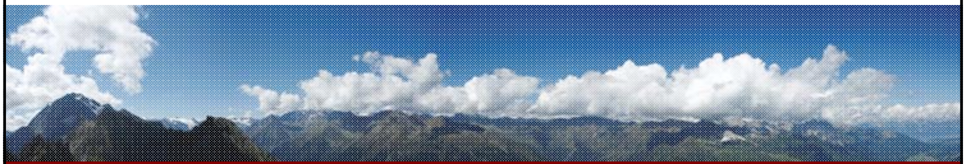
Intelligent Systems

Reasoning

Dr Anna Fensel



Where are we?

STI · INNSBRUCK

Agenda

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

MOTIVATION

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

Motivation

- 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

STI · INNSBRUCK

TECHNICAL SOLUTIONS

Theorem Proving and Resolution, Description Logics, and Logic Programming

STI · INNSBRUCK

THEOREM PROVING

Introduction to Theorem Proving and Resolution

Introduction – Basic Notions

STI · INNSBRUCK

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

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

where C is a conclusion sequent, and P_i's are premises sequents .
  - If an infererence rule has premises that are taken to be true (called an **axiom**), its conclusion automatically holds.
  - Example: Modus Ponens: From P, P → Q infer Q,
    Universal instantiation: From (∀x)p(x) infer p(A)

- Theorems:
  - Expressions that can be derived from the axioms and the rules of inference.

Resolution - Principle

STI · INNSBRUCK

- 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* ⊨ *Q*
  - This means that every interpretation *I* that satisfies *KB* also satisfies *Q*
  - Any interpretation *I* only satisfies either *Q* or ¬*Q*, but not both
  - Therefore, if in fact *KB* ⊨ *Q* holds, an interpretation that satisfies *KB*, satisfies *Q* and does not satisfy ¬*Q*
    - → **Hence KB ∪ {¬*Q*} is unsatisfiable, i.e., it is false under all interpretations**

Resolution - Principle

STI · INNSBRUCK

- 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. : $l_1 \vee \ldots \vee l_n$
  - A formula in clause normal form conjunction of a set of clauses
    - E.g.: $C_1 \wedge \ldots \wedge C_n = (l_n \vee \ldots \vee l_m) \wedge \ldots \wedge (l_i \vee \ldots \vee l_j)$
- 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

11

Resolution - The Resolution Rule

STI · INNSBRUCK

- 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, ai is taken to be the complement to bj).
  - The resulting clause contains all the literals that do not have complements. Formally:

$$\frac{a_1 \vee \ldots \vee a_i \vee \ldots \vee a_n, \quad b_1 \vee \ldots \vee b_j \vee \ldots \vee b_m}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_m}$$

  Where $a_1, \ldots, a_n, b_1, \ldots, b_m$ 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 σ.
  - σ Is then applied to the resolvent

12

Resolution - The Resolution Rule

- 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 last lecture) 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)) v P(x, f(y)) v Q(y) and ¬P(z, f(a)) v ¬ Q(z)
  - P(x, f(a)) and ¬P(z, f(a)) **unify** with substitution σ = {x/z}
  - Therefore, we derive the **resolvent clause** (to which σ is applied): P(z, f(y)) v Q(y) v ¬ Q(z)

Resolution – Complete Resolution Algorithm

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.

Resolution – Algorithm in Pseudocode

```
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 ⊢Q, and failure otherwise
  KB = union(KB, ¬Q)
  while false not in KB do
      pick 2 sentences, S1 and S2, in KB that contain
      literals that unify
             if none return "failure"
      resolvent = resolution-rule(S1, S2)
      KB = union(KB, resolvent)
  return "success"
```


Resolution - Converting to Clause Normal Form

- 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
  - $\neg(\neg a) = a$
  - $\neg(a \wedge b) = \neg a \vee \neg b$
  - $\neg(a \vee b) = \neg a \wedge \neg b$
  - $\neg(\exists X)\ a(X) = (\forall X)\ \neg a(X)$
  - $\neg(\forall X)\ b(X) = (\exists X)\ \neg b(X)$

Resolution - Converting to Clause Normal Form

STI · INNSBRUCK

- 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**

Resolution - Converting to Clause Normal Form

STI · INNSBRUCK

- Step 5: Skolemization
  - Skolem constant
    - $(\exists X)(dog(X))$ may be replaced by dog(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.

$(\forall X)(\exists Y)(mother(X,Y)) \Rightarrow (\forall X)mother(X,m(X))$

$(\forall X)(\forall Y)(\exists Z)(\forall W)(foo\ (X,Y,Z,W))$

$\Rightarrow (\forall X)(\forall Y)(\forall W)(foo(X,Y,f(X,Y),w))$

Resolution - Converting to Clause Normal Form

STI · INNSBRUCK

- 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

STI · INNSBRUCK

- (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
    - $\forall(x)\ (dog(X) \rightarrow animal(X))$
    - dog(fido)
  - Modus Ponens and {fido/X}
    - animal(fido)
    - $\forall(Y)\ (animal(Y) \rightarrow die(Y))$
  - Modus Ponens and {fido/Y}
    - die(fido)

20

Resolution – Complete Example

- Equivalent proof by Resolution

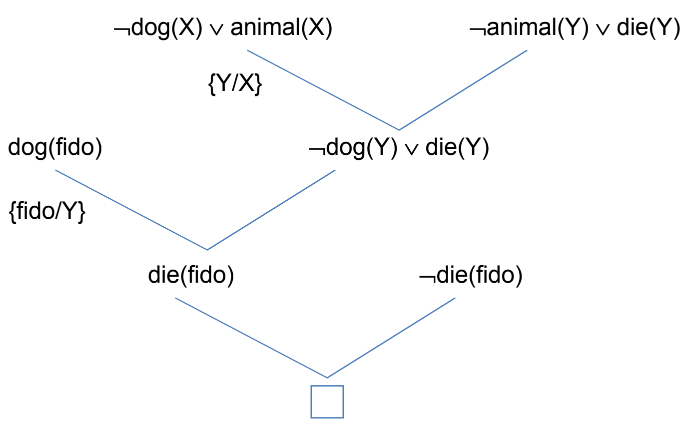
  – Convert predicates to clause normal form

Predicate form	Clause form
1. ∀(x) (dog(X) → animal(X))	¬dog(X) ∨ animal(X)
2. dog(fido)	dog(fido)
3. ∀(Y) (animal(Y) → die(Y))	¬animal(Y) ∨ die(Y)

  – Negate the conclusion

4. ¬die(fido)	¬die(fido)

Resolution - Complete Example



Resolution proof for the “dead dog” problem

Resolution – Further Considerations

STI · INNSBRUCK

- 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

Resolution – Strategies

STI · INNSBRUCK

- Order of clause combination is important
  - N clauses → 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

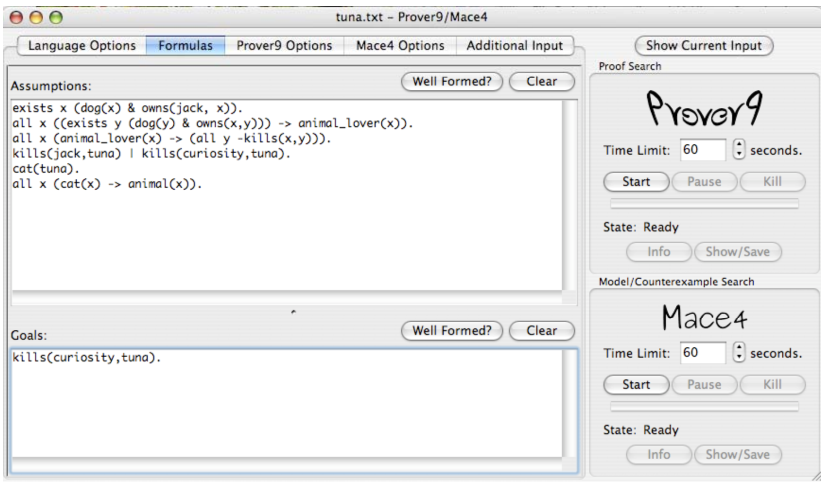
Concrete System: Prover9

STI · INNSBRUCK

- 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 Input

STI · INNSBRUCK



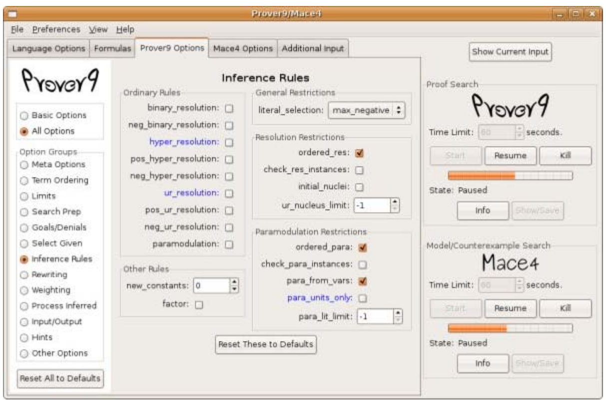
Concrete System: Prover9 Proof

STI · INNSBRUCK

```
============================== 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 ==========================
```


Concrete System: Prover9 Options

STI · INNSBRUCK



Limitations of First-OrderTheorem Proving

STI · INNSBRUCK

- Entailment: KB ⊨ 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's a proof, we'll 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

29

Limitations of First-OrderTheorem Proving

STI · INNSBRUCK

- 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 decideable) formalisms or formalisms with different expressivity are more suitable:
  - Description Logics
  - Logic Programming

30

STI · INNSBRUCK

DESCRIPTION LOGICS

Description Logic

STI · INNSBRUCK

- 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

Description Logic Syntax

STI · INNSBRUCK

- **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 ⊔: Man ⊔ Woman
  - Intersection ⊓ : Doctor ⊓ Mother
  - Existential restriction ∃: ∃hasChild.Doctor (some child is a doctor)
  - Value(universal) restriction ∀: ∀hasChild.Doctor (all children are doctors)
  - Complement /negation¬: Man ⊑ ¬Mother
  - Number restriction ≥n, ≤n
- **Axioms**
  - Subsumption ⊑ : Mother ⊑ Parent

Description Logic Syntax - Concepts

STI · INNSBRUCK

- 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 ⊑)
  - Complete concepts (defined by ≡)
- Example incomplete concept defintion:
  - Man ⊑ Person ⊓ Male
  - Intended meaning: If an individual is a man, we can conclude that it is a person and male.
  - Man(x) ⇒ Person(x) ∧ Male(x)
- Example complete concept definition:
  - Man ≡ Person ⊓ Male
  - Intended meaning: Every individual which is a male person is a man, **and** every man is a male person.
  - Man(x) ⇔ Person(x) ∧ Male(x)

Description Logic Syntax- Roles

- 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 ≤1 *R*

Description Logic Syntax- Roles

- 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 ≡ hasChild
    hasChild(X,Y) ⇔ hasChild(Y,X)

Description Logic Knowledge Bases

STI · INNSBRUCK

- 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
    - Bear ⊑ Animal ⊓ Large
    - transitive(hasAncestor)
    - hasChild ≡ hasParent
  - Abox (assertions): Axioms, which describe concrete instance data and holds assertions about individuals
    - hasAncestor(Susan, Granny)
    - Bear(Winni Puh)
- From a theoretical point of view this division is arbitrary but it is a useful simplification

37

A basic Description Logic - ALC

STI · INNSBRUCK

- Smallest propositionally closed DL is ALC
  - Only atomic roles
  - Concept constructors: ⊔ , ⊓, ¬
  - Restricted use of quantifiers: ∃, ∀
- "Propositionally closed" Logic in general:
  - Provides (implicitly or explicitly) conjunction, union and negation of class descriptions
- Example:
  - Person ⊓ ∀hasChild.(Doctor ⊔ ∃hasChild.Doctor)

38

A basic Description Logic - ALC

STI · INNSBRUCK

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

$$
\begin{aligned}
C, D \longrightarrow\ & A \mid && \text{(atomic concept)} \\
& \top \mid && \text{(universal concept)} \\
& \bot \mid && \text{(bottom concept)} \\
& C \sqcap D \mid && \text{(intersection)} \\
& C \sqcup D \mid && \text{(disjunction)} \\
& \neg C \mid && \text{(negation)} \\
& \forall R.C \mid && \text{(value restriction)} \\
& \exists R.C \mid && \text{(existential quantification)}
\end{aligned}
$$

A basic Description Logic - ALC

STI · INNSBRUCK

- Individual assertions :
  - a ∈ C
  - Mary is a Woman.
- Role assertions:
  - ⟨a, b⟩ ∈ R
  - E.g. Marry loves Peter.
- Axioms:
  - C ⊑ D
  - C ≡ D, because C ≡ D ⇔ C ⊑ D and D ⊑ C
  - E.g.: A Dog is an animal. A man is a male Person.

The Description Logic Family

- 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., isChildOf ≡ 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

41

Description Logic - Semantics

- 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 $(\Delta^I, \cdot^I)$
  - Δ^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 $\Delta^I \times \Delta^I$

42

Description Logic - Semantics

STI · INNSBRUCK

- 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

Description Logic - Semantics

STI · INNSBRUCK

- Mapping ALC to First Order Logic:

A (atomic concept)	$A(x)$
$\top$	$\top$
$\bot$	$\bot$
$C \sqcap D$	$tr(C) \wedge tr(D)$
$C \sqcup D$	$tr(C) \vee tr(C)$
$\neg C$	$\neg tr(C)$
$\forall R.C$	$\forall y : R(x,y) \rightarrow tr(C,y)$
$\exists R.C$	$\exists y : R(x,y) \wedge tr(C,y)$

$a \in A$	$A(a)$
$\langle a, b \rangle \in R$	$R(a,b)$

$C \sqsubseteq D$	$\forall x.tr(C,x) \rightarrow tr(D,x)$
$C \equiv D$	$\forall x.tr(C,x) \leftrightarrow tr(D,x)$

Description Logic - Reasoning

STI · INNSBRUCK

- 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

45

Description Logic - Reasoning

STI · INNSBRUCK

- Reasoning Task are typically reduced to KB satisfiability sat(A) w.r.t. to a knowledge base A
  - **Instance checking**: instance(a,C, A) $\Leftrightarrow \neg$sat(A $\cup$ {a: $\neg$ C})
  - **Concept satisfiability**: csat(C) $\Leftrightarrow$ sat(A $\cup$ {a: $\neg$ C})
  - **Concept subsumption**: $B \sqsubseteq A \Leftrightarrow A \cup \{\neg B \sqcap C\}$ is not satisfiable $\Leftrightarrow$ $\neg$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

46

Concrete System: Pellet

STI · INNSBRUCK

- Description Logic reasoner
  - Homepage: http://clarkparsia.com/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
- Comprehensive hands-on tutorial available:
  - http://clarkparsia.com/pellet/tutorial/iswc09

47

Concrete System: Pellet

STI · INNSBRUCK

- 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

48

Concrete System: Pellet

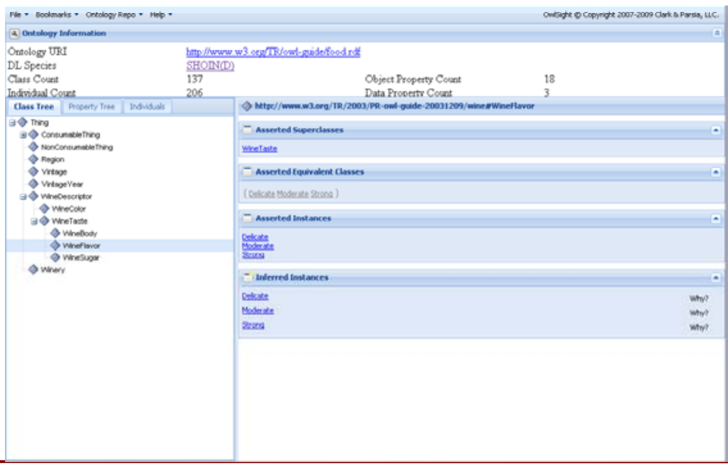
STI · INNSBRUCK

- 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 developement

Concrete System: Pellet

STI · INNSBRUCK

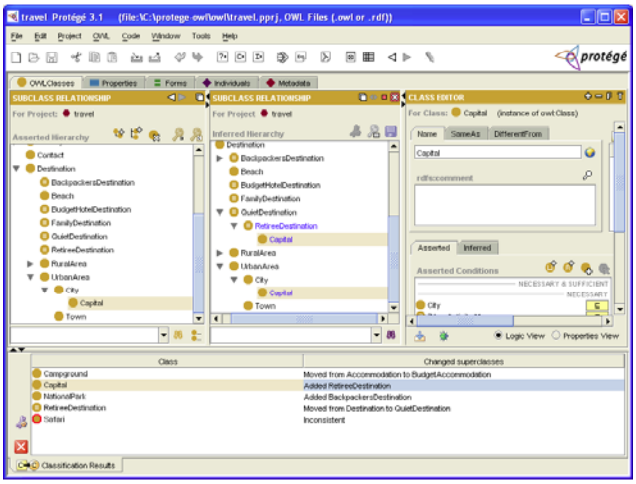
- Pellet is integrated in OwlSight and performs classification of loaded ontologies
  - Lightweight , web-based ontology browser: http://pellet.owldl.com/owlsight/



Concrete System: Pellet

STI · INNSBRUCK

- Classification Results in the Protege Editor



STI · INNSBRUCK

LOGIC PROGRAMMING

Logic Programming

STI · INNSBRUCK

- 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-precedural) programming language
    - Logic programs do not specifcy 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 systems procedurally interprets (in some way) a general declarative statement which only defines truth conditions that should hold

Logic Programming Basics

STI · INNSBRUCK

- 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)

Logic Programming Basics – Horn Logic

STI · INNSBRUCK

- Syntactically a LP rule is a First Order Logic Horn Clause
- The semantics of LP are different form 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:
  - $(\forall) \neg C_1 \vee \ldots \vee \neg C_n \vee H$
- This can be rewritten to closer correspond to a rule-like form:
  - $(\forall) C_1 \wedge \ldots \wedge C_n \rightarrow H$
- In LP systems usually the following (non First Order Logic) syntax is used:
  - $H :- C_1,\ldots,C_n$
- Such rules can be evaluated very **efficiently**, since the resolution of two Horn clauses is a again a Horn clause

Logic Programming – Syntax

STI · INNSBRUCK

- 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)

Logic Programming – Syntax

STI · INNSBRUCK

- From terms and predicates we can build atoms:
  - For n-ary predicate symbol p and terms t_1, ..., t_n, p(t_1, ..., t_n) is an atom
  - A ground atom is an atom without variables
- Examples:
  - p(x)
  - loves(Jonny, Mary), worksAt(Jonny, SomeCompany)
  - worksAt(loves(Mary), 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 literals is a literal without variables

57

Logic Programming – Syntax

STI · INNSBRUCK

- Rules
  - Given a rule of the form H :- B_1,...,B_n we call
    - H the **head** of the rule (its consequent)
    - B_1 …. 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, ..., B_n
  - B_1, …, B_n are also called **subgoals**
- Examples:
  - parent(x) :- hasChild(x,y)
  - father(x) :- parent(x), male(x)
  - hasAunt(z,y) :- hasSister(x,y), hasChild(x,z)

58

Logic Programming – Syntax

STI · INNSBRUCK

- 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) ansers if Jonny is a father

59

Logic Programming – Semantics

STI · INNSBRUCK

- There are two main approaches to define the semantics of LP
  1. Model theoretic semantics
  2. Computional semanitcs
- 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**

60

Logic Programming – Semantics

- 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) → 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)}, etc.

Logic Programming – Semantics

- 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 $(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n)$ 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

Logic Programming - Negation

STI · INNSBRUCK

- How do we handle negation in Logic Programs?
- Horn Logic only permits negation in limited form
  - Consider $(\forall)\ \neg C1 \lor \ldots \lor \neg Cn \lor H$
- 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:
  - $(\forall)\ C1 \land \ldots \land Ci \land \text{not } Cn \rightarrow H$
  - H :- B1, … Bi, not Bn

Logic Programming - Negation

STI · INNSBRUCK

- 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

Logic Programming - Negation

- As soon as **negation** is allowed, cycles in a dependency graph become problematic.
    - Example: What is the meaning of win(x) :- 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

Logic Programming - Subsets

- Classical Logic Programming
    - Allows function symbols
    - Does not allow negation
    - Is **turing complete**
- Full Logic Programming is **not decideable**
    - 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

Logic Programming - Datalog

STI · INNSBRUCK

- 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

69

Logic Programming - Reasoning Tasks

STI · INNSBRUCK

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

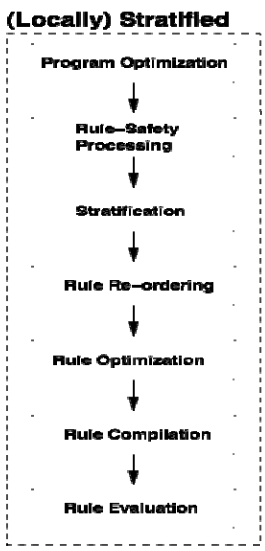
70

Concrete Logic Programming System: IRIS

- 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)

Concrete Logic Programming System: IRIS

- 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



Concrete Logic Programming System: IRIS

STI · INNSBRUCK

- 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).

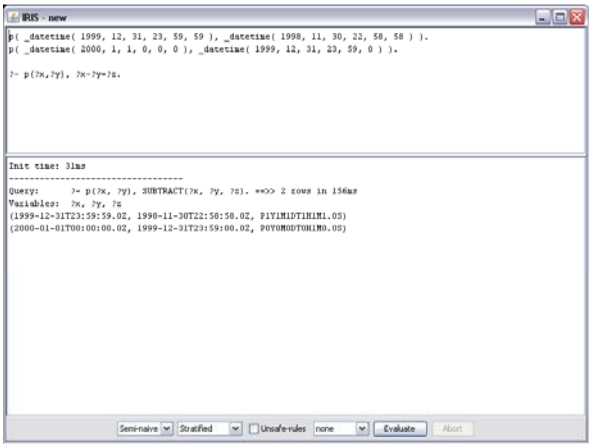
- Output:
  Init time: 14ms

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

Concrete Logic Programming System: IRIS

STI · INNSBRUCK

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



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

STI · INNSBRUCK

- 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 satisfiablity checking
- Efficient Tbox (schema) reasoning
- ABox reasoning (query answering) do not scale so well

Logic Programming Summary

STI · INNSBRUCK

- 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

STI · INNSBRUCK

REFERENCES

79

STI · INNSBRUCK

References and Further Information

- Mandatory Reading:
  - Schöning, U., Logic for Computer Scientists (2nd 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/Description_logic
  - http://en.wikipedia.org/wiki/Theorem_proving

80

Next Lecture

#	Title
1	Introduction
2	Propositional Logic
3	Predicate Logic
4	Reasoning
5	**Search Methods**
6	CommonKADS
7	Problem-Solving Methods
8	Planning
9	Software Agents
10	Rule Learning
11	Inductive Logic Programming
12	Formal Concept Analysis
13	Neural Networks
14	Semantic Web and Services

Questions?