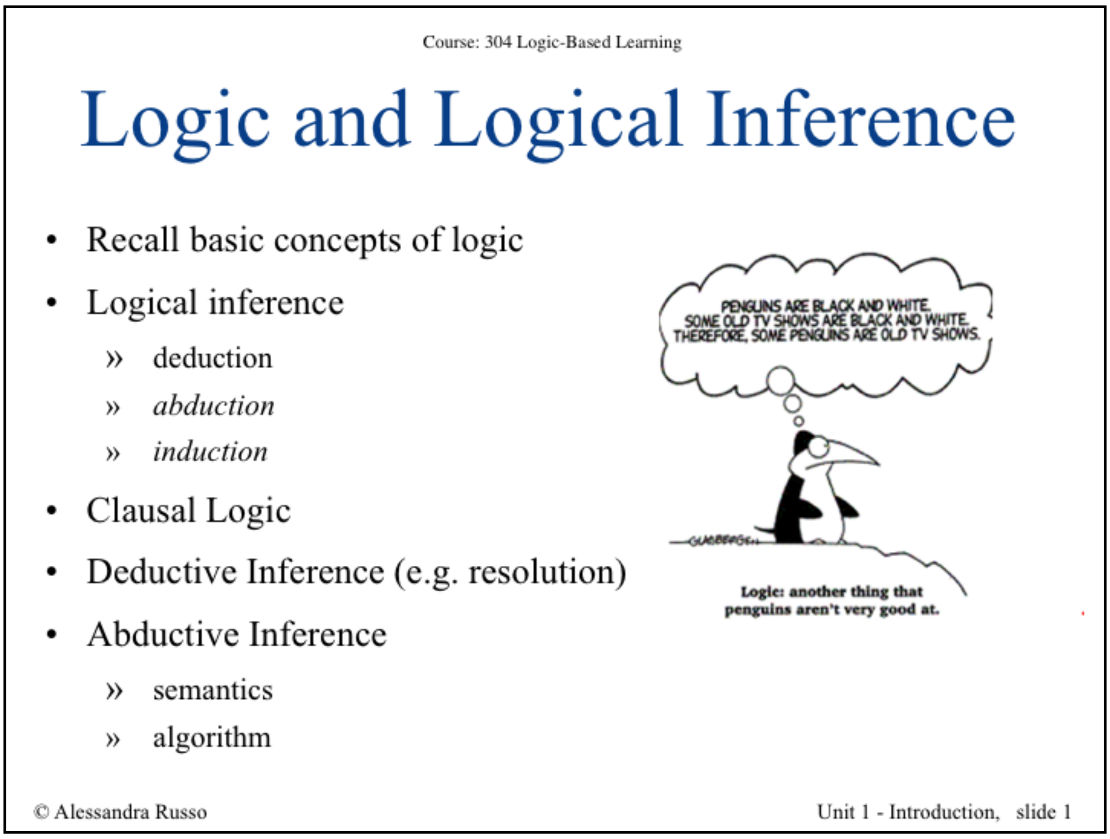


Course: 304 Logic-Based Learning

Logic and Logical Inference

- Recall basic concepts of logic
- Logical inference
  - » deduction
  - » *abduction*
  - » *induction*
- Clausal Logic
- Deductive Inference (e.g. resolution)
- Abductive Inference
  - » semantics
  - » algorithm

PENGUINS ARE BLACK AND WHITE. SOME OLD TV SHOWS ARE BLACK AND WHITE. THEREFORE, SOME PENGUINS ARE OLD TV SHOWS.
GUISBERSCH

Logic: another thing that penguins aren't very good at.

© Alessandra Russo

Unit 1 - Introduction, slide 1

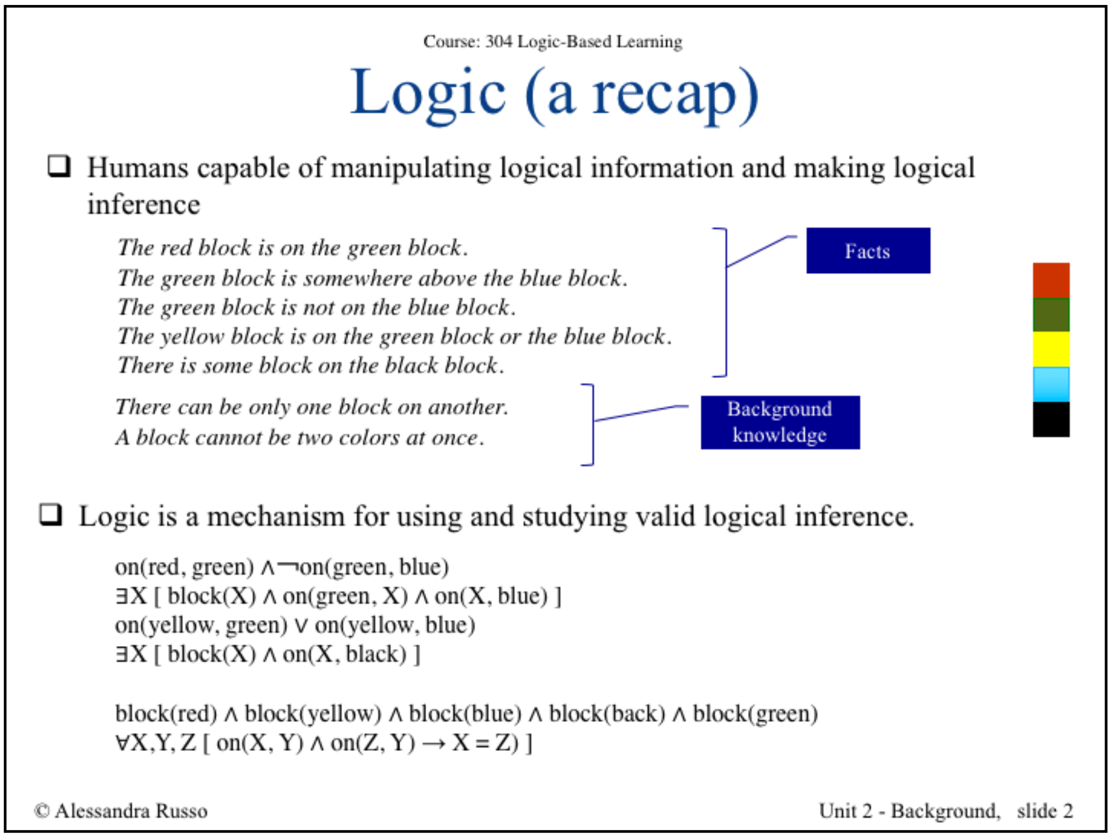
In this two hours lecture we will cover key background knowledge that students taking this course are expected to know. Part of the content of this lecture might have already been covered or will be covered in other courses (e.g. Introduction of AI, Logic and AI Programming for MSc students). But it will still be useful to recap them in order to familiarize with the terminology that we will be using.

We will quickly recall basic concepts of propositional and predicate logic. We will present three types of logic-based inference, known as deduction, abduction and induction and briefly illustrate how they differ from each other. We will then focus on the computational aspect of predicate logic. Introduce the concept of clausal logic and present one of the main proof methods used for computing the consequences of clausal theories. This is called *resolution*. We will then consider a subset of predicate logic, called Horn clauses, which is at the core of Prolog declarative programming, and illustrate how this specific language allows for a more efficient computational mechanism called SLD-resolution.

We will briefly mention the semantics of Horn clause theories, give some examples of SLD-resolution and show how it is extended to handle deductive inferences for normal clauses. These are clauses that include negation as failure.

In the second part of this lecture, we will focus on the second type of inference, namely abductive inference. We will introduce the notion of an abductive model of a problem domain, discuss the role of constraints in solving an abductive inference task and give various examples of abductive derivations. We will conclude with a semi-formal definition of the abductive inference proof procedure (also referred to as operational semantics of abduction) and with an example proof that instantiate some of the steps of the proof procedure.

Your first tutorial will be based on defining models of given clausal theories, and constructing deductive and abductive proofs for problems that are formalised using the various types of predicate logic formalisms presented here.


Course: 304 Logic-Based Learning

Logic (a recap)

- Humans capable of manipulating logical information and making logical inference

The red block is on the green block.
The green block is somewhere above the blue block.
The green block is not on the blue block.
The yellow block is on the green block or the blue block.
There is some block on the black block.

There can be only one block on another.
A block cannot be two colors at once.

- Logic is a mechanism for using and studying valid logical inference.

$\mathrm{on}(red, green) \wedge \neg \mathrm{on}(green, blue)$
$\exists X\ [\ \mathrm{block}(X) \wedge \mathrm{on}(green, X) \wedge \mathrm{on}(X, blue)\ ]$
$\mathrm{on}(yellow, green) \vee \mathrm{on}(yellow, blue)$
$\exists X\ [\ \mathrm{block}(X) \wedge \mathrm{on}(X, black)\ ]$

$\mathrm{block}(red) \wedge \mathrm{block}(yellow) \wedge \mathrm{block}(blue) \wedge \mathrm{block}(back) \wedge \mathrm{block}(green)$
$\forall X,Y,Z\ [\ \mathrm{on}(X, Y) \wedge \mathrm{on}(Z, Y) \rightarrow X = Z)\ ]$

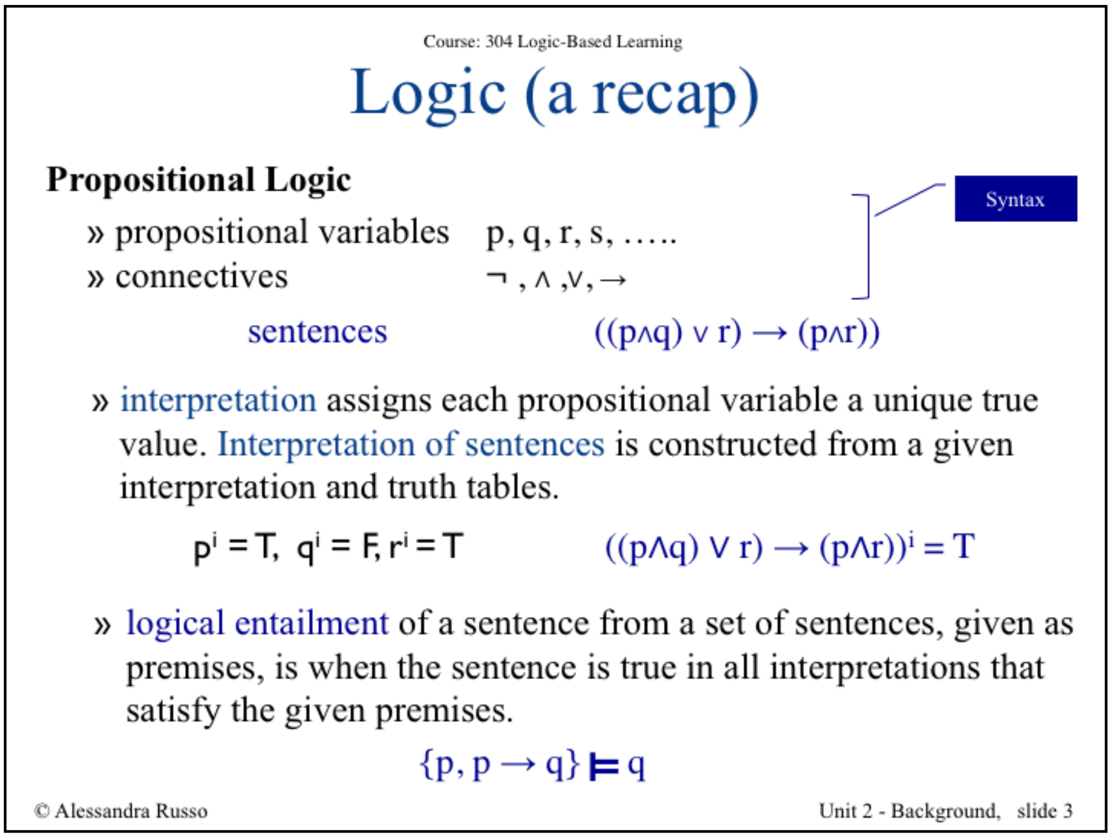
© Alessandra Russo Unit 2 - Background, slide 2

People have the ability of acquiring information about the world and use this information to further derive knowledge. They can, for instance, represent, manipulate logical information, including not just simple facts but also more complex forms of information, such as negations, choices, constraints, etc. For instance, given the above set of premises (e.g. some facts) about the arrangement of five blocks in a tower (in a situation where there are only these five blocks) people are capable of inferring the exact configuration of the tower. To reach conclusions from given premises we mentally construct *proofs*: sequence of intermediate steps where at each step straightforward intermediate conclusions are derived.

Formal logic, or logic (for simplicity), is a mechanism for using and studying valid logical inferences. It provides a formal language with precise syntax and unambiguous semantics, as well as precise rules for manipulating expressions written in this formal language in a way that it respects its meaning.

Logic is therefore defined in terms of a syntax (language), a formal semantics and a set of rules that help manipulate mechanically given information (or knowledge).

You should all have already studied a basic course on logic and have an idea of the syntax and semantics of propositional logic and predicate logic. Above is an example of formal representation, in predicate logic, of given facts and background knowledge. Applying, for instance, natural deduction it would be possible to derive the exact configuration of the tower as conjunction of five ground atoms. We will not show how this proof can be constructed as natural deduction, as this type of automated proof procedure is outside the scope of this course. But we will see later a particular representation style of predicate logic, called “clausal logic”, for which we will present an inference mechanism called resolution, which is closely related to how Prolog computes answers to given queries.



Course: 304 Logic-Based Learning

Logic (a recap)

Propositional Logic

» propositional variables p, q, r, s, …..
» connectives ¬ , ∧ ,∨, →

Syntax

sentences $((p\wedge q) \vee r) \rightarrow (p\wedge r))$

» interpretation assigns each propositional variable a unique true value. Interpretation of sentences is constructed from a given interpretation and truth tables.

$p^i = T,\ q^i = F, r^i = T$ $((p\wedge q) \vee r) \rightarrow (p\wedge r))^i = T$

» logical entailment of a sentence from a set of sentences, given as premises, is when the sentence is true in all interpretations that satisfy the given premises.

$\{p, p \rightarrow q\} \models q$

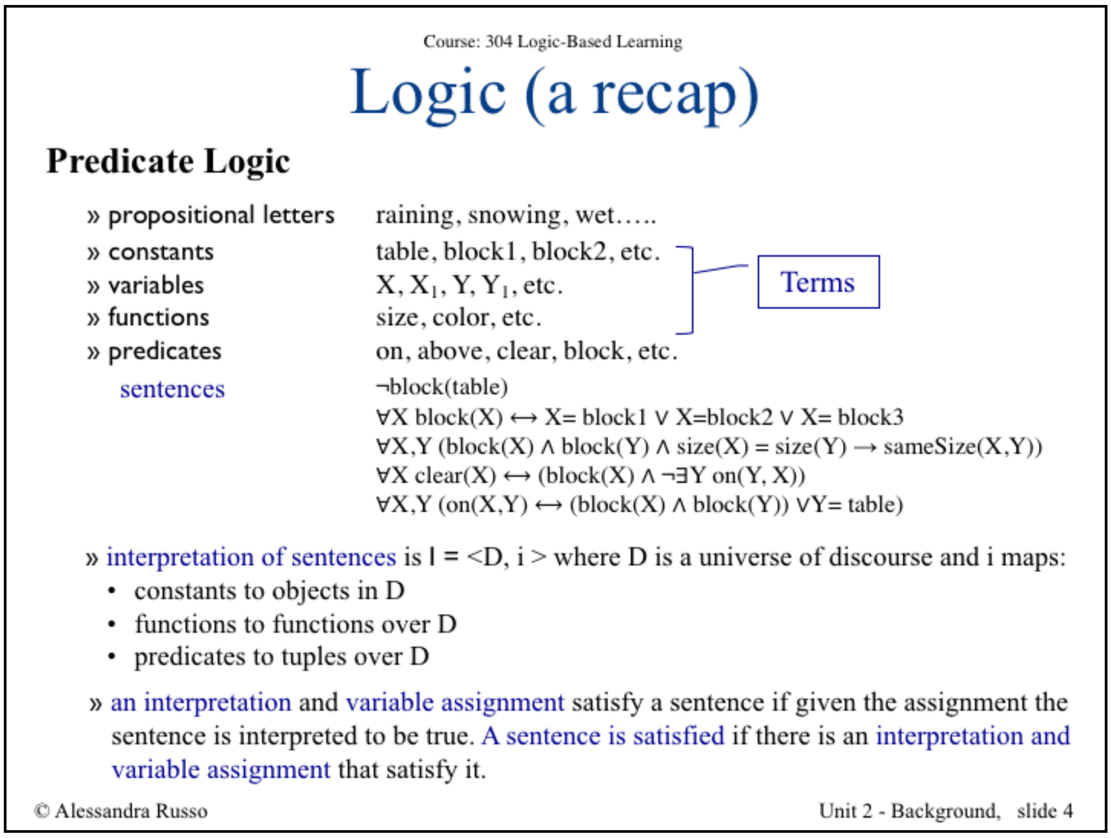
© Alessandra Russo Unit 2 - Background, slide 3

Propositional Logic is concerned with propositions and their logical interrelationships. A proposition is normally a possible "condition" of the world about which we want to say something. The condition need not be true in order for us to talk and reason about it. In fact, we might want to say that it is false if some other proposition is true. As any other logic, propositional logic consists of a syntax and a semantics. The former defines the language with which we can express properties about the world; whereas the semantics defines the meaning of sentences written in this language.

The syntax of propositional logic is given by the main *logical connectives* ∧, ∨, ¬, → and *extra-logical symbols*, also called propositional variables. For convention, propositional variables can be written using sequence of alphanumeric characters beginning with a lower case character. Sentences are propositional variables or expressions inductively defined in terms of simpler sentences composed together via the logical connectives. For instance $((p\wedge q) \vee r) \rightarrow (p\wedge r))$ is an implication sentence formed by the two simpler sentences $(p\wedge q) \vee r)$ and $(p\wedge r)$ linked together by means of the implication (→) connective. Remember connectives have a fixed operator precedence and if sentences are constructed without the use of parenthesis the predefined precedence applies.

The semantics of propositional logic is concerned with logical meaning of sentences. This is determined by the logical meaning of the connectives, which is fixed and given in terms of truth tables, and the logical meaning of the propositional variables. The latter is not fixed and it is defined in terms of an *interpretation*: assignment of a truth value (Boolean meaning) to each propositional variable in the language. We will denote an interpretation using the letter *i* and the meaning of a propositional variables or (complex) expression under an interpretation *i* by superscripting the variable or expression with *i*.

An interpretation that makes a given sentence true is said to *satisfy* the sentence. A sentence is *unsatisfiable* if and only if it is not satisfied by any interpretation. A sentence is said to be *valid* if and only if it is satisfied by every possible interpretation. Given a set S of sentences, a model of S is an interpretation that satisfies all the sentences in S. Finally, key to the logic is the notion of logical entailment. A set S of sentences logically entails a sentence φ (written S ⊨ φ) if and only if every interpretation that satisfies S also satisfies φ.


Course: 304 Logic-Based Learning

Logic (a recap)

Predicate Logic

» propositional letters	raining, snowing, wet…..
» constants	table, block1, block2, etc.
» variables	X, X_1, Y, Y_1, etc.
» functions	size, color, etc.
» predicates	on, above, clear, block, etc.
sentences	$\neg block(table)$ $\forall X\ block(X) \leftrightarrow X= block1 \vee X=block2 \vee X= block3$ $\forall X,Y\ (block(X) \wedge block(Y) \wedge size(X) = size(Y) \rightarrow sameSize(X,Y))$ $\forall X\ clear(X) \leftrightarrow (block(X) \wedge \neg\exists Y\ on(Y, X))$ $\forall X,Y\ (on(X,Y) \leftrightarrow (block(X) \wedge block(Y)) \vee Y= table)$

Terms

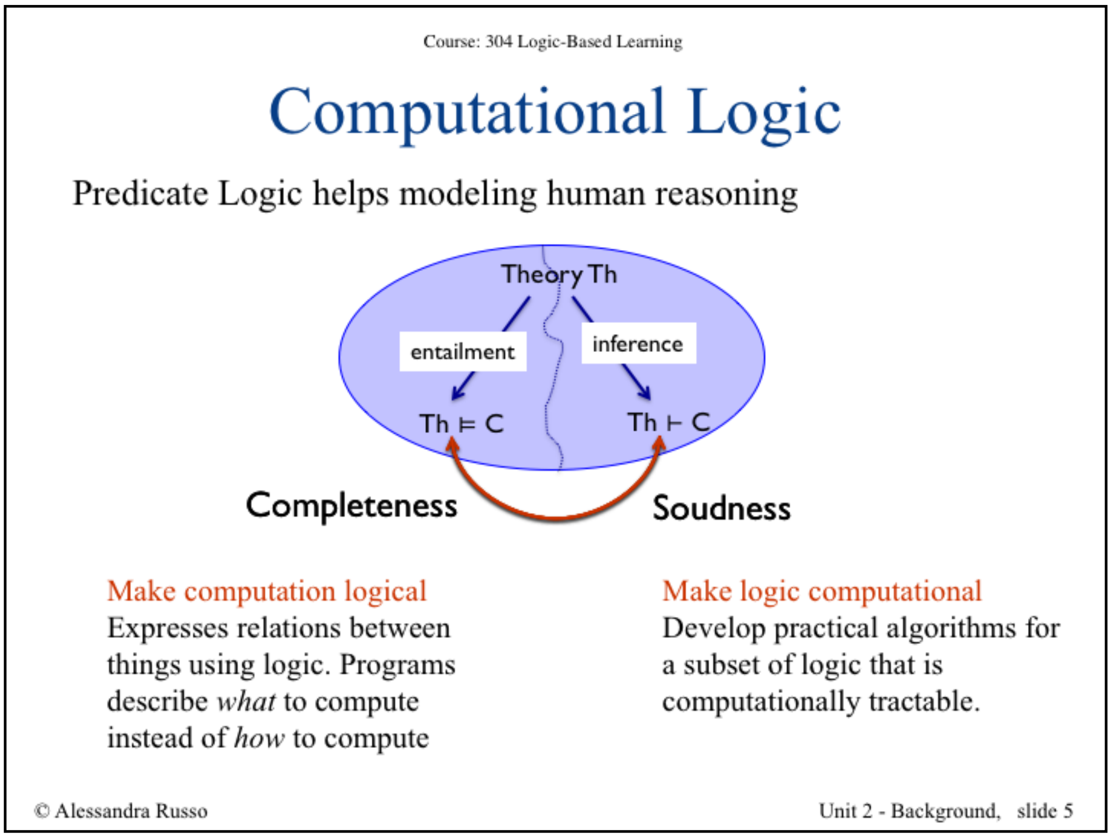
» interpretation of sentences is I = <D, i > where D is a universe of discourse and i maps:
- constants to objects in D
- functions to functions over D
- predicates to tuples over D

» an interpretation and variable assignment satisfy a sentence if given the assignment the sentence is interpreted to be true. A sentence is satisfied if there is an interpretation and variable assignment that satisfy it.

© Alessandra Russo Unit 2 - Background, slide 4

Propositional Logic does a good job of allowing us to talk about relationships among propositions, and it gives us the machinery to derive logical conclusions based on these relationships. Unfortunately, when we want to say general properties, as for instance "*if one person knows a second person, then the second person knows the first*" , we find that Propositional Logic is inadequate. Predicate Logic solves this problem by providing a finer grain of representation. In particular, it provides us with a way of talking about *individual objects* and their *interrelationships*. It allows us to assert the existence of objects with a given relationship and allows us to talk about all/some objects having a given relationship and to apply those facts to specific cases. The formal language extends that of propositional logic with two additional logical operators, quantifiers ∀ and ∃, and extra-logical symbols that include constants, function and predicates. The formal language is indeed more expressive that propositional logic in the sense that it allows us to talk about elements of a given domain, using constant symbols and variables; it allows us to specify functions over this domain, by means of function symbols, and mainly it allows us to define and specify relations among elements of the domain, by means of predicate symbols. By convention, variables are assumed to be written as sequences of alphanumeric characters beginning with a capital case character whereas function and predicates to begin with a small case character. Terms in the languages are constants, variables and any term constructed by applying a function to term arguments.

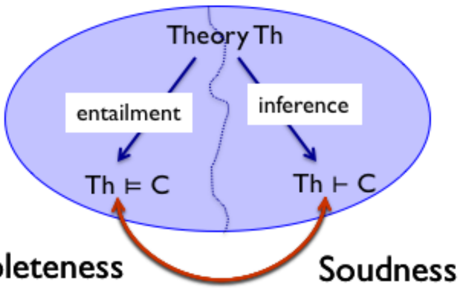
Logical sentences are negations, conjunctions, disjunctions, implications, and equivalences constructed exactly as in propositional logic, except that the elementary components are most of the time relational sentences rather than propositional constants. *Quantified sentenc*es are formed from a quantifier, a variable, and an embedded sentence. The embedded sentence is called the scope of the quantifier. There are two types of quantified sentences: universally quantified sentences and existentially quantified sentences. Each variable that appears in a sentence is quantified. A sentence is *ground* if and only if it contains no variables. For example, the sentence *block(table)* is ground, whereas the sentence $\forall X.block(X)$ is not. An interpretation is given by a domain of discourse and a mapping that assigns constant symbols to elements of the domain, function symbols to functions with the same arity over the domain and each predicate to a relation with the same arity over the domain. Satisfiability depends on a given interpretation and variable assignment.



Course: 304 Logic-Based Learning

Computational Logic

Predicate Logic helps modeling human reasoning



Make computation logical
Expresses relations between things using logic. Programs describe *what* to compute instead of *how* to compute

Make logic computational
Develop practical algorithms for a subset of logic that is computationally tractable.

© Alessandra Russo

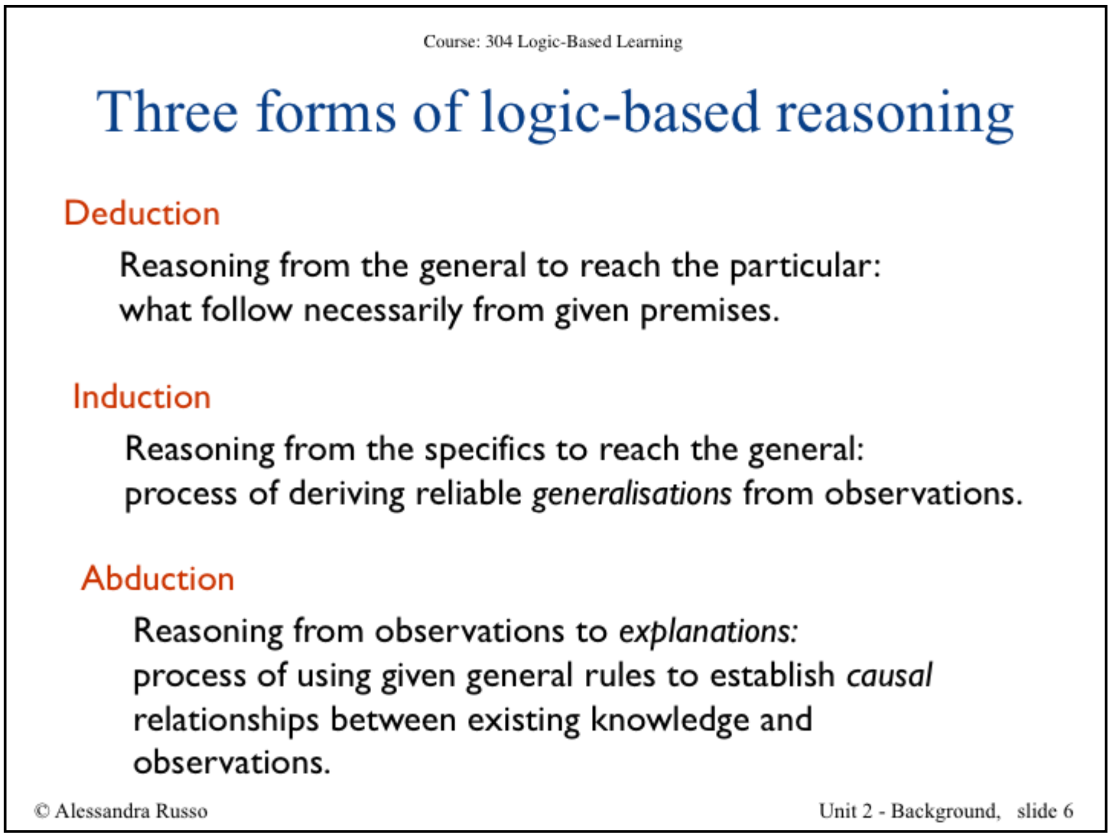
Unit 2 - Background, slide 5

Logic is, however, not only a mechanism for modeling the real-world. The semantics provides the mathematical framework for formally studying validity and correctness of the logical model. But its syntax and proof procedures provide also ways for syntactically manipulating sentences and infer knowledge from given assumptions or premises.

Computational Logic is the area of logic that focuses on the development of declarative programming mechanisms, such as logic programming, which make "computations logical". But, at the same time, it investigates algorithms capable of computing logic-based inference in a practical way so to return solutions to posed questions. In Logic Programming, for instance, a program consists of a set of axioms (facts or extensional knowledge) and a set of rules. They define relations between entities in a logical manner. Logic programming systems, such as Prolog, compute the consequences of the axioms and rules in order to answer queries.

Part of Computational Logic is also Automated Theorem Proving that studies tractable algorithms for computing logical inferences. Deductive databases, for instance, use Datalog (a subset of predicate logic) to model relational databases, and queries expressed in Datalog can be computed using SQL. Rules defining new predicates can instead be computed using *views* of intentional relations. All these areas can be seen as belonging to computational logic.

The two aspects of computational logic (i.e. defining a problem in logic term and providing algorithms for solving logical inference in a practical way) are typical of various types of logic-based reasoning. Depending on the problem, different logic-based computations may be required. Going back to few of the greatest philosophers in the 20th century (e.g. Karl Popper and subsequently Pierce), there are three different forms of reasoning: *deduction*, *abduction* and *induction*. Computational problems formalised in a logical manner can refer to any one of these three forms. In the first case the problem is referred to as a *deductive task*, in the second case as an *abductive task* and in the third case an *inductive task*. On the other hand, different computational algorithms have been developed to support the computation of these tasks. We will illustrate the difference between these forms of reasoning, by presenting first deduction, then defining abduction and then focusing on induction. We will look at them as computational tasks. So we concentrate on subset of predicate logic that is computationally tractable (i.e. Horn clauses).


Course: 304 Logic-Based Learning

Three forms of logic-based reasoning

Deduction

Reasoning from the general to reach the particular:
what follow necessarily from given premises.

Induction

Reasoning from the specifics to reach the general:
process of deriving reliable *generalisations* from observations.

Abduction

Reasoning from observations to *explanations:*
process of using given general rules to establish *causal* relationships between existing knowledge and observations.

© Alessandra Russo

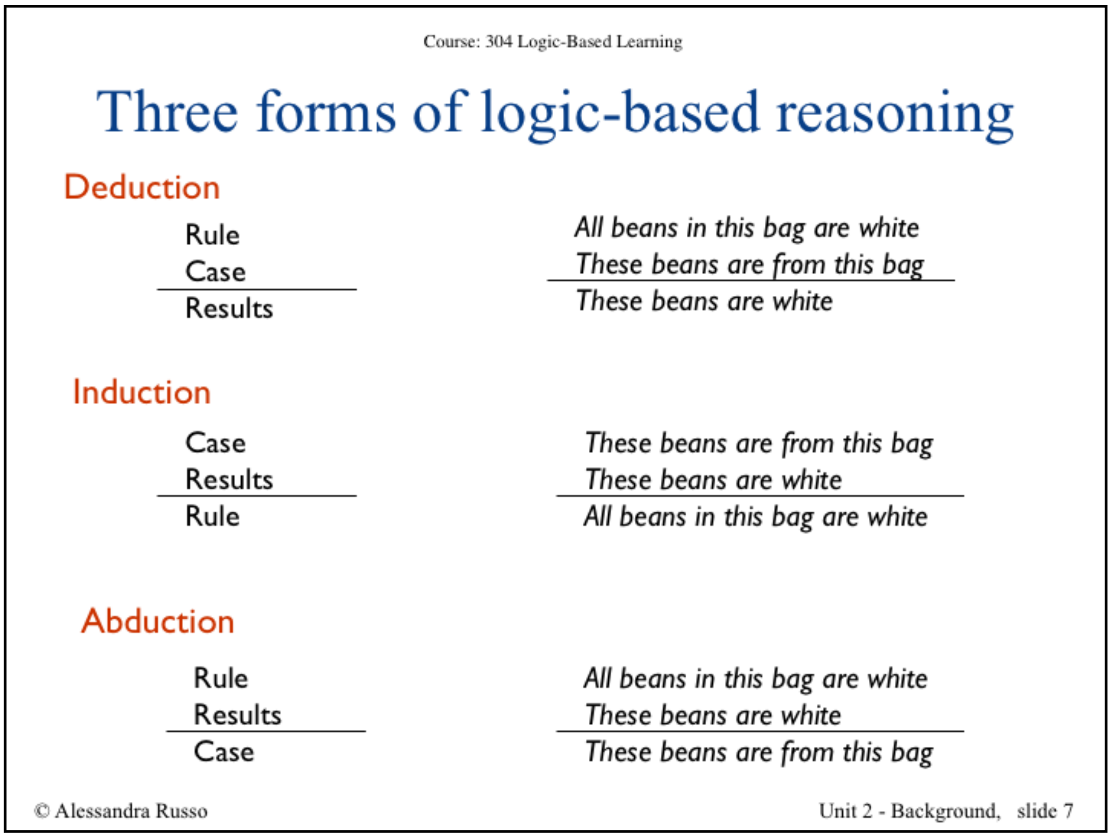
Unit 2 - Background, slide 6

Abduction and induction are two forms of reasoning that alongside deduction have played a prominent role in the study of logic and philosophy of science. Historically, these three types of reasoning have their roots in the work of Aristotle, but were placed in their modern context of logic-based reasoning by C.S. Peirce around the twentieth century. The distinction between deduction, on the one hand, and induction and abduction, on the other hand, corresponds to the distinction between necessary and non-necessary inference. In deductive inferences, what is inferred is *necessarily true* if the premises from which it is inferred are true; that is, the truth of the premises guarantees the truth of the conclusion. But not all inferences are of this type. Abduction and induction are non-necessary inferences. They are instead *ampliative inferences*.

The key distinction is that whereas deduction aims to make explicit the consequences already implicit in some existing body of knowledge, abduction and induction aim to discover genuinely new knowledge from empirical data relating to some phenomena.

Abduction is the process of *explanation* — reasoning from effects to possible causes; while induction is the process of *generalisation* — reasoning from samples to wider populations. Significant progress has been made in the areas of Artificial Intelligence (AI) and Machine Learning (ML), on formalising and automating these forms of reasoning. This progress has led to deeper understanding and increasing real-world applications. Much of this success comes from work in the areas of Abductive Logic Programming (ALP) and Inductive Logic Programming (ILP). These are very active area of research, but with still many open problems that are worth PhD thesis.

(You might be the next PhD student working in this area. Think about it!)


Course: 304 Logic-Based Learning

Three forms of logic-based reasoning

Deduction

Rule	*All beans in this bag are white*
Case	*These beans are from this bag*
Results	*These beans are white*

Induction

Case	*These beans are from this bag*
Results	*These beans are white*
Rule	*All beans in this bag are white*

Abduction

Rule	*All beans in this bag are white*
Results	*These beans are white*
Case	*These beans are from this bag*

© Alessandra Russo

Unit 2 - Background, slide 7

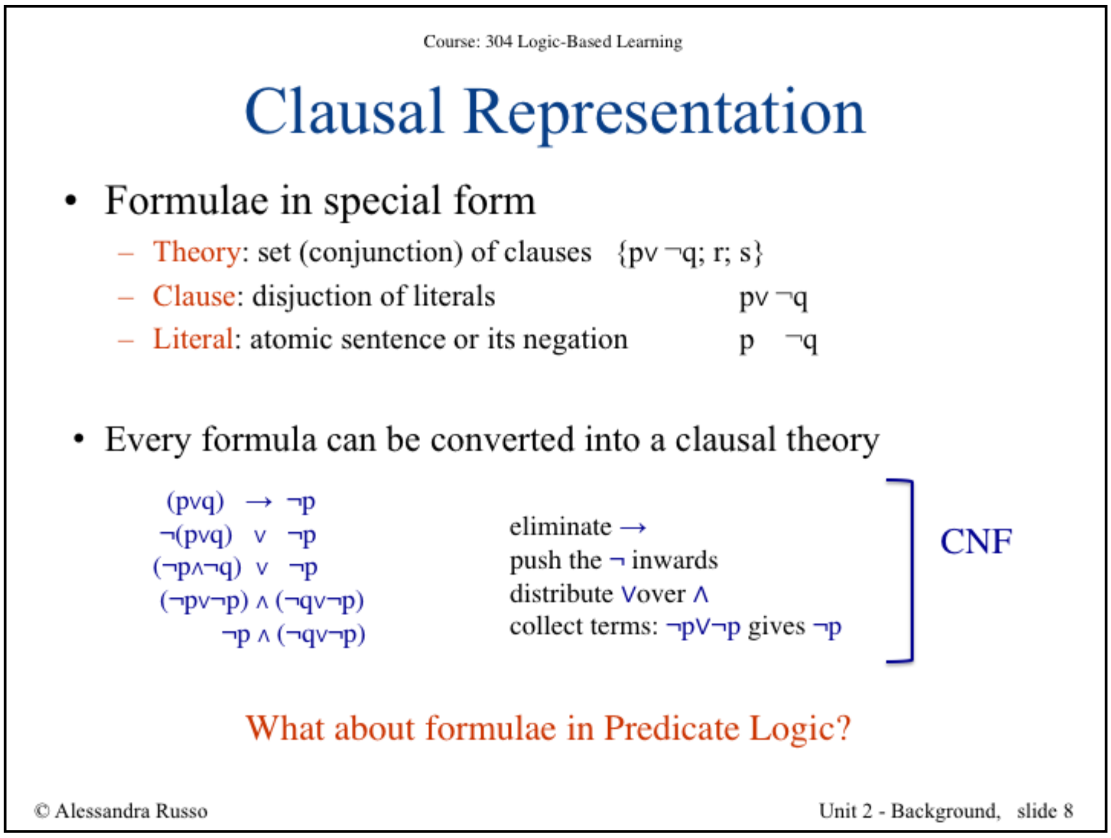
Let's consider a general inference step to be defined in terms of a logical syllogism of the form: *Rule and Case*, given as premises, lead to *Results* as conclusion.

Deduction, has therefore the syllogistic form in which given a rule and some case it is possible to infer conclusions that are already implicitly covered by the premises. The validity of its inference step is guaranteed by the fact that all possible relevant information needed to reach the conclusion is in the premises already. So the principle of a deductive inference is that true premises yield to true conclusions. As it starts from general rules, the conclusions are generated through correct (syntactic) unfolding of the rules given in the premises. In the next few slides we will refer to resolution for clausal logic, and in particular for Horn clauses, as one of the mechanisms for computing these deductive unfolding (or inference steps).

Induction can be understood, to a certain extent, as the inverse of deduction. It starts from the "Case" and the observations and infer the general rule that would, consistently with the Case, covers the given observations. So it is the inference of a general rule from the observation of results in given cases. It is called ampliative because its amplifies the generality of its conclusion (i.e. the learned rules) beyond the boundary of the knowledge expressed in the premises.

Abduction is another kind of inversion of deduction. Cases, called in this form of reasoning *abductive solutions*, are the inference of particular instances from general rules and the results that the general rule would give if applied to these instances. So this inference method proceeds from rules and results to cases.

In his work Pierce emphasized that deduction, induction and abduction are the main three types of valid reasoning and human thinking is the results of combinations of (some of) these types of reasoning. Within the last 15 years researchers have started exploring and investigating possible interrelationships between these three forms of reasoning and in particular between abduction and induction both in philosophical/theoretical terms as well as in computational terms. This has lead to interesting breakthroughs in the last 15 years in the area of logic-based learning and new ways of logically *formalising the notion of scientific inquiries through formal and computational integration of abduction and induction*. We will cover this part of the course when we will look at the HAIL approach to logic-based learning.


Course: 304 Logic-Based Learning

Clausal Representation

- Formulae in special form
  - Theory: set (conjunction) of clauses {pv ¬q; r; s}
  - Clause: disjuction of literals pv ¬q
  - Literal: atomic sentence or its negation p ¬q
- Every formula can be converted into a clausal theory

(pvq) → ¬p	
¬(pvq) v ¬p	eliminate →
(¬p∧¬q) v ¬p	push the ¬ inwards
(¬pv¬p) ∧ (¬qv¬p)	distribute ∨ over ∧
¬p ∧ (¬qv¬p)	collect terms: ¬p∨¬p gives ¬p

CNF

What about formulae in Predicate Logic?

© Alessandra Russo Unit 2 - Background, slide 8

As the focus in this course is not only to study logic-based learning as a inference task but also to explore how this inference task can be computed, we will restrict ourselves to the subset of predicate logic that is computational tractable and for which efficient automated proof procedures able to compute logical inference exist. We will start with a subset of predicate logic called Horn clauses (which is a subset of Prolog language). We will then consider "normal clauses", which extend Horn clauses with the notion of negation as failure, and subsequently use Answer Set Programming language. The latter is a computational logic language and environment that has recently gained wide attention due of the computational efficiency of the ASP solvers used to find solutions to problems written in this language.

Before we recap basic concepts of Horn clauses and SLD, we summarise the concepts of *clausal logic* and resolution in both propositional and predicate logic which are at the basis of Horn clauses and SLD resolution used by Prolog.

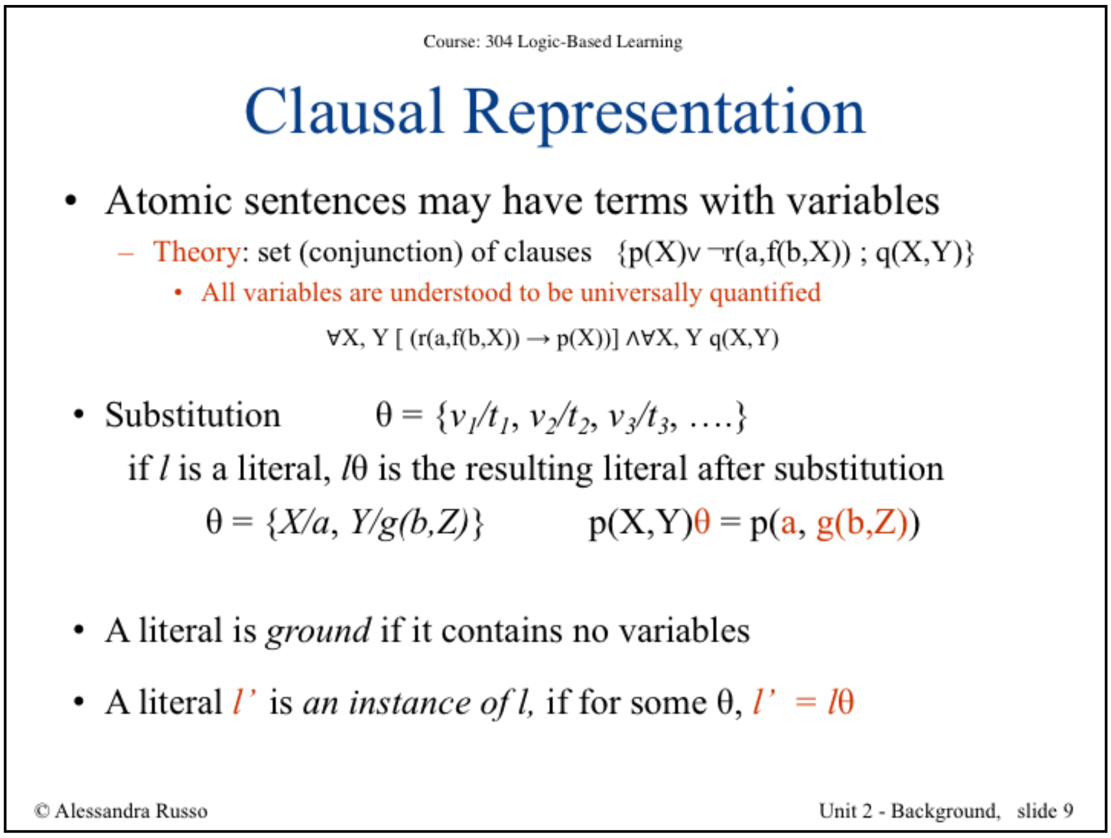
We might have already encountered the definition of clausal logical. A clausal theory is a set of clauses. A clause is a disjunction of literals and a literal is a atomic sentence or a negated atomic sentence, called respectively positive and negative literals. Clausal finite theories can also be seen as Conjunctive Normal Form (CNF) formulae, since all the clauses in the theory are in conjunction with each other and each clause is a disjunction of literals. In the propositional case it is possible to transform any sentence into an equivalent CNF. An example is given in this slide. This guarantees that restricting ourselves to formulas in CNF does not semantically restrict the expressivity of the language or of the problems that can be models.

The transformation process includes the following steps:

i) change implication into disjunction using the property that: p → q is equivalent to ¬pvq.

ii) push negation inwards using the property that: ¬(pvq) is equivalent to ¬p∧¬q and that ¬(p∧q) is equivalent to ¬pv¬q.

iii) distribute ∨ over ∧ using the property that: p ∨ (q∧s) is equivalent to (p ∨ q) ∧ (p ∨ s).


Course: 304 Logic-Based Learning

Clausal Representation

- Atomic sentences may have terms with variables
  - Theory: set (conjunction) of clauses $\{p(X) \vee \neg r(a,f(b,X)) \,;\, q(X,Y)\}$
    - All variables are understood to be universally quantified

$$\forall X, Y\, [\, (r(a,f(b,X)) \rightarrow p(X))] \wedge \forall X, Y\; q(X,Y)$$

- Substitution $\theta = \{v_1/t_1, v_2/t_2, v_3/t_3, \ldots\}$

  if l is a literal, $l\theta$ is the resulting literal after substitution

  $\theta = \{X/a,\ Y/g(b,Z)\}$ $\quad p(X,Y)\theta = p(a, g(b,Z))$

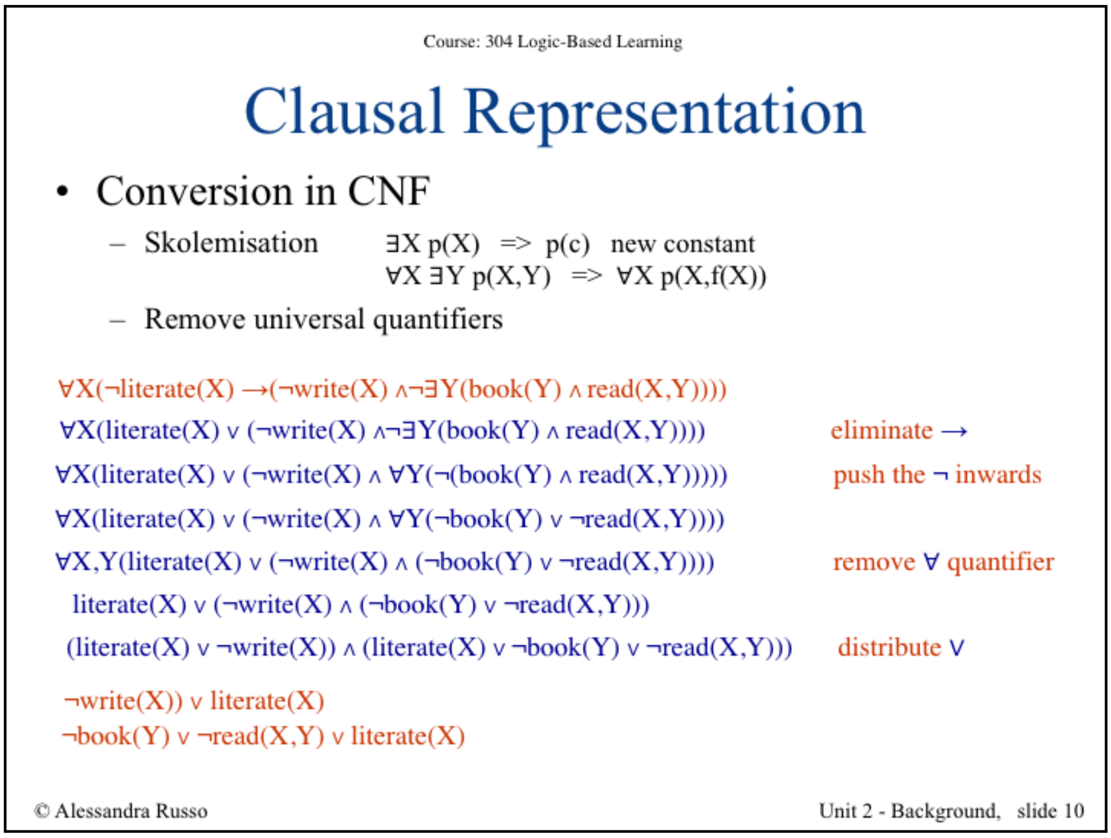
- A literal is *ground* if it contains no variables
- A literal l' is *an instance of l,* if for some θ, $l' = l\theta$

© Alessandra Russo Unit 2 - Background, slide 9

The transformation of first-order logic sentences into clausal form is more complex. This is due to the presence of variables and quantifiers. All variables that appear in a first-order logic sentence are quantified either with existential or universal quantifiers. But a first-order clause is a disjunction of literals where all variables that appear in the literals are universally quantified. So the transformation process requires extra steps. In the next slide we describe these steps and we give a couple of examples.

Key concepts, which you may have also seen in previous courses, are the notions of *substitution*, and *instances* of first-order sentences. A substitution is a systematic replacement of all occurrences of a term in a sentence by another term. To avoid name clashes variables in the introduced new term should not already appear in the sentence. For example, in the example above, the variable Y cannot be replaced by X or any term that mentions X (e.g. p(a, g(b,X))) would not be a correct substitution.

One of the key important aspects of substitution is the grounding of a clause. This is done by instantiating all the variables that appear in a clause with terms that do not include variables. A *ground literal* is a literal with no variable (i.e. all its terms are ground terms). A *ground clause* generated from a given non-ground clause by replacing all unground terms in the latter with ground terms is called *instance* of the (non-ground) clause.


Course: 304 Logic-Based Learning

Clausal Representation

- Conversion in CNF
  - Skolemisation $\exists X\ p(X) \Rightarrow p(c)$ new constant
    $\forall X\ \exists Y\ p(X,Y) \Rightarrow \forall X\ p(X,f(X))$
  - Remove universal quantifiers

$\forall X(\neg literate(X) \rightarrow (\neg write(X) \wedge \neg\exists Y(book(Y) \wedge read(X,Y))))$

$\forall X(literate(X) \vee (\neg write(X) \wedge \neg\exists Y(book(Y) \wedge read(X,Y))))$ eliminate →

$\forall X(literate(X) \vee (\neg write(X) \wedge \forall Y(\neg(book(Y) \wedge read(X,Y)))))$ push the ¬ inwards

$\forall X(literate(X) \vee (\neg write(X) \wedge \forall Y(\neg book(Y) \vee \neg read(X,Y))))$

$\forall X,Y(literate(X) \vee (\neg write(X) \wedge (\neg book(Y) \vee \neg read(X,Y))))$ remove ∀ quantifier

$literate(X) \vee (\neg write(X) \wedge (\neg book(Y) \vee \neg read(X,Y)))$

$(literate(X) \vee \neg write(X)) \wedge (literate(X) \vee \neg book(Y) \vee \neg read(X,Y))$ distribute ∨

$\neg write(X)) \vee literate(X)$
$\neg book(Y) \vee \neg read(X,Y) \vee literate(X)$

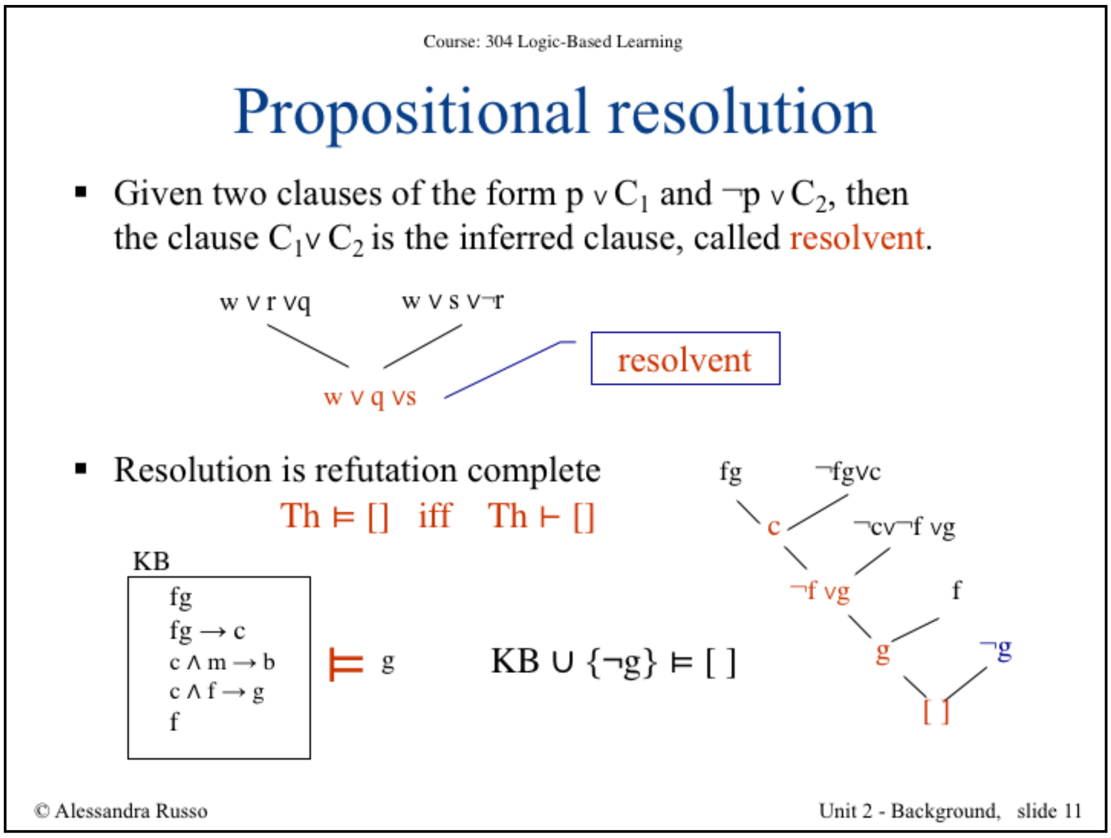
© Alessandra Russo Unit 2 - Background, slide 10

Two key steps are needed when transforming a first-order sentence into a clausal theory or CNF formula. As in the propositional case, implication should be changed into disjunction. Negation can then be pushed inwards using also the additional rules: $\neg\exists X p(X)$ is equivalent to $\forall X \neg\, p(X)$ and $\neg\forall X\ p(X)$ is equivalent to $\exists X \neg p(X)$. This steps helps eliminating existential quantifiers but at the same time may introduce also new existential quantifiers. Remember that the target clausal form assumes all variables to be universally quantified. So all remaining existentially quantified variables can be eliminated by introducing *skolem* term. When a subformula $\exists Y\ F[Y]$ exists in a given formula (note that here *F* can represent more than just a single predicate and the *Y* in square brackets denotes that the variable *Y* appears in the formula *F*), then the existential quantifier can be removed by introducing a Skolem term. Skolemisation works as follows: i) if the subformula is not in the scope of any universal quantifier then a new *skolem* constant name is introduced and all occurrences of Y are substituted with this new Skolem constant name, ii) if the subformula occurs within the scope of one or more universal quantifiers for variables X_1, X_2, .. X_n, then a Skolem term $sk(X_1, X_2, .. X_n)$ is introduced and all occurrences of Y are substituted with this new term.

(i.e. Note that the new Skolem constants (or Skolem terms) must not appear already anywhere else in the theory and essentially are assumed to extend the language (or signature) of the theory).

Once all existential quantifiers have been removed, all universal quantifiers can then be moved to the front of the sentence using the property that $(\forall X\ \varphi_1(X)) \wedge \varphi_2$, is equivalent to $\forall X(\ \varphi_1(X) \wedge \varphi_2)$ after appropriate variables renaming in case X already appears in φ_2.

Finally, ∨ can be distributed over ∧. An example is given in this slide.



Course: 304 Logic-Based Learning

Propositional resolution

- Given two clauses of the form $p \vee C_1$ and $\neg p \vee C_2$, then the clause $C_1 \vee C_2$ is the inferred clause, called resolvent.

w v r vq
w v s v¬r
w v q vs
resolvent

- Resolution is refutation complete

$$Th \vDash [] \quad \text{iff} \quad Th \vdash []$$

KB

fg
fg → c
c ∧ m → b
c ∧ f → g
f

$\vDash$ g

$$KB \cup \{\neg g\} \vDash [\,]$$

fg
¬fgvc
c
¬cv¬f vg
¬f vg
f
g
¬g
[ ]

© Alessandra Russo

Unit 2 - Background, slide 11

Once sentences are transformed into set of clauses, specific automated inference algorithms can be applied. One of the most well known inference algorithm is resolution. A resolution step is applied between two clauses that include respectively literals opposite in sign (i.e. a positive literal and its corresponding negative literal). The inference of a resolution step is the generation of a new clause given by the disjunction of all the literals that appear in the two given clauses with the exception of the literals opposite in sign. This new clause is called the *resolvent clause*.

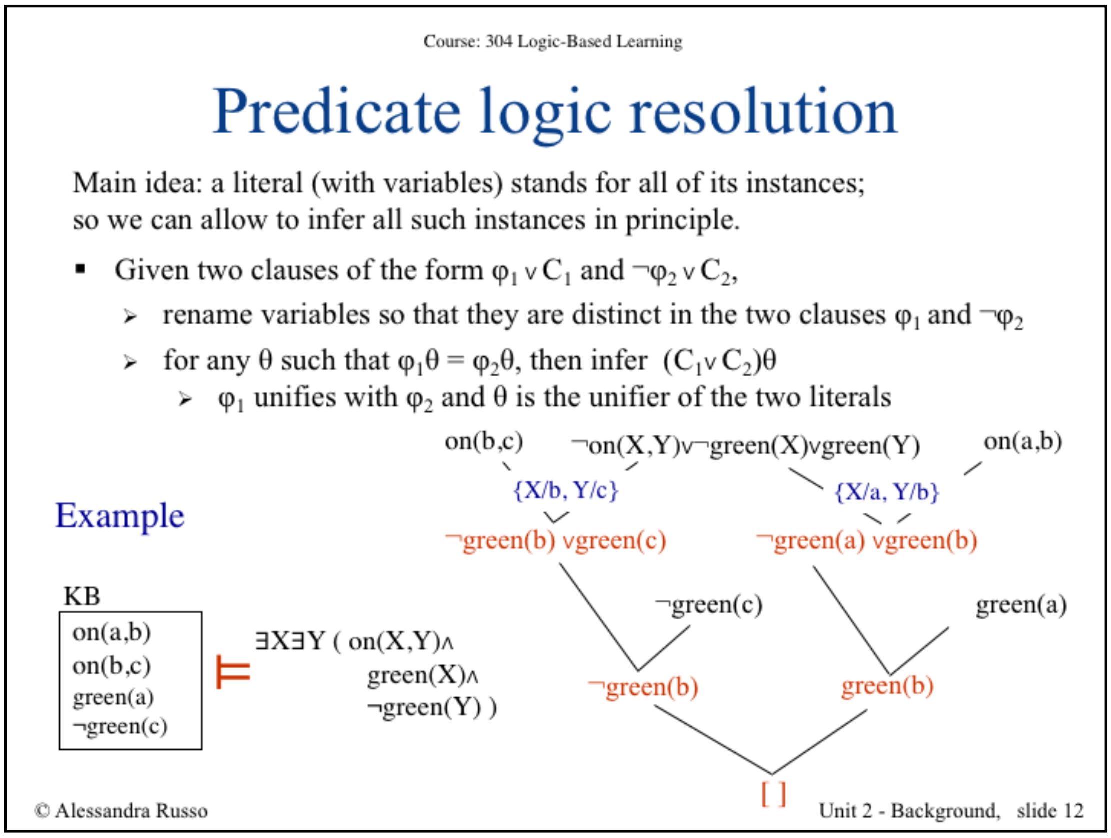
We have mentioned in previous slides that deductive inference is the sequential application of inference steps applied to given premises and/or conclusions generated from previous steps in the derivation. A derivation of a clause c from a given clausal theory Th is therefore a sequence of clauses $c_1, \ldots c_n$ where c_n is the clause c and every other clause c_i is either in Th or is the resolvent of two earlier clauses in the sequence. Resolution is a *sound* proof procedure, namely

$$Th \vdash c \text{ by resolution} \qquad \text{THEN} \qquad Th \vDash c$$

But the opposite does not hold directly, but it holds in a refutation way. For instance, consider the case of $p \vDash p \vee q$. Resolution cannot be directly applied to the given clausal theory Th = {p} and infer $p \vee q$. However if the problem is transformed into a refutation problem, resolution proof procedure can be applied. So resolution is both sound and refutation complete.

The above problem expressed as a refutation problem becomes as follows, where [ ] empty clause denotes *false*: $\{p\ ,\ \neg(p \vee q)\} \vDash [\,]$. We would have to show that the set of clauses $\{p, \neg p, \neg q\}$ can derive the empty clause [ ]. Note that as shown also in the above examples not all clauses need to be used. It is easy to see that this is indeed the case. So to use resolution the logical entailment problem has to be "rephrased" as a refutation problem first. This is also the basic principle underlying SLD refutation in Prolog, where the given query is in essence a negated goal. More to come later on in this lecture when we will recap the notion of SLD.

In the above slide an example of simple propositional resolution is given. Note that the given query (i.e. g) or clause that we want to prove is first negated and added to the existing set of clauses and the proof tries to derive the empty clause.



Course: 304 Logic-Based Learning

Predicate logic resolution

Main idea: a literal (with variables) stands for all of its instances; so we can allow to infer all such instances in principle.

- Given two clauses of the form $\varphi_1 \vee C_1$ and $\neg\varphi_2 \vee C_2$,
  - rename variables so that they are distinct in the two clauses φ_1 and $\neg\varphi_2$
  - for any θ such that $\varphi_1\theta = \varphi_2\theta$, then infer $(C_1 \vee C_2)\theta$
    - φ_1 unifies with φ_2 and θ is the unifier of the two literals

Example

KB

on(a,b)
on(b,c)
green(a)
¬green(c)

$\models \exists X \exists Y\ (\text{on}(X,Y) \wedge \text{green}(X) \wedge \neg\text{green}(Y))$

on(b,c) ¬on(X,Y)∨¬green(X)∨green(Y) on(a,b)

{X/b, Y/c} {X/a, Y/b}

¬green(b) ∨green(c) ¬green(a) ∨green(b)

¬green(c) green(a)

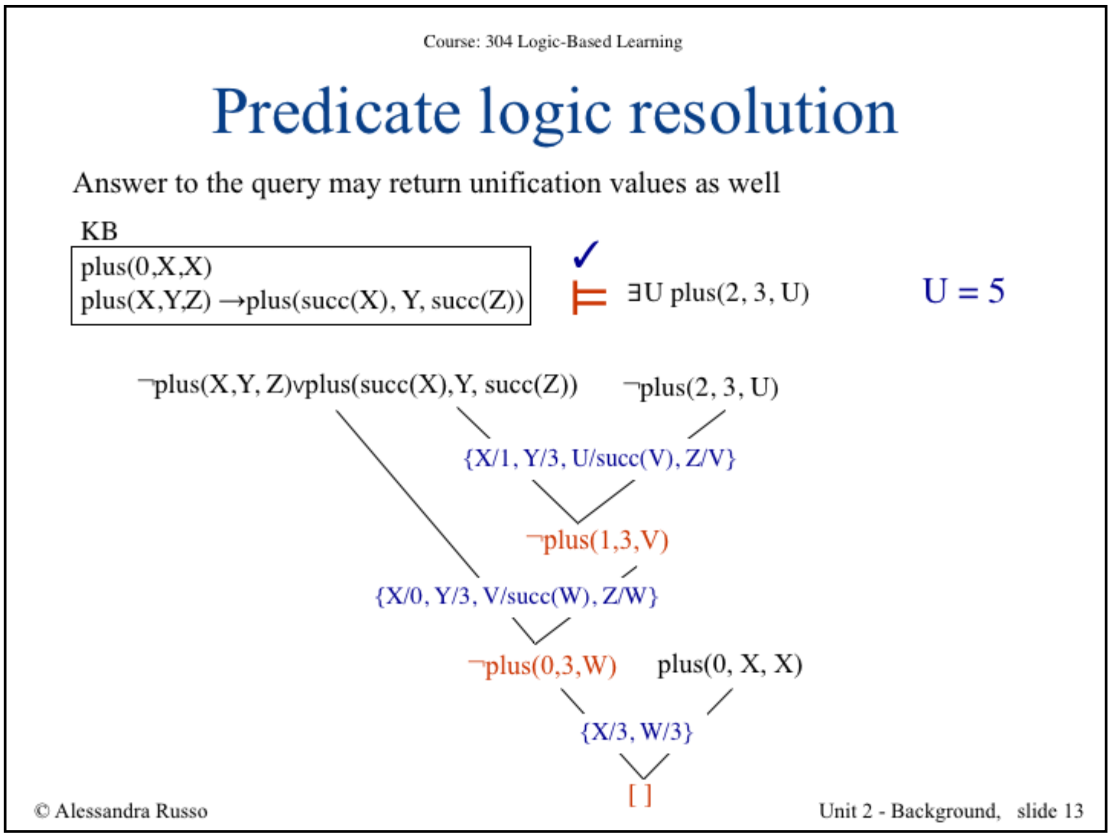
¬green(b) green(b)

[ ]

© Alessandra Russo Unit 2 - Background, slide 12

Resolution in first-order logic is more complex. It is still based on the idea of “resolving” opposite literals that appear in two clauses but variables play an important role here. Literals may have clearly unground terms. In this case they are understood as standing for all possible instances so in principle the resolution could happen by referring to any of such instances. Identifying which instance to use is the role of the unification step. Firstly, all variables occurring in the two clauses to resolve should be renamed with unused variables to avoid name clashes. Then considering two opposite literals in the two clauses, if it is possible to find a substitution that applied to both these two literals makes them equal, then the resolution step can be applied and the substitution has to be systematically applied to occurrences of those variables in the literals left in the resolvent. For instance, in the example above, the left most literal on(b,c) can resolve with the opposite literal ¬on(X,Y) in the top middle clause by means of the substitution {X/b, and Y/c}. These two literals are then said to “unify” under this substitution and the substitution is applied to the rest of the literals in the resolvent clause. This explains the resolvent ¬green(b) ∨ green(c). And so on.

Note that in this example of resolution the given query has been negated, transformed into a clause and added to the given set of clauses (KB). The negation has had the effect of eliminating the existential quantifiers, hence no skolemisation has been needed.



Predicate logic resolution

Answer to the query may return unification values as well

KB

plus(0,X,X)
plus(X,Y,Z) →plus(succ(X), Y, succ(Z))

✓ $\models$ ∃U plus(2, 3, U) U = 5

¬plus(X,Y, Z)vplus(succ(X),Y, succ(Z))
¬plus(2, 3, U)
{X/1, Y/3, U/succ(V), Z/V}
¬plus(1,3,V)
{X/0, Y/3, V/succ(W), Z/W}
¬plus(0,3,W)
plus(0, X, X)
{X/3, W/3}
[ ]

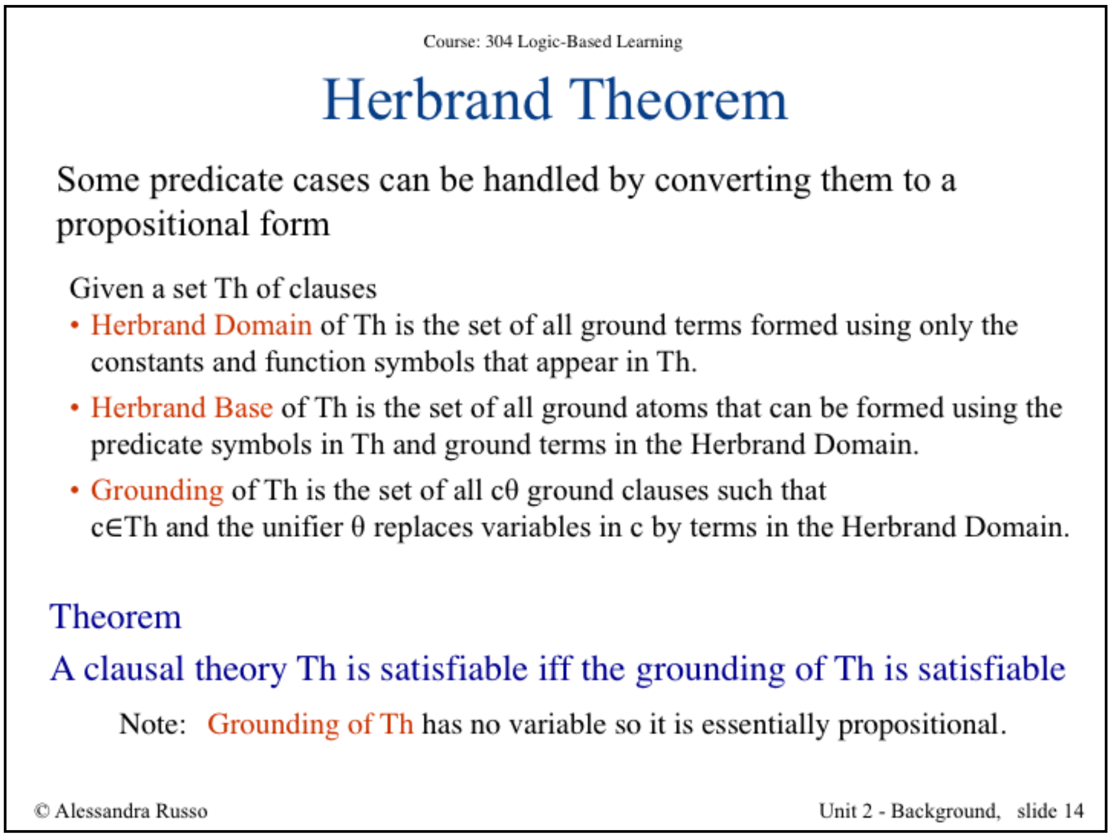
The previous example has shown that the resolution mechanism can allow us to answer existential types of questions. But what about if we actually want to find out particular values for which our query is derivable from a given set of clauses? The resolution process uses unification and this same unification constructs during the proof a substitution function that defines the value of given variables in the query that have led to a successful proof.

Consider the above example. In this case, the query is negated and transformed into a clause with variable U universally quantified. The blue labels in the derivation show the unification applied to each resolution step. Since U = succ(V), and V= succ(W) and W = 3, the three unifications give U = succ(succ(3)) which is indeed 5.

Similar principles happen in SLD resolution in Prolog when queries with variables are posed to a Prolog program and the Prolog returns yes with the value of the variables that appear in the query.

However, it is possible to construct resolutions that never terminate. This is often the case when function symbols appear in the given clauses and the resolution process is applied in a kind of "depth-first" manner.

To tackle this problem a possible way for making resolution more efficient is to use the notion of Most General Unifier (MGU). This is the most general unifier that can be constructed between two given clauses so that any other unifier between these clauses can be generated by concatenating another unifier function to the MGU. But formal notions and properties about MGU go beyond the content of this course.



Course: 304 Logic-Based Learning

Herbrand Theorem

Some predicate cases can be handled by converting them to a propositional form

Given a set Th of clauses

- Herbrand Domain of Th is the set of all ground terms formed using only the constants and function symbols that appear in Th.
- Herbrand Base of Th is the set of all ground atoms that can be formed using the predicate symbols in Th and ground terms in the Herbrand Domain.
- Grounding of Th is the set of all $c\theta$ ground clauses such that $c \in Th$ and the unifier θ replaces variables in c by terms in the Herbrand Domain.

Theorem

A clausal theory Th is satisfiable iff the grounding of Th is satisfiable

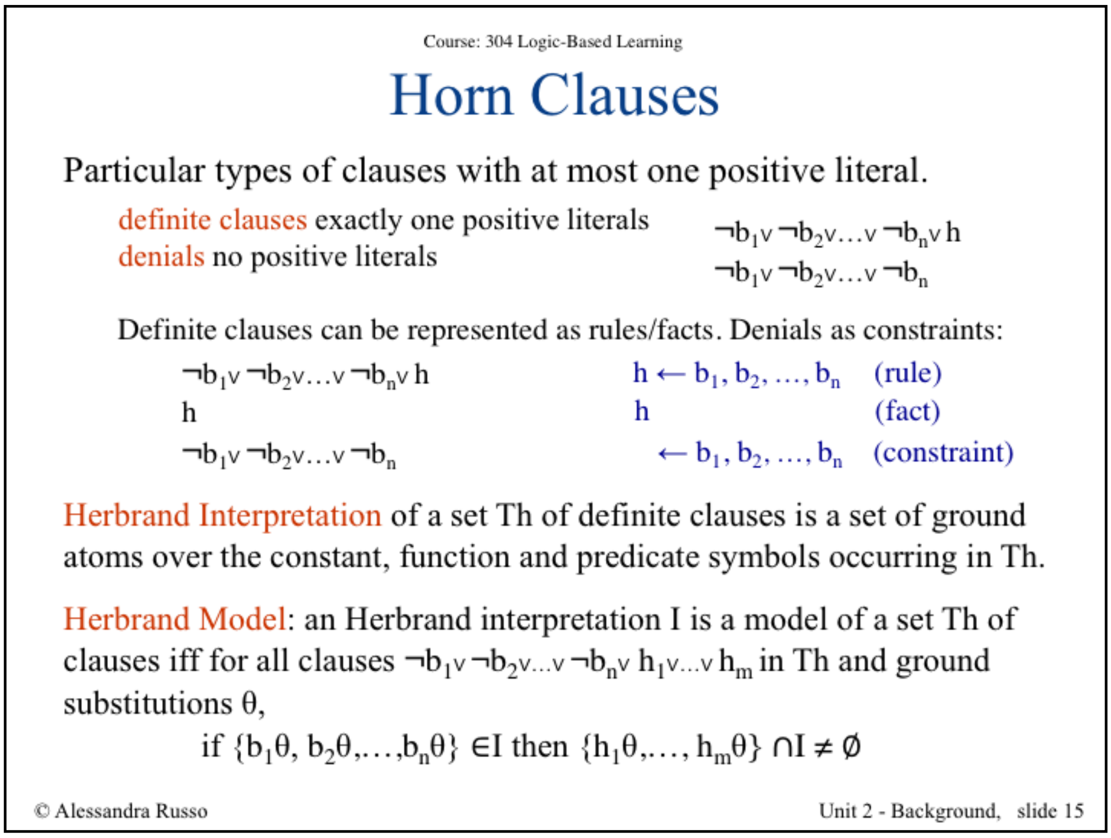
Note: Grounding of Th has no variable so it is essentially propositional.

© Alessandra Russo Unit 2 - Background, slide 14

We have recap so far the notion of propositional and predicate logic, and presented basic concepts of clausal logic. We have seen the notion of resolution as an example of inference mechanism that is applicable to clausal logic both in the case of propositional and first-order logic. Semantically, the notion of interpretation for clausal logic is the same as that for first-order logic. Any interpretation can be constructed for a given set of clauses. But results have shown that it is sufficient to consider a special class of interpretations, called *Herbrand interpretations*. An Herbrand interpretation uses the notion of *Herbrand Domain* as its universe of discourse. An Herbrand Domain is a set of all ground terms that can be constructed using the constants and function symbols that occur in the set of clauses considered.

The *Herbrand base* of a given set Th is instead the set of ALL ground atoms that can be formed from the predicate symbols in Th and terms in the Herbrand Domain. The grounding of Th, denoted *ground(Th)*, is the set of all ground instances of the clauses in Th that can be constructured using the terms in the Herbrand Domain. This notion of grounding will be revisited later in the course when we will be talking about Answer Set Programming (ASP). It is particularly useful because it reduces a first-order representation to a representation that is essentially propositional. Results have also shown that a clausal theory is satisfiable if and only if its grounding is satisfiable.

Existing solvers can be used to determine the satisfiability of a set of (ground) clauses, and these seem to work better than resolution-based proof strategy. The ASP solver Clingo, which you will be using later in the course, is one of such solvers.


Course: 304 Logic-Based Learning

Horn Clauses

Particular types of clauses with at most one positive literal.

definite clauses exactly one positive literals $\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n \vee h$
denials no positive literals $\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n$

Definite clauses can be represented as rules/facts. Denials as constraints:

$\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n \vee h$	$h \leftarrow b_1, b_2, \ldots, b_n$	(rule)
h	h	(fact)
$\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n$	$\leftarrow b_1, b_2, \ldots, b_n$	(constraint)

Herbrand Interpretation of a set Th of definite clauses is a set of ground atoms over the constant, function and predicate symbols occurring in Th.

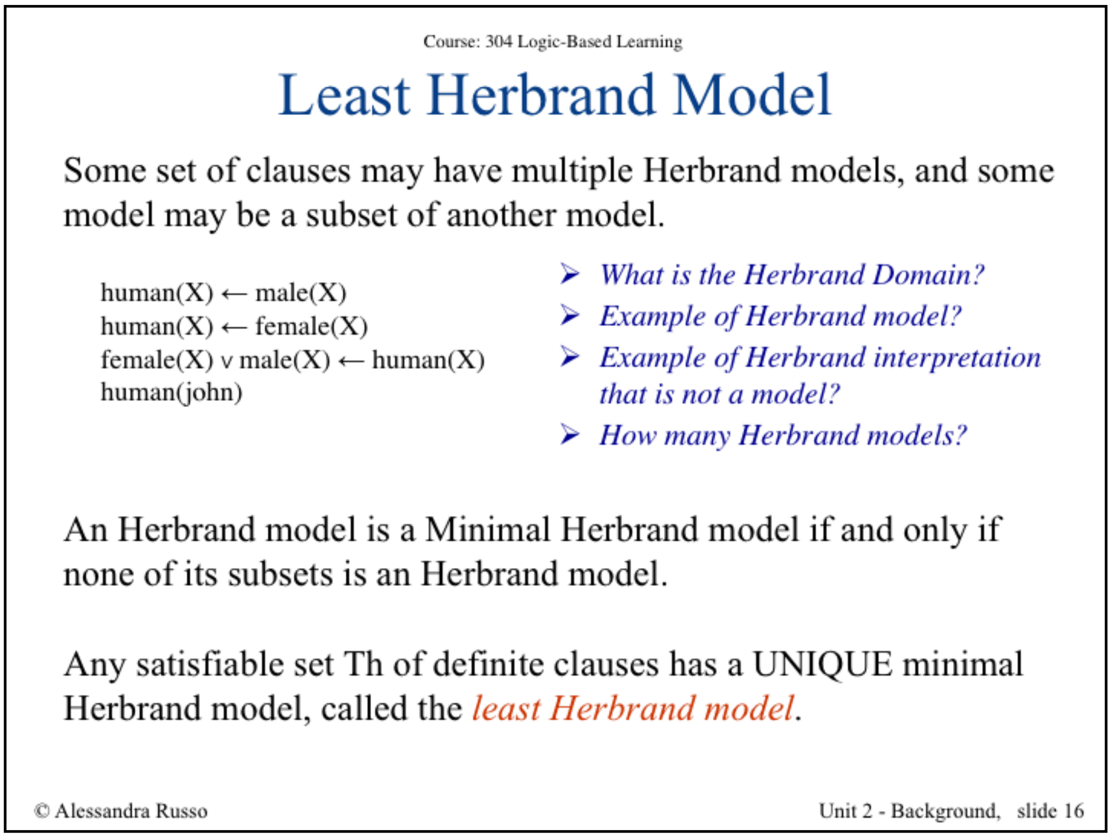
Herbrand Model: an Herbrand interpretation I is a model of a set Th of clauses iff for all clauses $\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n \vee h_1 \vee \ldots \vee h_m$ in Th and ground substitutions θ,

$$\text{if } \{b_1\theta, b_2\theta, \ldots, b_n\theta\} \in I \text{ then } \{h_1\theta, \ldots, h_m\theta\} \cap I \neq \emptyset$$

© Alessandra Russo Unit 2 - Background, slide 15

We have seen so far how clausal representation is supported by inference mechanisms that are refutation complete. The notion of clausal theory and resolution is at the heart of logic programming and Prolog, in particular what is known as "pure" Prolog. Declarative programs are written using a special types of clauses, called *definite clauses* and queries posed to a Prolog program are assumed to be *denials*. A definite clause is a clause with exactly one positive literal. Denials are instead clauses with zero positive literals. Denials are also referred to as constraints. We will see in the next few slides that in an abductive inference, constraints play a more general role than that of a query. A definite clause can be read as a *rule* of the form shown in the slide. The positive literal is the head of the rule and the atoms of the negative literals form the body of the rule. Definite clauses composed of just the positive literal are also called *facts* (rules with empty body). Constraints can also be understood as special rules with *false* as head of the rule. We will use the notation shown in blue from now to represent definite clauses and denials.

In the previous slide we have mentioned, but not defined, the notion of Herbrand interpretation. Let's define it now for Horn clauses. Given a set of definite clauses Th, an **Herbrand interpretation** of Th is a set of ground atoms constructed from the Herbrand Domain and the predicate symbols that appear in the clauses. Essentially it is a SUBSET of the Herbrand Based of Th. The assumption is that ground atoms that appear in an Herbrand interpretation I are true, and all the other ground atoms from the Herbrand Based of Th that are not in the interpretation I are assumed to be false. An Herbrand interpretation I is an **Herbrand model** of a clause C, written as $\neg b_1 \vee \neg b_2 \vee \ldots \vee \neg b_n \vee h_1 \vee \ldots \vee h_m$, if for all substitutions θ for which $b_1\theta$, …. $b_n\theta$ are true in I at least one of $h_1\theta$, …. $h_m\theta$ has to be true in I. It is easy to see how this definition captures the notion of satisfiability of rules, facts and denials as special types of clauses. In the case of rules, the clauses will have m=1 (just one positive literal), and an Herbrand interpretation I is a model of a rule if for all substitutions θ for which *body*(C)θ is true in I the head *head*(C)θ has to be true in I. In the case of a fact, this definition also applies and essentially gives that I is a model of a fact if the fact is true in I. In the case of denials, all substitutions θ for which *body*(C)θ is true in I the *head*(C)θ CANNOT be true in I, since the head is assumed to be the *false* value. Hence an Herbrand interpretation is a model of a denials if it is the case that for every substitution θ we have that *body*(C)θ is not true in I. We simply say that the body of the constraint should never be satisfied in an Herbrand model.


Course: 304 Logic-Based Learning

Least Herbrand Model

Some set of clauses may have multiple Herbrand models, and some model may be a subset of another model.

human(X) ← male(X)
human(X) ← female(X)
female(X) v male(X) ← human(X)
human(john)

- *What is the Herbrand Domain?*
- *Example of Herbrand model?*
- *Example of Herbrand interpretation that is not a model?*
- *How many Herbrand models?*

An Herbrand model is a Minimal Herbrand model if and only if none of its subsets is an Herbrand model.

Any satisfiable set Th of definite clauses has a UNIQUE minimal Herbrand model, called the *least Herbrand model*.

© Alessandra Russo

Unit 2 - Background, slide 16

Given a set of clauses, there can be no Herbrand model (i.e., the set is unsatisfiable) or only one or many Herbrand models. In Chapter 2 of Luc De Raedt textbook you will find an algorithm for computing Herbrand models of a given set of clauses. Note that such an algorithm may not terminate. This is because the set of clauses may accept an infinite Herbrand model. A typical example would be if we consider the clauses:

nat(0)

nat(succ(X)) ← nat(X)

The above set of clauses accepts an infinite Herbrand model, as the Herbrand domain is infinite.

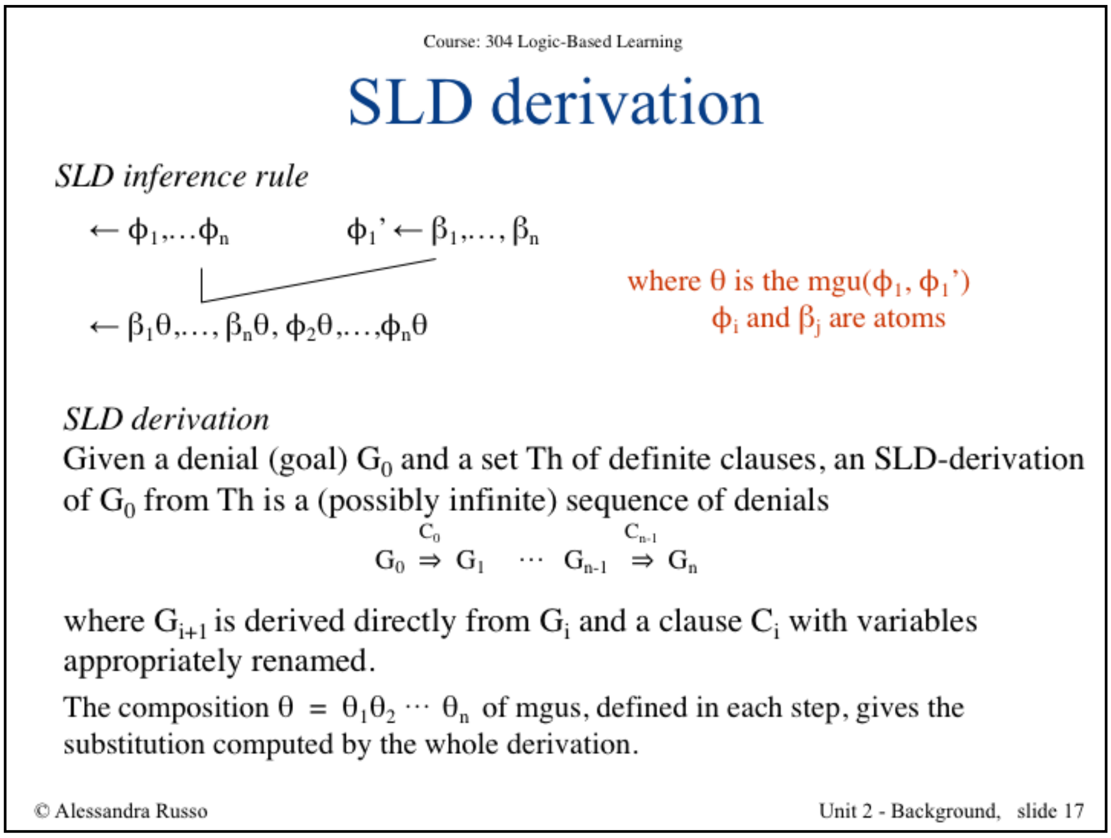
In cases when the algorithm terminates, but it fails to construct an Herbrand model, we say that the given set of clauses is *unsatisfiable*. The underlying theoretical result, similar to the theorem shown in Slide 14, is that a given set Th of clauses has a model (i.e. is satisfiable) if and only if it has an Herbrand model. This property is very useful when we want to compute the logical entailment of clause from a given set Th of clauses (i.e., Th ⊨ c). Using the same notion of refutation, that we have seen in the previous slides, this task can be computed by showing that the attempt to construct an Herbrand model for Th ∪{¬c} fails.

A set Th of clauses may have more than one Herbrand model. Some of these models may be subset of other models. An example is given in the slide. Answers to the above questions are discussed in class.

So what is relevant to compute is always the minimal Herbrand models. A set of clauses may have more than one minimal model, as it is the case for the example given in this slide.

But, theoretical results have shown that when we restrict our attention to only definite clauses, there is only one unique minimal Herbrand model. This is referred to as the *Least Herbrand Model.*

The least Herbrand model is an important concept because it captures the semantics of the set of definite clauses. In essence the least Herbrand model of a definite program consists of all ground atoms that are logically entailed by the program.



Course: 304 Logic-Based Learning

SLD derivation

SLD inference rule

$\leftarrow \phi_1,\ldots\phi_n \qquad \phi_1' \leftarrow \beta_1,\ldots,\beta_n$

$\leftarrow \beta_1\theta,\ldots,\beta_n\theta, \phi_2\theta,\ldots,\phi_n\theta$

where θ is the mgu(ϕ_1, ϕ_1')
ϕ_i and β_j are atoms

SLD derivation

Given a denial (goal) G_0 and a set Th of definite clauses, an SLD-derivation of G_0 from Th is a (possibly infinite) sequence of denials

$$G_0 \overset{C_0}{\Rightarrow} G_1 \quad \cdots \quad G_{n-1} \overset{C_{n-1}}{\Rightarrow} G_n$$

where G_{i+1} is derived directly from G_i and a clause C_i with variables appropriately renamed.

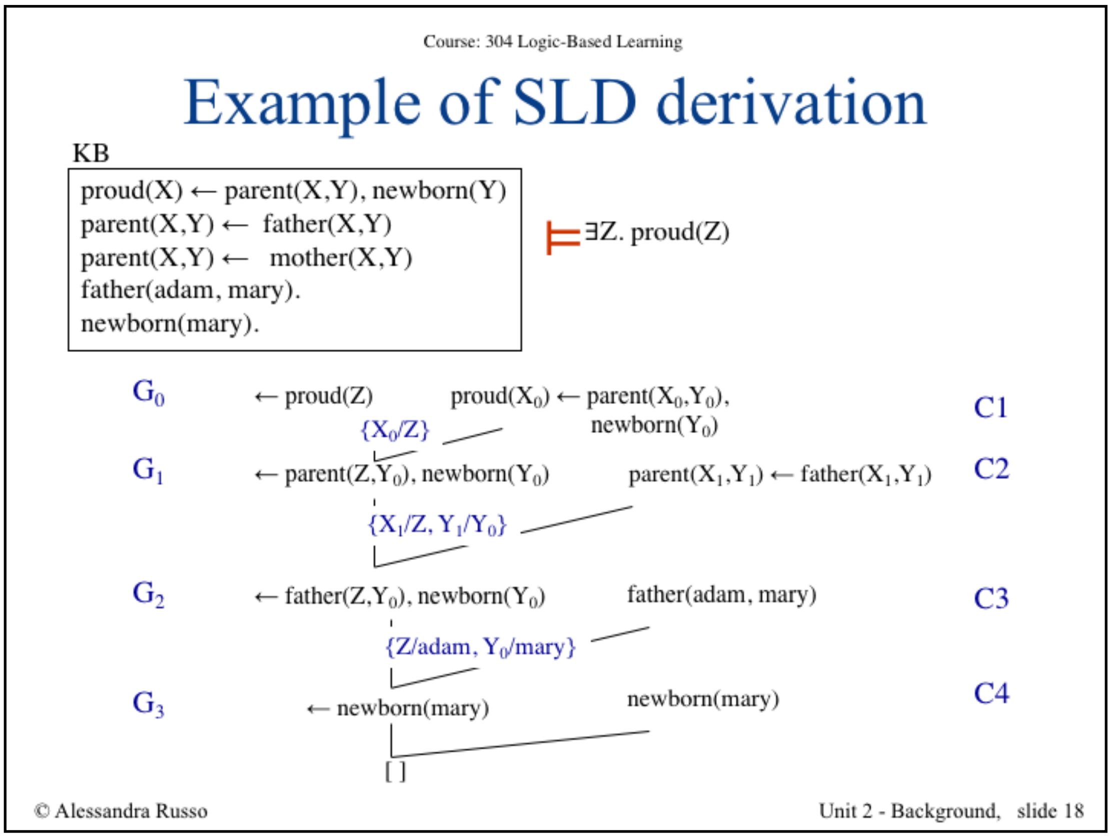
The composition $\theta = \theta_1\theta_2 \cdots \theta_n$ of mgus, defined in each step, gives the substitution computed by the whole derivation.

© Alessandra Russo

Unit 2 - Background, slide 17

All the logical concepts introduced so far for propositional, predicate logic and clausal theory are at the basis of pure "Prolog" declarative programming approach. A pure Prolog program, as defined in Luc De Raedt's textbook, is a set of definite clauses and therefore its semantics is given by its Least Herbrand Model. The inference procedure used by Prolog is a specialized version of resolution that exploits the fact that the given theory is a set of definite clauses and not general clauses.

To start with, the query of a Prolog program is not an arbitrary clause (that we might want to prove to be derivable from our theory) but a denial clause. Arbitrary clauses have to be rewritten in a certain way in order to accommodate this syntactic restriction of Prolog. Given a set of definite clauses and a denial there can be many resolution proofs that can be constructed. But to systematically explore all possible derivations, Prolog uses a special form of resolution called *SLD-resolution*. The SLD derivation is a linear sequence of application of an SLD inference rule. The SLD inference rule is applied between a denial clause and a definite clause in the original set Th of clauses. In the first application of this rule the denial clause is the given query, expressed in denial form (remember SLD is still a resolution proof method and as such is still refutation based). Each subsequent denial clause is generated by resolution between the previously derived denial and a definite clause in the given theory Th. This is formally defined in the slide above. Moreover, more than one atom in the given (or inferred) denial may unify with existing clauses, so by convention SLD uses a selection rule that picks always the left most atom in a denial before applying the resolution inference. The name SLD stays for *Linear Resolution for Definite clauses with Selection function.*


Course: 304 Logic-Based Learning

Example of SLD derivation

KB

$proud(X) \leftarrow parent(X,Y), newborn(Y)$
$parent(X,Y) \leftarrow father(X,Y)$
$parent(X,Y) \leftarrow mother(X,Y)$
$father(adam, mary).$
$newborn(mary).$

$\models \exists Z.\ proud(Z)$

G_0 $\leftarrow proud(Z)$ $proud(X_0) \leftarrow parent(X_0,Y_0), newborn(Y_0)$ C1

$\{X_0/Z\}$

G_1 $\leftarrow parent(Z,Y_0), newborn(Y_0)$ $parent(X_1,Y_1) \leftarrow father(X_1,Y_1)$ C2

$\{X_1/Z, Y_1/Y_0\}$

G_2 $\leftarrow father(Z,Y_0), newborn(Y_0)$ $father(adam, mary)$ C3

$\{Z/adam, Y_0/mary\}$

G_3 $\leftarrow newborn(mary)$ $newborn(mary)$ C4

[ ]

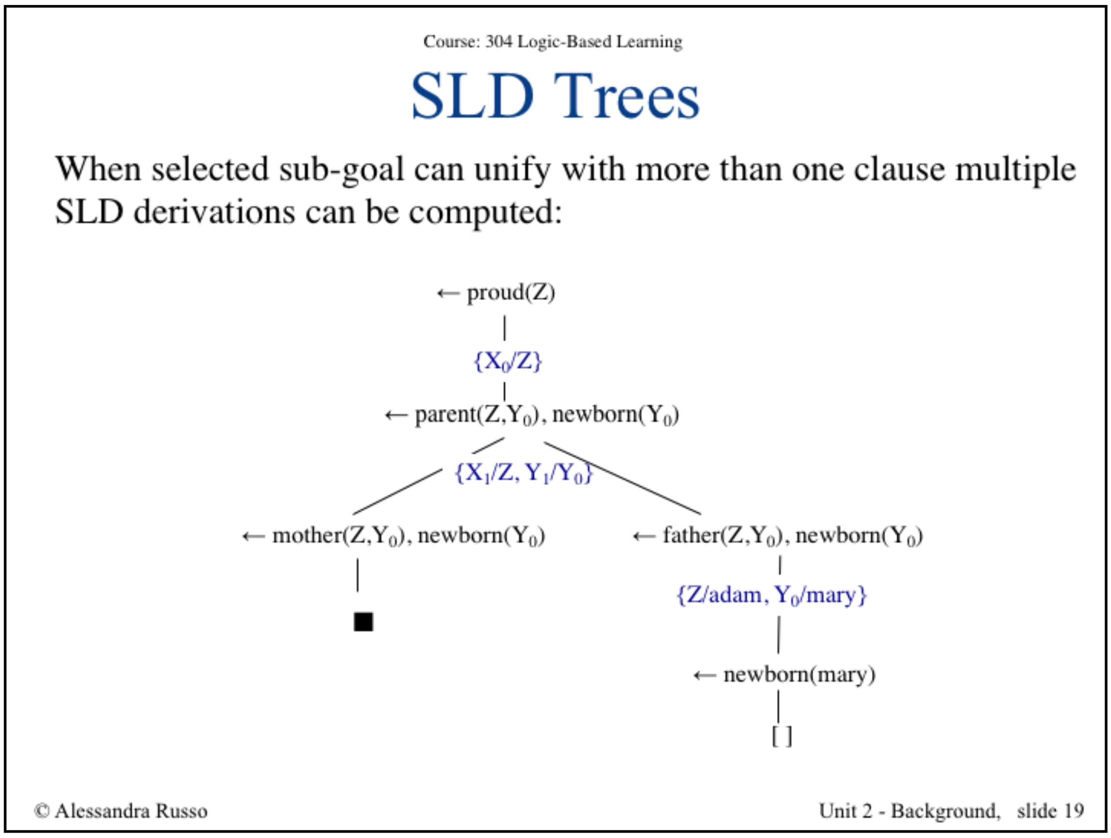
© Alessandra Russo Unit 2 - Background, slide 18

The above is a simple refutation example. Note that the background program describes only "positive knowledge" — it does not state who is not proud. Nor does it convey what it means for someone not to be a parent. The problem of expressing negative knowledge will be investigated in subsequent lectures of this course. As part of this lecture we will briefly show how SLD resolution copes with set of rules that include negation as failure. Later in the course you will learn about semantics of theories with negation as failure.

A sequence of G_0, G_1,…,G_{n+1} goals that terminates with an empty clause is a successful SLD derivation. Different Horn clauses can be chosen during the derivation. Different choices would lead to different SLD derivations. Some of these will also not be successful derivations. An SLD derivation fails when the selected sub-goal cannot be unified with any clause and therefore the sequence cannot be extended further.

So, by definition, an SLD derivation of a goal G_0 is a failed derivation if the last element in the derivation is not the empty clause and it cannot be resolved with any other clause given in the initial theory (KB).

An example of failed derivation could be constructed for the same goal and KB given in this slide, if the third clause in the KB is chosen as clause C_2 instead. In this case the next subgoal G_2 would be the denial $\leftarrow mother(Z,Y_0), newborn(Y_0)$. The selected literal $mother(Z,Y_0)$ would in this case not be unifiable with any other clause in the KB.



Course: 304 Logic-Based Learning

SLD Trees

When selected sub-goal can unify with more than one clause multiple SLD derivations can be computed:

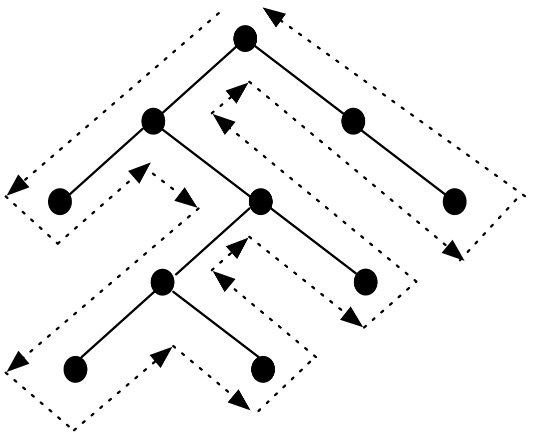
$$\begin{tikzcd}[column sep=small]
 & \leftarrow proud(Z) \arrow[d, dash, "\{X_0/Z\}"] & \\
 & \leftarrow parent(Z,Y_0), newborn(Y_0) \arrow[dl, dash, "\{X_1/Z, Y_1/Y_0\}"'] \arrow[dr, dash] & \\
\leftarrow mother(Z,Y_0), newborn(Y_0) \arrow[d, dash] & & \leftarrow father(Z,Y_0), newborn(Y_0) \arrow[d, dash, "\{Z/adam, Y_0/mary\}"] \\
\blacksquare & & \leftarrow newborn(mary) \arrow[d, dash] \\
 & & [\,]
\end{tikzcd}$$

© Alessandra Russo

Unit 2 - Background, slide 19

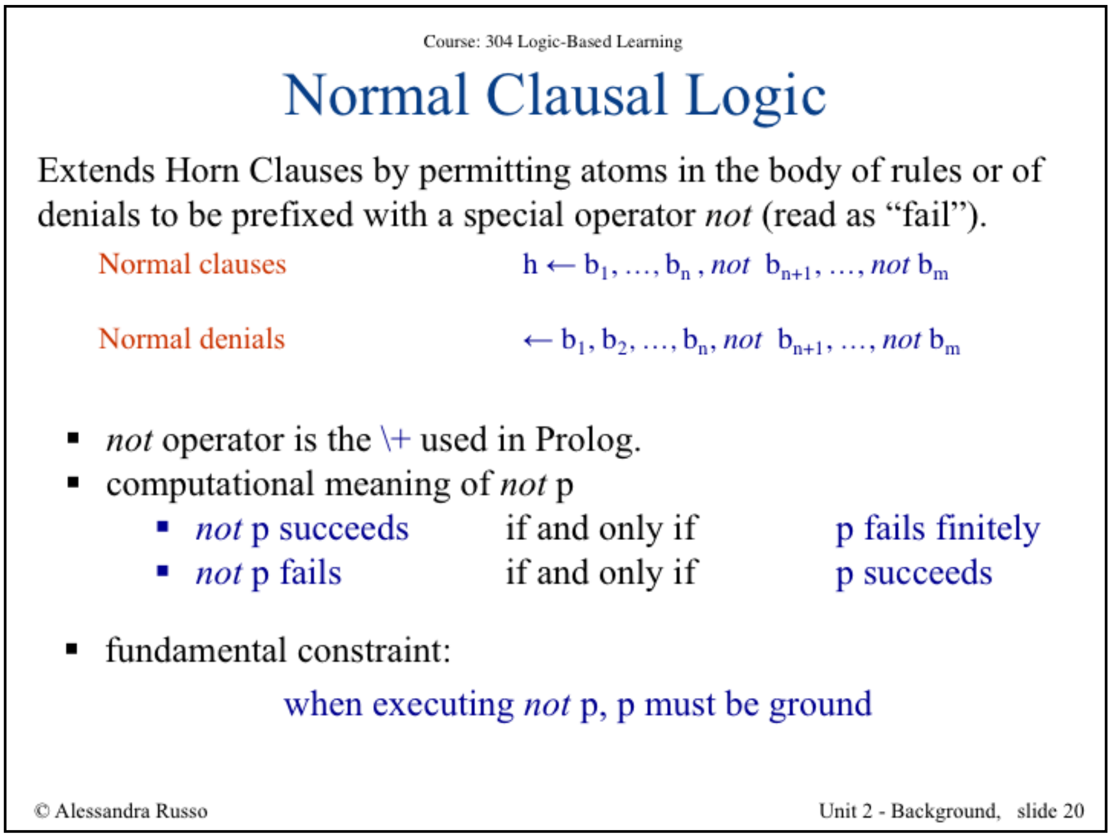
If the selected sub-goals may be resolved with more than one given clause then different derivations can be constructed. All such derivations may be represented by a (possibly infinite) SLD tree of a given goal G_0. An example of SLD tree is given in the slide. Note that only the generated sub-goals are shown in this SLD tree as it is assumed that we are using an SLD-resolution and that each generated sub-goal is unified with an existing clause in the given KB.

An SLD tree essentially represents the search space of the derivations. Each branch of the tree is an SLD computation (or SLD refutation). Generating one or all (where possible) SLD computations means performing a systematic search over the search tree of such refutation. Existing Prolog systems often exploit some ordering on the give program clauses (i.e. KB), e.g. the ordering in which the clauses are written in the program, from top to bottom. This imposes the ordering on the edges descending from a node of the SLD-tree. The tree is then traversed in a **depth-first manner** following this ordering. For a finite SLD-tree this strategy is complete. Whenever a leaf node of the SLD-tree is reached the traversal continues by backtracking to the last preceding node of the path with unexplored branches (see figure below). If the empty clause is reached the answer substitution of the completed refutation is returned.



Notice, however, that an SLD tree may be infinite. This is why sometimes Prolog computation do not terminate.

Can you think of an example where the tree is infinite (e.g. it has an infinite branch)?


Course: 304 Logic-Based Learning

Normal Clausal Logic

Extends Horn Clauses by permitting atoms in the body of rules or of denials to be prefixed with a special operator *not* (read as "fail").

Normal clauses	$h \leftarrow b_1, \ldots, b_n, \textit{not}\ b_{n+1}, \ldots, \textit{not}\ b_m$
Normal denials	$\leftarrow b_1, b_2, \ldots, b_n, \textit{not}\ b_{n+1}, \ldots, \textit{not}\ b_m$

- *not* operator is the \+ used in Prolog.
- computational meaning of *not* p
  - *not* p succeeds if and only if p fails finitely
  - *not* p fails if and only if p succeeds
- fundamental constraint:

  when executing *not* p, p must be ground

© Alessandra Russo Unit 2 - Background, slide 20

One of the most common representation of computational logic is Horn clauses. However, for those of you familiar with Prolog programming language, Prolog programs often include a special operator, denoted as "\+". This is a "fail" operator also called *negation by failure*.

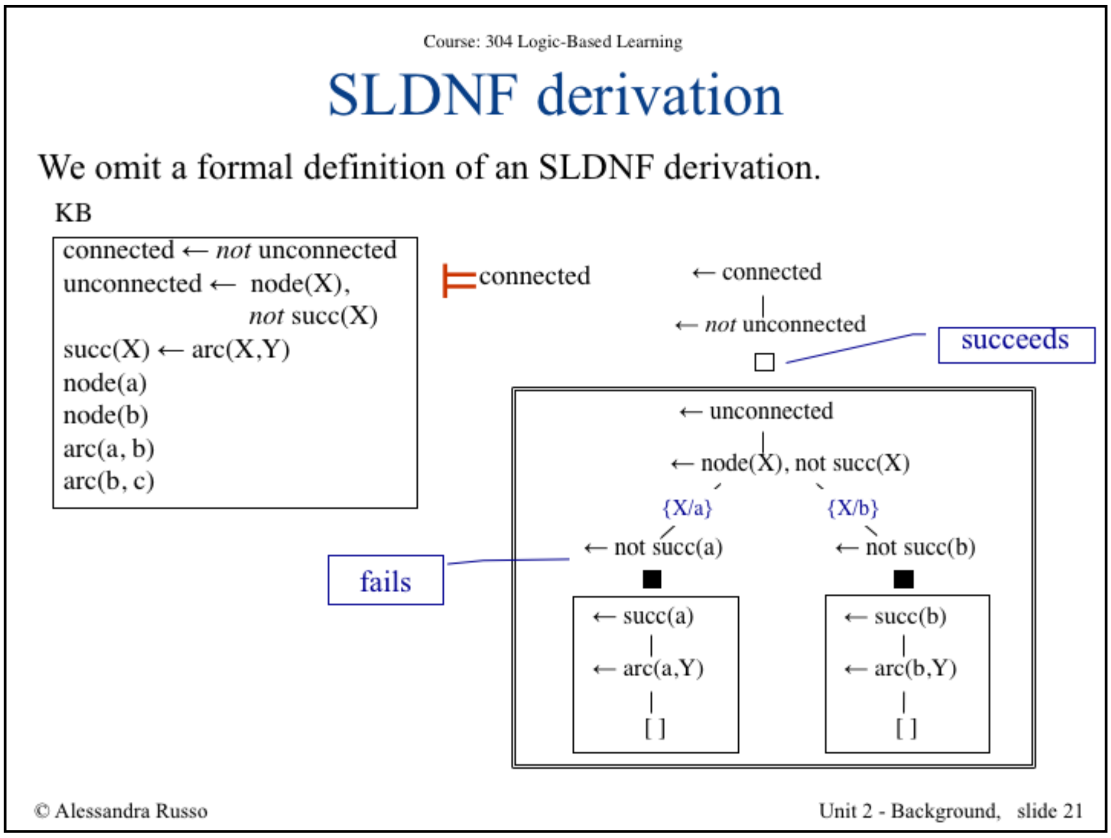
A literal "*not* p" can in fact be read as "fail to prove p". The name "negation by failure" is related to semantics of this operator, which in turns is related to the notion of Clark Completion semantics. Understanding Clark Completion is outside the scope of this course and you might learn more about it in the Knowledge Representation course in the forth year. But, in very simple terms, semantically "*not* p" means "assuming **not p** to be true in a model of a given program because the program fails to prove p". This is intuitively related to the notion of Least Herbrand Model and the property that everything that is provable from a program is true in the model of the program. So failing to prove p means stating that p is false and therefore that the negation of p is true (but omitted) in the model of the program.

Atoms prefixed with the negation by failure operator can also appear in denials. We refer to these as normal denials. But you can immediately notice that if in a denial, the index n=0 and the index m=1, we could have a denial of the form ← not p(X) (as the atoms in the clauses may be predicates). **This type of expression cannot be evaluated if the atom p(X) is not ground. When we reach in the derivation process a query of the form ← not p(X) the variable X should have already be grounded in the previous steps of the derivation.**

An important constraint is therefore that when evaluating ← not p, the atom p must be ground.

If this is not the case then we say that the derivation "floundered", meaning does not know the answer. In the next slide we show an example that illustrates this.

Note that in normal clausal logic the fail operator never appears in the head of a rule, as this would correspond asking to prove what should not be proved, whereas clausal theories define what should be provable!


Course: 304 Logic-Based Learning

SLDNF derivation

We omit a formal definition of an SLDNF derivation.

KB

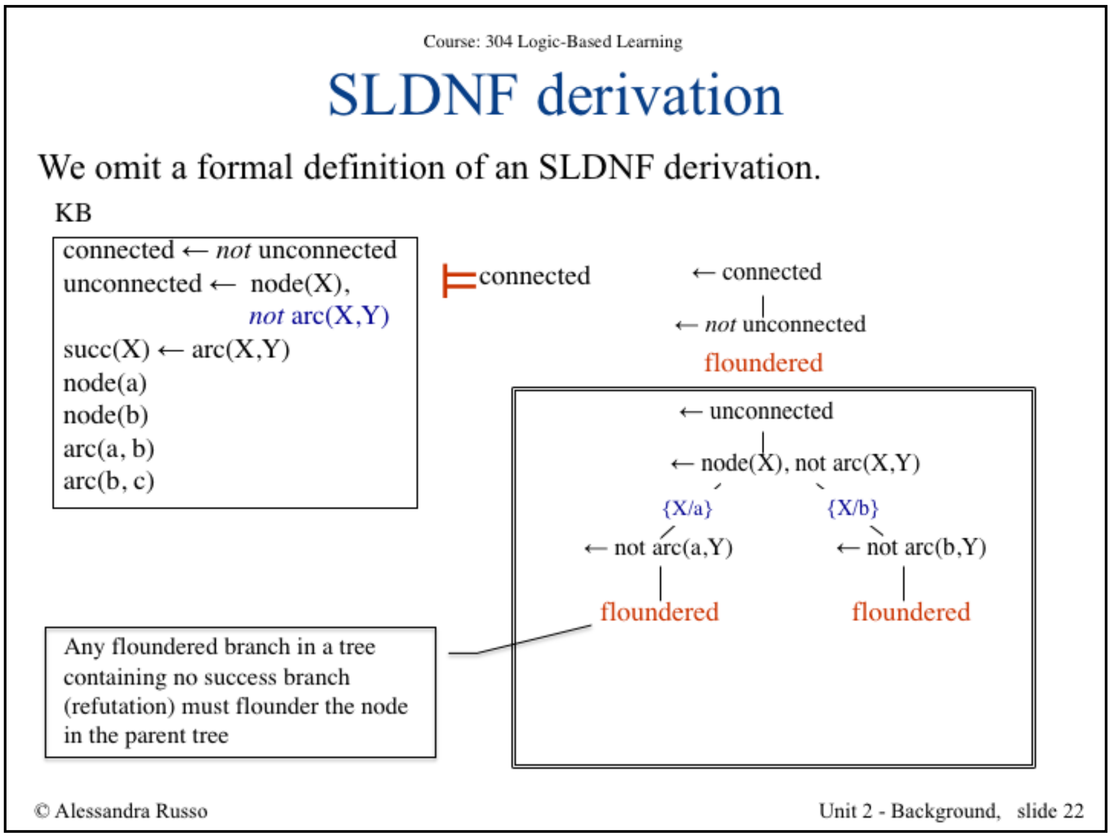
```
connected ← not unconnected
unconnected ← node(X),
              not succ(X)
succ(X) ← arc(X,Y)
node(a)
node(b)
arc(a, b)
arc(b, c)
```

© Alessandra Russo

Unit 2 - Background, slide 21

In the above example derivation to evaluate the goal, the second denial sub-goal ← *not unconnected* need to be proved. A new derivation is created. This is indicated with the double lined box. The goal of this new derivation is to see if it is possible to prove unconnected, hence the denial subgoal ← unconnected. Now for the goal ← not unconnected to succeed this, derivation process has to finish with failure all all possible branches, that is all possible ways of proving ← unconnected has to fail.

The subgoal ← node(X), not succ(X) has two possible unifications because there are 2 facts about node in the KB. In each derivation branch a fail subgoal is generated. Consider the left branch. The fail subgoal ← not succ(a) is now ground. So this means that it is possible to start a new derivation with the new goal ← succ(a) to see whether this is provable. In this case this subgoal is provable so the parent denial goal fail ← not succ(a) fails (denoted with the black square). Similarly for the left branch. Now every possible derivation of the preceeding subgoal ← unconnected has failed. So it can be concluded that the derivation of the parent denial subgoal ← *not* unconnected succeeds (denoted with the empty square). Hence the initial query succeeds.


Course: 304 Logic-Based Learning

SLDNF derivation

We omit a formal definition of an SLDNF derivation.

KB

```
connected ← not unconnected
unconnected ← node(X),
              not arc(X,Y)
succ(X) ← arc(X,Y)
node(a)
node(b)
arc(a, b)
arc(b, c)
```

⊨connected
← connected
← not unconnected
floundered
← unconnected
← node(X), not arc(X,Y)
{X/a}
{X/b}
← not arc(a,Y)
← not arc(b,Y)
floundered
floundered

Any floundered branch in a tree containing no success branch (refutation) must flounder the node in the parent tree

© Alessandra Russo

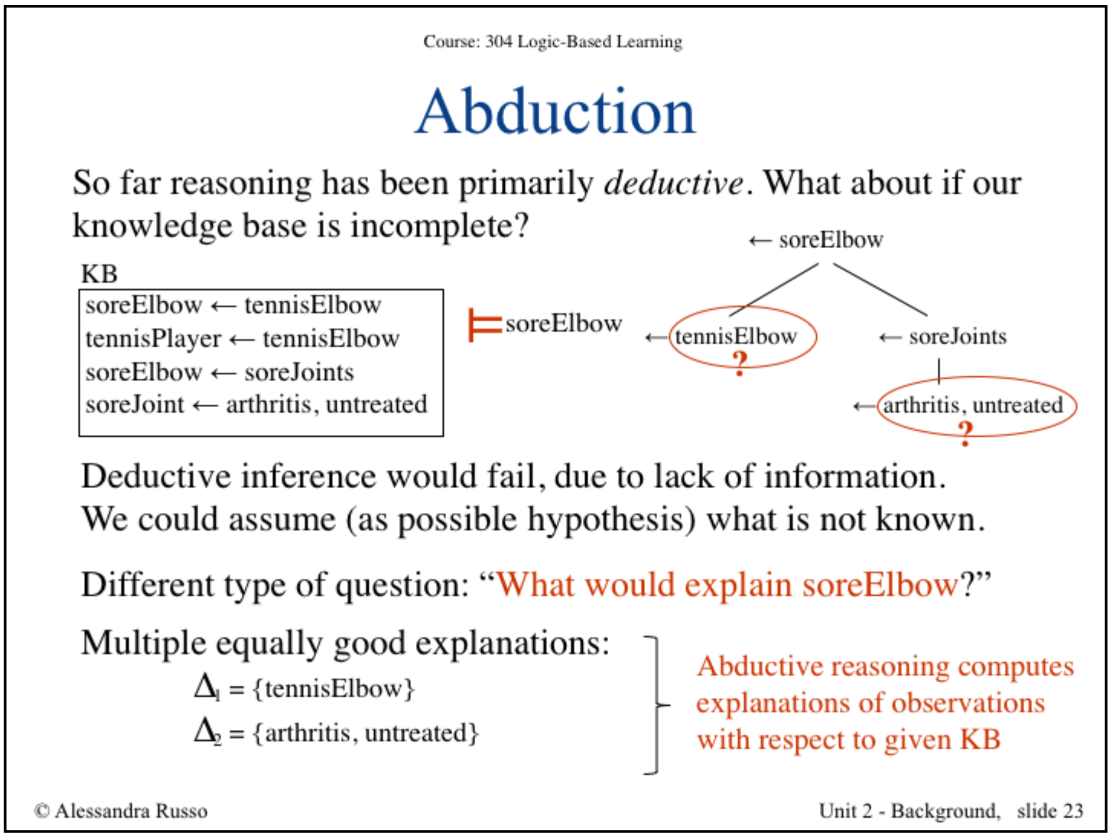
Unit 2 - Background, slide 22

In the above example a slight modification has been made in the KB in the definition of unconnected. Note that the condition "**not arc(X,Y)**" mentions a variable Y that does neither appear in the head nor in a positive literal before the fail condition. In this case the SLD derivation will eventually generate a subgoal that is a fail goal (i.e. ← not arc(a,Y)) where the fail is applied to a not ground atom.

To prove such a subgoal a new derivation tree should start with the subgoal ← arc(a,Y). But the implementation of negation as failure in SLDNF computes the negation of an atom by just testing whether the atom is provable or not. It is not able to generate bindings for variables, but only test whether sub-goals succeed or fail. So some Prolog implementations of the operator \+ only works when applied to a literal containing no variables, i.e., a ground literal. So to guarantee reasonable answers to queries to programs containing negation, the negation operator must be allowed to apply only to ground literals. If it is applied to a non-ground literal, the program is said to flounder. You could also think of as, in principle, requiring to search through possibly infinite derivations in order to show that does not exist such a value of Y. This is why in the SLDNF the strategy adopted is that whenever a non-ground fail literal is encountered as sub-goal, the derivation consider it **not possible to evaluate** and a floundering condition is reported.

So the question is why bother with negation by failure? The answer is that it provides more flexible knowledge representation. In the 4th year course of Knowledge Representation you will hopefully see how negation as failure is an ideal operator for expressing default assumptions and capture default reasoning. In the context of this course, it is key for expressing existing knowledge about a problem domain and help learning in the presence of defaults hypothesis that can also capture default conditions.

We will also see that in the context of Answer Set Programming problems such as infinite computation of Prolog are no longer an issue and floundering cases are controlled but the requirement of *safe variables* and *safe rules*.



Course: 304 Logic-Based Learning

Abduction

So far reasoning has been primarily *deductive*. What about if our knowledge base is incomplete?

KB

soreElbow ← tennisElbow
tennisPlayer ← tennisElbow
soreElbow ← soreJoints
soreJoint ← arthritis, untreated

$\models$ soreElbow

← soreElbow
←tennisElbow
?
← soreJoints
←arthritis, untreated
?

Deductive inference would fail, due to lack of information.
We could assume (as possible hypothesis) what is not known.

Different type of question: "What would explain soreElbow?"

Multiple equally good explanations:

Δ_1 = {tennisElbow}

Δ_2 = {arthritis, untreated}

Abductive reasoning computes explanations of observations with respect to given KB

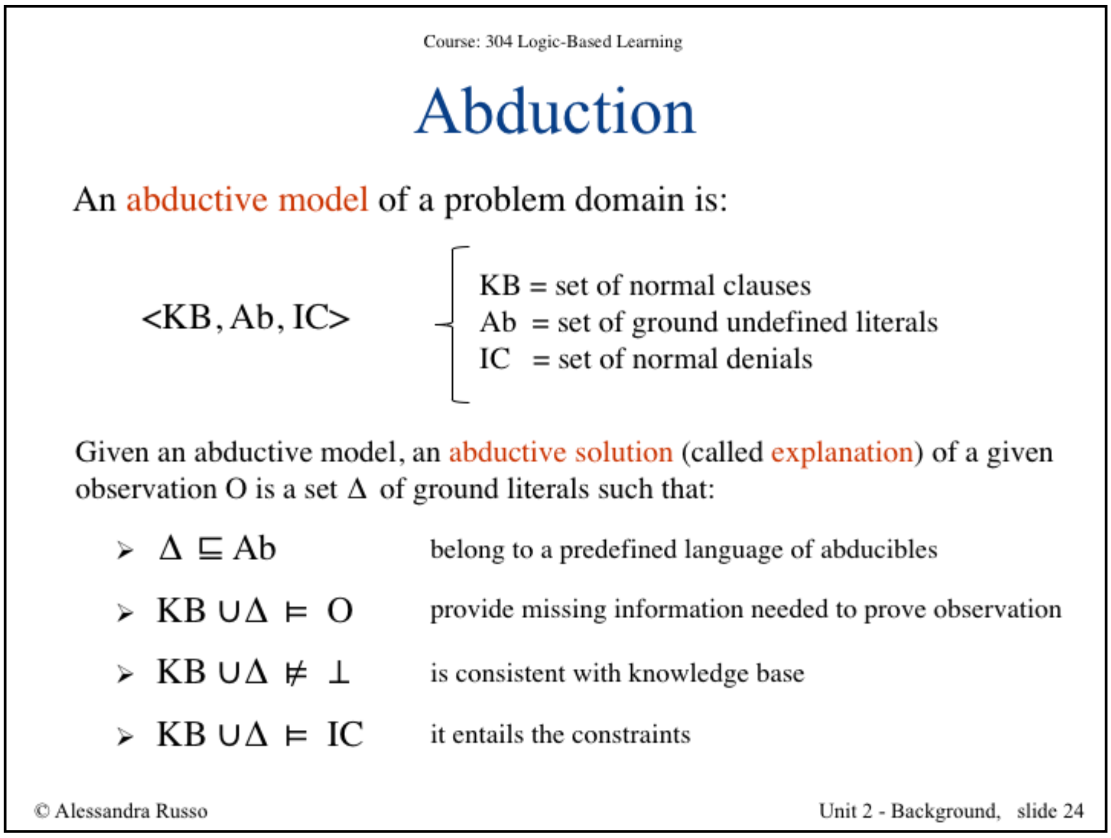
© Alessandra Russo

Unit 2 - Background, slide 23

So far we have considered different classes of predicate logics and seen mechanisms for computing consequences of given theories (or knowledge bases). As mentioned in slide 6 what deduction proves is information already implicit in the premises (rules and cases). But what about if the cases are not complete, for instance we don't know enough information about a certain situation but we knowledge the general rules? A resolution derivation will fail as it would terminate with clauses (or denials) that cannot be resolved for lack of information. Abduction addresses this point. It is particularly suited when we have an idea of the type of knowledge that we might not have enough information about, but that we want to conjecture about (or identify) in order to prove some given observations.

For instance, consider the example given in this slide. The observation is a symptom (sore elbow). There might be various reasons why somebody might have a sore elbow and given general rule-based knowledge we can, through abduction, identify possible explanations. When performing abduction, the kind of question we want to answer is not whether a given goal is provable from a given theory, but rather "what is a possible explanation for a given goal".

Different explanations could be generated. Some are subset of others. Explanations should not be unnecessarily strong (i.e. include more ground literals that what is needed) or unnecessarily weak (too few to prove the goal in all circumstances). They have to be parsimonious. Abductive algorithms for normal clauses aims to compute a "minimal" set of explanations that, together with the given theory, proves the goal. Furthermore, explanations should be identified among what is considered to be plausible in a given problem domain. For instance, in the above scenario information about tennis ranking of the player would not help explain a sore elbow. So when modelling a problem domain in an abductive term, we should think about a vocabulary of what are considered to be plausible explanations for the given types of observations. We refer to this vocabulary as the set of **abducibles**.


Course: 304 Logic-Based Learning

Abduction

An abductive model of a problem domain is:

<KB, Ab, IC>

- KB = set of normal clauses
- Ab = set of ground undefined literals
- IC = set of normal denials

Given an abductive model, an abductive solution (called explanation) of a given observation O is a set Δ of ground literals such that:

➢ $\Delta \sqsubseteq Ab$	belong to a predefined language of abducibles
➢ $KB \cup \Delta \models O$	provide missing information needed to prove observation
➢ $KB \cup \Delta \not\models \perp$	is consistent with knowledge base
➢ $KB \cup \Delta \models IC$	it entails the constraints

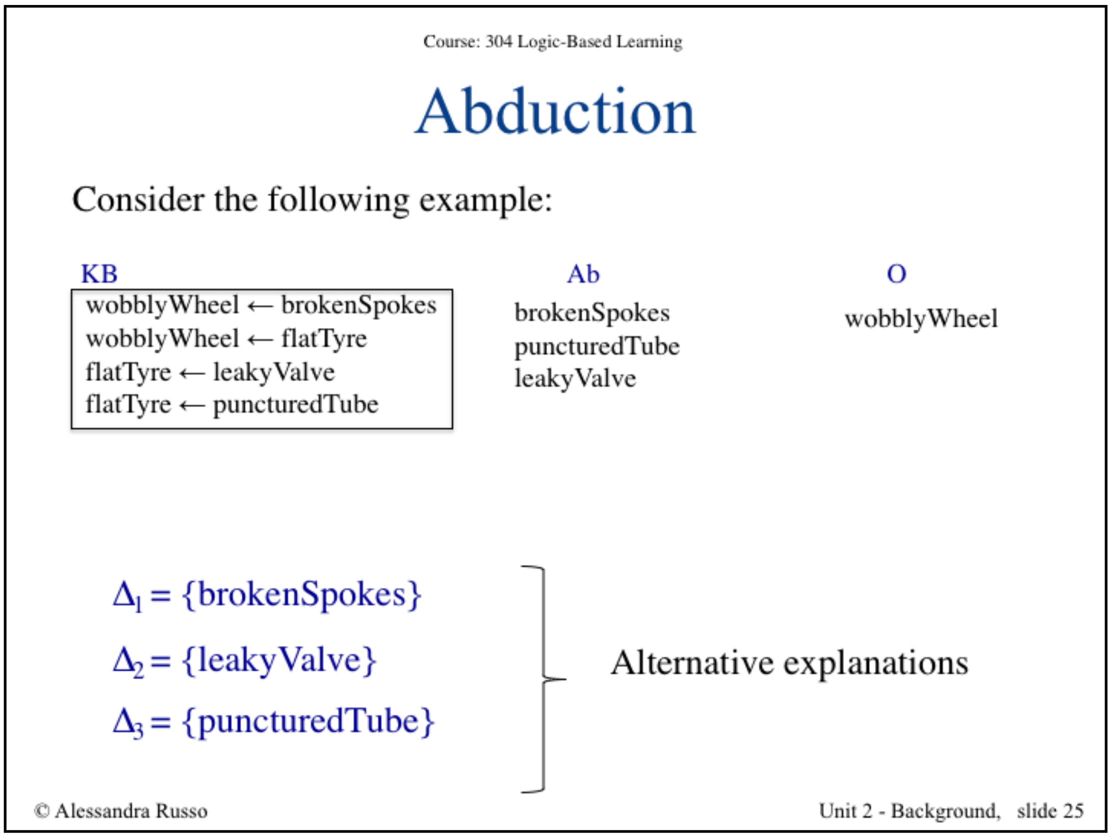
© Alessandra Russo — Unit 2 - Background, slide 24

A problem domain can therefore be modelled abductively by a tuple composed of three elements: a theory (or set of normal clauses) KB, a set of abducibles (Ab), i.e. undefined predicates expressing the concepts which are unknown and therefore for which the KB is incomplete, and a set of constraints (IC). The latter are assumed to be expressed as normal denials.

Given an abductive model, this can be used to compute explanations of observations. For a given observation O (or goal G), an abductive solution is a set of ground abducibles (i.e. from the predefined set of literals in Ab), that is consistent with the background knowledge KB, that together with KB satisfies the denial constraints and finally that together with KB derives the observation. The integrity constraints are used to prune explanations that are not desirables but also to force some abducibles to be part of a solution for them to be satisfied.

The four conditions given above are, in principle, independent of the language in which KB, Ab, IC are formulated. They apply to problems represented as definite clauses, as well as to problems formalised as normal logic programs, and to problems formalised as answer set programs (as you will see later in this course). Depending on the type of language and formalisation you use to express the problem, the underlying semantics would differ. When the KB is a set of definite clauses, the augmented theory $KB \cup \Delta$ will also be a set of definite clauses that accepts a unique minimal model, the Least Herbrand Model. The entailment relation, in this case, would be under the minimal model semantics. So the last three conditions will simply require that the given goal and the constraints **are satisfied** in the Least Herbrand Model of $KB \cup \Delta$.

You will see later on in the course that when $KB \cup \Delta$ is a set of normal clauses more that one model may exist. In this case the latter condition maybe interpreted in consistent view (i.e. requiring that $KB \cup \Delta \cup IC$ is consistent) or in an entailment way (i.e. requiring that all models of $KB \cup \Delta$ satisfy the IC.

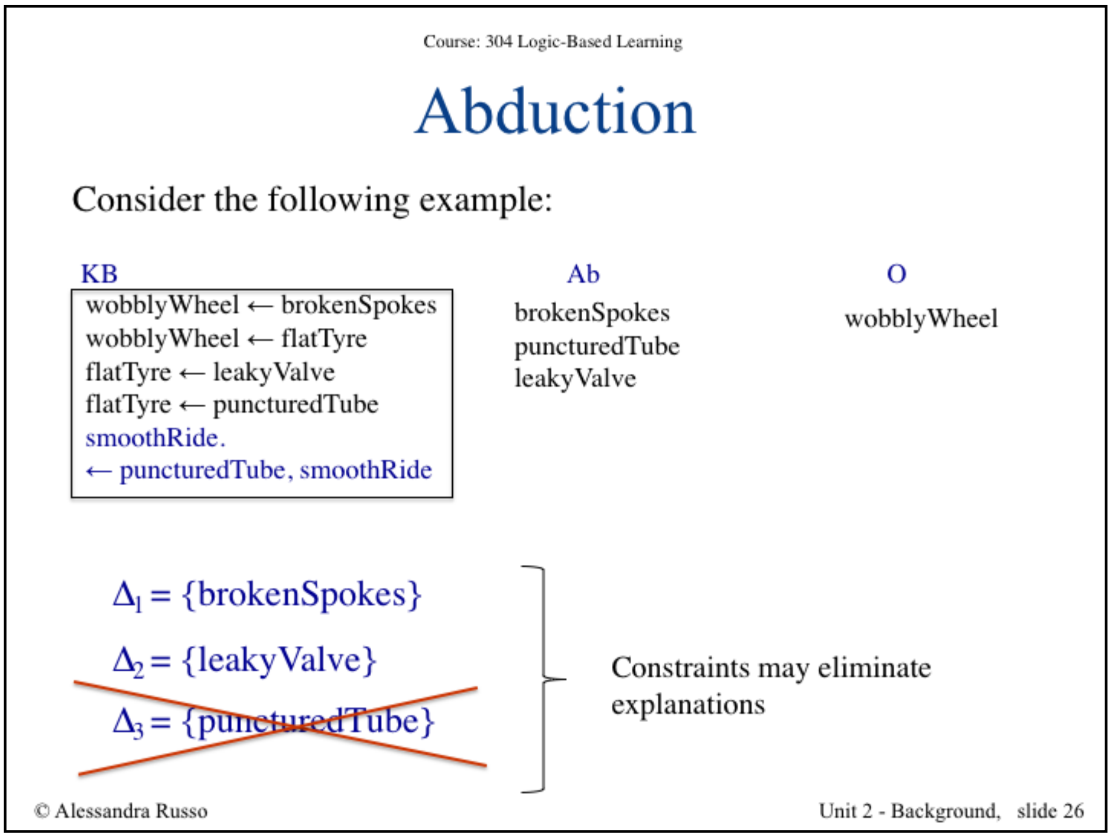


Abduction

Consider the following example:

KB	Ab	O
wobblyWheel ← brokenSpokes	brokenSpokes	wobblyWheel
wobblyWheel ← flatTyre	puncturedTube	
flatTyre ← leakyValve	leakyValve	
flatTyre ← puncturedTube		

$$\left.\begin{array}{l}\Delta_1 = \{\text{brokenSpokes}\} \\ \Delta_2 = \{\text{leakyValve}\} \\ \Delta_3 = \{\text{puncturedTube}\}\end{array}\right\} \text{Alternative explanations}$$



Abduction

Consider the following example:

KB

$\text{wobblyWheel} \leftarrow \text{brokenSpokes}$
$\text{wobblyWheel} \leftarrow \text{flatTyre}$
$\text{flatTyre} \leftarrow \text{leakyValve}$
$\text{flatTyre} \leftarrow \text{puncturedTube}$
$\text{smoothRide}.$
$\leftarrow \text{puncturedTube}, \text{smoothRide}$

Ab

brokenSpokes
puncturedTube
leakyValve

O

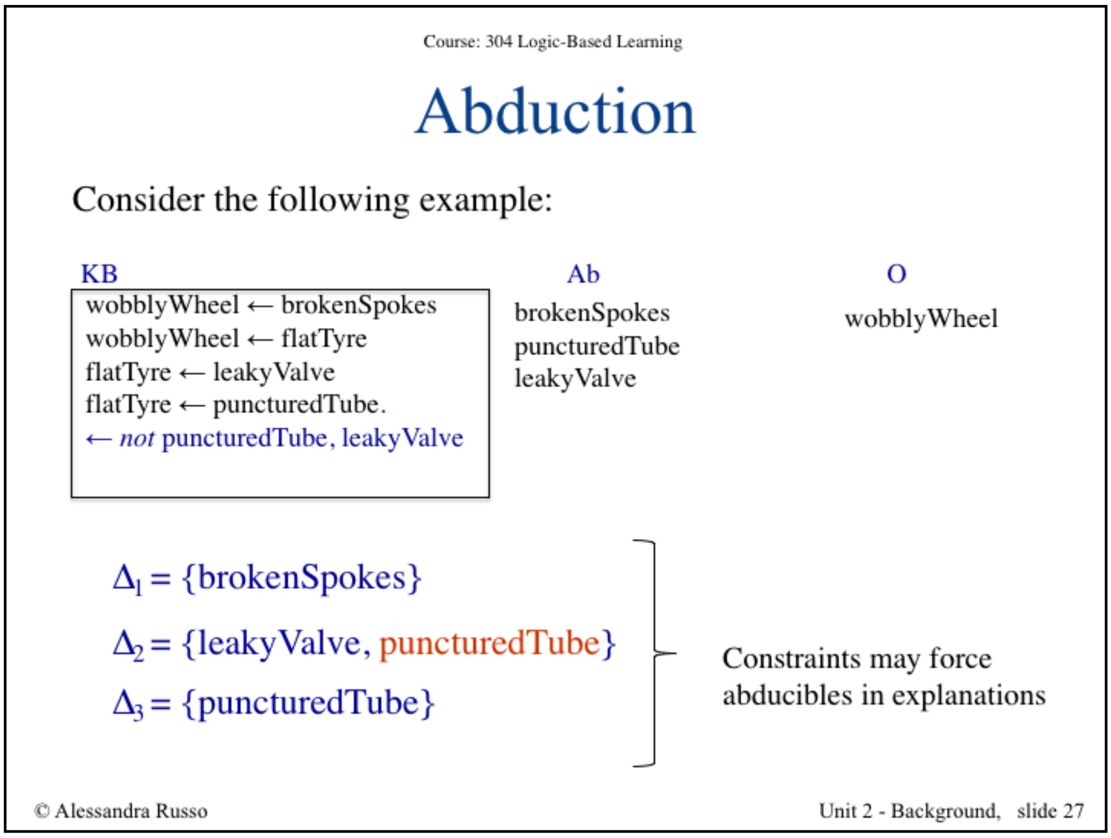
wobblyWheel

$\Delta_1 = \{\text{brokenSpokes}\}$

$\Delta_2 = \{\text{leakyValve}\}$

~~$\Delta_3 = \{\text{puncturedTube}\}$~~

Constraints may eliminate explanations



Abduction

Consider the following example:

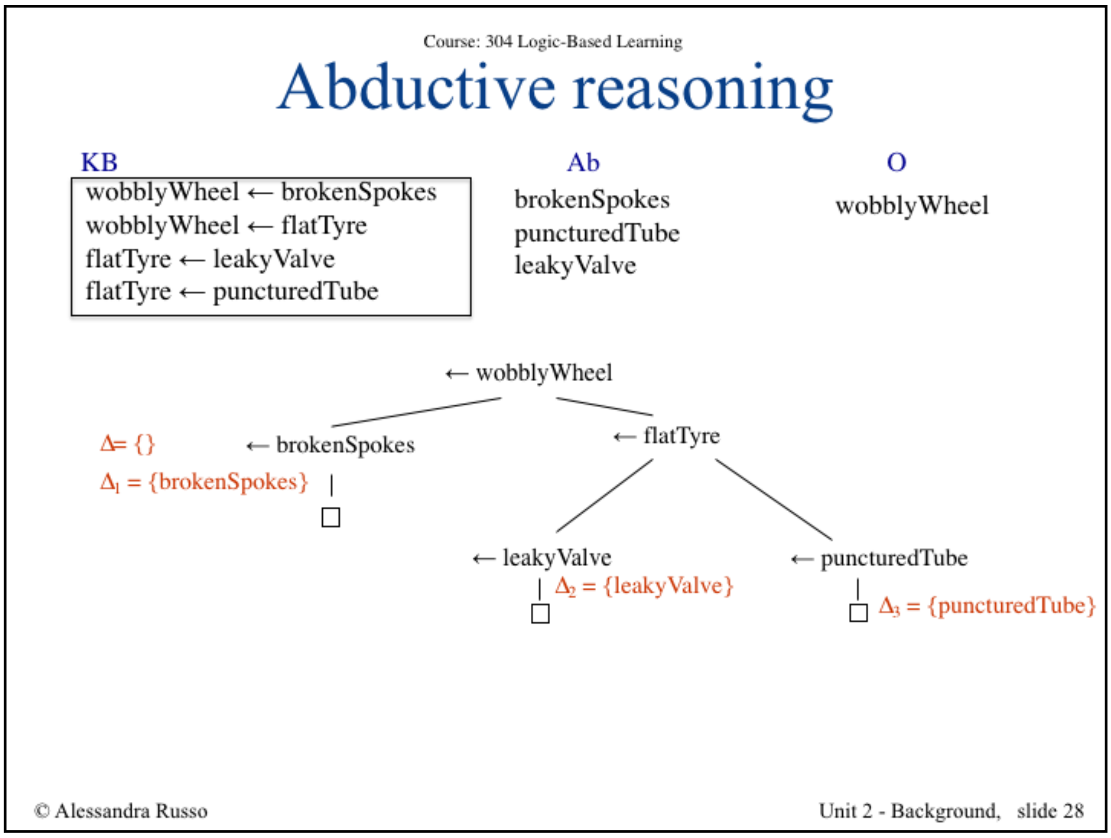
KB	Ab	O
wobblyWheel $\leftarrow$ brokenSpokes	brokenSpokes	wobblyWheel
wobblyWheel $\leftarrow$ flatTyre	puncturedTube	
flatTyre $\leftarrow$ leakyValve	leakyValve	
flatTyre $\leftarrow$ puncturedTube.		
$\leftarrow$ *not* puncturedTube, leakyValve		

$\Delta_1 = \{\text{brokenSpokes}\}$

$\Delta_2 = \{\text{leakyValve}, \text{puncturedTube}\}$

$\Delta_3 = \{\text{puncturedTube}\}$

Constraints may force abducibles in explanations



Abductive reasoning

KB

wobblyWheel ← brokenSpokes
wobblyWheel ← flatTyre
flatTyre ← leakyValve
flatTyre ← puncturedTube

Ab

brokenSpokes
puncturedTube
leakyValve

O

wobblyWheel

← wobblyWheel

$\Delta = \{\}$ ← brokenSpokes

$\Delta_1 = \{brokenSpokes\}$ □

← flatTyre

← leakyValve

$\Delta_2 = \{leakyValve\}$ □

← puncturedTube

□ $\Delta_3 = \{puncturedTube\}$

Abductive reasoning

KB

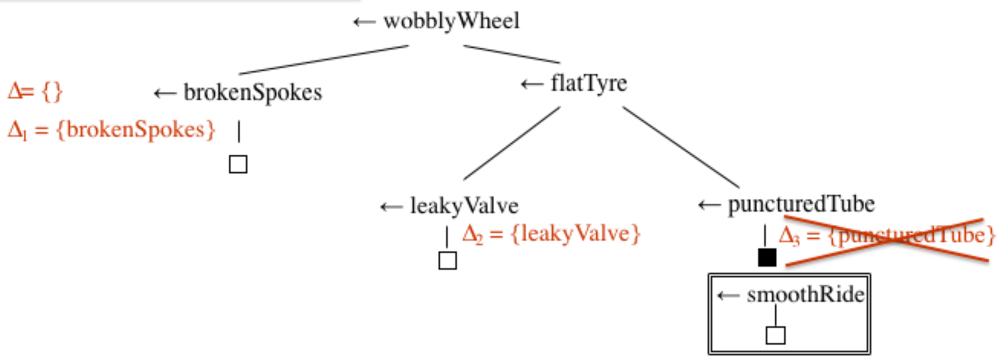
wobblyWheel ← brokenSpokes
wobblyWheel ← flatTyre
flatTyre ← leakyValve
flatTyre ← puncturedTube
smoothRide.
← puncturedTube, smoothRide

Ab

brokenSpokes
puncturedTube
leakyValve

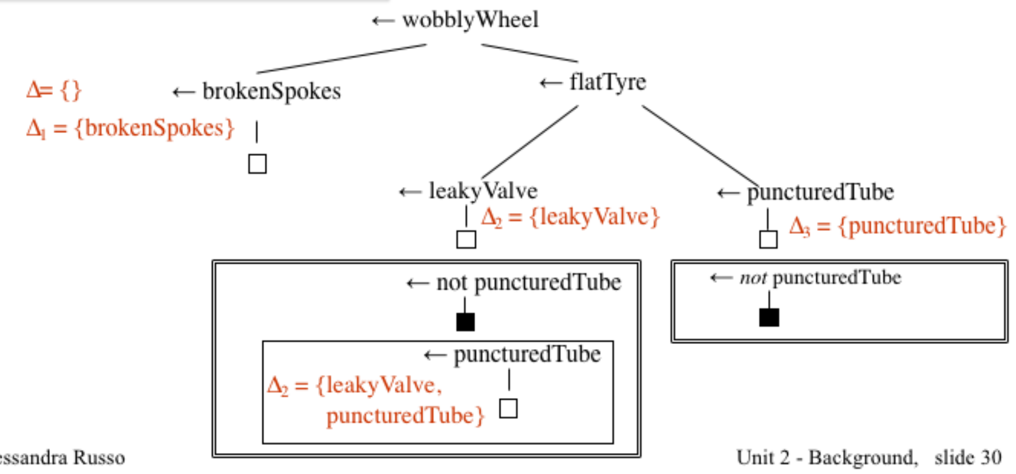
O

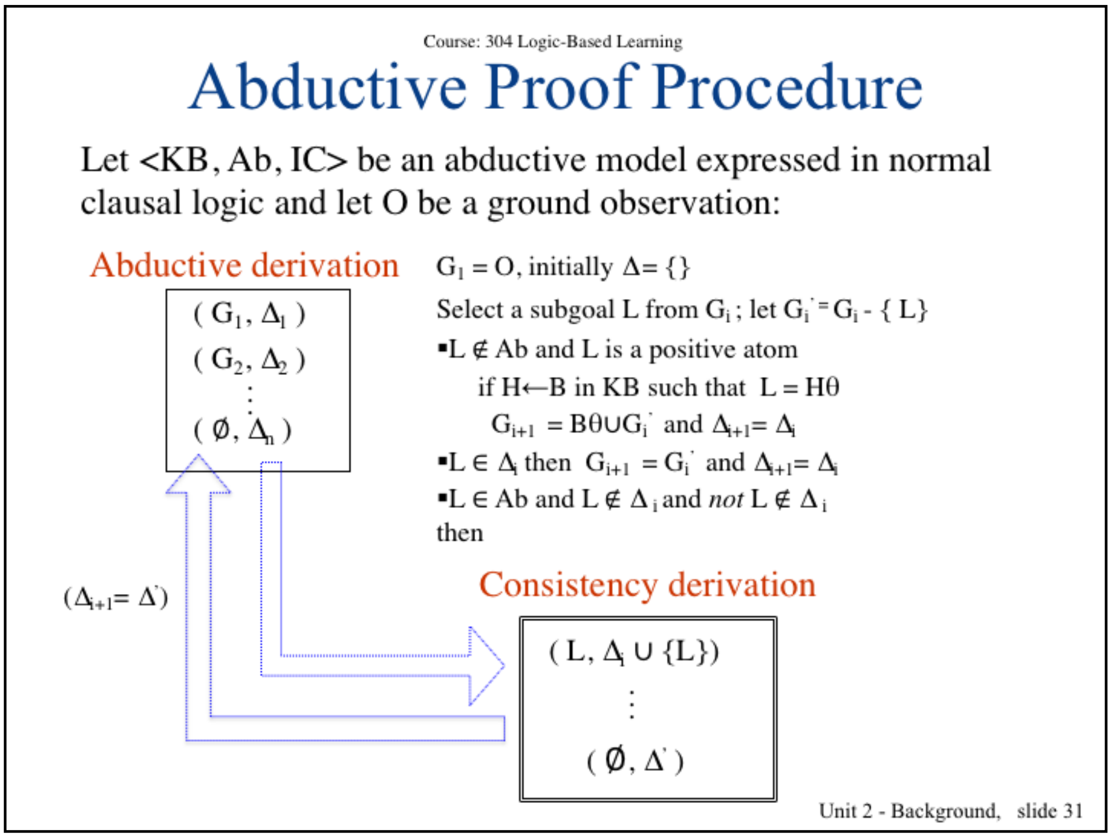
wobblyWheel


← wobblyWheel
Δ= {}
← brokenSpokes
Δ_1 = {brokenSpokes}
← flatTyre
← leakyValve
Δ_2 = {leakyValve}
← puncturedTube
Δ_3 = {~~puncturedTube~~}
← smoothRide

Abductive reasoning

KB	Ab	O
wobblyWheel ← brokenSpokes	brokenSpokes	wobblyWheel
wobblyWheel ← flatTyre	puncturedTube	
flatTyre ← leakyValve	leakyValve	
flatTyre ← puncturedTube		
←*not* puncturedTube, leakyValve		





Course: 304 Logic-Based Learning

Abductive Proof Procedure

Let <KB, Ab, IC> be an abductive model expressed in normal clausal logic and let O be a ground observation:

Abductive derivation

(G_1, Δ_1)

(G_2, Δ_2)

$\vdots$

$(\emptyset, \Delta_n)$

$G_1 = O$, initially $\Delta = \{\}$

Select a subgoal L from G_i; let $G_i' = G_i - \{L\}$

- $L \notin Ab$ and L is a positive atom
  - if $H \leftarrow B$ in KB such that $L = H\theta$
  - $G_{i+1} = B\theta \cup G_i'$ and $\Delta_{i+1} = \Delta_i$
- $L \in \Delta_i$ then $G_{i+1} = G_i'$ and $\Delta_{i+1} = \Delta_i$
- $L \in Ab$ and $L \notin \Delta_i$ and *not* $L \notin \Delta_i$

then

Consistency derivation

$(L, \Delta_i \cup \{L\})$

$\vdots$

$(\emptyset, \Delta')$

$(\Delta_{i+1} = \Delta')$

Unit 2 - Background, slide 31

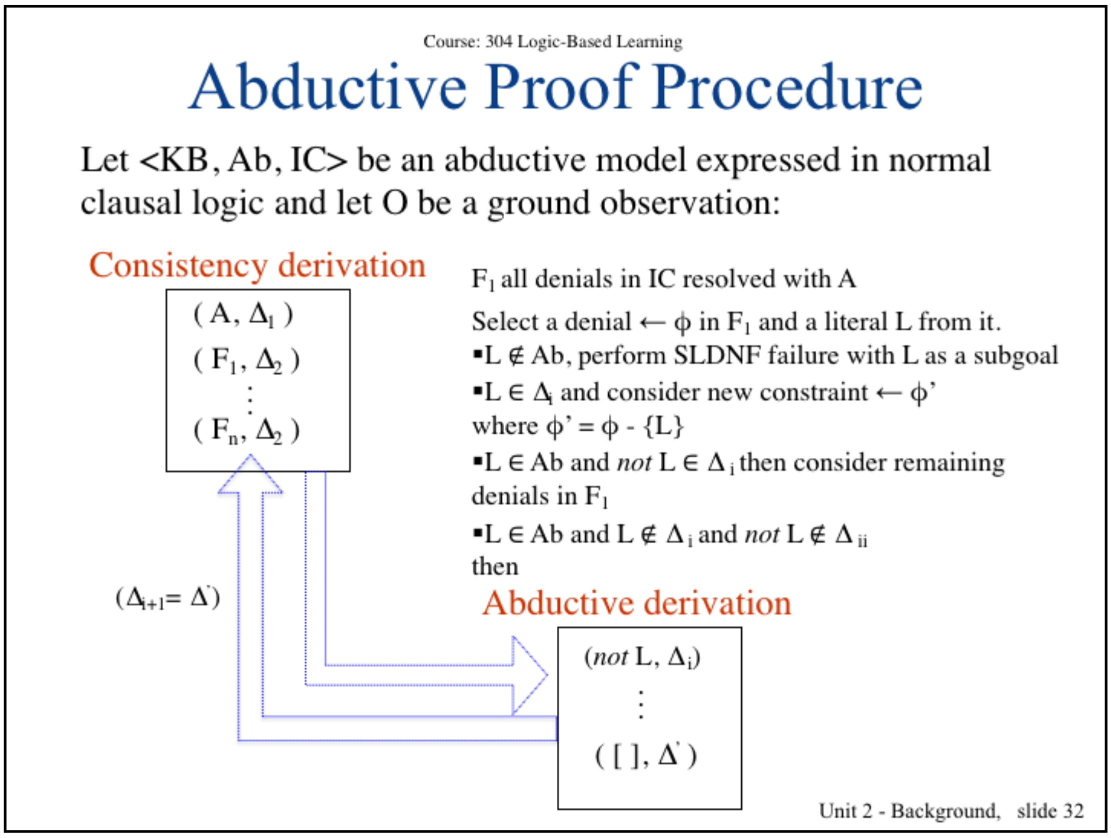
This and the next slide try to formalise the different steps of an abductive reasoning proof procedure, called also operational semantics of abduction. The general feature of this procedure is that it is composed of two different phases, called abductive and consistency phases that interleave each other, i.e. one can call the other and viceversa. The important thing to understand is, at which point each of these phases is called and why. Note that to guarantee consistency between assumptions and negated predicates, the set of abducibles Ab is assumed to include the negation of all defined predicates and abducibles; and a consistency constraint of the form $\leftarrow$A, *not* A, for any predicate A, is assumed to be implicitly part of IC.

Starting from a given observation the first phase is always an abductive phase. So the observation become the goal of the abductive phase derivation that needs to be proved. At the beginning the set of explanations is empty as nothing has been assumed yet. The abductive derivation is a derivation that succeeds when no further subgoals are left to be proved. This phase can be called at any point of the proof. So the current goal could be of different types:

i)The current goal is not an abducible. This is often the case for instance of the given observation. Then an SLDNF derivation is initiated with the current goal (or observation in the first abductive phase) to be proved by refutation. The standard SLDNF resolution takes place. Note that the third bullet point above includes also the case of when a non abducible but negative literal is generated during the SLDNF derivation as subgoal. This is because we have said that implicitly the negation of all defined predicated are also assumed to be added to Ab, not because they are not defined but because we will need them to guarantee consistency.

ii)The current goal is an abducible (positive or negative). Since an abductive phase has to conclude on success, when such a goal is generated, if it is already part of the current Δ, then goal is assumed to be proved and the derivation proceeds with the rest of the pending sub-goals.

iii)The current goal is an abducible not yet assumed (remember this could also be the negation as failure of a predicate defined in BK). In this case it can be assumed but a consistency phase needs to be triggered to verify that such an assumption does not introduce inconsistencies. The consistency phase is clearly a failure derivation. So it succeeds when the derivation finishes with a failure.



Course: 304 Logic-Based Learning

Abductive Proof Procedure

Let <KB, Ab, IC> be an abductive model expressed in normal clausal logic and let O be a ground observation:

Consistency derivation

(A, Δ_1)

(F_1, Δ_2)

$\vdots$

(F_n, Δ_2)

F_1 all denials in IC resolved with A

Select a denial $\leftarrow \phi$ in F_1 and a literal L from it.

▪L $\notin$ Ab, perform SLDNF failure with L as a subgoal

▪L $\in \Delta_i$ and consider new constraint $\leftarrow \phi'$ where $\phi' = \phi - \{L\}$

▪L $\in$ Ab and *not* L $\in \Delta_i$ then consider remaining denials in F_1

▪L $\in$ Ab and L $\notin \Delta_i$ and *not* L $\notin \Delta_{ii}$ then

Abductive derivation

$(\textit{not}\ L, \Delta_i)$

$\vdots$

$([\,], \Delta')$

$(\Delta_{i+1} = \Delta')$

Unit 2 - Background, slide 32

In the consistency phase, the new assumption is added to the current set Δ and resolved with any constraint that includes it. The new set of resolvent denials have each of them to fail for the consistency phase to succeed. A literal is selected from the first resolvent denial. Different cases need to be considered here:

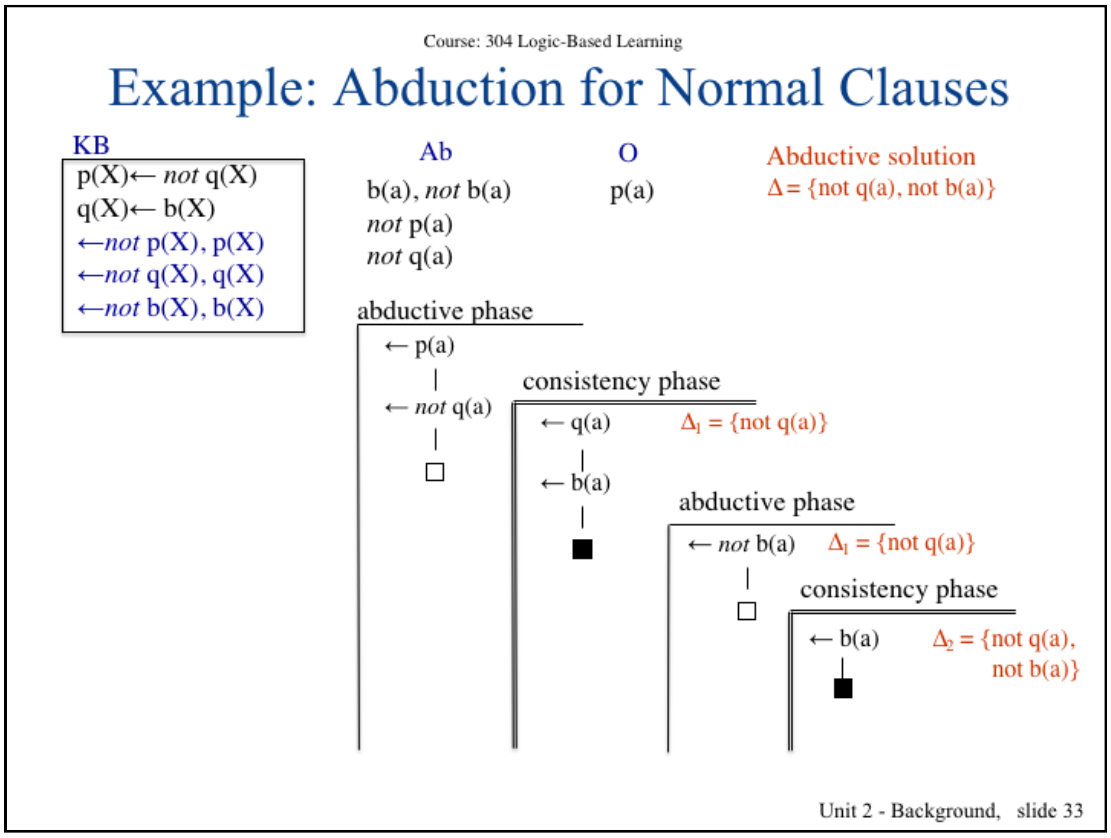
i)The selected literal is not an abducible (this means is a positive literal). So SLDNF failure derivation is applied here.

ii)The selected literal is an abducible already assumed in Δ. Then this literal in the denial resolvent success, so for the resolvent to be proved to fail the remaining literals will need to be checked for failure. So a new resolvent is generated by removing from it the selected literal and adding the new resolvent at the front of the set of pending resolvent denial left to check.

iii)The selected literal is an abducible whose negation is already assumed in Δ. In this case literal fails (by resolution with the literal in Δ) and the denial can be removed and considered to be satisfied. The next denial to check is picked and the consistency derivation starts again.

iv)The selected literal is an abducible who is not assumed and its negation is also not assumed. In this case since the literal has to fail (because we are in a consistency phase derivation) an abductive derivation of its negation (as subgoal) is started. If this abductive derivation succeeds then the chosen literal fails and the consistency derivation of the initially chosen denial succeed. The next pending denial is picked and the consistency derivation starts again.

An example derivation is given in the next slide.



Course: 304 Logic-Based Learning

Example: Abduction for Normal Clauses

KB

$p(X) \leftarrow not\ q(X)$
$q(X) \leftarrow b(X)$
$\leftarrow not\ p(X), p(X)$
$\leftarrow not\ q(X), q(X)$
$\leftarrow not\ b(X), b(X)$

Ab

$b(a)$, $not\ b(a)$
$not\ p(a)$
$not\ q(a)$

O

$p(a)$

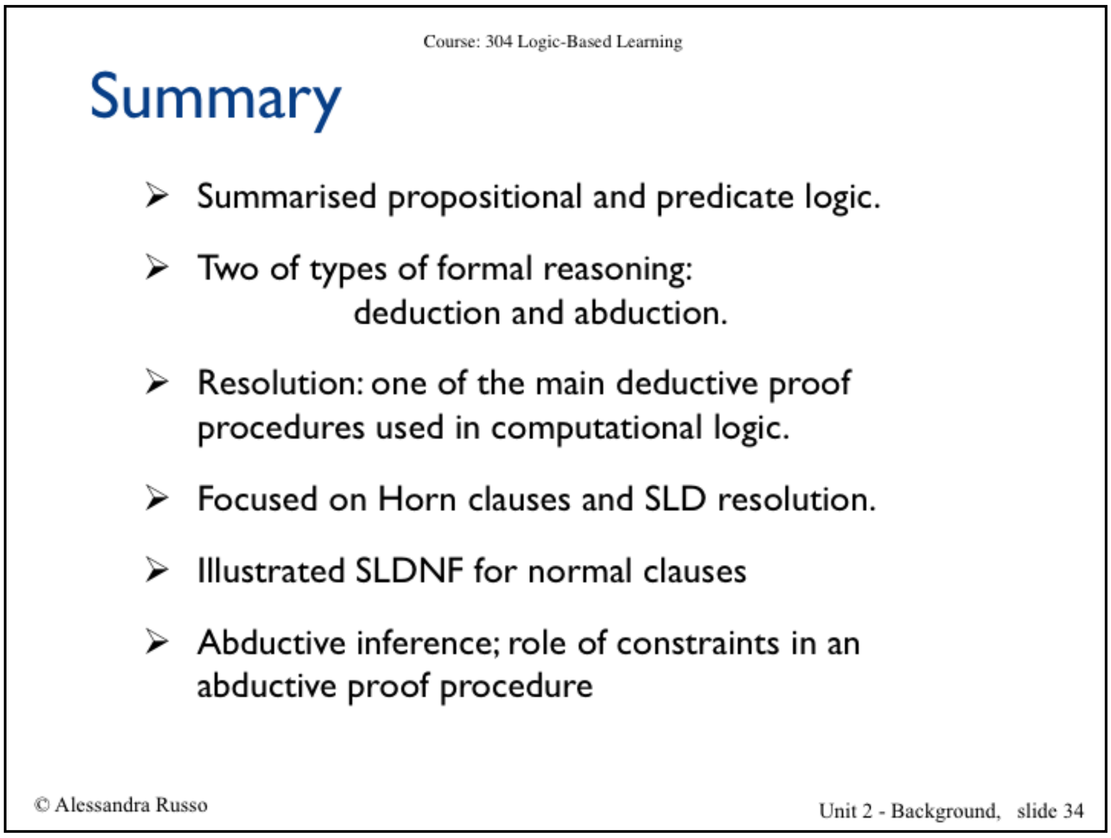
Abductive solution

$\Delta = \{not\ q(a), not\ b(a)\}$

Unit 2 - Background, slide 33

Later in the course you will see how this "top down", or goal-directed, abductive proof procedure is used to support also an inductive learning task where the outcome is not a set of ground literals but a set of predicate normal clauses.

We conclude here our general recap of background material that you need to know in order to fully understand the rest of this course.



Summary

- Summarised propositional and predicate logic.
- Two of types of formal reasoning:
  deduction and abduction.
- Resolution: one of the main deductive proof procedures used in computational logic.
- Focused on Horn clauses and SLD resolution.
- Illustrated SLDNF for normal clauses
- Abductive inference; role of constraints in an abductive proof procedure