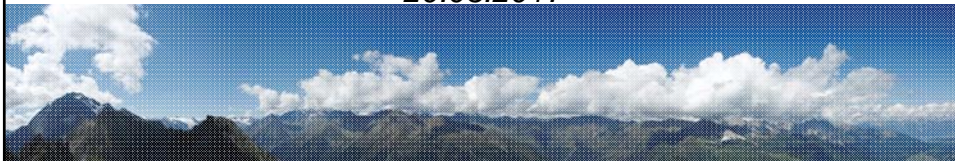


# Artificial Intelligence SS 2017

## Reasoning

Anna Fensel  
20.03.2017



© Copyright 2011 Dieter Fensel and Florian Fischer

### Where are we?



#	Title
0.1	Propositional Logic
0.2	Predicate Logic
1	Introduction
<b>2</b>	<b>Reasoning</b>
3	Search Methods
4	CommonKADS
5	Problem-Solving Methods
6	Planning
7	Software Agents
8	Rule Learning
9	Inductive Logic Programming
10	Neural Networks
11	Semantic Web and Services
12	Exam

- Motivation
- Technical Solution
  - Introduction to Theorem Proving and Resolution
  - Description Logics
  - Logic Programming
- Summary
- References

# MOTIVATION

- Basic results of mathematical logic show:
  - We can do logical reasoning with a **limited set of simple (computable) rules** in restricted formal languages like First-order Logic (FOL)
  - **Computers** can do reasoning
- FOL is interesting for this purpose because:
  - It is expressive enough to capture many foundational theorems of mathematics
  - Many real-world problems can be formalized in FOL
  - It is the most expressive logic that one can adequately approach with automated theorem proving techniques

- Due to its theoretical properties (decidability & complexity) First-Order Logic is not always an ideal solution
- This motivates research towards formalisms with more practically oriented computational properties and expressivity
  - Description Logics
    - Syntactic fragments of FOL
    - Focus on decidability and optimized algorithms key reasoning tasks (terminological reasoning / schema reasoning)
  - Logic Programming
    - Provides different expressivity than classical FOL (non-monotonic reasoning with non-classical negation)
    - Provides a very intuitive way to model knowledge
    - Efficient reasoning procedures for large data-sets

## **TECHNICAL SOLUTIONS**

Theorem Proving and Resolution, Description Logics,  
and Logic Programming

## **THEOREM PROVING**

Introduction to Theorem Proving and Resolution

- A proof system is collection of inference rules of the form:

$$\frac{P_1 \dots P_n}{C} \text{ name}$$

where C is a conclusion sequent, and  $P_i$ 's are premises sequents .

- If an inference rule has premises that are taken to be true (called an **axiom**), its conclusion automatically holds.
- Example: Modus Ponens: From  $P, P \rightarrow Q$  infer  $Q$ ,  
Universal instantiation: From  $(\forall x)p(x)$  infer  $p(A)$
- Theorems:
  - Expressions that can be derived from the axioms and the rules of inference.

- Resolution is a *refutation system*.
  - To prove a statement we attempt to refute its negation
- Basic idea: Given a consistent set of axioms  $KB$  and goal sentence  $Q$ , we want to show that  $KB \models Q$ 
  - This means that every interpretation  $I$  that satisfies  $KB$  also satisfies  $Q$
  - Any interpretation  $I$  only satisfies either  $Q$  or  $\neg Q$ , but not both
  - Therefore, if in fact  $KB \models Q$  holds, an interpretation that satisfies  $KB$ , satisfies  $Q$  and does not satisfy  $\neg Q$ 
    - **Hence  $KB \cup \{\neg Q\}$  is unsatisfiable, i.e., it is false under all interpretations**

- Resolution commonly requires sentences to be in clause normal form (also called conjunctive normal form or CNF)
  - A clause is a disjunction of literals (implicitly universally quantified)
    - E.g.:  $l_1 \vee \dots \vee l_n$
  - A formula in clause normal form conjunction of a set of clauses
    - E.g.:  $C_1 \wedge \dots \wedge C_n = (l_1 \vee \dots \vee l_m) \wedge \dots \wedge (l_1 \vee \dots \vee l_j)$
- High level steps in a resolution proof:
  - Put the premises or axioms into **clause normal form (CNF)**
  - Add the **negation** of the to be proven statement, in clause form, to the set of axioms
  - **Resolve** these clauses together, using the **resolution rule**, producing new clauses that logically follow from them
  - **Derive a contradiction** by generating the empty clause.
  - The **substitutions** used to produce the empty clause are those under which the opposite of the negated goal is true

- The **resolution rule** is a single inference that produces a new clause implied by two clauses (called the parent clauses) containing complementary literals
  - Two literals are said to be complements if one is the negation of the other (in the following,  $a_i$  is taken to be the complement to  $b_j$ ).
  - The resulting clause contains all the literals that do not have complements.  
Formally:
 
$$\frac{a_1 \vee \dots \vee a_i \vee \dots \vee a_n, \quad b_1 \vee \dots \vee b_j \vee \dots \vee b_m}{a_1 \vee \dots \vee a_{i-1} \vee a_{i+1} \vee \dots \vee a_n \vee b_1 \vee \dots \vee b_{j-1} \vee b_{j+1} \vee \dots \vee b_m}$$
  - Where  $a_1, \dots, a_n, b_1, \dots, b_m$  are literals, and  $a_i$  is the complement to  $b_j$
- The clause produced by the resolution rule is called the **resolvent** of the two input clauses.
  - A literal and its negation produce a resolvent only if they unify under some substitution  $\sigma$ .
  - $\sigma$  is then applied to the resolvent

- When the two initial clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair
  - However, only the pair of literals that are resolved upon can be removed: All other pairs of literals remain in the resolvent clause
- Modus ponens (see the very first lectures) can be seen as a special case of resolution of a one-literal clause and a two-literal clause
- Example: Given clauses
  - $P(x, f(a)) \vee P(x, f(y)) \vee Q(y)$  and  $\neg P(z, f(a)) \vee \neg Q(z)$
  - $P(x, f(a))$  and  $\neg P(z, f(a))$  **unify** with substitution  $\sigma = \{x/z\}$
  - Therefore, we derive the **resolvent clause** (to which  $\sigma$  is applied):  
 $P(z, f(y)) \vee Q(y) \vee \neg Q(z)$

1. Convert all the propositions of a knowledge base  $KB$  to **clause normal form** (CNF)
2. Negate  $Q$  (the to be proven statement) and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended:
  - a) Select two clauses. Call these the **parent clauses**.
  - b) Resolve them together using the **resolution rule**. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions.
  - c) If the resolvent is the **empty clause**, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

```

procedure resolution-refutation(KB, Q)
  ;; KB a set of consistent, true sentences
  ;; Q is a goal sentence that we want to derive
  ;; return success if  $KB \vdash Q$ , and failure otherwise
  KB = union(KB,  $\neg Q$ )
  while false not in KB do
    pick 2 sentences,  $S_1$  and  $S_2$ , in KB that contain
    literals that unify
      if none return "failure"
    resolvent = resolution-rule( $S_1, S_2$ )
    KB = union(KB, resolvent)
  return "success"

```

- Resolution expects input in **clause normal form**
- Step 1: Eliminate the logical connectives  $\rightarrow$  and  $\leftrightarrow$ 
  - $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$
  - $a \rightarrow b = \neg a \vee b$
- Step 2: Reduce the scope of negation
  - $\neg(\neg a) = a$
  - $\neg(a \wedge b) = \neg a \vee \neg b$
  - $\neg(a \vee b) = \neg a \wedge \neg b$
  - $\neg(\exists X) a(X) = (\forall X) \neg a(X)$
  - $\neg(\forall X) b(X) = (\exists X) \neg b(X)$



- Step 3: Standardize by renaming all variables so that variables bound by different quantifiers have unique names
  - $(\forall X) a(X) \vee (\forall X) b(X) = (\forall X) a(X) \vee (\forall Y) b(Y)$
- Step 4: Move all quantifiers to the left to obtain a *prenex normal form*
  - A formula is in prenex normal form if it is written as a string of quantifiers followed by a quantifier-free part
- Step 5: Eliminate existential quantifiers by using **skolemization**

- Step 5: Skolemization
  - Skolem constant
    - $(\exists X)(\text{dog}(X))$  may be replaced by  $\text{dog}(\text{fido})$  where the name fido is picked from the domain of definition of X to represent that individual X.
  - Skolem function
    - If the predicate has more than one argument and the existentially quantified variable is within the scope of universally quantified variables, the existential variable must be a function of those other variables.
      - $(\forall X)(\exists Y)(\text{mother}(X,Y)) \Rightarrow (\forall X)\text{mother}(X,m(X))$
      - $(\forall X)(\forall Y)(\exists Z)(\forall W)(\text{foo}(X,Y,Z,W))$
      - $\Rightarrow (\forall X)(\forall Y)(\forall W)(\text{foo}(X,Y,f(X,Y),w))$

## Resolution - Converting to Clause Normal Form



- Step 6: Drop all universal quantifiers
- Step 7: Convert the expression to the conjunctions of disjuncts
$$(a \wedge b) \vee (c \wedge d)$$
$$= (a \vee (c \wedge d)) \wedge (b \vee (c \wedge d))$$
$$= (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$
- Step 8: Split each conjunction into a separate clauses, which are just a disjunction of negated and nonnegated predicates, called literals
- Step 9: Rename so that no variable symbol appears in more than one clause.

$$(\forall X)(a(X) \wedge b(X)) = (\forall X)a(X) \wedge (\forall Y)b(Y)$$

19

## Resolution - Complete Example



- (Nearly) Classical example: Prove “Fido will die.” from the statements
  - “Fido is a dog.”
  - “All dogs are animals.”
  - “All animals will die.”
  - Changing premises to predicates
    - $\forall(x) (\text{dog}(X) \rightarrow \text{animal}(X))$
    - $\text{dog}(\text{fido})$
  - Modus Ponens and {fido/X}
    - $\text{animal}(\text{fido})$
    - $\forall(Y) (\text{animal}(Y) \rightarrow \text{die}(Y))$
  - Modus Ponens and {fido/Y}
    - $\text{die}(\text{fido})$

20

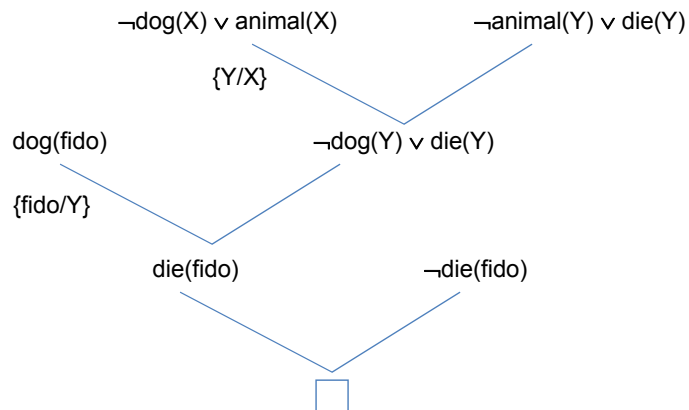
- Equivalent proof by Resolution

- Convert predicates to clause normal form

Predicate form	Clause form
1. $\forall(x) (\text{dog}(X) \rightarrow \text{animal}(X))$	$\neg\text{dog}(X) \vee \text{animal}(X)$
2. $\text{dog}(\text{fido})$	$\text{dog}(\text{fido})$
3. $\forall(Y) (\text{animal}(Y) \rightarrow \text{die}(Y))$	$\neg\text{animal}(Y) \vee \text{die}(Y)$

- Negate the conclusion

4. $\neg\text{die}(\text{fido})$	$\neg\text{die}(\text{fido})$
----------------------------------	-------------------------------

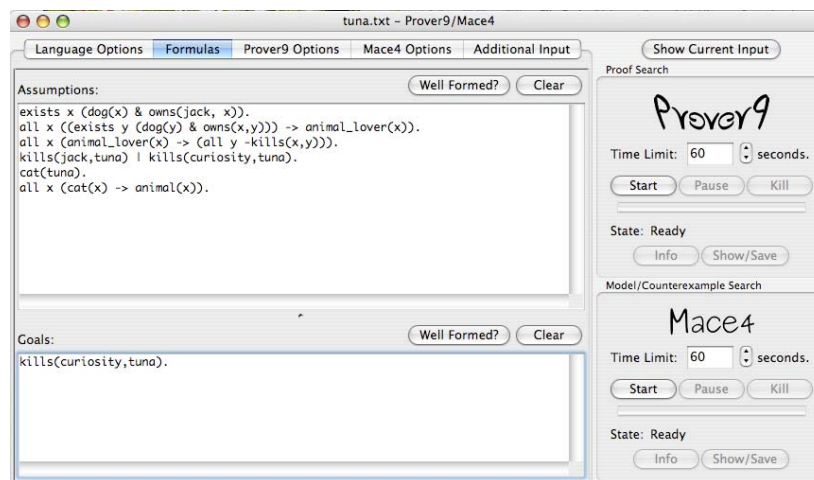


Resolution proof for the “dead dog” problem

- Resolution can be used to establish that a given sentence is entailed by a set of axioms
  - However, it **cannot**, in general, be used to generate **all logical consequences** of a set axioms
  - Also, the resolution **cannot** be used to prove that a given sentence is *not* entailed by a set of axioms
- Resolution defines a **search space**
  - The decision which clauses will be resolved against which others define the operators in the space
  - A **search method** is required
- Worst case: Resolution is exponential in the number of clauses to resolve

- Order of clause combination is important
  - N clauses  $\rightarrow$   $N^2$  ways of combinations or checking to see whether they can be combined
  - **Search heuristics** are very important in resolution proof procedures
- Strategies
  - Breadth-First Strategy
  - Set of Support Strategy
  - Unit Preference Strategy
  - Linear Input Form Strategy

- First-Order theorem prover
  - Homepage: <http://www.cs.unm.edu/~mccune/prover9/>
  - Successor of the well known “Otter” theorem prover
  - Under active development, available for several platforms and released under the GPL
  - Graphical user-interface and extensive documentation make Prover 9 comparatively user friendly
- Core of system builds on resolution / paramodulation as inference method
- Prover9 works in parallel with external component „Mace4“
  - Mace4 searches for finite models and counter examples
  - This helps to avoid wasting time searching for a proof by first finding a counter example



## Concrete System: Prover9 Proof



```
===== PROOF =====
% ----- Comments from original proof -----
% Proof 1 at 0.00 (+ 0.01) seconds.
% Length of proof is 14.
% Level of proof is 5.
% Maximum clause weight is 6.
% Given clauses 0.

1 (exists x (dog(x) & owns(jack,x))) # label(non_clause). [assumption].
2 (all x ((exists y (dog(y) & owns(x,y))) -> animal_lover(x))) # label(non_clause). [assumption].
3 (all x (animal_lover(x) -> (all y -kills(x,y)))) # label(non_clause). [assumption].
5 kills(curiosity,tuna) # label(non_clause) # label(goal). [goal].
6 -dog(x) | -owns(y,x) | animal_lover(y). [clausify(2)].
7 dog(c1). [clausify(1)].
8 -owns(x,c1) | animal_lover(x). [resolve(6,a,7,a)].
9 owns(jack,c1). [clausify(1)].
10 animal_lover(jack). [resolve(8,a,9,a)].
11 -animal_lover(x) | -kills(x,y). [clausify(3)].
14 kills(jack,tuna) | kills(curiosity,tuna). [assumption].
15 -kills(curiosity,tuna). [deny(5)].
16 -kills(jack,x). [resolve(10,a,11,a)].
17 $F. [back_unit_del(14),unit_del(a,16),unit_del(b,15)].

===== end of proof =====
```

27

## Concrete System: Prover9 Options



28

- Entailment:  $KB \models Q$ 
  - Entailment is a relation that is concerned with the **semantics** of statements
  - $Q$  is entailed by  $KB$  (a set of premises or assumptions) if and only if there is no logically possible world in which  $Q$  is false while all the premises in  $KB$  are true
- *Entailment for FOL is only semi-decidable*: If a conclusion follows from premises, then a complete proof system (like resolution) will find a proof.
  - If there is a proof, we will halt with it (eventually)
  - However, If there is **no** proof (i.e. a statement does not follow from a set of premises), the attempt to prove it may never halt

- From a practical point of view this is problematic
  - We cannot distinguish between the **non-existence of a proof** or the failure of an implementation to simply **find a proof in reasonable time**.
  - Theoretical completeness of an inference procedure does not make a difference in this cases
    - Does a proof simply take too long or will the computation never halt anyway?
- Due to its complexity and remaining limitations FOL is often not suitable for practical applications
- Often less expressive (but decidable) formalisms or formalisms with different expressivity are more suitable:
  - Description Logics
  - Logic Programming

# DESCRIPTION LOGICS

31

## Description Logic

- Most Description Logics are based on a 2-variable fragment of First Order Logic
  - **Classes** (concepts) correspond to unary predicates
  - **Properties** correspond to binary predicates
- Restrictions in general:
  - Quantifiers range over no more than 2 variables
  - Transitive properties are an exception to this rule
  - No function symbols (decidability!)
- Most DLs are decidable and usually have decision procedures for key reasoning tasks
- DLs have more efficient decision problems than First Order Logic
- We later show the very basic DL ALC as example
  - More complex DLs work in the same basic way but have different expressivity

32



- **Concepts/classes** (unary predicates/formulae with one free variable)
  - E.g. Person, Female
- **Roles** (binary predicates/formulae with two free variables)
  - E.g. hasChild
- **Individuals** (constants)
  - E.g. Mary, John
- **Constructors** allow to form more complex concepts/roles
  - Union  $\sqcup$ :  $\text{Man} \sqcup \text{Woman}$
  - Intersection  $\sqcap$ :  $\text{Doctor} \sqcap \text{Mother}$
  - Existential restriction  $\exists$ :  $\exists \text{hasChild}.\text{Doctor}$  (some child is a doctor)
  - Value(universal) restriction  $\forall$ :  $\forall \text{hasChild}.\text{Doctor}$  (all children are doctors)
  - Complement /negation  $\neg$ :  $\text{Man} \sqsubseteq \neg \text{Mother}$
  - Number restriction  $\geq n, \leq n$
- **Axioms**
  - Subsumption  $\sqsubseteq$ :  $\text{Mother} \sqsubseteq \text{Parent}$

- Classes/concepts are actually a set of individuals
- We can distinguish different types of concepts:
  - Atomic concepts: Cannot be further decomposed (i.e. Person)
  - Incomplete concepts (defined by  $\sqsubseteq$ )
  - Complete concepts (defined by  $\equiv$ )
- Example incomplete concept definition:
  - $\text{Man} \sqsubseteq \text{Person} \sqcap \text{Male}$
  - Intended meaning: If an individual is a man, we can conclude that it is a person and male.
  - $\text{Man}(x) \Rightarrow \text{Person}(x) \wedge \text{Male}(x)$
- Example complete concept definition:
  - $\text{Man} \equiv \text{Person} \sqcap \text{Male}$
  - Intended meaning: Every individual which is a male person is a man, **and** every man is a male person.
  - $\text{Man}(x) \Leftrightarrow \text{Person}(x) \wedge \text{Male}(x)$

- Roles relate to individuals to each other
  - I.e. `directedBy(Pool Sharks, Edwin Middleton)`, `hasChild(Jonny, Sue)`
- Roles have a **domain** and a **range**
- Example:
  - `Domain(directedBy, Movie)`
  - `Range(directedBy, Person)`
  - Given the above definitions we can conclude that Pool Sharks is a move and that Edwin Middleton is (was) a person.
- Functional Roles
  - Roles which have exactly one value
  - Usually used with primitive datavalues
  - A special case of (unqualified) number restriction  $\leq 1 R$

- Transitive Roles
  - Example: `hasAncestor`  
Simple in a rule language: `hasAncestor(X,Z) :- hasAncestor(X,Y), hasAncestor(Y,Z)`.
  - Requires more than one variable!
  - Transitivity can be captured in DLs by role hierarchies and transitive roles
- Symmetric Roles
  - Roles which hold in both directions
  - I.e. `hasSpouse`, `hasSibling`
- Inverse Roles
  - Roles are directed, but each role can have an inverse
  - I.e. `hasParent`  $\equiv$  `hasChild`  
`hasChild(X,Y)  $\Leftrightarrow$  hasChild(Y,X)`

- Typically a DL knowledge base (KB) consists of two components
  - Tbox (terminology): A set of inclusion/equivalence axioms denoting the conceptual schema/vocabulary of a domain
    - $\text{Bear} \sqsubseteq \text{Animal} \sqcap \text{Large}$
    - $\text{transitive}(\text{hasAncestor})$
    - $\text{hasChild} \equiv \text{hasParent}$
  - Abox (assertions): Axioms, which describe concrete instance data and holds assertions about individuals
    - $\text{hasAncestor}(\text{Susan}, \text{Granny})$
    - $\text{Bear}(\text{Winni Puh})$
- From a theoretical point of view this division is arbitrary but it is a useful simplification

- Smallest propositionally closed DL is ALC
  - Only atomic roles
  - Concept constructors:  $\sqcup$ ,  $\sqcap$ ,  $\neg$
  - Restricted use of quantifiers:  $\exists$ ,  $\forall$
- “Propositionally closed” Logic in general:
  - Provides (implicitly or explicitly) conjunction, union and negation of class descriptions
- Example:
  - $\text{Person} \sqcap \forall \text{hasChild}.(\text{Doctor} \sqcup \exists \text{hasChild}.\text{Doctor})$

- What can we express in ALC?
- ALC concept descriptions can be constructed as following:

$C, D \longrightarrow A$		(atomic concept)
$\top$		(universal concept)
$\perp$		(bottom concept)
$C \sqcap D$		(intersection)
$C \sqcup D$		(disjunction)
$\neg C$		(negation)
$\forall R.C$		(value restriction)
$\exists R.C$		(existential quantification)

- Individual assertions :
  - $a \in C$
  - Mary is a Woman.
- Role assertions:
  - $\langle a, b \rangle \in R$
  - E.g. Marry loves Peter.
- Axioms:
  - $C \sqsubseteq D$
  - $C \equiv D$ , because  $C \equiv D \Leftrightarrow C \sqsubseteq D$  and  $D \sqsubseteq C$
  - E.g.: A Dog is an animal. A man is a male Person.

- Description Logics are actually a family of related logics
  - Difference in expressivity and features, as well as complexity of inference
- Description Logics follow a naming schema according to their features
  - ALC = *Attributive Language with Complements*
  - S often used for ALC extended with transitive roles
- Additional letters indicate other extensions, e.g.:
  - H for role hierarchy
  - O for nominals, singleton classes
  - I for inverse roles (e.g.,  $\text{isChildOf} \equiv \text{hasChild}^{-}$ )
  - N for number restrictions
  - Q for qualified number restrictions
  - F for functional properties
  - R for limited complex role inclusion axioms, role disjointness
  - (D) for datatype support

- Semantics follow standard FOL model theory
  - Description Logics are a fragment of FOL
- The vocabulary is the set of names (concepts and roles) used
  - I.e. Mother, Father, Person, knows, isRelatedTo, hasChild, ...
- An interpretation  $I$  is a tuple  $(\Delta^I, \cdot^I)$ 
  - $\Delta^I$  is the domain (a set)
  - $\cdot^I$  is a mapping that maps:
    - Names of objects (individuals) to elements of the domain
    - Names of unary predicates (classes/concepts) to subsets of the domain
    - Names of binary predicates (properties/roles) to subsets of  $\Delta^I \times \Delta^I$

- The semantics of DL are based on standard First Order Model theory
- A translation is usually very straightforward, according to the following correspondences (for ALC):
  - A description is translated to a first-order formula with one free variable
  - An individual assertion is translated to a ground atomic formula
  - An axiom is translated to an implication, closed under universal implication
- More complex DLs can be handled in a similar way

- Mapping ALC to First Order Logic:

$A$ (atomic concept)	$A(x)$
$\top$	$\top$
$\perp$	$\perp$
$C \sqcap D$	$tr(C) \wedge tr(D)$
$C \sqcup D$	$tr(C) \vee tr(D)$
$\neg C$	$\neg tr(C)$
$\forall R.C$	$\forall y : R(x, y) \rightarrow tr(C, y)$
$\exists R.C$	$\exists y : R(x, y) \wedge tr(C, y)$
$a \in A$	$A(a)$
$\langle a, b \rangle \in R$	$R(a, b)$
$C \sqsubseteq D$	$\forall x. tr(C, x) \rightarrow tr(D, x)$
$C \equiv D$	$\forall x. tr(C, x) \leftrightarrow tr(D, x)$

- Main reasoning tasks for DL systems:
  - **Satisfiability**: Check if the assertions in a KB have a model
  - **Instance checking**: Check if an instance belongs to a certain concept
  - **Concept satisfiability**: Check if the definition of a concept can be satisfied
  - **Subsumption**: Check if a concept A subsumes a concept B (if every individual of a concept B is also of concept A)
  - **Equivalence**:  $A \equiv B \Leftrightarrow B \sqsubseteq A$  and  $A \sqsubseteq B$
  - **Retrieval**: Retrieve a set of instances that belong to a certain concept

- Reasoning Task are typically reduced to KB satisfiability  $\text{sat}(A)$  w.r.t. to a knowledge base A
  - **Instance checking**:  $\text{instance}(a, C, A) \Leftrightarrow \neg \text{sat}(A \cup \{a: \neg C\})$
  - **Concept satisfiability**:  $\text{csat}(C) \Leftrightarrow \text{sat}(A \cup \{a: \neg C\})$
  - **Concept subsumption**:  $B \sqsubseteq A \Leftrightarrow A \cup \{\neg B \sqcap C\}$  is not satisfiable  $\Leftrightarrow \neg \text{sat}(A \cup \{\neg B \sqcap C\})$
  - **Retrieval**: Instance checking for each instance in the Abox
- Note: **Reduction** of reasoning tasks to one another in polynomial time only in **propositionally closed logics**
- DL reasoners typically employ tableaux algorithms to check satisfiability of a knowledge base

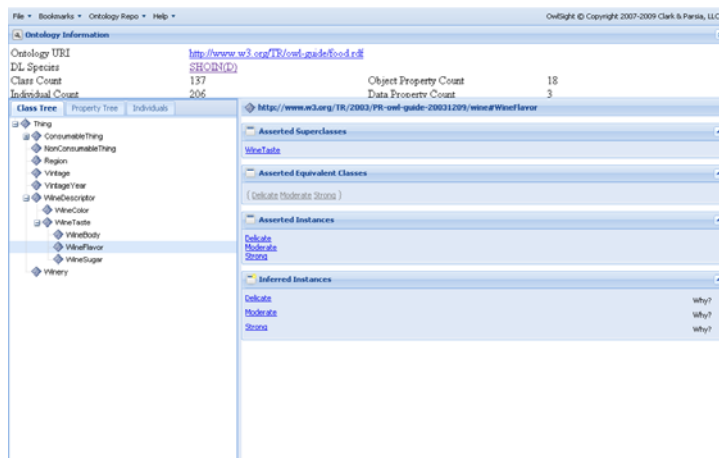
- Description Logic reasoner
  - Homepage: <https://github.com/stardog-union/pellet>
  - Written in Java and available from Dual-licensed AGPL license for open-source applications
  - Proprietary license available for commercial applications
- Sound and complete reasoner aimed at OWL-DL inference based on tableaux procedure
  - Covers all constructs in OWL-DL
  - Supporting major part of OWL 2 specification
- Integrates with popular toolkits and editors
  - E.g. Jena, Protege, TopBraid Composer

- Pellet supports expected standard DL reasoning tasks
  - Consistency, classification, realization
  - Conjunctive query answering
  - Concept satisfiability
- Additionally Pellet supports:
  - SPARQL-DL queries
  - Datatype reasoning
    - User-defined datatypes
    - N-ary datatype predicates
  - Rules support (DL-safe SWRL rules)
  - Explanation and debugging features
    - Axiom pinpointing service

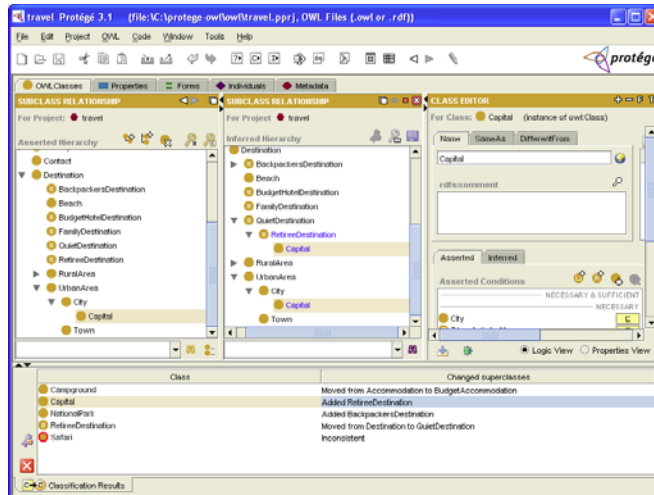


- Explanation & Debugging support
  - Motivation: It is hard to understand large and/or complex ontologies
- Examples:
  - Why is a specific subclass relation inferred?
  - Why is an ontology inconsistent?
- Pellet provides axiom pinpointing service:
  - For any inference, returns the (minimal set of) source axioms that cause the inference
- Applications can track additional provenance information
- Axiom pinpointing is the first step to explanations
  - Precise justifications (pinpoint parts of axioms) are ongoing work in Pellet's development

- Pellet is integrated in OwlSight and performs classification of loaded ontologies
  - Lightweight, web-based ontology browser: <https://www.w3.org/2001/sw/wiki/OwLSight>



- Classification Results in the Protege Editor



# LOGIC PROGRAMMING

- What is Logic Programming?
- Various different perspectives and definitions possible:
  - Computations as deduction
    - Use formal logic to express data and programs
  - Theorem Proving
    - Logic programs evaluated by a theorem prover
    - Derivation of answer from a set of initial axioms
  - High level (non-procedural) programming language
    - Logic programs do not specify control flow
    - Instead of specifying **how** something should be computed, one states **what** should be computed
  - Procedural interpretation of a declarative specification of a problem
    - A LP system procedurally interprets (in some way) a general declarative statement which only defines truth conditions that should hold

- Logic Programming is based on a subset of First Order Logic called Horn Logic
- Horn Logic can serve as a simple KR formalism and allows to express
  - IF <condition> THEN <result> rules
- Under certain restrictions reasoning over knowledge bases based on such rules is **decideable** (in contrast to general ATP within First Order Logic)

- **Syntactically** a LP rule is a **First Order Logic Horn Clause**
- The semantics of LP are different from the standard Tarski style FOL semantics (details later)
  - Herbrand semantics differ from FOL semantics in the structures it considers to be models (details later)
  - **Minimal model semantics**
- A FOL Horn clause is a disjunction of literals with one positive literal, with all variables universally quantified:
  - $(\forall) \neg C_1 \vee \dots \vee \neg C_n \vee H$
- This can be rewritten to closer correspond to a rule-like form:
  - $(\forall) C_1 \wedge \dots \wedge C_n \rightarrow H$
- In LP systems usually the following (non First Order Logic) syntax is used:
  - $H :- C_1, \dots, C_n$
- Such rules can be evaluated very **efficiently**, since the resolution of two Horn clauses is again a Horn clause

- The LP vocabulary consists of:
  - Constants: b, cow, "somestring"
  - Predicates: p, loves
  - Function symbols: f, fatherOf
  - Variables: x, y
- Terms can be:
  - Constants
  - Variables
  - Constructed terms (i.e. function symbol with arguments)
- Examples:
  - cow, b, Jonny,
  - loves(John)

- From terms and predicates we can build atoms:
  - For n-ary predicate symbol  $p$  and terms  $t_1, \dots, t_n$ ,  $p(t_1, \dots, t_n)$  is an atom
  - A ground atom is an atom without variables
- Examples:
  - $p(x)$
  - $\text{loves}(\text{Jonny}, \text{Mary}), \text{worksAt}(\text{Jonny}, \text{SomeCompany})$
  - $\text{worksAt}(\text{loves}(\text{Mary}), \text{SomeCompany})$
- Literals
  - A literal is a an atom or its negation
  - A positive literal is an atom
  - A negative literal is a negated atom
  - A ground literal is a literal without variables

- Rules
  - Given a rule of the form  $H :- B_1, \dots, B_n$  we call
    - $H$  the **head** of the rule (its consequent)
    - $B_1 \dots B_n$  the **body** of the rule (the antecedent or conditions)
  - The head of the rule consists of one positive literal  $H$
  - The body of the rule consists of a number of literals  $B_1, \dots, B_n$
  - $B_1, \dots, B_n$  are also called **subgoals**
- Examples:
  - $\text{parent}(x) :- \text{hasChild}(x,y)$
  - $\text{father}(x) :- \text{parent}(x), \text{male}(x)$
  - $\text{hasAunt}(z,y) :- \text{hasSister}(x,y), \text{hasChild}(x,z)$

- Facts denote assertions about the world:
  - A rule without a body (no conditions)
  - A ground atom
- Examples:
  - hasChild(Jonny, Sue)
  - Male(Jonny).
- Queries allow to ask questions about the knowledge base:
  - Denoted as a rule without a head:
    - ?- B1,....,Bn.
- Examples:
  - ? - hasSister(Jonny,y), hasChild(Jonny , z) gives all the sisters and children of Jonny
  - ? - hasAunt(Mary,y) gives all the aunts of Mary
  - ?- father(Jonny) answers if Jonny is a father

- There are two main approaches to define the semantics of LP
  1. Model theoretic semantics
  2. Computational semantics
- Model-theoretic semantics
  - Defines the meaning of a model in terms of its **minimal Herbrand model**.
- Computational semantics (proof theoretic semantics)
  - Define the semantics in terms of an **evaluation strategy** which describes how to compute a model
- These two semantics are different in style, but agree on the **minimal model**
- LP semantics is **only** equivalent to standard FOL semantics
  - Concerning ground entailment
  - As long as LP is not extended with **negation**

- Semantics of LP vs Semantics of FOL
  - Semantics LP defined in terms of **minimal Herbrand model**
    - Only one minimal model
  - Semantics FOL defined in terms of **First-Order models**
    - Typically, infinitely many First-Order models
  - The minimal Herbrand model is a First-Order model
  - Every Herbrand model is a First-Order model
  - There exist First-Order models which are not Herbrand models
- Example:
  - $p(a), p(x) \rightarrow q(x)$
  - The Minimal Herbrand model:  $\{p(a), q(a)\}$
  - This is actually the only Herbrand model!
  - First-Order models:  $\{p(a), q(a)\}, \{p(a), q(a), p(b)\}, \text{ etc.}$

- Recall:
  - Terms not containing any variables are ground terms
  - Atoms not containing any variables are ground atoms
- The **Herbrand Universe U** is the set of all ground terms which can be formed from
  - Constants in a program
  - Function symbols in a program
- Example:  $a, b, c, f(a)$
- The **Herbrand Base B** is the set of all ground atoms which can be built from
  - Predicate symbols in a program
  - Ground terms from U
- Example:  $p(a), q(b), q(f(a))$

- A **Herbrand Interpretation I** is a subset of the Herbrand Base B for a program
  - The domain of a Herbrand interpretation is the Herbrand Universe U
  - Constants are assigned to themselves
  - Every function symbol is interpreted as the function that applies it
    - If f is an n-ary function symbol ( $n > 0$ ) then the mapping from  $U^n$  to U defined by  $(t_1, \dots, t_n) \rightarrow f(t_1, \dots, t_n)$  is assigned to f

- A **Herbrand Model M** is a Herbrand Interpretation which makes every formula true, so:
  - Every fact from the program is in M
  - For every rule in the program: If every positive literal in the body is in M, then the literal in the head is also in M
- The model of a Logic Program P is the **minimal Herbrand Model**
  - This least Herbrand Model is the intersection of all Herbrand Models
  - Every program has a Herbrand Model. Thus every model also has a minimal Herbrand Model.
  - This model is **uniquely** defined only for programs without negation
- A very intuitive and easy way to capture the semantics of LP
- As soon as negation is allowed a unique minimal model is not guaranteed anymore



- How do we handle negation in Logic Programs?
- Horn Logic only permits negation in limited form
  - Consider  $(\forall) \neg C1 \vee \dots \vee \neg Cn \vee H$
- Special solution: **Negation-as-failure (NAF)**:
  - Whenever a fact is not entailed by the knowledge base, its negation is entailed
  - This is a form of “Default reasoning”
  - This introduces non-monotonic behavior (previous conclusions might need to be revised during the inference process)
- NAF is not classical negation and pushes LP **beyond** classical First Order Logic
- This allows a form of negation in rules:
  - $(\forall) C1 \wedge \dots \wedge Ci \wedge \text{not } Cn \rightarrow H$
  - $H :- B1, \dots Bi, \text{not } Bn$

- Logic Programs can also contain **recursion**
- Example:
  - `ancestor(x,y) :- hasParent(x, y)`
  - `ancestor(x,z) :- ancestor(x,y), ancestor(y,z).`
- This is a problem as soon as negation is allowed since the minimal model is not uniquely defined anymore
- It is useful to consider this using a dependency graph
  - A predicate is a node in the graph
  - There is a directed edge between predicates q and p if they occur in a rule where q occurs in the head and p in the body.
  - If the dependency graph contains a cycle then the program is recursive

- As soon as **negation** is allowed, cycles in a dependency graph become problematic.
  - Example: What is the meaning of  $\text{win}(x) :- \text{not win}(x)$  ?
- A Solution: **Stratification**
  - Mark edges with negation in the dependency graph
  - Separate predicates which are connected through a **positive edge** in a individual **stratum**
  - Strata can be (partially) ordered
  - If each predicate occurs only in one stratum, then the program is called **stratifiable**
  - Each stratum can be evaluated as usual and independently from other strata
  - This guarantees a unique interpretation of a Logic Program using negation

- Classical Logic Programming
  - Allows function symbols
  - Does not allow negation
  - Is **turing complete**
- Full Logic Programming is **not decidable**
  - Prolog programs are not guaranteed to terminate
- Several ways to guarantee the evaluation of a Logic Program
  - One is to enforce **syntactical restrictions**
  - This results in subsets of full logic programming
  - **Datalog** is such a subset

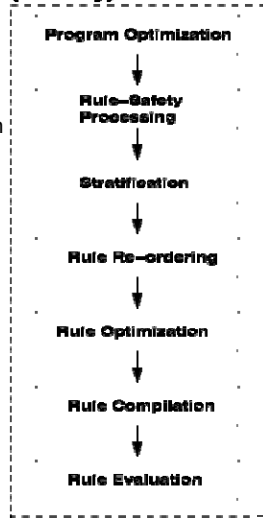
- Datalog is a syntactic subset of Prolog
  - Originally a rule and query language for deductive databases
- Considers knowledge bases to have two parts
  - Extensional Database (EDB) consists of facts
  - Intentional Database (IDB) consists of non-ground rules
- Restrictions:
  1. Datalog disallows function symbols
  2. Imposes **stratification** restrictions on the use of recursion + negation
  3. Allows only range restricted variables (**safe variables**)
- Safe Variables:
  - Only allows range restricted variables, i.e. each variable in the conclusion of a rule must also appear in a not negated clause in the premise of this rule.
  - This limits evaluation of variables to finitely many possible bindings.

- The typical reasoning task for LP systems is **query answering**
  - Ground queries, i.e.  $?- \text{loves}(\text{Mary}, \text{Joe})$
  - Non-ground query, i.e.  $?- \text{loves}(\text{Mary}, x)$
- Non-ground queries can be reduced to a series of ground queries
  - $?- \text{loves}(\text{Mary}, x)$
  - Replace  $x$  by every possible value
- In Logic Programming ground queries are equivalent to entailment of facts
  - Answering  $?- \text{loves}(\text{Mary}, \text{Joe})$  w.r.t. a knowledge base  $A$  is equivalent to checking  
 $A \models \text{loves}(\text{Mary}, \text{Joe})$

- Java based Datalog reasoner
  - Developed at STI Innsbruck
  - Freely available open source project
  - Homepage: <http://www.iris-reasoner.org/>
- Extensions:
  - Stratified / Well-founded default negation
  - XML Schema data types
  - Various built-in predicates (Equality, inequality, assignment, unification, comparison, type checking, arithmetic, regular expressions,... )
- Highly modular and includes different reasoning strategies
  - Bottom-up evaluation with Magic Sets optimizations (forward-chaining)
  - Top-down evaluation by SLDNF resolution (backward-chaining)

- An example of a concrete combination of components within IRIS:
  - Program Optimization
    - Rewriting techniques from deductive DB research (e.g. Magic sets rewriting)
  - Safety Processing & Stratification
    - Ensure specific syntactic restrictions
  - Rule Re-ordering
    - Minimize evaluation effort based on dependencies between expressions
  - Rule Optimizations
    - Join condition optimization
    - Literal re-ordering
  - Rule compilation
    - Pre-indexing
    - Creation of „views“ on required parts of information

**(Locally) Stratified**



- Example program (also see online demo at <http://www.iris-reasoner.org/demo>):

```
man('homer').
woman('marge').
hasSon('homer','bart').
isMale(?x) :- man(?x).
isFemale(?x) :- woman(?x).
isMale(?y) :- hasSon(?x,?y).
```

- Query:
 

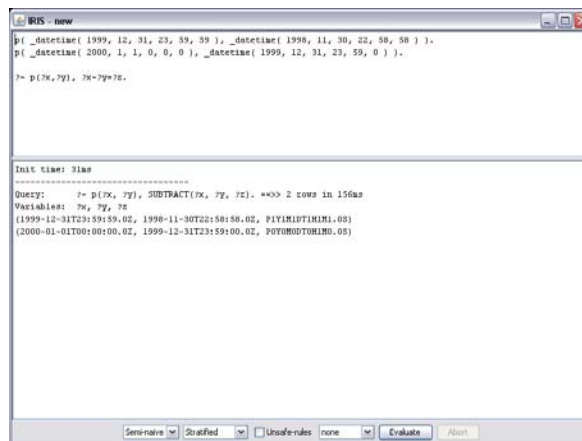
```
?-isMale(?x).
```

- Output:
 

```
Init time: 14ms
-----
Query: ?- isMale(?x). ==>> 2 rows in 1ms
Variables: ?x
('bart')
('homer')
```

73

- IRIS performing computations using semi-naïv evaluation in combination with stratification and complex datatypes



```
IRIS - new
p( _datetime( 1999, 12, 31, 23, 59, 59 ), _datetime( 1998, 11, 30, 22, 58, 58 ) ).
p( _datetime( 2000, 1, 1, 0, 0, 0 ), _datetime( 1999, 12, 31, 23, 59, 0 ) ).

?- p(?X,?Y), ?X<=?Y.

Init time: 31ms
-----
Query:      ?- p(?X, ?Y), SUBTRACT(?X, ?Y, ?Z). ==>> 2 rows in 156ms
Variables: ?X, ?Y, ?Z
(1999-12-31T23:59:59.0Z, 1998-11-30T22:58:58.0Z, P1Y1M10T1H1M1.0S)
(2000-01-01T00:00:00.0Z, 1999-12-31T23:59:00.0Z, P0Y0M0T0H0M0.0S)
```

74

# SUMMARY

## Theorem Proving Summary

- Resolution is a sound and complete inference procedure for First-Order Logic
- Due to its complexity and remaining limitations FOL is often not suitable for practical applications
- Often formalisms expressivity and complexity results are more appropriate:
  - Description Logics
  - Logic Programming

## Description Logic Summary



- Description Logics are a syntactic fragment of First Order Logic, based on basic building blocks
  - Concepts
  - Roles
  - Individuals
- Limited constructs for building complex concepts, roles, etc.
  - Many different Description Logics exist, depending on choice of constructs
- Inference in Description Logics focuses on consistency checking and classification
  - Main reasoning task: Subsumption
  - Usually reasoning tasks in DLs can all be reduced to satisfiability checking
- Efficient Tbox (schema) reasoning
- ABox reasoning (query answering) do not scale so well

77

## Logic Programming Summary



- Logic Programming (without negation) is **syntactically** equivalent to Horn subset of First Order Logic
- The **semantics** of a Logic Program are however based on its minimal Herbrand Model
- Logic Programming comes in various variants for different applications (as programming language, for knowledge representation)
  - Full Logic Programming including function symbols is not decidable
  - Datalog is a syntactic restriction of LP, with desirable computational properties
  - Negation-as-failure introduced non-monotonic behavior and pushes LP beyond the expressivity of First Order Logic
- A typical inference task for LP engines is conjunctive query answering

78

# REFERENCES

## References and Further Information

- **Mandatory Reading:**
  - Schöning, U., Logic for Computer Scientists (2<sup>nd</sup> edition), 2008, Birkhäuser
    - Chapter 1 & 2: Normal Forms, Resolution
    - Chapter 3: Horn Logic, Logic Programming
  - Baader, F. et al., The Description Logic Handbook, 2007, Cambridge University Press
    - Chapter 2: Basic Description Logics
- **Further Reading:**
  - Lloyd, J.W., Foundations of logic programming, 1984, Springer
  - Robinson, A. and Voronkov, A. Handbook of Automated Reasoning, Volume I, 2001, MIT Press
    - Chapter 2: Resolution Theorem Proving
  - Ullman, J. D., Principles of Database and Knowledge-Base Systems, Volume I, 1988, Computer Science Press
    - Chapter 3: Logic as a Data Model (Logic Programming & Datalog)
- **Wikipedia links:**
  - [http://en.wikipedia.org/wiki/Logic\\_programming](http://en.wikipedia.org/wiki/Logic_programming)
  - [http://en.wikipedia.org/wiki/Description\\_logic](http://en.wikipedia.org/wiki/Description_logic)
  - [http://en.wikipedia.org/wiki/Theorem\\_proving](http://en.wikipedia.org/wiki/Theorem_proving)



## Next Lecture



#	Title
0.1	Propositional Logic
0.2	Predicate Logic
1	Introduction
2	Reasoning
<b>3</b>	<b>Search Methods</b>
4	CommonKADS
5	Problem-Solving Methods
6	Planning
7	Software Agents
8	Rule Learning
9	Inductive Logic Programming
10	Neural Networks
11	Semantic Web and Services
12	Exam



81

## Questions?



Email: [anna.fensel@uibk.ac.at](mailto:anna.fensel@uibk.ac.at)  
Web: <http://bit.ly/2mwtf1h>  
Twitter: @alphaverda  
Facebook: <https://www.facebook.com/STIInnsbruck/>

82