

6.034 Notes: Section 8.1

Slide 8.1.1

A sentence written in conjunctive normal form looks like $((A \text{ or } B \text{ or not } C) \text{ and } (B \text{ or } D) \text{ and } (\text{not } A) \text{ and } (B \text{ or } C))$.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

6.034 - Spring 03 • 1

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**

6.034 - Spring 03 • 2

Slide 8.1.2

Its outermost structure is a conjunction. It's a conjunction of multiple units. These units are called "clauses."

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
- A, B, and $\neg C$ are **literals**

6.034 - Spring 03 • 3

Slide 8.1.3

A clause is the disjunction of many things. The units that make up a clause are called literals.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
- A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.



6.034 - Spring 03 • 4

Slide 8.1.4

And a literal is either a variable or the negation of a variable.

Slide 8.1.5

So you get an expression where the negations are pushed in as tightly as possible, then you have ors, then you have ands. This is like saying that every assignment has to meet each of a set of requirements. You can think of each clause as a requirement. So somehow, the first clause has to be satisfied, and it has different ways that it can be satisfied, and the second one has to be satisfied, and the third one has to be satisfied, and so on.

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
- A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.
- Each clause is a requirement that must be satisfied and can be satisfied in multiple ways



6.034 - Spring 03 • 5

Conjunctive Normal Form

- Conjunctive normal form (CNF) formulas:

$$(A \vee B \vee \neg C) \wedge (B \vee D) \wedge (\neg A) \wedge (B \vee C)$$

- $(A \vee B \vee \neg C)$ is a **clause**, which is a disjunction of literals
- A, B, and $\neg C$ are **literals**, each of which is a variable or the negation of a variable.
- Each clause is a requirement that must be satisfied and can be satisfied in multiple ways
- Every sentence in propositional logic can be written in CNF



6.034 - Spring 03 • 6

Slide 8.1.6

You can take any sentence in propositional logic and write it in conjunctive normal form.

Slide 8.1.7

Here's the procedure for converting sentences to conjunctive normal form.

Converting to CNF



6.034 - Spring 03 • 7

Converting to CNF

1. Eliminate arrows using definitions

Slide 8.1.8

The first step is to eliminate single and double arrows using their definitions.



6.034 - Spring 03 • 8

Slide 8.1.9

The next step is to drive in negation. We do it using DeMorgan's Laws. You might have seen them in a digital logic class. Not (phi or psi) is equivalent to (not phi and not psi). And, Not (phi and psi) is equivalent to (not phi or not psi). So if you have a negation on the outside, you can push it in and change the connective from and to or, or from or to and.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$$



6.034 - Spring 03 • 9

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$$

3. Distribute **or** over **and**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

Slide 8.1.10

The third step is to distribute or over and. That is, if we have (A or (B and C)) we can rewrite it as (A or B) and (A or C). You can prove to yourself, using the method of truth tables, that the distribution rule (and DeMorgan's laws) are valid.



6.034 - Spring 03 • 10

Slide 8.1.11

One problem with conjunctive normal form is that, although you can convert any sentence to conjunctive normal form, you might do it at the price of an exponential increase in the size of the expression. Because if you have A and B and C OR D and E and F, you end up making the cross-product of all of those things.

For now, we'll think about satisfiability problems, which are generally fairly efficiently converted into CNF.

Converting to CNF

1. Eliminate arrows using definitions
2. Drive in negations using De Morgan's Laws

$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$

$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$$

3. Distribute **or** over **and**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

4. Every sentence can be converted to CNF, but it may grow exponentially in size



6.034 - Spring 03 • 11

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

Slide 8.1.12

Here's an example of converting a sentence to CNF.

6.034 - Spring 03 • 12

Slide 8.1.13

First we get rid of both arrows, using the rule that says "A implies B" is equivalent to "not A or B".

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

6.034 - Spring 03 • 13

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

2. Drive in negations

$$(\neg A \wedge \neg B) \vee (\neg C \vee D)$$

6.034 - Spring 03 • 14

Slide 8.1.14

Then we drive in the negation using deMorgan's law.

CNF Conversion Example

$$(A \vee B) \rightarrow (C \rightarrow D)$$

1. Eliminate arrows

$$\neg(A \vee B) \vee (\neg C \vee D)$$

2. Drive in negations

$$(\neg A \wedge \neg B) \vee (\neg C \vee D)$$

3. Distribute

$$(\neg A \vee \neg C \vee D) \wedge (\neg B \vee \neg C \vee D)$$

6.034 - Spring 03 • 15

Slide 8.1.15

Finally, we distribute or over and to get the final CNF expression.

6.034 Notes: Section 8.2

Slide 8.2.1

We have talked a little bit about proof, 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 now is resolution. Which is the way that pretty much every modern automated theorem-prover is implemented. It seems to be the best way for computers to think about proving things.

Propositional Resolution

6.034 - Spring 03 • 1

Propositional Resolution

- Resolution rule:

$$\begin{array}{l} \alpha \vee \beta \\ \neg\beta \vee \gamma \\ \hline \alpha \vee \gamma \end{array}$$

6.034 - Spring 03 • 2

Slide 8.2.2

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".

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".

Slide 8.2.3

It turns out that this one rule is all you need to prove anything in propositional logic. 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. It goes like this.

Propositional Resolution

- Resolution rule:

$$\begin{array}{l} \alpha \vee \beta \\ \neg\beta \vee \gamma \\ \hline \alpha \vee \gamma \end{array}$$

- Resolution refutation:

6.034 - Spring 03 • 3

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF

6.034 - Spring 03 • 4

Slide 8.2.4

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.

Slide 8.2.5

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:

$$\frac{\alpha \vee \beta \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- Resolution refutation:
 - Convert all sentences to CNF
 - Negate the desired conclusion (converted to CNF)

6.034 - Spring 03 • 5

Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- 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

6.034 - Spring 03 • 6

Slide 8.2.6

Now we apply the resolution rule until one of two things happens. We might derive "false", which means that the conclusion did, in fact, follow from the things that we had assumed. 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. Or, we might find ourselves in a situation where we can't apply the resolution rule any more, but we still haven't managed to derive false.

Slide 8.2.7

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. 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. 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

- Resolution rule:


$$\frac{\alpha \vee \beta \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- 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.

6.034 - Spring 03 • 7

Prove R

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

Step	Formula	Derivation



6.034 - Spring 03 • 8


Slide 8.2.8
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.

Slide 8.2.9
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.

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given




6.034 - Spring 03 • 9

Prove R

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

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



6.034 - Spring 03 • 10


Slide 8.2.10
Now, "P implies R" turns into "not P or R".

Slide 8.2.11
Similarly, "Q implies R" turns into "not Q or R"

Prove R

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

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




6.034 - Spring 03 • 11

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion



6.034 - Spring 03 • 12

Slide 8.2.12

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.


Slide 8.2.13

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).

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion




6.034 - Spring 03 • 13

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2



6.034 - Spring 03 • 14

Slide 8.2.14

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


Slide 8.2.15

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

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4




6.034 - Spring 03 • 15

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4



6.034 - Spring 03 • 16


Slide 8.2.16
Similarly, we can take lines 3 and 4, resolve away R, and get "not Q".

Slide 8.2.17
By resolving away Q in lines 5 and 7, we get R.

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7




6.034 - Spring 03 • 17

Prove R

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

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8



6.034 - Spring 03 • 18

Slide 8.2.18
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.


Slide 8.2.19
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.

Prove R

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

$$\begin{array}{l} \text{false} \vee R \\ \neg R \vee \text{false} \\ \hline \text{false} \vee \text{false} \end{array}$$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8



6.034 - Spring 03 • 19

Prove R

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

$\text{false} \vee R$

$\neg R \vee \text{false}$

$\text{false} \vee \text{false}$

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$\neg P \vee R$	Given
3	$\neg Q \vee R$	Given
4	$\neg R$	Negated conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	*	4,8

6.034 - Spring 03 • 20

Slide 8.2.20
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.

Slide 8.2.21
Here's a question. Does "P and not P" entail Z?

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

The Power of False

Prove Z

1	P
2	$\neg P$

Step	Formula	Derivation

6.034 - Spring 03 • 21

The Power of False

Prove Z

1	P
2	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion

6.034 - Spring 03 • 22

Slide 8.2.22
We start by writing down the assumptions and the negation of the conclusion.

Slide 8.2.23
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	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

6.034 - Spring 03 • 23

The Power of False

Prove Z

1	P
2	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Note that $(P \wedge \neg P) \rightarrow Z$ is **valid**

Slide 8.2.24

Because we can prove Z from "P and not P" using a sound proof procedure, then "P and not P" entails Z.

Slide 8.2.25

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. This problem has been recently addressed in AI by a move to probabilistic reasoning methods. Unfortunately, they're out of the scope of this course.

The Power of False

Prove Z

1	P
2	$\neg P$

Step	Formula	Derivation
1	P	Given
2	$\neg P$	Given
3	$\neg Z$	Negated conclusion
4	*	1,2

Note that $(P \wedge \neg P) \rightarrow Z$ is **valid**

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

Example Problem

Convert to CNF

Prove R

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

Slide 8.2.26

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

Slide 8.2.27

So, the first formula turns into "P or Q".

Example Problem

Convert to CNF

Prove R

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

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$

Example Problem

Prove R

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

Convert to CNF

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$

- $\neg(\neg P \vee P) \vee R$
- $(P \wedge \neg P) \vee R$
- $(P \vee R) \wedge (\neg P \vee R)$

6.034 - Spring 03 • 28

Slide 8.2.28

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.

Slide 8.2.29

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)".

Example Problem

Prove R

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

Convert to CNF

- $\neg(\neg P \vee Q) \vee Q$
- $(P \wedge \neg Q) \vee Q$
- $(P \vee Q) \wedge (\neg Q \vee Q)$
- $(P \vee Q)$

- $\neg(\neg P \vee P) \vee R$
- $(P \wedge \neg P) \vee R$
- $(P \vee R) \wedge (\neg P \vee R)$

- $\neg(\neg R \vee S) \vee \neg(\neg S \vee Q)$
- $(R \wedge \neg S) \vee (S \wedge \neg Q)$
- $(R \vee S) \wedge (\neg S \vee S) \wedge (R \vee \neg Q) \wedge (\neg S \vee \neg Q)$
- $(R \vee S) \wedge (R \vee \neg Q) \wedge (\neg S \vee \neg Q)$

6.034 - Spring 03 • 29

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

6.034 - Spring 03 • 30

Slide 8.2.30

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.

Slide 8.2.31

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

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
8	S	4,7
9	$\neg Q$	6,8
10	P	1,9
11	R	3,10
12	*	7,11

6.034 - Spring 03 • 31

Proof Strategies

Slide 8.2.32

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.



6.034 - Spring 03 • 32

Slide 8.2.33

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 a 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.



6.034 - Spring 03 • 33

Proof Strategies

- Unit preference: prefer a resolution step involving a 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.

Slide 8.2.34

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.




6.034 - Spring 03 • 34

6.034 Notes: Section 8.3

Slide 8.3.1

We are going to use resolution refutation to do proofs in first-order logic. It's a fair amount trickier than in propositional logic, though, because now we have variables to contend with.

First-Order Resolution




6.034 - Spring 03 • 1

First-Order Resolution

$$\begin{array}{l} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

uppercase letters: constants

lowercase letters: variables



6.034 - Spring 03 • 2

Slide 8.3.2

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(A)**. What would you be able to conclude? **Q(A)**, right? You ought to be able to conclude **Q(A)**.

Slide 8.3.3

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

$$\begin{array}{l} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Syllogism:


All men are mortal

Socrates is a man

Socrates is mortal

uppercase letters: constants

lowercase letters: variables



6.034 - Spring 03 • 3

First-Order Resolution

$$\begin{array}{l} \forall x. P(x) \rightarrow Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Syllogism:

All men are mortal

Socrates is a man


Socrates is mortal

uppercase letters: constants

lowercase letters: variables

$$\begin{array}{l} \forall x. \neg P(x) \vee Q(x) \\ P(A) \\ \hline Q(A) \end{array}$$

Equivalent by definition of implication



6.034 - Spring 03 • 4

Slide 8.3.4

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

Slide 8.3.5

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. A universally quantified statement has to be true of every object in the universe, including whatever object is denoted by the constant symbol **A**. And now, we can apply the propositional resolution rule.

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.

First-Order Resolution

$\frac{\forall x. P(x) \rightarrow Q(x)}{P(A)} \quad Q(A)$	<p>Syllogism: All men are mortal Socrates is a man Socrates is mortal</p>	<p>uppercase letters: constants lowercase letters: variables</p>
$\frac{\forall x. \neg P(x) \vee Q(x)}{P(A)} \quad Q(A)$	<p>Equivalent by definition of implication</p>	
$\frac{\neg P(A) \vee Q(A)}{P(A)} \quad Q(A)$	<p>Substitute A for x, still true then Propositional resolution</p>	

6.034 - Spring 03 • 5

First-Order Resolution

$\frac{\forall x. P(x) \rightarrow Q(x)}{P(A)} \quad Q(A)$	<p>Syllogism: All men are mortal Socrates is a man Socrates is mortal</p>	<p>uppercase letters: constants lowercase letters: variables</p>
$\frac{\forall x. \neg P(x) \vee Q(x)}{P(A)} \quad Q(A)$	<p>Equivalent by definition of implication</p>	
$\frac{\neg P(A) \vee Q(A)}{P(A)} \quad Q(A)$	<p>Substitute A for x, still true then Propositional resolution</p>	

6.034 - Spring 03 • 6

Slide 8.3.6

Now, we have to do two jobs before we can see how to do first-order resolution.

The first is to figure out how to convert from sentences with the whole rich structure of quantifiers into a form that lets us use resolution. We'll need to convert to clausal form, which is a kind of generalization of CNF to first-order logic.

The second is to automatically determine which variables to substitute in for which other ones when we're performing first-order resolution. This process is called unification.

We'll do clausal form next, then unification, and finally put it all together.

Slide 8.3.7

Clausal form (which is also sometimes called "prenex normal form") is like CNF in its outer structure (a conjunction of disjunctions, or an "and" of "ors"). But it has no quantifiers. Here's an example conversion.

Clausal Form

- like CNF in outer structure
- no quantifiers

$$\forall x. \exists y. P(x) \rightarrow R(x, y)$$

$$\neg P(x) \vee R(x, F(x))$$

6.034 - Spring 03 • 7

Converting to Clausal Form**Slide 8.3.8**

We'll go through a step-by-step procedure for systematically converting any sentence in first-order logic into clausal form.

Slide 8.3.9

The first step you guys know very well is to eliminate arrows. You already know how to do that. You convert an equivalence into two implications. And anywhere you see alpha right arrow beta, you just change it into not alpha or beta.

Converting to Clausal Form**1. Eliminate arrows**

$$\alpha \leftrightarrow \beta \Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$$

$$\alpha \rightarrow \beta \Rightarrow \neg \alpha \vee \beta$$

6.034 - Spring 03 • 9

Converting to Clausal Form**1. Eliminate arrows**

$$\alpha \leftrightarrow \beta \Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$$

$$\alpha \rightarrow \beta \Rightarrow \neg \alpha \vee \beta$$

2. Drive in negation

$$\neg(\alpha \vee \beta) \Rightarrow \neg \alpha \wedge \neg \beta$$

$$\neg(\alpha \wedge \beta) \Rightarrow \neg \alpha \vee \neg \beta$$

$$\neg \neg \alpha \Rightarrow \alpha$$

$$\neg \forall x. \alpha \Rightarrow \exists x. \neg \alpha$$

$$\neg \exists x. \alpha \Rightarrow \forall x. \neg \alpha$$

6.034 - Spring 03 • 10

Slide 8.3.10

The next thing you do is drive in negation. You already basically know how to do that. We have deMorgan's laws to deal with conjunction and disjunction, and we can eliminate double negations.

As a kind of extension of deMorgan's laws, we also have that **not (for all x, alpha)** turns into **exists x such that not alpha**. And that **not (exists x such that alpha)** turns into **for all x, not alpha**. The reason these are extensions of deMorgan's laws, in a sense, is that a universal quantifier can be seen abstractly as a conjunction over all possible assignments of **x**, and an existential as a disjunction.

Slide 8.3.11

The next step is to rename variables apart. The idea here is that every quantifier in your sentence should be over a different variable. So, if you had two different quantifications over **x**, you should rename one of them to use a different variable (which doesn't change the semantics at all). In this example, we have two quantifications involving the variable **x**. It's especially confusing in this case, because they're nested. The rules are like those for a programming language: a variable is captured by the nearest enclosing quantifier. So the **x** in **Q(x,y)** is really a different variable from the **x** in **P(x)**. To make this distinction clear, and to automate the downstream processing into clausal form, we'll just rename each of the variables.

Converting to Clausal Form**1. Eliminate arrows**

$$\alpha \leftrightarrow \beta \Rightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$$

$$\alpha \rightarrow \beta \Rightarrow \neg \alpha \vee \beta$$

2. Drive in negation

$$\neg(\alpha \vee \beta) \Rightarrow \neg \alpha \wedge \neg \beta$$

$$\neg(\alpha \wedge \beta) \Rightarrow \neg \alpha \vee \neg \beta$$

$$\neg \neg \alpha \Rightarrow \alpha$$

$$\neg \forall x. \alpha \Rightarrow \exists x. \neg \alpha$$

$$\neg \exists x. \alpha \Rightarrow \forall x. \neg \alpha$$

3. Rename variables apart

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

$$\forall x_1. \exists y_2. (\neg P(x_1) \vee \exists x_3. Q(x_3, y_2))$$

6.034 - Spring 03 • 11

Skolemization**4. Skolemize**

6.034 - Spring 03 • 12

Slide 8.3.12

Now, here's the step that many people find confusing. The name is already a good one. Step four is to skolemize, named after a logician called Thoralf Skolem. Imagine that you have a sentence that looks like: there exists an **x** such that **P(x)**. The goal here is to somehow arrive at a representation that doesn't have any quantifiers in it. Now, if we only had one kind of quantifier in first-order logic, it would be easy because we could just mention variables and all the variables would be implicitly quantified by the kind of quantifier that we have. But because we have two quantifiers, if we dropped all the quantifiers off, there's a mess, because you don't know which kind of quantification is supposed to apply to which variable.

Slide 8.3.13

The Skolem insight is that when you have an existential quantification like this, you're saying there is such a thing as a unicorn, let's say that P means "unicorn". There exists a thing such that it's a unicorn. You can just say, all right, well, if there is one, let's call it Fred. That's it. That's what Skolemization is. So instead of writing exists an x such that $P(x)$, you say $P(\text{Fred})$. The trick is that it absolutely must be a new name. It can't be any other name of any other thing that you know about. If you're in the process of inferring things about John and Mary, then it's not good to say, oh, there's a unicorn and it's John -- because that's adding some information to the picture. So to Skolemize, in the simple case, means to substitute a brand-new name for each existentially quantified variable.

Skolemization

4. Skolemize

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

6.034 - Spring 03 • 13

Skolemization

4. Skolemize

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

6.034 - Spring 03 • 14

Slide 8.3.14

For example, if I have exists x, y such that $R(x, y)$, then it's going to have to turn into $R(\text{Thing1}, \text{Thing2})$. Because we have two different variables here, they have to be given different names.

Slide 8.3.15

But the names also have to persist so that if you have exists x such that $P(x)$ and $Q(x)$, then if you skolemize that expression you should get $P(\text{Fleep})$ and $Q(\text{Fleep})$. You make up a name and you put it in there, but every occurrence of this variable has to get mapped into that same unique name.

Skolemization

4. Skolemize

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

6.034 - Spring 03 • 15

Skolemization

4. Skolemize

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

6.034 - Spring 03 • 16

Slide 8.3.16

If you have different quantifiers, then you need to use different names.

Slide 8.3.17

All right. If that's all we had to do it wouldn't be too bad. But there's one more case. We can illustrate it by looking at two interpretations of "Everyone loves someone".

In the first case, there is a single **y** that everyone loves. So we do ordinary skolemization and decide to call that person **Englebert**.

Skolemization**4. Skolemize**

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

$$\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$$

6.034 - Spring 03 • 17

Skolemization**4. Skolemize**

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

$$\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$$
- substitute new function of all universal vars in outer scopes

6.034 - Spring 03 • 18

Slide 8.3.18

In the second case, there is a different **y**, potentially, for each **x**. So, if we were just to substitute in a single constant name for **y**, we'd lose that information. We'd get the same result as above, which would be wrong. So, when you are skolemizing an existential variable, you have to look at the other quantifiers that contain the one you're skolemizing, and instead of substituting in a new constant, you substitute in a brand new function symbol, applied to any variables that are universally quantified in an outer scope.

Slide 8.3.19

In this case, what that means is that you substitute in some function of **x**, for **y**. Let's call it **Beloved(x)**. Now it's clear that the person who is loved by **x** depends on the particular **x** you're talking about.

Skolemization**4. Skolemize**

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

$$\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$$
- substitute new function of all universal vars in outer scopes

$$\forall x. \exists y. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Beloved}(x))$$

6.034 - Spring 03 • 19

Skolemization**4. Skolemize**

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

$$\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$$
- substitute new function of all universal vars in outer scopes

$$\forall x. \exists y. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Beloved}(x))$$

$$\forall x. \exists y. \forall z. \exists w. P(x, y, z) \wedge R(y, z, w) \Rightarrow$$

$$P(x, F(x), z) \wedge R(F(x), z, G(x, z))$$

6.034 - Spring 03 • 20

Slide 8.3.20

So, in this example, we see that the existential variable **w** is contained in the scope of two universally quantified variables, **x**, and **z**. So, we replace it with **G(x,z)**, which allows it to depend on the choices of **x** and **z**.

Note also, that I've been using silly names for Skolem constants and functions (like **Englebert** and **Beloved**). But you, or the computer, are only obliged to use new ones, so things like **F123221** are completely appropriate, as well.

Slide 8.3.21

Now we can drop the universal quantifiers because we just replaced all of the existential quantifiers with Skolem constants or functions. Now there's only one kind of quantifier left, so we can just drop them without losing information.

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6.034 – Spring 03 • 21

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6. Distribute or over and; return clauses

$$P(z) \vee (Q(z, w) \wedge R(w, z)) \Rightarrow \\ \{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\}$$

6.034 – Spring 03 • 22

Slide 8.3.22

And then we convert to clauses. This just means multiplying out the and's and the or's, because we already eliminated the arrows and pushed in the negations. We'll return a set of sets of literals. A literal, in this case, is a predicate applied to some terms, or the negation of a predicate applied to some terms.

I'm using set notation here for clauses, just to emphasize that they aren't lists; that the order of the literals within a clause and the order of the clauses within a set of clauses, doesn't have any effect on its meaning.

Slide 8.3.23

Finally, we can rename the variables in each clause. It's okay to do that because **for all x, P(x) and Q(x)** is equivalent to **for all y, P(y)** and **for all z, P(z)**. In fact, you don't really need to do this step, because we're assuming that you're always going to rename the variables before you do a resolution step.

Convert to Clausal Form: Last Steps**5. Drop universal quantifiers**

$$\forall x. \text{Loves}(x, \text{Beloved}(x)) \Rightarrow \text{Loves}(x, \text{Beloved}(x))$$

6. Distribute or over and; return clauses

$$P(z) \vee (Q(z, w) \wedge R(w, z)) \Rightarrow \\ \{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\}$$

7. Rename the variables in each clause

$$\{\{P(z), Q(z, w)\}, \{P(z), R(w, z)\}\} \Rightarrow \\ \{\{P(z_1), Q(z_1, w_1)\}, \{P(z_2), R(w_2, z_2)\}\}$$

6.034 – Spring 03 • 23

Example: Converting to clausal form**Slide 8.3.24**

So, let's do an example, starting with English sentences, writing them down in first-order logic, and converting them to clausal form. Later, we'll do a resolution proof using these clauses.


6.034 – Spring 03 • 24

Slide 8.3.25

John owns a dog. We can write that in first-order logic as **there exists an x such that D(x) and O(J, x)**. So, we're letting **D** stand for "is a dog" and **O** stand for "owns" and **J** stand for John.


Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J,x)$

6.034 – Spring 03 • 25

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J,x)$
$D(Fido) \wedge O(J, Fido)$

6.034 – Spring 03 • 26

Slide 8.3.26

To convert this to clausal form, we can start at step 4, Skolemization, because the previous three steps are unnecessary for this sentence. Since we just have an existential quantifier over **x**, without any enclosing universal quantifiers, we can simply pick a new name and substitute it in for **x**. Let's call **x** **Fido**. This will give us two clauses with no variables, and we're done.


Slide 8.3.27

Anyone who owns a dog is a lover of animals. We can write that in FOL as **For all x, if there exists a y such that D(y) and O(x,y), then L(x)**. We've added a new predicate symbol **L** to stand for "is a lover of animals".

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J,x)$
$D(Fido) \wedge O(J, Fido)$


b. Anyone who owns a dog is a lover-of-animals
$\forall x. (\exists y. D(y) \wedge O(x,y)) \rightarrow L(x)$

6.034 – Spring 03 • 27

Example: Converting to clausal form

a. John owns a dog
$\exists x. D(x) \wedge O(J,x)$
$D(Fido) \wedge O(J, Fido)$

b. Anyone who owns a dog is a lover-of-animals
$\forall x. (\exists y. D(y) \wedge O(x,y)) \rightarrow L(x)$
$\forall x. (\neg \exists y. (D(y) \wedge O(x,y)) \vee L(x)$

6.034 – Spring 03 • 28

Slide 8.3.28

First, we get rid of the arrow. Note that the parentheses are such that the existential quantifier is part of the antecedent, but the universal quantifier is not. The answer would come out very differently if those parens weren't there; this is a place where it's easy to make mistakes.

Slide 8.3.29

Next, we drive in the negations. We'll do it in two steps. I find that whenever I try to be clever and skip steps, I do something wrong.

Example: Converting to clausal form

a. John owns a dog

$\exists x. D(x) \wedge O(J, x)$

$D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
--

$\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
--

$\forall x. (\neg \exists y. (D(y) \wedge O(x, y)) \vee L(x)$

$\forall x. \forall y. \neg (D(y) \wedge O(x, y)) \vee L(x)$
--

$\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$

6.034 – Spring 03 • 29

Example: Converting to clausal form

a. John owns a dog

$\exists x. D(x) \wedge O(J, x)$

$D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
--

$\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
--

$\forall x. (\neg \exists y. (D(y) \wedge O(x, y)) \vee L(x)$

$\forall x. \forall y. \neg (D(y) \wedge O(x, y)) \vee L(x)$
--

$\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$

$\neg D(y) \vee \neg O(x, y) \vee L(x)$

6.034 – Spring 03 • 30

Slide 8.3.30

There's no skolemization to do, since there aren't any existential quantifiers. So, we can just drop the universal quantifiers, and we're left with a single clause.

Slide 8.3.31

Lovers of animals do not kill animals. We can write that in FOL as **For all x, L(x) implies that (for all y, A(y) implies not K(x,y))**. We've added the predicate symbol **A** to stand for "is an animal" and the predicate symbol **K** to stand for **x kills y**.

Example: Converting to clausal form

a. John owns a dog

$\exists x. D(x) \wedge O(J, x)$

$D(\text{Fido}) \wedge O(J, \text{Fido})$

c. Lovers-of-animals do not kill animals
--

$\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x, y))$
--

b. Anyone who owns a dog is a lover-of-animals
--

$\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
--

$\forall x. (\neg \exists y. (D(y) \wedge O(x, y)) \vee L(x)$

$\forall x. \forall y. \neg (D(y) \wedge O(x, y)) \vee L(x)$
--

$\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$

$\neg D(y) \vee \neg O(x, y) \vee L(x)$

6.034 – Spring 03 • 31

Example: Converting to clausal form

a. John owns a dog

$\exists x. D(x) \wedge O(J, x)$

$D(\text{Fido}) \wedge O(J, \text{Fido})$

b. Anyone who owns a dog is a lover-of-animals
--

$\forall x. (\exists y. D(y) \wedge O(x, y)) \rightarrow L(x)$
--

$\forall x. (\neg \exists y. (D(y) \wedge O(x, y)) \vee L(x)$

$\forall x. \forall y. \neg (D(y) \wedge O(x, y)) \vee L(x)$
--

$\forall x. \forall y. \neg D(y) \vee \neg O(x, y) \vee L(x)$

$\neg D(y) \vee \neg O(x, y) \vee L(x)$

c. Lovers-of-animals do not kill animals
--

$\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x, y))$
--

$\forall x. \neg L(x) \vee (\forall y. A(y) \rightarrow \neg K(x, y))$
--

$\forall x. \neg L(x) \vee (\forall y. \neg A(y) \vee \neg K(x, y))$
--

6.034 – Spring 03 • 32

Slide 8.3.32

First, we get rid of the arrows, in two steps.

Slide 8.3.33

Then we're left with only universal quantifiers, which we drop, yielding one clause.

a. John owns a dog

$\exists x. D(x) \wedge O(J,x)$

$D(Fido) \wedge O(J, Fido)$

b. Anyone who owns a dog is a lover-of-animals

$\forall x. (\exists y. D(y) \wedge O(x,y)) \rightarrow L(x)$

$\forall x. (\neg \exists y. (D(y) \wedge O(x,y)) \vee L(x)$

$\forall x. \forall y. \neg (D(y) \wedge O(x,y)) \vee L(x)$

$\forall x. \forall y. \neg D(y) \vee \neg O(x,y) \vee L(x)$

$\neg D(y) \vee \neg O(x,y) \vee L(x)$


c. Lovers-of-animals do not kill animals

$\forall x. L(x) \rightarrow (\forall y. A(y) \rightarrow \neg K(x,y))$

$\forall x. \neg L(x) \vee (\forall y. A(y) \rightarrow \neg K(x,y))$

$\forall x. \neg L(x) \vee (\forall y. \neg A(y) \vee \neg K(x,y))$

$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$



6.034 – Spring 03 • 33


Slide 8.3.34

We just have three more easy ones. "Either Jack killed Tuna or curiosity killed Tuna." Everything here is a constant, so we get **K(J,T)** or **K(C,T)**.

More converting to clausal form

d. Either Jack killed Tuna or curiosity killed Tuna

$K(J,T) \vee K(C,T)$



6.034 – Spring 03 • 34

Slide 8.3.35

"Tuna is a cat" just turns into **C(T)**.


More converting to clausal form

d. Either Jack killed Tuna or curiosity killed Tuna

$K(J,T) \vee K(C,T)$

e. Tuna is a cat

$C(T)$



6.034 – Spring 03 • 35

Slide 8.3.36

And "All cats are animals" is **not C(x) or A(x)**. I left out the steps here, but I'm sure you can fill them in.

Okay. Next, we'll see how to match up literals that have variables in them, and move on to resolution.

More converting to clausal form

d. Either Jack killed Tuna or curiosity killed Tuna


$K(J,T) \vee K(C,T)$

e. Tuna is a cat

$C(T)$

f. All cats are animals

$\neg C(x) \vee A(x)$



6.034 – Spring 03 • 36

6.034 Notes: Section 8.4

Slide 8.4.1

We introduced first-order resolution and said there were two issues to resolve before we could do it. First was conversion to clausal form, which we've done. Now we have to figure out how to instantiate the variables in the universal statements. In this problem, it was clear that A was the relevant individual. But it is not necessarily clear at all how to do that automatically.

First-Order Resolution		
$\frac{\forall x. P(x) \rightarrow Q(x)}{P(A)} \\ Q(A)$	Syllogism: All men are mortal Socrates is a man Socrates is mortal	uppercase letters: constants lowercase letters: variables
$\frac{\forall x. \neg P(x) \vee Q(x)}{P(A)} \\ Q(A)$	Equivalent by definition of implication	The key is finding the correct substitutions for the variables.
$\frac{\neg P(A) \vee Q(A)}{P(A)} \\ Q(A)$	Substitute A for x, still true then Propositional resolution	

6.034 - Spring 03 • 1

Substitutions

Slide 8.4.2

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

Slide 8.4.3

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


Substitutions

$P(x, f(y), B)$: an atomic sentence

Substitutions

$P(x, f(y), B)$: an atomic sentence

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



6.034 - Spring 03 • 4


Slide 8.4.4
A substitution is a set of variable-term pairs, written this way. It says that whenever you see variable v_i , you should substitute in term t_i . There should not be more than one entry for a single variable.

Slide 8.4.5
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 $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant




6.034 - Spring 03 • 5

Substitutions

$P(x, f(y), B)$: an atomic sentence

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



6.034 - Spring 03 • 6

Slide 8.4.6
Here's another substitution instance of our sentence: $P(x, f(A), B)$. We've put the constant A in for the variable y .


Slide 8.4.7
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 $\{v_1/t_1, \dots, v_n/t_n\}$	Comment
$P(z, f(w), B)$	$\{x/z, y/w\}$	Alphabetic variant
$P(x, f(A), B)$	$\{y/A\}$	
$P(g(z), f(A), B)$	$\{x/g(z), y/A\}$	


6.034 - Spring 03 • 7



Substitutions

$P(x, f(y), B)$: an atomic sentence

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



6.034 - Spring 03 • 8


Slide 8.4.8
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.

Slide 8.4.9
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 $\{v_1/t_1, \dots, v_n/t_n\}$	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



6.034 - Spring 03 • 9

Substitutions


$P(x, f(y), B)$: an atomic sentence

Substitution instances	Substitution $\{v_1/t_1, \dots, v_n/t_n\}$	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:

$P(x, f(y), B) \{y/A\} = P(x, f(A), B)$

$P(x, f(y), B) \{y/A, x/y\} = P(A, f(A), B)$




6.034 - Spring 03 • 10

Slide 8.4.10
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. 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.

So, in this second example, we substitute A in for y , then y in for x , and then we keep going and substitute A in for y again.

Slide 8.4.11
Now we'll look at the process of unification, which is finding a substitution that makes two expressions match each other exactly.

Unification



6.034 - Spring 03 • 11

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$

6.034 - Spring 03 • 12

Slide 8.4.12

So, expressions ω_1 and ω_2 are unifiable if and only if there exists a substitution S such that we get the same thing when we apply S to ω_1 as we do when we apply S to ω_2 . That substitution, S , is called a unifier of ω_1 and ω_2 .

Slide 8.4.13

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 $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$

6.034 - Spring 03 • 13

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x

6.034 - Spring 03 • 14

Slide 8.4.14

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

Slide 8.4.15

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 $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**


s	$\omega_1 s$	$\omega_2 s$
{y/x}	x	x
{x/y}	y	y

6.034 - Spring 03 • 15

Unification

- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
$\{y/x\}$	x	x
$\{x/y\}$	y	y
$\{x/f(f(A)), y/f(f(A))\}$	$f(f(A))$	$f(f(A))$



6.034 - Spring 03 • 16

Slide 8.4.16
But you could also substitute something else, like $f(f(A))$ for x and for y , and you'd get matching expressions.


Slide 8.4.17
Or, you could substitute some constant, like A , in for both x and y .

Some of these unifiers seem a bit arbitrary. Binding both x and y to A , or to $f(f(A))$ is a kind of over-commitment.

Unification


- Expressions ω_1 and ω_2 are **unifiable** iff there exists a substitution s such that $\omega_1 s = \omega_2 s$
- Let $\omega_1 = x$ and $\omega_2 = y$, the following are **unifiers**

s	$\omega_1 s$	$\omega_2 s$
$\{y/x\}$	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



6.034 - Spring 03 • 17

Most General Unifier



6.034 - Spring 03 • 18


Slide 8.4.18
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.

Slide 8.4.19
 G is a most general unifier of ω_1 and ω_2 if and only if for all unifiers S , there exists an S -prime such that the result of applying G followed by S -prime to ω_1 is the same as the result of applying S to ω_1 ; and the result of applying G followed by S -prime to ω_2 is the same as the result of applying S to ω_2 .

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. An MGU is 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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$



6.034 - Spring 03 • 19

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

ω_1	ω_2	MGU
$P(x)$	$P(A)$	$\{x/A\}$

6.034 - Spring 03 • 20

Slide 8.4.20

So, let's do a few examples together. What's a most general unifier of $P(x)$ and $P(A)$? A for x .

Slide 8.4.21

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

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 $\{x/y\}$

6.034 - Spring 03 • 21

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

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 $\{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$

6.034 - Spring 03 • 22

Slide 8.4.22

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

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 $\{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$
$P(x, B, B)$	$P(A, y, z)$	$\{x/A, y/B, z/B\}$

6.034 - Spring 03 • 23

Slide 8.4.23

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

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 $\{x/y\}$
$P(f(x), y, g(y))$	$P(f(x), z, g(x))$	$\{y/x, z/x\}$
$P(x, B, B)$	$P(A, y, z)$	$\{x/A, y/B, z/B\}$
$P(g(f(v)), g(u))$	$P(x, x)$	$\{x/g(f(v)), u/f(v)\}$

6.034 - Spring 03 • 24

Slide 8.4.24

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 $\omega_1 s = (\omega_1 g) s'$ and $\omega_2 s = (\omega_2 g) s'$

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!

6.034 - Spring 03 • 25

Unification Algorithm

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

6.034 - Spring 03 • 26

Slide 8.4.26

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
```

6.034 - Spring 03 • 27

Unification Algorithm

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



6.034 - Spring 03 • 28

Slide 8.4.28

If x is equal to y, then we don't have to do any work and we return s, the substitution we were given.

Slide 8.4.29

If either x or y is a variable, then we go to a special subroutine that's shown in upcoming slides.

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)
```

6.034 - Spring 03 • 29



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,
```



6.034 - Spring 03 • 30

Slide 8.4.30

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

Slide 8.4.31

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)
```



6.034 - Spring 03 • 31

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
```

6.034 - Spring 03 • 32

Slide 8.4.32

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

Slide 8.4.33

Finally, (if we get to this case, then x and y are either lists of predicate or function arguments, 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.

Because, at each stage, we find the most general unifier, we make as few commitments as possible as we go along, and therefore we never have to back up and try a different substitution.

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))
}
```

6.034 - Spring 03 • 33

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){
```

6.034 - Spring 03 • 34

Slide 8.4.34

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.

Slide 8.4.35

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)
```

6.034 - Spring 03 • 35

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)
```

6.034 - Spring 03 • 36

Slide 8.4.36

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).

Slide 8.4.37

If var occurs anywhere in x, with substitution s applied to it, 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 s), return fail
```

6.034 - Spring 03 • 37

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 s), return fail
  else return add({var/x}, s)
}
```

6.034 - Spring 03 • 38

Slide 8.4.38

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.

Slide 8.4.39

Here are a few more examples of unifications, just so you can practice. If you don't see the answer immediately, try simulating the algorithm.

Some Examples

ω_1	ω_2	MGU
A(B, C)	A(x, y)	{x/B, y/C}
A(x, f(D,x))	A(E, f(D,y))	{x/E, y/E}
A(x, y)	A(f(C,y), z)	{x/f(C,y), y/z}
P(A, x, f(g(y)))	P(y, f(z), f(z))	{y/A, x/f(z), z/g(y)}
P(x, g(f(A)), f(x))	P(f(y), z, y)	none
P(x, f(y))	P(z, g(w))	none

6.034 - Spring 03 • 39

6.034 Notes: Section 8.5

Slide 8.5.1

Now we know how to convert to clausal form and how to do unification. So now it's time to put it all together into first-order resolution.

Resolution with Variables

6.034 – Spring 03 • 1

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta \quad \neg \varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

6.034 – Spring 03 • 2

Slide 8.5.2

Here's the rule for first-order resolution. It says if you have a formula **alpha or phi** and another formula **not psi or beta**, and you can unify phi and psi with unifier theta, then you're allowed to conclude **alpha or beta** with the substitution theta applied to it.

Slide 8.5.3

Let's look at an example. Let's say we have **P(x) or Q(x,y)** and we also have **not P(A) or R(B,z)**. What are we going to be able to resolve here? We look for two literals that are negations of one another, and try to resolve them. It looks like we can resolve **P(x)** and **not P(A)**, so **P(x)** will be **phi**, **Q(x,y)** will be **alpha**, **P(A)** will be **psi** and **R(B,z)** will be **beta**. The unifier will be {x/A}.

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta \quad \neg \varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{}$$

$$\theta = \{x/A\}$$

6.034 – Spring 03 • 3

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta} \\ (\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta}$$

$$\theta = \{x/A\}$$

6.034 – Spring 03 • 4

Slide 8.5.4

So, we get rid of the P literals, and end up with **Q(x,y) or R(B,z)**, but then we have to apply our substitution (the most general unifier that was necessary to make the literals match) to the result.

Slide 8.5.5

Finally, we end up with **Q(A,y) or R(B,z)**.

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta} \\ (\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta} \\ Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$

6.034 – Spring 03 • 5

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta} \\ (\alpha \vee \beta)\theta$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, x)}{(Q(x, y) \vee R(B, z))\theta}$$

$$\frac{P(x) \vee Q(x, y) \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta} \\ Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$

6.034 – Spring 03 • 6

Slide 8.5.6

Now let's explore what happens if we have x's in the other formula. What if we replaced the z in the second sentence by an x?

Slide 8.5.7

The x's in the two sentences are actually different. There is an implicit universal quantifier on the outside of each of these sentences (remember that during the process of conversion to clausal form, we first get rid of the existentially quantified variables, then drop the remaining quantifiers, which are over universally quantified variables.) So, in order to avoid being confused by the fact that these two variables named x need not refer to the same thing, we will "rename them apart".

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta} \\ (\alpha \vee \beta)\theta$$

$$\forall x, y. \quad P(x) \vee Q(x, y) \\ \forall x. \quad \neg P(A) \vee R(B, x)$$

$$\frac{\forall x, y. \quad P(x) \vee Q(x, y) \quad \forall z. \quad \neg P(A) \vee R(B, z)}{(Q(x, y) \vee R(B, z))\theta} \\ Q(A, y) \vee R(B, z)$$

$$\theta = \{x/A\}$$

6.034 – Spring 03 • 7

Resolution with Variables

$$\frac{\alpha \vee \varphi \quad \text{MGU}(\varphi, \psi) = \theta}{\neg \varphi \vee \beta} \quad (\alpha \vee \beta)\theta$$

$\forall x, y. \quad P(x) \vee Q(x, y)$ $\forall z. \quad \neg P(A) \vee R(B, z)$
$$\frac{(Q(x, y) \vee R(B, z))\theta}{Q(A, y) \vee R(B, z)}$$
$$\theta = \{x/A\}$$

$\forall x, y. \quad P(x) \vee Q(x, y)$ $\forall x. \quad \neg P(A) \vee R(B, x)$
$$\frac{P(x_1) \vee Q(x_1, y_1) \quad \neg P(A) \vee R(B, x_2)}{(Q(x_1, y_1) \vee R(B, x_2))\theta}$$
$$\frac{}{Q(A, y_1) \vee R(B, x_2)}$$
$$\theta = \{x_1/A\}$$

Slide 8.5.8

So that means that before you try to do a resolution step, you're really supposed to rename the variables in the two sentences so that they don't share any variables in common. You won't usually need to do this that explicitly on your paper as you work through a proof, but if you were going to implement resolution in a computer program, or if you find yourself with the same variable in both sentences and it's getting confusing, then you should rename the sentences apart.

The easiest thing to do is to just go through and give every variable a new name. It's OK to do that. You just have to do it consistently for each clause. So you could rename to **P(x₁) or Q(x₁, y₁)**, and you can name this one **not P(A) or R(B, x₂)**. And then you could apply the resolution rule and you don't get into any trouble.

Slide 8.5.9
Okay. Now that we know how to do resolution, let's practice it on the example that we started in the section on clausal form. We want to prove that curiosity killed the cat.

Curiosity Killed the Cat

6.034 – Spring 03 • 9

Curiosity Killed the Cat

1	D(Fido)	a
2	O(1,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(1,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f

6.034 – Spring 03 • 10

Slide 8.5.10
Here are the clauses that we got from the original axioms.

Slide 8.5.11
Now we assert the negation of the thing we're trying to prove, so we have **not K(C,T)**.

Curiosity Killed the Cat


1	D(Fido)	a
2	O(1,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(1,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg

6.034 – Spring 03 • 11

file:///C:/Documents%20and%20Settings/Administrator/My%20Documents/6.034/07/lessons/Chapter8/logicII-handout-07.html (35 of 60)4/20/2007 7:55:21 AM

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8


6.034 – Spring 03 • 12

Slide 8.5.12
We can apply the resolution rule to any pair of lines that contain unifiable literals. Here's one way to do the proof. We'll use the "set-of-support" heuristic (which says we should involve the negation of the conclusion in the proof), and resolve away K(C,T) from lines 5 and 8, yielding K(J,T).

Slide 8.5.13
Then, we can resolve C(T) and **not** C(x) in lines 6 and 7 by substituting T for x, and getting A(T).


Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}

6.034 – Spring 03 • 13

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}


6.034 – Spring 03 • 14

Slide 8.5.14
Using lines 4 and 9, and substituting J for x and T for y, we get **not** L(J) or **not** A(T).

Slide 8.5.15
From lines 10 and 11, we get **not** L(J).


Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11

6.034 – Spring 03 • 15

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}



6.034 – Spring 03 • 16

Slide 8.5.16


From 3 and 12, substituting J for x, we get **not D(y) or not O(J,y)**.

Slide 8.5.17

From 13 and 2, substituting Fido for x, we get **not D(Fido)**.

Curiosity Killed the Cat


1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}
14	$\neg D(Fido)$	13,2 {y/Fido}



6.034 – Spring 03 • 17

Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	$\neg D(y) \vee \neg O(x,y) \vee L(x)$	b
4	$\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$	c
5	$K(J,T) \vee K(C,T)$	d
6	C(T)	e
7	$\neg C(x) \vee A(x)$	f
8	$\neg K(C,T)$	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	$\neg L(J) \vee \neg A(T)$	4,9 {x/J, y/T}
12	$\neg L(J)$	10,11
13	$\neg D(y) \vee \neg O(J,y)$	3,12 {x/J}
14	$\neg D(Fido)$	13,2 {y/Fido}
15	*	14,1



6.034 – Spring 03 • 18

Slide 8.5.18


And finally, from lines 14 and 1, we derive a contradiction. Yay! Curiosity did kill the cat.

Slide 8.5.19

So, if we want to use resolution refutation to prove that something is valid, what would we do? What do we normally do when we do a proof using resolution refutation?

Proving validity

- How do we use resolution refutation to prove something is valid?



6.034 – Spring 03 • 19

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms



6.034 – Spring 03 • 20

Slide 8.5.20

We say, well, if I know all these things, I can prove this other thing I want to prove. We prove that the premises entail the conclusion.

Slide 8.5.21

What does it mean for a sentence to be valid, in the language of entailment? That it's true in all interpretations. What that means really is that it should be derivable from nothing. A valid sentence is entailed by the empty set of sentences. The valid sentence is true no matter what. So we're going to prove something with no assumptions.

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences



6.034 – Spring 03 • 21

Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences
- To prove validity by refutation, negate the sentence and try to derive contradiction.



6.034 – Spring 03 • 22

Slide 8.5.22

We can prove it by resolution refutation by negating the sentence and trying to derive a contradiction.

Slide 8.5.23

So, let's do an example. Imagine that we would like to show the validity of this sentence, which is a classical Aristotelian syllogism.

Proving validity: example

- Syllogism

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



6.034 – Spring 03 • 23


Proving validity: example

Syllogism

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

Negate and convert to clausal form

$$\neg((\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A))$$
$$\neg((\forall x. \neg P(x) \vee Q(x)) \vee \neg P(A) \vee Q(A))$$
$$(\forall x. \neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$
$$(\neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$



6.034 – Spring 03 • 24


Slide 8.5.24
We start by negating it and converting to clausal form. We get rid of the arrows and drive in negations to arrive at this sentence in clausal form.

Slide 8.5.25
We enter the clauses into our proof.

Proving validity: example

Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.		
5.		




6.034 – Spring 03 • 25

Proving validity: example

Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.	$Q(A)$	1,2
5.	■	3,4



6.034 – Spring 03 • 26

Slide 8.5.26
Now, we can resolve lines 1 and 2, substituting A for X, and get Q(A).
And we can resolve 3 and 4, to get a contradiction.

Slide 8.6.1

In this section, we're going to look at three techniques for making logical proof more useful, and then conclude by talking about the limits of first-order logic.

Miscellaneous Logic Topics

- Factoring
- Green's trick
- Equality
- Completeness and decidability

6.034 - Spring 03 • 1

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete

6.034 - Spring 03 • 2

Slide 8.6.2

The version of the first-order resolution rule that we have shown you is called binary resolution because it involves two literals, one from each clause being resolved. It turns out that this form of resolution is not complete for first-order logic. There are sets of unsatisfiable clauses that will not generate a contradiction by successive applications of binary resolution.

Slide 8.6.3

Here's a pair of clauses. **P(x) or P(y)** and **not P(v) or not P(w)**. Can we get a contradiction from them using binary resolution?

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$\begin{aligned} &P(x) \vee P(y) \\ &\neg P(v) \vee \neg P(w) \end{aligned}$$

6.034 - Spring 03 • 3

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$\begin{aligned} &P(x) \vee P(y) \\ &\neg P(v) \vee \neg P(w) \end{aligned}$$

- We should!

6.034 - Spring 03 • 4

Slide 8.6.4

We should be able to! They are clearly unsatisfiable. There is no possible interpretation that will make both of these clauses simultaneously true, since that would require **P(x) and not (P x)** to be true for everything in the universe.

Slide 8.6.5

But when we apply binary resolution to these clauses, all we can get is something like **P(x) or not P(w)** (your variables may vary).

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$P(x) \vee P(y)$$

$$\neg P(v) \vee \neg P(w)$$
- We should!
- But all we can get is $P(x) \vee \neg P(w)$

6.034 - Spring 03 • 5

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$P(x) \vee P(y)$$

$$\neg P(v) \vee \neg P(w)$$
- We should!
- But all we can get is $P(x) \vee \neg P(w)$
- And from there all we can do is get back to one of the original clauses

6.034 - Spring 03 • 6

Slide 8.6.6

If we use binary resolution on this new clause with one of the parent clauses, we get back one of the parent clauses. We do not get a contradiction. So, we have shown by counterexample that binary resolution is not, in fact, a complete strategy.

Slide 8.6.7

It turns out that there is a simple extension of binary resolution that is complete. In that version, known as generalized resolution, we look for subsets of literals in one clause that can be unified with the negation of a subset of literals in the other clause. In our example from before, each P literal in one clause can be unified with its negation in the other clause.

Factoring

- Generalized resolution lets you resolve away multiple literals at once

6.034 - Spring 03 • 7

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

6.034 - Spring 03 • 8

Slide 8.6.8

An alternative to using generalized resolution is to introduce a new inference rule, in addition to binary resolution, called factoring. In factoring, if you can unify two literals within a single clause, alpha and beta in this case, with unifier theta, then you can drop one of them from the clause (it doesn't matter which one), and then apply the unifier to the whole clause.

Slide 8.6.9

So, for example, we can apply factoring to this sentence, by unifying $P(x,y)$ and $P(v,A)$.

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

$$Q(y) \vee P(x, y) \vee P(v, A)$$

6.034 – Spring 03 • 9

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

$$\frac{Q(y) \vee P(x, y) \vee P(v, A)}{(Q(y) \vee P(x, y))\{x / v, y / A\}} \\ Q(A) \vee P(v, A)$$

6.034 – Spring 03 • 10

Slide 8.6.10

We get $Q(Y)$ or $P(x,Y)$, and then we have to apply the substitution $\{x/v, y/A\}$, which yields the result $Q(A)$ or $P(v,A)$.

Note that factoring in propositional logic is just removing duplicate literals from sentences, which is obvious and something we've been doing without comment.

Slide 8.6.11

And binary resolution, combined with factoring, is complete in a sense that we'll study more carefully later in this section.

Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

$$\frac{Q(y) \vee P(x, y) \vee P(v, A)}{(Q(y) \vee P(x, y))\{x / v, y / A\}} \\ Q(A) \vee P(v, A)$$

- Binary resolution plus factoring is complete

6.034 – Spring 03 • 11

Green's Trick

- Use resolution to get answers to existential queries

6.034 – Spring 03 • 12

Slide 8.6.12

One thing you can do with resolution is ask for an answer to a question. If your desired conclusion is that there exists an x such that $P(x)$, then in the course of doing the proof, we'll figure out what value of x makes $P(x)$ true. We might be interested in knowing that answer. For instance, it's possible to do a kind of planning via theorem proving, in which the desired conclusion is "There exists a sequence of actions such that, if I do them in my initial state, my goal will be true at the end." Generally, you're not just interested in whether such a sequence exists, but in what it is.

One way to deal with this is Green's trick, named after Cordell Green, who pioneered the use of logic in software engineering applications. We'll see it by example.

Slide 8.6.13


Let's say we know that all men are mortal and that Socrates is a man. We want to know whether there are any mortals. So, our desired conclusion, negated and turned into clausal form would be **not Mortal(x)**. Green's trick will be to add a special extra literal onto that clause, of the form **Answer(x)**.

Green's Trick

- Use resolution to get answers to existential queries

$\exists x. \text{Mortal}(x)$

1.	$\neg \text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\text{Socrates})$	
3.	$\neg \text{Mortal}(x) \vee \text{Answer}(x)$	
4.		
5.		




6.034 – Spring 03 • 13

Green's Trick

- Use resolution to get answers to existential queries

$\exists x. \text{Mortal}(x)$

1.	$\neg \text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\text{Socrates})$	
3.	$\neg \text{Mortal}(x) \vee \text{Answer}(x)$	
4.	$\text{Mortal}(\text{Socrates})$	1,2
5.	$\text{Answer}(\text{Socrates})$	3,5



6.034 – Spring 03 • 14

Slide 8.6.14


Now, we do resolution as before, but when we come to a clause that contains only the answer literal, we stop. And whatever the variable x is bound to in that literal is our answer.

Slide 8.6.15

When we defined the language of first-order logic, we defined a special equality predicate. And we also defined special semantics for it (the sentence term1 equals term2 holds in an interpretation if and only if term1 and term2 both denote the same object in that interpretation). In order to do proofs that contain equality statements in them, we have to add a bit more mechanism.

Equality


- Special predicate in syntax and semantics; need to add something to our proof system



6.034 – Spring 03 • 15

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation



6.034 – Spring 03 • 16

Slide 8.6.16

One strategy is to add one more special proof rule, just as we did with factoring. The new proof rule is called paramodulation. It's kind of hairy and hard to use, though, so we are not going to do it in this class (though it is implemented in most serious theorem provers).

Slide 8.6.17

Another strategy, which is easier to understand, and instructive, is to treat equality almost like any other predicate, but to constrain its semantics via axioms.

Just to make it clear that Equals, which we will write as Eq, is a predicate that we're going to handle normally in resolution, we'll write it with a word rather than the equals sign.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

6.034 – Spring 03 • 17



Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x. \text{Eq}(x, x)$$

6.034 – Spring 03 • 18



Slide 8.6.18

Equals has two important sets of properties. The first three say that it is an equivalence relation. First, it's symmetric: every x is equal to itself.

Slide 8.6.19

Second, it's reflexive. If x is equal to y then y is equal to x.

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x. \text{Eq}(x, x)$$

$$\forall x, y. \text{Eq}(x, y) \rightarrow \text{Eq}(y, x)$$

6.034 – Spring 03 • 19



Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x. \text{Eq}(x, x)$$

$$\forall x, y. \text{Eq}(x, y) \rightarrow \text{Eq}(y, x)$$

$$\forall x, y, z. \text{Eq}(x, y) \wedge \text{Eq}(y, z) \rightarrow \text{Eq}(x, z)$$

6.034 – Spring 03 • 20



Slide 8.6.20

Third, it's transitive. That means that if x equals y and y equals z, then x equals z.

Slide 8.6.21

The other thing we need is the ability to "substitute equals for equals" into any place in any predicate. That means that, for each place in each predicate, we'll need an axiom that looks like this: for all x and y , if x equals y , then if P holds of x , it holds of y .

Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x. \text{Eq}(x, x)$$

$$\forall x, y. \text{Eq}(x, y) \rightarrow \text{Eq}(y, x)$$

$$\forall x, y, z. \text{Eq}(x, y) \wedge \text{Eq}(y, z) \rightarrow \text{Eq}(x, z)$$

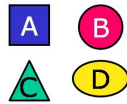
- For every predicate, allow substitutions

$$\forall x, y. \text{Eq}(x, y) \rightarrow (P(x) \rightarrow P(y))$$

6.034 – Spring 03 • 21

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is



6.034 – Spring 03 • 22

Slide 8.6.22

Let's go back to our old geometry domain and try to prove what the hat of A is.

Slide 8.6.23

We know that these axioms (our old KB4) entail that $\text{hat}(A) = A$. We'll have to add in the equality axioms, as well.

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is
- Axioms in FOL (plus equality axioms)

$$\text{Above}(A, C)$$

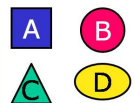
$$\text{Above}(B, D)$$

$$\neg \exists x. \text{Above}(x, A)$$

$$\neg \exists x. \text{Above}(x, B)$$

$$\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$$

$$\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$$



6.034 – Spring 03 • 23

Proof Example

- Let's go back to our old geometry domain and try to prove what the hat of A is
- Axioms in FOL (plus equality axioms)

$$\text{Above}(A, C)$$

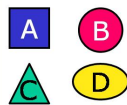
$$\text{Above}(B, D)$$

$$\neg \exists x. \text{Above}(x, A)$$

$$\neg \exists x. \text{Above}(x, B)$$

$$\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x$$

$$\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x$$



- Desired conclusion: $\exists x. \text{hat}(A) = x$
- Use Green's trick to get the binding of x

6.034 – Spring 03 • 24

Slide 8.6.24

Let's see if we can derive that, using resolution refutation and Green's trick.

Slide 8.6.25

Here's the result of my clausal-form converter run on those axioms.

The Clauses		
1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.		
11.		
12.		

6.034 - Spring 03 • 25

Slide 8.6.26

Now, our goal is to prove **exists x such that Eq(hat(A),x)**. That is negated and turned into clausal form, yielding **not Eq(hat(A),x)**. And we add in the answer literal, so we can keep track of what the answer is.

The Query		
1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	

6.034 - Spring 03 • 26

Slide 8.6.27

Here's the proof. The answer is A! And we figured this out without any kind of enumeration of interpretations.

The Proof		
1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	conclusion
11.	Above(sk(A), A) v Answer(A)	6, 10 {x/A}
12.	Answer(A)	11, 3 {x/sk(A)}

6.034 - Spring 03 • 27

Slide 8.6.28

What if we wanted to use the same axioms to figure out what the hat of D is? We just change our query and do the proof. Here it is.

Hat of D		
1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(D), x) v Answer(x)	conclusion
11.	~Above(x,D) v Answer(x)	5, 10 {x1/x}
12.	Answer(B)	11, 2 {x/B}


6.034 - Spring 03 • 28

Slide 8.6.29
Here's a worked example of a problem with equality.

Who is Jane's Lover?

- Jane's lover drives a red car
- Fred is the only person who drives a red car
- Who is Jane's lover?


1.	Drives(lover(Jane))	
2.	$\sim \text{Drives}(x) \vee \text{Eq}(x, \text{Fred})$	
3.	$\sim \text{Eq}(\text{lover}(\text{Jane}), x) \vee \text{Answer}(x)$	
4.	$\text{Eq}(\text{lover}(\text{Jane}), \text{Fred})$	1,2 {x/lover(Jane)}
5.	Answer(Fred)	3,4 {x/Fred}



6.034 – Spring 03 • 29

Completeness and Decidability

- Complete: If KB entails S, then we can prove S from KB



6.034 – Spring 03 • 30


Slide 8.6.30
Now, let's see what we can say, in general, about proof in first-order logic.

Remember that a proof system is complete, if, whenever the KB entails S, we can prove S from KB.

Slide 8.6.31
In 1929, Gödel proved a completeness theorem for first-order logic: There exists a complete proof system for FOL. But, living up to his nature as a very abstract logician, he didn't come up with such a proof system; he just proved one existed.

Completeness and Decidability


- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*



6.034 – Spring 03 • 31

Completeness and Decidability

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*
- Robinson's Completeness Theorem: *Resolution refutation is a complete proof system for FOL*



6.034 – Spring 03 • 32

Slide 8.6.32
Then, in 1965, Robinson came along and showed that resolution refutation is sound and complete for FOL.

Slide 8.6.33

So, we know that if a proof exists, then we can eventually find it with resolution. Unfortunately, we no longer know, as we did in propositional resolution, that eventually the process will stop. So, it's possible that there is no proof, and that the resolution process will run forever.

This makes first-order logic what is known as "semi-decidable". If the answer is "yes", that is, if there is a proof, then the theorem prover will eventually halt and say so. But if there isn't a proof, it might run forever!

Completeness and Decidability

- **Complete:** If KB entails S , then we can prove S from KB
- **Gödel's Completeness Theorem:** *There exists a complete proof system for FOL*
- **Robinson's Completeness Theorem:** *Resolution refutation is a complete proof system for FOL*
- **FOL is semi-decidable:** if the desired conclusion follows from the premises then eventually resolution refutation will find a contradiction.
 - If there's a proof, we'll halt with it
 - If not, maybe we'll halt, maybe not



6.034 - Spring 03 • 33

Adding Arithmetic

6.034 - Spring 03 • 34



Slide 8.6.34

So, things are relatively good with regular first-order logic. And they're still fine if you add addition to the language, allowing statements like $P(x)$ and $(x + 2 = 3)$. But if you add addition and multiplication, it starts to get weird!

Slide 8.6.35

In 1931, Gödel proved an incompleteness theorem, which says that there is no consistent, complete proof system for FOL plus arithmetic. (Consistent is the same as sound.) Either there are sentences that are true, but not provable, or there are sentences that are provable, but not true. It's not so good either way.

Adding Arithmetic

- **Gödel's Incompleteness Theorem:** *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.



6.034 - Spring 03 • 35

Adding Arithmetic

- **Gödel's Incompleteness Theorem:** *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
 - $P = \text{"P is not provable."}$



6.034 - Spring 03 • 36

Slide 8.6.36

Here's the roughest cartoon of how the proof goes. Arithmetic gives you the ability to construct code names for sentences within the logic, and therefore to construct sentences that are self-referential. This sentence, P , is sometimes called the Gödel-sentence. P is "P is not provable".

Slide 8.6.37

If P is true, then P is not provable (so the system is incomplete). If P is false, then P is provable (so the system is inconsistent).

This result was a huge blow to the current work on the foundations of mathematics, where they were, essentially, trying to formalize all of mathematical reasoning in first-order logic. And it pre-figured, in some sense, Turing's work on uncomputability.

Ultimately, though, just as uncomputability doesn't worry people who use computer programs for practical applications, incompleteness shouldn't worry practical users of logic.

Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
 - $P = \text{"}P \text{ is not provable."}$
 - If P is true: it's not provable (incomplete)
 - If P is false: it's provable (inconsistent)



6.034 - Spring 03 • 37

6.034 Notes: Section 8.7

Slide 8.7.1

Now that we've studied the syntax and semantics of logic, and know something about how to do inference in it, we're going to talk about how logic has been applied in real domains, and look at an extended example.

Logic in the Real World



6.034 - Spring 03 • 1

Logic in the Real World

- Encode information formally in web pages



6.034 - Spring 03 • 2

Slide 8.7.2

There is currently a big resurgence of logical representations and inference in the context of the web. As it stands now, web pages were written in natural language (English or French, etc), by the people and for the people. But there is an increasing desire to have computer programs (web agents or 'bots) crawl the web and figure things out by "reading" web pages. As we'll see in the next module of this course, it can be quite hard to extract the meaning from text written in natural language. So the World-Wide Web Consortium, in conjunction with people in universities and industry, are defining a standard language, which is essentially first-order logic, for formally encoding information in web pages. Information that is written in this formal language will be much easier to extract automatically.

Slide 8.7.3

It is becoming more appealing, in business, to have computers talk directly to one another, and to leave humans out of the loop. One place this can happen is in negotiating simple contracts between companies to deliver goods at some price. Benjamin Grosz, who is a professor in the Sloan School, works on using non-monotonic logic (a version of first-order logic, in which you're allowed to have conflicting rules, and have a system for deciding which ones have priority) to specify a company's business rules.

Logic in the Real World

- Encode information formally in web pages
- Business rules



6.034 - Spring 03 • 3

Logic in the Real World

- Encode information formally in web pages
- Business rules
- Airfare pricing



6.034 - Spring 03 • 4

Slide 8.7.4

Another example, which we'll pursue in detail, is the language the airlines use to specify the rules on their airfares. It turns out that every day, many times a day, airlines revise and publish (electronically) their fare structures. And, as many of you know, the rules governing the pricing of airplane tickets are pretty complicated, and certainly unintuitive. In fact, they're so complicated that the airlines had to develop a formal language that is similar to logic, in order to describe their different kinds of fares and the restrictions on them.

Amazingly, there are on the order of 20 million different fares! To generate a price for a particular proposed itinerary, it requires piecing together a set of fares to cover the parts of the itinerary. Typically, the goal is to find the cheapest such set of fares, subject to some constraints.

Slide 8.7.5

We're not going to worry about how to do the search to find the cheapest itinerary and fare structure (that's a really hard and interesting search problem!). Instead, we'll just think about pricing a particular itinerary.

Pricing an airline ticket is not as simple as adding up the prices for the individual flight legs. There are many different pricing schemes, each depending on particular attributes of the combination of flights that the passenger proposes to take. For instance, at some point in 1998, American Airlines had 29 different fares for going from Boston to San Francisco, ranging in price from \$1943 to \$231, each with a different constraint on its use.

In this discussion, we won't get into the actual ticket prices; instead we'll work on writing down the logical expressions that describe when a particular fare applies to a proposed itinerary.

Airfare Pricing

- Ignore, for now, finding the best itinerary
- Given an itinerary, what's the least amount we can pay for it?
- Can't just add up prices for the flight legs; different prices for different flights in various combinations and circumstances



6.034 - Spring 03 • 5

Fare Restrictions

- Passenger under 2 or over 65
- Passenger accompanying someone paying full fare
- Doesn't go through an expensive city
- No flights during rush hour
- Stay over Saturday night
- Layovers are legal
- Round-the-world itinerary that doesn't backtrack
- Regular two phase round-trip
- No flights on another airline
- This fare would not be cheaper than the standard price



6.034 - Spring 03 • 6

Slide 8.7.6

Here are some examples of airfare restrictions that we might want to encode logically:


- The passenger is under 2 or over 65
- The passenger is accompanying another passenger who is paying full fare
- It doesn't go through an expensive city
- There are no flights during rush hour (defined in local time)
- The itinerary stays over a Saturday night
- Layovers are legal: not too short; not too long
- Round-the-world itinerary that doesn't backtrack
- The itinerary is a regular two-trip round-trip
- This price applies to one flight, as long as there is no other flight in this itinerary operated by El Cheapo Air
- If the sum of the fares for a circle trip with three legs is less than the "comparable" round-trip price between the origin and any stopover point, then you must add a surcharge.

Slide 8.7.7

The first step in making a logical formalization of a domain is coming up with an *ontology*. According to Leibniz (a philosopher from the 17th century), ontology is "the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident." Whoa! I wish our lecture notes sounded that deep.

Ontology

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.



Leibniz

6.034 - Spring 03 • 7


Slide 8.7.8

Now there are web sites called www.ontology.org with paper titles like "The Role of Ontological Engineering in B2B Net Markets". That's just as scary.

Ontology

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

The Role of Ontological Engineering in B2B Net Markets



Leibniz

ONTOLOGY.ORG
ENABLING VIRTUAL BUSINESS

6.034 - Spring 03 • 8

Slide 8.7.9


For us, more prosaically, an ontology will be a description of the kinds of objects that you have in your world and their possible properties and relations. Making up an ontology is a lot like deciding what classes and methods you'll need when you design an object-oriented program.

Ontology

- What kinds of things are there in the world?
- What are their properties and relations?

Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

The Role of Ontological Engineering in B2B Net Markets



Leibniz

ONTOLOGY.ORG
ENABLING VIRTUAL BUSINESS

6.034 - Spring 03 • 9

Airfare Domain Ontology

Slide 8.7.10

Okay. So what are the kinds of things we have in the airfare domain? That's a hard question, because it depends on the level of abstraction at which we want to make our model. Probably we don't want to talk about particular people or airplanes; but we might need to talk about people in general, in terms of various properties (their ages, for example, but not their marital status), or airplane types. We will need a certain amount of detail though, so, for instance, it might matter which airport within a city you're using, or which terminal within an airport. Often you have to adjust the level of abstraction that you use as you go along.

Slide 8.7.11

Here's a list of the relevant object types I came up with:

- passenger
- flight
- city
- airport
- terminal
- flight segment (a list of flights, to be flown all in one "day")
- itinerary (a passenger and a list of flight segments)

Airfare Domain Ontology

- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)



6.034 – Spring 03 • 11

Airfare Domain Ontology

- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)
- list
- number



6.034 – Spring 03 • 12

Slide 8.7.12

We'll also need some non-concrete object types, including

- list
- number

Slide 8.7.13

Once we know what kinds of things we have in our world, we need to come up with a vocabulary of constant names, predicate symbols, and function symbols that we'll use to talk about their properties and relations.

There are two parts to this problem. We have to decide what properties and relations we want to be able to represent, and then we have to decide how to represent them.

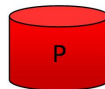
Representing Properties



6.034 – Spring 03 • 13

Representing Properties

- Object P is red
 - Red(P)
 - Color(P, Red)
 - color(P) = Red
 - Property(P, Color, Red)



6.034 – Spring 03 • 14

Slide 8.7.14

Let's talk, for a minute, about something simple, like saying that an object named P is red. There are a number of ways to say this, including:

- Red(P)
- Color(P, Red)
- color(P) = Red
- Property(P, Color, Red)

Slide 8.7.15

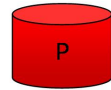
Let's look at the difference between the first two. $\text{Red}(P)$ seems like the most straightforward way to say that P is red. But what if we wanted to write a rule saying that all the blocks in a particular stack, S , are the same color? Using the second representation, we could say:

$\text{exists } c. \text{ all } b. \text{ In}(b, S) \rightarrow \text{Color}(b, c)$

In this case, we have *reified* redness; that is, we've made it into an object that can be named and quantified over. It will turn out that it's often useful to use this kind of representation.

Representing Properties

- Object P is red
 - $\text{Red}(P)$
 - $\text{Color}(P, \text{Red})$
 - $\text{color}(P) = \text{Red}$
 - $\text{Property}(P, \text{Color}, \text{Red})$



- All the blocks in stack S are the same color

$$\exists c. \forall b. \text{In}(b, S) \rightarrow \text{Color}(b, c)$$

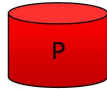
6.034 – Spring 03 • 15

Representing Properties

- Object P is red
 - $\text{Red}(P)$
 - $\text{Color}(P, \text{Red})$
 - $\text{color}(P) = \text{Red}$
 - $\text{Property}(P, \text{Color}, \text{Red})$
- All the blocks in stack S are the same color

$$\exists c. \forall b. \text{In}(b, S) \rightarrow \text{Color}(b, c)$$
- All the blocks in stack S have the same properties

$$\forall p. \exists v. \forall b. \text{In}(b, S) \rightarrow \text{Property}(b, p, v)$$



6.034 – Spring 03 • 16

Slide 8.7.16

It's possible to go even farther down this road, in case, for instance, we wanted to say that all the blocks in stack S have all the same properties:

$\text{all } p. \text{ exists } v. \text{ all } b. \text{In}(b, S) \rightarrow \text{Property}(b, p, v)$

That is, for every property, there's a value, such that every block in S has that value for the property.

Some people advocate writing all logical specifications using this kind of formalization, because it is very general; but it's also kind of hard to read. We'll stick to representations closer to $\text{Color}(P, \text{Red})$.

Slide 8.7.17

The particular relations we'll use come from two sources. Some relations will be used to specify the basic facts in our knowledge-base.

Basic Relations

6.034 – Spring 03 • 17

Basic Relations

- $\text{Age}(\text{passenger}, \text{number})$
- $\text{Nationality}(\text{passenger}, \text{country})$
- $\text{Wheelchair}(\text{passenger})$
- $\text{Origin}(\text{flight}, \text{airport})$
- $\text{Destination}(\text{flight}, \text{airport})$
- $\text{Departure_Time}(\text{flight}, \text{number})$
- $\text{Arrival_Time}(\text{flight}, \text{number})$
- $\text{Latitude}(\text{city}, \text{number})$
- $\text{Longitude}(\text{city}, \text{number})$
- $\text{In_Country}(\text{city}, \text{country})$
- $\text{In_City}(\text{airport}, \text{city})$
- $\text{Passenger}(\text{itinerary}, \text{passenger})$
- $\text{Flight_Segments}(\text{itinerary}, \text{passenger}, \text{segments})$
- Nil
- $\text{cons}(\text{object}, \text{list}) \Rightarrow \text{list}$

6.034 – Spring 03 • 18

Slide 8.7.18

Here are some of the basic relations in our domain. I've named the arguments with the types of objects that we expect to be there. This is just an informal convention to show you how we intend to use the relations. (There are logics that are strongly typed, which require you to declare the types of the arguments to each relation or function).

Slide 8.7.19

So, we might describe passenger Fred using the sentences

```
Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)
```

This only serves to encode a very simple version of this domain. We haven't started to talk about time zones, terminals, metropolitan areas (usually it's okay to fly into San Jose and then out of San Francisco, as if they were the same city), airplane types, how many reservations a flight currently has, and so on and so on.

Basic Relations

- Age(passenger, number)
- Nationality(passenger, country)
- Wheelchair(passenger)
- Origin(flight, airport)
- Destination(flight, airport)
- Departure_Time(flight, number)
- Arrival_Time(flight, number)
- Latitude(city, number)
- Longitude(city, number)
- In_Country(city, country)
- In_City(airport, city)
- Passenger(itinerary, passenger)
- Flight_Segments(itinerary, passenger, segments)
- Nil
- cons(object,list) => list

```
Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)
```

6.034 – Spring 03 • 19

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies}_{37}(i)$$

6.034 – Spring 03 • 20

Slide 8.7.20

Other relations will be used to express the things we want to infer. An example in this domain might be `Qualifies_for_fare_class_37` or something else equally intuitive. As we begin trying to write down a logical expression for `Qualifies_for_fare_class_37` in terms of the basic relations, we'll find that we want to define more intermediate relations. It will be exactly analogous to defining functions when writing a Scheme program: it's not strictly necessary, but no-one would ever be able to read your program (and you probably wouldn't be able to write it correctly) if you didn't.

Slide 8.7.21

We will often define relations using implications rather than equivalence. It makes it easier to add additional pieces of the definition (circumstances in which the relation would be true).

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies}_{37}(i)$$

- Implication rather than equivalence
 - easier to specify definitions in pieces
- $$\forall i. R(i) \wedge S(i) \rightarrow \text{Qualifies}_{37}(i)$$

6.034 – Spring 03 • 21

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i. P(i) \wedge Q(i) \rightarrow \text{Qualifies}_{37}(i)$$

- Implication rather than equivalence
 - easier to specify definitions in pieces
 - can't use the other direction
 - if you need it, write the equivalence

$$\forall i. (P(i) \wedge Q(i)) \vee (R(i) \wedge S(i)) \leftrightarrow \text{Qualifies}_{37}(i)$$

6.034 – Spring 03 • 22

Slide 8.7.22

However, written this way, we can't infer anything from knowing that the relation holds of some objects. If we need to do that, we have to write out the equivalence.

Slide 8.7.23

Okay. Let's start with a very simple rule. Let's say that an itinerary has the *Infant_Fare* property if the passenger is under age 2. We can write that as

$$\text{all } i, a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{Infant_Fare}(i)$$
Infant Fare

$$\forall i, a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

6.034 – Spring 03 • 23

Infant Fare

$$\forall i, a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

6.034 – Spring 03 • 24

Slide 8.7.24

It's not completely obvious that this is the right way to write the rule. For instance, it's useful to note that this is equivalent to saying

$$\text{all } i. (\text{exists } a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{Infant_Fare}(i)$$
Slide 8.7.25

This second form is clearer (though the first form is closer to what we'll use next, when we talk about rule-based systems). Note, also, that changing the implication to equivalence in these two statements makes them no longer be equivalent.

Infant Fare

$$\forall i, a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence

6.034 – Spring 03 • 25

Infant Fare

$$\forall i, a, p. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{ Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence
- What about $a < 2$?

6.034 – Spring 03 • 26

Slide 8.7.26

We just snuck something in here: $a < 2$. We'll need some basic arithmetic in almost any interesting logic domain. How can we deal with that? We saw, in the previous section, that adding arithmetic including multiplication means that our language is no longer complete. But that's the sort of thing that worries logicians more than practitioners.

Slide 8.7.27

In this domain we'll need addition and subtraction and greater-than. One strategy would be to axiomatize them in logic, but that's usually wildly inefficient. Most systems for doing practical logical inference include built-in arithmetic predicates that will automatically evaluate themselves if their arguments are instantiated. So if, during the course of a resolution proof, you had a clause of the form

$$P(a) \vee (3 < 2) \vee (a > 1 + 2)$$

it would automatically simplify to

$$P(a) \vee (a > 3)$$

Infant Fare

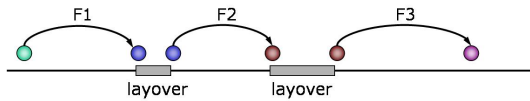
$$\forall i, a, p. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$$

$$\forall i (\exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence
- What about $a < 2$?
 - axiomatize arithmetic
 - build it in to theorem prover

$$P(a) \vee (3 > 2) \vee (a > 1 + 2) \Rightarrow P(a) \vee (a > 3)$$

6.034 – Spring 03 • 27

**Well-Formed Segment**

6.034 – Spring 03 • 28

Slide 8.7.28

Okay. Now, let's go to a significantly harder one. This isn't exactly a fare restriction; it's more of a correctness criterion on the flights within the itinerary. The idea is that an itinerary can be made up of multiple flight segments; each flight segment might, itself, be made up of multiple flights. In order for the itinerary to be well-formed, the flight segments are not required to have any particular relation to each other. However, there are considerable restrictions on a flight segment. One way to think about a flight segment is as a sequence of flights that you would do in one day (though it might actually last for more than one day if you're going for a long time).

Slide 8.7.29

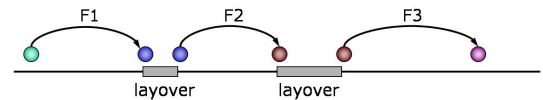
For a flight segment to be well-formed, it has to satisfy the following properties:

- The departure and arrival airports match up correctly
- The layovers (gaps between arriving in an airport and departing from it) aren't too short (so that there's a reasonable probability that the passenger will not miss the connection)
- The layovers aren't too long (so that the passenger can't spend a week enjoying him or herself in the city; we need to be sure to charge extra for that!)

So, let's work toward developing a logical specification of the well-formedness of a flight segment. A flight segment is a list of flights. So, we'll make a short detour to talk about lists in logic, then come back to well-formed flight-segments.

Well-Formed Segment

- Departure and arrival airports match up correctly
- Layovers aren't too short
- Layovers aren't too long



6.034 – Spring 03 • 29

Lists in Logic

- Nil : constant
- cons : function

6.034 – Spring 03 • 30

Slide 8.7.30

In the list of relations for the domain, we included a constant Nil and a function cons, without explanation. Here's the explanation.

Slide 8.7.31

We can make and use lists in logic, much as we might do in Scheme. We have a constant that stands for the empty list. Then, we have a function `cons` that, given any object and a list, denotes the list that has the object argument as its head (`car`) and the list argument as its tail (`cdr`).

So `cons(A, cons(B, Nil))` is a list with two elements, A and B.

Lists in Logic

- Nil : constant
- cons : function
- `cons(A, cons(B, Nil))` : list with two elements



6.034 – Spring 03 • 31

Lists in Logic

- Nil : constant
- cons : function
- `cons(A, cons(B, Nil))` : list with two elements
- $\forall x. \text{LengthOne}(\text{cons}(x, \text{Nil}))$

$$\forall l, x. l = \text{cons}(x, \text{Nil}) \rightarrow \text{LengthOne}(l)$$

$$\forall l. (\exists x. l = \text{cons}(x, \text{Nil})) \rightarrow \text{LengthOne}(l)$$



6.034 – Spring 03 • 32

Slide 8.7.32

We can also use the power of unification to specify conditions to assertions. So we can write **for all x** `lengthOne(cons(x, Nil))`, which is a more compact way of saying that every list that is equal to the cons of an element onto Nil has the property of being lengthOne.

Slide 8.7.33

Now that we know how to do some things with lists, we'll go back to the problem of ensuring that a flight segment is well-formed. This is basically a condition on all the layovers in the segment, so we'll have to run down the list making sure it's all okay.

Well-Formed Segment: Base Case

Define recursively, going down the list of flights



6.034 – Spring 03 • 33

Well-Formed Segment: Base Case

Define recursively, going down the list of flights

Any segment with 1 flight is well-formed

$$\forall f. \text{WellFormed}(\text{cons}(f, \text{Nil}))$$



6.034 – Spring 03 • 34

Slide 8.7.34

We can start with a simple case. If the flight segment has one flight, then it's well-formed. We can write this as **all f. WellFormed(cons(f, Nil))**.

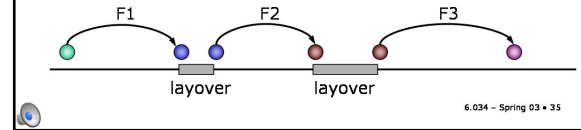
Slide 8.7.35

Now, let's do the hard case. We can say that a flight segment with more than one flight is well-formed if the first two flights are contiguous (end and start in the same airport), the layover time between the first two flights is legal, and the rest of the flight segment is well-formed.

Well-Formed Segment: Recursion

A flight segment with at least two flights is well-formed if

- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed

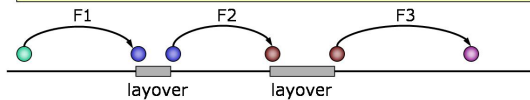


6.034 - Spring 03 • 35

Well-Formed Segment: Recursion

A flight segment with at least two flights is well-formed if

- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed

$$\forall f_1, f_2, r. \text{Contiguous}(f_1, f_2) \wedge \text{LegalLayover}(f_1, f_2) \wedge \text{WellFormed}(\text{cons}(f_2, r)) \rightarrow \text{WellFormed}(\text{cons}(f_1, \text{cons}(f_2, r)))$$


6.034 - Spring 03 • 36

Slide 8.7.36

In logic, that becomes

```
all f1, f2, r. Contiguous(f1, f2) ^ LegalLayover(f1, f2) ^
  WellFormed(cons(f2, r)) -> WellFormed(cons(f1, cons(f2, r)))
```

Slide 8.7.37

Note that we've invented some vocabulary here. Contiguous and LegalLayover are neither given to us as basic relations, nor the relation we are trying to define. We made them up, just as you make up function names, in order divide our problem into conquerable sub-parts.

Helper Relations

6.034 - Spring 03 • 37

Helper Relations

- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2. (\exists c. \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

6.034 - Spring 03 • 38

Slide 8.7.38

What makes two flights contiguous? The arrival airport of the first has to be the same as the departure airport of the second. We can write this as

```
all f1, f2. (exists c. Destination(f1, c) ^ Origin(f2, c))
  -> Contiguous(f1, f2)
```

Slide 8.7.39

Now, what makes layovers legal? They have to be not too short and not too long.

```
all f1, f2.
  LayoverNotTooShort(f1, f2) ^ LayoverNotTooLong(f1, f2) ->
    LayoverLegal(f1, f2)
```

Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \\ (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

6.034 – Spring 03 • 40

Slide 8.7.40

Let's say that passengers need at least 30 minutes to change planes.

That comes out fairly straightforwardly as

```
all f1, f2. (exists t1, t2.
  Arrival_Time(f1, t1) ^ Departure_Time(f2, t2) ^ (t2 - t1 > 30))
->
  Layover_Not_Too_Short(f1, f2)
```

Helper Relations

- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2. (\exists c. \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

- Layovers are legal if they're not too short and not too long

$$\forall f_1, f_2. \text{LayoverNotTooShort}(f_1, f_2) \wedge \text{LayoverNotTooLong}(f_1, f_2) \rightarrow \text{LayoverLegal}(f_1, f_2)$$

6.034 – Spring 03 • 39

Slide 8.7.41

This is a very simple version of the problem. You could imagine making this incredibly complex and nuanced. How long does it take someone to change planes? It might depend on: whether they're in a wheelchair, whether they have to change terminals, how busy the terminals are, how effective the inter-terminal transportation is, whether they have small children, whether the airport has signs in their native language, whether there's bad weather, how long the lines are at security, whether they're from a country whose citizens take a long time to clear immigration.

You probably wouldn't want to add each of these things as a condition in the rule about layovers. Rather, you would want this system, ultimately, to be connected to a knowledge base of common sense facts and relationships, which could be used to deduce an expected time to make the connection. Common-sense reasoning is a fascinating area of AI with a long history. It seems to be (like many things!) both very important and very hard.

Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \\ (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

- These are like the rules the airlines use, but it could involve all of common sense to know how long to allow someone to change planes

6.034 – Spring 03 • 41

Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \\ (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

6.034 – Spring 03 • 42

Slide 8.7.42

We'll continue in our more circumscribed setting, to address the question of what makes a layover not be too long. This will have an easy case and a hard case. The easy case is that a layover is not too long if it's less than three hours:

```
all f1, f2. (exists t1, t2.
  ArrivalTime(f1, t1) ^ DepartureTime(f2, t2) ^
  (t2 - t1 < 180)) -> LayoverNotTooLong(f1, f2)
```

Slide 8.7.43

Now, for the hard case. Let's imagine you've just flown into Ulan Bator, trying to get to Paris. And there's only one flight per day from Ulan Bator. We might want to say that your layover is not too long if there are no flights from here to your next destination that go before the one you're scheduled to take (and that have an adequately long layover).

```
all f1, f2 (exists o, d, t2.
  Origin(f2, o) ^ Destination(f2, d) ^ DepartureTime(f2, t2) ^
  ~ exists f3, t3. ( Origin(f3, o) ^ Destination(f3, d) ^
    DepartureTime(f3, t3) ^ (t3 < t2)
    ^ LayoverNotTooShort(f1, f3))) ->
  LayoverNotTooLong(f1, f2)
```

Of course, you can imagine all sorts of common-sense information that might influence this definition of LayoverNotTooLong, just as in the previous case.

We haven't been writing these definitions with efficiency in mind. In all likelihood, if we tried to put them into a regular theorem prover, we would never get an answer out. In the next segment of material, we'll see how to use a restricted version of first-order logic to get fairly efficient logical programs. And we'll continue this example there.

Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) \\ (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

- A layover is also not too long if there was no other way to make the next leg of your journey sooner

$$\forall f_1, f_2. (\exists o, d, t_2. \text{Origin}(f_2, o) \wedge \text{Destination}(f_2, d) \wedge \\ \text{DepartureTime}(f_2, t_2) \wedge \\ \neg \exists f_3, t_3. (\text{Origin}(f_3, o) \wedge \text{Destination}(f_3, d) \wedge \\ \text{DepartureTime}(f_3, t_3) \wedge (t_3 < t_2) \wedge \\ \text{LayoverNotTooShort}(f_1, f_3))) \\ \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

6.034 – Spring 03 • 43