CSC384h: Intro to Artificial Intelligence

- Knowledge Representation
  - This material is covered in chapters 7—10 of the text.
  - Chapter 7 provides a useful motivation for logic, and an introduction to some basic ideas. It also introduces propositional logic, which is a good background for first-order logic.
  - What we cover here is mainly covered in Chapters 8 and 9. However, Chapter 8 contains some additional useful examples of how first-order knowledge bases can be constructed. Chapter 9 covers forward and backward chaining mechanisms for inference, while here we concentrate on resolution.
  - Chapter 10 covers some of the additional notions that have to be dealt with when using knowledge representation in AI.

Knowledge Representation

- Consider the task of understanding a simple story.
- How do we test understanding?
- Not easy, but understanding at least entails some ability to answer simple questions about the story.

Example.

- Three little pigs

Example.

- Three little pigs

Example.

- Why couldn't the wolf blow down the house made of bricks?

- What background knowledge are we applying to come to that conclusion?
  - Brick structures are stronger than straw and stick structures.
  - Objects, like the wolf, have physical limitations. The wolf can only blow so hard.

Why Knowledge Representation?

- Large amounts of knowledge are used to understand the world around us, and to communicate with others.
- We also have to be able to reason with that knowledge.
  - Our knowledge won't be about the blowing ability of wolfs in particular, it is about physical limits of objects in general.
  - We have to employ reasoning to make conclusions about the wolf.
  - More generally, reasoning provides an exponential or more compression in the knowledge we need to store. I.e., without reasoning we would have to store a infeasible amount of information: e.g., Elephants can't fit into teacups.

Logical Representations

- AI typically employs logical representations of knowledge.

- Logical representations useful for a number of reasons:

Logical Representations

- They are mathematically precise, thus we can analyze their limitations, their properties, the complexity of inference etc.

- They are formal languages, thus computer programs can manipulate sentences in the language.

- They come with both a formal syntax and a formal semantics.

- Typically, have well developed proof theories: formal procedures for reasoning at the syntactic level (achieved by manipulating sentences).

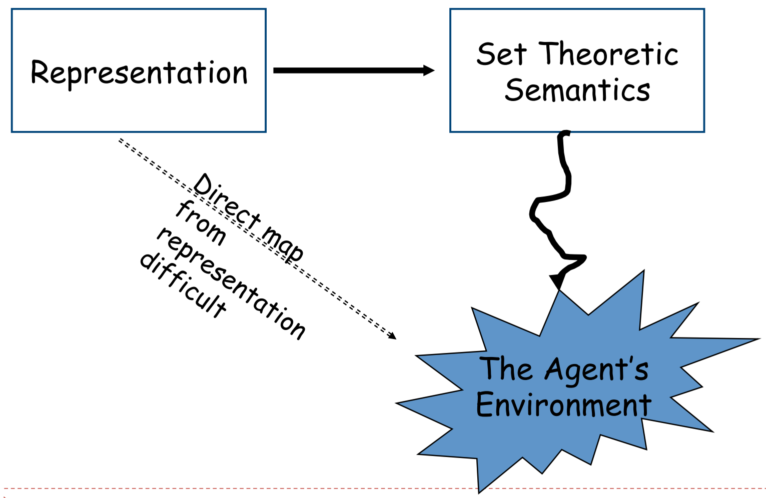
Set theoretic semantics

- Suppose our knowledge is represented in our program by some collection of data structures. We can think of these as a collection of strings (sentences).
- We want a clear mapping from this set of sentences to features of the environment. What are sentences asserting about environment?
  - In other words, we want to be able to provide an intuitive interpretation of any piece of our representation.
  - Similar in spirit to having an intuitive understanding of what individual statements in a program mean. It does not mean that it is easy to understand the whole, but it provides the means to understand the whole by understanding the parts.

Set theoretic semantics

- Set theoretic semantics facilitates both goals.
  - It is a formal characterization, and it can be used to prove a wide range of properties of the representation.
  - It maps arbitrarily complex sentences of the logic down into intuitive assertions about the real world.
  - It is based on notions that are very close to how we think about the real world. Thus it provides the bridge from the syntax to an intuitive understanding of what is being asserted.

Set theoretic semantics



Semantics Formal Details

- A set of objects. These are objects in the environment that are important for your application.
- Distinguished subsets of objects. Properties.
- Distinguished sets of tuples of objects. Relations.
- Distinguished functions mapping tuples of objects to objects. Functions.

Example

- Teaching CSC384, want to represent knowledge that would be useful for making the course a successful learning experience.

- **Objects:**
  - **students, subjects, assignments, numbers.**
- **Predicates:**
  - **difficult(*subject*), CSMajor(*student*).**
- **Relations:**
  - **handedIn(*student, assignment*)**
- **Functions:**
  - **Grade(*student, assignment*) → *number***

First Order Logic

1. **Syntax:** A grammar specifying what are legal syntactic constructs of the representation.

2. **Semantics:** A formal mapping from syntactic constructs to set theoretic assertions.

First Order Syntax

Start with a set of primitive symbols.

1. *constant* symbols.
2. *function* symbols.
3. *predicate* symbols (for predicates and relations).
4. *variables.*

- Each function and predicate symbol has a specific arity (determines the number of arguments it takes).

First Order Syntax—Building up.

- *terms* are used as names (perhaps complex nested names) for objects in the domain.
- *Terms* of the language are either:
  - a variable
  - a constant
  - an expression of the form $f(t_1, \ldots t_k)$ where
    - (a) f is a function symbol;
    - (b) k is its arity;
    - (c) each t_i is a term

- 5 is a term—a symbol representing the number 5. John is a term—a symbol representing the person John.
- +(5,5) is a term—a symbol representing the number 10.

First Order Syntax—Building up.

- **Note:** constants are the same as functions taking zero arguments.
- Terms denote objects (things in the world):
  - constants denote specific objects;
  - functions map tuples of objects to other objects
    - bill, jane, father(jane), father(father(jane))
    - X, father(X), hotel7, rating(hotel7), cost(hotel7)
  - Variables like X are not yet determined, but they will eventually denote particular objects.

First Order Syntax—Building up.

- Once we have terms we can build up *formulas*. Terms represent (denote) objects, *formulas* represent true/false assertions about these objects.
- We start with *atomic formulas* these are
  - expressions of the form $p(t_1, \ldots t_k)$ where
  - (a) p is a predicate symbol;
  - (b) k is its arity;
  - (c) each t_i is a term

Semantic Intuition (formalized later).

- Atoms denote facts that can be true or false about the world
  - father_of(jane, bill), female(jane), system_down()
  - satisfied(client15), satisfied(C)
  - desires(client15,rome,week29), desires(X,Y,Z)
  - rating(hotel7, 4), cost(hotel7, 125)

First Order Syntax—Building up.

- Atomic formulas

- The negation (NOT) of a formula is a new formula
  - ¬f (-f)

  Asserts that f is false.

- The conjunction (AND) of a set of formulas is a formula.
  - $f_1 \land f_2 \land \ldots \land f_n$ where each f_i is formula

  Asserts that each formula fi is true.

First Order Syntax—Building up.

- The disjunction (OR) of a set of formulas is a formula.
  - $f_1 \lor f_2 \lor \ldots \lor f_n$ where each f_i is formula

  Asserts that at least one formula fi is true.

- Existential Quantification ∃.
  - ∃X. f where X is a variable and f is a formula.

  Asserts there is some object such that once X is bound to that object, f will be true.

- Universal Quantification ∀.
  - ∀X.f where X is a variable and f is a formula.

  Assets that f is true for every object X can be bound to.

First Order Syntax—abbreviations.

- Implication:
  - f1→ f2

  Take this to mean
  - ¬f1 ∨ f2.

Semantics.

- Formulas (syntax) can be built up recursively, and can become arbitrarily complex.
- Intuitively, there are various distinct formulas (viewed as strings) that really are asserting the same thing
  - ∀X,Y. elephant(X) ∧ teacup(Y) → largerThan(X,Y)
  - ∀X,Y. teacup(Y) ∧ elephant(X) → largerThan(X,Y)
- To capture this equivalence and to make sense of complex formulas we utilize the semantics.

Semantics.

- A formal mapping from formulas to semantic entities (individuals, sets and relations over individuals, functions over individuals).
- The mapping is mirrors the recursive structure of the syntax, so we can give any formula, no matter how complex a mapping to semantic entities.

Semantics—Formal Details

- First, we must fix the particular first-order language we are going to provide semantics for. The primitive symbols included in the syntax defines the particular language. L(F,P,V)
- *F = set of function (and constant symbols)*
  - *Each symbol f in F has a particular arity.*
- *P = set of predicate and relation symbols.*
  - *Each symbol p in P has a particular arity.*
- *V = an infinite set of variables.*

Semantics—Formal Details

- An interpretation (model) is a tuple
  $\langle D, \Phi, \Psi, v\rangle$
  - D is a non-empty set (domain of individuals)
  - Φ is a mapping: $\Phi(f) \to (D^k \to D)$
    - maps k-ary function symbol f, to a function from k-ary tuples of individuals to individuals.
  - Ψ is a mapping: $\Psi(p) \to (D^k \to$ True/False)
    - maps k-ary predicate symbol p, to an indicator function over k-ary tuples of individuals (a subset of D^k)
  - v is a variable assignment function. $v(X) = d \in D$ (it maps every variable to some individual)

Intuitions: Domain

- Domain D: $d \in D$ is an *individual*
- E.g., *{ craig, jane, grandhotel, le-fleabag, rome, portofino, 100, 110, 120 …}*
- Underlined symbols denote domain individuals (as opposed to symbols of the first-order language)
- Domains often infinite, but we'll use finite models to prime our intuitions

Intuitions: Φ

- $\Phi(f) \to (D^k \to D)$

Given *k*-ary function f, *k* individuals, what individual does *f(d1, …, dk)* denote

- 0-ary functions (constants) are mapped to specific individuals in D.
  - *Φ(client17) = craig, Φ(hotel5) = le-fleabag, Φ (rome) = rome*
- 1-ary functions are mapped to functions in $D \to D$
  - *Φ(minquality)=f_minquality:*
    f_minquality(craig) = 3stars
  - *Φ(rating)=f_rating:*
    f_rating(grandhotel) = 5stars
- 2-ary functions are mapped to functions from $D^2 \to D$
  - *Φ(distance)=f_distance:*
    f_distance(toronto, sienna) = 3256
- n-ary functions are mapped similarly.

Intuitions: Ψ

- $\Psi(p) \rightarrow (D^k \rightarrow \text{True/False})$
  - given *k*-ary predicate, *k* individuals, does the relation denoted by p hold of these? *Ψ(p)(<d1, … dk>) = true?*
- 0-ary predicates are mapped to true or false.
  Ψ(rainy) = True   Ψ(sunny) = False
- 1-ary predicates are mapped indicator functions of subsets of D.
  - *Ψ(satisfied) = p_satisfied:*
    p_satisfied(craig) = True
  - *Ψ(privatebeach) = p_privatebeach:*
    p_privatebeach(le-fleabag) = False
- 2-ary predicates are mapped to indicator functions over D^2
  - *Ψ(location) = p_location: p_location(grandhotel, rome) = True*
    p_location(grandhotel, sienna) = False
  - *Ψ(available) = p_available:*
    p_available(grandhotel, week29) = True
- n-ary predicates..

Intuitions: v

- v exists to take care of quantification. As we will see the exact mapping it specifies will not matter.
- Notation: v[X/d] is a new variable assignment function.
  - Exactly like v, except that it maps the variable X to the individual d.
  - Maps every other variable exactly like v:
    v(Y) = v[X/d](Y)

Semantics—Building up

Given language L(F,P,V), and an interpretation
$I = \langle D, \Phi, \Psi, v\rangle$

a) Constant c (0-ary function) denotes an individual
$I(c) = \Phi(c) \in D$

b) Variable X denotes an individual
$I(X) = v(X) \in D$ (variable assignment function).

c) Ground term t = $f(t_1,\dots, t_k)$ denotes an individual
$I(t) = \Phi(f)(I(t_1),\dots I(t_k)) \in D$

We recursively find the denotation of each term, then we apply the function denoted by f to get a new individual.

Hence terms always denote individuals under an interpretation I

Semantics—Building up

Formulas

a) Ground atom a = $p(t_1,\dots t_k)$ has truth value

$I(a) = \Psi(p)(I(t_1), \dots, I(t_k)) \in \{ \text{True}, \text{False} \}$

We recursively find the individuals denoted by the t_i, then we check to see if this tuple of individuals is in the relation denoted by p.

Semantics—Building up

Formulas

b) Negated formulas $\neg f$ has truth value

$I(\neg f)$ = True if $I(f)$ = False
$I(\neg f)$ = False if $I(f)$ = True

c) And formulas $f_1 \land f_2 \land \ldots \land f_n$ have truth value

$I(f_1 \land f_2 \land \ldots \land f_n)$ = True if every $I(f_i)$ = True.
$I(f_1 \land f_2 \land \ldots \land f_n)$ = False otherwise.

d) Or formulas $f_1 \lor f_2 \lor \ldots \lor f_n$ have truth value

$I(f_1 \lor f_2 \lor \ldots \lor f_n)$ = True if any $I(f_i)$ = True.
$I(f_1 \lor f_2 \lor \ldots \lor f_n)$ = False otherwise.

Semantics—Building up

Formulas

e) Existential formulas $\exists X.\ f$ have truth value

$I(\exists X.\ f)$ = True if there exists a $d \in D$ such that

$I'(f)$ = True

where $I' = \langle D, \Phi, \Psi, v[X/d] \rangle$

False otherwise.

I' is just like I except that its variable assignment function now maps X to d. "d" is the individual of which "f" is true.

Semantics—Building up

Formulas

f) Universal formulas $\forall X.f$ have truth value

$I(\forall X.f)$ = True if for all $d \in D$

$I'(f)$ = True

where $I' = \langle D, \Phi, \Psi, v[X/d] \rangle$

False otherwise.

Now "f" must be true of every individual "d".

Hence formulas are always either True or False under an interpretation I

Example

D = {bob, jack, fred}
happy is true of all objects.
I(∀X.happy(X))

1. Ψ(happy)(v[X/bob](X)) = Ψ(happy)(bob) = True

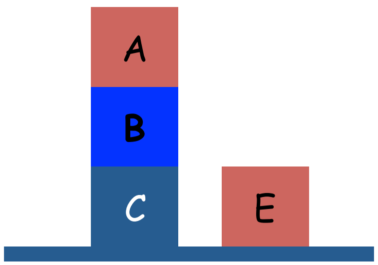
2. Ψ(happy)(v[X/jack](X)) = Ψ(happy)(jack) = True

3. Ψ(happy)(v[X/fred](X)) = Ψ(happy)(fred) = True

Therefore I(∀X.happy(X)) = True.

Models—Examples.

Environment



Language (Syntax)

- Constants: a,b,c,e
- Functions:
  - No function
- Predicates:
  - on: binary
  - above: binary
  - clear: unary
  - ontable: unary

Models—Examples.

Language (syntax)

- Constants: a,b,c,e
- Predicates:
  - on (binary)
  - above (binary)
  - clear (unary)
  - ontable(unary)

A possible Model I_1 (semantics)

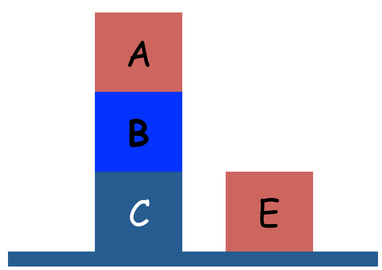
- D = {A, B, C, E}
- Φ(a) = A, Φ(b) = B, Φ(c) = C, Φ(e) = E.
- Ψ(on) = {(A,B),(B,C)}
- Ψ(above)= {(A,B),(B,C),(A,C)}
- Ψ(clear)={A,E}
- Ψ(ontable)={C,E}

Models—Examples.

Model I_1

- D = {A, B, C, E}
- Φ(a) = A, Φ(b) = B, Φ(c) = C, Φ(e) = E.
- Ψ(on) = {(A,B),(B,C)}
- Ψ(above) = {(A,B),(B,C),(A,C)}
- Ψ(clear)={A,E}
- Ψ(ontable)={C,E}

Environment



Models—Formulas true or false?

Model I_1

- D = {A, B, C, E}
- Φ(a) = A, Φ(b) = B, Φ(c) = C, Φ(e) = E.
- Ψ(on) = {(A,B),(B,C)}
- Ψ(above) = {(A,B),(B,C),(A,C)}
- Ψ(clear)={A,E}
- Ψ(ontable)={C,E}

$\forall$X,Y. on(X,Y)→above(X,Y)

X=A, Y=B

X=C, Y=A

…

$\forall$X,Y. above(X,Y)→on(X,Y)

X=A, Y=B

X=A, Y=C

Models—Examples.

Model I_1

- D = {A, B, C, E}
- Φ(a) = A, Φ(b) = B, Φ(c) = C, Φ(e) = E.
- Ψ(on) = {(A,B),(B,C)}
- Ψ(above) = {(A,B), (B,C),(A,C)}
- Ψ(clear)={A,E}
- Ψ(ontable)={C,E}

$\forall X \exists Y.\ (clear(X) \lor on(Y,X))$

X=A
X=C, Y=B
...

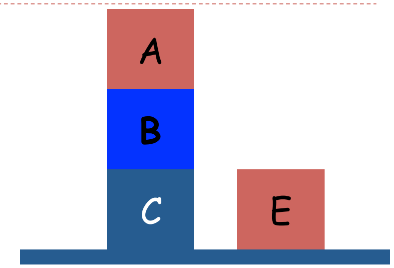
$\exists Y \forall X.(clear(X) \lor on(Y,X))$

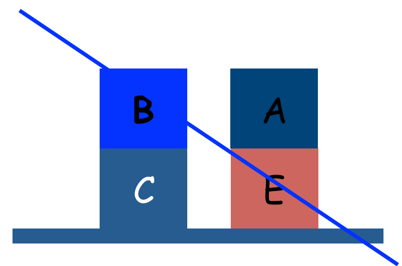
Y=A ? No! (X=C)
Y=C? No! (X=B)
Y=E? No! (X=B)
Y=B ? No! (X=B)

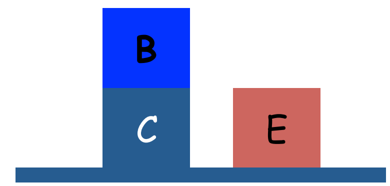
KB—many models

KB

1. on(b,c)
2. clear(e)







Models

- Let our Knowledge base KB, consist of a set of formulas.
- We say that I is a model of KB or that I satisfies KB
  - If, every formula $f \in$ KB is true under I
- We write I ⊨ KB if I satisfies KB, and I⊨*f* if *f* is true under I.

What's Special About Models?

- When we write KB, we intend that the real world (i.e. our set theoretic abstraction of it) is one of its models.
- This means that every statement in KB is true in the real world.
- Note however, that not every thing true in the real world need be contained in KB. We might have only incomplete knowledge.

Models support reasoning.

- Suppose formula f is not mentioned in KB, but is true in every model of KB; i.e.,
  $I \models KB \rightarrow I \models f.$
- Then we say that f is a logical consequence of KB or that KB entails f .
- Since the real world is a model of KB, f must be true in the real world.
- This means that entailment is a way of finding new true facts that were not explicitly mentioned in KB.

??? If KB doesn't entail f, is f false in the real world?

Logical Consequence Example

- elephant(clyde)
  - the individual denoted by the symbol *clyde* in the set deonted by *elephant* (has the property that it is an *elephant*).
- **teacup(cup)**
  - *cup* is a teacup.
- Note that in both cases a unary predicate specifies a set of individuals. Asserting a unary predicate to be true of a term means that the individual denoted by that term is in the specified set.
  - Formally, we map individuals to TRUE/FALSE (this is an indicator function for the set).

Logical Consequence Example

- $\forall X,Y.\text{elephant}(X) \land \text{teacup}(Y) \rightarrow \text{largerThan}(X,Y)$
  - For all pairs of individuals if the first is an elephant and the second is a teacup, then the pair of objects are related to each other by the *largerThan* relation.
  - For pairs of individuals who are not elephants and teacups, the formula is immediately true.

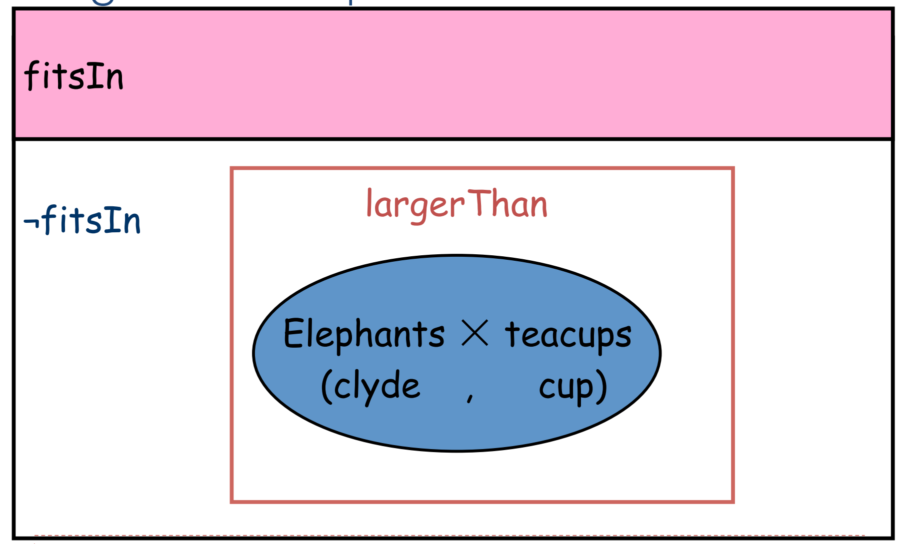
Logical Consequence Example

- $\forall X,Y.\text{largerThan}(X,Y) \rightarrow \neg\text{fitsIn}(X,Y)$
  - For all pairs of individuals if X is larger than Y (the pair is in the largerThan relation) then we cannot have that X fits in Y (the pair cannot be in the fitsIn relation).
  - (The relation largerThan has a empty intersection with the fitsIn relation).

Logical Consequences

- ¬fitsIn(clyde,cup)
- We know largerThan(clyde,teacup) from the first implication. Thus we know this from the second implication.

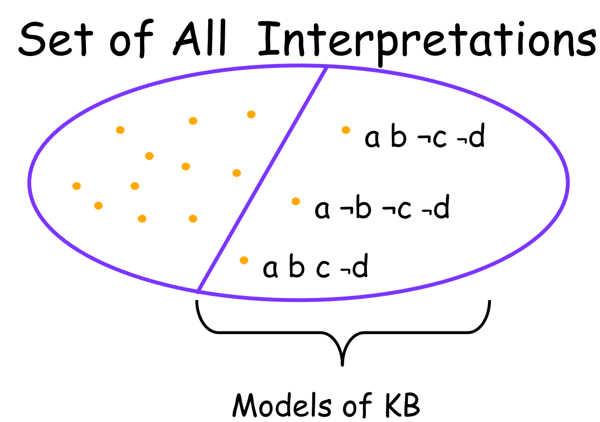
Logical Consequences



Logical Consequence Example

- If an interpretation satisfies KB, then the set of pairs *elephant* X *teacup* must be a subset of *largerThan*, which is disjoint from *fitsIn*.
- Therefore, the pair (*clyde,cup)* must be in the complement of the set *fitsIn*.
- Hence, ¬fitsIn(clyde,cup) must be true in every interpretation that satisfies KB.
- ¬fitsIn(clyde,cup) is a logical consequence of KB.

Models Graphically



Consequences? a, c → b, b → c, d → b, ¬b → ¬c

Models and Interpretations

- the more sentences in KB, the fewer models (satisfying interpretations) there are.
- The more you write down (as long as it's all true!), the "closer" you get to the "real world"! Because Each sentence in KB rules out certain unintended interpretations.
- This is called axiomatizing the domain

Computing logical consequences

- We want procedures for computing logical consequences that can be implemented in our programs.
- This would allow us to reason with our knowledge
  - Represent the knowledge as logical formulas
  - Apply procedures for generating logical consequences
- These procedures are called proof procedures.

Proof Procedures

- Interesting, proof procedures work by simply manipulating formulas. They do not know or care anything about interpretations.
- Nevertheless they respect the semantics of interpretations!
- We will develop a proof procedure for first-order logic called resolution.
  - Resolution is the mechanism used by PROLOG

Properties of Proof Procedures

- Before presenting the details of resolution, we want to look at properties we would like to have in a (any) proof procedure.
- We write $KB \vdash f$ to indicate that f can be proved from KB (the proof procedure used is implicit).

Properties of Proof Procedures

- Soundness
  - KB ⊢ f → KB ⊨ f

    i.e all conclusions arrived at via the proof procedure are correct: they are logical consequences.

- Completeness
  - KB ⊨ f → KB ⊢ f

    i.e. every logical consequence can be generated by the proof procedure.

- Note proof procedures are computable, but they might have very high complexity in the worst case. So completeness is not necessarily achievable in practice.

Resolution

- Clausal form.
  - Resolution works with formulas expressed in clausal form.
  - A literal is an atomic formula or the negation of an atomic formula. dog(fido), ¬cat(fido)
  - A clause is a disjunction of literals:
    - ¬owns(fido,fred) ∨ ¬dog(fido) ∨ person(fred)
    - We write

      (¬owns(fido,fred), ¬dog(fido), person(fred))
  - A clausal theory is a conjunction of clauses.

Resolution

- Prolog Programs
  - Prolog programs are clausal theories.
  - However, each clause in a Prolog program is Horn.
  - A horn clause contains at most one positive literal.
    - The horn clause

      ¬q1 v ¬q2 v … v ¬qn v p

      is equivalent to

      q1 ∧ q2 ∧… ∧ qn ⇒ p

      and is written as the following rule in Prolog:

      p :- q1 , q2 ,… ,qn

Resolution Rule for Ground Clauses

- The resolution proof procedure consists of only one simple rule:
  - From the two clauses
    - (P, Q1, Q2, …, Qk)
    - (¬P, R1, R2, …, Rn)
  - We infer the new clause
    - (Q1, Q2, …, Qk, R1, R2, …, Rn)
  - Example:
    - (¬largerThan(clyde,cup), ¬fitsIn(clyde,cup)
    - (fitsIn(clyde,cup))

      ⇒ ¬largerThan(clyde,cup)

Resolution Proof: Forward chaining

- Logical consequences can be generated from the resolution rule in two ways:
  1. Forward Chaining inference.
     - If we have a sequence of clauses C1, C2, …, Ck
     - Such that each Ci is either in KB or is the result of a resolution step involving two prior clauses in the sequence.
     - We then have that KB ⊢ Ck.

     Forward chaining is sound so we also have KB ⊨ Ck

Resolution Proof: Refutation proofs

2. Refutation proofs.
   - We determine if KB ⊢ f by showing that a contradiction can be generated from KB Λ ¬f.
   - In this case a contradiction is an empty clause ().
   - We employ resolution to construct a sequence of clauses C1, C2, …, Cm such that
     - Ci is in KB Λ ¬f, or is the result of resolving two previous clauses in the sequence.
     - Cm = () i.e. its the empty clause.

Resolution Proof: Refutation proofs

- If we can find such a sequence C1, C2, …, Cm=(), we have that
  - KB ⊢ f.
  - Furthermore, this procedure is sound so
    - KB ⊨ f
- And the procedure is also complete so it is capable of finding a proof of any f that is a logical consequence of KB. I.e.
  - If KB ⊨ f then we can generate a refutation from KB Λ ¬f

Resolution Proofs Example

Want to prove likes(clyde,peanuts) from:

1. (elephant(clyde), giraffe(clyde))
2. (¬elephant(clyde), likes(clyde,peanuts))
3. (¬giraffe(clyde), likes(clyde,leaves))
4. ¬likes(clyde,leaves)

Forward Chaining Proof:

- 3&4 → ¬giraffe(clyde) [5.]
- 5&1 → elephant(clyde) [6.]
- 6&2 → likes(clyde,peanuts) [7.] ✓

Resolution Proofs Example

1. (elephant(clyde), giraffe(clyde))
2. (¬elephant(clyde), likes(clyde,peanuts))
3. (¬giraffe(clyde), likes(clyde,leaves))
4. ¬likes(clyde,leaves)

Refutation Proof:

- ¬likes(clyde,peanuts) [5.]
- 5&2 → ¬elephant(clyde) [6.]
- 6&1 → giraffe(clyde) [7.]
- 7&3 → likes(clyde,leaves) [8.]
- 8&4 → () ✓

Resolution Proofs

- Proofs by refutation have the advantage that they are easier to find.
  - They are more focused to the particular conclusion we are trying to reach.
- To develop a complete resolution proof procedure for First-Order logic we need :
  1. A way of converting KB and f (the query) into clausal form.
  2. A way of doing resolution even when we have variables (unification).

Conversion to Clausal Form

To convert the KB into Clausal form we perform the following 8-step procedure:

1. **Eliminate Implications.**
2. **Move Negations inwards (and simplify ¬¬).**
3. **Standardize Variables.**
4. **Skolemize.**
5. **Convert to Prenix Form.**
6. **Distribute conjunctions over disjunctions.**
7. **Flatten nested conjunctions and disjunctions.**
8. **Convert to Clauses.**

C-T-C-F: Eliminate implications

We use this example to show each step:

$$\forall X.p(X) \rightarrow \Big((\forall Y.p(Y) \rightarrow p(f(X,Y))) \land \neg(\forall Y. \neg q(X,Y) \land p(Y))\Big)$$

1. Eliminate implications: $A \rightarrow B$ ➔ $\neg A \lor B$

$$\forall X. \neg p(X) \lor \Big((\forall Y.\neg p(Y) \lor p(f(X,Y))) \land \neg(\forall Y. \neg q(X,Y) \land p(Y)) \Big)$$

C-T-C-F: Move ¬ Inwards

$\forall X. \neg p(X)$
$\quad \vee \big(\ (\forall Y. \neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge \neg(\forall Y. \neg q(X,Y) \wedge p(Y)) \big)$

2. Move Negations Inwards (and simplify ¬¬)

$\forall X. \neg p(X)$
$\quad \vee \big(\ (\forall Y. \neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge (\exists Y. q(X,Y) \vee \neg p(Y)) \big)$

C-T-C-F: : ¬ continue…

Rules for moving negations inwards

- $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
- $\neg\forall X. f \rightarrow \exists X. \neg f$
- $\neg\exists X. f \rightarrow \forall X. \neg f$
- $\neg\neg A \rightarrow A$

C-T-C-F: Standardize Variables

$\forall X. \neg p(X)$
$\quad \vee \big(\ (\forall Y. \neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge (\exists Y. q(X,Y) \vee \neg p(Y)) \big)$

3. Standardize Variables (Rename variables so that each quantified variable is unique)

$\forall X. \neg p(X)$
$\quad \vee \big(\ (\forall Y. (\neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge (\exists Z. q(X,Z) \vee \neg p(Z)) \big)$

C-T-C-F: Skolemize

$\forall X. \neg p(X)$
$\quad \vee \big(\ (\forall Y. \neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge (\exists Z. q(X,Z) \vee \neg p(Z)) \big)$

4. Skolemize (Remove existential quantifiers by introducing new function symbols).

$\forall X. \neg p(X)$
$\quad \vee \big((\forall Y. \neg p(Y) \vee p(f(X,Y)))$
$\quad\quad \wedge (q(X,g(X)) \vee \neg p(g(X))) \big)$

C-T-C-F: Skolemization

Consider $\exists Y.$elephant(Y) $\wedge$ friendly(Y)

- This asserts that there is some individual (binding for Y) that is both an elephant and friendly.
- To remove the existential, we **invent** a name for this individual, say ***a*.** This is a new constant symbol **not equal to any previous constant symbols** to obtain:

  elephant**(a)** $\wedge$ friendly**(a)**

- This is saying the same thing, since we do not know anything about the new constant **a**.

C-T-C-F: Skolemization

- It is essential that the introduced symbol "a" is **new.** Else we might know something else about "a" in KB.
- If we did know something else about "a" we would be asserting more than the existential.
- In original quantified formula we know nothing about the variable "Y". Just what was being asserted by the existential formula.

C-T-C-F: Skolemization

Now consider $\forall X \exists Y.$ loves(X,Y).

- This formula claims that for every *X* there is some *Y* that X loves (perhaps a different Y for each X).
- Replacing the existential by a new constant won't work

  $\forall X.$loves(X,**a**).

  Because this asserts that there is a **particular** individual "a" loved by every X.

- To properly convert existential quantifiers scoped by universal quantifiers we must use **functions** not just constants.

C-T-C-F: Skolemization

- We must use a function that mentions every universally quantified variable <u>that scopes the existential</u>.
- In this case *X* scopes *Y* so we must replace the existential *Y* by a function of *X*

  $\forall X.$ loves(X,g(X)).

  where g is a **new** function symbol.

- This formula asserts that for every X there is some individual (given by g(X)) that X loves. g(X) can be different for each different binding of X.

C-T-C-F: Skolemization Examples

- $\forall XYZ\ \exists W.r(X,Y,Z,W) \rightarrow \forall XYZ.r(X,Y,Z,h1(X,Y,Z))$
- $\forall XY\exists W.r(X,Y,g(W)) \rightarrow \forall XY.r(X,Y,Z,g(h2(X,Y)))$
- $\forall XY\exists W\forall Z.r(X,Y,W) \land q(Z,W)$

  $\rightarrow \forall XYZ.r(X,Y,h3(X,Y)) \land q(Z,h3(X,Y))$

C-T-C-F: Convert to prenix

$$\forall X.\ \neg p(X) \vee \Big(\forall Y. \neg p(Y) \vee p(f(X,Y)) \land q(X,g(X)) \vee \neg p(g(X)) \Big)$$

5. Convert to prenix form. (Bring all quantifiers to the front—only universals, each with different name).

$$\forall X \forall Y.\ \neg p(X) \vee \Big(\neg p(Y) \vee p(f(X,Y)) \land q(X,g(X)) \vee \neg p(g(X)) \Big)$$

C-T-C-F: Conjunctions over disjunctions

$$\forall X \forall Y.\ \neg p(X) \vee \Big((\neg p(Y) \vee p(f(X,Y))) \land (q(X,g(X)) \vee \neg p(g(X))) \Big)$$

6. Conjunctions over disjunctions

   $A \vee (B \land C) \rightarrow (A \vee B) \land (A \vee C)$

$$\forall XY.\ (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \land (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X)))$$

C-T-C-F: flatten & convert to clauses

7. **Flatten nested conjunctions and disjunctions.**

   $(A \vee (B \vee C)) \rightarrow (A \vee B \vee C)$

8. **Convert to Clauses** (remove quantifiers and break apart conjunctions).

$$\forall XY.\ (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))) \land (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X)))$$

a) $\neg p(X) \vee \neg p(Y) \vee p(f(X,Y))$

b) $\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X))$

Unification

- Ground clauses are clauses with no variables in them. For ground clauses we can use syntactic identity to detect when we have a P and ¬P pair.

- What about variables can the clauses
  - (P(john), Q(fred), R(X))
  - (¬P(Y), R(susan), R(Y))

  Be resolved?

Unification.

- Intuitively, once reduced to clausal form, all remaining variables are universally quantified. So, implicitly (¬P(Y), R(susan), R(Y)) represents a whole set of ground clauses like
  - (¬P(fred), R(susan), R(fred))
  - (¬P(john), R(susan), R(john))
  - …

- So there is a "specialization" of this clause that can be resolved with (P(john), Q(fred), R(X))

Unification.

- We want to be able to match conflicting literals, even when they have variables. This matching process automatically determines whether or not there is a "specialization" that matches.

- We don't want to over specialize!

Unification.

- (¬p(X), s(X), q(fred))
- (p(Y), r(Y))
- Possible resolvants
  - (s(john), q(fred), r(john)) {Y=X, X=john}
  - (s(sally), q(fred), r(sally)) {Y=X, X=sally}
  - (s(X), q(fred), r(X)) {Y=X}

- The last resolvant is "most-general", the other two are specializations of it.
- We want to keep the most general clause so that we can use it future resolution steps.

Unification.

- unification is a mechanism for finding a "most general" matching.
- First we consider substitutions.
  - A substitution is a finite set of equations of the form

    $V = t$

    where V is a variable and t is a term not containing V. (t might contain other variables).

Substitutions.

- We can apply a substitution σ to a formula f to obtain a new formula $f\sigma$ by simultaneously replacing every variable mentioned in the left hand side of the substitution by the right hand side.

$$p(X,g(Y,Z))[X=Y,\ Y=f(a)] \Rightarrow p(Y,g(f(a),Z))$$

- Note that the substitutions are not applied sequentially, i.e., the first Y is not subsequently replaced by f(a).

Substitutions.

- We can compose two substitutions. θ and σ to obtain a new substition $\theta\sigma$.

Let $\theta = \{X_1=s_1, X_2=s_2, \ldots, X_m=s_m\}$

$\sigma = \{Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$

To compute $\theta\sigma$

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \ldots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$

   we apply σ to each RHS of θ and then add all of the equations of σ.

Substitutions.

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \ldots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$
2. Delete any identities, i.e., equations of the form V=V.
3. Delete any equation $Y_i=s_i$ where Y_i is equal to one of the X_j in θ.

The final set S is the composition $\theta\sigma$.

Composition Example.

θ = {X=f(Y), Y=Z}, σ = {X=a, Y=b, Z=Y}

$\theta\sigma$

Substitutions.

- The empty substitution ε = {} is also a substitution, and it acts as an identity under composition.
- More importantly substitutions when applied to formulas are associative:

$$(f\theta)\sigma = f(\theta\sigma)$$

- Composition is simply a way of converting the sequential application of a series of substitutions to a single simultaneous substitution.

Unifiers.

- A unifier of two formulas f and g is a substitution σ that makes f and g syntactically identical.
- Not all formulas can be unified—substitutions only affect variables.

  p(f(X),a) p(Y,f(w))

- This pair cannot be unified as there is no way of making a = f(w) with a substitution.
- Note we typically use UPPER CASE to denote variables, lower case for constants.

MGU.

- A substitution σ of two formulas f and g is a Most General Unifier (MGU) if

1. σ is a unifier.
2. For every other unifier θ of f and g there must exist a third substitution λ such that

$$\theta = \sigma\lambda$$

- This says that every other unifier is "more specialized than σ. The MGU of a pair of formulas f and g is unique up to renaming.

MGU.

p(f(X),Z)   p(Y,a)

1. σ = {Y = f(a), X=a, Z=a} is a unifier.

   p(f(X),Z)σ =
   p(Y,a)σ =

   But it is not an MGU.

2. θ = {Y=f(X), Z=a} is an MGU.
   p(f(X),Z) θ =
   p(Y,a) θ =

MGU.

p(f(X),Z)   p(Y,a)

3. $\sigma = \theta\lambda$, where λ={X=a}

   σ = {Y = f(a), X=a, Z=a}
   λ ={X=a}
   $\theta\lambda$ =

MGU.

- The MGU is the "least specialized" way of making clauses with universal variables match.
- We can compute MGUs.
- Intuitively we line up the two formulas and find the first sub-expression where they disagree. The pair of subexpressions where they first disagree is called the disagreement set.
- The algorithm works by successively fixing disagreement sets until the two formulas become syntactically identical.

MGU.

To find the MGU of two formulas f and g.

1. $k = 0$; $\sigma_0 = \{\}$; $S_0 = \{f,g\}$
2. If S_k contains an identical pair of formulas stop, and return σ_k as the MGU of f and g.
3. Else find the disagreement set $D_k=\{e_1,e_2\}$ of S_k
4. If e_1 = V a variable, and e_2 = t a term not containing V (or vice-versa) then let

   $\sigma_{k+1} = \sigma_k \{V=t\}$ (Compose the addital substitution)

   $S_{k+1} = S_k\{V=t\}$ (Apply the additional substitution)

   k = k+1
   GOTO 2
5. Else stop, f and g cannot be unified.

MGU Example 1.

S_0 = {p(f(a), g(X)) ; p(Y,Y)}

MGU Example 2.

S_0 = {p(a,X,h(g(Z))) ; p(Z,h(Y),h(Y))}

MGU Example 3.

S_0 = {p(X,X) ; p(Y,f(Y))}

Non-Ground Resolution

- Resolution of non-ground clauses. From the two clauses
  (L, Q1, Q2, …, Qk)
  (¬M, R1, R2, …, Rn)

  Where there exists σ a MGU for L and M.

  We infer the new clause

  (Q1σ, …, Qkσ, R1σ, …, Rnσ)

Non-Ground Resolution E.G.

1. (p(X), q(g(X)))
2. (r(a), q(Z), ¬p(a))

L=p(X); M=p(a)
σ = {X=a}

3. R[1a,2c]{X=a} (q(g(a)), r(a), q(Z))

The notation is important.

- "R" means resolution step.
- "1a" means the first (a-th) literal in the first clause i.e. p(X).
- "2c" means the third (c-th) literal in the second clause, ¬p(a).
  - 1a and 2c are the "clashing" literals.
- {X=a} is the substitution applied to make the clashing literals identical.

Resolution Proof Example

"Some patients like all doctors. No patient likes any quack. Therefore no doctor is a quack."

Resolution Proof Step 1.
Pick symbols to represent these assertions.

p(X): X is a patient
d(x): X is a doctor
q(X): X is a quack
l(X,Y): X likes Y

Resolution Proof Example

Resolution Proof Step 2.
Convert each assertion to a first-order formula.

1. Some patients like all doctors.

F1.

Resolution Proof Example

2. No patient likes any quack

F2.

3. Therefore no doctor is a quack.

Query.

Resolution Proof Example

Resolution Proof Step 3.

Convert to Clausal form.

F1.

F2.

Negation of Query.

Resolution Proof Example

Resolution Proof Step 4.

Resolution Proof from the Clauses.

1. p(a)
2. (¬d(Y), l(a,Y))
3. (¬p(Z), ¬q(R), ¬l(Z,R))
4. d(b)
5. q(b)

Answer Extraction.

- The previous example shows how we can answer true-false questions. With a bit more effort we can also answer "fill-in-the-blanks" questions (e.g., what is wrong with the car?).
- As in Prolog we use free variables in the query where we want the fill in the blanks. We simply need to keep track of the binding that these variables received in proving the query.
  - parent(art, jon) –is art one of jon's parents?
  - parent(X, jon) -who is one of jon's parents?

Answer Extraction.

- A simple bookkeeping device is to use an predicate symbol answer(X,Y,…) to keep track of the bindings automatically.
- To answer the query parent(X,jon), we construct the clause
  (¬ parent(X,jon), answer(X))
- Now we perform resolution until we obtain a clause consisting of only answer literals (previously we stopped at empty clauses).

Answer Extraction: Example 1

1. father(art, jon)
2. father(bob,kim)
3. (¬father(Y,Z), parent(Y,Z))
   i.e. *all fathers are parents*
4. (¬ parent(X,jon), answer(X))
   i.e. the query is: who is parent of jon?

Here is a resolution proof:

5. R[4,3b]{Y=X,Z=jon}
   (¬father(X,jon), answer(X))
6. R[5,1]{X=art} answer(art)

so art is parent of jon

Answer Extraction: Example 2

1. (father(art, jon), father(bob,jon)) //either bob or art is parent of jon
2. father(bob,kim)
3. (¬father(Y,Z), parent(Y,Z)) //i.e. *all fathers are parents*
4. (¬ parent(X,jon), answer(X)) //i.e. query is parent(X,jon)

Here is a resolution proof:

5. R[4,3b]{Y=X,Z=jon} (¬father(X,jon), answer(X))
6. R[5,1a]{X=art} (father(bob,jon), answer(art))
7. R[6,3b] {Y=bob,Z=jon}
   (parent(bob,jon), answer(art))
8. R[7,4] {X=bob} (answer(bob), answer(art))

A disjunctive answer: either bob or art is parent of jon.

Factoring

1. (p(X), p(Y)) // ∀ X.∀ Y. ¬p(X) ➔ p(Y)
2. (¬p(V), ¬p(W)) // ∀ V.∀W. p(V) ➔ ¬p(W)

- These clauses are intuitively contradictory, but following the strict rules of resolution only we obtain:

3. R[1a,2a](X=V) (p(Y), ¬p(W))
   Renaming variables: (p(Q), ¬p(Z))
4. R[3b,1a](X=Z) (p(Y), p(Q))

No way of generating empty clause!

Factoring is needed to make resolution over non-ground clauses complete, without it resolution is incomplete!

Factoring.

- If two or more literals of a clause C have an mgu θ, then Cθ with all duplicate literals removed is called a **factor** of C.
- C = (p(X), p(f(Y)), ¬q(X))
  θ = {X=f(Y)}
  Cθ = (p(f(Y)), p(f(Y)), ¬q(f(Y))) ➔ (p(f(Y)), ¬q(f(Y)) is a factor

Adding a factor of a clause can be a step of proof:

1. (p(X), p(Y))
2. (¬p(V), ¬p(W))
3. f[1ab]{X=Y} p(Y)
4. f[2ab]{V=W} ¬p(W)
5. R[3,4]{Y=W} ().

Prolog

- Prolog search mechanism is simply an instance of resolution, except
  1. Clauses are Horn (only one positive literal)
  2. Prolog uses a specific depth first strategy when searching for a proof. (Rules are used first mentioned first used, literals are resolved away left to right).

Prolog

- Append:

1. append([], Z, Z)
2. append([E1 | R1], Y, [E1 | Rest]) :-
   append(R1, Y, Rest)

Note:

- The second is actually the clause
  (append([E1|R1], Y, [E1|Rest]) , ¬append(R1,Y,Rest))
- [ ] is a constant (the empty list)
- [X | Y] is cons(X,Y).
- So [a,b,c] is short hand for cons(a,cons(b,cons(c,[])))

Prolog: Example of proof

- Try to prove : append([a,b], [c,d], [a,b,c,d]):

1. append([], Z, Z)
2. (append([E1|R1], Y, [E1|Rest]),
   ¬append(R1,Y,Rest))
3. ¬append([a,b], [c,d], [a,b,c,d])

4. R[3,2a]{E1=a, R1=[b], Y=[c,d], Rest=[b,c,d]}
   ¬append([b], [c,d], [b,c,d])
5. R[4,2a]{E1=b, R1=[], Y=[c,d], Rest=[c,d]}
   ¬append([], [c,d], [c,d])
6. R[5,1]{Z=[c,d]} ()

Review: One Last Example!

Consider the following English description

- Whoever can read is literate.
- Dolphins are not literate.
- Flipper is an intelligent dolphin.

- Who is intelligent but cannot read.

Example: pick symbols & convert to first-order formula

- Whoever can read is literate.
  $\forall X.\ read(X) \rightarrow lit(X)$
- Dolphins are not literate.
  $\forall X.\ dolp(X) \rightarrow \neg lit(X)$
- Flipper is an intelligent dolphin
  $dolp(flipper) \wedge intell(flipper)$

- Who is intelligent but cannot read?
  $\exists X.\ intell(X) \wedge \neg read(X).$

Example: convert to clausal form

- $\forall X.\ read(X) \rightarrow lit(X)$
  $(\neg read(X), lit(X))$
- Dolphins are not literate.
  $\forall X.\ dolp(X) \rightarrow \neg lit(X)$
  $(\neg dolp(X), \neg lit(X))$
- Flipper is an intelligent dolphin.
  $dolp(flipper)$
  $intell(flipper)$
- who are intelligent but cannot read?
  $\exists X.\ intell(X) \wedge \neg read(X).$
  $\rightarrow \forall X.\ \neg intell(X) \vee read(X)$
  $\rightarrow (\neg intell(X), read(X), answer(X))$

Example: do the resolution proof

1. $(\neg read(X), lit(X))$
2. $(\neg dolp(X), \neg lit(X))$
3. $dolp(flip)$
4. $intell(flip)$
5. $(\neg intell(X), read(X), answer(X))$

6. R[5a,4] X=flip. $(read(flip), answer(flip))$
7. R[6,1a] X=flip. $(lit(flip), answer(flip))$
8. R[7,2b] X=flip. $(\neg dolp(flip), answer(flip))$
9. R[8,3] $answer(flip)$

so flip is intelligent but cannot read!