

Introducing relations

Lecture 6 First Order Logic

We have seen that Propositional Logic is in general too tedious to use, and it can't express many things we want to express.

In this chapter, we introduce a more powerful logic, FOL, which can express many more things in a more succinct way. We will also see the techniques needed to perform reasoning in FOL.

Reference:

- Textbook chapter 8 and 9

Why **propositional** logic is **not expressive enough**?

- Our language consists **only of propositional symbol**...
So what? Every sentence is either true or false!
- and these propositional symbols are all **completely unrelated**.
This is a big problem: if we want to say "for any room, if the room has no wumpus and the room has no pit, then the room is okay", then the sentences "the room has wumpus", "the room has pit" and "the room is okay" must relate in some way for each room.
- So the first thing to do is to add a new type of sentences: **relation among objects**.
- The interpretation is **not hardcoded into the language**: it is up to the user to define what each object means, just like it is up to the user to define what each proposition means.

AI(0270)

AI(0270)-6.1

Predicates

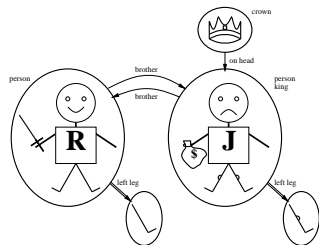
The basic sentences are called **predicates**:

$$\text{Predicate}(\text{Obj}_1, \text{Obj}_2, \dots, \text{Obj}_n)$$

where the Obj_i are "objects". Syntactically it is called **terms**.

Being sentences, they have a truth value of either true or false.

E.g.: in the world containing the King John and his brother Richard:



$\text{King}(\text{John})$ is true
 $\text{King}(\text{Richard})$ is false
 $\text{Brother}(\text{John}, \text{Richard})$ is true
 $\text{Brother}(\text{Richard}, \text{John})$ is true

The *King* predicate needs 1 object: **unary** predicate (or "property").
 The *Brother* predicate needs 2: **binary** predicate (or "relation").

AI(0270)-6.2

AI(0270)-6.3

Functions

- Some relations have the property that **for any given object, exactly one object is related to it**. E.g., LeftLeg, Mother, etc.
- For these objects, it makes sense to **indirectly** name the object. E.g., the LeftLeg of Richard, the Mother of John, etc.
We allow some object to have no "name", so that we don't need to invent a symbol for "left leg of richard", "left leg of mother of richard", etc.
- In logic we have **functions**: $\text{LeftLegOf}(\text{Richard})$, $\text{MotherOf}(\text{John})$, etc.
- Functions are "**total**", i.e., $\text{LeftLegOf}(\text{LeftLegOf}(\text{Richard}))$ is an object. But one can always say it is a special object *NoSuchObject*.
- There is no "procedure"** telling us who is the Mother of John. We will reason **without knowing what object** it refers to.
- An object might have multiple name (and be values of different functions). The logic needs some way to know what is **equivalent**...

The special relation: equality

- There is a special binary relation called "**=**" (**equality**).
- It's **interpretation is fixed** by the language: it is true if the two objects are the **same object**.
- For convenience, we write it in a special "infix" syntax: e.g.,

$$\text{Richard} = \text{Richard}$$

$$\neg(\text{Richard} = \text{John})$$

- Clearly, **sometimes we need to tell the reasoning system before the system knows about equality**: e.g., the latter relation must be told, or the reasoning system will not know Richard is not another name of John.
- Once equality of two objects is established, **everything that apply to the one object will also apply to the other object**.

AI(0270)-6.4

AI(0270)-6.5

Universal Quantifier

- Why **replacing propositions** like John_Is_King by **predicates** like $\text{King}(\text{John})$?
- Because we can then express things like "for any object being a king, it must be a person".
- But for this to be possible, we need two things: a **sentence** which express "**for anything**", and a **term** which express the "**anything**".
- How to write it? We should be familiar with this already:

$$\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$$

- A variable like x represents an **object**, not a **predicate**. We call this logic **First Order Logic** (FOL) because of this restriction.
Object is consider to be the "bottom" building blocks, then things derived directly from objects (predicates), then things derived from directly from predicates, ... Our logic allows variables to represent the bottom (first order) building block.

Meaning of universal quantifier

You can understand the **universal quantifier** like a **HUGE conjunction**, replacing the variable by each domain object symbol:

$(King(Richard) \Rightarrow Person(Richard))$
 $\wedge (King(John) \Rightarrow Person(John))$
 $\wedge (King(Crown) \Rightarrow Person(Crown))$
 $\wedge (King(NoSuchObject) \Rightarrow Person(NoSuchObject))$
 $\wedge \dots$

Implication is normally use to make universally quantified sentences.

Conjunction immediately within for-all usually have unintended meaning. Because we have a conjunct " $King(x) \dots$ ", so it says every object is a king—probably not what we really mean.

Note: by convention, we use lower case words for variables. A term with no variable (e.g., $John$ or $LeftLeg(John)$) is called a **ground term**.

AI(0270)-6.6

Existential Quantifier

We also have the **existential quantifier**, meaning "for some". E.g., there is some crown which is on head of John:

$\exists x Crown(x) \wedge OnHead(x, John)$

This can be interpreted like a **big disjunction**:

$(Crown(John) \wedge OnHead(John, John))$
 $\vee (Crown(NoSuchObject) \wedge OnHead(NoSuchObject, John))$
 $\vee (Crown(LeftLeg(John)) \wedge OnHead(LeftLeg(John), John))$
 $\vee (Crown(TheCrown) \wedge OnHead(TheCrown, John)) \vee \dots$

Conjunction is normally used to make existentially quantified sentences.

Implication and **disjunction** immediately within "exists" normally give sentences with unintended meaning.

E.g., $\exists x Crown(x) \vee OnHead(x, John)$ is true if $Crown(x)$ or $OnHead(x)$ is true for any x . In words, either something is a crown, or something is on John's head.

AI(0270)-6.7

Default values

What will happen if there is **no object in the system at all**?

This normally won't happen, but our reasoning system must be able to cope with it in a nice way.

We will use the following convention: if there is no x at all...

- $\forall x P(x)$ is **true**.
- $\exists x P(x)$ is **false**.

This makes our logic simpler: e.g., to keep the value of $\forall x P(x)$ when the system learns objects one by one:

- Initially say $\forall x P(x)$ is true (by definition).
- When a new object X arrives, find $P(X)$, "and" it together with the kept value for $\forall x P(x)$. Similar for \exists .

AI(0270)-6.8

Syntax of FOL

Now we have everything in place. Let's summarize:

- A **sentence** can be either **simple** or **complex**.
- Simple sentences** are always **predicates**, or the predefined **equality** relation.
- Complex sentences** can be formed by using our logic **connectives** (and, or, not, implies, equivalent to) on **sentences**.
- Complex sentences** can also be formed by introducing a variable and **quantify** a sentence with the variable **universally** or **existentially**.
- A **predicate** is composed of a predicate symbol and some **terms**.
- A **term** can be an **object symbol** or a **variable symbol** introduced previously. It can also be a **function** operating on some terms. So we can have $FatherOf(MotherOf(June))$, $MotherOf(x)$, etc.

AI(0270)-6.9

Particulars on the syntax

- The **precedence** except quantifiers: same as propositional logic, except that **equality** has very high precedence.
- Quantifier** group and bound everything on its right, unless constrained by parentheses. So $\neg \forall x P(x) \vee Q(x) \Rightarrow \exists y Q(y)$ means:

$\neg(\forall x (((P(x) \vee Q(x)) \Rightarrow (\exists y Q(y))))))$

- If a variable appears in multiple quantifiers, e.g.,

$\forall x P(x) \Rightarrow \exists x Q(x)$

The variable is bounded to the **closest enclosing quantifier**. E.g.,

$\forall x(P(x) \Rightarrow (\exists y Q(y)))$

But we will avoid using this rule, by never using the same variables for 2 quantifiers in the same sentence.

AI(0270)-6.10

Queries with existential quantifier

Suppose we ask a question that is **existentially quantified**, e.g.,

$\exists x Child(x, Alice)$

If the answer is affirmative, we want a **proof** telling us **some objects** x that satisfies the statement, by giving us a **set of substitutions**, e.g.,

$\{ \{x/Bob\}, \{x/Cindy\} \}$

Set of sets, since there can be multiple substitutions, each for multiple variables.

So in effect the query is a question like "do you know any child of Alice", and the answer is something like "Yes, they are Bob and Cindy."

What if the reasoning program doesn't know the values? E.g., what will happen if we tell KB that Room-1-2 is breezy, and then ask "do you know any neighbouring room of Room-1-2 that contains a pit?"

We will know the answer when we talk about inference.

AI(0270)-6.11

Example: numbers

- What we have defined is enough to represent numbers, so much that a reasoning system can perform arithmetics like addition (slowly).
- To do so, we introduce the symbol 0 representing the number 0. We introduce the unary predicate *NatNum*, true for all natural numbers.
- We define a function called the **successor function** *S*, so the successor (next number) of 0 is *S*(0), the successor of *S*(0) is *S*(*S*(0)), etc.
- We want all of them to be distinct. The following suffices:

$$\forall n \quad 0 \neq S(n)$$

$$\forall m, n \quad m \neq n \Rightarrow S(m) \neq S(n)$$
- We want to define addition. The following suffices:

$$\forall m \quad \text{NatNum}(m) \Rightarrow +(m, 0) = m$$

$$\forall m, n \quad \text{NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow +(m, S(n)) = S(+ (m, n))$$

AI(0270)-6.12

Syntactic sugar

- So natural numbers and addition can be defined in FOL, starting from 1 predicate, 2 functions and 6 axioms.
 - 2 axioms for defining *NatNum*, 2 for making equality, 2 for addition.
- We can further define subtraction, multiplication, etc.
- This suffices from a theoretical point of view, but is very cumbersome. Normally we give names to numbers, like 1, 2, etc. We also allow functions like additions to be written in infix notation like 1+2.
- Turning a concept into a small set of axioms: **axiomatization**.
- The axioms we see are called **Peano axioms**, with one rule **missing**: mathematical induction. But that is 2nd-order logic.
 - For any unary "parametrized sentence" *s* (*n*), if *s* (0) is true, and $\forall n \quad \text{NatNum}(n) \wedge s(n) \Rightarrow s(n+1)$ is true, then $\forall n \quad \text{NatNum}(n) \Rightarrow s(n)$ is true. But FOL disallow us to write *s*, which is not a term.

AI(0270)-6.13

Sets and lists

- Similarly, sets and lists can be axiomatized in FOL.
- Instead of deriving them in class, we simply let you read the book.
- Syntactic sugar:

	Set	List
Empty	{ }	[]
With element	{ <i>a</i> , <i>b</i> }	[<i>a</i> , <i>b</i>]
Membership	<i>a</i> ∈ <i>S</i>	<i>Member</i> (<i>a</i> , <i>L</i>)
Adding (prepending) an element	{ <i>a</i> <i>S</i> }	[<i>a</i> <i>L</i>]
Subset	<i>S</i> ₁ ⊆ <i>S</i> ₂	
Union	<i>S</i> ₁ ∪ <i>S</i> ₂	
Intersection	<i>S</i> ₁ ∩ <i>S</i> ₂	

AI(0270)-6.14

Higher order logic?

There is one question: **why stop at first-order logic?** Why leaving out Mathematical Induction?

- **Higher-order logic** allows **variables** to **represent** more complicated things, like **predicates**, etc.
- This vastly increase the **expressiveness** of the logic.
- But there is an unlucky fact proven by the **Gödel's incompleteness theorem**: once we have Mathematical Induction, inference cannot be **complete** and **sound** at the same time.
- In other words, there are true statement in the system that cannot be proven, no matter how much time you spend!
- To be able to reason effectively, we must restrict ourselves to something less complex.

AI(0270)-6.15

Other logic

So we cannot add second-order logic statements. But there are other directions of extensions that are more useful to agent designers...

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What agents believe about sentences)
Propositional Logic	facts	true, false, unknown
FOL	facts, objects, relations	true, false, unknown
Temporal logic	facts, objects, relations, time	true, false, unknown
Probability theory	facts, objects, relations	Degree of belief
Fuzzy logic	facts with degree of truth	range of truth believed

Is it impossible for these to be expressed in FOL? Not quite. E.g., we can axiomatize time to use FOL for temporal logic.

But having the agent **directly** aware of the phenomenon does help the agent to make more efficient decisions. We will see some of them later in the course.

AI(0270)-6.16

The wumpus world: telling and asking

- What to **TELL** the KB when percept arrives?
- The percept must be associated with **time** to make it useful. Let's denote time by integers: a "percept sentence" looks like:

$$\text{Percept}([\text{Stench}, \text{Breeze}, \text{Glitter}, \text{None}, \text{None}], 5)$$
- The KB has rules to turn percepts sentence into knowledge. E.g.,

$$\forall t, s, b, m, c \quad \text{Percept}([s, b, \text{Glitter}, m, c], t) \Rightarrow \text{Glitter}(t)$$
- How to decide on an action? Make a **query sentence**:

$$\exists a \quad \text{BestAction}(a, 5)$$
- How to define *BestAction*? A simple one: if we sense Glitter, best action is Grab. So our KB has a "reflex rule":

$$\forall t \quad \text{Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t)$$

AI(0270)-6.17

Representing rooms, pits and wumpus

- How to represent rooms? We simply use a 2-element list like [1, 1].
- How to keep track of the current room? E.g., $At([2, 3], 5)$.
At time 5, the agent is at room 2, 3.
- We write rules to turn temporal knowledge into properties. E.g.,
 $\forall s, t \quad At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s)$
- Where is the pits? Just have a predicate like $Pit([2, 4])$
- To know about pits, we need to know adjacency. So we need...
 $\forall x, y, a, b \quad Adjacent([x, y], [a, b]) \Leftrightarrow$
 $[a, b] \in \{[x + 1, y], [x, y + 1], [x - 1, y], [x, y - 1]\}$
- With this, we can define rules to discover pits.

AI(0270)-6.18

Diagnostic rules vs. Causal rules

There are two ways to let the agent know where are the pits...

- **Diagnostic** rules: tell the agent what percepts means. E.g.,
 $\forall s \quad Breezy(s) \Rightarrow \exists r \quad Adjacent(r, s) \wedge Pit(r)$
 $\forall s \quad \neg Breezy(s) \Rightarrow \neg \exists r \quad Adjacent(r, s) \wedge Pit(r)$
- **Causal** rules: tell the agent what can cause percepts. E.g.,
 $\forall r \quad Pit(r) \Rightarrow \forall s \quad (Adjacent(r, s) \Rightarrow Breezy(s))$
 $\forall s \quad (\forall r \quad Adjacent(r, s) \Rightarrow \neg Pit(r)) \Rightarrow \neg Breezy(s)$

Both work (and in fact, the two above are equivalent) in our situation.

In situations involving **changing rules**, it is much easier to use causal rules (since they change less rapidly).

Reasoning based on causal rules is said to be **model-based**.

AI(0270)-6.19

Inference

Now is time to think about how to do automatic reasoning. Recall that in Propositional Logic, we have three strategies to do inference:

- Use **truth-table**.
- Use **resolution**.
- Restrict sentences to Horn clauses, and use **chaining**.

What about FOL? We can't apply the exact algorithms we know before, since now it involves **quantifiers** that we don't know how to deal with.

Is it possible to **replace quantifiers** by something else so that we can apply the 3 methods we already know for propositions?

To do this we need to know a bit more about quantifiers .

AI(0270)-6.20

What happens when quantifier's nests?

- When the **same quantifier's nest**, the **meaning does not change** if we change the order. E.g., the following sentences are the same.

$$\forall x \forall y \quad Student(x) \wedge Exam(y) \Rightarrow \neg Like(x, y)$$

$$\forall y \forall x \quad Student(x) \wedge Exam(y) \Rightarrow \neg Like(x, y)$$

- Most of the time we just write it like:

$$\forall x, y \quad Student(x) \wedge Exam(y) \Rightarrow \neg Like(x, y)$$

- This is true **also for existential quantifiers**

- But what if **different types of quantifier's nest**? Let's see:

$$\exists x \forall y \quad Student(x) \wedge (Exam(y) \Rightarrow Like(x, y))$$

$$\forall y \exists x \quad Student(x) \wedge (Exam(y) \Rightarrow Like(x, y))$$

- They **mean completely different** things! (Some student likes all exams; each exam has some student who like it.)
But "each exam has some student who like it" is not really accurate. Why?

AI(0270)-6.21

Moving quantifiers towards the top-level

We usually don't want the complication to have a quantifier in the "middle". Instead we want them to have "**top-level**" **scope**.

Many rules in this chapter requires all quantifier to be top-level.

E.g., instead of $P(A) \wedge (\forall x \quad Q(A, x) \vee R(x))$

we prefer $\forall x \quad P(A) \wedge (Q(A, x) \vee R(x))$

Just moving them out usually work, since \forall is really a big conjunction which moves readily in and out of other conjunctions and disjunctions (by **distribution** rule).

But, for **negation**, we need a little bit of twist, since when we move negation into a conjunction it becomes a disjunction, as dictated by **De Morgan's rule**.

AI(0270)-6.22

The relationship between quantifiers

$(\neg x \vee \neg y)$ and $\neg(x \wedge y)$ are the same, while $(\neg x \wedge \neg y)$ and $\neg(x \vee y)$ are the same. So the following are equivalent:

$$\exists x \neg P \quad \text{and} \quad \neg \forall x P$$

$$\forall x \neg P \quad \text{and} \quad \neg \exists x P$$

So we can **move quantification out of negations**. How about **conditionals** and **biconditionals**? We can **eliminate** them. E.g., The following sentences are all equivalent:

$$\neg \forall x \quad P(x) \Rightarrow \exists y \quad Q(x, y)$$

$$\exists x \quad \neg(P(x) \Rightarrow \exists y \quad Q(x, y))$$

$$\exists x \quad \neg(\neg P(x) \vee \exists y \quad Q(x, y))$$

$$\exists x \quad P(x) \wedge \neg \exists y \quad Q(x, y)$$

$$\exists x \quad P(x) \wedge \forall y \quad \neg Q(x, y)$$

$$\exists x \forall y \quad P(x) \wedge \neg Q(x, y)$$

AI(0270)-6.23

Variable substitutions

- Our rules will **manipulate the variables** of the sentence. But we have no syntax to do so yet!
- Or rather, we have, but didn't formalize it. Recall that we can have **substitution** like $\{x/Cindy, y/Bob\}$
- This can be seen as renaming x to $Cindy$, y to Bob . We allow renaming from variables to variables as well.
- We use the notation $SUBST$ to represent a substitution. E.g.: $SUBST(\{x/Cindy\}, Child(x, Alice))$ means $Child(Cindy, Alice)$

AI(0270)-6.24

Rule for Universal quantifiers

Since universal quantifiers are really big conjunctions, to deal with it we can "break the conjunctions to the conjuncts".

- **Universal Instantiation (UI)**: for any variable v , sentence α , and ground term (i.e., variable-free term) g , we have

$$\frac{\forall v \alpha}{SUBST(\{v/g\}, \alpha)}$$

E.g., if we know $\forall t \text{ Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t)$, then we know $\text{Glitter}(5) \Rightarrow \text{BestAction}(\text{Grab}, 5)$, the substitution being $\{t/5\}$.

Repeated application of UI allows us to make many different sentences from a single universally quantified sentence.

AI(0270)-6.25

Rule for Existential quantifiers

How about existential quantifier? We do something like this:

- **Existential Instantiation (EI)**: for any sentence α , variable v , and constant symbol K not appearing anywhere else in KB:

$$\frac{\exists v \alpha}{SUBST(\{v/K\}, \alpha)}$$

E.g., given $\exists r \text{ Adjacent}(r, [1, 2]) \wedge \text{Pit}(r)$, we can perform the substitution $\{r/R_{12345}\}$, and get $\text{Adjacent}(R_{12345}, [1, 2]) \wedge \text{Pit}(R_{12345})$, given that R_{12345} is never used elsewhere. R_{12345} is called a **Skolem constant**.

It doesn't add any knowledge to the KB. The resulting rule is **equivalent** to the original rule, so the original rule can be discarded after EI, and we are left with a rule with no \exists .

It makes sense because an object can have multiple names. Later when we find $[2, 2]$ has pit, we can say $R_{12345} = [2, 2]$

AI(0270)-6.26

Conjunctive Normal Form in FOL

We usually want things to be simple: just like propositional logic we have CNF in FOL. A FOL sentence is in CNF if it is consisting of **clauses**, combined with a big conjunction:

- As in propositional logic, each clause is a **disjunction of terms**, each term being a **predicate** or its **negation**.
- The predicates can contain **variables**. But all variables are treated as **universally quantified**, all on the **leftmost** of the clause. There is no existential quantifier.
 - Indeed, if we make sure all variable are of different names, they can all be written outside the big conjunction. We normally won't write out the \forall : it is implied.
- We can also assume that no two terms in the same sentence is the same (redundant) or the negation of each other (provide no knowledge).

AI(0270)-6.27

Example violations

Let's see some non-CNF sentence and their corresponding CNF.

- $\exists s \text{ Like}(s, CS/S0270)$ —don't use \exists . CNF: $\text{Like}(S, CS/S0270)$
- $\forall x \text{ Student}(x) \Rightarrow \forall y \text{ Exam}(y) \Rightarrow \text{Hate}(x, y)$ —can't use \forall inside clause. Also, can't use implication.
CNF: $\neg \text{Student}(x) \vee \neg \text{Exam}(y) \vee \text{Hate}(x, y)$
- $\neg \text{OK}(x) \vee (\neg \text{Breezy}(x) \wedge \neg \text{Smelly}(x))$ —can't use conjunction.
CNF: two clauses $\neg \text{OK}(x) \vee \neg \text{Breezy}(x)$
 $\neg \text{OK}(x) \vee \neg \text{Smelly}(x)$
- $P(x) \vee P(x) \vee Q(x) \vee \neg Q(x)$ —redundant (no knowledge in it).

AI(0270)-6.28

Normalization

1. **Turn implications to disjunctions** ($p \Rightarrow q \equiv \neg p \vee q$). If there are equivalences, turn it into two rules ($p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$).
2. **Move all negations inwards** from outer-most negation to inner-most ones. This requires the use of the De Morgan's rule.
3. **Rename variables** so that there is no quantifier using the same variable name.
4. **Move all quantifiers** to the left. This never change the meaning, as the variables are now all distinct, and no negation is on the way.
5. **Skolemize**: remove all existential quantifiers by new function or constant symbols. See next slide.
6. **Distribute and flatten**: move \vee inwards with the distribution rule ($p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$). Break top-level conjunctions to clauses.
7. **Remove redundancy**. If a term appear twice, remove one. If a term and its negation both appears, remove the whole clause.

AI(0270)-6.29

Skolemization

- To **remove** “ \exists ”, we can sometimes use Existential Instantiation. E.g., $\exists s \text{ Like}(s, \text{CS/S0270})$ becomes $\text{Like}(S_1, \text{CS/S0270})$, where S_1 is a Skolem constant.
- But it requires the \exists to be top-level. What about the following?
 $\forall x \exists y \text{ Loves}(x, y)$
- Each x can use a **different value** for y , so we might not have a **single** constant which can represent y .
- We introduce a new function: **Skolem function**. Now it becomes
 $\forall x \text{ Loves}(x, \text{Lover}(x))$
- In general, the skolem function will take all variables of \forall enclosing it as arguments. E.g., $\forall x \exists y \forall z \exists h P(x, y, z, h)$ becomes
 $\forall x, z P(x, F(x), z, G(x, z))$

AI(0270)-6.30

Example

$$\forall x \text{ Breezy}(x) \Rightarrow \exists y \text{ Neighbour}(x, y) \wedge \text{Pit}(y)$$

- Remove implications: $\forall x \neg \text{Breezy}(x) \vee \exists y \text{ Neighbour}(x, y) \wedge \text{Pit}(y)$
- Move negations inwards, rename variables: nothing to do.
- Move quantifiers left: $\forall x \exists y \neg \text{Breezy}(x) \vee (\text{Neighbour}(x, y) \wedge \text{Pit}(y))$
- Remove quantifiers: $\neg \text{Breezy}(x) \vee (\text{Neighbour}(x, F(x)) \wedge \text{Pit}(F(x)))$
- Distribute disjunctions and flatten:
 $\neg \text{Breezy}(x) \vee \text{Neighbour}(x, F(x))$
 $\neg \text{Breezy}(x) \vee \text{Pit}(F(x))$

Now it is in CNF. And, it is **guaranteed to carry the same knowledge** as the original sentence (although nearly completely unreadable).

The resulting sentences in effect forces $F(x)$ to mean “a room around x with a pit, if there is one”. If you know that it is a bit easier to read.

AI(0270)-6.31

We have an inference engine!

- Every existentially quantified sentence can be replaced by a sentence with Skolem constants or Skolem functions.
- Every universally quantified sentence can be replaced by many sentences, one for each object in the domain.
- After these, we have a lot of predicates operating on different terms. We can assign a different propositional symbol for each term.
- Now all we get is... **propositional logic!**

We call this **propositionalization**. It works as long as the domain is finite. But what if we have functions (infinitely many objects)?

Luckily, **every provable statement in FOL has a finite proof**. So we can use **iterative deepening** to deal with it.

E.g., first consider only *John*, then also *MotherOf(John)*, then *MotherOf(MotherOf(John))*, etc.

AI(0270)-6.32

Non-entailed sentences: loop forever?!

- The algorithm is **sound** and **complete**, i.e., it says yes whenever the query is provable, and won't say yes for any nonentailed sentence.
- What if the query is unprovable? We search deeper and deeper, each time we get a negative result (saying the query is not proved)...
- ... and **don't know when to stop!**
This is not surprising: all IDS works this way.
- Is it a problem of our inference strategy, or a problem of FOL in general? Unluckily, it is a problem of FOL.
This is surprising: no matter what algorithm you write, it behaves like that.
- FOL is said to be **semi-decidable**:
 - There is a program that says “yes” to all entailed queries.
 - There is no program that says “no” to all nonentailed queries!

AI(0270)-6.33

Curse of efficiency

- Let's say the domain contains n objects (including all function values). Then every k -ary predicate creates n^k propositions!
- and every universal quantifier give rise to n times as many sentences. E.g., if we have a sentence with k universally quantified variables, each becomes n^k sentences!
- It is clear that using propositions to deal with FOL is **not very practical**.
- How to be more efficient? Instead of **instantiating** everything at the beginning, we want instantiation to be intelligent: instantiating **only when needed**.
- E.g., suppose we have $\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$. We don't want to instantiate x until somebody ask, e.g., $\text{Evil}(\text{John})$...
- ... even then we want only to instantiate $\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$.

AI(0270)-6.34

The unification problem

- Suppose we want to prove $P(A, F(B))$, and a rule tells something about $P(x, y)$. We want to know that the “correct” substitution to apply is $\{x/A, y/F(B)\}$.
- What if we want to prove $P(A, F(G(x)))$, and a rule tells something about $P(y, F(z))$? We want to know $\{y/A, z/G(x)\}$ is the best substitution to apply.
We can actually apply $\{y/A, z/G(A)\}$, but that is just a sub-case of the above. So having the above we automatically know this.
- What if we want to prove $P(A, F(G(x)))$, and a rule tells something about $P(y, F(A))$? We want to know that nothing can be done.
There is one loophole here: what if $G(B) = A$? At current moment, let's assume that we don't have equality. We will tell how to deal with it later.

Unification: Given two lists of terms, find the most general substitution, or “unifier” (**mg**u), that makes the two lists exactly the same.

AI(0270)-6.35

How to find mgu?

Suppose we want to unify x and y , when we already have the set of substitutions θ :

- **Degenerated case**— x is the same as y : do nothing.
- **Basic case**— x is a variable not substituted by θ , and y is either not a variable or is not substituted by θ :
 - If x occurs in y and its substitutions, no unifier can be found.
 - Otherwise, we add $\{x/y\}$ to θ .
- **Substituted variable**— x or y is a variable substituted by θ : do the substitution before retry.
- **Complex forms**— x and y are both Predicate or Function: If x and y has the same form, recursively unifier each component. Otherwise fail.
 - Same form: same predicate symbol, and same number of components.

AI(0270)-6.36

In code...

```
# subst is the bindings that is already done so far. At the start it is empty.
# The function returns a substitution
def Unify(x, y, subst):
    if x = y:
        return subst # nothing to add
    if IsVariable(x):
        return UnifyVar(x, y, subst) # just add the variable
    if IsVariable(y):
        return UnifyVar(y, x, subst)
    if IsFunction(x) and IsFunction(y):
        if not x.func() = y.func():
            return failure # different functions can't unify
        return Unify(x.args(), y.args, subst)
    if IsList(x) and IsList(y):
        subst0 = Unify(First(x), First(y), subst)
        if subst0 = failure:
            return failure
        return Unify(Rest(x), Rest(y), subst0)
    return failure
```

AI(0270)-6.37

Unifying variables

```
def UnifyVar(var, x, subst):
    if (subst substitutes var):
        val = substitution of var in subst
        return Unify(x, val, subst)
    if (subst substitutes x):
        val = substitution of x in subst
        return Unify(var, val, subst)
    if (Occurs(var, x, subst)): # var occurs in x or its substitutions in subst
        return fail
    return Append(subst, {var, x})
```

Here the most time consuming step is the **occurrence check**. E.g., if we want to unify x with $F(p, q, r)$, then we must check whether p, q and r is substituted by something involving x in $subst$.

The amount of time involved ranges from linear to quadratic.

Many logic programming systems simply **omit the occurrence check**, which make the system incomplete (loops forever for entailed sentences).

AI(0270)-6.38

Lifting Modus Ponens

Let's try doing chaining in FOL first.

- We want to be able to apply Modus Ponens, **without going to Propositional Logic**. Modus Ponens originally looks like this:

$$\frac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$$

- In FOL, we will modify Modus Ponens so that it performs substitutions: Suppose $SUBST(\theta, p_1) = p_1', \dots, SUBST(\theta, p_n) = p_n'$, then

$$\frac{p_1', \dots, p_n', (p_1 \wedge \dots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

- We say that it **lifts** Modus Ponens from PL to FOL.
- E.g., if we know $King(x) \wedge Greedy(x) \Rightarrow Evil(x)$, and we know $King(John)$ and $Greedy(John)$, we infer $Evil(John)$.

AI(0270)-6.39

Forward Chaining in FOL

How to do forward chaining in FOL?

- As in Propositional Logic...
 - All knowledge must be in **definite clause**, i.e., has exactly one positive predicate.
 - Everytime we pick a definite clause in the KB, and see if KB can prove all the premises of the sentence.
- Unlike Propositional Logic chaining, we need to use unification to prove each premise, since the facts can have variables (e.g., $Greedy(x)$, everything is greedy).

AI(0270)-6.40

A simple implementation

```
def FOL_FC_Ask(KB, query): # returns either a substitution or false
    while true:
        new = []
        for r in KB.clauses:
            new_r = r.rename_vars()
            premises = new_r.premises
            for each substitution s s.t. SUBST(s, p) ∈ KB for all premises p:
                result = SUBST(s, new_r.conclusion)
                if result is not a renaming of a fact in KB:
                    new.append(result)
                    unifier = Unify(query, result)
                    if unifier != fail:
                        return unifier
        if new = []:
            return false
        Add all facts in new as facts of KB
```

How to find each substitution s ? We'll come back to this a bit later.

AI(0270)-6.41

Example

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

Query: Is West a criminal?

- First, we need to convert the knowledge into a form suitable for chaining, i.e., definite clauses.
- Then we convert the query to a single predicate.
- Finally we apply Forward Chaining to prove the knowledge.

AI(0270)-6.42

Converting to

1. $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
2. $Own(Nono, M_1)$
3. $Missile(M_1)$
4. $Missile(x) \wedge Own(Nono, x) \Rightarrow Sell(West, x, Nono)$
5. $Missile(x) \Rightarrow Weapon(x)$
6. $Enemy(x, America) \Rightarrow Hostile(x)$
7. $American(West)$
8. $Enemy(Nono, America)$

Our query: $Criminal(West)$

AI(0270)-6.43

Applying FC

We have 4 implication clauses: (1), (4), (5) and (6).

First iteration: (1) has no binding that work. (4)–(6) adds these to KB:

9. $Sell(West, M_1, Nono)$
10. $Weapon(M_1)$
11. $Hostile(Nono)$

Second iteration: (1) has a binding that work, giving the result.

12. $Criminal(West)$

At this point the algorithm stops. Actually, even if it didn't prove the query (e.g., because we asked something which is not entailed), the algorithm would stop, since no more knowledge can be inferred.

Of course it terminates... there's a finite number of objects.

AI(0270)-6.44

Analysis of the algorithm

- It is **sound**, since applications of Modus Ponens is correct.
- If there is **no functional symbols**, the same proof as FC in PL shows that FC is **complete**.
- Then we can lift the proof from PL to FOL, to prove that FC in FOL can also prove all entailed predicates. But we will skip the proof.
The proof is similar to the proof for resolution, that we will show later.
- So FC in FOL is also **complete**.
- But there is a problem: if there is a functional symbol (so that the domain is infinite), when a query is false it loops forever.
- This is a general problem of FOL, and cannot be avoided, since entailment for **definite clauses** is also semidecidable.
- Still, we want to ask whether it is possible to make things faster.

AI(0270)-6.45

Matching rules with known facts

- In the inner loop, we need to find every possible substitution so that premises are satisfied.
- E.g., we might need to check for $Missile(x) \wedge Owns(Nono, x)$.
- How to do that? We can find all facts that matches $Missile(x)$, and then see whether any of them matches $Owns(Nono, x)$.
- But if $Owns(Nono, x)$ has much fewer answers than $Missile(x)$, then it makes sense to **reverse the ordering**.
- What if the sentence is **much longer**, say containing 10 premises? It turn out that the matching required is NP-hard.
Indeed, all CSP can easily be formulated as one such matching.
- Luckily, most real-world KB has only short sentences, with small arity functions.

AI(0270)-6.46

Avoiding repeated application of Modus Ponens

- The FOL_FC_Ask algorithm actually repeats all the works that has been done in the previous iterations.
- When a clause is tried again, the same premises are tried, producing exactly the **same** result, and is **discarded** as a "renamed" clause in KB.
- One way to avoid it: at iteration t , make sure that applications of Modus Ponens have **at least 1 premise** that is known **only** at iteration $t - 1$, not before.
- Reason: if all premises of an application to Modus Ponens are known before that, it has been done before, and there's no point repeating it.
- So instead of "**for r in $KB.clauses$** ", we only iterate through clauses with a premise **newly known** at the last iteration, and set the bindings for them **before** looking for other bindings of it.
Further improvement is based on avoiding checking everything everytime.

AI(0270)-6.47

Backward chaining

- There is an inefficiency of FC that is intrinsic to its **data-driven** characteristics: it finds a lot of irrelevant facts.
- In most applications, FC is restricted to a subset of clauses, where it is really desirable to know every derivable knowledge.
- E.g., in the wumpus world, it is desirable to use FC to determine all rooms that are **breezy, smelly**, etc.
It "caches" the results in KB, so later when we want to know if a cell has a wumpus, we consult KB directly without working through individual percepts.
- For other clauses, a more focused, goal-driven approach is required: backward chaining (BC).
- The idea is very simple: work backwards from goal to facts, using the implication clauses that we have.

AI(0270)-6.48

The algorithm

A simple implementation looks like this...

```
# goals is a list of goals to satisfy
# bindings is the known substitutions
# The function returns a "set" of substitutions
def FOL_BC_Ask(KB, goals, known_substs):
    if goals is empty:
        return {known_substs} # only 1 answer
    curr_goal = goals.First() # try this first
    answers = {} # Set of answers, initially empty
    for clause in KB.clauses with conclusion unifiable with curr_goal:
        s = Unify(clause.conclusion, curr_goal)
        new_goals = [Subst(s, p) for p in clause.premises]
        new_goals.AppendAll(goals.Rest())
        new_substs = Compose(known_substs, s)
        answers.Union(FOL_BC_Ask(KB, new_goals, new_substs))
    return answers
```

It's not particularly efficient. Prolog improves the efficiency . . . while sacrificing completeness.

AI(0270)-6.49

Variables in query

What will happen if we ask *Criminal(c)*?

- The unification process will try to unify *Criminal(c)* to something concrete, in particular *West*.
- At the end, the search returns true, with the substitution {*c/West*}.
- What it shows? It means that *Criminal(West)* is provable.
- I.e., our query actually means $\exists c$ *Criminal(c)*.

So in backward chaining, variables in **knowledge base** is **universally** quantified, while variables in **query** is **existentially** quantified!

This is unique to chaining. Also, there is no way to ask "is it provable that everybody is a criminal?" in chaining.

In resolution we can do it.

AI(0270)-6.50

Resolution Refutation in FOL

- Propositionalization is slow, Chaining allows only definite (or Horn) clauses to be reasoned. For general FOL reasoning, one use **resolution**.
- Let's recall what needs to be done for PL resolution:
 1. **Pre-process** all sentences so that all clause are in CNF, i.e., a disjunction of propositions.
 2. **Negate the query**, giving us a sentence. Convert it into CNF.
 3. In each iteration, repeatedly choose two sentences known in the previous iteration, and apply **resolution** to them. Add the resulting knowledge to KB.
 4. If at any time the **empty** clause is generated, the query is proved.
 5. If at an iteration, no new knowledge can be derived using resolution, we declare the query to be false.

AI(0270)-6.51

Inference rule: Generalized Resolution

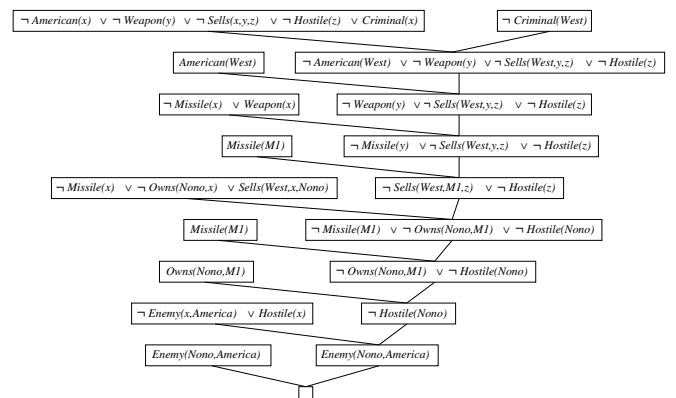
- Lift resolution for use with FOL.
- As in Modus Ponens, this involves unifications .
- **Generalized Resolution:** for (positive or negative) terms $p_1, p_2, \dots, p_m, q_1, q_2, \dots, q_n$, where $\text{UNIFY}(p_i, \neg q_j) = \theta$:

$$\frac{p_1 \vee p_2 \vee \dots \vee p_m, \quad q_1 \vee q_2 \vee \dots \vee q_n}{\text{SUBST}(\theta, \quad p_1 \vee p_2 \vee \dots \vee p_{i-1} \vee p_{i+1} \vee \dots \vee p_m \vee q_1 \vee q_2 \vee \dots \vee q_{j-1} \vee q_{j+1} \vee \dots \vee q_n)}$$

- In other words, once we have **substitutions** that get p_i and q_j to be **exact reverse of each other**, we can **combine** two sentences by **concatenating** them, make the substitution, and **removing** both p_i and q_j .

AI(0270)-6.52

Example proof tree



AI(0270)-6.53

Answering existential questions

- What if we instead ask $\exists x \text{ Criminal}(x)$?
- Of course, we can still **negate** it to get $\neg \text{Criminal}(x)$ (universally quantified) and **perform refutation**.
- The question is, **how to get the variable binding?**
Recall that for existential queries, we want a substitution list for the variable.
- During the refutation proof, there will be a **substitution made for x** .
If not, then we can conclude that x can really be anything.
- This gives us the binding required.
- If we want a **list of bindings**, we can **continue the search** for alternative proof of *False*, remembering not to bind x to the values already found.
- Unluckily, there can be **non-constructive proof**. Then there the binding is not meaningful. (It involves a Skolem constant/function).

AI(0270)-6.54

How good is it?

- Like in PL, resolution refutation is **sound and complete**.
- In other words, for entailed sentences, resolution with “proof by contradiction” will give a proof after a **finite** number of steps.
For nonentailed sentences it can loop forever, or simply say “no” after some steps. Remember that entailment in FOL is semi-decidable.
- Why? Because we can **lift the proof** in the propositional context to the FOL context.
- To show this, we start (without proof) with the “Herbrand's theorem”:

If a set of CNF clauses are unsatisfiable, then there is a finite set of their instantiation, formed **only** using the function and object symbols of the clauses, is also unsatisfiable.

AI(0270)-6.55

Completeness of FOL

- Suppose we give a provable query to the resolution algorithm. I.e., we have a set of FOL clauses S being unsatisfiable.
- By Herbrand's theorem, there is a **set of instantiations** S' being unsatisfiable.
- We now propositionalize it. We know the resolution closure must contain the empty clause, by the proof in the last chapter.
- Since propositionalization is just mechanical substitutions, we conclude that the original set S' is also provable using resolution.
- The remaining question is: in FOL resolution, we work without going to propositions. **Will this cause the proof to be missed?**

AI(0270)-6.56

The lifting lemma

- How to show that the proof will not be missed? This requires exact and boring **properties of mgu**. This results in the following:

Suppose there are two FOL clauses C_1 and C_2 with no shared variables, with two instantiation C_1' and C_2' .

If C' is the result of resolving C_1' and C_2' , then there is one way to resolve C_1 and C_2 so that the result C has an instantiation C' .

In other words, for every propositional resolution, there is a FOL resolution that does the same thing.

- Therefore, for each propositional resolution done, we can perform the **corresponding** FOL resolution. The result is always a generalization of the result of the propositional resolution.
- At the end we get the empty clause, which **only generalization is the empty clause itself**. I.e., we will get a proof.

AI(0270)-6.57

Dealing with equality by axioms

Now is time to talk about **how to deal with equality**.

Strategy 1: **Axiomize equality**. E.g.,

$x = x$	Reflexive
$x = y \Rightarrow y = x$	Commutative
$x = y, y = z \Rightarrow x = z$	Associative
$x = y \Rightarrow F(x) = F(y)$	Function values, one per function
$x = y \Rightarrow R(x) \Leftrightarrow R(y)$	Relation truth, one per relation
...	

- This will give us the full power of equality, without modifying our reasoning engine!
- But this can simplify sentences and also complicates it, so it generates a lot of useless sentences.

AI(0270)-6.58

Dealing with equality with one more rule

There is an alternative: add another rule to supplement resolution.

- **Demodulation**: have a separate rule that performs substitution. E.g., if $Unify(x, z) = \theta$ and a clause with a predicate $m_n[z]$ containing z , then

$$\frac{x = y, \quad m_1 \vee \dots \vee m_n[z]}{SUBST(\theta, m_1 \vee \dots \vee m_n[y])}$$

E.g., if we have $F(B) = A$ and $Q(p, F(p))$, then we have $Q(B, A)$. ($x = F(B), y = A, z = F(p)$). But this can't use clauses like $P(x) \vee x = A$.

- **Paramodulation**: Extend the rule to make it complete.

$$\frac{l_1 \vee \dots \vee l_k \vee (x = y), \quad m_1 \vee \dots \vee m_n[z]}{SUBST(\theta, l_1 \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_n[y])}$$

E.g., if we have $P(y) \vee (F(B) = y)$ and $Q(p, F(p))$, then we can add the clause $P(y) \vee Q(B, y)$.

AI(0270)-6.59

Efficiency issues

- We have a complete inference procedure for FOL, even with equality. It is time to talk about efficiency.
- The easiest implementation: just try everything in rounds, as we have done so far. Each round add new clauses, which are unified with the goal to see whether we have succeeded.
- This is data-driven, so it is **unguided** and will end up with a lot of **unrelated** clauses. Unluckily, we have no option of "backward chaining" this time to make it focused.
 - We are doing refutation, so the goal (empty clause) gives us no clue where to start searching.
- So resolution is typically slower than chaining. If we only need the power of Horn clauses, we won't use resolution.
- But then, like other types of searching, there are rooms for **heuristics** that can keep the search more focused.

AI(0270)-6.60

Unit Preference

- One very simple observation is that **the final sentence contains no terms at all**.
- To **reduce** the number of terms in a sentence, one has to **combine with an atomic sentence**, i.e., those which has only one term.
- So it seems to be a good strategy to try all applications of these sentences **before all other sentences**.
- This is called the **unit preference**: we prefer combining with sentences with only one term (unit sentence).
- It can be viewed as a special case of a general strategy to **prefer shorter sentences**—an A^* search using the number of terms as the heuristic function.
- But branching factor is still large. We need some strategy to **make it unnecessary to generate some sentences**.

AI(0270)-6.61

Set of Support

To get something that looks like backward chaining, we observe that **not all sentences have equal chance of getting contractions**.

- The original KB can be assumed to be **not contradicting**. Then...
- **The negated query must be a sentence** combined using resolution.
- By **searching only these sentences**, the search space is significantly reduced.
 - We keep **two separate sets** of sentences. One set, the **set of support**, is the **newly created sentences**. The other known facts in the KB are known as **axioms**.
 - Whenever applying resolution, **the first one always come from the set of support**. The second sentence may be in either set.
 - Now the search is quite similar to backward chaining, esp. if query is short.
- More importantly, the strategy is still **complete**.

AI(0270)-6.62

Example

Suppose we have the following KB:

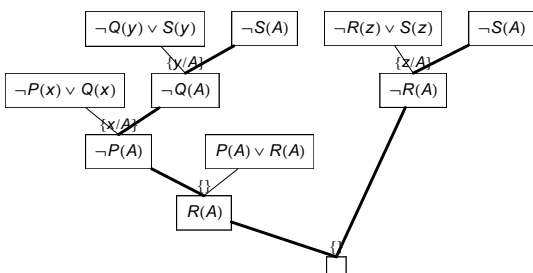
- $P(A) \vee R(A)$
- $\neg P(x) \vee Q(x)$
- $\neg Q(y) \vee S(y)$
- $\neg R(z) \vee S(z)$

and wants to proof $S(A)$.

AI(0270)-6.63

Example

Limiting to resolve with set of support (initially $\neg S(A)$), we get something like this:



The thick line show the main "trunk" of the proof: it contains the new sentences created during the search.

AI(0270)-6.64

Subsumption

- The set of support algorithm works quite well at the beginning, **when the set of support contains only a few sentences**.
- But when we proves more and more sentences and add them into the set of support, the branching factor escalates quickly.
- One improvement: **remove generated sentences** if they are not useful. E.g., **subsumption**—remove a specialized sentence from KB altogether if a more general sentence is known.
- Some examples:
 - If we know $P(x)$, remove $P(A)$.
 - If we know $Q(x)$ or $\neg P(x)$, remove $\neg P(x) \vee Q(x)$.

AI(0270)-6.65

Input Resolution

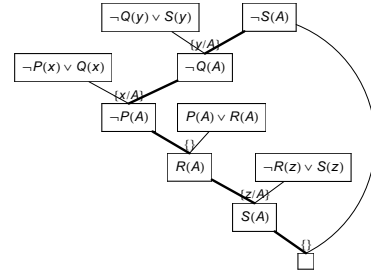
- With set of support, we can treat that each clause currently in the KB as an **operator** for resolution, each clause in the set of support is a **state**. The starting state is the negated query, the goal is the empty clause.
- This adds exponentially many operators during the search, which slow things down quickly. One way to avoid this: **don't use anything in the set of support as an operator**.
So all resolution involves exactly one sentence in the set of support.
- But the important question is: **does this preserve completeness?**
- Answer: **no!** I.e., some true sentences becomes unprovable. E.g., fails for our set of sentences.
- For restricted set of sentences this can be complete. E.g., for Horn clause it is complete.
But then, for Horn clauses we always have chaining.

AI(0270)6.66

Linear Resolution: Restoring completeness

One modification can restore completeness: **linear resolution**:

Also allow application of resolution of the current sentence α **with any ancestor sentence of α** . Our case:



This is at the cost of increasing the depth of the proof.

AI(0270)6.67