix of the vending machine is known.

Ordinarily, this is an easy problem, since if somebody (call her Sally) wants a cookie, she will pick the button that is most likely to give a cookie. The example below however, has a faulty vending machine: the button b is broken, and gives out randomly bagels or cookies.

The code below (link) is the translation of the Church version and implements the model explained above. The question we ask is: knowing the workings of the vending machine, and given that Sally chooses button b, what is the food item she wants? The program is very similar in structure to the assembly line example of the previous section.

```
choose_action(Action) :-
    goal(Goal),
    subquery(action(Action), P, [inner(Goal)]),
    pr(Action, Goal, P).
P::pr(_,_,P).

inner(Goal) :-
    action(Action),
    transition(Action, Goal).

0.5::action(a); 0.5::action(b).
0.5::goal(bagel); 0.5::goal(cookie).

% the transition matrix
0.9::vending_machine(a, bagel); 0.1::vending_machine(a, cookie).
0.5::vending_machine(b, bagel); 0.5::vending_machine(b, cookie).

transition(Action, Result) :- vending_machine(Action, Result).

query(goal(_)).

evidence(choose_action(b)).
```

Even though the result of button b is totally random, the result of the query gives a strong preference for cookies:

- P(goal_food = bagel) = 0.3
- P(goal_food = cookie) = 0.7

The reason for this phenomenon is that the algorithm implicitly takes *negative evidence* into account: if Sally would have wanted a bagel, she would have pushed button a, since this one is much more likely to give a bagel. Hence, since button b was pushed, the cookie is much more likely.

5.1.3 Preferences

We now consider the situation where we know that Sally has a preference; we only don't know what it is. A higher-level prior on the preference is introduced (lines 15-17 in 5.1). We observe the actions Sally takes, on try to infer her preferred food. In the listing below (link) , we observe her pushing button `b` several times. What is her goal?

```
1  :-use_module(library(lists)).
2  
3  choose_action(Action, ID) :-
4      goal(Goal,ID),
5      subquery(action(Action), P, [inner(Goal)]),
6      pr(Action, Goal, ID, P).
7  P::pr(_,_,_,P).
8  
9  inner(Goal) :-
10     action(Action),
11     transition(Action, Goal).
12 
13 values(Values, N) :- % select N values between 0 and 1
14     findall(Y, (between(0, N, X), Y is X/N), Values).
15 preference(Preference) :-
16     values(Values, 30),
17     select_uniform(preference, Values, Preference, _).
18 
19 0.5::action(a); 0.5::action(b).
20 P::flip(N) :- preference(P).
21 goal(bagel, ID) :- flip(ID).
22 goal(cookie,ID) :- \+flip(ID).
23 
24 
25 0.9::vending_machine(a, bagel); 0.1::vending_machine(a, cookie).
26 0.1::vending_machine(b, bagel); 0.9::vending_machine(b, cookie).
27 
28 transition(Action, Result) :- vending_machine(Action, Result).
29 
30 query(goal(_,4)). % Set N > 3, to have 'unbiased' distribution
31                   % (= not directly linked to observation)
32 
33 evidence(choose_action(b, 1)). % or
34 %evidence(goal(cookie, 1)).
35 evidence(choose_action(b, 2)).
36 evidence(choose_action(b, 3)).
```

Program 5.1: ProbLog: Reasoning about preferences

Lines 13-14 use a common construct to create a list of numbers between 0 and 1, to approximate a continuous distribution over this interval. This example is actually a simple *hierarchical model*, which will be studied in detail in section 5.3. In this case the structure is simple: instead of having two equally likely goals (`goal(bagel, _)` or `goal(cookie, _)`, we impose a prior on the parameter of their distribution. This *preference parameter* is picked uniformly for the range [0,1]. Bagels are more likely if `Preference` is close to 1, and cookies when `Preference` is closer to zero. This thanks to the `flip/1` predicate. After observing 3 pushes on the button b (lines 33-36), which is much more likely to give a cookie than a bagel, we start to believe that Sally prefers cookies over bagels. This can be seen by querying on the values of `preference(_)`.

In the example above, we actually query on the goal of Sally. It is important to set the ID value of the goal to a value different than one used in the evidence, because what is important is what we know about her next choice, not the previous one. This is another case of circumventing unwanted memoization. The resulting values are:

- P(bagel) = 21.1%
- P(cookie) = 78.9%

although these values change slightly if the resolution of the preferences - 30 in the code - is changed.

This result corresponds with the one found in [1].

5.1.4 Epistemic States

In this chapter, the concept of social cognition is taken a step further: we assume we don't know how the vending machine works, but that Sally does know. Based on her actions and goal, we try to infer the workings (parameters) of the vending machine.

- We want to know the probabilities of the resulting foods of pushing buttons a and b. We put a uniform prior on these effects[1].
- We know that Sally wants a cookie, and observe that she pushes button b.
- We define the vending machine with the help of the predicates `a_effects` and `b_effects`.

The ProbLog code (link) is shown below:

```
:-use_module(library(lists)).

choose_action(Action) :-
    goal(Goal), transition(Action, Goal),
    subquery(action(Action), P),
    pr(Action, Goal, P).
P::pr(_,_,P).

inner(Goal) :-
    action(Action),
    transition(Action, Goal).
```

[1]in [1], the authors use a Dirichlet(1,1) distribution, but this is the same as a uniform distribution

```

values(Values, N) :- % select N values between 0 and 1
    findall(Y, (between(0, N, X), Y is X/N), Values).

0.5::action(a);   0.5::action(b).
0.5::goal(bagel); 0.5::goal(cookie).

a_effects(P) :-
    values(V, 10),
    select_uniform(a_effects, V, P, _).
b_effects(P) :-
    values(V, 10),
    select_uniform(b_effects, V, P, _).

Pb::vending_machine(a, bagel); Pc::vending_machine(a, cookie) :-
    a_effects(P), Pb is P, Pc is 1.0-P.
Pb::vending_machine(b, bagel); Pc::vending_machine(b, cookie) :-
    b_effects(P), Pb is P, Pc is 1.0-P.

transition(Action, Result) :- vending_machine(Action, Result).

evidence(goal(cookie)).
evidence(choose_action(b)).

query(b_effects(_)).
```

A few comments:

- The `choose_action/1` predicate is defined differently: the `transition` term is pulled from the evidence, and put in the body of the predicate. If we would not do this, we could not influence the effects through the evidence. It is namely so that evidence in the `subquery/3` predicate is only used in a local copy created by the predicate.

- `a_effects` and `b_effects` define the parameters of the (multinomial) vending machine, and are based on uniform sampling from a set of values whose length can be determined. This is our standard way in ProbLog to approximate a continuous distribution.

- The parameters of the probabilistic rule `vending_machine/2` are only instantiated in its body.

The results of this program are similar to its Church counterpart. The values of `a_effects` are not impacted by the observation. However, by taking the implicit evidence into account (if Sally believes she should push b to get a cookie, button is probably more likely to give a bagel), the effect of button a should tilt to returning a cookie. This should be studied further and as such, this example is not complete.

In [1], the authors take the concept of inferring the workings of a machine based on the observation of somebodies action one step further. They change the vending machine: it now only has one button, and the food item you get is determined by the number of pushes. This change concerns the definition of the `action` predicate, while the rest of the program can stay virtually the same. The code below shows the new definition of the action predicate. Basically, at each time step N, we evaluate whether to stop (`next(stop, N)`) or not (`next(go_on, N)`). We limit the length of an action (here below, the maximum length is 3), which is why we need an additional stop condition, when N is zero. The complete code for this program can be found here.

```
% ...
0.3::next(go_on, N); 0.7::next(stop, N).

action([], N) :- next(stop, N).
action([], 0).
action([a|R], N) :-
    next(go_on, N), N1 is N-1,
    action(R, N1).

action(Action) :-
    between(1, 3, N),
    length(Action, N),
    action(Action, N).
% ...
```

5.1.5 A Communication Game

This section explores a two-player game, where both players goal is to infer the actions of the other one. Concretely, a teacher selects randomly a weighted die, and he has to communicate which die he picked to the learner. However, he can only give him one face of the die. Both players know the dies and their weights.

In Church, the authors set up two functions, `teacher` and `learner`, that each query the other in a recursive fashion:

```
 (define (teacher die)
  (query
   (define side (side-prior))
   side
   (equal? die (learner side))))

 (define (learner side)
  (query
   (define die (die-prior))
   die
   (equal? side (teacher die))))
```

A similar setup can be achieved in ProbLog:

```
 %% teacher(+Die, ?Side).
P::teacher(Die, Side) :-
    side(Side),
    subquery(die(Die), P, [learner(Side, Die)]).

%% learner(+Side, ?Die)
P::learner(Side, Die) :-
    die(Die),
    subquery(side(Side), P, [teacher(Die, Side)]).
```

However, a few precautions are in order:

- The recursion above is infinite; to halt it, we introduce a `Depth` parameter which stops the recursion when depth 0 is reached. This was also done in the Church version. At that moment, the learner assumes that the teacher just rolled the die, and communicated the side that came up.
- We must make certain that the evidence is both feasible and instantiated at all times. This required the addition of new predicates `learner_evidence` and `teacher_evidence`.

The entire Problog program can be found here. We also had to introduce additional evidence to ensure that the probabilities would some up to one. The evidence and query are shown in the snippet below:

```
received(Side) :- % introduce the given side as evidence
    depth(D),     % this will normalize the result
    learner(Side, _, D).
evidence(received(green)).

query(learner(green, _, D)) :-
    depth(D).
```

The `received` predicate says that at at his reasoning depth `D`, the learner is shown a green side. This is submitted as evidence. If no evidence is introduced, the result would be the joint probability $P(Side = green, Die = Die)$. The two resulting values for $Die = a$ and $Die = b$ are not normalized, since there is also probability mass for other values of $Side$. By submitting $Side = green$ as evidence, we reassign this probability mass to $Side = green$ and effectively compute the conditional distribution $P(Die = Die|Side = green)$.

Since die a cannot give a red side when rolled, this results in this phenomenon: when the reasoning depth is 0, and the learner receives a green side, he believes that die b is more likely, since it has a higher probability of giving green. However, if the reasoning depth is increased, the learner thinks that die a is more likely than b; he namely believes that is b was the true die, the teacher would show him a red side, since die a cannot result in a red side. Since the teacher does not do this, the learner increases his confidence in die a.

For a depth of zero, the resulting probabilities are:

- `learner(green,a,0): 0.4`
- `learner(green,b,0): 0.6`

and for depth 1:

- `learner(green,a,1): 0.58196721`
- `learner(green,b,1): 0.41803279`

These are also the values obtained in Church. Increasing the reasoning depth leads to ever more confidence in die a.

5.1.6 Communicating with Words

In [1], the previous example is extended to a concrete example, where a speaker tries to communicate a certain state by using certain words - exactly how humans communicate. Almost exactly the same setup can be used, where teacher is replaced by speaker, learner by listener, die by state and side by words. Concretely, the speaker tries to communicate how many plants have sprouted by using the words none, some or all. The ProbLog code, almost an identical copy of the previous example, can be found here.

- `listener(some,0,1): 0`
- `listener(some,1,1): 0.46153846`
- `listener(some,2,1): 0.46153846`
- `listener(some,3,1): 0.076923077`

By construction, hearing 'some' cannot mean 0. But the basic meaning of some can mean equally likely 1, 2 or 3. However, if the speaker would have wanted to communicate 3, he would likely have said 'all'. Thus, after one layer of reasoning, the probability of 3 is significantly reduced in favor of 1 and 2, who remain equally likely.

5.2 Learning as conditional Inference

In cognition, there is no hard line between reasoning on the one hand and learning on the other. Both can be seen as a form of conditional inference ([1]). The learning task is defined as inference in a model that has both a latent value of interest, the *hypothesis*, and a sequence of observations, the *data points*. In Church, the general structure of such a model is:

```
(query
 (define hypothesis (prior))
 hypothesis
 (equal? observed-data (repeat N (lambda () (observe hypothesis)))))
```

This section is based on chapter 8 of [1].

5.2.1 Learning about Coins

To sketch how a similar method can be implemented in ProbLog, consider the following example (link), where we learn whether a coin is fair, based on our prior assumption of fairness of the coin, and a sequence of observation that challenge our prior assumption. We assume initially with 99,9 % certainty that the coin we're going to flip is fair. However, an observation of several consecutive heads makes us doubt this quasi-certain prior assumption:

```
0.999::fair_coin.
trick_coin :- \+fair_coin.

0.5::coin_flip(h,N); 0.5::coin_flip(t,N) :-
    fair_coin.
0.95::coin_flip(h,N); 0.05::coin_flip(t,N) :-
    \+fair_coin.

observe([],0).
observe([Face|T], N) :-
    coin_flip(Face,N),
    N1 is N-1,
    observe(T, N1).

query(fair_coin).
query(trick_coin).

observed_sequence([h,h,h,h,h,h,h,h,h,h,h,h,h,h,h]). % observed sequence
evidence(observe(S, N)) :-
    observed_sequence(S),
    length(S,N).
```

Program 5.2: ProbLog: Learning about coins

The figure 5.2 shows the so-called *learning curve*, namely how the posterior of the random variable `fair_coin` evolves with increasing evidence. With a few observed consecutive heads, we stay close to our initial prior of 0.999. As the number of heads increases, our posterior probability falls, and from 15 consecutive heads, for this particular prior, the posterior becomes almost zero. So we no longer belief that the coin is fair.

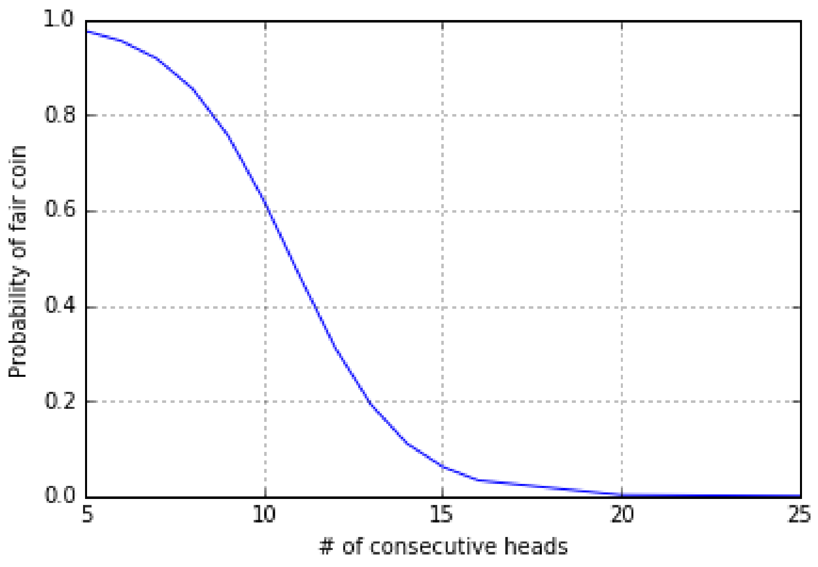


Figure 5.2: Learning curve

5.2.2 Learning a Continuous Parameter

Continuous parameters are not available in ProbLog. To use them, we must move to Distributional Clauses. However, we can simulate the use of a continuous parameter with arbitrary precision by implementing a term with has multiple values. This is done below (link). This is the classical example of estimating the bias of a coin based on the observation of a sequence of throws.

The issue with working this way is obviously that each additional value for the parameters increases the number of possible worlds, which might lead to computational problems. This is further explored in chapter 6. However, the obtained result agrees nicely with the one in [1].

Line 7 uses the `select_uniform` method, which is a nice way to avoid using long and tedious annotated disjunctions. It is part of the `lists` library [20].

```
1  % Learning Continuous Parameters
2
3  :-use_module(library(lists)).
4  weights([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]).
5  coin_weight(X) :-
6      weights(Weights),
7      select_uniform(weight, Weights, X, _).
8
9  PH::coin_flip(h,N); PT::coin_flip(t,N) :-
10     coin_weight(P),
11     PH is P, PT is 1.0-P.
12
```

```
observe([],0).
observe([Face|T], N) :-
    coin_flip(Face,N),
    N1 is N-1,
    observe(T, N1).

query(coin_weight(_)).

observed_sequence([h,h,t,h,h]).
evidence(observe(S, N)) :-
    observed_sequence(S),
    length(S,N).
```

Program 5.3: ProbLog: Learning a continuous parameter

The result of this program are represented in figure 5.3.

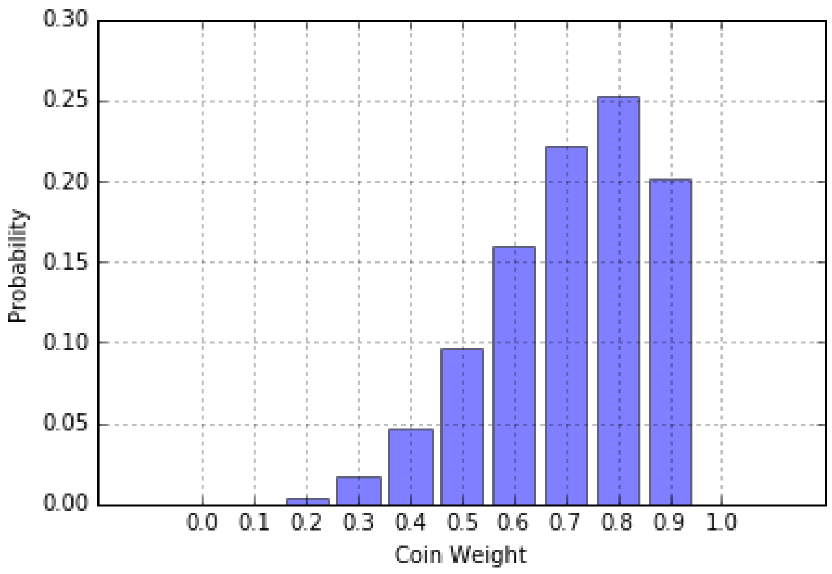


Figure 5.3: Learning a continuous parameter

We observe that most probability mass is put on the higher probabilities - a higher probability of throwing heads.

This program can be extended by using another prior, like a Beta distribution, with the method as described in the Polya - de Finetti problem (section 4.4.2).

5.2.3 Estimating Causal Power

In this example (link), we implement *elemental causal induction*: inferring how strongly a potential cause C impacts a certain event E. This event can be caused by both C and by background effects B. The Church version of this example is shown below:

```
(define samples
  (mh-query 10000 1
            (define cp (uniform 0 1)) ;;causal power of C to cause E.
            (define b (uniform 0 1))  ;;background probability of E.

            ;;the noisy causal relation to get E given C:
            (define (E-if-C C)
              (or (and C (flip cp))
                  (flip b)))

            ;;infer the causal power:
            cp

            ;;condition on some contingency evidence:
            (and (E-if-C true)
                 (E-if-C true)
                 (not (E-if-C false))
                 (E-if-C true))))
```

Program 5.4: Church: Estimating Causal Power

Several instances of the event e (both positive and negative) are introduced as evidence. Mind the use of the identifier in the term `e(ID, C)`, which is necessary to circumvent memoization. Without it, ProbLog would give an `InconsistentEvidence` error.

```
:-use_module(library(lists)).

values([0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]).
cp(X) :-
    values(Values),
    select_uniform(cp, Values, X, _).
b(X) :-
    values(Values),
    select_uniform(b, Values, X, _).

% e(ID, C).
P::e(_,t) :- % probability of E if C = true
    cp(P).
P::e(_,f) :- % probability of E if C = false
    b(P).

evidence(e(1,t), true).
evidence(e(2,t), true).
evidence(e(3,f), false).
evidence(e(4,t), true).
evidence(e(5,t), false).

query(cp(_)).
```

Program 5.5: ProbLog: Estimating Causal Power

The probability distribution over the causal power is shown in figure 5.4.

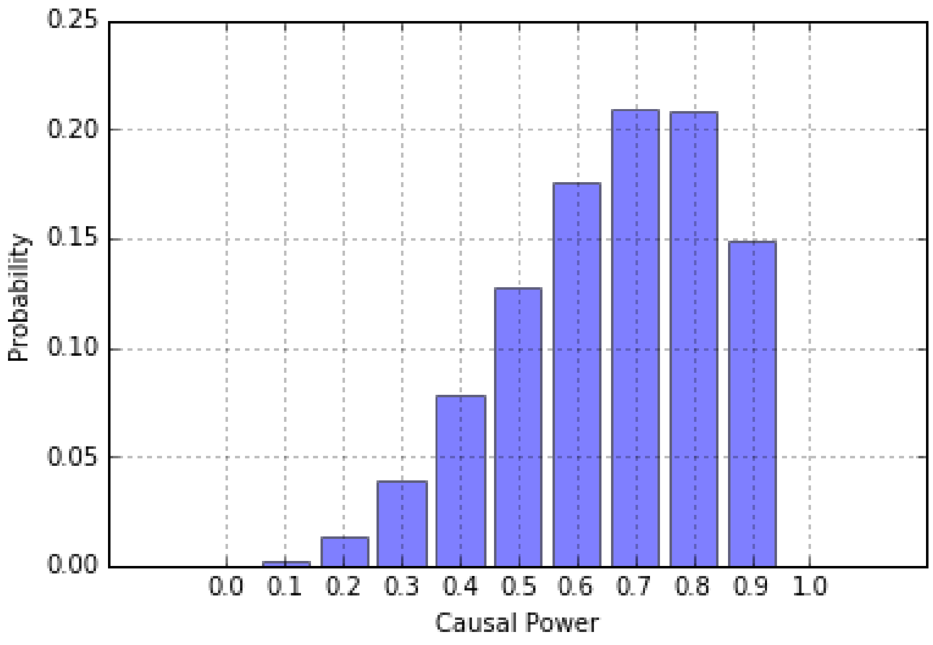


Figure 5.4: Inferring causal power

5.2.4 Rational Rules

In this more elaborate example (link), the authors explore how children learn rule-based concepts. Each rule is a disjunction of conjunctions. A disjunction is represented as a list of conjunctions, a conjunction is a list of attribute values that have to match. Attributes are true (1) or zero (0), or don't care (?). Rules of arbitrary length are induced by the `formula` term; the maximum length of a rule is set to 2 conjunctions. The number of attributes is limited (2 instead of 4 as in [1]), since otherwise, execution time becomes excessive. As such, this program is more proof-of-concept than a true translation of the program in [1]. Furthermore, negative evidence is not implemented, but this should be a straightforward extension.

This program also features a noisy-equal rule, allowing rules that do not agree with the example to have a probability larger than zero. Furthermore, it uses a specific python file `generate_unique_id` to generate a unique identifier for each flip, since there was no obvious way in this example to circumvent the implicit stochastic memoization[2].

```
% Represent a rules as a list of conjunctions: Rule = [Conj1, Conj2, ...]
% Represent a conjuction as a list of value - attribute pairs:
%   [Att1 = 0/1/?, Att2 = 0/1/?, ...] (-> ? = don't care)

:-use_module(library(lists)).
:-use_module('generate_unique_id.py').  % Help with noisy_equal:
                                         % generate unique identifier

% Params
tau(0.3).               % stopping probability of grammar
noise_param(X) :-       % noise probability
    X is exp(-5).
max_len_rule(2).        % Maximum # of Conj in a rule
numAttr(2).             % Number of attributes (4 in Rational Rules)

P::flip(P, _, _).
P::flip(P,_).
P::flip(P).

formula_flip(N) :-
    tau(Tau),
    flip(Tau, N, 0).
conj_flip(M, N) :-
    tau(Tau),
    flip(Tau, M, N).

formula([Conj|Formula], N) :- % N is max number of conjunctions
    N > 0,
    formula_flip(N),  % only recursion if myflip is true
    make_conj(Conj, N),
    N1 is N-1,
    formula(Formula, N1).

% 2 stop conditions:
%   1. formula_flip is false
formula([Conj], N) :-
    N > 0,
    \+formula_flip(N),
    make_conj(Conj, N).
```

[2]An alternative would have been the *gensym* library from SWI-Prolog, but this library is not available in ProbLog

```
%   2. N = 0
formula([], 0).

%% Make Predicates
make_conj(Conj, RuleID) :-
    numAttr(NumAttr),
    conj(Conj, NumAttr, RuleID).

conj([],0,_).
conj([Symbol|Conj], N, RuleID) :-
    N > 0,
    select_uniform(conj_id(N, RuleID), [0,1,?], Symbol, _), % uncomment to enable
    %select_uniform(conj_id(N, RuleID), [0,1], Symbol, _),  % uncomment to disable
    N1 is N-1,
    conj(Conj, N1, RuleID).

% eval_pos(Formula, PosExample) => evaluate a formula on a positive example
eval_pos([],_) :- false. % !! empty conj is always false
eval_pos([F1|Rest], Example) :- % a formula is true if first Conj is true:
    eval_conj(F1, Example)
    ;                           % or if a next conjuction in the formula is true:
    eval_pos(Rest, Example).

eval_conj([], []).
eval_conj([C1|FRest], [E1|ERest]) :-
    (C1 = ?;            % a conjuction is true if every predicate is true
    gen_unique_id(ID),
    noisy_equal(C1, E1, ID)),
    eval_conj(FRest, ERest).

P::noisy_equal(A, B, ID) :-
    writenl(ID),
    (A = B, P is 0.9999999;
    not(A = B), noise_param(P)).

observe :-
        max_len_rule(Max),
        formula(F, Max),
        eval_pos(F, [1,1]). % one oberved positive example

query(formula(F, Max)) :-
    max_len_rule(Max).
evidence(observe).
```

Program 5.6: ProbLog: Rational Rules

The result is a list of all rules of length 1 or 2, and a probability distribution over them corresponding to how well they agree with the positive example (attribute 1 = True, attribute 2 = True):

```
formula([[0, 0], [0, 0]],2): 3.2245232e-07
formula([[0, 0], [0, 1]],2): 4.8176443e-05
formula([[0, 0], [0, ?]],2): 4.8176448e-05
formula([[0, 0], [1, 0]],2): 4.7856163e-05
formula([[0, 0], [1, 1]],2): 0.0071024837
...
formula([[?, ?], [?, 1]],2): 0.0071024851
formula([[?, ?], [?, ?]],2): 0.0071024851
        formula([[?, ?]],2): 0.14915219
```

In a trivial manner, this example can be extended to also handle negative evidence.

5.3 Hierarchical Models

The goal of this chapter is to mimic the hierarchical structure of the human mind: the organization of knowledge in multiple layers of abstraction. A characteristic of these models from an implementation point of view is the propagation of probability distributions over other distribution, mostly under the form of Dirichlet distribution (which is well suited for this). This also implies that this cannot practically be done with ProbLog, a language that only allows discrete distributions. In theory, it would be possible to discretize the distributions, but this would lead to a combinatorial explosion of the number of possible worlds.

Instead, we will only use Distributional Clauses (DC) in this section. This will also be the case for sections 5.5 and 5.6.

This section is based on chapter 9 of [1].

5.3.1 Learning a Shared Prototype

This section concerns the learning of a shared prototype. This situation arises when we have multiple categories. With each category we associate a prototype - a typical representative of the category. In our case, this prototype will be a set of common parameters that are used as the arguments of a multinomial distribution. Initially, we consider each category as being independent. In the next example, we add one level in the hierarchy, such that every category inherits its prototype from a common general prototype.

To make this more concrete, the authors of [1] consider a number of bags of marbles. Each bag has its own prototype. This is done by drawing the distribution over the colored marbles from a bag from a Dirichlet distribution with fixed parameters. When each bag has its own fixed prototype, transfer of knowledge between categories is not possible.

The relation between a bag and its prototype in Church is defined as:

```
 (define bag->prototype
    (mem (lambda (bag) (dirichlet '(1 1 1 1 1)))))
```

In DC, this becomes:

```
bag_prototype(Bag) ~ dirichlet([1,1,1,1,1]).
```

In Church, a sample is drawn according to a multinomial distribution:

```
(define (draw-marble bag) (multinomial colors (bag->prototype bag)))
```

In DC, this becomes:

```
marble(Bag, N) ~ finite([Pblack:black, Pblue:blue, Pgreen:green,
                         Porange:orange, Pred:red]) :=
    bag_prototype(Bag) ~= [Pblack, Pblue, Pgreen, Porange, Pred].
```

The binary predicate `~=` associates a value with the distribution `bag_prototype`. Since it has an argument `Bag`, the prototype will be different for each bag. The inclusion of `N` in `marble(Bag, N)` makes that at each position in the sampling sequence, a new sample is drawn.

The evidence is a list of marbles drawn:

```
evidence(   [marble(bag1, 1) ~= blue, marble(bag1, 2) ~= blue,
             marble(bag1, 3) ~= black, marble(bag1, 4) ~= blue,
             ...]
```

Since we infer a multi-valued distribution (the domain are the possible colors of marbles), the inference is done with the `eval_query_distribution_eval` clause [3]:

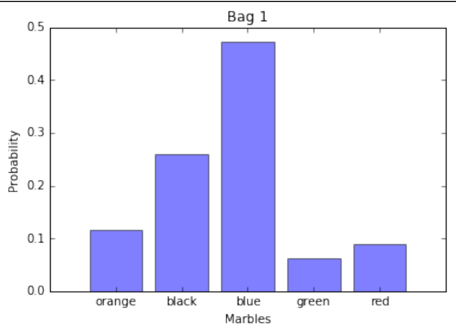
```
eval_query_distribution_eval(X, Evidence, [],
                             marble(Bag, Next) ~= X, N, LP, _, _).
```

Here is `X` the value we want to store, `Next` the next identifier not included in the evidence, `N` the number of requested samples, and `LP` a list of `probability:value` pairs. The number of samples is requested, since in importance sampling, it might happen that samples are returned with weight zero, who have no impact on the final result. If the evidence becomes more constricting, the number of samples with zero weight increases. More on this in chapter 6.

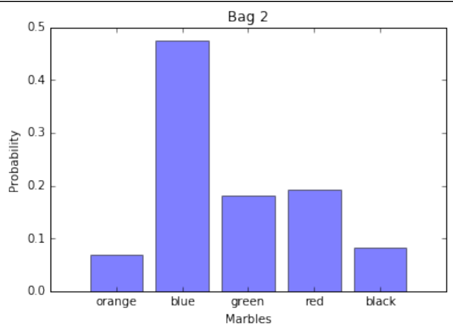
The entire program can be found on github. The results of this program are summarized in figure 5.5.

For all bags for which we have evidence, we do indeed find that the posterior adapts to the evidence: all bags are majority blue, with another color as secondary. On the other hand, the bag for which we have no observations (bag N), still has a uniform distribution, as the prototype suggests.

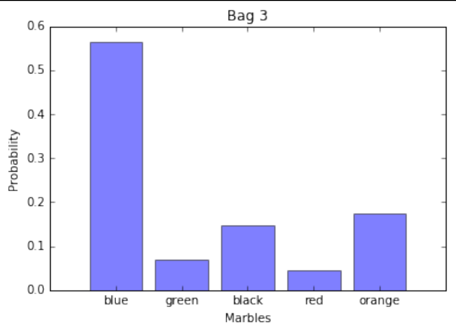
[3]The eval_query_distribution clause should do the same thing, but is much less stable



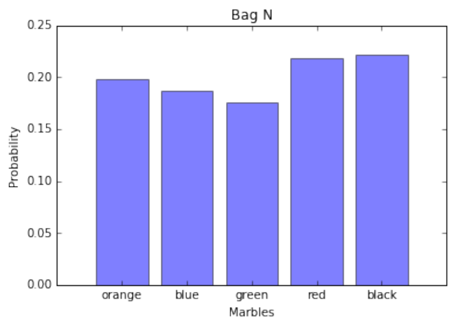
Distribution of Bag 1



Distribution of Bag 2



Distribution of Bag 3



Distribution of Bag N

Figure 5.5: Bags without global prototype

5.3.2 The Blessing of Abstraction

The previous example is not what we want: we want what we learned about one category (bag) to have an impact on other categories (bags). The answer lies in adding another level of abstraction, namely a global prototype, specifying a prior on the specific mixtures of each bag.

```
prototype_temp ~ dirichlet([1,1,1,1,1]).
global_prototype ~ val(Proto) :=
    prototype_temp ~= Ps,
    map_multiply(5, Ps, Proto).
```

This global prototype is in itself a scaled up version of parameters drawn from a Dirichlet distribution. The `map_multiply` clause multiply each element of `Ps` by 5, and stores the result in `Proto`. Now, instead of deriving the distribution parameters of each bag from a Dirichlet distribution with fixed arguments, we use the parameters as generated by the global prototype:

```
bag_prototype(Bag) ~ dirichlet(Prototype) :=
    global_prototype ~= Prototype.
```

The code in Church looks very similar:

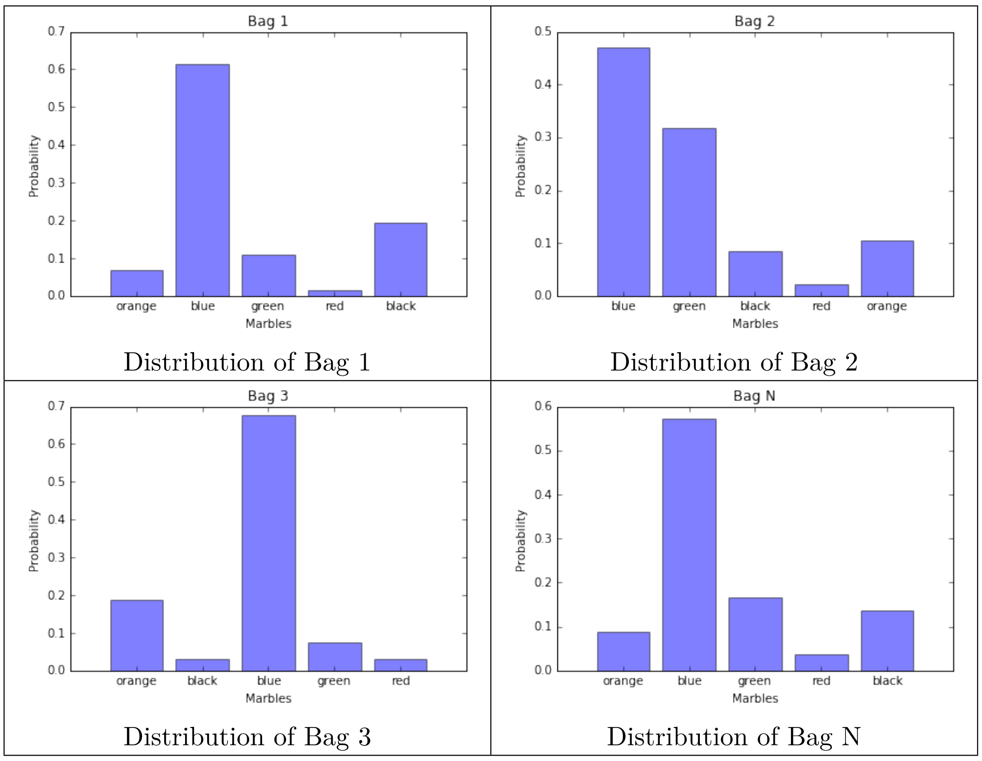
```
(define prototype (map (lambda (x) (* 5 x)) (dirichlet '(1 1 1 1 1))))

(define bag->prototype
    (mem (lambda (bag) (dirichlet prototype))))
```

This shows that Church and DC are relatively similar in syntax structure.

Inference is done in the same way as in the previous section. Figure 5.6 summarizes the results of this program. The evidence for bag 2 was limited to 2 instead of 6 observations. In accordance with the observations from [1], two points can be made:

1. Even though we only have two observations for bag 2, the blue balls are more frequent than the green ones, because the global prototype prefers similar bags

2. Even with no observations, bag N still has a similar structure as the other bags, namely a preference for blue and a uniform distribution over the other colors (since the three other bags each have another secondary color). This phenomena is known as *Transfer Learning*.



Distribution of Bag 1 Distribution of Bag 2

Distribution of Bag 3 Distribution of Bag N

Figure 5.6: Bags with global prototype

5.3.3 The Shape Bias

The *shape bias* is an application of learning an overhypothesis; it describes the inductive bias in young children where they generalize a label for an object based on a common shape, rather than for example the same color or texture.

The program we'll use to study this shape bias uses the compound Dirichlet-mutinomial model that has been used in the previous models of this section. Bags of marbles become categories (`cat1`, `cat2`, `cat3`, ...), the colors become features (`shape`, `color`, `texture`, `size`), and each features has its own list of allowed values - in the original program each feature has 10 distinct values, designated by an integer from 1 to 10.

The program in listing 5.7 shows the translation of the entire Church model into ProbLog. It is very similar to the Church code.

```
% Parameters
shapes([0,1,2,3,4,5,6,7,8,9,10]).
colors([0,1,2,3,4,5,6,7,8,9,10]).
textures([0,1,2,3,4,5,6,7,8,9,10]).
sizes([0,1,2,3,4,5,6,7,8,9,10]).

phi_shapes ~ dirichlet([1,1,1,1,1,1,1,1,1,1,1]).
phi_colors ~ dirichlet([1,1,1,1,1,1,1,1,1,1,1]).
phi_textures ~ dirichlet([1,1,1,1,1,1,1,1,1,1,1]).
phi_sizes ~ dirichlet([1,1,1,1,1,1,1,1,1,1,1]).

% regularity parameters: how strongly we expect the global prototype
% to project (ie. determine the local prototypes):
alpha_shapes   ~ gamma(1, 1). % exponential distribution as
alpha_colors   ~ gamma(1, 1). % a special case of gamma distribution
alpha_textures ~ gamma(1, 1). % gamma(1, beta) = exp(1/beta)
alpha_sizes    ~ gamma(1, 1).

prototype_shapes ~ val(Proto) :=
    phi_shapes ~= Ps,
    alpha_shapes ~= A,
    map_multiply(A, Ps, Proto).

prototype_colors ~ val(Proto) :=
    phi_colors ~= Ps,
    alpha_colors ~= A,
    map_multiply(A, Ps, Proto).

prototype_textures ~ val(Proto) :=
    phi_textures ~= Ps,
    alpha_textures ~= A,
    map_multiply(A, Ps, Proto).

prototype_sizes ~ val(Proto) :=
    phi_sizes ~= Ps,
    alpha_sizes ~= A,
    map_multiply(A, Ps, Proto).

cat_shape(Cat) ~ dirichlet(Ps) :=
        prototype_shapes ~= Ps.
cat_color(Cat) ~ dirichlet(Ps) :=
        prototype_colors ~= Ps.
cat_texture(Cat) ~ dirichlet(Ps) :=
        prototype_textures ~= Ps.
cat_size(Cat) ~ dirichlet(Ps) :=
        prototype_sizes ~= Ps.
category_prototype(Category) ~ val(proto(Shape, Color, Texture, Size)) :=
        cat_shape(Category) ~= Shape,
        cat_color(Category) ~= Color,
        cat_texture(Category) ~= Texture,
        cat_size(Category) ~= Size.

draw_object(Category, N) ~ val(object(Shape, Color, Texture, Size)) :=
        draw_shape(Category, N) ~= Shape,
        draw_color(Category, N) ~= Color,
        draw_texture(Category, N) ~= Texture,
        draw_size(Category, N) ~= Size.

draw_shape(Cat, N) ~ finite([P0:0, P1:1, P2:2, P3:3, P4:4, P5:5,
                            P6:6, P7:7, P8:8, P9:9, P10:10]) :=
        category_prototype(Cat) ~= proto(Shape,_,_,_),
        Shape = [P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10].

draw_color(Cat, N) ~ finite([P0:0, P1:1, P2:2, P3:3, P4:4, P5:5,
                            P6:6, P7:7, P8:8, P9:9, P10:10]) :=
        category_prototype(Cat) ~= proto(_,Color,_,_),
        Color = [P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10].

draw_texture(Cat, N) ~ finite([P0:0, P1:1, P2:2, P3:3, P4:4, P5:5,
                              P6:6, P7:7, P8:8, P9:9, P10:10]) :=
        category_prototype(Cat) ~= proto(_,_,Texture,_),
        Texture = [P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10].

draw_size(Cat, N) ~ finite([P0:0, P1:1, P2:2, P3:3, P4:4, P5:5,
                            P6:6, P7:7, P8:8, P9:9, P10:10]) :=
        category_prototype(Cat) ~= proto(_,_,_,Size),
77         Size = [P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10].
78
79 evidence([draw_object(cat1, 1) ~= object(1,1,1,1),
80           draw_object(cat1, 2) ~= object(1,2,2,2),
81           draw_object(cat2, 1) ~= object(2,3,3,1),
82           draw_object(cat2, 2) ~= object(2,4,4,2),
83           draw_object(cat3, 1) ~= object(3,5,5,1),
84           draw_object(cat3, 2) ~= object(3,6,6,2),
85           draw_object(cat4, 1) ~= object(4,7,7,1),
86           draw_object(cat4, 2) ~= object(4,8,8,2),
87           draw_object(cat5, 1) ~= object(5,9,9,1)]).
88
89 test(N) :-
90     init,
91     evidence(Evidence),
92     eval_query_distribution_eval(X, Evidence, [],
93                                  draw_object(cat5, 2) ~= object(X,_,_,_), N, LP, _, _),
94     writeln(LP).
95
96 :- initialization(test(50000)).
```

Program 5.7: DC: The Shape Bias

Every hyperparameter ϕ is drawn from a Dirichlet distribution (lines 7-10). Every kind of feature has its own hyperparameter. These hyperparameters will be used as the parameters of the multinomial distributions from which the category prototypes are drawn.

Besides the Dirichlet parameters, each kind of feature also has a regularity parameter, which determines how influential the global prototype of that feature will be. These α parameters are determined in lines 13 to 17 as the result of draws from an exponential distribution, which is implicitly defined as a Gamma distribution with first parameter equal to 1.

Next, the different (global) prototypes per feature are defined. These are random variables determined by the point-wise product of their respective ϕ and α hyperparameters (lines 19-37).

Following that, the prototypes per category are sampled. For each feature prototype, a category-feature prototype is created (lines 39-46), and all these category-feature prototypes are put together to generate a prototype per category (`category_prototype` - lines 47-51).

Once each category has its prototype, namely the different multinomial parameters per feature, sample objects can be drawn. This is achieved with the lines 53-77. Mind the addition of a identifier for each sample, to avoid that, due to memoization, we would each time draw the same object for each category.

The resulting object is data structure `object/4`, where the arguments are the values of the features (`shape`, `color`, `texture`, `size`).

Finally, the evidence is created, namely a sequence of objects observed for each category. Observe that the category is determined by the first feature. This the shape of the object, hence the shape bias.

We then sample from category `cat5`, and expect that the distribution over the shapes heavily tilts towards 5, since the observed object of category `cat5` has shape 5.

However, in this configuration, it is practically impossible to obtain enough samples: the inference method as used by Distributional Clauses is simply not efficient enough. In order to obtain results, we must simplify the model. This can be done in a few ways: reduce the number of features, reduce the domain size for the features, reduce the number of observed objects, or a combination of all three.

5.3.4 X-Bar Theory

Children are able to acquire their language capacities very quickly and from noisy and sparse data. One explanation for this phenomenon is that there are a number of higher-order constraints, each with a limited set of possible values. As a consequence, the complexity of the learning process is significantly reduced.

One concrete theory is the X-bar phrase structure [14], which provides a hierarchical model for phrase structure. Phrases are formed according to one basic template:

$$XP \rightarrow SpecX' \tag{5.1}$$

$$X' \rightarrow XComp \tag{5.2}$$

X is a lexical category, like nouns, verbs, ... According to X-bar theory, all phrases have the same basic structure:

- They have a *head*, which is a word of category X,
- They have a *specifier* (*Spec*), and a *complement* (*Comp*), whereby the latter more closely associated is with the head.

Languages differ in the order in which heads appear in sentences with respect to their complements. In English, the direct object of a verb is placed right from the head (the verb, in this case). However, adjectives are placed to left of their head, the noun.

Consistency of languages with respect to head order, can be of great use to the learner. After observing only a few instances, she can already learn the dominant head direction, and move on from there. However, since there are always exceptions, the learned rules must be probabilistic. In [1], a simplified model of X-bar structure learning is given (link). The DC version is given below (link).

```
%% Choose a data set:
data([[d, n]]).
%data([[d, n], [d,n]]).
```

```
%data([[d, n], [t,v], [v, adv]]).

categories([d,n,t,v,a,adv]) := true.

% head_comp(Head, Comp)
head_comp(d,n) := true.
head_comp(t,v) := true.
head_comp(n,a) := true.
head_comp(v,adv) := true.
head_comp(a,none) := true.
head_comp(adv,none) := true.

language_direction ~ beta(1, 1).

flip(P, ID) ~ finite([PL:left, PR:right]) :=
    PL is P, PR is 1.0-P.

temp(Head, ID) ~ dirichlet([LangDir, LangDir1]) := % for use in head_phrase
    language_direction ~= LangDir,
    LangDir1 is 1.0-LangDir.

head_phrase(Head, ID) ~ val(P) :=
    temp(Head, ID) ~= [P, _].

generate_phrase(Head) ~ val([Head]) := % no complement
    head_comp(Head, none).

generate_phrase(Head) ~ val([Head, Comp]) := % Comp right of Head
    \+head_comp(Head, none),
    gensym(g, Unique),
    head_phrase(Head, Unique) ~= Dir,
    flip(Dir, Unique) ~= Direction,
    flip(Dir, Unique) ~=  right,
    head_comp(Head, Comp).

generate_phrase(Head) ~ val([Comp, Head]) := % Comp left of Head
    \+head_comp(Head, none),
    gensym(g, Unique),
    head_phrase(Head, Unique) ~= Dir,
    flip(Dir, Unique) ~= Direction,
    head_comp(Head, Comp).

uniform_draw(ID) ~ uniform(Categories) :=
    categories(Categories).

```

```
observe_phrase(ID) ~ val(Phrase) :=
    uniform_draw(ID) ~= Category,
    generate_phrase(Category) ~= Phrase.
```

Program 5.8: DC: X-bar Theory

The question we ask is: based on the observations as defined in the predicate `data`, how likely are we to put the adjective `a` before the noun `n`, and vice versa.

The translation from Church to DC is pretty straightforward:

- A *case* statement in Church is translated to a sequence of `head_comp(Head, Comp)` terms,
- Stochastic functions like `language-direction` are directly translated by means of the corresponding stochastic primitive in Distributional Clauses.
- The `generate_phrase` clauses are a bit more elaborate, but have basically the same structure as in Church. The switching is done inside the body of the clause, as in standard Prolog.
- We made use of the `gensym` predicate to create unique atoms. This guarantees that flips and observations are recreated every time they are called upon.

The results obtained with the DC version correspond very closely to the Church implementation. This is also the case when we use additional evidence (lines 3 and 4).

5.4 Occam's Razor

Occam's Razor is the general principle, named after William of Ockham, that all else being equal, the simplest explanation should be preferred. The mind works in a similar manner: while for more complex models it is easier to explain phenomena, we do not inevitable prefer them.

There was no need to work with DC for this chapter. Thus, all programs were implemented in ProbLog.

This section is based on chapter 10 of [1].

5.4.1 The Size Principle

The *size principle* is a simple case of Bayes Occam's razor. It states that for hypotheses that generate data uniformly, the hypothesis that (1) is consistent with the data, and (2) has the smallest extension (domain), is the most probable.

The Church code of this example is given below:

```
(define samples
   (mh-query
    100 100

    (define (hypothesis->set  hyp)
      (if (equal? hyp  'Big) '(a b c d e f) '(a b c))

    (define hypothesis (if (flip) 'Big 'Small))
    (define (observe N)
      (repeat N (lambda () (uniform-draw (hypothesis->set hypothesis)))))

    hypothesis

    (equal? (observe 1) '(a))))
```

Program 5.9: Church: The Size Principle

This principle is illustrated in listing 5.9, where the `Big` hypothesis has 6 elements, while the `Small` one only has 3 possibilities. When the data is consistent with both hypotheses, the smaller one is preferred (see figure 5.7).

This is translated in ProbLog in listing 5.10. The `repeat` operation of Church is ported to a recursive `observe/2` predicate. Evidence and query are almost a direct copy.

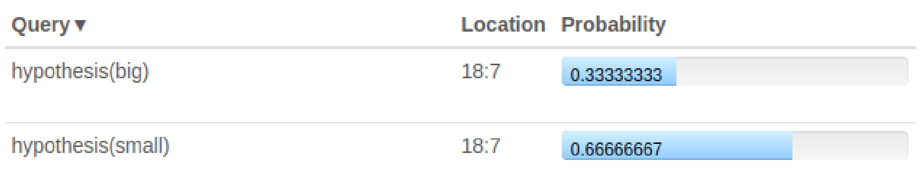
```
hypothesis_set(big, [a,b,c,d,e,f]).
hypothesis_set(small, [a,b,c]).

0.5::hypothesis(big); 0.5::hypothesis(small).

observe(0,[]).
observe(N, [Observation|Others]) :-
    hypothesis(Hypo),
    hypothesis_set(Hypo, Set),
    select_uniform(N, Set, Observation, _),
    N1 is N-1,
    observe(N1, Others).

query(hypothesis(_)).
evidence(observe(1, [a])).
```

Program 5.10: ProbLog: The Size Principle

Figure 5.7 shows the result of program 5.10. When we observe only one `a`, the small hypothesis is preferred over the big one.

5.4.2 The Rectangle Game

In this game, the data consists of a set of randomly sampled points from inside an unknown rectangle. The goal is to infer the rectangle. The Church code can be found here.



Query ▾	Location	Probability
hypothesis(big)	18:7	0.33333333
hypothesis(small)	18:7	0.66666667

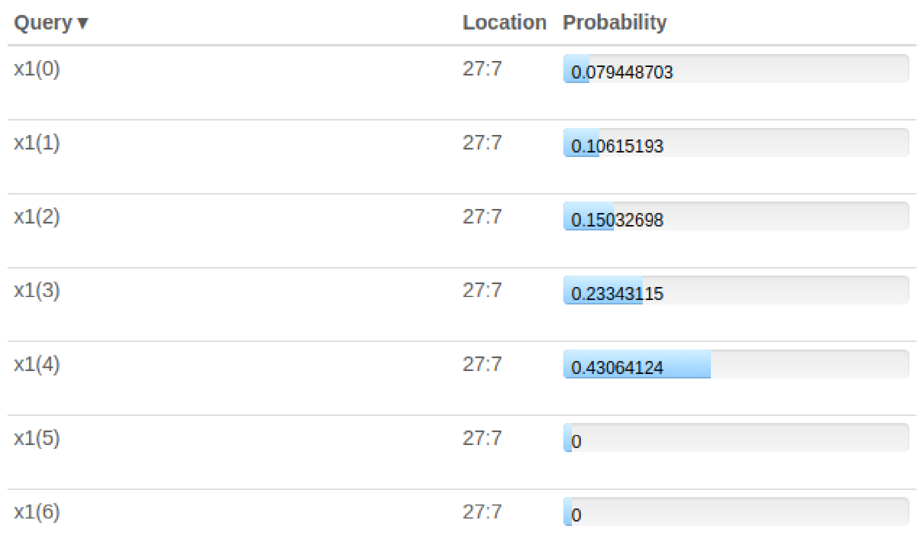
Figure 5.7: The Size Principle

We observe two points in the plane ((4,7) and (5,4)), and must infer the 4 corners of the rectangle that contains them. The ProbLog code is show in 5.11.

Mind the use of the combined identifier of the `select_uniform` terms, to guarantee uniqueness. The distribution of the x_1-coordinate is shown in figure 5.8 - this is only a partial view of the entire solution.

```
max(10).
set(Set) :-
    max(Max), findall(X, between(0, Max, X), Set).
set(Min,Max,Set) :-
     findall(X, between(Min, Max, X), Set).

x1(X) :- set(Set), select_uniform(x1, Set, X, _).
x2(X) :- set(Set), select_uniform(x2, Set, X, _).
y1(Y) :- set(Set), select_uniform(y1, Set, Y, _).
y2(Y) :- set(Set), select_uniform(y2, Set, Y, _).

concept(X, Y, N) :-
    x1(X1), x2(X2),
    X2 > X1,
    set(X1, X2, SetX), select_uniform([x,N], SetX, X, _),
    y1(Y1), y2(Y2),
    Y2 > Y1,
    set(Y1, Y2, SetY), select_uniform([y,N], SetY, Y, _).

evidence(concept(4, 7, 1)).
evidence(concept(5, 4, 2)).

query(x1(_)).
query(x2(_)).
query(y1(_)).
query(y2(_)).
```

Program 5.11: ProbLog: The Rectangle Gamse



Query ▾	Location	Probability
x1(0)	27:7	0.079448703
x1(1)	27:7	0.10615193
x1(2)	27:7	0.15032698
x1(3)	27:7	0.23343115
x1(4)	27:7	0.43064124
x1(5)	27:7	0
x1(6)	27:7	0

Figure 5.8: Result of the Rectangle Game (partial)

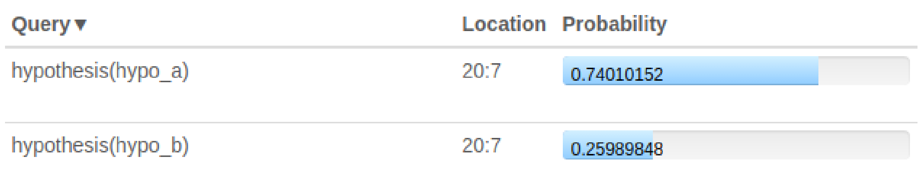
5.4.3 Bayes Occam's Razor

The concept of Bayes Occam's razor extends the size principle: models that are simpler have higher posterior probabilities when they fit the data well, compared to more complex models. In the program (link) below, `hypo_a` is simpler than `hypo_b`, since it put most of its probability mass on a smaller section of its domain. This part of the domain also happens to be the space where most of the observations came from. As a consequence, `hypo_a` is much likelier than `hypo_b`, although both hypotheses can account for the data - see figure 5.9.

```
hypo_param(hypo_a, [0.375,0.375,0.125,0.125]).
hypo_param(hypo_b, [0.25,0.25,0.25,0.25]).

0.5::hypothesis(hypo_a); 0.5::hypothesis(hypo_b).

observe(0,[]).
observe(N, [Letter|Others]) :-
    pick_letter(N, Letter),
    N1 is N-1,
    observe(N1, Others).

Pa::pick_letter(N, a); Pb::pick_letter(N, b); Pc::pick_letter(N, c);
Pd::pick_letter(N, d) :-
    between(1,100,N),
    hypothesis(Hypo),
    hypo_param(Hypo, [Pa,Pb,Pc,Pd]).

```

```
evidence(observe(8, [a,b,a,b,c,d,b,b])).
query(hypothesis(_)).
```

Program 5.12: ProbLog: Bayes Occam's Razor



Query ▾	Location	Probability
hypothesis(hypo_a)	20:7	0.74010152
hypothesis(hypo_b)	20:7	0.25989848

Figure 5.9: Bayes Occam's Razor

5.4.4 Model Selection with the Bayesian Occam's Razor

This section uses the concept of the Bayesian Occam's razor to determine a probability distribution over models. A first example (link) concerns the determination whether a coin is fair, or not, based on a sequence of observations. This example is an extension of example 5.2.1.

Programs 5.13 and 5.14 show the Church and ProbLog implementations of this principle, respectively.

- Both programs make use of a `fair_prior`
- They create both a coin of which the weight depends on the result of a flip with weight `fair_prior`
- subsequently, a list of throws is created and compared with the observed data.

Line 9 of the Church code is a little involved: it basically creates a list of conditions saying that for each element of the `observed-data`, the coin is flipped and its result (`#t` or `#f`) must correspond with the result of the expression `(equal? datum 'h)`, where `datum` is an element of `observed-data`. The difficulty comes from the fact that `(flip p)` returns true or false, and not head or tails.

```
1  (query
2       (define fair-coin? (flip fair-prior))
3       (define coin-weight (if fair-coin?
4                               0.5
5                               (beta (first pseudo-counts) (second pseudo-counts)))
6
7       (define make-coin (lambda (weight) (lambda () (if (flip weight) 'h 't))))
8       (list (if fair-coin? 'fair 'unfair) coin-weight)
9       (map (lambda (datum) (condition (equal? (flip coin-weight)
10                                            (equal? datum 'h)))) observed-data))
```

Program 5.13: Church: Bayesian Model Selection

```
P::make_coin(Coin, P).

fair_prior(0.999).
P::fair_coin :- fair_prior(P).

coin_weight(0.5) :- fair_coin.
coin_weight(P) :-   \+ fair_coin,
    weights(W), select_uniform(cw, W, P, _).

coin(C) :-
    coin_weight(P),
    make_coin(C,P).

sample(C,h) :-   coin(C).
sample(C,t) :- \+coin(C).

observe(0, []).
observe(N, [Sample|Rest]) :-
    sample(N, Sample),
    N1 is N-1,
    observe(N1, Rest).

% fair coin, probability of H = 0.5
observed_data([h,h,t,h,t,h,h,h,t,h]).
evidence(observe(N,S)) :-
    observed_data(S),
    length(S,N).

query(fair_coin).
query(coin_weight(_)).
```

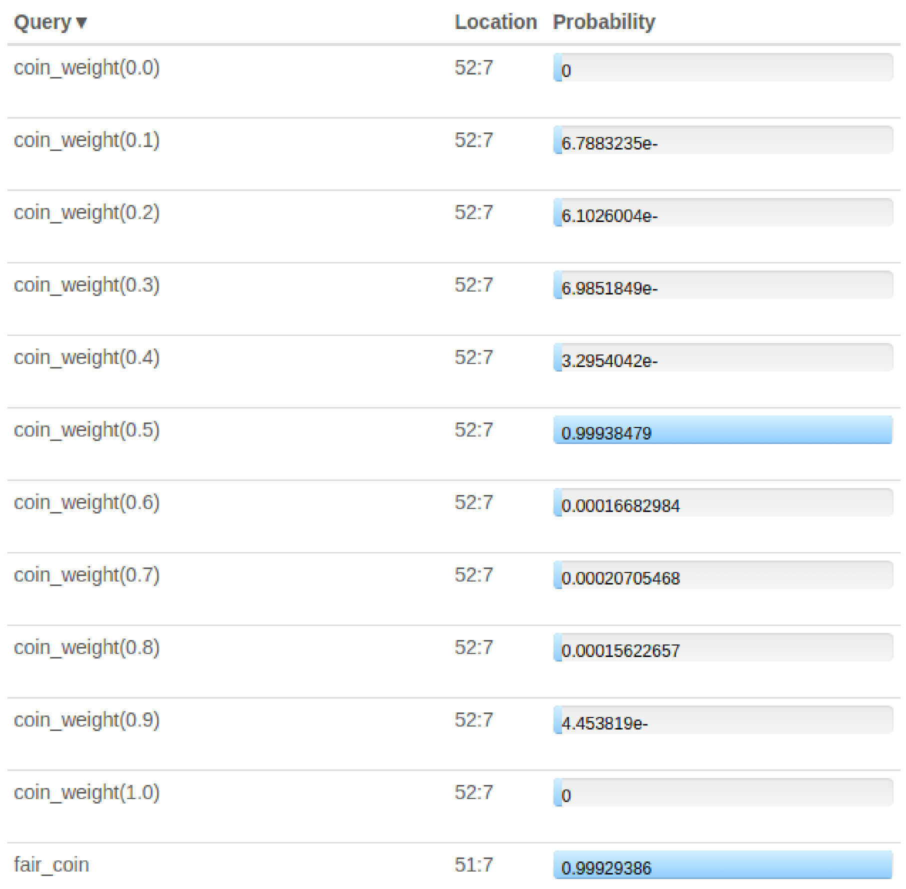
Program 5.14: ProbLog: Bayesian Model Selection

The result of the two queries of program 5.14 is shown in figure 5.10.

5.4.5 Scene Inference

Static Model

This is an example (link) that's particular well suited for a symbolic language as ProbLog. The goal is to determine whether we observe 1 or 2 objects when we see an image of two connected patches. The program generates an image based an object that has 4 parameters: the x- and y-coordinates of its initial position, its vertical



Query ▾	Location	Probability
coin_weight(0.0)	52:7	0
coin_weight(0.1)	52:7	6.7883235e-
coin_weight(0.2)	52:7	6.1026004e-
coin_weight(0.3)	52:7	6.9851849e-
coin_weight(0.4)	52:7	3.2954042e-
coin_weight(0.5)	52:7	0.99938479
coin_weight(0.6)	52:7	0.00016682984
coin_weight(0.7)	52:7	0.00020705468
coin_weight(0.8)	52:7	0.00015622657
coin_weight(0.9)	52:7	4.453819e-
coin_weight(1.0)	52:7	0
fair_coin	51:7	0.99929386

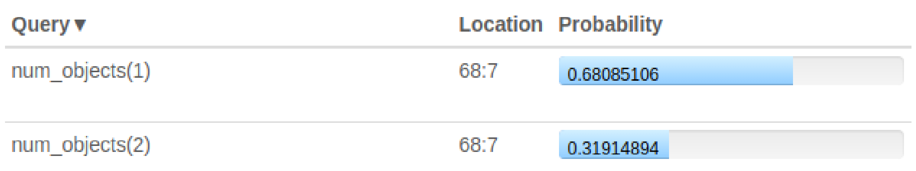
Figure 5.10: Fair or unfair coin?

size and its color. Inference allows then to switch from the observed scene, with it's individual pixels, to the mental image of the object or objects that form(s) that image. The resulting inference, namely the distribution over the number of observed objects, is shown in figure 5.11. It is about two times as likely that we observe 1 object instead of 2.

```
% Scene Inference

:-use_module(library(lists)).

% object(ID, x_location, y_location, vertical_size, color).
% image = [[_,_,_,_], [_,_,_,_]].

make_object(ID, object(ID, XLoc, YLoc, VertSize, Color)) :-
    select_uniform(xloc(ID), [0,1,2,3], XLoc, _),
    select_uniform(yloc(ID), [0,1], YLoc, _),
    select_uniform(vertsize(ID), [1,2], VertSize, _),
    select_uniform(color(ID), [1,2], Color, _).

%object_appearance(Object, Image).
object_appearance(Object, [[A,B,C,D], [E,F,G,H]]) :-
    set_pixel(Object,0,0,A), set_pixel(Object,1,0,B),
    set_pixel(Object,2,0,C), set_pixel(Object,3,0,D),
    set_pixel(Object,0,1,E), set_pixel(Object,1,1,F),
    set_pixel(Object,2,1,G), set_pixel(Object,3,1,H).

%set_pixel(Object, XCoord, YCoord, Pixel).
set_pixel(object(_, XLoc, _, _, _), XCoord, _, 0) :-
    XCoord < XLoc.
set_pixel(object(_, XLoc, _, _, _), XCoord, _, 0) :-
    XLoc1 is XLoc + 1,
    XLoc1 =< XCoord.
set_pixel(object(_, _, YLoc, _, _), _, YCoord, 0) :- YCoord < YLoc.
set_pixel(object(_, _, YLoc, VertSize, _), _, YCoord, 0) :-
    YLoc1 is YLoc + VertSize,
    YLoc1 =< YCoord.
set_pixel(object(_, XLoc, YLoc, VertSize, Color), XCoord, YCoord, Color) :-
    XLoc = XCoord,
    YLoc =< YCoord,
    YLoc1 is YLoc + VertSize,  YLoc1 > YCoord.

% layerr(ObjectImage, BackgroundImage, CompoundImage)
layer([ORow1, ORow2], [IRow1, IRow2], [CRow1, CRow2]) :-
    layer_row(ORow1, IRow1, CRow1),
    layer_row(ORow2, IRow2, CRow2).
layer_row([],[],[]).
layer_row([0|ORow], [IPix|IRow], [IPix|CRow]) :-
    layer_row(ORow, IRow, CRow).
layer_row([C|ORow], [IPix|IRow], [C|CRow]) :-
    C>0,
    layer_row(ORow, IRow, CRow).

observed_image([[0,1,0,0],
                [0,1,0,0]]).

0.5::num_objects(1); 0.5::num_objects(2).

image(Image) :-
    num_objects(1),
    make_object(1,Object1),
    object_appearance(Object1, Image).
```

```
image(Image) :-
    num_objects(2),
    make_object(1, Object1),
    make_object(2, Object2),
    object_appearance(Object1, Image1),
    object_appearance(Object2, Image2),
    layer(Image1, Image2, Image).

evidence(image(Image)) :-
    observed_image(Image).

query(num_objects(_)).
```

Program 5.15: ProbLog: Scene Inference - Static Model



Query ▾	Location	Probability
num_objects(1)	68:7	0.68085106
num_objects(2)	68:7	0.31914894

Figure 5.11: Static Scene Inference

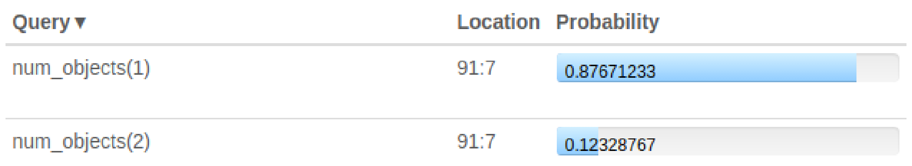
Dynamic Model

This example (link) adds an additional layer to the previous one: this time, we observe two consecutive images, and see one or more moving objects. The movement of an object is implemented with the `move(Current, Next)` method. It is clear that if we see 2 patches moving in unison, it is very likely that these patches are part of the same object. This increased confidence is reflected in the resulting inference, as shown in figure 5.12.

```
...
% move(Current, Next)
move(object(ID, XLoc, YLoc, VertSize, Color),
     object(ID, NewXLoc, YLoc, VertSize, Color)) :-
    select_weighted(step(ID), [0.3,0.4,0.3], [-1,0,1], Step, _),
    NewXLoc is XLoc + Step.

observed_image1([[0,1,0,0],
                 [0,1,0,0]]).
observed_image2([[0,0,1,0],
                 [0,0,1,0]]).

```

```
...
image2(Image) :-
    num_objects(1),
    make_object(1, Object1),
    move(Object1, NextObject1),
    object_appearance(NextObject1, Image).
image2(Image) :-
    num_objects(2),
    make_object(1, Object1), make_object(2, Object2),
    move(Object1, NextObject1), move(Object2, NextObject2),
    object_appearance(NextObject1, Image1),
    object_appearance(NextObject2, Image2),
    layer(Image1, Image2, Image).


evidence(image1(Image)) :-
    observed_image1(Image).
evidence(image2(Image)) :-
    observed_image2(Image).

query(num_objects(_)).
```

Program 5.16: ProbLog: Scene Inference - Dynamic Model (snippet)



Query ▾	Location	Probability
num_objects(1)	91:7	0.87671233
num_objects(2)	91:7	0.12328767

Figure 5.12: Dynamic Scene Inference

5.5 Mixture Models

In the chapter on hierarchical models (section 5.3), models with a hierarchical structure were introduced: based on a general prototype, a specific prototype was derived for each category. This enabled, amongst other things, faster learning, based on less examples.

However, in chapter 5.3, we always assumed that we knew from which category an observation came. This section treats models for which we don't know a priori how to divide up the observations.

We do still, however, assume that we know how many categories there are. If we drop this assumption, we move into the realm of *non-parametric models* (section 5.6).

This section is based on chapter 11 of [1].

5.5.1 Learning Categories

This example (link) is a modification from the bags-of-marbles example of section 5.3, but now we assume that the bag each marble is drawn from is unobserved. We assume that the bags are distributed uniformly.

```
phi ~ dirichlet([1,1,1]).
alpha ~ val(0.1).
prototype ~ val(Ps) :=
        phi ~= Phi,alpha ~= Alpha,
        map_multiply(Alpha, Phi, Ps).

bag_prototype(Bag) ~ dirichlet(Prototype) :=
        prototype ~= Prototype.

obs_bag(Obs) ~ finite([P:bag1,P:bag2,P:bag3]) :=
        P is 1/3.

draw_marble(Obs) ~ finite([Pb:blue, Pg:green, Pr:red]) :=
        obs_bag(Obs) ~= Bag,
        bag_prototype(Bag) ~= [Pb,Pg,Pr].

evidence([draw_marble(obs1) ~= red,  draw_marble(obs2) ~= red,
          draw_marble(obs3) ~= blue, draw_marble(obs4) ~= blue,
          draw_marble(obs5) ~= red, draw_marble(obs6) ~= blue]).

test(N) :-
        init,
        evidence(Evidence),
        query(Evidence,[], (obs_bag(obs1) ~= Bag1,
                            obs_bag(obs2) ~= Bag2, Bag1 = Bag2), N, P1),
        write('obs1 and obs2 from same bag: '), writeln(P1),
        query(Evidence,[], (obs_bag(obs1) ~= Bag1,
                            obs_bag(obs3) ~= Bag2, Bag1 = Bag2), N, P2),
        write('obs1 and obs3 from same bag: '), writeln(P2).

:- initialization(test(10000)).
```

Program 5.17: DC: Learning Categories

In program 5.17, lines 10 and 11 create the Distributional Clause from which the bag will be sampled. We now ask not about the distributions of bags, but whether observations come from the same bag (lines 24/25 and 27/28).

When we perform this inference, we get the following result (this is sampling based, so it can vary with different runs):

- Obs1 and Obs2 come from the same bag: P = 57,04 %
- Obs1 and Obs3 come from the same bag: P = 2,15 %

This corresponds with the values obtained in [1].

We can go beyond this, and no longer assume that each bag is equally likely. As such we also can learn the bag probabilities (the *mixture distribution*). This means that line

```
obs_bag(Obs) ~ finite([P:bag1,P:bag2,P:bag3]) :=
        P is 1/3.
```

from program 5.17 must be changed to

```
bag_mixture ~ dirichlet([1,1,1]).
```

The entire program can be found here. This is an example of a *mixture model*. The result of this query is:

- Obs1 and Obs2 come from the same bag: P = 89,68 %
- Obs1 and Obs3 come from the same bag: P = 5,09 %

Once again, this corresponds with the result of [1].

5.5.2 Topic Models

In order to look at a more advanced model, we now consider the issue of *topic models* and more specifically *Latent Dirichlet Allocation* [26]. This model gets as input a number of documents as *bag-of-words*, and then tries to cluster together the documents with a common topic. The details are discussed in chapter 11 of [1].

The Church code of this example is slightly involved, but the translation to Distributional Clauses is relatively straightforward. The difficult part is translating the integration of higher-order functions as `map` or `repeat` into other functions. As an example, consider the definition of the `document->topics` function, which takes as input a document ID, and returns randomly a list of topics, one for each word of the document:

```
(define document->topics (mem (lambda (doc-id)
  (repeat  (document->length doc-id)
           (lambda () (multinomial topics (document->mixture-params doc-id)
           ))))))
```

Program 5.18: LDA in Church: code snippet

This function is represented in DC as multiple terms:

- A distributional clause `doc_topics/1` that takes as argument the document ID, but delegates all the work to an normal clause `doc_topics/3`, with arguments the ID and the length of the document.
- `doc_topics/3` in turn, will recursively generate a list of random topics. This corresponds to the `repeat` operation in Church.
- `doc_topics/3` invokes another distributional clause, `pick_topic`, that corresponds with the `multinomial` function from Church. It picks a topic from the topic list according to the parameters for the document in question.

All this is shown in the listing below:

```
doc_topics(DocID) ~ val(Topics) :=
    doc_length(DocID, N),
    doc_topics(DocID, N, Topics).

doc_topics(DocID, 0, []) := true.
doc_topics(DocID, N, [Topic|Topics]) :=
    N > 0,
    pick_topic(DocID, N) ~= Topic,
    N1 is N-1,
    doc_topics(DocID, N1, Topics).

pick_topic(DocID, N) ~ finite(L) :=
    topics(Topics),
    doc_mixparams(DocID) ~= Params,
    zip(Params, Topics, L).
```

Program 5.19: LDA in DC: code snippet

The entire program, together with some code to summarize distributions over discrete values, can be found in the github repository.

Again, all the evidence as used in Church could not be used in DC; it made the inference impractical. The program does work, however, when the number of observed documents and their length is reduced.

The code for the section on *Unknown Number of Categories*, just as all other code for this chapter, can be found on github.

5.6 Non-parametric Models

This chapter deals with the construction of infinite discrete distributions, and more specifically the *Dirichlet process*. There are many ways to construct this model [21]. In this section, the *stick-breaking* metaphor will be used.

This section is based on chapter 12 of [1].

5.6.1 Stick-Breaking

This is a generative process to construct a discrete probability distribution. To draw a sample, walk down the list of natural numbers, and at each number k:

1. If it doesn't yet exist, pick a value β_k from a distribution $\mathcal{B}eta(1, \alpha)$. If it does exist, look up its value.
2. Use this β_k as the weight of a coin, and flip this coin. If it comes up heads, return k as the value of the sample you are drawing.
3. If the coin comes up tails, move on to the next number $k + 1$, and repeat the procedure.

It is thus critically important that we take into account the trace left by the previous samples, namely the values β_k associated with each natural number k.

How this procedure is implemented in Church is laid out in program 5.20. It concisely shows the two aspects of the stick-breaking procedure:

- The generation of the β_k's (also called *sticks*), with the `sticks` function. As the authors of [1] note, it is critically important to memoize this function.
- The function `pick-a-stick` that either returns the current index `J`, or otherwise recursively calls itself with the next index as argument.

```
(define (pick-a-stick sticks J)
  (if (flip (sticks J))
      J
      (pick-a-stick sticks (+ J 1))))

(define sticks
 (mem (lambda (index) (beta 1 alpha))))
```

Program 5.20: Church: Stick-breaking

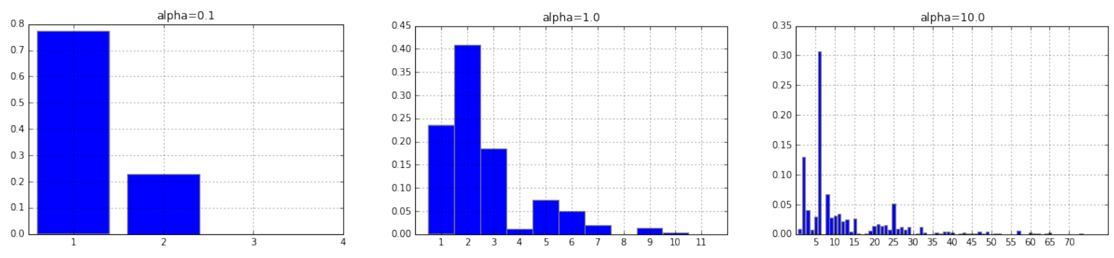
The DC version of 5.20 is shown in 5.21. The crucial memoization is done by giving the `sticks/1` clause an `Index` argument. In this way, an invocation of `sticks(Index) ~= Value` will always return the same value for the same index. The `pick_a_stick/3` follows the same procedure as outlined above: either `flip(Index, N)` is true (lines 10-12), and the `Sample` variable is matched with

`Index`, or it is false and `pick_a_stick/3` calls itself (lines 14-17). The addition of `N`, the sample number, to `flip/2` is needed to avoid the same value of the flip for each index.

```
1   sticks(Index, Alpha) ~ beta(1, Alpha).
2   sticks(Index) ~ val(P) :=
3       alpha(Alpha),
4       sticks(Index, Alpha) ~= P.
5
6   flip(Index, UniqueID) ~ finite([P1:true, P2:false]) :=
7       sticks(Index) ~= P,
8       P1 is P, P2 is 1.0-P.
9
10  pick_a_stick(Index, Sample, N) :=
11      flip(Index, N) ~= true,
12      Sample = Index.
13
14  pick_a_stick(Index, Sample, N) :=
15      flip(Index, N) ~= false,
16      Index1 is Index+1,
17      pick_a_stick(Index1, Sample, N).
```

Program 5.21: DC: Stick-breaking

The concentration parameter α regulates the spread of the discrete samples: if α is small, the β_k's will be large (Beta-distribution skewed to the right), and thus is it very likely the coin flips heads and the sample-generating process stops early. Thus smaller values are preferred. When α is large, the opposite happens, and more different values will be sampled. Figure 5.13 shows how the probability distribution of this stick-breaking process evolves when α increases.



$\alpha = 0.1$ $\alpha = 1.0$ $\alpha = 10.0$

Figure 5.13: Evolution of the stick-breaking distribution as function of α

As mentioned previously, it is important that already generated sticks are re-used when generating later samples. Experimentation with Distributional Clauses showed that this did not happen with the `query` primitive. Thus, to ensure correct behavior, the samples were constructed explicitly with a recursive predicate `get_samples`:

```
get_samples(0, []) := true.
get_samples(N, [Sample|Samples]) :=
    N > 0,
    pick_a_stick(1, Sample, N),
    N1 is N-1,
    get_samples(N1, Samples).
```

The actual generation can then be accomplished with the `generate_backward/2` clause:

```
generate_backward(get_samples(N, Samples), _).
```

5.6.2 The Dirichlet Process

The stick-breaking generative process from the previous section defines a probability distribution over an infinite set, the range of natural numbers. The authors of [1] extend this distribution: instead of returning a natural number as a sample, a draw from another distribution, the *base distribution* is returned. The *Dirichlet process* associates thus with each natural number, coming from the generative process above, a draw from the base distribution. They call this process *stochastic memoization*, and define a higher order function `DPmem` that takes as input the base distribution, and that returns a discrete distribution of which the values are drawn from the base distribution, and the distribution is determined by the stick-breaking process.

Something similar can be done in Distributional Clauses:

```
base_distr(N) ~ gaussian(0.0, 1.0).
dPMem(N) ~ val(X) :=
    pick_a_stick(1, Index, N),
    base_distr(Index) ~= X.
```

Here the base distribution is a normalized Gaussian. The `dPMem` essentially picks a natural number `Index`, and then associates with this number a sample from the base distribution. The next time this number comes up, the same value will be returned - it is memoized.

Figure 5.14 shows a distribution resulting from stochastic memoization of this Gaussian base distribution. In this example, only 4 unique samples were produced.

The code for this section and for the previous one, can be found online.

5.6.3 Applications

In [1], the authors use stochastic memoized functions (and most commonly the `gensym` function) as priors on discrete variables. The advantage is that the number of objects must not be predefined, but can nevertheless be steered by adjusting the value of the concentration parameter α.

In the *Infinite Relational Model* example, based on [23], data about relations between objects is given, and the goal is to cluster the objects in classes. Whether a

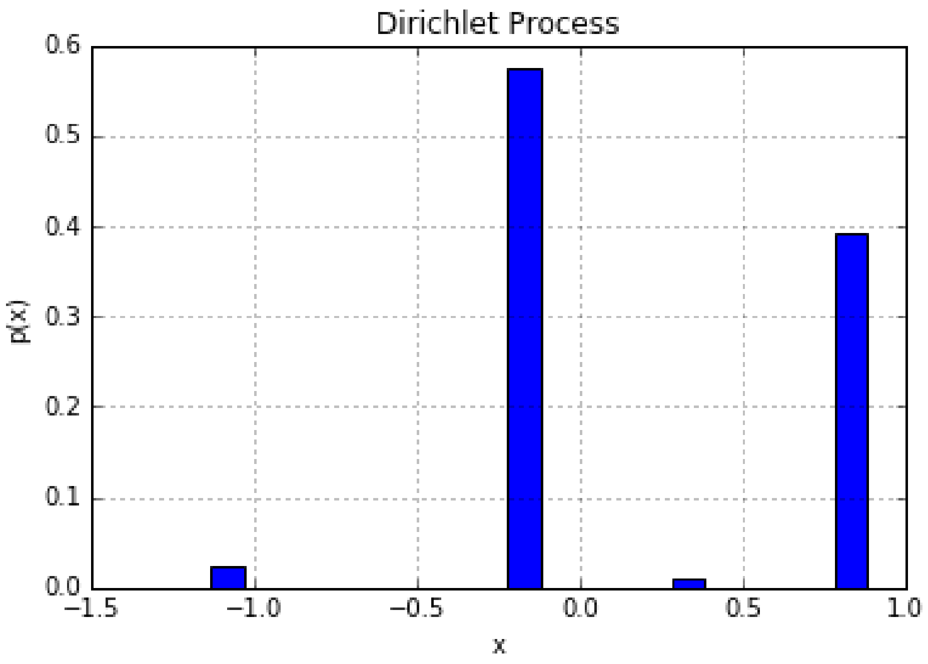


Figure 5.14: Stochastic Memoization of a Gaussian ($\alpha = 1.0$)

relation holds or not depends on the combination of classes from which the objects come.

The concrete example gives a list of pair of individuals, whom either talk or don't talk to each other. A stochastic memoized prior is put on the distribution of classes. A slightly simplified version of the Church program in given in 5.22

```
(define samples
  (query
   (define class-distribution (DPmem 1.0 gensym))
   (define object->class
     (mem (lambda (object) (class-distribution))))
   (define classes->parameters
     (mem (lambda (class1 class2) (beta 0.5 0.5))))
   (define (talks object1 object2)
     (flip (classes->parameters (object->class object1) (object->class object2))))

   (list (equal? (object->class 'tom) (object->class 'fred)))

   (and (talks 'tom 'fred)
        (talks 'tom 'jim)
        ; ...
        )))
```

Program 5.22: Church: Infinite Relational Model

A simplified version of the DC implementation of this model is given in program 5.23. The complete program can be found on the github page (link).

In this case, no explicit sample generation as above is needed, since we are not interested in the distribution of the memoized stochastic function, but rather in an ordinary query. The generation and remembering the indices of the stick-breaking process is done with the definition of the evidence.

```
base_distr(N) ~ val(X) := gensym('class', X).

object_class(Object) ~ val(Class) :=
    dPMem(Object) ~= Class.

classes_params(Class1, Class2) ~ beta(0.5, 0.5).
talks(Object1, Object2) ~ finite([PT:true, PF:false]) :=
    object_class(Object1) ~= Class1,
    object_class(Object2) ~= Class2,
    classes_params(Class1, Class2) ~= PT, PF is 1.0-PT.

evidence :=
    talks(tom, fred) ~= true,
    talks(tom, jim) ~= true,
    % ...

test(N) :-
    init,
    query([evidence], [], (object_class(tom)  ~= Class1,
                          object_class(fred) ~= Class1), N, P),
    writeln(P).
```

Program 5.23: DC: Infinite Relational Model

The query of whether `tom` and `fred` belong to the same class is done by formulating the query such that the instantiations of the class of `tom` and `fred` match with each other: `object_class(tom)  ~= Class1, object_class(fred) ~= Class1`. Another query was also considered: do `tom` and `mary` belong to the same class?

The result of these two queries is:

- tom/fred : 51%
- tom/mary : 21%

These are results that are similar as those of Church. However, Church is more efficient: the execution time is in the order off a few seconds, while DC almost takes a minute to complete. This is likely due to a lot of evidence (in total 12 `talks/2` relation are taken into account). Chapter 6 studies these issues in more detail.

5.6.4 Hierarchical Combinations of Non-Parametric Models

In this section, multiple non-parametric models are combined in a hierarchical fashion. One such example is the *Nested Chinese Restaurant Process* (nCRP) [24]: multiple subordinate Dirichlet processes depend a top-level process. In Church, this is implemented in a few lines of code:

```
(define top-gensym (make-gensym "t"))
(define top-level-category (DPmem 1.0 top-gensym))

(define subordinate-gensym (make-gensym "s"))
(define subordinate-category
  (DPmem 1.0
         (lambda (parent-category)
           (list (subordinate-gensym) parent-category))))

(define (sample-category) (subordinate-category (top-level-category)))
```

Program 5.24: Church: Nested Chinese Restaurant Process

Basically, the `subordinate-category` function is now a stochastic memoized function that takes as argument the category of its higher level, which in turn is a stochastic memoized function `top-level-category`. In DC adding this argument to a memoized function also requires adding an additional identifier to the different clauses that make up the stick-breaking process. In this way, we can avoid the implicit memoization of logic programming.

The code in 5.25 shows how the `dPMem` clauses was adapted: an additional `Level` identifier was added. Furthermore, the top and subordinate invocations are treated differently, since in the subordinate case, we first need to invoke the top-level `dPMem` clause. Lines 6 and 11 show that the level identifier is also passed to the `stick-breaking` clauses. This is necessary to avoid that the implicit memoization would generate the same values for each `dPMem` sample. Also noteworthy is the use of the `ParentCat` identifier in line 11, to be able to distinguish between subordinate from different top-level categories.

```
1   base_distr(top, N) ~ val(X) := gensym('t', X).
2   base_distr(subordinate, N) ~ val(X) := gensym('s', X).
3
4   % dPMem(Level, N).
5   dPMem(top, N) ~ val(X) :=
6       pick_a_stick(top, 1, Index, N),
7       base_distr(top, Index) ~= X.
8
9   dPMem(subordinate, N) ~ val(c(X, ParentCat)) :=
10      dPMem(top, N) ~= ParentCat,
11      pick_a_stick(subordinate(ParentCat), 1, Index, N),
12      base_distr(subordinate, Index) ~= X.
```

Program 5.25: DC: Nested Chinese Restaurant Process

This method allows to construct sets of hierarchically nested categories. To generate data, each category must be associated with its own probability distribution. In the code snippet below, based on the Church code from [1], this is done by drawing the parameters of a category from a Dirichlet distribution. The pseudo-counts for the Dirichlet distributions of the subordinate categories come from their respective top-level category (lines 5-7).

```
1  possible_observations([a, b, c, d, e, f, g]) := true.
2
3  topcat_params(Cat) ~ dirichlet([1,1,1,1,1,1,1]).
4
5  subcat_params(Cat) ~ dirichlet(L) :=
6      Cat = c(SubCat, TopCat),
7      topcat_params(TopCat) ~= L.
8
9  sample_cat(N) ~ val(Cat) :=
10     dPMem(subordinate, N) ~= Cat.
11
12 sample_obs(N) ~ finite(L) :=
13     sample_cat(N) ~= Cat,
14     subcat_params(Cat) ~= Params,
15     possible_observations(PosObs),
16     zip(PosObs, Params, L).
```

Program 5.26: Generating samples from a Nested Chinese Restaurant Process

This program combines a Dirichlet/multinomial hierarchical structure, to generate data, with a nCRP to generate the hierarchical structure. The complete DC program of this example can be found on the github page.

Chapter 6

Performance Evaluation

The goal of this chapter is to compare the performance of Church, ProbLog and Distributional Clauses. However, this is not always straightforward, because of the differences between ProbLog on the one hand, and Church and DC on the other:

1. ProbLog gives an exact solution, while the solution offered by both Church and DC is always approximate

2. ProbLog can only handle discrete distributions, while Church and DC can also handle continuous parameters

In an effort to accurately compare these different languages, a purely discrete model is considered first, and in a next subsection, a hybrid model is studied. Both these models will have an analytical solution, so we can compare the results with the exact solution. Focus will be on two aspects: speed and precision.

In the discrete model, ProbLog will give the exact solution, so precision is not an issue. I will focus on how more evidence impacts the performance. DC and Church will be compared with regard to both speed and precision. There will be an obvious trade-off: more samples will lead to higher precision, but will also slow down the execution.

In the hybrid model, ProbLog is confronted with a trade-off: the continuous parameter will be discretized. A finer resolution will lead to higher precision, but will slow down the execution. Church and DC will once again be compared with regard to speed and precision.

A few notes:

- All experiments were done on a Lenovo Ideapad 510 with an Intel i7-6500u dual-core CPU at 2.5GHz, and 8 GB DDR4 RAM.

- Since I only had access to the web interface, all experiments with Church were executed within Google Chrome (version 58.0.3029.110 (64-bit))

- The ProbLog version is 2.1.0.18 with Python 2.7.12

- The YAP version (for DC) is YAP 6.2.2

6.1 Purely Discrete Model

A simple discrete model is used for this first experiment: learning whether or not a coin is fair. This example was already programmed in ProbLog in section 5.2.1:

1. A coin is either fair, with probability 0.999, or it is unfair.
2. If it is fair, it lands head half the time, and tails half the time.
3. If it is unfair, it lands heads 95% of the time, and tails only 5% of the time.

We observe a sequence of heads of length N. Analytically, this problem is solved as follows:

1. Define event $\boldsymbol{F}$ as drawing a fair coin, and $\boldsymbol{T} = \neg\boldsymbol{F}$ as drawing a trick coin.
2. The prior: $P(\boldsymbol{F}) = 0.999$
3. Likelihood of N heads if the coin is fair: $P(\{h_i\}^N|\boldsymbol{F}) = \prod_{i=0}^{N} p(h|\boldsymbol{F}) = 0.5^N$
4. Likelihood of N heads if the coin is unfair: $P(\{h_i\}^N|\boldsymbol{T}) = \prod_{i=0}^{N} p(h|\boldsymbol{T}) = 0.95^N$
5. Posterior of a fair coin: $P(\boldsymbol{F}|\{h_i\}^N) = \frac{1}{Z}P(\{h_i\}^N|\boldsymbol{F})P(\boldsymbol{F})$[1]

where $\{h_i\}^N$ represents a sequence of N heads. Plugging in the number of observed heads then gives the required probability.

6.1.1 ProbLog

As mentioned previously, this probabilistic model was implemented in ProbLog in section 5.2.1. It can also be found on github (link).

The model itself is fixed, including the number of probabilistic facts and thus the number of atomic choices and number of worlds ω that are well-founded models of the logic program. Only the evidence (the number of observed heads) changes. Figure 6.1 shows the how the execution delay of the ProbLog program evolves as function of the amount of evidence. Each delay is measured as the average value of 10 runs. The increase is almost linear: the R^2 value is 0.98. However, there also is a second-order term with a low coefficient present.

This delay includes both compilation of the program to a weighted boolean formula, and the inference with weighted model counting. Since only the evidence changes during runs, only the inference part of the execution is impacted.

Obviously, all the results obtained with ProbLog for $P(\boldsymbol{F}|\{h_i\}^N)$ correspond perfectly with the theoretical results.

[1] Z is the normalization factor: $Z = P(\{h_i\}^N|\boldsymbol{F})P(\boldsymbol{F}) + P(\{h_i\}^N|\boldsymbol{T})P(\boldsymbol{T})$

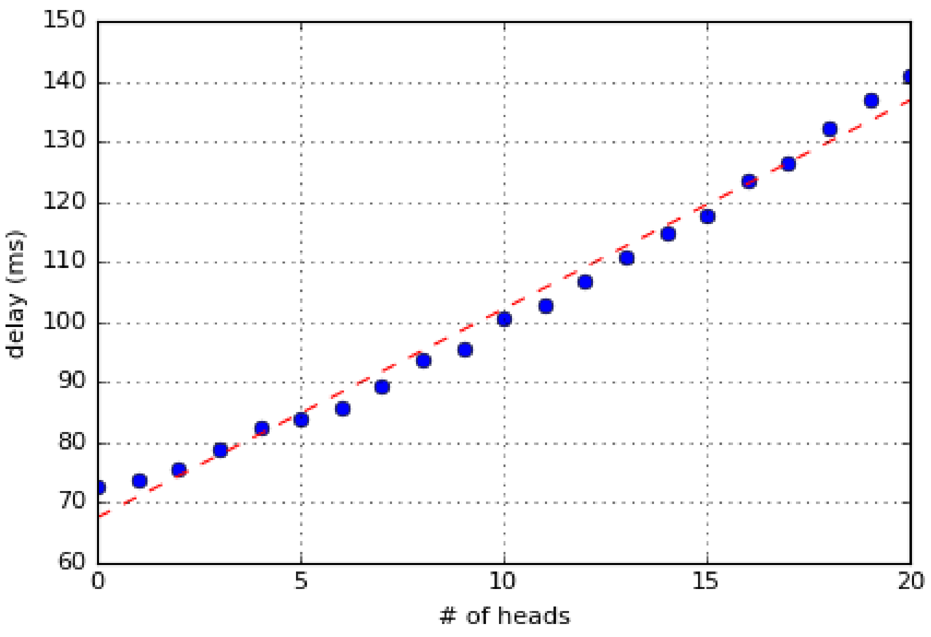


Figure 6.1: ProbLog: Execution delay

6.1.2 Church

Since Church is sampling-based, the result will be an approximation of the real value. Thus, I'll first study the convergence behavior of the program, before execution time is considered. In order to do this, I set the value of observed heads to 17. In this case, it is highly likely to conclude that the coin is unfair, although there is still about 1.7% chance that it is fair.

A slightly adapted version of the program presented in chapter 8 of [1] was used:

```
(define observed-data '(h h h h h h h h h h h h h h h h h))
(define num-flips (length observed-data))

(define (samples N)
  (mh-query
     N 10

     (define fair-prior 0.999)
     (define fair-coin? (flip fair-prior))

     (define make-coin (lambda (weight) (lambda () (if (flip weight) 'h 't))))
     (define coin (make-coin (if fair-coin? 0.5 0.95)))

     fair-coin?

```

```
      (equal? observed-data (repeat num-flips coin))))

(define nsamples '(100 200 300 400 500 600 700 800 900 1000))
(define (get-mean sample-list) (mean (map (lambda (s) (if s 1 0))
                                           sample-list)))
(map (lambda (n) (repeat 10 (lambda () (get-mean (samples n)))))
       nsamples)
```

The inference method is thus Metropolis-Hasting, as explained in 2.1.4. After some experimentation, I kept the burn-in rate of 10 samples, and let the number of samples vary from 100 to 1000 with a step size of 100. The value of $P(\boldsymbol{F}|\{h_i\}^N)$ is then estimated by computing the ratio of samples for which the coin is fair (by means of the `get-mean` function).

Every instance was ran 10 times, to get an impression of the average result and the variance. The result is shown in figure 6.2, where the red line represents the true value of concluding the coin is fair. The blue line is the mean over the 10 runs, and the shaded area around the mean covers 1 standard deviation[2].

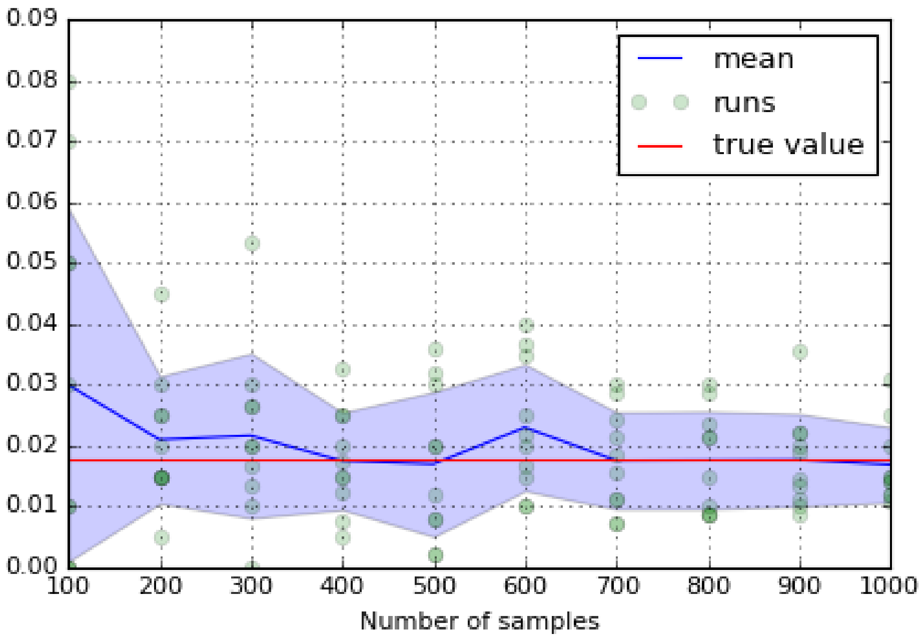


Figure 6.2: Church: Sampling Performance

Initially, the spread is large, but it becomes smaller when the number of samples increases. Similarly, while initially the mean is off, it does approach the true value after when about 900 samples are generated. The standard deviation with 1000

[2]The true distribution of values in this case is certainly not symmetric around the mean, since the domain is limited by 0 at the lower limit. However, representing the spread in this way gives an impression of the incertitude due to the sampling process.

samples is about 0.006; with 10000 samples, this reduces to about 0.001, which means that we are within 6% of the true value.

Comparing exact (ProbLog) and approximate (Church) methods requires some pragmatism. For this purpose, I chose to proceed with 1000 samples in Church, although this choice is arbitrary and highly dependent on the application. Even so, this gave a delay of 710 ms (averaged over 50 runs). Compare this with the value of ProbLog for the same number of heads (138 ms), and we can conclude that, for this application and these assumptions, ProbLog performs an order of magnitude better than Church.

6.1.3 Distributional Clauses

The version of the program in Distributional Clauses closely mimics the Church version. The program is shown below (link):

```
fair_prior(0.999) := true.
is_fair ~ finite([PT:true, PF:false]) :=
    fair_prior(PT),
    PF is 1.0-PT.

coin_flip(N) ~ finite([0.5:h, 0.5:t]) :=   % fair coin
    is_fair ~= true.
coin_flip(N) ~ finite([0.95:h, 0.05:t]) := % trick coin
    is_fair ~= false.

% observations(N, ListofObs)
observations(0, []) := true.
observations(N, [Sample|Rest]) :=
    coin_flip(N) ~= Sample,
    N1 is N-1,
    observations(N1, Rest).

observed_data([h,h,h,h,h, h,h,h,h,h, h,h,h,h,h, h,h]).

test(NSamples) :-
    init,
    observed_data(ObservedData),
    length(ObservedData, L),
    query([observations(L, ObservedData)], [], is_fair ~= true, NSamples, P),
    writeln(P).
```

In this case, since it concerns a binary variable, the `query` clause is used. In ProbLog, we specify the number of samples we wish to generate in this clause. However, due to the importance sampling routine, and the fact that the evidence constrains the posterior, some samples will have a weight of zero, which means they won't count when computing the probability of `is_fair ~= true`. A corollary is that sometimes, we have to set the numbers of samples very high when the evidence points to an a priori rare situation, to get in the end enough samples to accurately assess the wanted result.

Once again, I work with an observation of a sequence of 17 heads. A bit of experimentation showed that we will have to set the number of samples very high to obtain results comparable with those in Church. At least 100000 samples were needed. The result is shown in figure 6.3, where the red line represents the true value, the blue line is the mean, taken over 10 runs for each number of samples, and the blue shaded area is the region of one standard deviation around the mean (the same caveat applies concerning the non-symmetry around the mean). Even if the number of samples is high, it can happen that all the generated samples have a weight zero, so no conclusions can be drawn from that particular run. For example, when using 100000 samples, it did happen that the returned value of the a posteriori probability is zero.

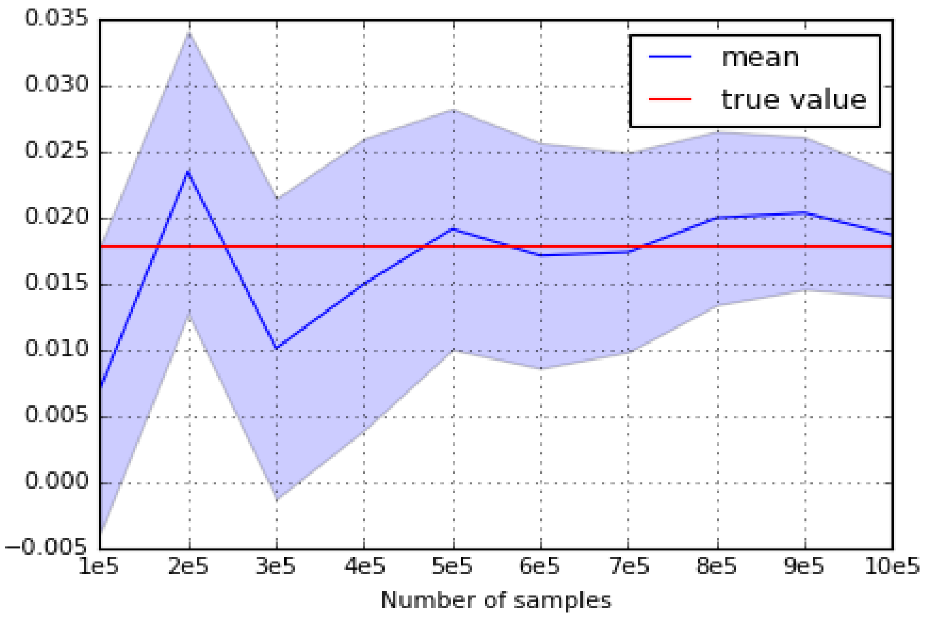


Figure 6.3: DC: Sampling Performance

When the number of samples is 700000, we are closely in the same performance region as the Church run with 1000 samples from above. However, in DC, this took 7340 ms (averaged over 10 runs), almost two orders of magnitude more than Church. Thus, we can conclude that the inference mechanism of DC, importance sampling with likelihood weighting, is less efficient than the Metropolis-Hastings sampling of Church.

6.2 Hybrid Model

In this section, the inference of a continuous parameter is compared for Church, Distributional Clauses and ProbLog (where this parameter is approximated by a multivalued discrete parameter). The example that I'll use is the estimation of the weight of coin, as in 5.2.2, and is a extension of the program used in the previous section.

The example can be summarized as follows:

- There is a coin of which the weight w (probability of landing heads) is unknown. The prior distribution of this weight is uniform in the domain [0,1]: $p(w) = U(0, 1)$.
- The coin is flipped several times. During these flips, we observe a sequence $\boldsymbol{S}$ of N_H heads and N_T tails.
- It can then be shown ([18]) that the posterior distribution $p(w|\boldsymbol{S})$ is a beta-distribution $\mathcal{B}(\alpha, \beta)$ with $\alpha = N_H + 1$ and $\beta = N_T + 1$.

The existence of this exact solution will provide a baseline with which the different results can be compared.

6.2.1 ProbLog

The ProbLog version of this program was already explored in section 5.2.2. The weight value w was discretized:

```
weights([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]).
coin_weight(X) :-
   weights(Weights),
   select_uniform(weight, Weights, X, _).
```

This `coin_weight` is the variable that is queried, after observing a sequence of heads and tails. If this sequence contains 4 heads and 1 tail, the posterior distribution looks like figure 6.4, whereby 21 discrete values of the weigth w are used. To compare, a (scaled) version of the $\mathcal{B}eta(5, 2)$ distribution is also drawn (in red).

The issue here however, is the number of discrete points to use to approximate the continuous parameter. Each additional value increases the number of atomic choices in a linear fashion. The execution time of the program will then increase according to the time complexity of the algorithm.

Figure 6.5 shows the increase in execution delay when the number of choice points increases (blue line), and the best quadratic fit (pink). This increase is almost perfectly quadratic, with a relatively small coëfficient for the quadratic term. This shows that the complexity of the ProbLog algorithm - the combination of compilation to a weighted boolean formula and the different inference steps - is of order $\mathcal{O}(n^2)$. This stands in contrast with the result of figure 6.1, where only varying evidence was considered, and the delay was linear.

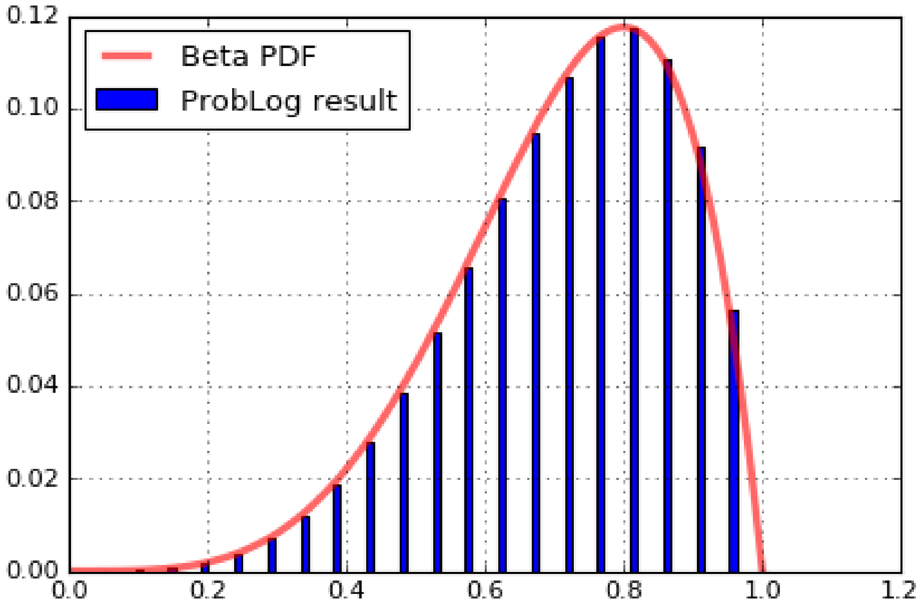


Figure 6.4: ProbLog: Continuous Parameter

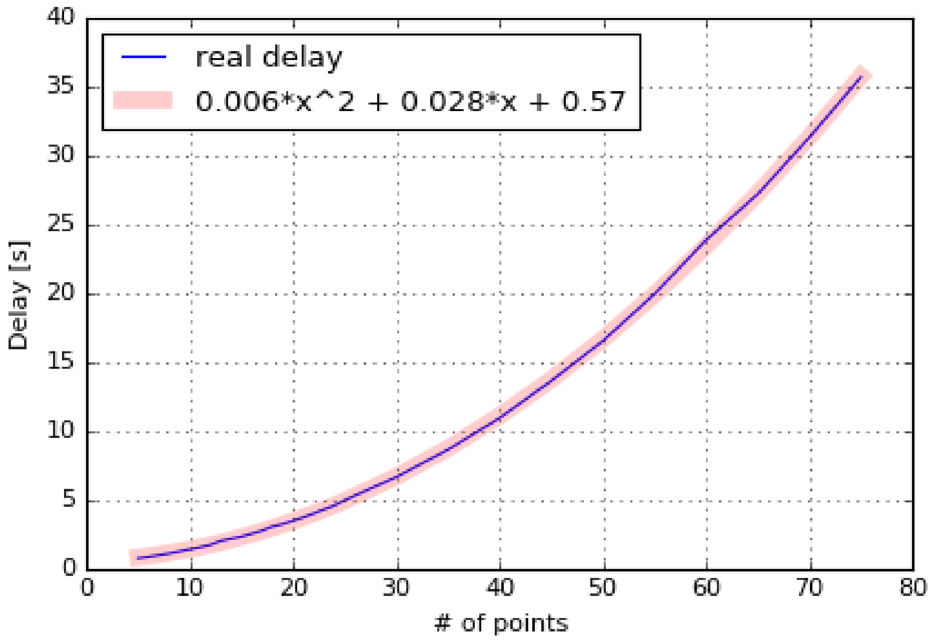


Figure 6.5: ProbLog: execution delay as function of number of atomic choices

Increasing the resolution of a simulated continuous variable by one is a linear increase: it is similar to adding a new branch to the tree of possible worlds, as shown in 6.6. Each dot represents a choice.

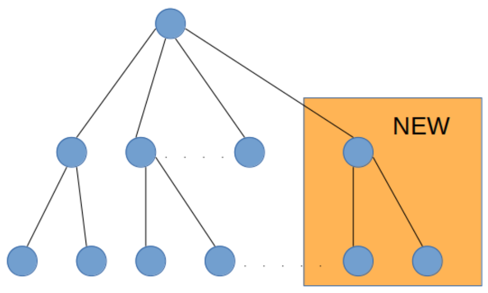


Figure 6.6: ProbLog: Horizontal (linear) addition

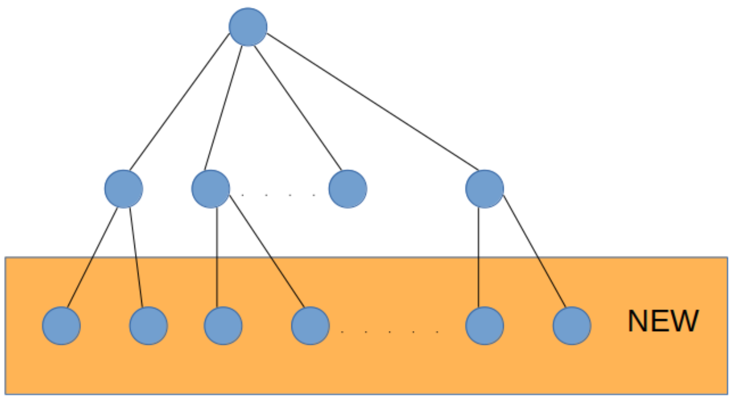


Figure 6.7: ProbLog: Vertical (exponential) addition

However, there is also another way of increasing the complexity of the program (or identically, the size of the tree of choice points), namely by adding additional layers - increasing the depth of the tree. Figure 6.7 shows how this is accomplished figuratively, while the ProbLog program below shows how this can be done in reality:

```
0.5::initial_flip.

0.5::layer(1) :- initial_flip.
0.5::layer(1) :- \+initial_flip.

0.5::layer(N) :-
    N > 1,
    N1 is N-1,
    layer(N1).
0.5::layer(N) :-
    N > 1,
    N1 is N-1,
    \+layer(N1).
```

```
14
15 query(initial_flip).
16 evidence(layer( 55 )).
```

Line 1 sets up the initial choice: `initial_flip`. Lines 8 to 15 keep adding layers: the choice in each layer is determined by chance, but also by the value of the preceding layer. We add the last layer in evidence (line 18), while we query `initial_flip`. Since each layer introduces two additional worlds, the number of possible worlds increases exponentially. Because of the symmetry, the result of `query(initial_flip)` will always be 0.5.

The result is shown in figure 6.8: the blue line is the delay that was measured - for each depth, the average was taken over 5 runs. The pink line is the best fit of an exponential-quadratic term. Clearly, the time complexity of this program increases as $\mathcal{O}(e^{n^2})$, where n is the number of layers. This is consistent with the observations that (a) the time complexity of the inference algorithm is $\mathcal{O}(m^2)$, where m in this case are the number of choices, and (b) that the program increases the number of atomic choices in an exponential fashion.

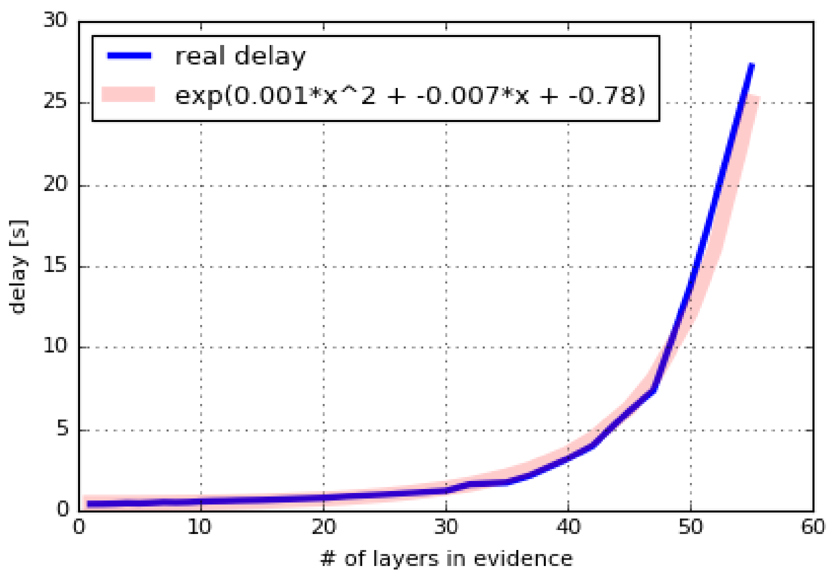


Figure 6.8: ProbLog: Exponential growth of the execution time

6.2.2 Church

In order to compare the performance of these approximate methods when it concerns a continuous distribution, we need a method to compare the distribution obtained through sampling, with the exact distribution. One way to measure the difference between two distributions is the *Kullback - Leibler divergence*, which basically measures the increase in entropy when we compute the entropy of a distribution Q not relative to its own distribution, but with respect to another distribution P. This is the Kullback-Leibler divergence from Q to P is often denoted $D_{KL}(P||Q)$:

$$D_{KL}(P||Q) = -\int_{-\infty}^{+\infty} p(x)\log(q(x))dx + \int_{-\infty}^{+\infty} p(x)\log(p(x))dx \tag{6.1}$$

$$= \int_{-\infty}^{+\infty} p(x)\log\frac{p(x)}{q(x)}dx \tag{6.2}$$

If $D_{KL}(P||Q)$ is small, then distribution Q closely resembles P.

The Church program used in this subsection is the one used in chapter 8 of [1]:

```
(define observed-data '(h h h t h))
(define num-flips (length observed-data))
(define num-samples 2000)
(define prior-samples (repeat num-samples (lambda () (uniform 0 1))))

(define samples
  (mh-query
   num-samples 10

    (define coin-weight (uniform 0 1))

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

    coin-weight

    (equal? observed-data (repeat num-flips coin))))
```

In order to compute the KL divergence of the distribution generated by this program, I followed this procedure:

1. Generate N samples from it,
2. Divide these samples over 20 bins between 0 and 1,
3. Compute the KL-divergence between the frequencies of these bins, and the true distribution $\mathcal{B}eta(5,2)$

Figure 6.9 sketches the behavior of (the logarithm of) this divergence. As the number of samples N increases, the KL-divergence decreases. However, from 1000 samples onward, the marginal gain becomes small.

6.2.3 Distributional Clauses

To compare the performance of DC to that of Church, the same procedure was applies. The program below is the implementation of the previous Church program in DC. Mind that a $\mathcal{B}eta(1,1)$ distribution is used - this distribution is identical to a uniform distribution over [0,1].

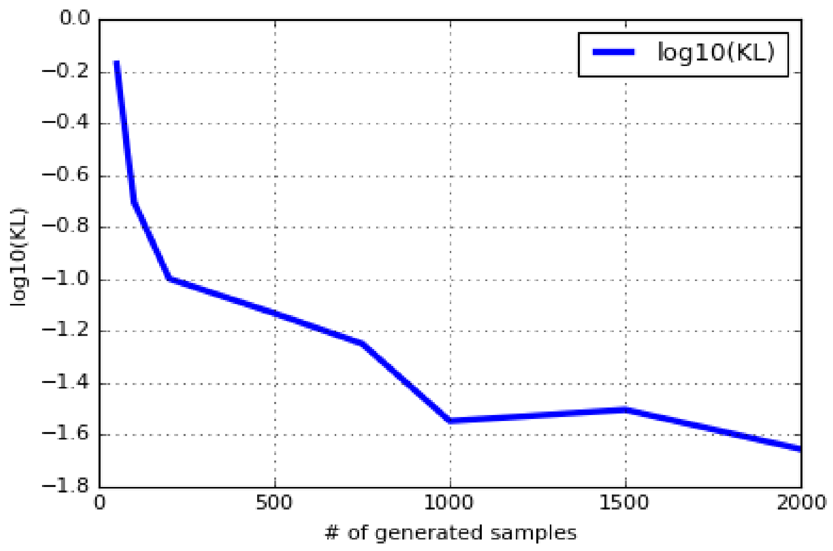


Figure 6.9: Church: KL-divergence

```
observed_data([h,h,h,t,h]).
coin_weight ~ beta(1,1). % uniform distribution of [0,1]

coin_flip(N) ~ finite([PH:h, PT:t]) :=
    coin_weight ~= PH, PT is 1.0-PH.

% observations(N, Obs)
observations(0, []) := true.
observations(N, [Sample|Rest]) :=
    coin_flip(N) ~= Sample,
    N1 is N-1,
    observations(N1, Rest).

test(NSamples) :-
    init,
    observed_data(ObservedData),
    length(ObservedData, L),
    eval_query_distribution_eval(X, [observations(L, ObservedData)], [],
                                 coin_weight ~= X, NSamples, LP, _, _),
    writeln(LP).
```

Again, different number of samples were generated. For this program and this evidence (4 heads and 1 tail), about 3% of the requested samples were effectively generated. Figure 6.10 shows the evolution of the KL-divergence. Comparing this to figure 6.9, we see that we must request about 15000 samples (corresponding to about 500 generated samples), to obtain a result comparable with 1000 samples in

Church. These two numbers will be used to compare the run time.

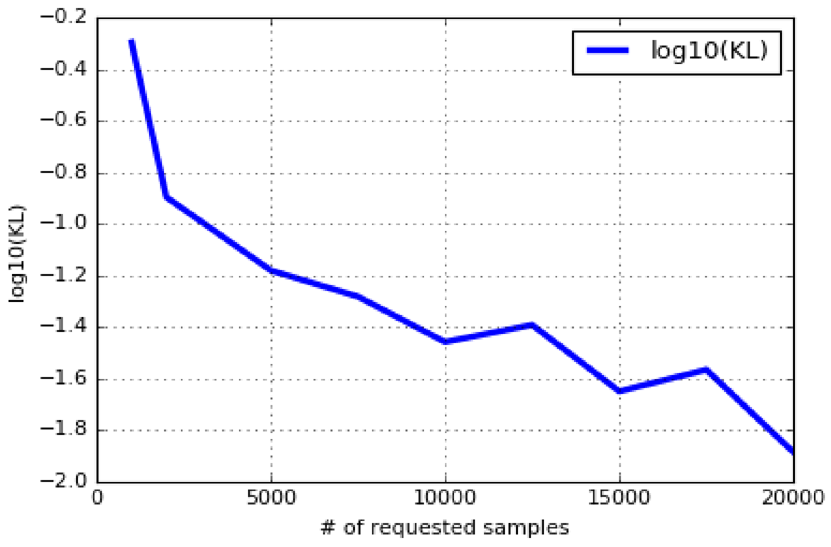


Figure 6.10: DC: KL-divergence

The average execution times were:

- Church: 121.61 ms (100 runs)
- DC: 312.38 ms (100 runs)

Thus Church is still faster, but the difference is less than in the previous section. This is most likely due to the evidence being less restricting.

Chapter 7

Conclusion

The goal of this thesis was to port a number of probabilistic cognitive models from Church, a functional probabilistic programming language, to ProbLog, a probabilistic logic programming language. The different cognitive models represented probabilistic representations of aspects of intelligent reasoning.

The first conclusion is that for certain types of problems, most notably those problems that require the propagation of distributions over distributions, like hierarchical models, the implementation in ProbLog became unwieldy. Discretizing the probability distributions leads to a combinatorial explosion of the number of possible worlds. Therefore, these type of models were implemented in Distributional Clauses, another probabilistic logic language that does allow the use of continuous distributions.

The key to translating programs from Church to ProbLog is the identification of the evidence, the query and the generative model. The two former items can normally be readily translated; the model may require some work. A small list of practical issues relating to this was examined in chapter 3.

A key difference, which can lead to a slew of problems, are the different approaches to memoization. This must be done explicitly in Church, while it is implicit in ProbLog. It requires continual attention to anticipate possible problems due to memoization.

Distributional Clauses is closer to Church than ProbLog, both in syntax, semantics (availability of continuous distributions) and inference (sampling based). However, many ideas and practices from ProbLog can be re-used when working with Distributional Clauses, mostly techniques related to logic programming.

Chapter 6 studied the performance of the three different probabilistic programming languages. Two aspects were examined: precision and speed. Since the inference mechanism of ProbLog is based on weighted model counting, it will always returns the exact answer. The precision of both Church and DC increases when the number of generated samples increases. In general, we can say that to obtain the same level of precision, Church is more efficient than DC, surely when there is a lot of evidence that restricts the area of the posterior where the probability mass is concentrated. ProbLog, in turn, can suffer from a combinatorial increase of the number of possible worlds, resulting in an exponential time complexity.

Some models require further study. Section 5.1 on inference about inference contains some models that might need improvement. Another chapter than may need additional work is the last section on non-parametric models, where the higher-order programming in Church (the `DPmem` function) posed particular problems and those might have been solved more efficiently. Finally, some of the DC models can surely be improved and made more efficient.

Bibliography

[1] N.D. Goodman and J.B. Tenenbaum, *Probabilistic Models of Cognition*, Retrieved 18/04/2017, from `https://probmods.org/v1/`

[2] Wikipedia, *Cognitive Science* , Retrieved 18/04/2017,

[3] Judea Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*, 1988 Morgan Kaufmann Publishers Inc.

[4] *probabilistic-programming.org*, Retrieved 18/04/2017, `http://probabilistic-programming.org/wiki/Home`

[5] A. Dries et al., *ProbLog2: Probabilistic logic programming*, Lecture Notes in Computer Science, 9286, pp. 312 - 315, Springer, 2015.

[6] D. Fierens et al., *Inference and learning in probabilistic logic programs using weighted Boolean formulas*, Theory and Practice of Logic Programming, 15:3, pp. 358 - 401, Cambridge University Press, 2015.

[7] L. De Raedt and A. Kimmig, *Probabilistic Programming Concepts*, Machine Learning, 100:1, pp. 5 - 47, Springer New York LLC, 2015.

[8] Wikipedia, *Prolog syntax and semantics*, Retrieved May 2017

[9] Ivan Bratko, *Prolog Programming for Artificial Intelligence (4th Edition)*, 2011, Pearson Education

[10] Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., & Tenenbaum, J., *Church: a language for generative models*, 2008, Uncertainty in AI

[11] Sato T, Kameya Y., *PRISM: A language for symbolic-statistical modeling*, 1997 Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97), Morgan Kaufmann Publishers, pp 13301339

[12] Poole D., *Abducing through negation as failure: Stable models within the independent choice logic*, 2000, Journal of Logic Programming 44(13):535

[13] Vennekens J., Verbaeten S., Bruynooghe M. *Logic programs with annotated disjunctions*, 2004, Proceedings of the 20th International Conference on Logic Programming (ICLP-04), Springer, Lecture Notes in Computer Science, vol 3132, pp 431445

[14] Jackendoff, R. S. *X Syntax: A Study of Phrase Structure*, 1981, Linguistic inquiry monographs. MIT Press.

[15] Gopnik, A., & Sobel, D.M., *Detecting blickets: How young children ues information about causal powers in categorization and induction*, 2000, Child Development, 75(5), 1205-1222.

[16] Davide Nitti, *Hybrid Probabilistic Logic Programming*, PhD thesis, 2016

[17] Gutmann et al., *The Magic of Logical Inference in Probabilistic Programming*, Theory and Practice of Logic Programming, 2011

[18] Christopher Bishop,*Pattern Recognition and Machine Learning*, Springer-Verlag New York, 2006

[19] David J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003

[20] ProbLog 2.1 documentation, http://problog.readthedocs.io, retrieved 20/04/17

[21] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, *Hierarchical Dirichlet processes*, J. American Statistical Association, vol. 101, no. 476, pp. 15661581, Dec. 2006.

[22] J. Sethuraman. *A constructive definition of Dirichlet priors.* Statistica Sinica, 4:639650, 1994.

[23] Kemp, C., Tenenbaum, J. B., Griffiths, T. L., Yamada, T. & Ueda, N., *Learning systems of concepts with an infinite relational model.*, 2006 Proceedings of the 21st National Conference on Artificial Intelligence

[24] Blei, D. M., Griffiths, T. L., Jordan, M. I., and Tenenbaum, J. B, *Hierarchical topic models and the nested chinese restaurant process.*, 2004, Advances in Neural Information Processing Systems 16

[25] Hastings, W.K., *Monte Carlo Sampling Methods Using Markov Chains and Their Applications*, 1970, Biometrika. 57 (1): 97109.

[26] Blei, D., Ng, A., Jordan, M., *Latent Dirichlet Allocation*, 2003, The Journal of Machine Learning Research.

Appendix A

Logic Programming

This appendix provides a short overview of logic programming and related concepts. It is focused on Prolog, although other logic programming languages exist (Answer Set Programming, Datalog). This overview is based on [9] and [8].

A.1 Data Types

A Prolog program is a collection of *terms*, the only data type present in Prolog. It has several subtypes:

- An *atom* is a general-purpose name with no inherent meaning. It is composed of a sequence of characters that is parsed by the Prolog reader as a single unit.
- *Numbers* are ordinary float or integers.
- *Variables* are strings beginning with an uppercase letter. The are placeholders for arbitrary terms and can become instantiated (bound to a specific value) through a process called *unification*.
- A *compound term* is a term composed of an atom called the *functor* and a number of arguments, which are terms in their own right: i.e. $f(x_1, x_2, ..., x_n)$. The number of arguments N is called the arity of the term.
- A *list* is a special case of a compound term. it is defined recursively as being a functor '.' having two term as arguments: the first term is an arbitrary element and is called the head of the list; the second element is another list and is called the tail.

A.2 Prolog Programs

Prolog programs describe relations, defined with *clauses*. Specifically, Prolog uses *Horn clauses*, a turing-complete subset of first-order predicate logic. There are two type of clauses: facts and rules.

A *rule* is a clause of the form:

```
Head :- Body.
```

and is read “Head is true if Body is true”. The body is a set of calls to predicates, which are the goals of the rules. Goals can be part of a *conjunction*, in which case they all have to be true, or can be part of a disjunction, meaning just one has to be true on order for the entire body to be true.

A *fact* is a rule with an empty body, and is always true.

A.3 Evaluation of Programs

The execution of a Prolog program is initiated by the user’s posting of a single goal, called the *query*. Prolog tries to find a resolution refutation of the negated query, as is standard in traditional first-order logic inference. It uses a resolution method called *SLD resolution*. If the negated query can be refuted, the query follows logically from the program. In that case, all variable bindings are reported to the user.

When multiple possibilities for unification are encountered (multiple clause heads can match a given call), the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any of the goals fail, the systems backtracks the nearest choice-point, and examines another alternative until another goal fails or all goals succeed. When backtracking, all variable bindings since the most recent choice-point are undone.

A.4 Semantics

The meaning of a logic program are the conclusions that can be drawn from it - all the possible worlds entailed by the program. Under a declarative reading, the order of rules, and of goals within rules, is irrelevant since logical disjunction and conjunction are commutative. Procedurally, however, it is often important to take into account Prolog’s execution strategy, either for efficiency reasons, or due to the semantics of impure built-in predicates for which the order of evaluation matters. Also, as Prolog interpreters try to unify clauses in the order they are provided, failing to give a correct ordering can lead to infinite recursion.

Master thesis filing card

Student: Koen Boeckx

Title: Implementing Cognitive Models in ProbLog

Dutch title: Implementatie van Cognitieve Modellen in ProbLog

UDC: 621.3

Abstract:
Probabilistic programming languages have become a very powerful and popular tool for statistical modeling and inference. One of these languages, Church, has been used to model a number of cognitive models in the online book *Probabilistic Models of Cognition* by Noah Goodman and Joshua Tenenbaum. Another probabilistic programming language, ProbLog, is based on Prolog and is being developed at the department of Computer Science at the Katholieke Universiteit Leuven. The goal of this master thesis is to take the cognitive models as they are programmed in Church, and port them to ProbLog. It is therefore necessary to establish a set of general rules applicable for the transition from Church to ProbLog. Some of these models however, are based on hierarchical structures that pass continuous probability distributions from one layer to the next. Since ProbLog cannot handle continuous distributions, Distributional Clauses, another probabilistic logic programming language of the Prolog family was used to model these programs. Finally, the performance of these probabilistic languages was compared. Both speed of execution and precision were taken into account.

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Thesis supervisors: Prof. dr. Luc De Raedt
Dr. Angelika Kimmig

Assessors: Dr. Jonas Vlasselaer
Dr. ir. Tinne De Laet

Mentor: