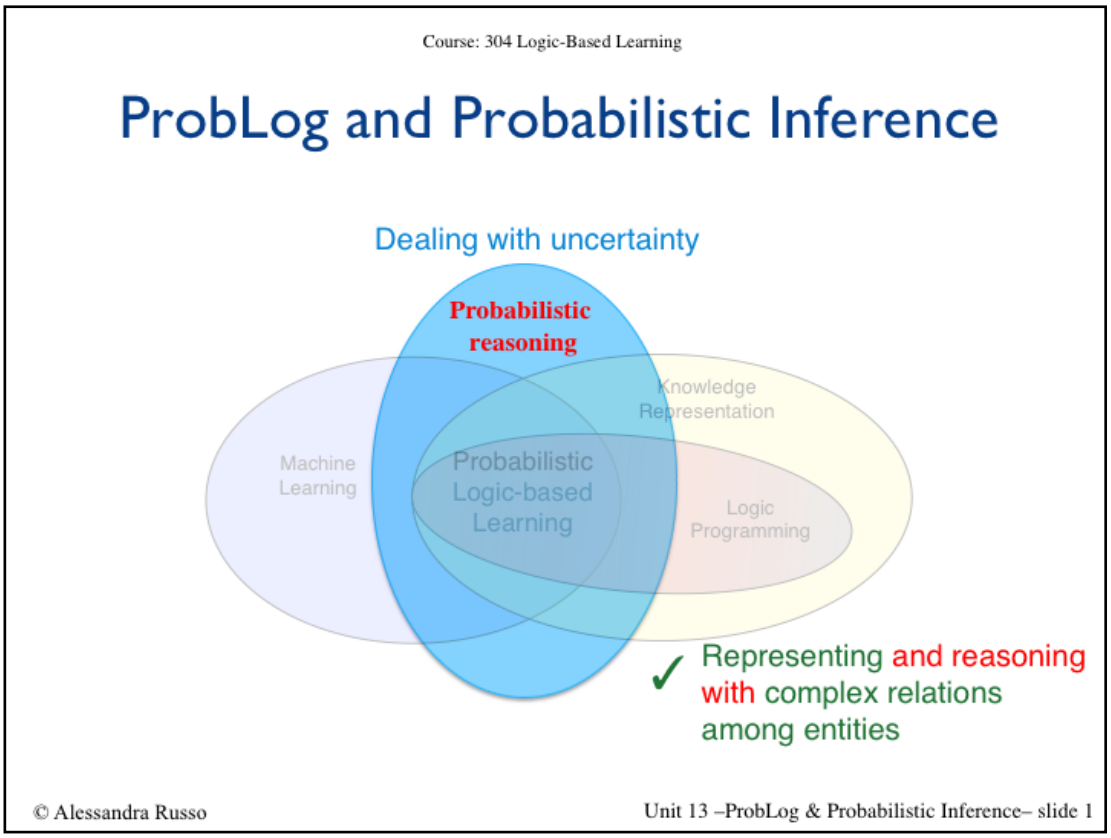


We have introduced in the previous lecture the notion of distribution semantics, defined the notion of possible worlds and probability distribution over possible worlds and saw how these notions can be used to compute the probability of an atom and any arbitrary formula from a given probabilistic logic program. We have primarily concentrated on propositional probabilistic logic programs.

In this lecture we will look more closely at the process of inference (i.e. deduction) from probabilistic logic programs. We will define three different types of inference "**EVID**" for probability of a given evidence, **MARG**, for computation of marginal probability of a query given an evidence and **MPE**, for computation of Most Probable Explanation. We will then see how to compute some of these types of inference where the emphasis is on the efficiency of the computation. We will also see examples of effective ways of computing probability of queries.

Many different probabilistic logic programming languages have been proposed in the literature, some of which are based on distribution semantics and have been proved to be equivalent in their expressivity. The simplest of these is ProbLog. So we introduce some of the basic features of ProbLog, in terms of its syntax.

The material included in this lecture note is extracted from the paper

"ProbLog: A Probabilistic Prolog and its Application in Link Discovery", Luc De Raedt, Angelika Kimmig and Hannu Toivonen, IJCAI 2007.

You can find additional material on the ProbLog Website:

https://dtai.cs.kuleuven.be/problog/

Course: 304 Logic-Based Learning

Success probability of a query

(Recap)

- ✓ Probabilistic logic program defines probability distribution P_F over (non-probabilistic) logic programs.
- ✓ Probabilistic fact defines the probability that the fact belongs to a randomly sampled program. Probabilities are mutually independent.

$$P_F(F') = \prod_{f_i \in F'} p_i \times \prod_{f_i \in F \setminus F'} (1 - p_i)$$

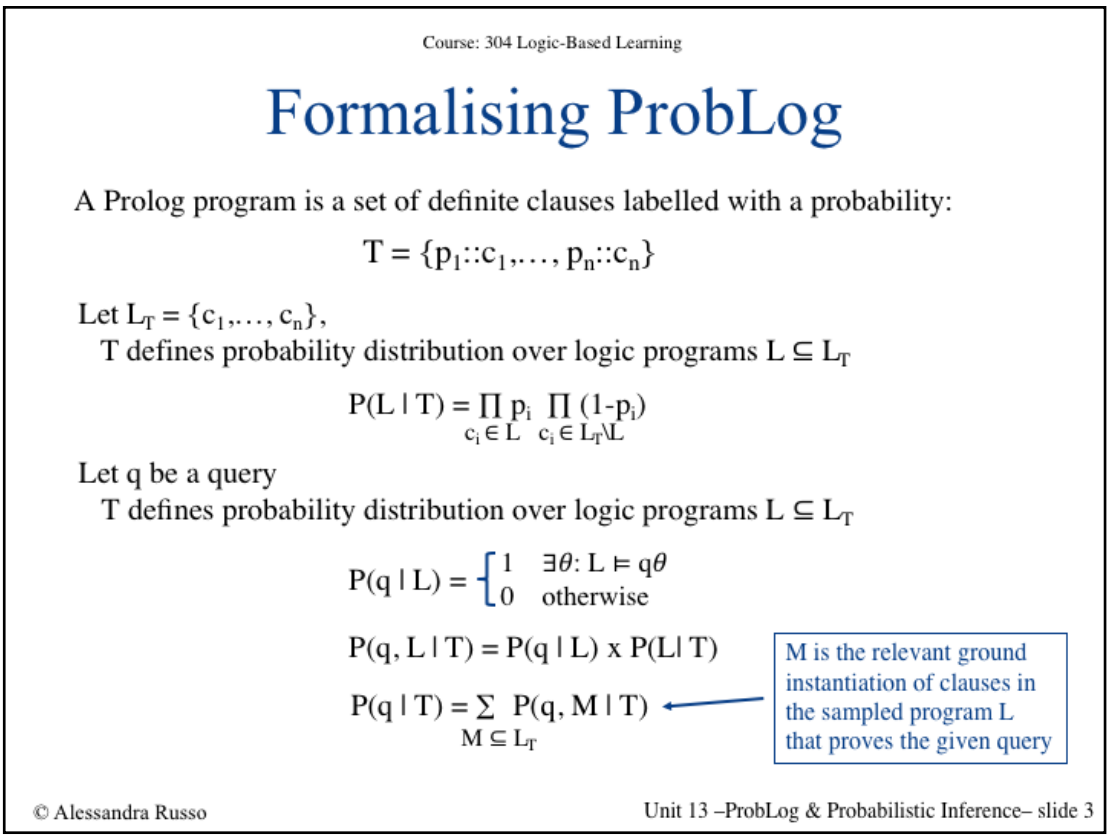
- ✓ **Success probability P(q)** of a query **q** defines the probability that the query succeeds in a randomly sampled logic program.

$$P(q) = \sum_{\substack{F' \subseteq F \\ F' \cup R \models q}} P_F(F')$$

To recap the main points covered in the previous lecture. We have seen that a probabilistic logic program captures essentially a probability distribution over non probabilistic logic programs, by defining for each (probabilistic) fact the probability that it belongs to a randomly sampled logic program. In reformulating the problem in terms of distribution over logic programs (instead of possible worlds), we can focus on the probability distribution over atomic choices (forming F') over the set F of probabilistic facts of a given probabilistic logic program. F' in the formula above is the set of facts chosen to be true in a sampled logic program (corresponding to a sampled possible world). So P(F') is the probability of the possible worlds whose facts chosen to be true are those that appear in F'.

Similarly, we can define the success probability of a query P(q) as the probability that the query will succeed in a randomly sampled logic program (you can read this as randomly sample possible world).

The question now is how do we efficiently compute the success probability of a query?

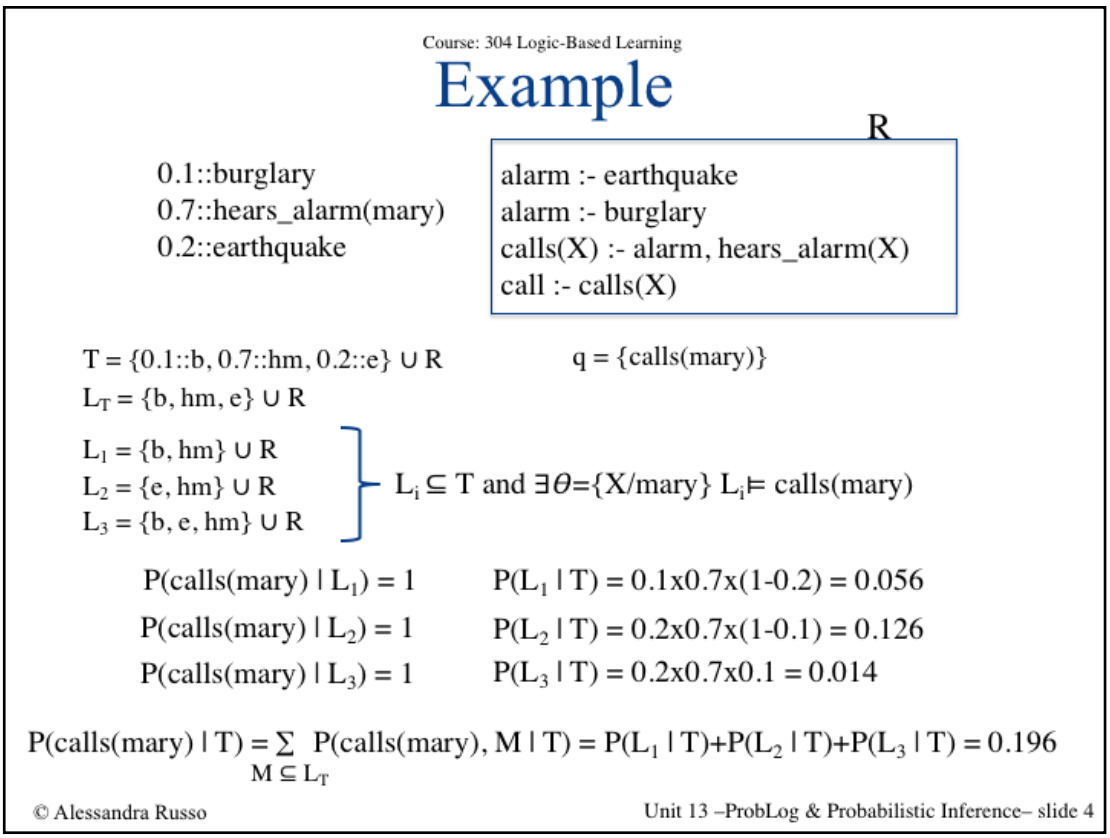


This slide summarises some main definitions for ProbLog. We will consider in this course the subset of ProbLog composed of definite clauses without function symbols, and with no additional Prolog built-in predicates. More extended version can be found on the ProbLog website (https://dtai.cs.kuleuven.be/problog/).

In the general case, a ProbLog program T is a set of definite clauses (possibly) labelled with a probability. We use here the term clauses to refer also to ground facts (i.e. a clause with no body literals). We denote with L_T the set of all clauses in T without the probability label. A ProbLog program T defines essentially a probability distribution over all logic programs $L \subseteq L_T$ that can be constructed from L_T by sampling the clauses that have a probability label. Normally, a ProbLog program can include also clauses with no probability label. These are assumed to be always part of any sampled program L.

We can define the probability of a given sampled program L as the product of the probability of the clauses that have been sampled in L, and 1 minus the probability of the clauses in T that have not been sampled in L. Note that the clauses in L are unground, so in practice we do not compute the probability of the sampled unground program, **but the probability of the instantiation of the sampled clauses that are relevant to compute the success probability of a given query.** We will see later what is meant by "clauses that are relevant to the given query". So whereas in Prolog we are normally interested to determine whether a query succeeds or fails, in ProbLog we are typically interested in computing the probability that a query succeeds.

So a given query can be proved from many of the sampled programs $L \subseteq L_T$ and, for each sample program, there will be a unification θ that determines essentially a relevant ground instantiation of the clauses in the sampled program L that proves the given query. In the equation P(q | T) we denote such relevant ground instantiation of L with M.



We consider our usual example and instantiate the equations given in the previous slide. In the example above only facts are probabilistic, whereas rules are not. T is the set of all clauses with respective probabilities, whereas L_T is the set of all clauses in T but without the labelled probability. Note that any non probabilistic rule is assumed to be part of any subset of L_T.

The set of all $L \subseteq L_T$ includes many logic programs but not all of them entail the given query calls(mary). So not all of them are relevant to the entailment of the given query *calls(mary)*. In this slide we give just the three logic programs L_1, L_2 and L_3 that entail *calls(mary)*. The P(*calls(mary)*|L_i)=0 for every $L_i \subseteq L_T$ different from L_1, L_2 and L_3, whereas P(*calls(mary)*|L_1)=1, P(*calls(mary)*|L_2)=1 and P(*calls(mary)*|L_3)=1. The instantiations of these three logic programs that are relevant to the query *calls(mary)* are the same as the L_1, L_2 and L_3 since the probability labels are only assigned to ground facts in T. In the next few slides we will see that probability labels can be assigned to also unground clauses in a given program.

Hence, $M_1 = L_1$, $M_2 = L_2$ and $M_3 = L_3$.

As next step we compute the P(L_i|T) but given that P(*calls(mary)*|L_i)=1 for only L_1, L_2 and L_3, we need to compute only P(L_1|T), P(L_2|T) and P(L_3|T), as shown in the slide.

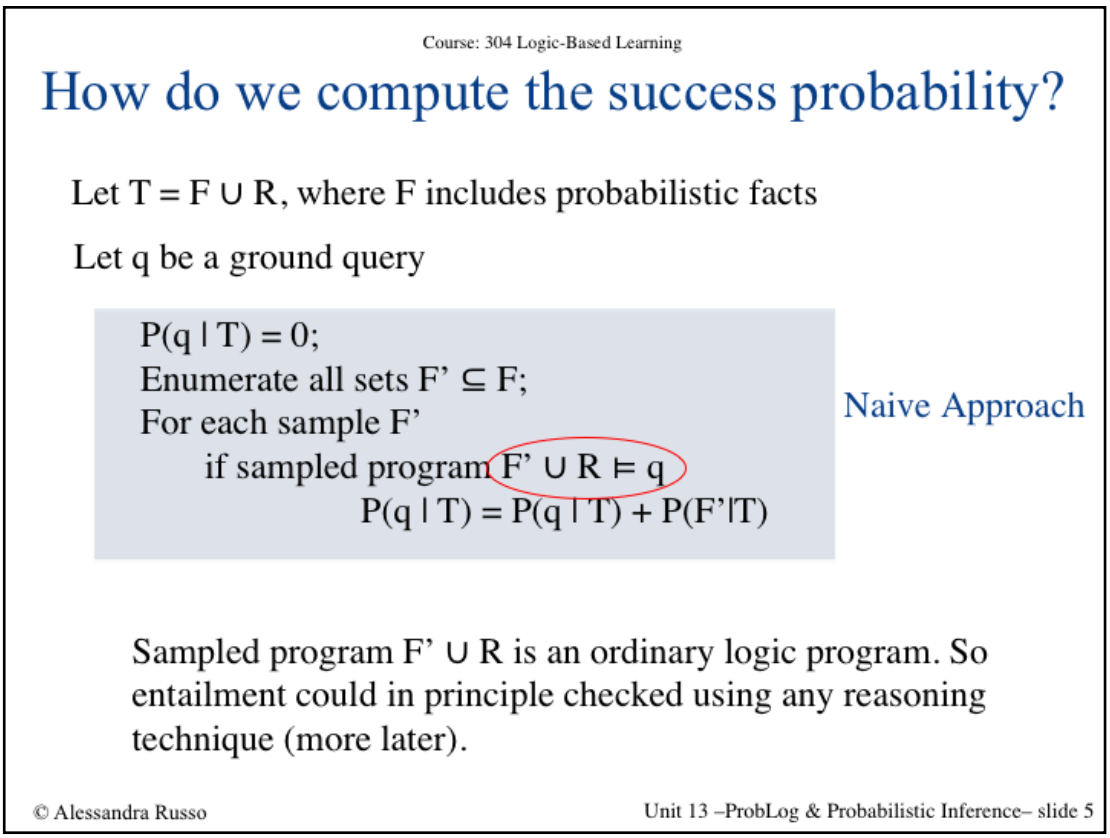
We are now in the position to calculate the probability of the query *calls(mary)* given the program T.

The instantiation of the

$$P(calls(mary) \mid T) = \sum_{L_i \subseteq LT} P(calls(mary), L_i \mid T) = \sum_{L_i \subseteq LT} P(calls(mary) \mid L_i)P(L_i \mid T) =$$

$$= P(L1 \mid T) + P(L2 \mid T) + P(L3 \mid T) = 0.196$$

Note that in the example above each P(L_i|T) includes also 1- the probability of the probabilistic atoms that are not used in the entailment of the query. This is to guarantee that the alternative ways of entailing a given query are **pairwise mutually exclusive**.

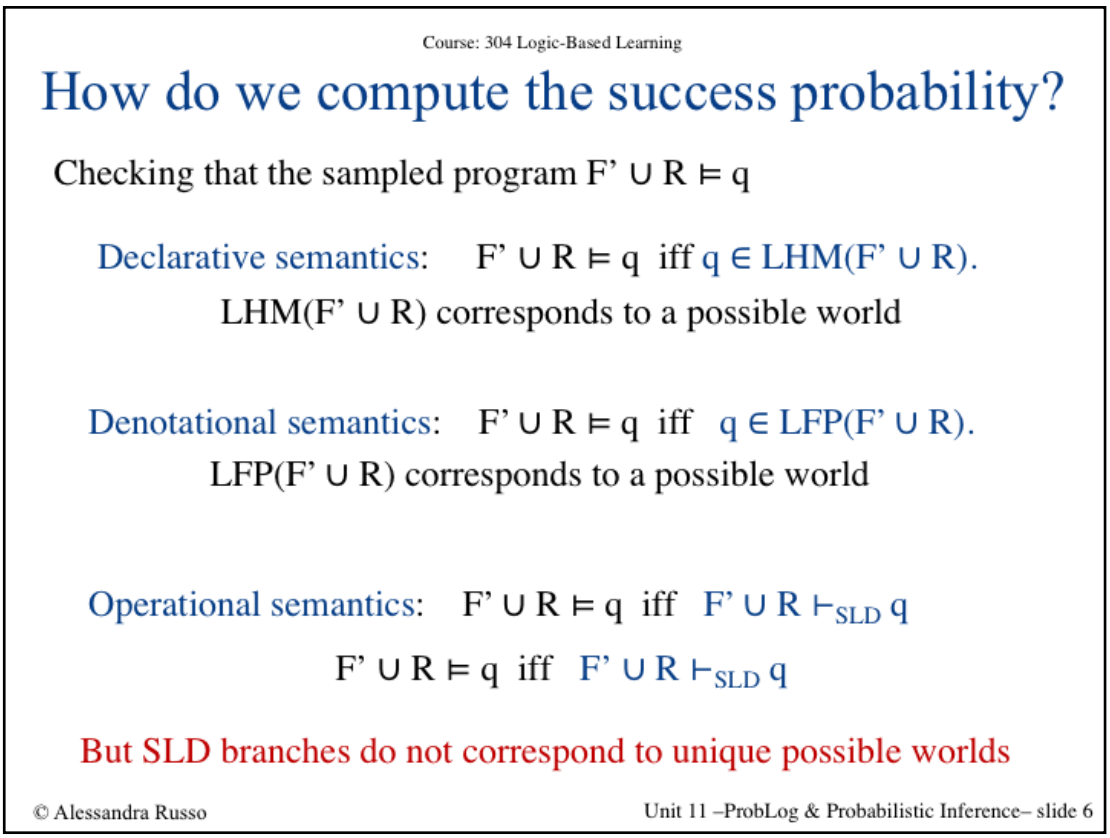


Given that we have a distribution over logic programs, how do we calculate the success probability of a given query? Let's assume a probabilistic logic program $T = F \cup R$, and let F be the set of probabilistic facts in T. Let q be a given ground query. L_T is the program identical to T but with no labelled probabilities.

A method could be to enumerate first all possible samples F' of facts from the given F but without probability (each $F' \subseteq L_F$). Each subset F' defines a non-probabilistic logic program $L = F' \cup R$, whose facts are atomic facts from F chosen to be true in F'.

So we can check for each sample F', if the query is entailed by the corresponding (non-probabilistic) logic program L. If so, compute the probability P(F'| T) since facts in F' are independent. This probability can be computed as indicated in slide 3 or, equivalently, as we have seen in the previous unit by computing the probability of the possible world that corresponds to F', and adding this probability to the current probability P(q | T).

The key point is checking the entailment of the query from the sample logic program.



Let's see how we can compute an answer for F' ∪ R ⊨ q.

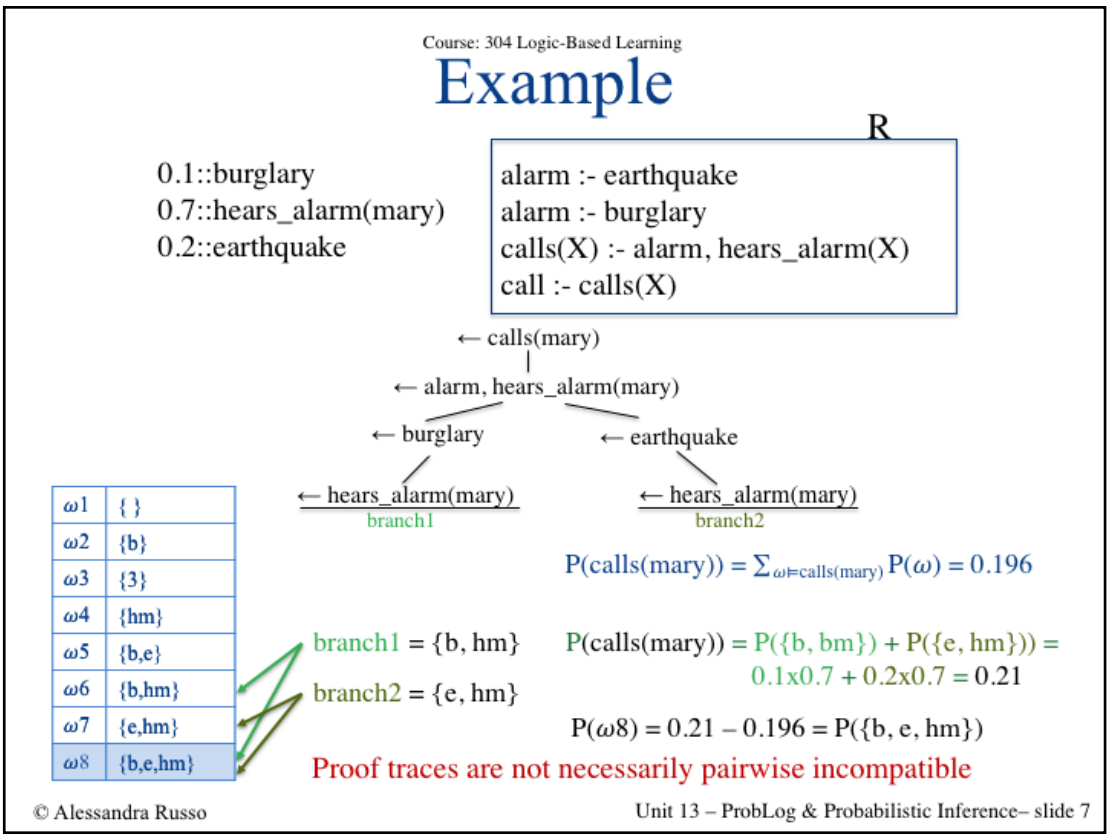
We can use a pure **model-theoretic approach**. From a pure model-theoretical semantics,

F' ∪ R ⊨ q if and only if q is true in every model of the logic program F' ∪ R

So q has to be true in the intersection of all models of F' ∪ R. Our assumption so far has always been that our sampled programs F' ∪ R are definite logic programs. We have seen at the beginning of our course that models of a definite program are Herbrand models. the intersection of all Herbrand models is called the **Least Herbrand model (LHM)** and that the definite logic program has a unique LHM. So to check if F' ∪ R ⊨ q it is sufficient to check that q is true in the LHM(F' ∪ R). We have also seen in the previous lecture that given a sample F' of (probabilistic) facts, the resulting possible world corresponds to the unique LHM(F' ∪ R). So the probability of q being true in the LHM(F' ∪ R) is given by P(F').

Denotational semantics is a second method for checking that F' ∪ R ⊨ q. In this way the checking is done through **forward reasoning** by means of applications of the T_P-operator. The meaning of a sampled logic program is then given by the Least Fixed-Point of the T_P-operator. Similarly to the declarative semantics, each random choice of facts F' gives rise to only one Least Fixed-Point execution trace, that starts from the set F'. So to check whether F' ∪ R⊨ q reduces to check whether q belongs to the Least Fixed-Point of the sampled logic program F' ∪ R . We can compute the fixed-point of the sampled program and see if the query belongs to it. The similarity between the declarative and denotation semantics is not surprising since for definite programs the fixed-point operation is proved to compute the LHM of the program.

A third method is the operational semantic approach. This is based on SLD resolution and **backward chaining**. An SLD resolution can be constructed starting from the query that we want to check for a given sample program and see the existence of a proof, or SLD branch, that succeeds. But programs can have multiple rules with the same head. **In this case an SLD branch may correspond to multiple possible worlds.** So the computation of the probability of a query based on just successful branches needs to be defined in a different way to avoid counting overlapping worlds twice. We give an example of this problem in the next slide.



Let's revisit the example we saw in slide 4. Let's label the set of random choices of facts as:

$F_{\omega 1}=\emptyset$, $F_{\omega 2}=\{b\}$, $F_{\omega 3}=\{e\}$, $F_{\omega 4}=\{hm\}$, $F_{\omega 5}=\{b, e\}$, $F_{\omega 6}=\{b, hm\}$, $F_{\omega 7}=\{e, hm\}$ and $F_{\omega 8}=\{b, e, hm\}$.

We have seen in slide 20 of Unit 12 that the probability of the query "calls(mary)", in terms of possible worlds, is given by:

$P(calls(mary)) = \sum_{\omega \models calls(mary)} P(\omega) = P(\omega 6) + P(\omega 7) + P(\omega 8) = 0.1x0.7x0.8 + 0.9x0.7x0.2 + 0.1x0.7x0.2 = 0.056+0.126+0.014 = \mathbf{0.196}$.

In terms of the sampled (non-probabilistic) logic programs the same probability is computed as

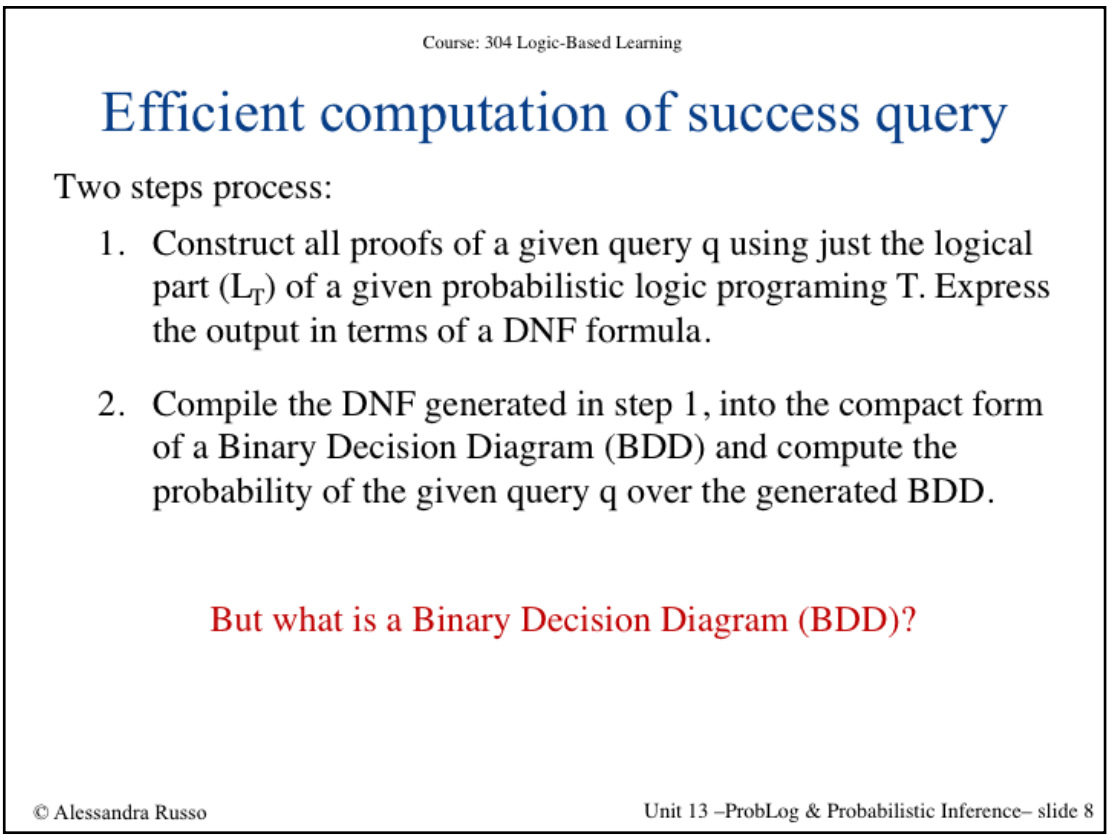
$P(calls(mary) \mid T) = \sum_{M \subseteq L_T} P(calls(mary), M \mid T) = P(L_1 \mid T)+P(L_2 \mid T)+P(L_3 \mid T) = \mathbf{0.196}$.

Let's now consider a pure proof theoretic approach. That is the SLD proof tree of the same query.

As shown in this slide we have two successful branches. If we consider the proof trace given by each of these two successful branches, we notice that the probabilistic facts that are included are {b, hm} and {e, hm}. These two branches are **semantically not mutual exclusive** as they can both be true in the third possible world {b, e, hm} that we have identified above. Specifically, the first branch {b, hm} is a compact representation for worlds $\omega 6$ and $\omega 8$ and the branch {e, hm} is a compact representation for worlds $\omega 7$ and $\omega 8$. So branches do not directly correspond to the possible worlds. Therefore, we can't calculate the probability of the query q in terms of successful branches of the proof for the query by simply multiplying the probability of the facts that appear in a success branch to compute the probability of the branch and then summing up the probabilities of all success branches. If we were going to do so we would get

$P(calls(mary)) = 0.1x0.7 + 0.2x0.7 = 0.07+0.14=\mathbf{0.21}$.

If we calculate the $P(\omega 8)$, we get $P(\omega 8) = 0.014$, which is exactly equal to the difference between the two probabilities for query calls(mary): i.e. $0.21 – 0.196 = 0.014$. This is also known as the **disjoint-sum-problem**. So, computing proof theoretically the probability of a query, given a probabilistic logic program, needs to take into account the fact that proof traces are not necessarily **pairwise incompatibles** and therefore the probability cannot be computed just as a summation over the proof traces.



Clearly generating all possible sampled logic programs, as described in the naïve algorithm is not a feasible task. So, how can we compute efficiently and proof theoretically the probability of a success query whilst avoiding the problem of pairwise incompatibility?

One of the inference approaches adopted by ProbLog uses the concept of BDD (Binary Decision Diagrams) as mechanisms for compiling the knowledge needed to prove a given query in a compact way. The approach described here has been published in

De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In Veloso, M.M., ed.: IJCAI. (2007) 2462–2467

The method consists of two steps:

Firstly, all proofs (i.e. a proof tree) of a given query q are constructed using just the logical part of a given probabilistic logic programing T, that is using only the L_T . The result of this step generates a DNF formula, over the probabilistic explanations used in the proof tree, where each disjunct corresponds to a successful branch of the proof tree. Each disjunct is a conjunction of ground instances of the probabilistic clauses (or facts) used in the branch derivation.

The second step uses a compilation of the knowledge expressed in the DNF formula into BDD in order to compute the probability of this formula in a more efficient way. This is important because computing the probability of a DNF formula is in general an NP-hard problem even when the disjunct formulae are independent.

Let's first see what BDDs are.

Course: 304 Logic-Based Learning

Introducing BDDs

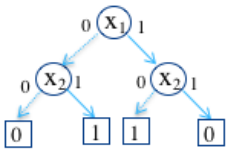
A binary decision diagram is an efficient graphical representation of a Boolean function over a set of variables.

Widely used in computer architecture and verification.

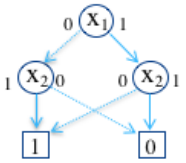
Can be seen as a variant of a Boolean decision tree (BDT).

$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

Boolean decision tree



Binary Decision Diagram



- Fixed variable ordering on all paths
- Shared use of identical subtrees
- Nodes with identical children are left out

What are the BDT and BDD of $x_1 \wedge x_2$?

© Alessandra Russo

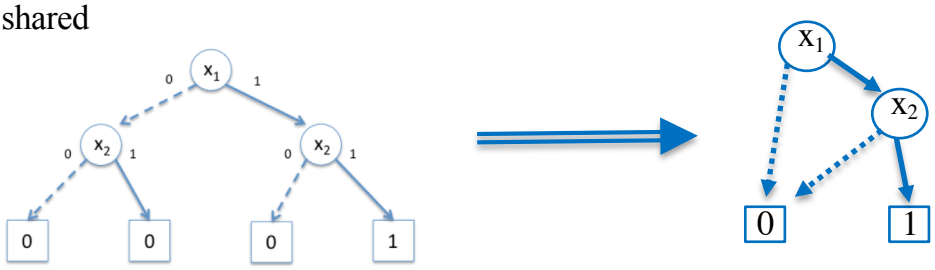
Unit 13 –ProbLog & Probabilistic Inference– slide 9

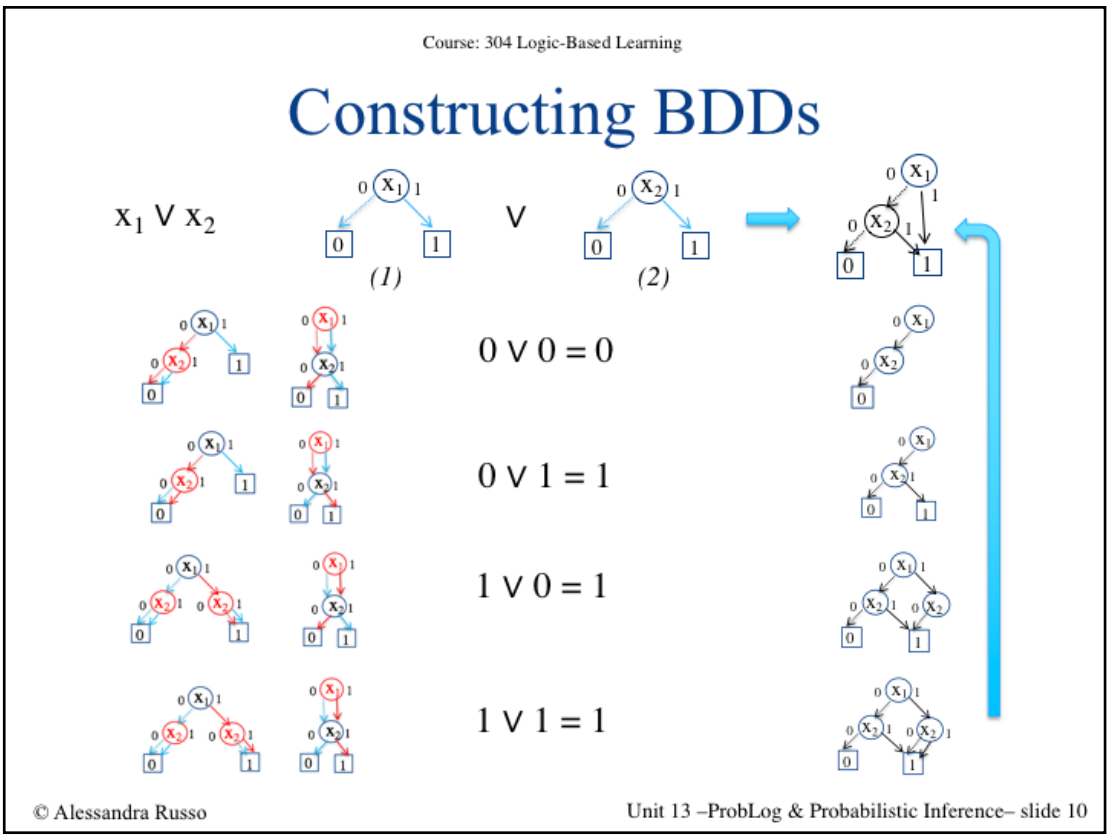
A Binary Decision Diagram is an efficient graphical representation of Boolean functions over a set of variables. These are functions that take Boolean values for a given set of variables and produce a Boolean output 1 or 0.

A Binary Decision Diagram can be seen as a variant of a Boolean Decision Tree. A binary decision tree takes in input a set of Boolean variables $x_1, \ldots, x_n$. At the root node one of the variables is tested, say, x_1. We get two subtrees, one for the case where $x_1 = 0$ and one where $x_1 = 1$. Each of the two subtrees is now testing another variable, each with another two subtrees, and so on. At the leaves we get either 0 or 1, which is the output of the function on the inputs, that constitute the path from the root to the leaf. In the slide above the dashed arrow correspond to the case where the tested variable is false and the full arrow corresponds to the case where the tested variable is true. Binary decisions trees have the **canonicity property:** if we test variables in a fixed order $x_1, \ldots, x_n$, then the binary decision tree is unique. If the order of the variables we test is fixed, we say that the decision tree is ordered. The negative property of a binary decision tree is the size: a binary decision tree of n variables will have $2^n - 1$ decision nodes, plus 2^n links at the lowest level, pointing to the return values 0 and 1.

We would like to make the representation of Boolean functions more compact than binary decision trees while preserving the canonicity property that for a given fixed ordering of the testing we have a unique compact representation. A BDD provides a solution to this problem.

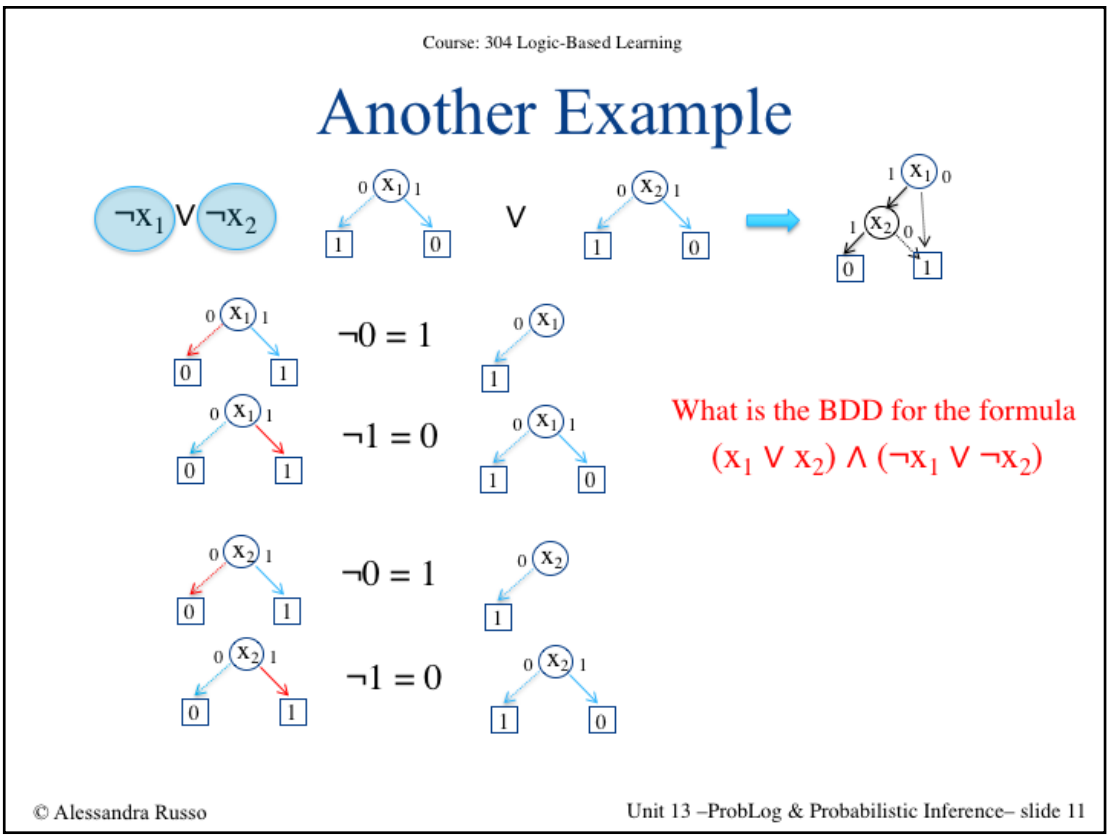
A BDD differs from a Boolean decision tree in two ways. Firstly, they allow redundant test of Boolean variables to be omitted. If a test of a variable leads to the same outcome there is no need to test for that variable. For instance, in the Boolean Decision Tree for $x_1 \wedge x_2$ the test of x2 on the false case of x_1 is not necessary as it leads to 0 output for both truth values of x_2. Secondly, a BDD allows identical subtrees to be shared





Constructing the BDD for a given Boolean formula requires applying the main Boolean operation to the BDD(s) that correspond to the respective sub-formulae. In this slide the operation is V and the two sub-formulae are the two Boolean variables x_1 and x_2. The BDDs for the Boolean variables are the simple diagrams *(1)* and *(2)* above. To compose them together we start from the top and work through both BDD in parallel. Where a test for a variable does not appear in one of the BDD we imagine as if the variable test exists and both the 0 and 1 cases lead to the same output of the BDD.
For instance, in (1) there is no test for x_2 variable, so we can imagine a test on each of the 0 and 1 edges for x_2. The outgoing edges 0 and 1 from x_2 lead both to 0 and both to 1 outcome of the BDD.
For the BDD (2) there is no test for x_1, so we can imagine a node x_1 above x_2 (as we need to preserve the order over the variables that we test) with edges 0 and 1 leading both to node x_2.We then follow in parallel each BDD from the root to the leaf output. When we get to the output in each respective BDD we then apply the logical operator and create a new BDD for the sequence of test done.
In the given two BDD, we start from x_1 and we follow the 0-successor in both BDDs for x_1. In the left BDD we imagine an x_2 test that leads to 0, for whatever value x_2 we consider. In the right BDD we imagine an x_1 test that leads to x_2. But for x_2 the BDD lead to different output. So we assume the case of x_2 test be zero. Both BDDs lead to zero, $0 \vee 0 = 0$, and we start constructing a first part of the resulting BDD. We now consider the edge 0 from x1 and the edge 1 from x_2 The left BDD goes to 0 and the right BDD goes to 1. So $0 \vee 1 = 1$. So we add an edge from x_2 to 1 in the new BDD. We now consider the x_1 and the edge 1 from it and the edge 0 from the added node x_2. The left BDD gives output 1. The right BDD gives in output 0 for x_2 test equal to 0. So $1 \vee 0 = 1$. Finally, we consider edge 1 x_1 and edge 1 from x_2. The left BDD gives 1 as output. The right BDD gives in output 1 as well. So $1 \vee 1 = 1$. We can complete now the application of the operation V to the given initial BDDs.

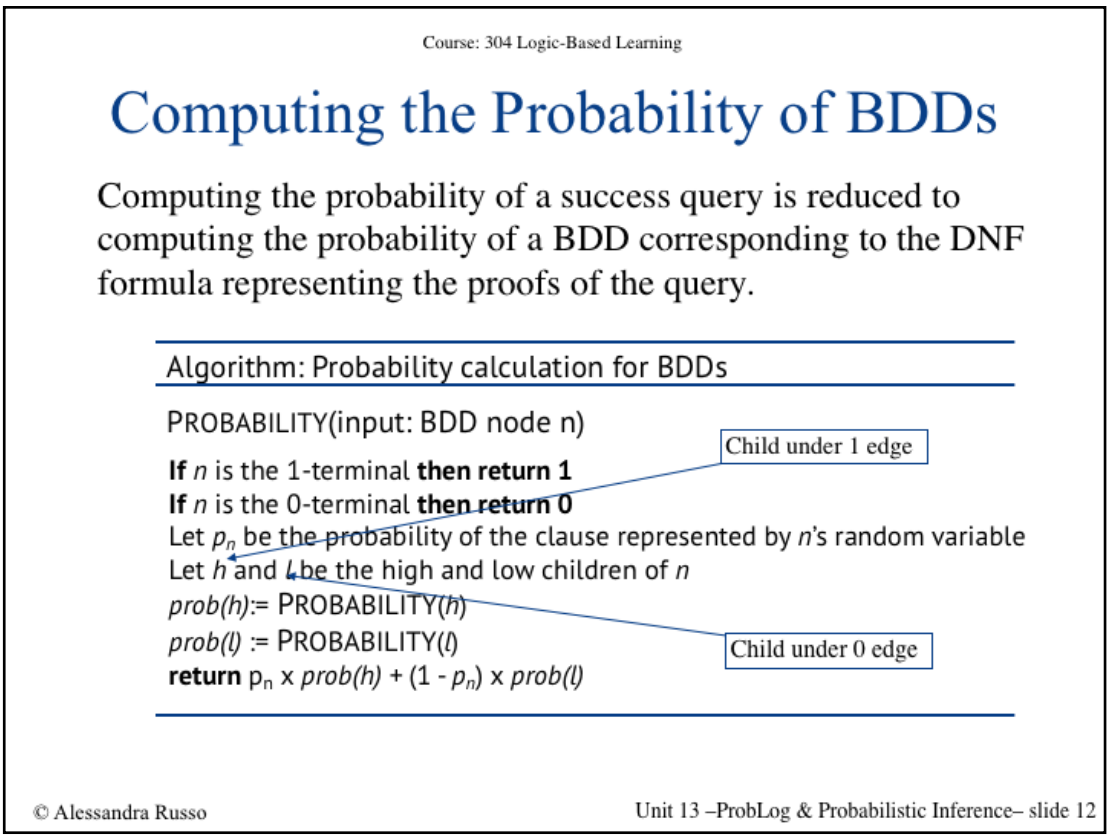
We can notice now that the generated BDD (bottom right diagram) has two edges coming out from the test x_1 and both ending up into the same output (1). This means that the test of x_1 is unnecessary in this branch We can therefore eliminate this node in the final BDD and link the test x1 directly to 1 from the edge labelled 1. This gives us the final BDD in the top right corner of the slide.



This slide provides another example. We want to construct the BDD for the formula $\neg x_1 \vee \neg x_2$. This formula includes two operations: $\neg$ and $\vee$. So we need first to construct the BDDs for the two subformulae $\neg x_1$ and $\neg x_2$, as shown in this slide and then compose together the resulting BDDs with the $\vee$ operation.

The BDDs that correspond to $\neg x_1$ and $\neg x_2$ are given above at the top of the slide. We can now apply the same process illustrated in the previous slide on these two BDDs to construct the BDD for formula $\neg x_1 \vee \neg x_2$. The resulting BDD is the one on the top right part of the slide.

Can we then construct the BDD for the complex formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$? This is included in the the tutorial.



We can now go back to the efficient computation of probability of a success query. In slide 8, we have stated that ProbLog uses a two steps algorithm:

Firstly, all proofs of the given query q are constructed using just the logical part (L_T) of a given probabilistic logic programing T. The proofs are expressed in terms of a DNF formula, where each disjunct corresponds to a proof and it is defined as the conjunction of the probabilistic facts that are used in the proof.

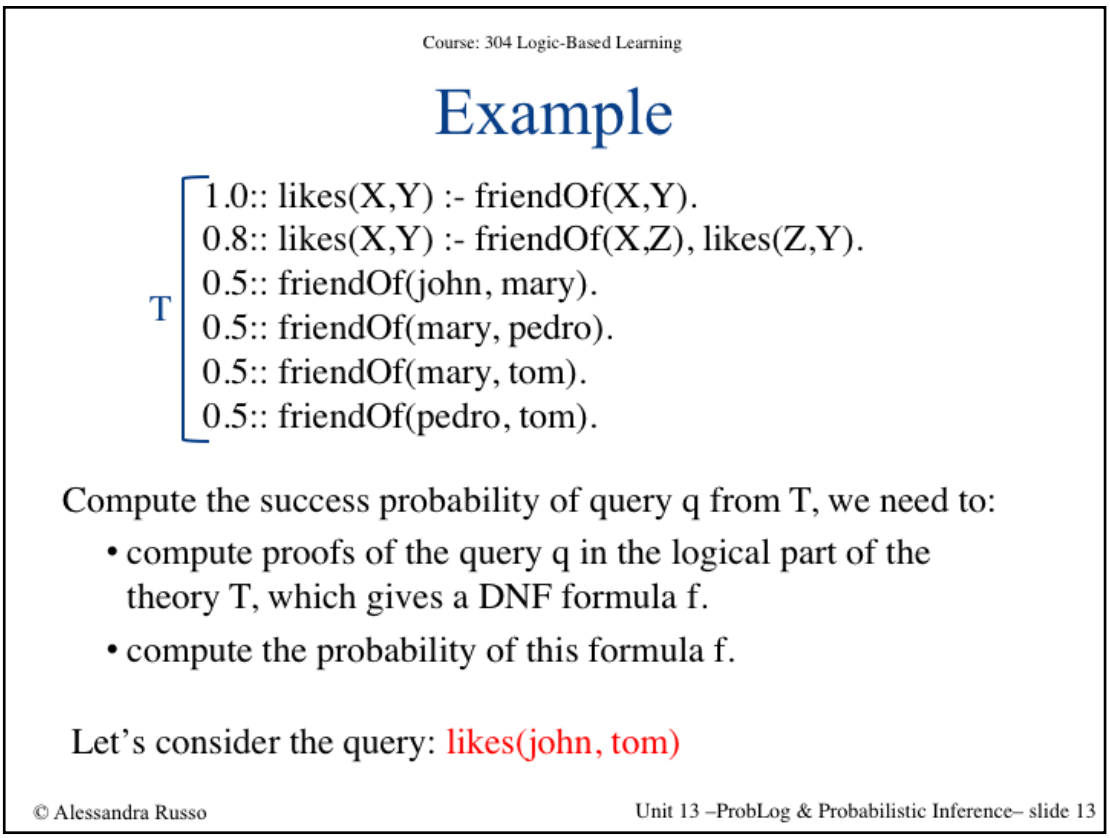
Secondly, the DNF generated in the first step, is compiled into a Binary Decision Diagram (BDD). The probability of the success query is then given by computing the probability of the generated BDD.

It is worth noticing that each branch in the BDD from the root node to the leaf (1 or 0) represents a complete variable assignment for the probabilistic variables. If variable x is assigned 0 (or 1), the branch to the "low" (or "high") child is taken. In BDD terms the "low" child is the child node following the 0 assignment to the parent node variable and the "high" child is the child node following the 1 assignment to the parent node variable. Each leaf is labeled by the outcome over the DNF formula, that can be seen as the outcome of the (probabilistic) Boolean function applied to the (probabilistic) Boolean variables. given the variable assignment represented by the corresponding path.

Given a BDD, it is easy to compute the probability of the corresponding Boolean function recursively from root node to leaf as described in the algorithm above.

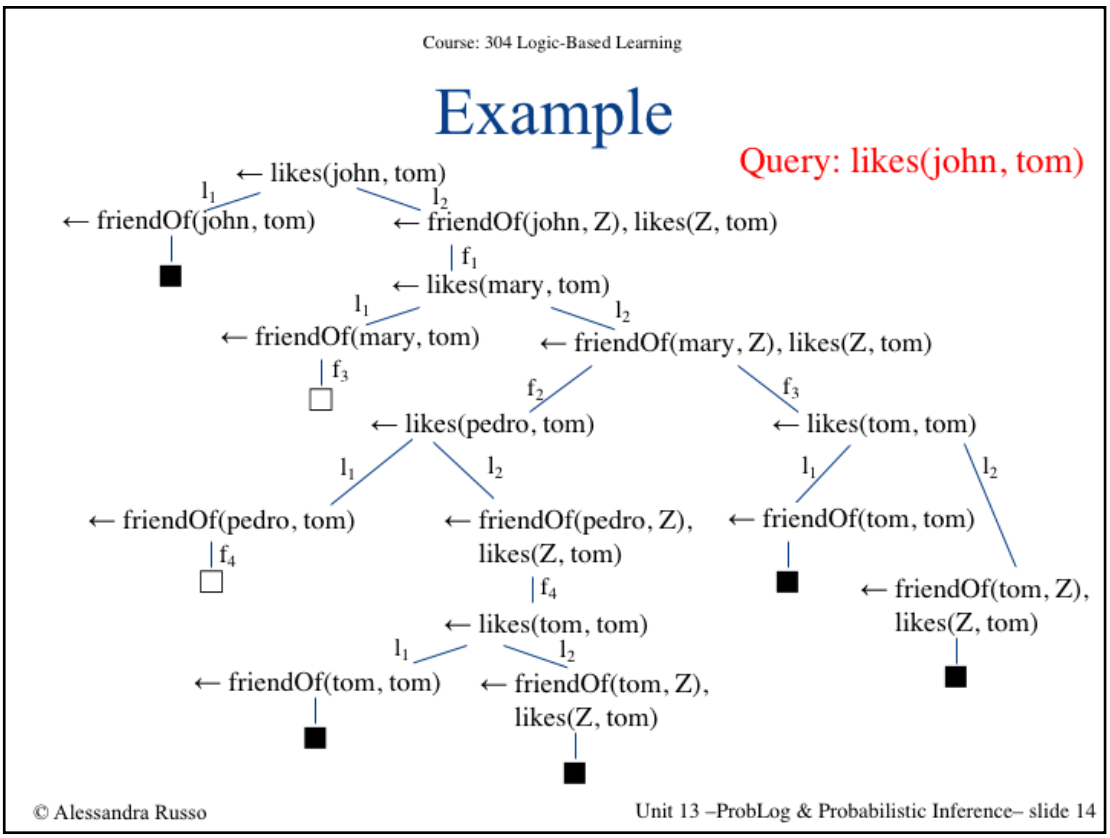
An example is given in the next slides, which is taken from the first publication on ProbLog1:

"ProbLog: A Probabilistic Prolog and its Application in Link Discovery", Luc De Raedt, Angelika Kimmig and Hannu Toivonen, IJCAI 2017.



Consider the above example of a probabilistic logic program. Note that in this case, also rules are labelled with a probability. We can interpret this probability as the probability of ground instance of the head atom of the clause. We have stated that to compute the success probability of a given query, we need to compute the proofs of the query q using the logical part of the theory T (that is L_T) . Its result will be a DNF formula f. We can then compute the probability of this formula f using its BDD representation.

Let's consider the query "**likes(john, tom)**".



If we consider the query "likes(john, tom)" and the probabilistic logic program T given in the previous slide, the ProbLog algorithm will first compute the SLD derivations for this query. This slide shows the different SLD proofs for the given query. Specifically, there are two successful derivations. Each branch is labelled with the specific clause or fact that has been used in the SLD resolution step. We have used the labels l1 and l2 for the derivation of the head of the first and second clause respectively, and f1 to f4 for the derivation of the facts labelled in the order in which they appear in the program.

Course: 304 Logic-Based Learning

Example

Successful derivations of likes(john, tom) are two sets of clauses:

$\{0.8::l_2, 0.5::f_1, 1.0::l_1, 0.5::f_3\}$ branch (br_1)

$\{0.8::l_2, 0.5::f_1, 0.8::l_2, 0.5::f_2, 1.0::l_1, 0.5::f_4\}$ branch (br_2)

Boolean random variable b_i indicates clause $p_i::c_i$ is in the proof, b_i has probability p_i of being true.

$P(q \mid br_i) = P(\bigwedge_{b_i \in cl(br_i)} b_i)$ Probability of query, given a successful proof

$P(q \mid T) = P(\bigvee_{br_i \in pr(q)} \bigwedge_{b_i \in cl(br_i)} b_i)$ Probability of success query, where $pr(q) = \{br_1, br_2\}$, $cl(br_i)$ is the set of clauses in br_i

$P(likes(john, tom) \mid T) = P((l_1 \wedge l_2 \wedge f_1 \wedge f_3) \vee (l_1 \wedge l_2 \wedge f_1 \wedge f_2 \wedge f_4))$

$= P((l_2 \wedge f_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2 \wedge f_4))$

We need to generate the DNF formula that corresponds to all proofs of the given query q from the probabilistic logic program T. The SLD derivation in the previous slide shows two successful proofs. Each successful proof has a set of clauses {p1::d1,…,pk::dk} ⊆ T. In our example, we have two sets of clauses:

$br_1 = \{0.8::l_2, 0.5::f_1, 1.0::l_1, 0.5::f_3\}$ and $br_2 = \{0.8::l_2, 0.5::f_1, 0.8::l_2, 0.5::f_2, 1.0::l_1, 0.5::f_4\}$.

We indicate with pr(q) the set of successful proofs of query q, $pr(q) = \{br_1, br_2\}$.

We can denote with b_i a Boolean random variable for each clause $p_i::c_i$ in T. The variable b_i has value true when the clause $p_i::c_i$ is in a successful proof, and we say that b_i has probability p_i of being true.

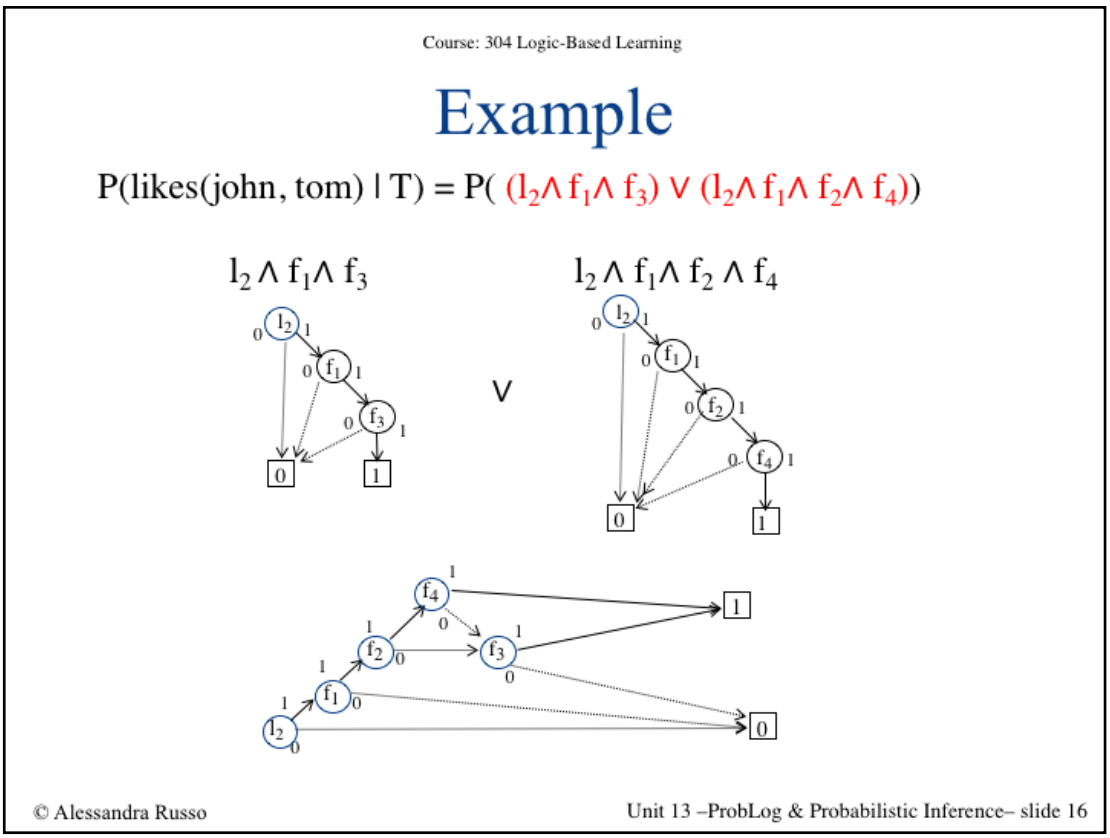
So, the probability of a particular successful proof involving clauses{p1::d1,…,pk::dk}⊆ T is given by the probability of the conjunctive formula $b_1 \wedge \ldots \wedge b_k$. Since a query can have multiple proofs, the probability that a query q succeeds equals the probability that the disjunction of these conjunctions is true. Thus the problem of computing the success probability of a ProbLog query can be reduced to that of computing the probability of a DNF formula.

In our example the probability of likes(john, tom) is reduced to the probability of the DNF formula:

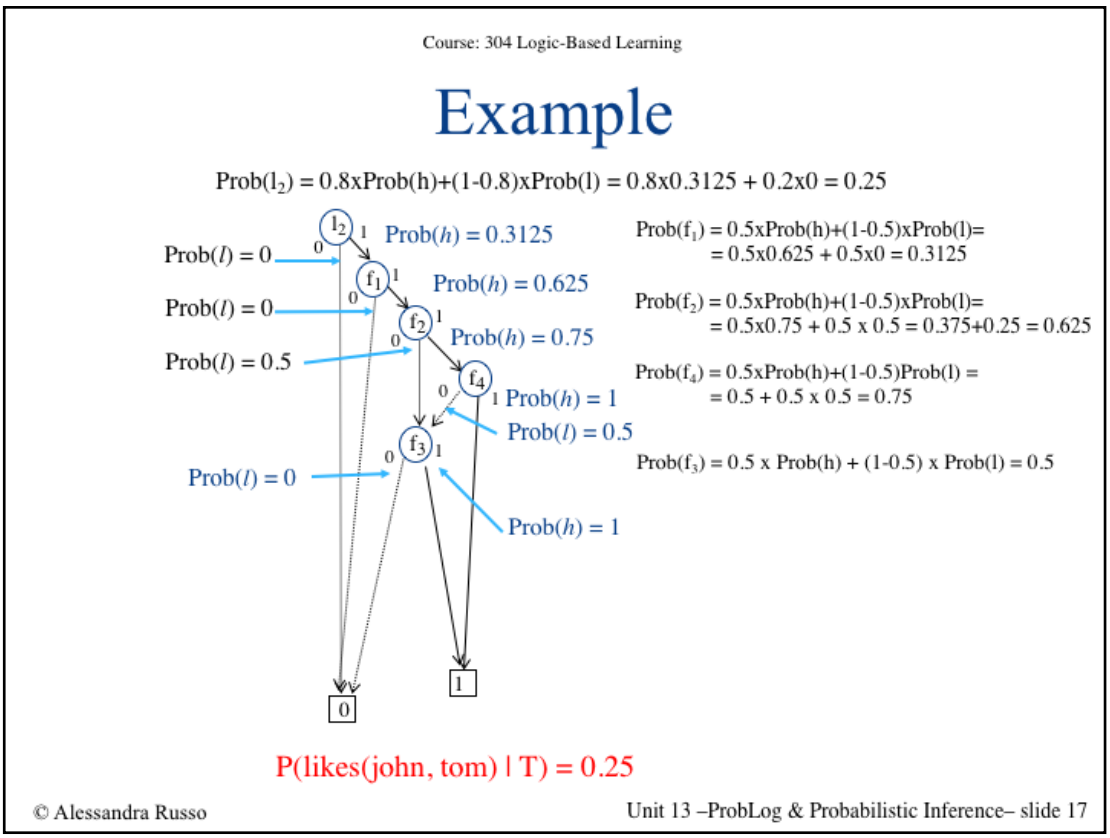
$$P(likes(john, tom) \mid T) = P((l_2 \wedge f_1 \wedge l_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2 \wedge l_1 \wedge f_4))$$

Since the probability of l_1 is equal to 1, we can simplify the formula above and have that

$$P(likes(john, tom) \mid T) = P((l_2 \wedge f_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2 \wedge f_4))$$

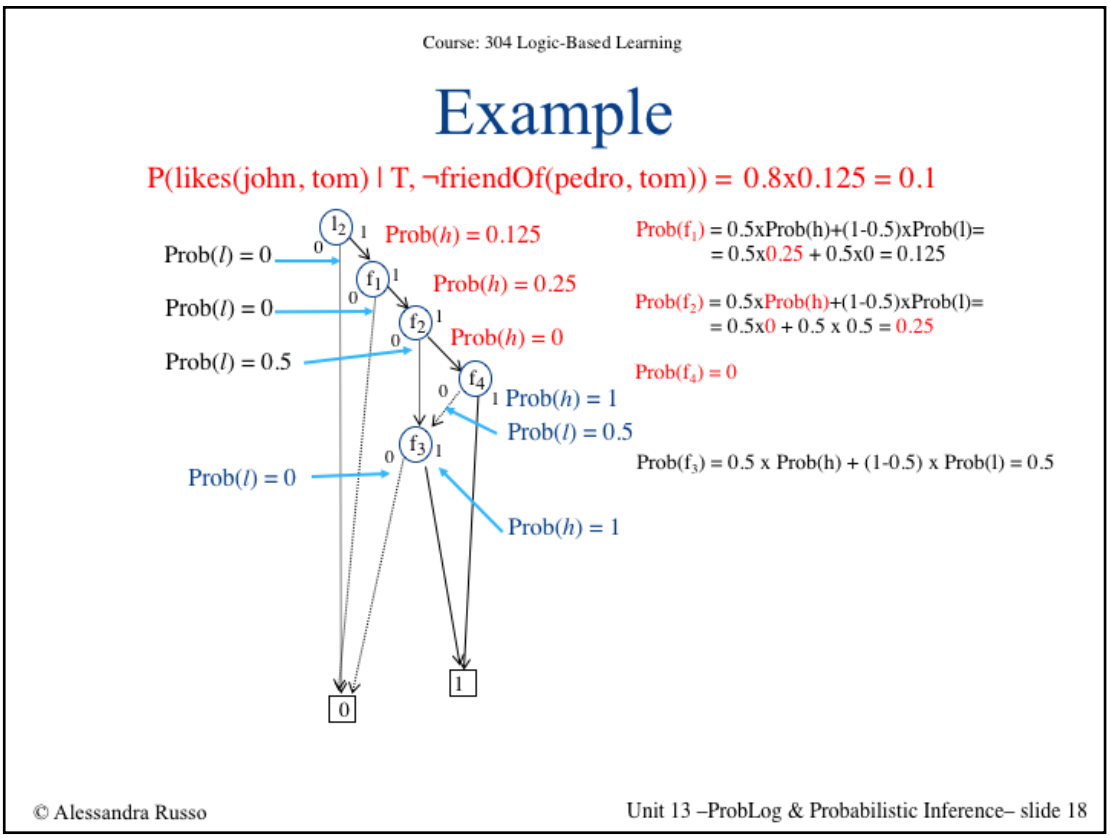


We can now construct the two BDDs for the two proof formulae. These are two conjunctive formulae. The full DNF formula for the success query is given by the disjunction of these two sub-formulae. Applying the algorithm informally described in slide12 the ProbLog generates the BDD given here at the bottom of the slide.



If you execute the same example in ProbLog online editor (https://dtai.cs.kuleuven.be/problog/editor.html) you will notice that the probability of the same query is equal to 0.24 instead of 0.25. You would get the same result as in the online editor if you round down to two decimal figures the probabilities you calculate at each node.

One advantage of this method is that once the BDD is constructed it is possible to calculate the probability of a query given an evidence.

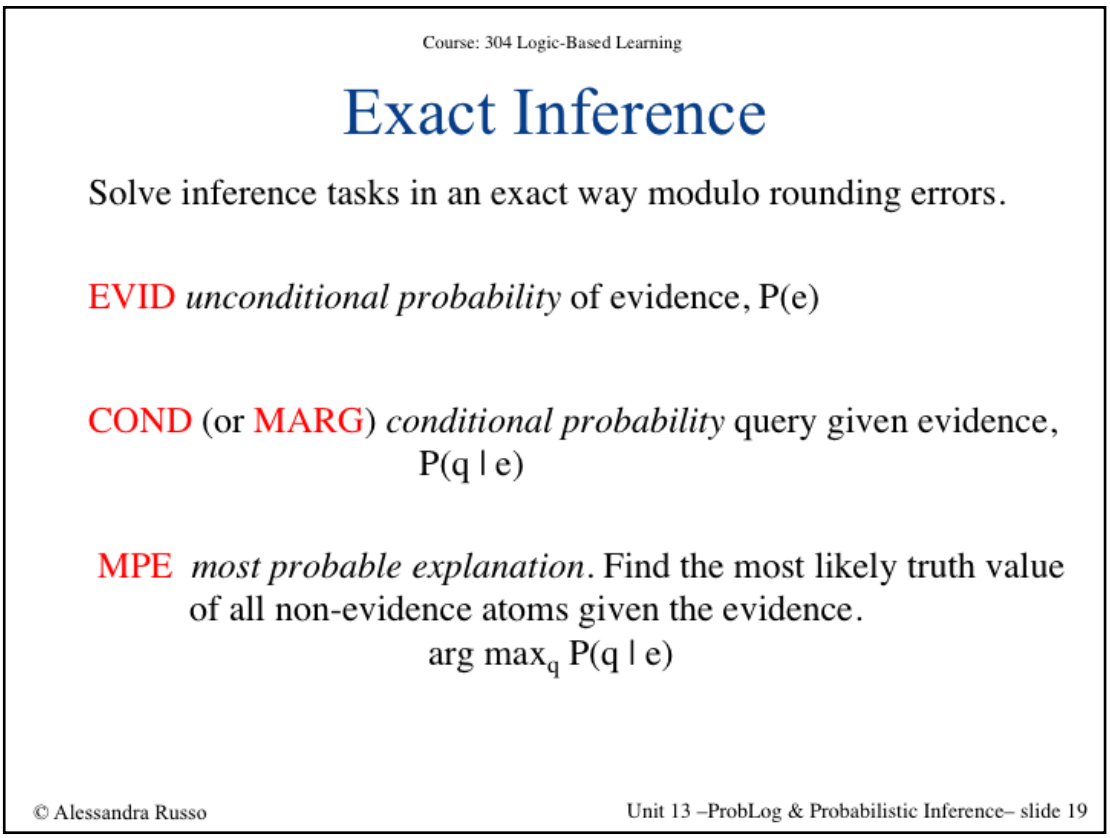


One advantage of this method is that once the BDD is constructed it is possible to calculate the probability of a query given some evidence. In general we can denote this probability as

$$P(q \mid T, b_1 \wedge \ldots \wedge b_k)$$

where b_i are (possibly negated) Boolean variable representing the truth-values of clauses.

In this case, the corresponding node in the BDD get its' probability set to 0 (if the b_i is negated) or to 1 (if the b_i is not negated) and the probability of the query is recomputed. So in this slide we consider the probability of the same query but conditional to the evidence ¬friendOf(pedro, tom). Once the probability of the nodes corresponding to the evidence is set, the probability of the query is recalculated. In this case the probability of P(likes(john, tom) | T, ¬friendOf(pedro, tom)) = 0.8 x 0.125 = 0.1.



We have mentioned in the previous slide the notion of probability of a query given some evidence. This is one of the different types of inference tasks that can be defined in the context of probabilistic logic programming.

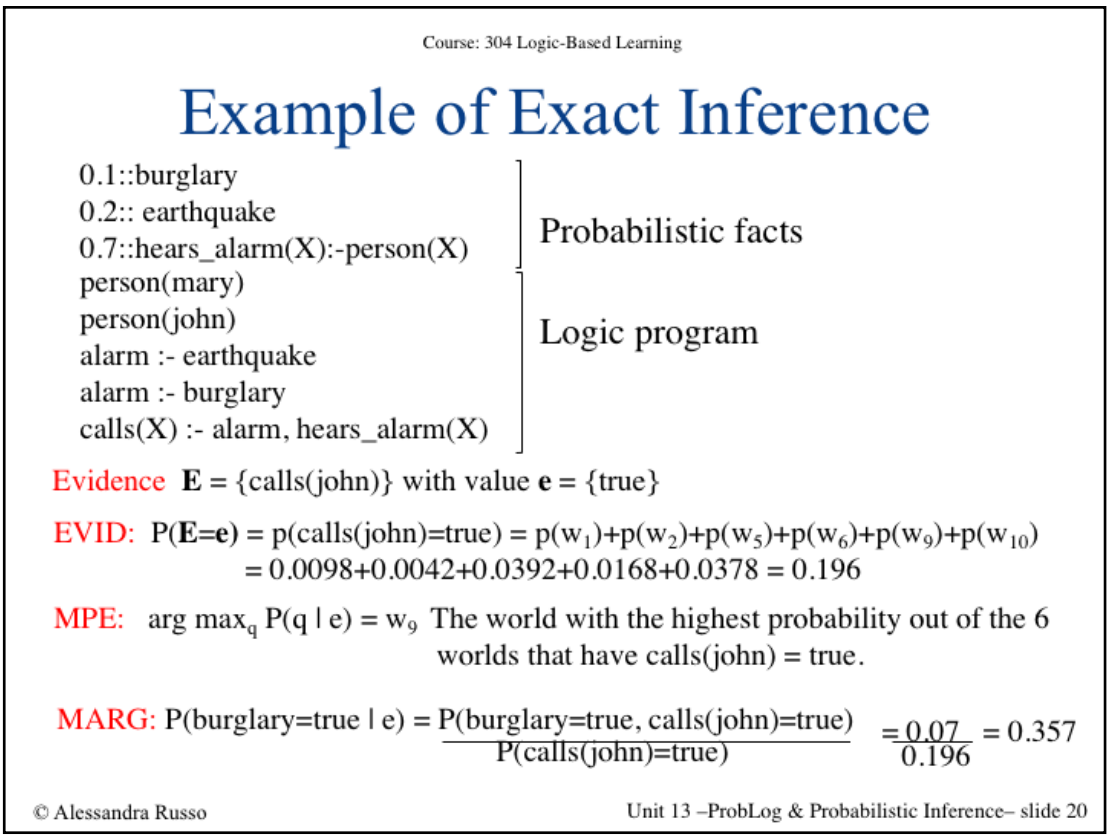
Let's consider a query *q* and an evidence *e* as conjunctions of ground literals respectively. We can define the following inference tasks:

- EVID task: compute an unconditional probability P(e), the probability of evidence. This terminology is especially used when P(e) is computed as part of a solution for the COND task. When there is no evidence, we talk of P(q), the probability of the query. An example of such a task is the probability of a given possible word.
- COND task: compute the conditional probability distribution of the query given the evidence, i.e., compute P(q | e). This type of inference is also referred in the literature as Marginal probability and denoted as MARG. It is clear that the special case of a MARG task where there is no evidence and just a given query (single or conjunction of atoms) is the same as the success probability of the given query. An example is what we have seen in the previous slide.
- MPE task: This is known as **most probable explanation.** It consists of finding the most likely truth value of all non-evidence atoms given the evidence. That is solving the optimization problem

  $arg\ max_q\ P(q|e)$

  with q being the unobserved atoms.

Exact inference tries to solve the above tasks in an exact way, modulo errors of computer floating point arithmetic. This can be sometime expensive. Advance features of ProbLog use approximate inference for solving the EVID task. But this is outside the scope of this course.



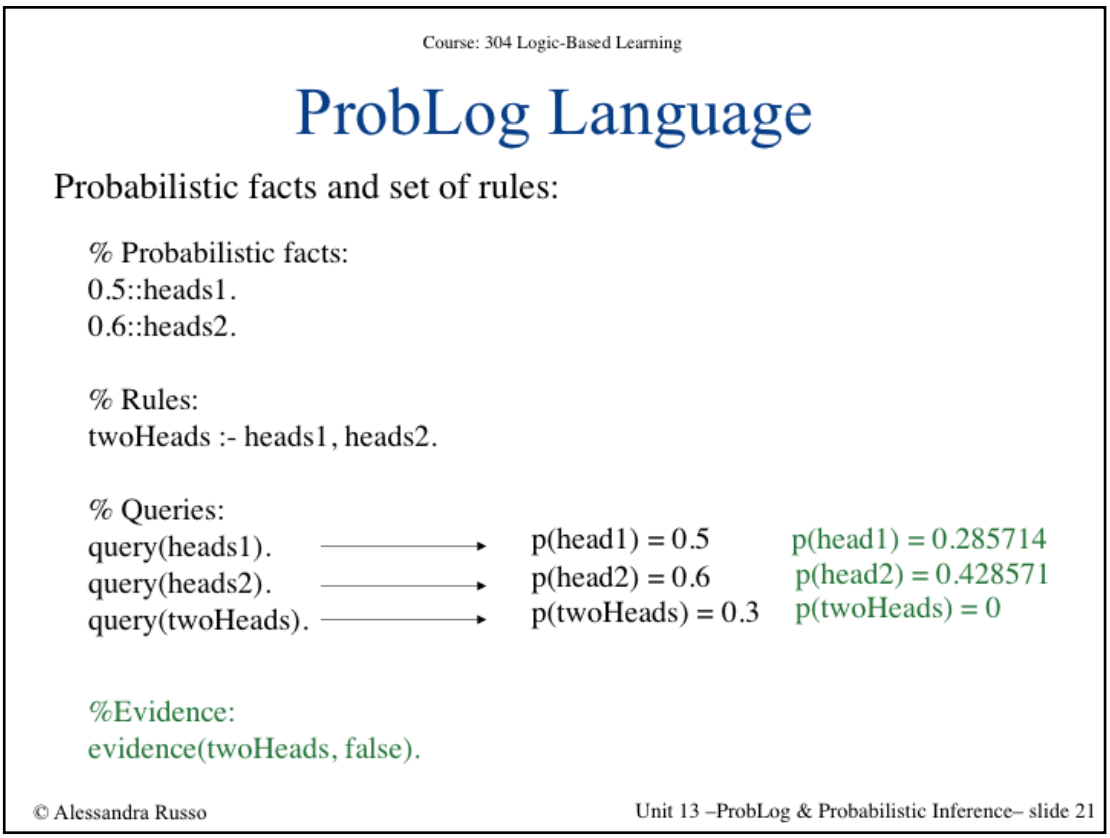
Let's consider now an example and see how the different exact inference tasks are defined in this example. So the predicates of probabilistic atoms are burglary/0, earthquake/0, hears_alarm/1 and predicates of derived atoms are person/1, alarm/0, calls/1. The intentional probabilistic fact is a syntactic sugar for the two ground probabilistic facts: 0.7::hears_alarm(mary), and 0.7::hears_alarm(john). Each probabilistic fact gives an *atomic choice*. A total choice is obtained by making an atomic choice for each probabilistic fact. Formally, a total choice is any subset of the set of all ground probabilistic atoms. Here we have 4 ground probabilistic atoms, so in total $2^4 = 16$ total choices. In the case of definite logic programs, a possible world is the LHM(CUR) where R is the logic program part of the given PLP. So we have in this case 16 possible worlds.

	Total choice C	P(C)
1	{ *burglary*, *earthquake*, *hears_alarm(john)*, *hears_alarm(mary)* }	0.0098
2	{ *burglary*, *earthquake*, *hears_alarm(john)* }	0.0042
3	{ *burglary*, *earthquake*, *hears_alarm(mary)* }	0.0042
4	{ *burglary*, *earthquake* }	0.0018
5	{ *burglary*, *hears_alarm(john)*, *hears_alarm(mary)* }	0.0392
6	{ *burglary*, *hears_alarm(john)* }	0.0168
7	{ *burglary*, *hears_alarm(mary)* }	0.0168
8	{ *burglary* }	0.0072
9	{ *earthquake*, *hears_alarm(john)*, *hears_alarm(mary)* }	0.0882
10	{ *earthquake*, *hears_alarm(john)* }	0.0378
11	{ *earthquake*, *hears_alarm(mary)* }	0.0378
12	{ *earthquake* }	0.0162
13	{ *hears_alarm(john)*, *hears_alarm(mary)* }	0.3528
14	{ *hears_alarm(john)* }	0.1512
15	{ *hears_alarm(mary)* }	0.1512
16	{ }	0.0648

EVID computes the p(calls(john)=true). We have 6 of the 16 models of the program where calls(john) is true, namely the models of total choice 1, 2, 5, 6, 9 and 10. So the EVID is the sum of the probabilities of these 6 worlds. P(E=e) = p(w1)+p(w2)+p(w5)+p(w6)+p(w9)+p(w10) = 0.196

MPE boils down to find the world with the highest probability out of the 6 worlds that have calls(john) = true. It can be verified that this is the world corresponding to total choice 9.

MARG computes the probability that there is a burglary given the evidence. There are 4 models in which P(calls(john)=true) both calls(john) and burglary are true (models 1, 2, 5 and 6), and their sum of probabilities is 0.07. Hence, P(burglary = true | calls(john) = true)= 0.07/ 0.196 = 0.357.



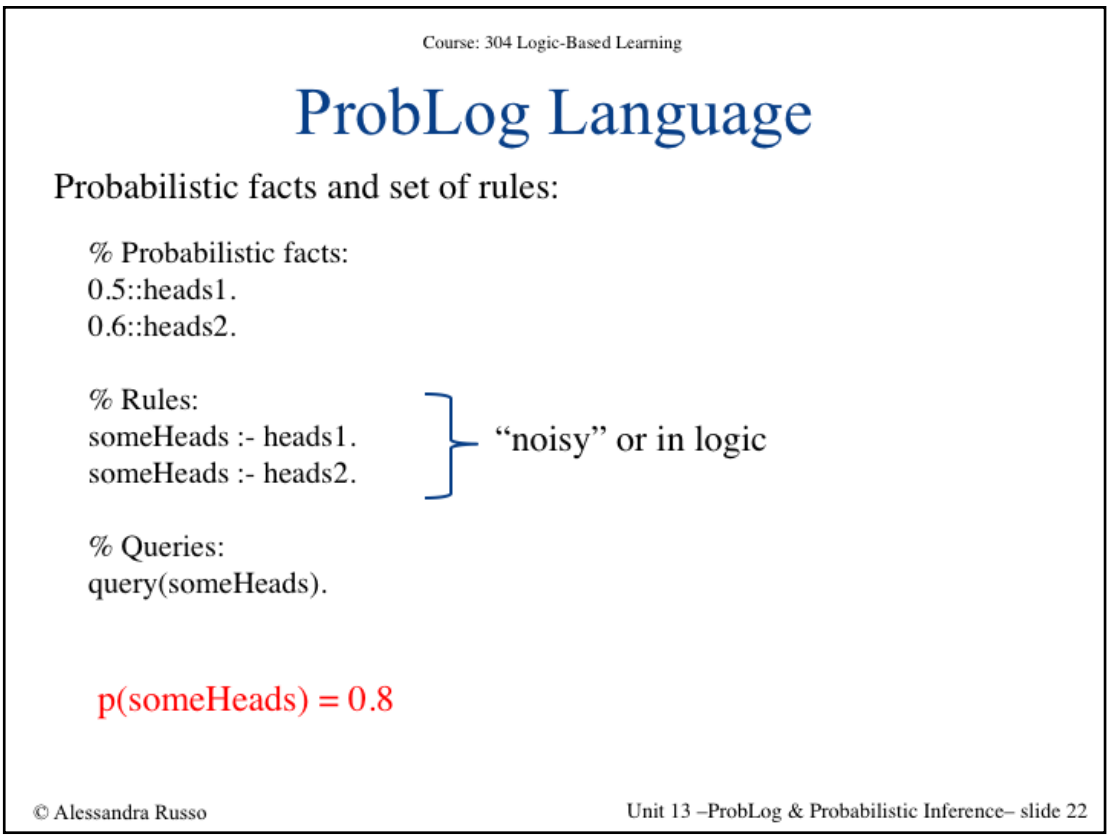
In the next few slides we summarise the basic features of the ProbLog language. The material presented in these slides is extracted from the ProbLog Tutorial Website (https://dtai.cs.kuleuven.be/problog/tutorial.html).

A ProbLog program is composed of a set of probabilistic facts and a logic program. In this example the predicates of probabilistic atoms include only *heads*/0, and predicates of derived atoms include only *twoHeads*/0. It must always be the case that the probabilistic atoms are disjoint from the derived atoms. Furthermore, if the rules are first-order, they must be **range restricted**, that is all variables in the head must appear in some positive body literal. This program defines the problem of tossing two coins. One coin is assumed to be fair (so the probability of landing on head is 50%). The second coin is biased, as the probability of landing on head is 60%. The rules define what it means for both coins to land on heads. The problem above does not have any additional evidence. So we can ask the probability of each of the given queries. The results are given in the slide. In the tutorial you will be asked to compute these probabilities in terms of the possible worlds.

In ProbLog you can also add **evidence** to the given input program by using the predicate "evidence". For example, in the program above we can include the statement "evidence(twoHeads,false)." Probability of the same three queries can be computed but conditional to the evidence. So, twoHeads has to be false, the possible world {heads1, heads2} cannot be considered as it is not consistent with the given evidence. So we have the remaining possible worlds: w_1={heads1}, w_2={heads2}, w_3={ }. The conditional probability of queries given the evidence is:

$$p(heads1 \mid \neg twoHeads) = p(heads1, \neg twoHeads)/p(\neg twoHeads)$$

Again you are asked in Tutorial 1 to calculate this probability.

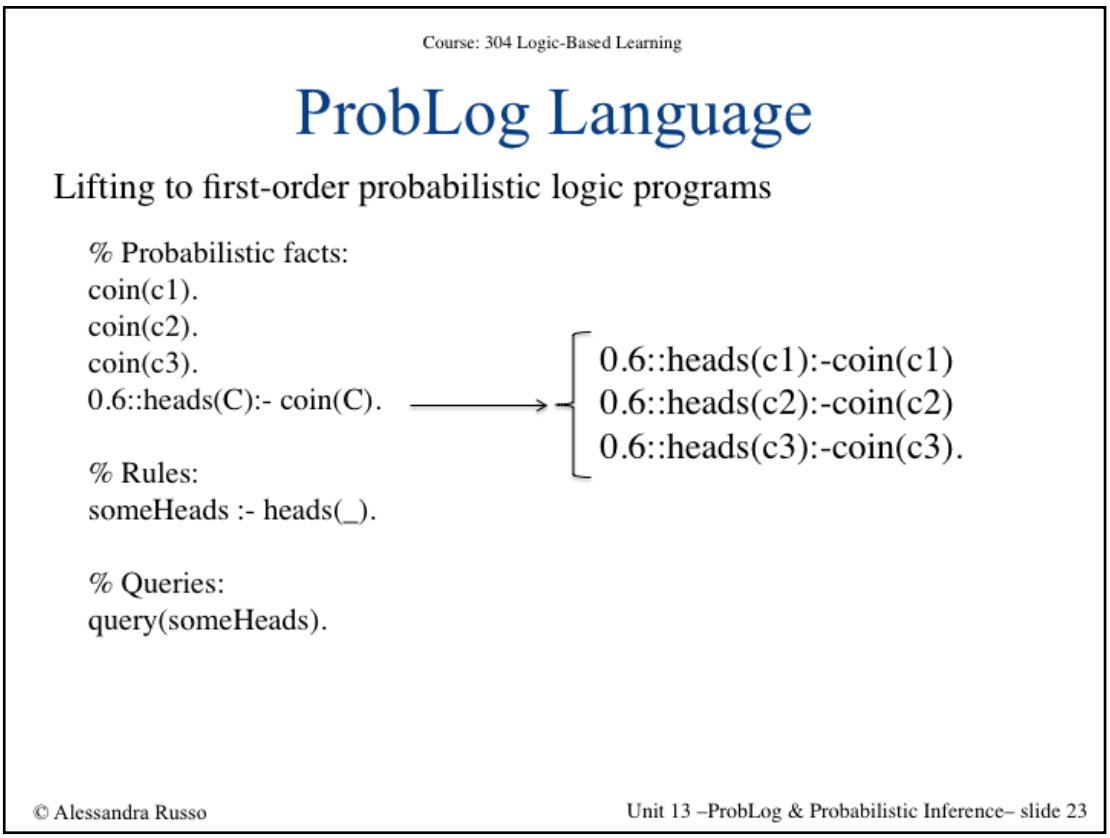


We consider now the case of a probabilistic logic program with multiple rules defining the same predicate. In the ProbLog tutorial this is the example used to show the case of "noisy" or in logic. The tutorial shows that we can calculate the probability of the query as:

P(someHeads) = 1- (1- p(heads1)(1-p(heads2))) = 1-(0.5x0.4) = 1-0.2 = 0.8

Can you show that this is indeed the probability of the query using the algorithm defined in Slide 12?

Again, in the tutorial you are asked to answer the above question.



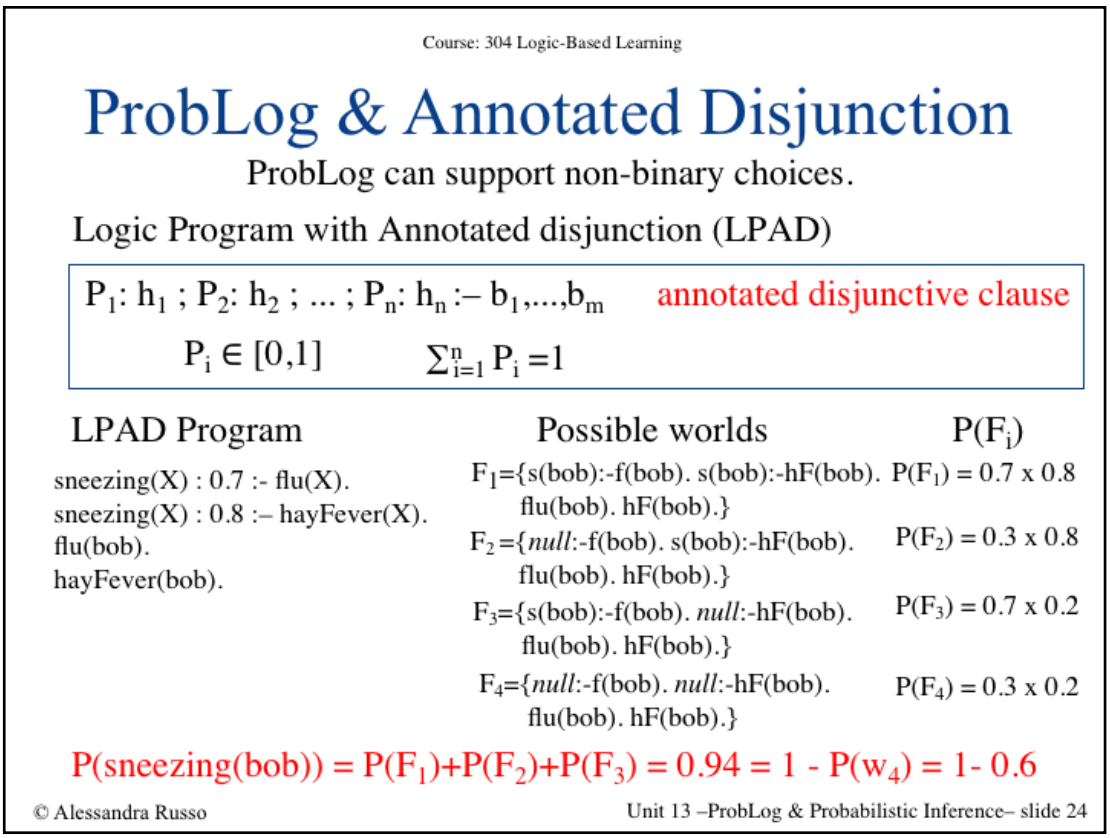
To continue the same example, rules can also have probability and the program can be first-order.

We have simplified the example to just three coins. The probabilistic part includes a rule with a probability and three facts with no probability. The rule is called **implicit probabilistic fact**, or **intentional probabilistic fact**, and it represents a set of probabilistic facts, that is 0.6::heads(c1). 0.6::heads(c2). 0.6::heads(c3). Normally, an implicit probabilistic fact is a rule with probability on the head and body literals that are not probabilistic. The idea is that the body literals define the domain of the variables that appear in the head of the rule. For instance, in the example above, the domain for the variable C is the set {c1,c2,c3}.

You can imagine the rule as instantiated. Each ground instance has a probability 0.6:

0.6:: heads(c1):-coin(c1).
0.6:: heads(c2):-coin(c2).
0.6::heads(c3):-coin(c3).

Again in the tutorial you are asked to show how the probability of the given query can be calculated using the algorithm described in slide 13.



A final feature that we will consider of ProbLog programs is the notion of **annotated disjunction.**

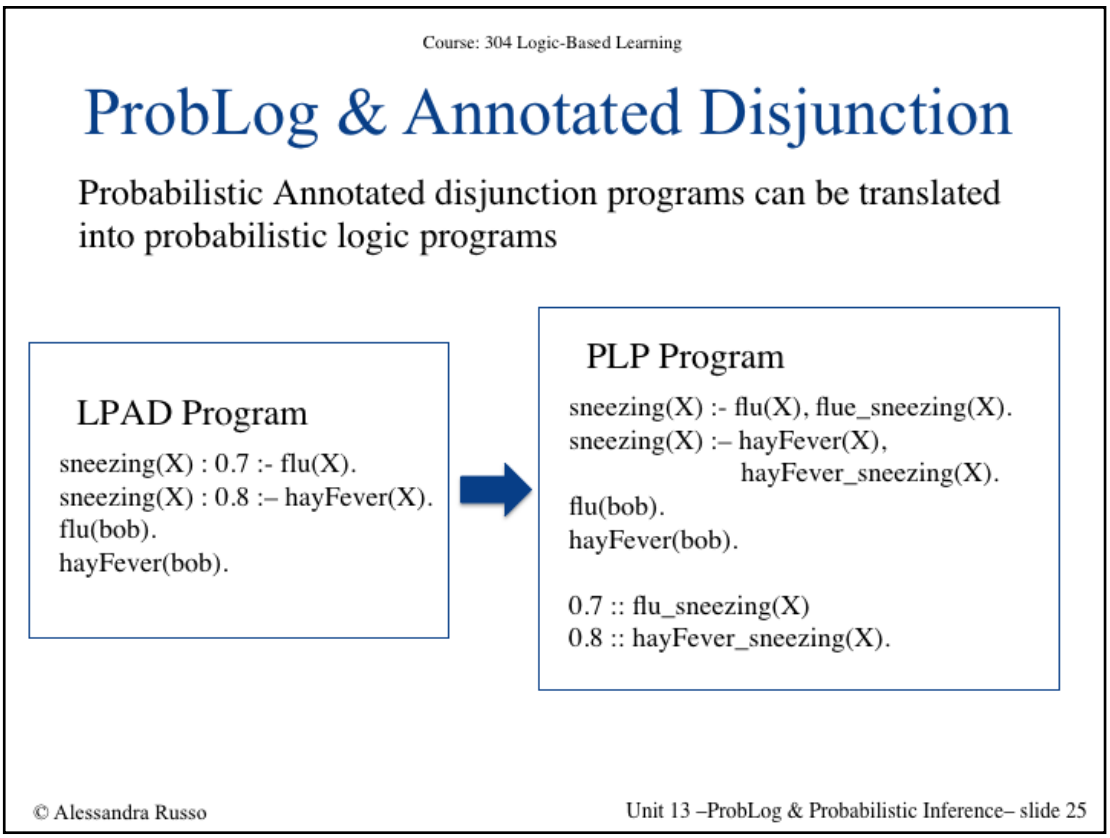
So far, probabilistic facts have been always Boolean probabilistic variables, with the probability associated to the fact being true and 1- probability associated to the fact being false. But it is also possible to represent multiple choices. This feature of ProbLog is known as **logic program with annotated disjunctions** also referred to as LPAD programs. The characteristic of these programs is that clauses may have disjunctive heads, where each head is labelled with a real number in the interval [0,1] with the property that sum of the numbers associated to the heads has to be equal to 1. In the example above the properties are: (i) Pi ∈ [0,1], for every $1 \leq i \leq n$; and (ii) $\sum_{1 \leq i \leq n} P_i = 1$.

An LPAD program is a finite set of annotated disjunction clauses. In slide 13 we have already seen an example of an LPAD program but with clauses that have only a single disjunct in the head. In that example we have the annotated disjunctive clause: 0.8:: likes(X,Y) :- friendOf(X,Z), likes(Z,Y). The assumption is that there is for each possible grounding of X and Y an implicit choice with the corresponding not likes(X,Y) that has probability 1-0.8 = 0.2.

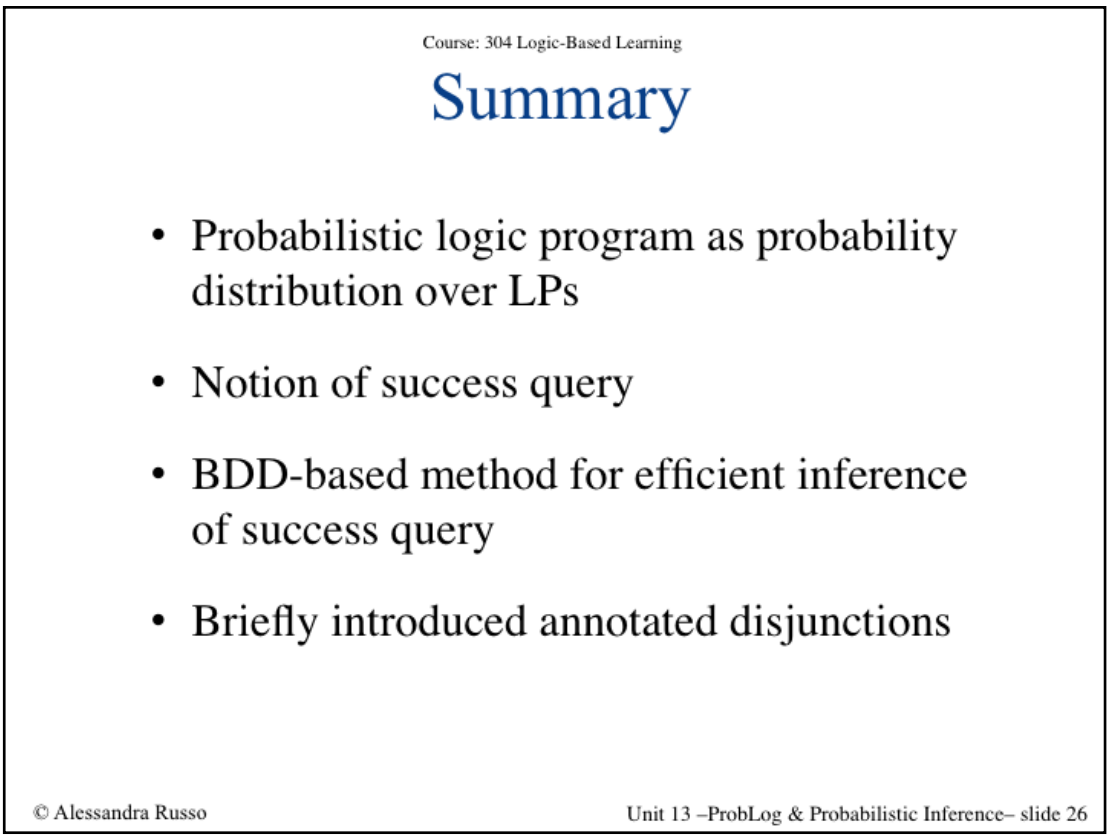
In the case of an LPAD program, possible worlds are constructed by selecting one atom from the head of each grounding of each annotated disjunctive clause. So, a given annotated disjunction clause $C = P_1: h_1 ; P_2: h_2 ; \ldots ; P_n: h_n :– b_1,\ldots,b_m$ corresponds to a set of ground probabilistic clauses $C\theta$, one for each possible grounding θ of the C clause. Each ground probabilistic clause $C\theta$ represents a choice among the n possible head atoms and is of the form: $C_k\theta = P_k: h_k :– b_1,\ldots,b_m$ for k=1,…,n.

In addition to these clauses the null clause *null* $:– b_1,\ldots,b_m$ is also assumed to be implicitly part of the LPAD program. Consider for instance the example given in this slide. The clauses:
sneezing(X) : 0.7 :- flu(X), *sneezing(X) : 0.8 :– hayFever(X),* can be seen respectively as
sneezing(X) : 0.7; null: 0.3 :- flu(X) and *sneezing(X) : 0.8; null:0.2 :– hayFever(X).*

So, we have 4 possible choices from the two annotated disjunctive clauses, as each of them has two disjunct heads. We can construct four possible programs $F_1,\ldots,F_4$ as indicated in this slide, one for each possible choice that we can make from the annotated disjunctions. We can compute the probability of each possible program and then compute the probability of a given query in the usual way as we have seen so far by summing the probabilities of the programs that satisfy the query.



We are not considering Probabilistic annotated disjunction programs. This is not really a restriction as in practice such a kind of program can be transformed into a probabilistic logic program of the type we have been seen so far. This slide show how such a transformation could we applied. Of course this is easier when the disjunctive head is a single atom. With multiple heads we will need to consider multiple conditions one of each possible head atom.



In summary, we have seen that a probabilistic logic program defines essentially a probability distribution over logic programs. We have introduced the syntax of ProbLog and defined the notion of success query. We have then defined the algorithm for computing efficiently success query using the notion of BDDs. To this end we have defined what a BDD is, shown how to construct a BDD from a given propositional formula and how to compute the probability of a BDD.

We have then introduced the notion of annotated disjunction as additional feature of ProbLog and given a brief example. More sophisticated algorithm has been given for ProbLog2 which make use of a generalisation of BDDs called d-DNNF to compute more efficiently EVID and MARG inference tasks. But this alternative algorithm is outside the scope of this course.