Knowledge Representation And Reasoning

- This material is covered in chapters 7—9 and 12 of the recommended 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 12 covers some of the additional notions that have to be dealt with when using knowledge representation in AI.

Knowledge Representation

- What is knowledge?
- Information we have about the world we inhabit
  - Both the physical and mental world.
  - We have knowledge about many abstract mental constructs and ideas.
- Besides knowledge we have various other mental attitudes and feelings about our environment.
  - John knows "…"
  - John fears "…"
  - But things get complex: John knows that he fears "…"
  - So knowledge can take a variety of forms, some quite complex.

Knowledge Representation

- What is Representation?
- Symbols standing for things in the world

CSC384

Words we use
in language

Symbols we use
in mathematics

Knowledge Representation

- Not all of our knowledge is symbolically represented.
- E.g., we do not symbolically represent the "pixels" that we perceive at the back of our retina.
- So intelligent agents also perform a great deal of low level "non-symbolic reasoning" over their perceptual inputs.
- But higher level "symbolically represented" knowledge also seems to be essential
  - This is the kind of knowledge that we learn by in school, by reading, etc.
- In this module we study symbolically represented knowledge.

Reasoning

- What is Reasoning (in the context of symbolically represented knowledge)
  - Manipulating our symbols to produce new symbols that represents new knowledge.
    Deriving a new sentence
  - Typically “symbols” are sequences of symbols, e.g., words in language sequenced together to form sentences.
  - So we will develop methods for manipulating ”sentences” to produce new “sentences”

Reasoning

- In language we can make up a huge variety of sentences.
- Each of these sentences makes some sort of claim or assertion about our world (mental or physical world).
- These claims could be true or false.
  - I am depressed, so the sentence "I feel happy." is false.
- Reasoning aims to be **TRUTH PERSERVING**.
- If we use reasoning to manipulate a collection of **TRUE** sentences, we want the newly derived sentences to also be **TRUE**.

If our reasoning is truth preserving we say that it is **SOUND**

Reasoning

- A more subtle idea is **COMPLETENESS**.
- Completeness says that our reasoning system is powerful enough to produce **ALL** sentences that must be true given our current collection of true sentences.
- Completeness requires a formal characterization of "sentence" in order to answer the question of if we have produced **ALL** true sentences.

Example Story Understanding.

- Three little pigs

Example.

- Three little pigs

Example.

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

- We are applying a lot of knowledge 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, Elephants can't fit into cars, instead of just knowing that larger objects can't fit into smaller objects.

Logical Representations

- AI typically employs logical representations of knowledge.
- Logical representations were originally developed to characterize laws of thought (proper ways of thinking).
- Have many features in common with natural language (although natural language is much more complex)

Logical Representations

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

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

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

4. Typically, have well developed proof theories: formal procedures for reasoning to produce new sentences.

In this module we will study First-Order logic, and a reasoning mechanism called resolution that operates on First-Order logic.

First-Order Logic (FOL)

- Two components: Syntax and Semantics.
  - In a programming language we have a syntax for an if statement: “if <boolean condition>: <expressions>”
  - The if statement also has a semantics: if <boolean condition> evaluates to TRUE then we execute <expressions>.
- Syntax gives the grammar or rules for forming proper sentences.
- Semantics gives the meaning of the sentences.

- The semantics are compositional—the meaning of complex sentences is determined by the meaning of their parts.
  - In the “if” statement, once we know what <boolean condition> is testing and what <expressions> are doing, we know what the “if” statement will do.

Basic Semantic entities of FOL

- We have a set of objects **D**. These are objects in the world that are important for our application. This set is often called the **domain of discourse.**
  - Often we will want to form tuples of objects, e.g., (d1, d2) with d1 $\in$ **D** and d2 $\in$ **D** is a pair of objects
  - A k-ary tuple is a subset of $\mathbf{D}^k$ = **D** X **D** X … **D** the k-wise Cartesian product of **D**.

- We can identify special sets of objects (subsets of **D**) that have some property in common. These sets are called **properties**.
  - E.g., **female**, **male**, **children**, **adult** could each be subsets that we identify as being useful in our application. If an object **d** is in the set **male**, we can say that d has the property male: **male(d)**.

Basic Semantic entities of FOL

- Sometimes individual objects are not sufficient, we want to identify special groups (tuples) of objects that are related to each other. We call these sets of tuples **relations**
  - E.g., **married** might be a special subset of pairs that we wish to keep track of in our application.
- Finally, we might want to keep track of functions over our objects. f: **D** → **D**
  - E.g., for d ∈ student, we might want a function faculty(d) that gives the faculty the student is registered in.
  - More generally we might want f : $\mathbf{D}^k$ → **D**, i.e., a function of many arguments mapping **D**.

Basic Syntactic symbols of FOL

- The syntax starts off with a different symbol for each the basic semantic entities (objects, functions, predicates, relations) that we have decided to utilize.
- We get to decide what symbols we use (but of course we want to use symbols that are easy to understand)
- These user specified symbols are called the **primitive symbols**

Syntax	**Semantics**
Constant symbols	A particular object d $\in$ **D**
Function symbols	Some function f: $\mathbf{D}^k \rightarrow \mathbf{D}$
Predicate symbols	Some subset of **D**
Relation symbols	Some subset of $\mathbf{D}^k$

Basic Syntactic symbols of FOL

- In addition we introduce some additional symbols that we will use to connect our basic symbols into sentences.

Syntax	**Semantics**
Constant symbols	A particular object d $\in$ **D**
Function symbols	Some function f: $\mathbf{D}^k \rightarrow \mathbf{D}$
Predicate symbols	Some subset of **D**
Relation symbols	Some subset of $\mathbf{D}^k$
Equality (commonly used relation)	Subset of $\mathbf{D}^2 = \{(d,d) \mid d \in \mathbf{D}\}$
Variables (as many as we need) x, y, z, …	A object d $\in$ **D** (which particular object can vary)
Logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$	…defined below…
Quantifiers $\forall$, $\exists$	…defined below…

Example

- Teaching CSC384, want to represent knowledge that would be useful for making the course a successful learning experience. So we might choose syntactic symbols like

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

First Order Syntax (the grammar)

We start with our basic syntactic symbols *constants, functions, predicates, relations,* and *variables.*

- *Note:* the function and relation symbols each have specific arities (determines the number of arguments it takes).

- From these we can build up **terms** and **sentences (formulas)**. Terms are ways of applying functions to build up new "names" for objects. Formulas, are ways of making true/false assertions.

First Order Syntax—Terms.

- *terms* are used as names (perhaps complex nested names) for objects in the domain.

Terms	
Constants	**c, john, mary**
variable	**x, y, z, …**
Function application	**$f(t_1, t_2, \ldots, t_k)$** t_i are already constructed terms

- 5 is a constant term: a symbol representing the number 5. John is a term—a symbol representing the person John.
- +(5,5) is a function application term—a new symbol representing the number 10.

First Order Syntax—Terms

- **Note:** constants are the same as functions taking zero arguments.
- Terms are names for 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—Sentences.

- Once we have terms we can build up *sentences* (*formulas*) Terms represent objects, *formulas* represent true/false assertions about these objects.

First Order Syntax—Sentences.

Formula	
Atomic Formula	**$p(t)$** or **$r(t_1, t_2, \ldots, t_k)$** p is a predicate symbol, r is a k-ary relation symbol t_i are terms
Negation	**$\neg f$** f is a formula
Conjunction	**$f_1 \wedge f_2 \wedge \ldots \wedge f_k$** f_i are formulas
Disjunction	**$f_1 \vee f_2 \vee \ldots \vee f_k$**
Implication	**$f_1 \rightarrow f_2$** f_1 and f_2 are formulas f_1 often called the antecedent, f_2 the consequence
Existential	**$\exists X.f$** f is a formula X is a variable
Universal	**$\forall X.f$** f is a formula X is a variable

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)
- Other formulas generate more complex assertions by composing these atomic formulas.
  - Their truth is dependent on the truth of the atomic formulas in them.

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
  - $\forall X,Y.\ elephant(X) \wedge teacup(Y) \rightarrow largerThan(X,Y)$
  - $\forall X,Y.\ teacup(Y) \wedge elephant(X) \rightarrow largerThan(X,Y)$

- To capture this equivalence and to make sense of complex formulas we utilize the semantics.

Semantics.

- A formal mapping from formulas to true/false assertions about our semantic entities (individuals, sets and relations over individuals, functions over individuals).

- The mapping mirrors the recursive structure of the syntax, so we can map any formula to a composition of assertions about the semantic entities.

Semantics—The language

- 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 relation symbol r ∈ P has a particular arity. (The predicate symbols always have arity 1)

V = an infinite set of variables.

Semantics—Primitive Symbols

- An interpretation (model) specifies the mapping from the primitive symbols to semantic entities. It is a tuple

  $\langle \mathbf{D}, \Phi, \Psi, \mathrm{V} \rangle$

  - **D** is a non-empty set of objects (domain of discourse)
  - Φ specifies the meaning of each primitive function symbol
    - also handles the primitive constant symbols (these can be viewed as being zero-arity functions.
  - Ψ specifies the meaning of each primitive predicate and relation symbol.
  - V specifies the meaning of the variables.

- Note, the semantics entity that a syntactic symbol maps to is often called the **meaning** of the symbol or the **denotation** of the symbol

Semantics—Primitive Symbols

Symbol	**Semantics**
constant Symbol **c**	Φ(**c**) $\in$ **D** (some particular object)
k-ary function symbol **f**	Φ(**f**) Some particular function $\mathbf{D}^k \rightarrow \mathbf{D}$
predicate symbol **p**	Ψ(**p**) Some particular subset of **D**
k-ary relation symbol **r**	Ψ(**r**) Some particular subset of $\mathbf{D}^k$
variable **x**	V(**x**) $\in$ **D** (some particular object)

Intuitions: Domain

- Domain D: $d \in D$ is an *individual*

- E.g., { *craig*, *jane*, *grandhotel*, *le-fleabag*,
  rome, *portofino*, *100*, *110*, *120* …}

- We use underlined symbols to talk about domain individuals (to syntactic symbols of the first-order language are not underlined)

- Domains often infinite, but we'll use finite models to prime our intuitions

Intuitions: Φ

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

- Constants (0-ary functions) are mapped to individuals in **D**.
  - Φ(client17) = *craig*, Φ(hotel5) = *le-fleabag*, Φ(rome) = *rome*
- 1-ary functions are mapped to particular functions in **D** → **D**
  - Φ(rating) = f_rating:
    f_rating(grandhotel) = *5stars*
- 2-ary functions are mapped to functions from $\mathbf{D}^2$ → **D**
  - Φ(distance) = f_distance:
    f_distance(toronto, sienna) = *3256*
- n-ary functions are mapped similarly.

Intuitions: Ψ

Given *k*-ary relation r, what does r denote

- 0-ary predicates are mapped to true or false.
  Ψ(rainy) = *True* Ψ(sunny) = *False*
- 1-ary predicates are mapped to subsets of **D**.
  - Ψ(privatebeach) = p_privatebeach: (the subset of hotels that have a private beach)
    e.g. p_privatebeack = {*grandhotel*, *fourseasons*}
    Note: *le-fleabag* is not a member of this set!
- 2-ary predicates are mapped to subsets of $\mathbf{D^2}$ (sets of pairs of individuals)
  - Ψ(available) = p_available: (set of pairs (hotel, week) that have free rooms)
    e.g., p_available = {(*grandhotel*, *week29*) , (*grandhotel*, *week30*)}
    Note: *fourseasons* has no available rooms!
- n-ary predicates..subsets of Dn

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, but maps the variable X to the individual d.
  - So for Y ≠ X:
    V[X/d](Y) = V(Y)
  - For X:
    V[X/d](X) = d

Semantics—Terms

Given language L(F,P,V), and an interpretation **I** = $\langle$**D**, Φ, Ψ, V$\rangle$ and any term **t**. **I(t)** is the denotation of **t** under **I**

Term	**Semantics**
constant Symbol **c**	I(**c**) = Φ(**c**) $\in$ **D** (some particular object)
variable **x**	I(**x**) = V(**x**) $\in$ **D** (some particular object)
Function application **f(t_1, t_2, …, t_k)**	I(**f(t_1, t_2, …, t_k)**) = Φ(**f**)(I(t_1), I(t_2), …, I(t_k)) First we obtain the denotation of each argument under I, then we apply the function Φ(**f**) to these interpreted terms

Hence terms always denote individuals under an interpretation I

Semantics—Formulas

Formulas will always be True or False under any interpretation **I**

formula	Semantics
Atomic formula $\mathbf{r}(t_1, t_2, \ldots, t_k)$	$I(\mathbf{r}(t_1, t_2, \ldots, t_k))$ = True if $(I(t_1), I(t_2), \ldots, I(t_k)) \in \Psi(\mathbf{r})$ false otherwise first we obtain the denotation of each argument under I. Then we check if this tuple of interpreted terms is in the set of tuples $\Psi(\mathbf{r})$

Ψ maps **r** to a subset of D^k (a subset of k-ary tuples of individuals). So the atomic formula is true if its arguments are in the stated relation.

Semantics—Formulas

Formula	Semantics
$\neg f$	$I(\neg f)$ = True if $I(f)$ = False False otherwise
$f_1 \wedge f_2 \wedge \ldots \wedge f_k$	$I(f_1 \wedge f_2 \wedge \ldots \wedge f_k)$ = True if $I(f_i)$ = True for every i False otherwise
$f_1 \vee f_2 \vee \ldots \vee f_k$	$I(f_1 \vee f_2 \vee \ldots \vee f_k)$ = True if $I(f_i)$ = True for any i False otherwise
$f_1 \rightarrow f_2$	$I(f_1 \rightarrow f_2)$ = True if $I(f_1)$ = False or $I(f_2)$ = True False otherwise

Standard rules for propositional logic that you would have seen before (check chap 7 if not)

Semantics—Formulas

Formula	Semantics
$\exists$**X.f**	I(**f**) = True if for some d $\in$ **D,** I'(**f**) = True I' = ⟨**D**, Φ, Ψ, V[X/d]⟩ False otherwise
$\forall$**X.f**	I(**f**) = True if for all d $\in$ **D,** I'(**f**) = True I' = ⟨**D**, Φ, Ψ, V[X/d]⟩ False otherwise

Quantifiers. Exists checks if f is true under some different variable mapping for the variable X. Forall checks if f is true under all possible mappings for the variable X.

Example

D = {bob, jack, fred}
I(happy) = {bob, jack, fred}
I(∀X.happy(X))

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

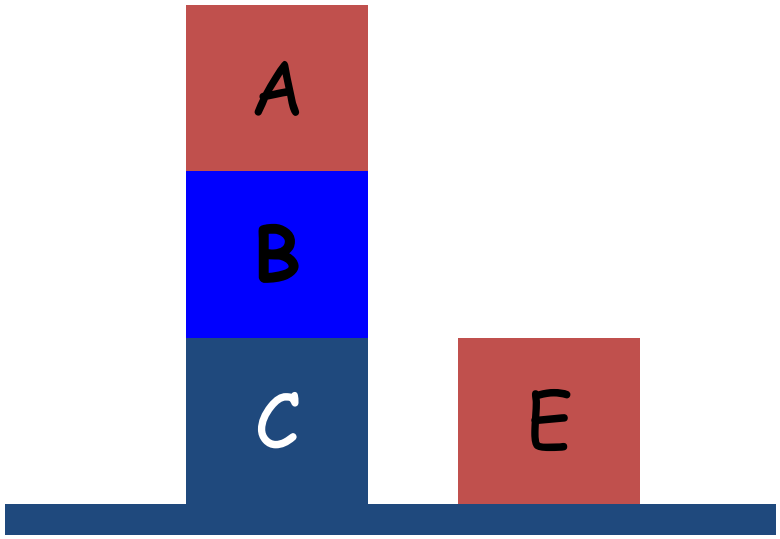
2. Ψ(happy)(v[X/jack](X)) = Ψ(happy)(jack) = True

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)

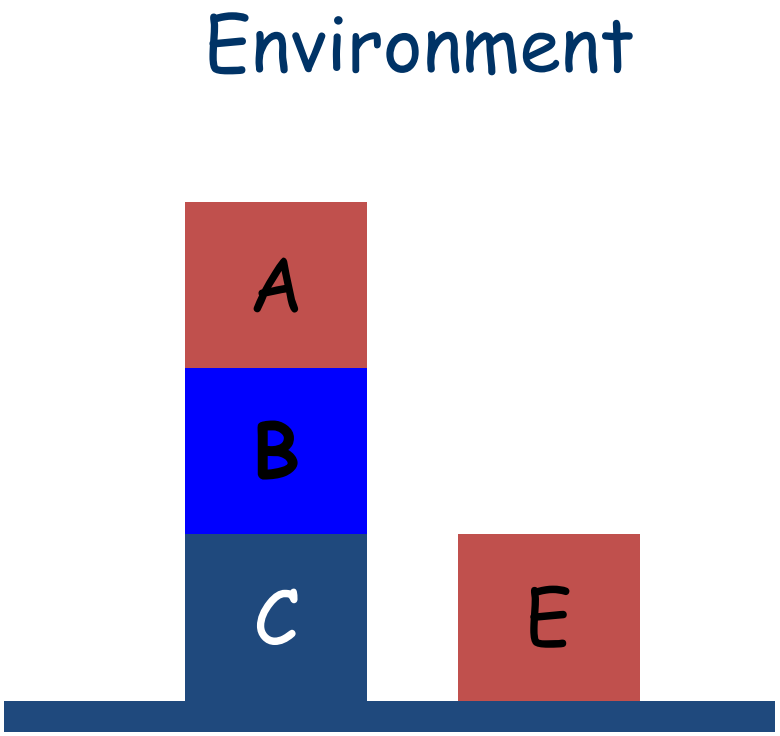
A possible Model I_1 (semantics)

- 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 Matches our environment

- 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—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}

∀X,Y. on(X,Y)→above(X,Y)

X=A, Y=B true

X=C, Y=A true (why?)

…

True!

∀X,Y. above(X,Y)→on(X,Y)

X=A, Y=B true

X=A, Y=A true

X=A, Y=C false

False!

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}

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

X=A true

X=C, Y=B true

…

True!

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

Y=A ? False! (X=C)

Y=C? False! (X=B)

Y=E? False! (X=B)

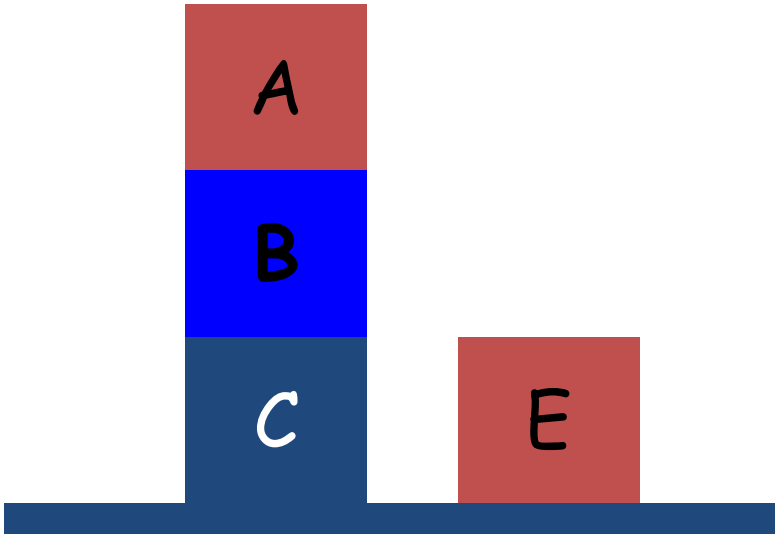
Y=B ? False! (X=B)

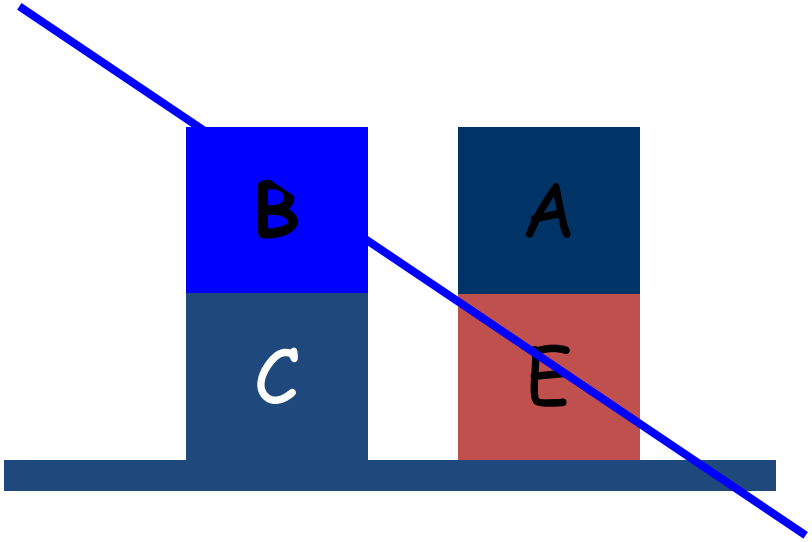
False!

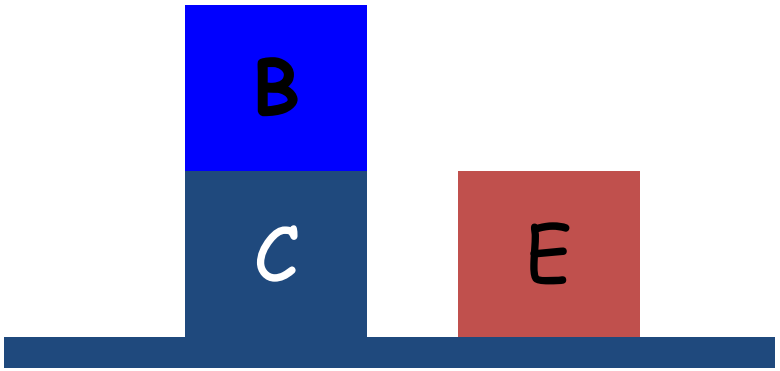
KB—many models

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 $\models$ KB if I satisfies KB, and I $\models f$ 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.

Logical Consequence Example

- $\forall X,Y.elephant(X) \wedge teacup(Y) \rightarrow largerThan(X,Y)$
  - 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

- $\forall X,Y.largerThan(X,Y) \rightarrow \neg fitsIn(X,Y)$

  - 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

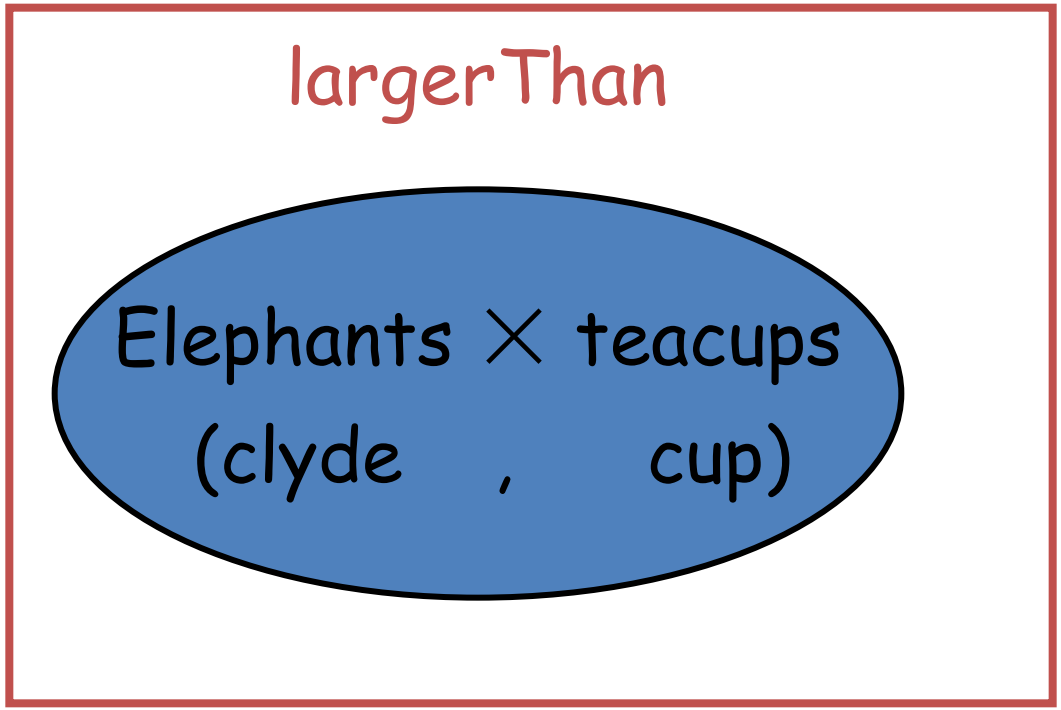
- ¬fitsIn(clyde,cup)

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

Logical Consequences

fitsIn

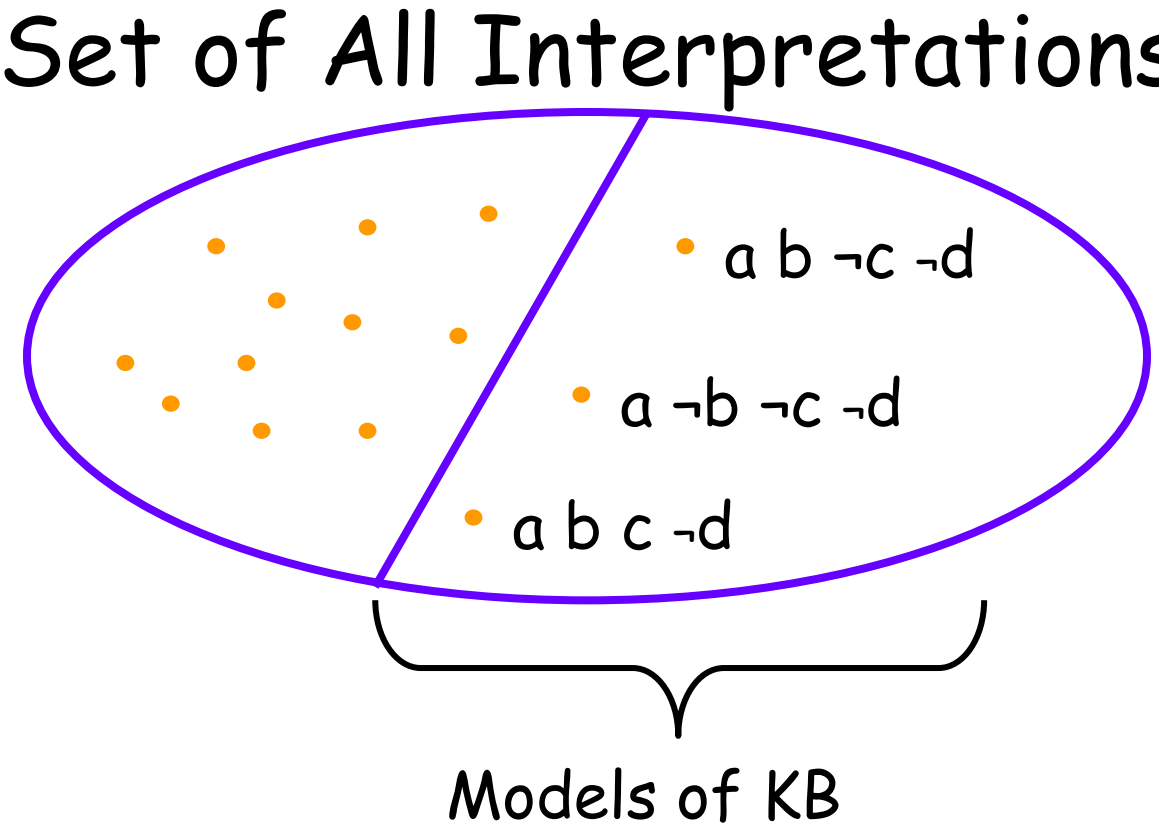
¬fitsIn



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



a, b, c, and d are atomic formulas

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

true, true, false, true, true

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.

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
  - $KB \vdash f \rightarrow KB \models f$

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

- Completeness
  - $KB \models f \rightarrow KB \vdash f$

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

- Note proof procedures for FOL have 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) $\vee$ ¬dog(fido) $\vee$ person(fred)
    - We write
      (¬owns(fido,fred), ¬dog(fido), person(fred))
  - A clausal theory is a conjunction of clauses.

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))

      $\Rightarrow$ ¬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 $\vdash$ Ck.

  Forward chaining is sound so we also have KB $\models$ Ck

Resolution Proof: Refutation proofs

2. Refutation proofs.
   - We determine if KB $\vdash$ f by showing that a contradiction can be generated from KB $\wedge \neg$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 $\wedge \neg$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 $\vdash$ f.
  - Furthermore, this procedure is sound so
    - KB $\models$ 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 $\models$ f then we can generate a refutation from KB $\wedge \neg$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:

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

1. Eliminate implications: $A \rightarrow B \Rightarrow \neg A \vee B$

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

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

$\forall X.\ \neg p(X)$

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

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

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

$\forall X.\ \neg p(X)$

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

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

C-T-C-F: Skolemize

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

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

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

C-T-C-F: Skolemization

Consider $\exists Y.\text{elephant}(Y) \wedge \text{friendly}(Y)$

- 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:

  $\text{elephant}(\mathbf{a}) \wedge \text{friendly}(\mathbf{a})$

- 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

  $\forall X.loves(X,\mathbf{a})$.

  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 that scopes the existential.

- 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

- $\forall XYZ\ \exists W.r(X,Y,Z,W)$ ➔ $\forall XYZ.r(X,Y,Z,h1(X,Y,Z))$

- $\forall XY \exists W.r(X,Y,g(W))$ ➔ $\forall XY.r(X,Y,Z,g(h2(X,Y)))$

- $\forall XY \exists W \forall Z.r(X,Y,W) \wedge q(Z,W)$

  ➔ $\forall XYZ.r(X,Y,h3(X,Y)) \wedge q(Z,h3(X,Y))$

C-T-C-F: Convert to prenix

$\forall X.\ \neg p(X)$

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

$\quad\quad \wedge\ q(X,g(X)) \vee \neg p(g(X)) \big)$

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

$\forall X \forall Y.\ \neg p(X)$

$\quad \vee \big(\neg p(Y) \vee p(f(X,Y))$

$\quad\quad \wedge\ q(X,g(X)) \vee \neg p(g(X)) \big)$

C-T-C-F: Conjunctions over disjunctions

$\forall X \forall Y.\ \neg p(X)$

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

$\quad\quad \wedge (q(X,g(X)) \vee \neg p(g(X)))\ )$

6. Conjunctions over disjunctions

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

$\forall XY.\quad (\neg p(X) \vee \neg p(Y) \vee p(f(X,Y)))$

$\quad \wedge (\neg p(X) \vee q(X,g(X)) \vee \neg p(g(X)))$

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).

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

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 $\theta\sigma$

Substitutions.

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

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

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.

$\theta = \{X=f(Y), Y=Z\}$, $\sigma = \{X=a, Y=b, Z=Y\}$

$\theta\sigma$

1. S = θ with σ applied to RH sides followed by σ
   = {X=f(Y){X=a,Y=b, Z=Y}, Y=Z{X=a,Y=b, Z=Y},
   X=a, Y=b, Z=Y}
   = {X=f(b), Y=Y, X=a, Y=b, Z=Y}
2. {X=f(b), ~~Y=Y,~~ X=a, Y=b, Z=Y} #delete identities
3. {X=f(b),~~X=a, Y=b,~~ Z=Y} #X and Y were in θ

$\theta\sigma$ = {X=f(b), Z=Y}

Substitutions.

- The empty substitution $\varepsilon = \{\}$ 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 (has some other substitution pairs added in). 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(f(a), a)
   p(Y,a)σ = p(f(a), a)

   But it is not an MGU.

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

MGU.

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

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

σ = {Y = f(a), X=a, Z=a}

θ = {Y=f(X), Z=a}

λ ={X=a}

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

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 additional 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.

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

$k = 0, \sigma_0 = \{\}$

1. $D_0 = \{f(a), Y\}$ – ok we have a variable Y and a term not containing Y.
2. $\sigma_1 = \{\}\{Y=f(a)\} = \{Y=f(a)\}$
   $S_1 = S_0\{Y=f(a)\} = \{p(f(a), g(X)) ; p(f(a), f(a))\}$
3. $k = 1$

1. $D_1 = \{g(X), f(a)\}$ – fail we do not have a variable and a term not containing the variable

Not unifiable!

MGU Example 2.

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

$k = 0, \sigma_0 = \{\}$

1. $D_0 = \{a, Z\}$ – ok we have a variable Z and a term not containing Z.
2. $\sigma_1 = \{\}\{Z=a\} = \{Z=a\}$
   $S_1 = S_0\{Z=a\} = \{p(a,X,h(g(a)); p(a,h(Y),h(Y))\}$
3. $k = 1$

1. $D_1 = \{X,h(Y)\}$ – ok
2. $\sigma_2 = \{Z=a\}\{X=h(Y)\} = \{Z=a,X=h(Y)\}$
   $S_2 = S_1\{X=h(Y)\} = \{p(a,h(Y),h(g(a)); p(a,h(Y),h(Y))\}$
3. $k = 2$

MGU Example 2.

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

1. $D_2 = \{g(a),Y\}$ – ok
2. $\sigma_3 = \{Z=a,X=h(Y)\}\{Y=g(a)\} = \{Z=a,X=h(g(a)), Y=g(a)\}$
   $S_3 = S_2\{Y=g(a)\} = \{p(a,h(g(a)),h(g(a)); p(a,h(g(a)),h(g(a)))\}$
3. $k = 3$

1. $D_3 = \{\}$ – done!

MGU = $\sigma_3 = \{Z=a,X=h(g(a)), Y=g(a)\}$

MGU Example 2.

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

MGU = $\sigma_3 = \{Z=a, X=h(g(a)), Y=g(a)\}$

Double check:

$p(a,X,h(g(Z))\ \sigma_3 = p(a,h(g(a)),h(g(a)))$

$p(Z,h(Y),h(Y))\sigma_3 = p(a,h(g(a)),h(g(a)))$

Correct!

MGU Example 3.

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

$k = 0, \sigma_0 = \{\}$

1. $D_0 = \{X, Y\}$ – ok we have a variable X and a term (Y) not containing X.
2. $\sigma_1 = \{\}\{X=Y\} = \{X=Y\}$
   $S_1 = S_0\{X=Y\} = \{p(Y,Y), p(Y,f(Y))\}$
3. $k = 1$

1. $D_1 = \{Y,f(Y)\}$ – fail! We have a variable Y but the term f(Y) contains Y!

Not Unifiable

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. You will need to use this notation on the exam!

- "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. $\exists X.\ p(X) \wedge \forall Y.\ (d(Y) \rightarrow l(X,Y))$

Resolution Proof Example

2. No patient likes any quack

F2. $\neg(\exists X.\ p(X) \wedge \exists Y.\ q(Y) \wedge l(X,Y))$

3. Therefore no doctor is a quack.

Query. $\forall X.\ d(X) \rightarrow \neg q(X)$

Resolution Proof Example

Resolution Proof Step 3.

Convert to Clausal form.

F1. $\exists X.\ p(X) \wedge \forall Y.\ (d(Y) \rightarrow l(X,Y))$

$\exists X.\ p(X) \wedge \forall Y.\ \neg d(Y) \vee l(X,Y)$

$\forall Y.\ p(a) \wedge (\neg d(Y) \vee l(a,Y))$ #Skolem constant

1. $p(a)$

2. $\neg d(Y) \vee l(a,Y)$

F2. $\neg(\exists X.\ p(X) \wedge \exists Y.\ q(Y) \wedge l(X,Y))$

$\forall X.\ \neg p(X) \vee \forall Y.\ \neg q(Y) \vee \neg l(X,Y)$

$\forall X \forall Y.\ \neg p(X) \vee \neg q(Y) \vee \neg l(X,Y)$

3. $\neg p(X) \vee \neg q(Y) \vee \neg l(X,Y)$

Resolution Proof Example

Resolution Proof Step 3.

Negation of Query.

¬(∀X. d(X) → ¬q(X))

¬(∀X. ¬d(X) ∨ ¬q(X))

∃X. ¬d(X) ∧ ¬q(X)

¬d(b) ∧ ¬q(b) #Skolem constant

4. ¬d(b)
5. ¬q(b)

Resolution Proof Example

Resolution Proof Step 4.

Resolution Proof from the Clauses.

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

If we have to resolve two clauses that have the same variable, we always rename the variables in one clause so that there are no shared variables

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)
6. R[3b, 5]{R=b} (¬p(Z), ¬l(Z,b))
7. R[6a,1] {Z=a} ¬l(a,b)
8. R[7,2b] {Y=b} ¬d(b)
9. R[8,4] () #contradiction proving the query is true.

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?).

- We can 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

  $(\neg$ 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 a 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)) // $\forall$ X.$\forall$ Y. $\neg$p(X) ➔ p(Y)
2. (¬p(V), ¬p(W)) // $\forall$ V.$\forall$W. p(V) ➔ $\neg$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} ().

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?
  $\exists\ X.\ intell(X) \wedge \neg read(X).$
  ➔ $\forall\ X.\ \neg\ intell(X) \vee read(X)$
  ➔ $(\neg intell(X), read(X), answer(X))$

Example: do the resolution proof

1. (¬read(X), lit(X))
2. (¬dolp(X), ¬lit(X))
3. dolp(flip)
4. intell(flip)
5. (¬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. (¬dolp(flip), answer(flip))
9. R[8,3] answer(flip)

so flip is intelligent but cannot read!