

1 Administrivia

As a reminder, I'm out of town Wed-Fri of this week. My office hours for tomorrow are cancelled, and Prof. Luger will be covering class on Thurs. He asks that you all bring your textbooks so that he can point to code examples in the book.

Reading for this section is Ch. 2 and Ch. 12.

2 Unification and Resolution

- Last time we talked about the syntax and semantics of FOPL. Now we'll talk about how you *do* things with it. Specifically, how you prove theorems in/with it.
- So let's look at how we might actually go about proving things given a bunch of statements (sentences) in FOPL.
- Much like our state space searches previously, we'll start with an initial setting and extend it by applying "successor" functions.
- In this case, our "successors" are called **inference rules** (also, **production rules** or **rewrite rules**).
- You've already seen an inference rule before – DeMorgan's law: $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$
- This says that any time we see something of the form " $\neg(P \wedge Q)$ ", we can rewrite it to the other form.
- You can imagine that if you have a pile of such rewrite rules, you can use them to rewrite all your axioms in sequence until you find one that matches your goal.
- Example: probably the oldest inference rule that you're all familiar with is **modus ponens** (Latin, lit. "mood that affirms"):

$$\frac{P \quad P \rightarrow Q}{Q}$$

- This rule allows is what allows us to rewrite $\forall X \text{ man}(X) \rightarrow \text{mortal}(X)$ plus $\text{man}(\text{socrates})$ into $\text{mortal}(\text{socrates})$
- Chapter 2 of your book gives a whole pile of such inference rules. Over history, philosophers and logicians have identified hundreds of different inference rules.
- It turns out, though, that most of those rewrite rules are equivalent, or you can derive some of them from others of them.
- We'd *like* to use the smallest set of inference rules that we can.

- So the first thing we have to ask is, by what criteria do we judge that a set of inference rules is acceptable?
- There are 2 properties that we normally identify:
 - **Complete:** every theorem that is *true* given your axioms can be *derived* from them with the rewrite rules.
 - **Sound:** you can't derive anything *false* using your rewrite rules (and given a consistent set of axioms, that is).
- So... How many inference rules do we *need* to be both complete and sound for FOPL?
- It turns out that the answer is 1! Somewhat surprisingly, the single rule, **resolution**, can be used to do inference in FOPL (with some minor caveats).
- Resolution is at the heart of most modern automated theorem proving systems and is built-in to Prolog.
- We'll talk about resolution in a bit. For the moment, we'll need the tool **unification** to get there.
- The fundamental problem is that in order to apply a rewrite rule, we need to know when two things are the "same".
- E.g., in going from $\text{man}(X) \rightarrow \text{mortal}(X)$ and $\text{man}(\text{socrates})$ to $\text{mortal}(\text{socrates})$ (via modus ponens), you need to perceive that X can be replaced with socrates without messing anything up.
- Other examples: are $\text{plus}(3, 2)$ and $\text{plus}(3, X)$ the same? What about $\text{likes}(X, \text{rabbit})$ and $\text{likes}(\text{sam}, Y)$? What about $\text{plus}(3, 2)$ and $\text{plus}(2, 3)$?
- Aha! The last two are *not* actually the same! They are *mathematically* equivalent, but we're worried about *syntactic* equivalence at the moment.
- Unification is simply a *string matching and replacement* operation. It looks at the string of a FOPL formula and replaces the strings denoting variables with the strings denoting other expressions or constants.
- Example: unifying $\text{likes}(\text{sam}, X)$ with $\text{likes}(Y, \text{rabbit})$. The two expressions are equivalent when you match Y to sam and X to rabbit . These are called **variable bindings**.
- In this case, written $\{\text{sam}/Y, \text{rabbit}/X\}$
- Note that you can bind in *either* direction.
- Also, a variable can bind to an arbitrary *term* – that is, to an arbitrary function of other constants or vars, etc.
- For example, $\text{likes}(X, \text{sister}(\text{dogsitter}(\text{barber}(\text{uncle}(\text{alice}))))$ unifies with $\text{likes}(\text{bob}, Y)$ to produce $\{\text{bob}/X, \text{dogsitter}(\text{barber}(\text{uncle}(\text{alice}))) / Y\}$.

- Also, variables can bind to variables: `plus(3, X)` and `plus(3, Y)` unify to $\{X/Y\}$.
- **Note!** Potential gotcha: you can't bind a variable to an expression containing itself! I.e., if you try to unify `X` with `p(X)` you could get an infinite recursion.
- The check to avoid this is called the *occurs check*.
- Also, you can only unify a universally quantified variable. This should make sense – if it's universally quantified, that says you can plug in whatever you like. If it's *existentially* quantified, you have to look for the specific thing that could go there.
- Finally, you have to make sure that you always bind the same variable the same way.
- When you unify `man(X) → mortal(X)` with `man(socrates)`, you have to make sure that `X` gets `socrates` *both* places. In general, once you have bound a variable, you have to propagate that binding throughout the current unification. The algorithm in your book handles this – you should make an effort to understand what it's doing...
- You can also **compose** two unifications – if you have produced $\{Y/X\}$ and $\{a/Y\}$ then you can compose those to yield $\{a/X\}$.
- In general, there are 3 results you can get from unification:
 1. Unification succeeded with no bindings (`plus(3, 2)` unifies with `plus(3, 2)` with no variable bindings).
 2. Unification succeeded with bindings
 3. Unification failed.
- There's also the notion of the **most general unifier** (MGU). Basically, this says that you should *not* choose a binding that restricts things more than normal. E.g., if you have `likes(X, rabbit)` and `likes(Y, rabbit)`, then you could choose the binding $\{X/Y\}$ or $\{Y/X\}$, but *not* $\{sam/X\}$ because that restricts you more than you need to. (This is also handled automatically by the unification function in the text.)
- Ok, now that we know when two expressions *match*, how do we use that info in proving things?
- Resolution theorem proving is essentially a proof by contradiction: you assert the opposite of what you're trying to prove and then see if you can find a contradiction.
- Remember that we've already concluded that $P \wedge \neg P$ is a contradiction (always false). So if you want to prove P , you can do so by asserting $\neg P$ and proceeding until you can find $P \wedge \neg P$.
- The problem is that it can be hard to recognize P . You have to deal with unification (recognize that $P(X)$ and $\neg P(f oo)$ can be unified under $f oo/X$), and you have to handle the fact that P itself might be part of or entailed by a larger expression.

- The resolution rule is that:

$$(a_1 \vee a_2 \vee \dots \vee a_n) \wedge (\neg a_1 \vee b_1 \vee \dots \vee b_m)$$

is equivalent to:

$$(a_2 \vee \dots \vee a_n \vee b_1 \vee b_2 \vee \dots \vee b_m)$$

- **Example:** if you know $\neg \text{dog}(X) \vee \text{animal}(X)$ and you know $\neg \text{animal}(Y) \vee \text{die}(Y)$ then you can conclude $\neg \text{dog}(Y) \vee \text{die}(Y)$
- That is, if we have all our knowledge in the form of a conjunction of disjuncts, then we can proceed to join terms together, eliminating opposites as we go, until we eventually we can find two statements that are each single terms and we can eliminate them.
- When we finally resolve two statements away to nothing, we have found the contradiction and have proved what we set out to prove.
- The problem is that we don't necessarily *have* things in a conjunct-of-disjunct form. Instead, we have messy things with implications, ANDs, ORs, NOTs, FORALLs, EXISTSs, etc.
- The first step of resolution theorem proving, therefore, is to convert each such statement into **clause form**.
- **Def:** a **clause** is a disjunction of literals (i.e., possibly negated predicates).
- There is a 9-step procedure for converting sentences to clause form. It's detailed in your book, but we'll go through it here.
 1. Eliminate implication. You can always change $P \rightarrow Q$ into $\neg P \vee Q$
 2. Move \neg inward. You can use DeMorgan's law to move \neg in through parens, and you know that $\neg \neg P$ is just P . More than that, though, you can convert $\neg \exists X ()$ into $\forall X \neg ()$ and, similarly $\neg \forall X ()$ into $\exists X \neg ()$.
 3. Rename all variables so that each quantifier has its own unique variable name associated with it. (This is so that we don't confuse the X from one quantifier with the X from another in the next step.)
 4. Now that every quantifier has its own name, we can move all quantifiers out to apply to the whole sentence. Note! You can't change the *order* of the quantifiers w.r.t. each other, just their position w.r.t. the sentence.
 5. Now we have to get rid of existential quantifiers. Remember that unification applies *only* to universally quantified variables, so in order to apply it, we need to get rid of the existentials. This is the least intuitive step in this procedure. Basically, we want to replace every existentially quantified var with an appropriate constant. The problem is that we don't *know* what an appropriate constant is. All we know is that there should be *some* constant that will fill that role, depending on how other variables are instantiated. The answer is to introduce a **skolem function**. This is a synthetic (i.e., made up out of whole cloth) function that *stands for* the constant that goes there. This function can, in general, depend on any universally quantified variable within whose scope it falls.

Example:

$$\forall X(\forall Y(\exists Z(X^2 + Y^2 = Z^2)))$$

can be converted into

$$\forall X(\forall Y(X^2 + Y^2 = f(X, Y)))$$

The function $f(X, Y)$ stands for some value of Z that would make that expression true. With the existential variables replaced with functions, we can drop the existential quantifiers altogether.

6. Drop all *universal* quantifiers, as *all* variables are now universally quantified.
 7. Convert to a conjunct of disjuncts. This just requires distributing \wedge over \vee and vice-versa until all of the \vee s are grouped together in clauses separated by \wedge s.
 8. Call each conjunct a separate clause in your KB.
 9. Finally, replace all variables in each clause with unique names. Basically, you just don't want the *same* variable name to occur in multiple different clauses (b/c that would break unification).
- Now you have your sentences (axioms) in a uniform representation and can apply resolution to prove things with them.
 - The *really* hard part, of course, is picking which pair of clauses to resolve at any step. That brings you back to search methods and heuristics...