

You can't have it all

Lecture 10

Inference in FOL

In the last lecture we see how FOL can enrich our Knowledge Represented so we can express many facts easily.

But we also see that naively applying inference rules would lead to very inefficient algorithms. We will see how to improve the situation in this lecture.

Reference:

- Textbook chapter 9: Sections 9.3–9.4
- Textbook chapter 10: Sections 10.2

AI(0270)

AI(0270)-10.1

We want an inference procedure which is...

- **Sound:** it only gives us things which entailed from the KB.
- **Complete:** if a sentence is entailed by a KB, it can be inferred.
- **Efficient:** it can quickly tell us a sentence is entailed.

First, a bad news: **inference in First-Order logic is semi-decidable.**

- We have a procedure to systematically deduce $KB \models s$
- But we have **no procedure** to systematically deduce $KB \not\models s!$
Similar to the "halting problem".
- In other words, the former procedure can take **any amount of time.**

So **we must make trade-off:** either drop complete or efficient.

Problem of our previous inference procedure

1. The **same knowledge** can be expressed in many many ways, making a huge branching factor.
E.g., why all the conjunctions? The KB is a big conjunction anyway!
2. Our knowledge is in **arbitrary form**, forcing us to use rules that generate **weaker** knowledge.
E.g., why we need Or-introduction or Existential-introduction at all? Because one of our knowledge might be $(P(A) \vee Q(A)) \Rightarrow R(A)$, and we only have $P(A)$.
3. **Universal elimination** is done **blindly**, and we don't know what object should we instantiate variables into.
E.g., we not only try $Missile(M1) \Rightarrow Weapon(M1)$, but also $Missile(Nono) \Rightarrow Weapon(Nono)$ and $Missile(West) \Rightarrow Weapon(West)$, which are basically useless.

AI(0270)-10.2

AI(0270)-10.3

The strategy

- **Pre-process** all sentences into a **canonical form before they enter our KB.** Most of the "trivial" rule are applied here.
- Once in KB, we **only use one or two inference rules.** We assume that all the other inference rules, if needed, are already applied during the pre-processing phase.
- The inference rule will do **more work** at a single shot, and should guarantee that:
 - The new knowledge that it generates is **also in canonical form.**
 - **Only non-trivial new sentences** are generated.
- The inference rule is responsible for **choosing objects for universal elimination.** We will **never directly apply universal elimination** without other knowledge.

The big picture

Before going any deeper, let's have an overview of what we will do.

- We will use a standard form which **only allows disjunction and negation** in sentences—either **Horn form** or **Conjunctive Normal Form.**
Any knowledge can be expressed in CNF, while Horn form is more restricted and can only express certain knowledge.
- Sentences can have variables, but must be **universally quantified.**
- We have two different procedure for inference: **chaining** and **resolution.** Resolution is **complete**, but sorely inefficient. Chaining is a bit more efficient, but work only on **Horn form** sentences.
- Chaining uses **Generalized Modus Ponens**, while resolution uses **generalized resolution**, as their **only** inference rule.
- The inference rule is responsible for **instantiating variables**, using a procedure called **unification.**

AI(0270)-10.4

Conjunctive Normal Form

A FOL sentence is in conjunctive normal form (CNF) if:

- The sentence either has no quantifiers, or has **only universal quantifiers** (i.e., no existential quantifier).
- **All** quantifiers are **all at the left** of the sentence.
For these two reasons, we usually don't write them out.
- Apart from the quantifications, the sentence is either a term or a **disjunction of terms.**
Why "conjunctive normal form" have disjunctions rather than conjunctions? We consider all sentences, or "clauses", to be in a big conjunction.
- Each term is either a **predicate** or the **negation** of it. No two terms in the same sentence is the same or the negation of each other.
- A predicate can contain object constants and variables introduced in the quantifier.

AI(0270)-10.5

Example violations

To begin, let's see some non-CNF sentence and their corresponding CNF.

- $\exists x \text{ Missile}(x)$ —don't use \exists . CNF: $\text{Missile}(M1)$.
- $\forall x \text{ Student}(x) \Rightarrow \forall y \text{ Exam}(y) \Rightarrow \text{Hate}(x, y)$ —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)-10.6

Normalization

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

AI(0270)-10.7

Skolemization

- How to **remove "∃"**? Example: $\exists x \text{ Missile}(x)$ becomes $\text{Missile}(M1)$, where M1 is a new constant ("skolem constant").
- But how the following differs?
 $\forall x \exists y \text{ Loves}(x, y)$
 $\exists y \forall x \text{ Loves}(x, y)$
Which one is the same as $\forall x \text{ Loves}(x, \text{BigLover})$?
- Answer**: the second. For the first, each x can use a **different value** for y , so the CNF above doesn't work. How to convert the first then?
- We introduce a new **function**: skolem function, so that 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)-10.8

Example

Let's try converting this:

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

- 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.

AI(0270)-10.9

Implication Normal Form

It is now clear that any FOL sentence can be written in CNF. But the resulting sentence is usually not easy to read.

It is usually easier to read if we write CNF in a form in which **all negated terms are converted to a conjunction** on the left as **premise**.

We call such a form the **Implication normal form**. E.g.,

- $\text{True} \Rightarrow \text{Missile}(M1)$
- $(\text{Student}(x) \wedge \text{Exam}(y)) \Rightarrow \text{Hate}(x, y)$
- $(\text{OK}(x) \wedge \text{Breezy}(x)) \Rightarrow \text{False}$
- $(\text{OK}(x) \wedge \text{Smelly}(x)) \Rightarrow \text{False}$
- $\text{Breezy}(x) \Rightarrow \text{Neighbour}(x, F(x))$
- $\text{Breezy}(x) \Rightarrow \text{Pit}(F(x))$

It is better to treat them as "reading form", implementation still uses CNF.

AI(0270)-10.10

Representation

How the KB is **implemented**?

- Canonical forms** actually makes it a lot easier: we can treat **sentences** as two **arrays of Predicates**, one for the positive terms, one for the negative ones.
- Each **predicate** is an **array** with the first element being the predicate name and the remaining elements being the **terms**.
- Terms** might be an array as well: we might have functions. But it may also be a simple symbol, e.g., a variable, a constant, etc.
- The whole KB is thus organized as a **list of such sentences in canonical form**.
- We will also need some data structures to make search faster.
More about that in the next lecture.

AI(0270)-10.11

Horn form and Generalized Modus Ponens

- A CNF sentence is in **Horn form** if there is **only one positive term**.
- All the clauses we see in the previous page are Horn forms. A negative example: $OK(x) \vee Pit(x) \vee Wumpus(x)$
- Recall that Modus Ponens is something like:

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

- We will generalize it so that it looks (somewhat) like this:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n \Rightarrow \beta, \alpha_1, \alpha_2, \dots, \alpha_n}{\beta}$$

- If we only use Modus Ponens, we can only deal with Horn clauses. This is exactly what we will do in **chaining**.

AI(0270)-10.12

Unification

But wait... we are talking about **predicates**, not **propositions**. It would be **very restricting** if we can't have variables in our inference rules!
As we said, nothing else do variable instantiation.

- All applications of rules require that two sentences have a **predicate in common**, or being **exact reverse of each other**.
- So the **reason for variable substitution** is to **make two originally different predicates to look exactly the same**, so that a rule can be applied.
- We call such process **unification**, and use the notion $UNIFY(P_1, P_2)$ to represent it. The result is a **substitution**, to be applied on inference rules.
- We assume there is **no clashing variable names** before unify.
- E.g.: $UNIFY(Knows(x, Jan), Knows(John, y)) = \{x/John, y/Jan\}$

AI(0270)-10.13

Most General Unifier (mgu)

- The result of unification is called a **unifier**.
- There can be **many unifiers** for the same unification, and the most general one is called the **most general unifier** (mgu). $UNIFY$ returns the mgu.
- E.g.:

$$UNIFY(Knows(x, y), Knows(John, z)) = \{x/John, y/z\}$$

$$(not \{x/John, y/Jan, z/Jan\})$$

- To do unification, we just need to **check that the predicates are in the same form** (both positive or negative, same predicate name), **unify each term** within the predicates, and make sure the same variable is not binded to two different things.
- But it can get complicated **with functions**.

AI(0270)-10.14

More examples

- $UNIFY(P(x, y, x), P(z, r, q))$: okay, can unify. (Unifier?)
- $UNIFY(P(x), \neg P(x))$: fail.
- $UNIFY(P(x), Q(x))$: fail.
- $UNIFY(P(x), P(F(a)))$: okay, can unify.
- $UNIFY(P(x, y, x), P(Z, r, T))$: fail. $UNIFY$ doesn't deal with equality, so we won't try to prove $Z = T$
- $UNIFY(P(x, F(x)), P(Alice, y))$: Okay, can unify.
- $UNIFY(P(F(x), F(x)), P(F(y), F(Alice)))$: Okay, can unify.
- $UNIFY(P(F(x), x), P(y, y))$: fail! We first make y unify to F(x), but then try to make y unify to x. This requires unifying x to F(x), which fails.

AI(0270)-10.15

Making unifications in sequence

We will need to be able to make unifications in sequence. E.g., we might want to make $P(z, y)$ and $P(x, x)$ unify, and then $H(B)$ and $H(x)$ to unify.

To do so, we **do unification in sequence**, adding bindings to the substitution one by one. E.g., first unification gives

$$\{z/x, y/x\}$$

We give this substitution to the second unification, so we get

$$\{z/x, y/x, x/B\}$$

So we can imagine that we **always give $UNIFY$ a substitution to add new bindings into**. At the beginning, we give it an empty substitution.

AI(0270)-10.16

Implementation

The procedure just needs to **iteratively do unification for each term**, and combine the results. The variable unification is the only non-trivial part.

```
// s is the unifier to be built
void UnifyVariable(Variable v, Term t, Unifier &s) {
    if ({v, t} is in s)
        Unify(t, t1, s);
    else if ({t, t1} is in s) // and t is a variable
        UnifyVariable(v, t1, s);
    else if (v occurs in t)
        fail;
    else {
        Add {v, t} to s;
        // This can end up with something like {v/A(x), x/y}
        // But it is okay if we use the resulting unifier in sequence
    }
}
```

AI(0270)-10.17

Generalized Modus Ponens

Finally we are ready to present the rule that can be used for Horn forms:

Generalized Modus Ponens: suppose θ is the mgu for unifying predicates p_1 and p'_1, p_2 and p'_2, \dots, p_n and p'_n . Let q be another predicate. Then

$$\frac{p'_1, p'_2, \dots, p'_n, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

- The use of Generalized Modus Ponens **always gives us a new sentence containing exactly one predicate.**
- The step is big: we do universal eliminations (via unifications), and-introduction and modus ponens **in a single step.**
- More importantly, **it only works on the mgu**, not random variable assignments.

AI(0270)-10.18

Example

Let's redo our example about the criminal called West.

Knowledge in INF form

1. $American(x) \wedge Weapon(y) \wedge Nation(z) \wedge Hostile(z) \wedge Sells(x, z, y) \Rightarrow Criminal(x)$
2. $Hostile(Nono)$
3. $Nation(Nono)$
4. $Missile(M1)$
5. $Own(Nono, M1)$
6. $Missile(x) \wedge Own(Nono, x) \Rightarrow Sell(West, Nono, x)$
7. $American(West)$
8. $Missile(x) \Rightarrow Weapon(x)$

AI(0270)-10.19

Example (Cont'd)

Modus Ponens on (6) using (4) and (5): (unifier $\{x/M1\}$)

9. $Sell(West, Nono, M1)$

Modus Ponens on (8) using (4): (unifier $\{x/M1\}$)

10. $Weapon(M1)$

Modus Ponens on (1) using (7), (10), (3) and (2):
(unifier $\{x/West, y/M1, z/Nono\}$)

11. $Criminal(West)$

It consists of 3 steps, so the **depth** of search is 3.

How about the **branching factor**? That depends on the number of applicable rules for Modus Ponens (3 here), and how many predicate for us to find unifications (not many).

AI(0270)-10.20

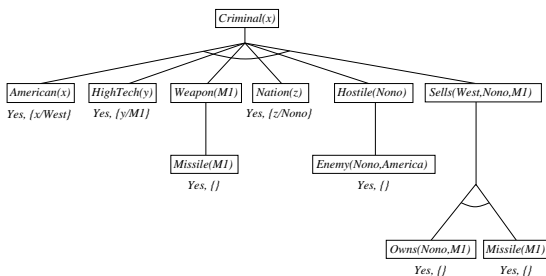
Backward chaining: the natural form of inference

- But the system will get slow quickly if we have more unrelated rules: **the system try to generate all unrelated knowledge.**
- How to **make sure that only needed Modus Ponens are tried?**
- Note that the use of modus ponens can only generate sentences that contains a **single positive predicate.**
It is a bit more flexible than it seems: we can have quantifiers, because they will go away during normalization. We can even have conjunction, by just inserting one more rule. But negations and disjunctions are out of reach.
- If the target is always a single known predicate, how about searching it **backwards**, i.e., from the goal?
- This is the basis of **backward chaining**: from the needed predicate, we find what rule can prove it, and then generate subgoal.
This match human behaviour quite well.

AI(0270)-10.21

Example

A successful deviation:



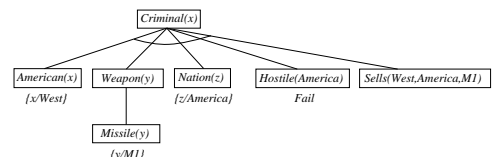
We start with the rule that makes "Criminal", which requires 5 conjuncts. The subgoals are to find bindings for each of them.

We don't really have the "Enemy" part: we have simplify the example a bit.

AI(0270)-10.22

A failed example

When we try the unifications one by one, we might end up being unable to do unification for a sub-goal. Then we fail and have to backtrack. E.g.,



We have made the binding $\{z/America\}$, so $Hostile(z)$ now becomes $Hostile(America)$. But there is no clause or conclusion that unify to $Hostile(America)$, so back-chaining **fails** and **back-track**.

AI(0270)-10.23

Reading the code of the book

The book contains the following pseudo-code which is quite hard to understand. But it is difficult to do better than the book.

```

function BACK-CHAIN(KB, q) returns a set of substitutions
  BACK-CHAIN-LIST(KB, {q}, {})

function BACK-CHAIN-LIST(KB, qlist, θ) returns a set of substitutions
  inputs: KB, a knowledge base
         qlist, a list of conjuncts forming a query (θ already applied)
         θ, the current substitution
  static: answers, a set of substitutions, initially empty

  if qlist is empty then return {θ}
  q ← FIRST(qlist)
  for each qi in KB such that θi ← UNIFY(q, qi) succeeds do
    Add COMPOSE(θ, θi) to answers
  end
  for each sentence (p1 ∧ ... ∧ pn ⇒ qi) in KB such that θi ← UNIFY(q, qi) succeeds do
    answers ← BACK-CHAIN-LIST(KB, SUBST(θi, {p1 ... pn}), COMPOSE(θ, θi)) ∪ answers
  end
  return the union of BACK-CHAIN-LIST(KB, REST(qlist), θ) for each θ ∈ answers
  
```

AI(0270)-10.24

Understanding it...

For us programmer, code is the most concrete words. Let's follow it.

- The BACK-CHAIN function calls the recursive BACK-CHAIN-LIST function, giving it an **initial qlist**: the **predicate we query** about.
- BACK-CHAIN-LIST receives *qlist*, a **list of predicates we need to prove**, and **θ**, a **substitution that we already committed in**.
- It's aim is to return a **set of substitution** that will prove **all predicates of qlist**
- If there is no more predicate in *qlist* for us to prove, just return the substitutions that we have already made.
- Otherwise, find the **first predicate** (subgoal) to prove, say *q*.

AI(0270)-10.25

Understand it... (cont'd)

- Look into KB to find everything that either:
 - **directly** tell us *q* under some unification. Try to compose the unifier with the substitution so far.
 - is a **rule** with the **conclusion** *q* under some unification. Recursively call BACK-CHAIN-LIST to complete the unification needed, and compose the resulting unifier with the substitution so far.
- for each of the combined substitutions found above:
 - Recursively call BACK-CHAIN-LIST to prove the **remaining predicates in qlist**.
- Return **the set of all substitutions** returned in the recursive calls.

AI(0270)-10.26

Practical concern

Consider what would happen for the following:

```

KB: P(x) ⇒ P(F(x))
    P(A)
Query: ∃x P(x)
    Note: inserted as P(x), not P(X1)!
Result: Infinitely many solutions x
    /A, P(A), P(P(A)), ...!
  
```

```

KB: P(x) ⇒ Q(x)
    Q(x) ⇒ P(x)
    P(A)
Query: ∃x Q(x)
Result: Infinite loop finding P(A)
    repeatedly!
  
```

Thus we must **limit search depth** (perhaps using iterative deepening), and have to **stop after some number of steps**—just like when we search in the game tree.

Why we need to insert P(x) instead of P(X1)? We will see it soon.

But a query P(A) can still generate infinite loop.

AI(0270)-10.27