6.825 Techniques in Artificial Intelligence

Resolution Theorem Proving: Propositional Logic

- Propositional resolution
- Propositional theorem proving
- Unification

Today we're going to talk about resolution, which is a proof strategy. First, we'll look at it in the propositional case, then in the first-order case. It will actually take two lectures to get all the way through this. Then, we'll have you do problem set 2, which involves using the resolution proof technique on a moderately big problem in first-order inference.

Propositional Resolution

At the end of propositional logic, we talked a little bit about proof, what it was, with the idea that you write down some axioms, statements that you're given, and then you try to derive something from them. And we've all had practice doing that in high school geometry and we've talked a little bit about natural deduction. So what we're going to talk about today is resolution. Which is the way that pretty much every modern automated theorem-prover is implemented. It's apparently the best way for computers to think about proving things.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

So here's the Resolution Inference Rule, in the propositional case. It says that if you know “alpha or beta”, and you know “not beta or gamma”, then you're allowed to conclude “alpha or gamma”. OK. Remember from when we looked at inference rules before that these greek letters are meta-variables. They can stand for big chunks of propositional logic, as long as the parts match up in the right way. So if you know something of the form “alpha or beta”, and you also know that “not beta or gamma”, then you can conclude “alpha or gamma”.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Resolution refutation:

It turns out that that one rule is all you need to prove things. At least, to prove that a set of sentences is not satisfiable. So, let's see how this is going to work. There's a proof strategy called Resolution Refutation, with three steps. And it goes like this.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Resolution refutation:
  - Convert all sentences to CNF

First, you convert all of your sentences to conjunctive normal form. You already know how to do this! Then, you write each clause down as a premise or given in your proof.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Resolution refutation:
  - Convert all sentences to CNF
  - Negate the desired conclusion (converted to CNF)

Then, you negate the desired conclusion -- so you have to say what you're trying to prove, but what we're going to do is essentially a proof by contradiction. You've all seen the strategy of proof by contradiction (or, if we're being fancy and Latin, *reductio ad absurdum*). You assert that the thing that you're trying to prove is false, and then you try to derive a contradiction. That's what we're going to do. So you negate the desired conclusion and convert that to CNF. And you add each of these clauses as a premise of your proof, as well.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Resolution refutation:
  - Convert all sentences to CNF
  - Negate the desired conclusion (converted to CNF)
  - Apply resolution rule until either
    - Derive false (a contradiction)
    - Can't apply any more

And then we apply the Resolution Rule until either you can derive "false" -- which means that the conclusion did, in fact, follow from the things that you had assumed, right? If you assert that the negation of the thing that you're interested in is true, and then you prove for a while and you manage to prove false, then you've succeeded in a proof by contradiction of the thing that you were trying to prove in the first place. So you run the resolution rule until you derive false or until you can't apply it anymore.

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Resolution refutation:
  - Convert all sentences to CNF
  - Negate the desired conclusion (converted to CNF)
  - Apply resolution rule until either
    - Derive false (a contradiction)
    - Can't apply any more
- Resolution refutation is sound and complete
  - If we derive a contradiction, then the conclusion follows from the axioms
  - If we can't apply any more, then the conclusion cannot be proved from the axioms.

What if you can't apply the Resolution Rule anymore? Is there anything in particular that you can conclude? In fact, you can conclude that the thing that you were trying to prove can't be proved. So resolution refutation for propositional logic is a complete proof procedure. So if the thing that you're trying to prove is, in fact, entailed by the things that you've assumed, then you can prove it using resolution refutation. In the propositional case. It's more complicated in the first-order case, as we'll see. But in the propositional case, it means that if you've applied the Resolution Rule and you can't apply it anymore, then your desired conclusion can't be proved.

It's guaranteed that you'll always either prove false, or run out of possible steps. It's complete, because it always generates an answer. Furthermore, the process is sound: the answer is always correct.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation

So let's just do a proof. Let's say I'm given “P or Q”, “P implies R” and “Q implies R”. I would like to conclude R from these three axioms. I'll use the word "axiom" just to mean things that are given to me right at the moment.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given

We start by converting this first sentence into conjunctive normal form. We don't actually have to do anything. It's already in the right form.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	$\neg$ P v R	Given

Now, “P implies R” turns into “not P or R”.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	$\neg$ P v R	Given
3	$\neg$ Q v R	Given

Similarly, “Q implies R” turns into “not Q or R

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion

Now we want to add one more thing to our list of given statements. What's it going to be?

Not R. Right? We're going to assert the negation of the thing we're trying to prove. We'd like to prove that R follows from these things. But what we're going to do instead is say not R, and now we're trying to prove false. And if we manage to prove false, then we will have a proof that R is entailed by the assumptions.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion

Now, we'll draw a blue line just to divide the assumptions from the proof steps.

And now, we look for opportunities to apply the resolution rule. You can do it in any order you like (though some orders of application will result in much shorter proofs than others).

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2

We can apply resolution to lines 1 and 2, and get “Q or R” by resolving away P.

Propositional Resolution Example

Prove R

1	P v Q
2	P $\rightarrow$ R
3	Q $\rightarrow$ R

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4

And we can take lines 2 and 4, resolve away R, and get “not P.”

Propositional Resolution Example

Prove R

1	P v Q
2	P $\rightarrow$ R
3	Q $\rightarrow$ R

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4

Similarly, we can take lines 3 and 4, resolve away R, and get “not Q”.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7

By resolving away Q in lines 5 and 7, we get R.

Propositional Resolution Example

Prove R

1	P v Q
2	$P \rightarrow R$
3	$Q \rightarrow R$

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7
9	•	4,8

And finally, resolving away R in lines 4 and 8, we get the empty clause, which is false. We'll often draw this little black box to indicate that we've reached the desired contradiction.

Propositional Resolution Example

Prove R

1	P v Q
2	P → R
3	Q → R

false v R
¬ R v false

~~false v false~~

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7
9	•	4,8

How did I do this last resolution? Let's see how the resolution rule is applied to lines 4 and 8. The way to look at it is that R is really "false or R", and that "not R" is really "not R or false". (Of course, the order of the disjuncts is irrelevant, because disjunction is commutative). So, now we resolve away R, getting "false or false", which is false.

Propositional Resolution Example

Prove R

1	P v Q
2	P → R
3	Q → R

false v R
¬ R v false

~~false v false~~

Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	P	
7	¬ Q	3,4
8	R	5,7
9	•	4,8

One of these steps is unnecessary. Which one? Line 6. It's a perfectly good proof step, but it doesn't contribute to the final conclusion, so we could have omitted it.

The Power of False

Here's a question. Does "P and not P" entail Z?

The Power of False

Prove Z

1	P
2	¬ P

Step	Formula	Derivation

It does, and it's easy to prove using resolution refutation.

The Power of False

Prove Z

1	P
2	¬ P

Step	Formula	Derivation
1	P	Given
2	¬ P	Given
3	¬ Z	Negated conclusion

We start by writing down the assumptions and the negation of the conclusion.

The Power of False

Prove Z

1	P
2	¬ P

Step	Formula	Derivation
1	P	Given
2	¬ P	Given
3	¬ Z	Negated conclusion
4	•	1,2

Then, we can resolve away P in lines 1 and 2, getting a contradiction right away.

The Power of False

Prove Z

1	P
2	¬ P

Step	Formula	Derivation
1	P	Given
2	¬ P	Given
3	¬ Z	Negated conclusion
4	•	1,2

Note that (P Æ ¬ P) → Z is valid

Because we can prove Z from "P and not P" using a sound proof procedure, then "P and not P" entails Z. And, by the theorem relating entailment and validity, we have that the sentence "P and not P implies Z" is valid.

The Power of False

Prove Z

1	P
2	¬ P

Step	Formula	Derivation
1	P	Given
2	¬ P	Given
3	¬ Z	Negated conclusion
4	•	1,2

Note that (P Æ ¬ P) → Z is valid

Any conclusion follows from a contradiction – and so strict logic systems are very brittle.

So, we see, again, that any conclusion follows from a contradiction. This is the property that can make logical systems quite brittle; they're not robust in the face of noise. We'll address this problem when we move to probabilistic inference.

Example Problem

Convert to CNF

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

Here's an example problem. Stop and do the conversion into CNF before you go to the next slide.

Example Problem

Prove R

1	(P → Q) → Q
2	(P → P) → R
3	(R → S) → ¬(S → Q)

Convert to CNF

- ¬(¬ P v Q) v Q
- (P Æ¬ Q) v Q
- (P v Q) Æ(¬ Q v Q)
- (P v Q)

So, the first formula turns into “P or Q”.

Example Problem

Prove R

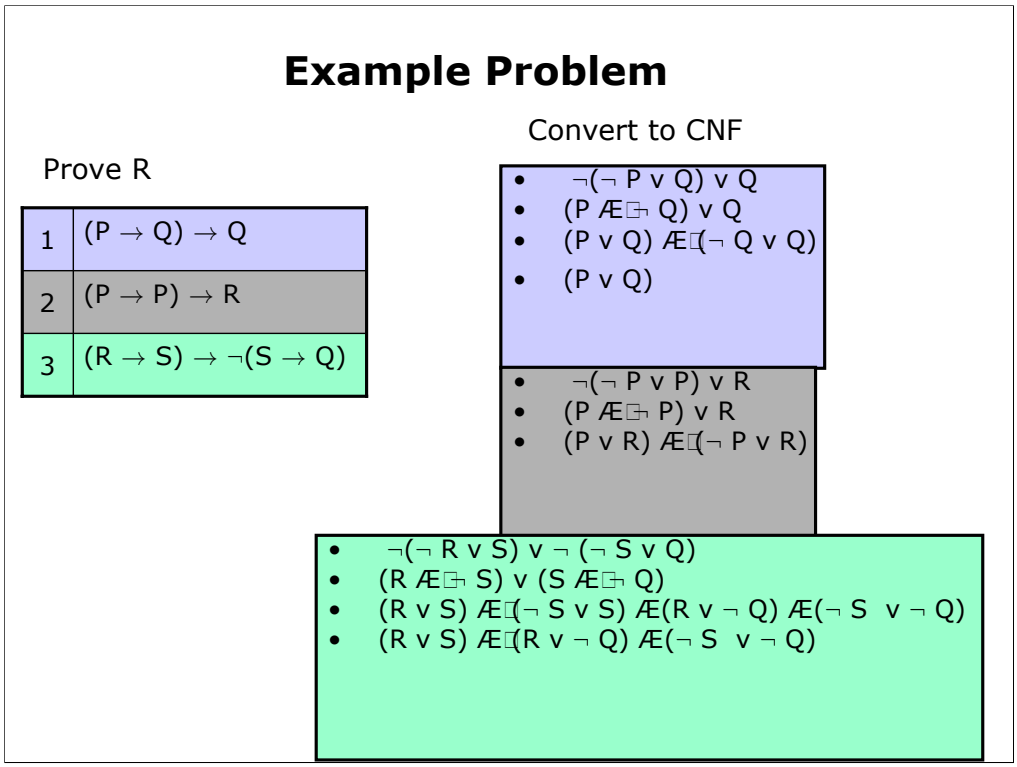
1	(P → Q) → Q
2	(P → P) → R
3	(R → S) → ¬(S → Q)

Convert to CNF

- ¬(¬ P v Q) v Q
- (P Æ¬ Q) v Q
- (P v Q) Æ(¬ Q v Q)
- (P v Q)

- ¬(¬ P v P) v R
- (P Æ¬ P) v R
- (P v R) Æ(¬ P v R)

The second turns into ("P or R" and "not P or R"). We probably should have simplified it into "False or R" at the second step, which reduces just to R. But we'll leave it as is, for now.



Finally, the last formula requires us to do a big expansion, but one of the terms is true and can be left out. So, we get "(R or S) and (R or not Q) and (not S or not Q)".

Resolution Proof Example

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

1	$P \vee Q$	
2	$P \vee R$	
3	$\neg P \vee R$	
4	$R \vee S$	
5	$R \vee \neg Q$	
6	$\neg S \vee \neg Q$	
7	$\neg R$	Neg

Lecture 7 • 32

Now we can almost start the proof. We copy each of the clauses over here, and we add the negation of the query.

Please stop and do this proof yourself before going on.

Resolution Proof Example

Prove R

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

	$P \vee Q$	
	$P \vee R$	
	$\neg P \vee R$	
	$R \vee S$	
	$R \vee \neg Q$	
	$\neg S \vee \neg Q$	
	$\neg R$	Neg
	S	4,7
	$\neg Q$	6,8
	P	1,9
	R	3,10
	•	7,11

Here's a sample proof. It's one of a whole lot of possible proofs.

Proof Strategies

In choosing among all the possible proof steps that you can do at any point, there are two rules of thumb that are really important.

Proof Strategies

- Unit preference: prefer a resolution step involving an unit clause (clause with one literal).
  - Produces a shorter clause – which is good since we are trying to produce a zero-length clause, that is, a contradiction.

The unit preference rule says that if you can involve a clause that has only one literal in it, that's usually a good idea. It's good because you get back a shorter clause. And the shorter a clause is, the closer it is to false.

Proof Strategies

- Unit preference: prefer a resolution step involving an unit clause (clause with one literal).
  - Produces a shorter clause – which is good since we are trying to produce a zero-length clause, that is, a contradiction.
- Set of support: Choose a resolution involving the negated goal or any clause derived from the negated goal.
  - We're trying to produce a contradiction that follows from the negated goal, so these are "relevant" clauses.
  - If a contradiction exists, one can find one using the set-of-support strategy.

The set-of-support rule says you should involve the thing that you're trying to prove. It might be that you can derive conclusions all day long about the solutions to chess games and stuff from the axioms, but once you're trying to prove something about what way to run, it doesn't matter. So, to direct your "thought" processes toward deriving a contradiction, you should always involve a clause that came from the negated goal, or that was produced by the set of support rule. Adhering to the set-of-support rule will still make the resolution refutation process sound and complete.

Recitation Problems

Using resolution refutation, prove the last sentence in each group from the rest of the sentences in the group.

$P \rightarrow Q$
$\neg P \rightarrow R$
$\neg Q \rightarrow \neg R$

$(P \rightarrow Q) \vee (R \rightarrow S)$
$(P \rightarrow S) \vee (R \rightarrow Q)$

$\neg(P \wedge \neg Q) \vee \neg(\neg S \wedge \neg T)$
$\neg(T \vee Q)$
$U \rightarrow (\neg T \rightarrow (\neg S \wedge P))$
$\neg U$

Use resolution refutation to do problem 6.5 from R&N.

Please do at least two of these problems before going on, and do the rest before the next recitation.

First-Order Resolution

OK, now what if we have variables? We're going to move to the first-order case. And there's going to be a resolution rule in the first-order case, and it's essentially the same inference rule, but the trick is what to do with the variables. And what to do with the variables is pretty hard and pretty complicated. We'll spend the rest of this lecture understanding what to do with the variables.

First-Order Resolution

$$\frac{\forall\, x.\ P(x) \rightarrow Q(x) \qquad P(A)}{Q(A)}$$

uppercase letters: constants

lowercase letters: variables

Let's try to get some intuition through an example. Imagine you knew "for all X, P of X implies Q of X." And let's say you also knew P of A. What would you be able to conclude?

Q of A, right? You ought to be able to conclude Q of A.

First-Order Resolution

$$\frac{\forall x.\ P(x) \rightarrow Q(x) \qquad P(A)}{Q(A)}$$

Syllogism:
All men are mortal
Socrates is a man
Socrates is mortal

uppercase letters: constants

lowercase letters: variables

This is actually Aristotle's original syllogism: From "All men are mortal" and "Socrates is a man", conclude "Socrates is a mortal".

First-Order Resolution

$$\frac{\forall x.\ P(x) \rightarrow Q(x) \qquad P(A)}{Q(A)}$$

Syllogism:
All men are mortal
Socrates is a man
Socrates is mortal

uppercase letters: constants

lowercase letters: variables

$$\frac{\forall x.\ \neg P(x) \vee Q(x) \qquad P(A)}{Q(A)}$$

Equivalent by definition of implication

So, how can we justify this conclusion formally. Well, the first step would be to get rid of the implication.

First-Order Resolution

$\forall$ x. P(x) $\rightarrow$ Q(x)

P(A)

Q(A)

Syllogism:

All men are mortal

Socrates is a man

Socrates is mortal

uppercase letters: constants

lowercase letters: variables

$\forall$ x. $\neg$ P(x) v Q(x)

P(A)

Q(A)

Equivalent by definition of implication

$\neg$ P(A) v Q(A)

P(A)

Q(A)

Substitute A for x, still true

then

Propositional resolution

Next, we could substitute the constant A in for the variable x in the universally quantified sentence. By the semantics of universal quantification, that's allowed. And now, we can apply the propositional resolution rule.

First-Order Resolution

$\forall$ x. P(x) $\rightarrow$ Q(x)

P(A)

Q(A)

Syllogism:

All men are mortal

Socrates is a man

Socrates is mortal

uppercase letters: constants

lowercase letters: variables

$\forall$ x. $\neg$ P(x) v Q(x)

P(A)

Q(A)

Equivalent by definition of implication

The key is finding the correct substitutions for the variables.

$\neg$ P(A) v Q(A)

P(A)

Q(A)

Substitute A for x, still true

then

Propositional resolution

The hard part is figuring out how to instantiate the variables in the universal statements. In this problem, it was clear that A was the relevant individual. But it not necessarily clear at all how to do that automatically.

Substitutions

In order to derive an algorithmic way of finding the right instantiations for the universal variables, we need something called substitutions.

Substitutions

P(x, F(y), B) : an atomic sentence

Here's an example of what we called an atomic sentence before, a predicate applied to some terms. There are two variables in here: x, and y. We can think about different ways that we can substitute terms into this expression, right? Those are called *substitution instances* of that expression.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment

A substitution is a set of variable-term pairs, written this way. It says that whenever you see variable I, you should substitute in term I. There should not be more than one entry for a single variable.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant

So here's one substitution instance. P(z,F(w),B). It's not particularly interesting. It's called an *alphabetic variant*, because we've just substituted some different variables in for x and y. In particular, we've put z in for x and w in for y, as shown in the substitution.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant
P(x, F(A), B)	{y/A}	

Here's another substitution instance of our sentence: P(x, F(A), B), We've put the constant A in for the variable y.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant
P(x, F(A), B)	{y/A}	
P(G(z), F(A), B)	{x/G(z), y/A}	

To get P(G(z), F(A), B), we substitute the term G(z) in for x and the constant A for y.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant
P(x, F(A), B)	{y/A}	
P(G(z), F(A), B)	{x/G(z), y/A}	
P(C, F(A), B)	{x/C, y/A}	Ground instance

Here's one more -- P(C, F(A), B). It's sort of interesting, because it doesn't have any variables in it. We'll call an atomic sentence with no variables a *ground instance*. Ground means it doesn't have any variables.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant
P(x, F(A), B)	{y/A}	
P(G(z), F(A), B)	{x/G(z), y/A}	
P(C, F(A), B)	{x/C, y/A}	Ground instance

学完lecture5以后，应该能够理解到这一点

You can think about substitution instances, in general, as being more specific than the original sentence. A constant is more specific than a variable. There are fewer interpretations under which a sentence with a constant is true. And even F(x) is more specific than y, because the range of F might be smaller than U.

You're not allowed to substitute anything in for a constant, or for a compound term (the application of a function symbol to some terms). You **are** allowed to substitute for a variable inside a compound term, though, as we have done with F in this example.

Substitutions

P(x, F(y), B) : an atomic sentence

Substitution instances	Substitution $\{v_1/t_1,\ldots, v_n/t_n\}$	Comment
P(z, F(w), B)	{x/z, y/w}	Alphabetic variant
P(x, F(A), B)	{y/A}	
P(G(z), F(A), B)	{x/G(z), y/A}	
P(C, F(A), B)	{x/C, y/A}	Ground instance

Applying a substitution:

subst({A/y},P(x, F(y), B)) = P(x, F(A), B)

P(x, F(y), B) {A/y} = P(x,F(A),B)

We'll use the notation of an expression followed by a substitution to mean the expression that we get by applying the substitution to the expression. And the book uses a function subst, with two arguments, the substitution and the expression.

To apply a substitution to an expression, we look to see if any of the variables in the expression have entries in the substitution. If they do, we substitute in the appropriate new expression for the variable, and continue to look for possible substitutions until no more opportunities exist.

Unification

Now we'll look at the process of unification, which is finding a substitution that makes two expressions match each other exactly.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s

So, expressions Omega 1 and Omega 2 are *unifiable* if there exists a substitution S such that (Omega-1 S) will be equal to (Omega-2 S). And that substitution S is called a *unifier* of omega1 and omega2.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s
- Let ω_1 = x and ω_2 = y, the following are unifiers

s	ω_1 s	ω_2 s

So, let's look at some unifiers of the expressions x and y. Since x and y are both variables, there are lots of things you can do to make them match.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s
- Let ω_1 = x and ω_2 = y, the following are unifiers

s	ω_1 s	ω_2 s
{y/x}	x	x

If you substitute x in for y, then both expressions come out to be x.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s
- Let ω_1 = x and ω_2 = y, the following are unifiers

s	ω_1 s	ω_2 s
{y/x}	x	x
{x/y}	y	y

If you put in y for x, then they both come out to be y.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s
- Let ω_1 = x and ω_2 = y, the following are unifiers

s	ω_1 s	ω_2 s
{y/x}	x	x
{x/y}	y	y
{x/F(F(A)), y/F(F(A))}	F(F(A))	F(F(A))

But you could also substitute something else, like F(F(A)) for x and for y, and you'd get matching expressions.

Unification

- Expressions ω_1 and ω_2 are unifiable iff there exists a substitution s such that ω_1 s = ω_2 s
- Let ω_1 = x and ω_2 = y, the following are unifiers

s	ω_1 s	ω_2 s
{y/x}	x	x
{x/y}	y	y
{x/F(F(A)), y/F(F(A))}	F(F(A))	F(F(A))
{x/A, y/A}	A	A

Or, you could substitute some constant, like A, in for both x and y.

Most General Unifier

Of the unifiers we considered on the previous slide, some of them seem a bit arbitrary. Binding both x and y to A, or to F(F(A)) is a kind of over-commitment.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

So, in fact, what we're really going to be looking for is not just any unifier of two expressions, but a **most general unifier**, or MGU.

G is a most general unifier of omega1 and omega2 if and only if for all unifiers S, there exists an S-prime such that (omega-1 S) equals (omega-1 G S-prime), and (omega-2 S) equals (omega-2 G S-prime).

A unifier is most general if every single one of the other unifiers can be expressed as an extra-substitution added onto the most general one.

It's a substitution that you can make that makes the fewest commitments, and can still make these two expressions equal.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}

So, let's do a few more examples together, and then you can do some as recitation problems. So, what's a most general unifier of P(x) and P(A)? A for x.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}

What about these two expressions? We can make them match up either by substituting x for y, or y for x. It doesn't matter which one we do. They're both "most general".

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}

Okay. What about this one? It's a bit tricky. You can kind of see that, ultimately, all of the variables are going to have to be the same. Matching the arguments to G forces y and x to be the same, And since z and y have to be the same as well (to make the middle argument match), they all have to be the same variable. Might as well make it x (though it could be any other variable).

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}
P(x, B, B)	P(A, y, z)	{x/A, y/B, z/B}

What about P(x, B, B) and P(A, y, z)? It seems pretty clear that we're going to have to substitute A for x, B for y, and B for Z.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}
P(x, B, B)	P(A, y, z)	{x/A, y/B, z/B}
P(G(F(v)), G(u))	P(x, x)	{x/G(F(v)), u/F(v)}

Here's a tricky one. It looks like x is going to have to simultaneously be G(F(v)) and G(u). How can we make that work? By substituting F(v) in for u.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}
P(x, B, B)	P(A, y, z)	{x/A, y/B, z/B}
P(G(F(v)), G(u))	P(x, x)	{x/G(F(v)), u/F(v)}
P(x, F(x))	P(x, x)	No MGU!

Now, let's try unifying P(x, F(x)) with P(x,x). The temptation is to say x has to be F(x), but then that x has to be F(x), etc. The answer is that these expressions are not unifiable.

Most General Unifier

g is a most general unifier of ω_1 and ω_2 iff for all unifiers s, there exists s′ such that ω_1 s = (ω_1 g) s′ and ω_2 s = (ω_2 g) s′

ω_1	ω_2	MGU
P(x)	P(A)	{x/A}
P(F(x), y, G(x))	P(F(x), x, G(x))	{y/x} or {x/y}
P(F(x), y, G(y))	P(F(x), z, G(x))	{y/x, z/x}
P(x, B, B)	P(A, y, z)	{x/A, y/B, z/B}
P(G(F(v)), G(u))	P(x, x)	{x/G(F(v)), u/F(v)}
P(x, F(x))	P(x, x)	No MGU!

The last time I explained this to a class, someone asked me what would happen if F were the identity function. Then, couldn't we unify these two expressions? That's a great question, and it illustrates a point I should have made before. In unification, we are interested in ways of making expressions equivalent, in every interpretation of the constant and function symbols. So, although it might be possible for the constants A and B to be equal because they both denote the same object in some interpretation, we can't unify them, because they are not required to be the same in every interpretation.

Unification Algorithm

unify(Expr x, Expr y, Subst s){

An MGU can be computed recursively, given two expressions x, and y, to be unified, and a substitution that contains substitutions that must already be made. The argument s will be empty in a top-level call to unify two expressions.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
  if s = fail, return fail
```


The algorithm returns a substitution if x and y are unifiable in the context of s, and fail otherwise. If s is already a failure, we return s.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
```


If x is equal to y, then we don’t have to do any work and we return fail.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
   else if x is a variable, return unify-var(x, y, s)
   else if y is a variable, return unify-var(y, x, s)
```


If either x or y is a variable, then we go to a special subroutine that's shown on the next slide.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
   else if x is a variable, return unify-var(x, y, s)
   else if y is a variable, return unify-var(y, x, s)
   else if x is a predicate or function application,
```


If x is a predicate or a function application, then y must be one also, with the same predicate or function.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
   else if x is a variable, return unify-var(x, y, s)
   else if y is a variable, return unify-var(y, x, s)
   else if x is a predicate or function application,
          if y has the same operator,
                 return unify(args(x), args(y), s)
```


If so, we'll unify the lists of arguments from x and y in the context of s.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
   else if x is a variable, return unify-var(x, y, s)
   else if y is a variable, return unify-var(y, x, s)
   else if x is a predicate or function application,
          if y has the same operator,
                 return unify(args(x), args(y), s)
          else   return fail
```


If not, that is, if x and y have different predicate or function symbols, we simply fail.

Unification Algorithm

```
unify(Expr x, Expr y, Subst s){
   if s = fail, return fail
   else if x = y, return s
   else if x is a variable, return unify-var(x, y, s)
   else if y is a variable, return unify-var(y, x, s)
   else if x is a predicate or function application,
          if y has the same operator,
                 return unify(args(x), args(y), s)
          else   return fail
   else                          ; x and y have to be lists
          return unify(rest(x), rest(y),
                                unify(first(x), first(y), s))
}
```


Finally, (if we get to this case, then x and y have to be lists, or something malformed), we go down the lists, unifying the first elements, then the second elements, and so on. Each time we unify a pair of elements, we get a new substitution that records the commitments we had to make to get that pair of expressions to unify. Each further unification must take place in the context of the commitments generated by the previous elements of the lists.

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

unify-var(Variable var, Expr x, Subst s){

Given a variable var, an expression x, and a substitution s, we need to return a substitution that unifies var and x in the context of s. What makes this tricky is that we have to first keep applying the existing substitutions in s to var, and to x, if it is a variable, before we're down to a new concrete problem to solve.

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
  if var is bound to val in s,
        return unify(val, x, s)
```


So, if var is bound to val in s, then we unify that value with x, in the context of s (because we're already committed that val has to be substituted for var).

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
  if var is bound to val in s,
        return unify(val, x, s)
  else if x is bound to val in s,
        return unify-var(var, val, s)
```


Similarly, if x is a variable, and it is bound to val in s, then we have to unify var with val in s. (We call unify-var directly, because we know that var is still a var).

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
  if var is bound to val in s,
        return unify(val, x, s)
  else if x is bound to val in s,
        return unify-var(var, val, s)
  else if var occurs anywhere in x, return fail
```


If var occurs anywhere in x, then fail. This is the "occurs" check, which keeps us from circularities, like binding x to F(x).

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
  if var is bound to val in s,
        return unify(val, x, s)
  else if x is bound to val in s,
        return unify-var(var, val, s)
  else if var occurs anywhere in x, return fail
  else   return add({var/x}, s)
}
```


Finally, we know var is a variable that doesn't already have a substitution, so we add the substitution of x for var to s, and return it.

Unify-var subroutine

Substitute in for var and x as long as possible, then add new binding

```
unify-var(Variable var, Expr x, Subst s){
  if var is bound to val in s,
        return unify(val, x, s)
  else if x is bound to val in s,
        return unify-var(var, val, s)
  else if var occurs anywhere in x, return fail
  else   return add({var/x}, s)
}
```

Note: last line incorrect in book!

Be careful. The last line of this algorithm in the book is incorrect.

Unification Problems

For each pair of sentences, give an MGU.

Color(Tweety, Yellow)	Color(x,y)
Color(Tweety, Yellow)	Color(x,x)
Color(Hat(John), Blue)	Color(Hat(y), x)
R(F(x), B)	R(y,z)
R(F(y), x)	R(x, F(B))
R(F(y), y, x)	R(x, F(A), F(v))
Loves(x, y)	Loves(y, x)
F(G(w), H(w, J(x, y)))	F(G(v), H(u, v))
F(G(w), H(w, J(x, u)))	F(G(v), H(u, v))
F(x, F(u, x))	F(F(y, A), F(z, F(B,z)))

Please do at least half of these problems before you go on to the next lecture, and all of them before the next recitation.

Inference using Unification

$\forall$ x. $\neg$ P(x) v Q(x)

P(A)

Q(A)

For universally quantified variables, find MGU {x/A} and proceed as in propositional resolution.

We'll spend next time talking about the first-order resolution inference rule in great detail. But just so you can see what unification is good for, here's an example inference rule.

In propositional resolution, we looked for a variable and its negation to resolve. They had to match exactly. In first-order unification, we'll look for two expressions to resolve. But now they don't have to match exactly. They just have to unify. The details are to come.