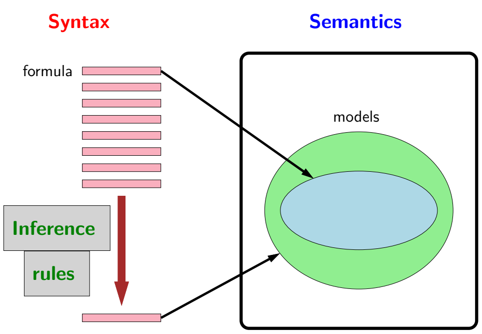
Logic: first-order modus ponens

First-order logic



- Now we look at inference rules which can make first-order inference much more efficient. The key is to do everything implicitly and avoid propositionalization; again the whole spirit of logic is to do things compactly and implicitly.

Definite clauses

$$\forall x\,\forall y\,\forall z\,(\mathsf{Takes}(x,y) \wedge \mathsf{Covers}(y,z)) \rightarrow \mathsf{Knows}(x,z)$$

Note: if propositionalize, get one formula for each value to (x,y,z), e.g., $(\mathsf{alice}, \mathsf{cs221}, \mathsf{mdp})$

Definition: definite clause (first-order logic)

A definite clause has the following form:

$$\forall x_1 \cdots \forall x_n\,(a_1 \wedge \cdots \wedge a_k) \rightarrow b$$

for variables $x_1, \dots, x_n$ and atomic formulas $a_1, \dots, a_k, b$ (which contain those variables).

- Like our development of inference in propositional logic, we will first talk about first-order logic restricted to Horn clauses, in which case a first-order version of modus ponens will be sound and complete. After that, we'll see how resolution allows to handle all of first-order logic.
- We start by generalizing definite clauses from propositional logic to first-order logic. The only difference is that we now have universal quantifiers sitting at the beginning of the definite clause. This makes sense since universal quantification is associated with implication, and one can check that if one propositionalizes a first-order definite clause, one obtains a set (conjunction) of multiple propositional definite clauses.

Modus ponens (first attempt)

Definition: modus ponens (first-order logic)

$$\frac{a_1, \dots, a_k \quad \forall x_1 \cdots \forall x_n (a_1 \wedge \cdots \wedge a_k) \rightarrow b}{b}$$

Setup:

Given $P(\text{alice})$ and $\forall x\, P(x) \rightarrow Q(x)$.

Problem:

Can't infer $Q(\text{alice})$ because $P(x)$ and $P(\text{alice})$ don't match!

Solution: substitution and unification

- If we try to write down the modus ponens rule, we would fail.
- As a simple example, suppose we are given $P(\text{alice})$ and $\forall x\, P(x) \rightarrow Q(x)$. We would naturally want to derive $Q(\text{alice})$. But notice that we can't apply modus ponens because $P(\text{alice})$ and $P(x)$ don't match!
- Recall that we're in syntax-land, which means that these formulas are just symbols. Inference rules don't have access to the semantics of the constants and variables — it is just a pattern matcher. So we have to be very methodical.
- To develop a mechanism to match variables and constants, we will introduce two concepts, substitution and unification for this purpose.

Substitution

$\text{Subst}[\{x/\text{alice}\}, P(x)] = P(\text{alice})$

$\text{Subst}[\{x/\text{alice}, y/z\}, P(x) \wedge K(x, y)] = P(\text{alice}) \wedge K(\text{alice}, z)$

Definition: Substitution

A substitution θ is a mapping from variables to terms.

$\text{Subst}[\theta, f]$ returns the result of performing substitution θ on f.

- The first step is substitution, which applies a search-and-replace operation on a formula or term.
- We won't define $\text{Subst}[\theta, f]$ formally, but from the examples, it should be clear what Subst does.
- Technical note: if θ contains variable substitutions x/alice we only apply the substitution to the free variables in f, which are the variables not bound by quantification (e.g., x in $\exists y, P(x, y)$). Later, we'll see how CNF formulas allow us to remove all the quantifiers.

Unification

$\text{Unify}[\text{Knows}(\text{alice}, \text{arithmetic}), \text{Knows}(x, \text{arithmetic})] = \{x/\text{alice}\}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(x, z)] = \{x/\text{alice}, y/z\}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(\text{bob}, z)] = \text{fail}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(x, F(x))] = \{x/\text{alice}, y/F(\text{alice})\}$

Definition: Unification

Unification takes two formulas f and g and returns a substitution θ which is the most general unifier:

$\text{Unify}[f, g] = \theta$ such that $\text{Subst}[\theta, f] = \text{Subst}[\theta, g]$

or "fail" if no such θ exists.

- Substitution can be used to make two formulas identical, and unification is the way to find the least committal substitution we can find to achieve this.
- Unification, like substitution, can be implemented recursively. The implementation details are not the most exciting, but it's useful to get some intuition from the examples.

Modus ponens

Definition: modus ponens (first-order logic)

$$\frac{a_1', \dots, a_k' \quad \forall x_1 \cdots \forall x_n (a_1 \wedge \cdots \wedge a_k) \rightarrow b}{b'}$$

Get most general unifier θ on premises:

- $\theta = \text{Unify}[a_1' \wedge \cdots \wedge a_k', a_1 \wedge \cdots \wedge a_k]$

Apply θ to conclusion:

- $\text{Subst}[\theta, b] = b'$

- Having defined substitution and unification, we are in position to finally define the modus ponens rule for first-order logic. Instead of performing a exact match, we instead perform a unification, which generates a substitution θ. Using θ, we can generate the conclusion b' on the fly.
- Note the significance here: the rule $a_1 \wedge \cdots \wedge a_k \rightarrow b$ can be used in a myriad ways, but Unify identifies the appropriate substitution, so that it can be applied to the conclusion.

Modus ponens example

Example: modus ponens in first-order logic

Premises:

$\text{Takes}(\text{alice}, \text{cs221})$

$\text{Covers}(\text{cs221}, \text{mdp})$

$\forall x \, \forall y \, \forall z \, \text{Takes}(x, y) \wedge \text{Covers}(y, z) \rightarrow \text{Knows}(x, z)$

Conclusion:

$\theta = \{x/\text{alice}, y/\text{cs221}, z/\text{mdp}\}$

Derive $\text{Knows}(\text{alice}, \text{mdp})$

- Here's a simple example of modus ponens in action. We bind x, y, z to appropriate objects (constant symbols), which is used to generate the conclusion $\text{Knows}(\text{alice}, \text{mdp})$.

Complexity

$$\forall x \, \forall y \, \forall z \, P(x, y, z)$$

- Each application of Modus ponens produces an atomic formula.
- If no function symbols, number of atomic formulas is at most

$$(\text{num-constant-symbols})^{(\text{maximum-predicate-arity})}$$

- If there are function symbols (e.g., F), then infinite...

$$Q(a) \quad Q(F(a)) \quad Q(F(F(a))) \quad Q(F(F(F(a)))) \quad \cdots$$

- In propositional logic, modus ponens was considered efficient, since in the worst case, we generate each propositional symbol.
- In first-order logic, though, we typically have many more atomic formulas in place of propositional symbols, which leads to a potentially exponentially number of atomic formulas, or worse, with function symbols, there might be an infinite set of atomic formulas.

Complexity

Theorem: completeness

Modus ponens is complete for first-order logic with only Horn clauses.

Theorem: semi-decidability

First-order logic (even restricted to only Horn clauses) is **semi-decidable**.

- If KB $\models f$, forward inference on complete inference rules will prove f in finite time.
- If KB $\not\models f$, no algorithm can show this in finite time.

- We can show that modus ponens is complete with respect to Horn clauses, which means that every true formula has an actual finite derivation.
- However, this doesn't mean that we can just run modus ponens and be done with it, for first-order logic even restricted to Horn clauses is semi-decidable, which means that if a formula is entailed, then we will be able to derive it, but if it is not entailed, then we don't even know when to stop the algorithm — quite troubling!
- With propositional logic, there were a finite number of propositional symbols, but now the number of atomic formulas can be infinite (the culprit is function symbols).
- Though we have hit a theoretical barrier, life goes on and we can still run modus ponens inference to get a one-sided answer. Next, we will move to working with full first-order logic.