Strategies for Deduction

1 Introduction

We have seen how we can represent some aspects of the world in FOPC. Now, how can we reason with it?

Remember that in logic, what we gethave is a deductive procedure. *Deduction* is the process of inferring new assertions from existing ones using inference rules.

Remember that, the way we will use this is to have a collection of sentences called a *knowledge base*, representing what we know. Then we will code up our problem as something to be deduced from the knowledge base. The sentence whose validity we are trying to establish is called the *goal*, or the *query*. If the sentence is already in the KB, that's fine. Otherwise, we can try proving it from other things in the KB using a logical inference procedure.

In general, the paradigm of reasoning is sometimes called *deductive information retrieval.*

Note that we haven't really shown how to do this yet for interesting problems involving actions, although we suggested an idea when looking at propositional calculus. We'll return to this issue later. In the meantime, we will simply assume that problems can be coded up as sentences to be proved.

Actually, as we'll see in a minute, we typically want to do just a little bit more than just find out if something is true. We discuss what in a minute.

First, we need to know what some valid inference rules might be. Basically, we have the rules from propositional calculus, but we need to add some to deal with quantifiers. Let's look at a possible list.

2 Inference

Inference is the process of deriving conclusions from premises using rules of inference. A rule of inference consists of

- a set of sentences called the premises
- a set of sentences called the conclusions

Whenever we have sentences that match the premises of the rule we may infer sentences matching the conclusions.

Here are some examples of rules of inference. Note that we write them using Greek letters to stand for sentences. The Greek letters are not part of the vocabulary our logic; rather, they are a way of talking about the sentences of our logic:

- Modus ponens (MP):

$$\frac{\begin{array}{l}\alpha \rightarrow \beta \\ \alpha\end{array}}{\beta}$$

- Modus tolens (MT):

$$\frac{\begin{array}{l}\alpha \rightarrow \beta \\ \neg\beta\end{array}}{\neg\alpha}$$

- "And" elimination (AE):

$$\frac{\alpha \wedge \beta}{\begin{array}{l}\alpha \\ \beta\end{array}}$$

- "And" introduction (AI):

$$\frac{\begin{array}{l}\alpha \\ \beta\end{array}}{\alpha \wedge \beta}$$

- "Or" introduction (OI):

$$\frac{\alpha}{\alpha \vee \beta}$$

- Simple resolution:

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

- Double negation elimination:

$$\frac{\neg\neg\alpha}{\alpha}$$

- Negation and Quantifiers:

$$\frac{\neg\ \forall \mathrm{x}\ \alpha}{\exists\ \mathrm{x}\ \neg\alpha}$$

$$\frac{\neg\ \exists \mathrm{x}\ \alpha}{\forall \mathrm{x}\ \neg\alpha}$$

Now there are some other rules for quantifiers. For example, we'd like to say that, "if something is true for everything, it is true for any given particular thing." To make it easy to express this, we introduce the notion of a *substitution*. A substitution is just a list variables paired with expressions. For example,

{x/Jan, y/Pat}

is a substition in which Jan is to be substituted for x, and Pat for y. In LISP-ish notation, we might represent this as a list of lists (or of dotted pairs):

((x Jan) (y Pat))

We sometimes refer to the elements of a substitution as *bindings*. (More on this idea later.)

We can write a substitution after an expression to mean applying the substitution to that expression. For example

Loves(x,y){x/Jan, y/Pat}

is equivalent to

Loves(Jan,Pat)

In general, we write $\alpha\sigma$ (sigma) to mean the result of applying the substitution σ to the expression α.

Now it is easier to express some other rules:

- Universal instantiation (UI):

  $$\frac{\forall \text{x}\ \alpha}{\alpha\{\text{x/C}\}}$$

  where C is any term that doesn't contain the variables x. Usually, we'd just want to substitute a constant. For example, if the sentence were

  ∀x ∃y Hates(x,y)

  it is okay to conclude

  ∃y Hates(Jan,y)

  It is also okay to conclude

  ∃y Hates(Father-of(Jan),y)

  Now, some folks allow substituting some terms with variables, if we are careful. E.g., the following is okay:

  ∃y Hates(Father-of(z),y)

  But it wouldn't be okay to conclude

  ∃y Hates(Father-of(y),y)

- Existential instantiation (EI):

  $$\frac{\exists \text{x}\ \alpha}{\alpha\{\text{x/N}\}}$$

  where N is a new term, i.e., a term not mentioned in any other sentence.

  The idea is that N may or may not refer to something we have talked about already, but we want to be careful that we don't use a symbol that may have already been spoken for. For example, if we have

  ∃x President(x,US)

  i.e., the US has a president, we are allowed to say

  President(Somejoe,US)

  as long as we haven't seen the symbol "Somejoe" before. If we weren't careful, we might accidently say

  President(Yeltin,US)

  which would probably be wrong. (Why? Only because somewhere else we might have said

  President(Yeltin,Russia)

  If we later find out that the president is Clinton, we can add the formula

  Somejoe=Clinton

Now suppose we have the following:

$\forall$c $\exists$x President(x,c)

It would be wrong to replace x with a constant here. Why? But there is a way! We can replace x with a function that maps countries to their leaders. For example, suppose we have a function "Leader", so that Leader(US) denotes Clinton, and Leader(Russia) denotes Yeltin, etc. Then we could say

$\forall$c President(Leader(c),c)

Now the problem is, we might not know what to call this function. So let's just make up a new function name! As long as no one has used this function symbol before, we always have the possibility to assign it the right semantics.

So in general

$$\frac{\forall \mathrm{y}\ \exists \mathrm{x}\ \alpha}{\forall \mathrm{y}\ \alpha\{\mathrm{x}/\mathrm{F}(\mathrm{y})\}}$$

as long as F is a new function.

In the most general case:

$$\frac{\forall \mathrm{y}_1,\ldots,\mathrm{y}_n\ \exists \mathrm{x}\ \alpha}{\forall \mathrm{y}_1,\ldots,\mathrm{y}_n\ \alpha\{\mathrm{x},\mathrm{F}(\mathrm{y}_1,\ldots \mathrm{y}_n)\}}$$

where F is a new function of all the universal variables.

Okay, so these appear to be some valid inference rules. However, we have not shown an effective way to use them.

- We would like an inference procedure in which
  - Everything that we deduce is logically implied. (*soundness*)
  - Everything that is logically implied is derivable. (*completeness*)
  - We can do this efficiently.
- The various inference rules we have seen so far are sound. Are they complete? It is certainly not obvious that our rules thus far will guarantee us a proof if there is one.
- It turns out that there are effective procedures for deduction in FOPC. (This is Godel's Completeness theorem; as most of you probably know, a very minor extension of our language, enough to allow arithmetic, loses this "completeness" property; that's Godel's Incompleteness Theorem.) Actually, Godel didn't show what such a procedure is. We will though.
- Efficiency is another matter. In general, effective schemes tend to be inefficient. Note, for example, that just applying universal instantial lets you stick in any ground term you like. So, using this rule unconstrained would be computationally very expensive.
- Think a bit about what people are capable of. We seem to be able to do this sort of reasoning only in a very limited way.

  At the same time, we don't break down altogether if we believe some contradictory things. What reasoning we do in "ordinary" situations, while not always correct, we can do pretty quickly, and we can reason over a *very* expressive language. So, it seems likely that reasoning to support ordinary action will not be deductively correct.

  Nevertheless, we will look at various purely deductive schemes, representing various trade-offs. We might want a good general scheme, or some fairly efficient but perhaps somewhat more limited scheme. For example, we might restrict what we can say a bit to get a faster algorithm.

3 Unification

Before we examine deductive inference procedures, let us think a bit about how we might use such a procedure to do deductive retrieval. Remember, the idea is to respond to a query (or, as we say more proof-theoretically, a "goal") by trying to prove it from sentences in a KB. For example, suppose

∀ x Loves(x,x)

is in the DB. (I.e., everyone loves his-or-herself.) Now suppose our query is

Loves(Jan,Jan)

We would like our deductive retriever to respond that, "yes", this query can be "found" in the KB, because we can prove it from what is in there. (How? E.g., by using Universal Instantiation). However, consider what we would like to get back for the query

∃ z Loves(z,Jan)

If either "Loves(Pat,Jan)" or "∀ x Loves(x,x)" is in the KB, our deduction procedure should tell us that this goal is "in the KB". But that really isn't enough. We would like to know *who* loves Jan. In fact, we would like to ask something more, namely, tell us everyone who loves Jan. So, what we need is some notion of an answer that is more than just whether the query is truth.

Intuitively, we would like to know what "assignment(s)" of the variable z would make the query true.

The notion of "assignment" that we will use is called *unification*. (It has nothing to do with Germany or Sun Yung Moon.) The idea is to find a substitution, which, when applied to two formulas, yields the same formula. For example,

Loves(x,x)/{x/Jan,z/Jan}
Loves(z,Jan)/{x/Jan,z/Jan}

produce the same result, namely "Loves(Jan,Jan)". Hence we refer to {x/Jan,z/Jan} as a *unifier* of these expressions, and say that they can be unified.

Note that it may be possible to unify two expressions in more than one way. For example

Loves(x,Jan)
Loves(z,Jan)

can be unified by {x/Pat,z/Pat} and by {x/Lynn,z/Lynn} (and in fact, by {x/Pat,z/Pat,w/George}). However, these unifiers seem rather ad hoc. In contrast, the unifier {x/z} seems more satisfying. More precisely, we can say that what we are interested in is the *most general unifier*. The most general unifier makes the least commitment possible. It is that unifier for which, after it is applied, produces an expression for which there is always another substitution that will produce the expression that any other unifier applied to the original expression will give us. For instance, after substituting z for x, we can always substitute Pat for z, and get the same result that applying the unifier {x/Pat,z/Pat} does.

Thus

Formula 1	Formula 2	Unifier
Loves(x,Jan)	Loves(Lynn,y)	{x/Lynn,y/Jan}
Loves(x,Jan)	Loves(x,Pat)	tilt!
Loves(x,Jan)	Loves(Pat,x)	tilt!
Loves(Jan,Pat)	Loves(Jan,Pat)	{ }
Loves(x,Father(x))	Loves(Lynn,y)	{x/Lynn,y/Father(Lynn)}
Loves(x,Father(x))	Loves(y,y)	tilt!

With this notion of unification in hand, we can say that what we want from deductive retrieval is to compute MGUs of the goal supported by the KB.

- By substituting the goal with the MGU, we can reconstruct the item we have found.
- We can still have many MGUs returned for a given query. E.g., if the query is "Loves(x,Pat)" and the KB contains "Loves(Jan,Pat)" and "Loves(Lynn,Pat)", each of these will cause a different MGU to be returned. (In general, there may be many different ways that the same goal can be deduced from the KB.)

  Thus, we need some strategy for dealing with the possibility of lots of answers to a query. One possibility is just return a list of all matches. Another is to just return the first answer; then retrieve another if later we decide we don't like the first one. (I.e., we implement the retriever so it returns a *stream*.)
- It is possible that an item matches the goals with the "empty" MGU. Thus, we have to be careful to distinguish success with no bindings from failure.
- Unification is sensitive to variable names. Thus we need to be very careful that there are no accidental name conflicts. One way to do this is to rename the variables in a formula before unification is attempted.

4 Backward Chaining

We have been assuming a very simple form of deductive retrieval, in which we find items in the KB that unify with the query.

We can start to add more deduction, but to do so, it is convenient to assume that everything in the KB is in a standard format. In particular, first note that we can assume that all the existential quantifiers have been eliminated. We can always do this by skolemization (i.e., existential elimination).

Note then that all quantifiers in the KB are universal, so we can drop quantifiers altogether. (BTW, we won't have quantifiers in the query, either, but it is probably right to say that the variables are being treated as existentially quantified, since we will be computing MGUs.)

The second assumption is that everything is either an atomic sentence or of the form $P_1 \wedge P_2 \wedge \ldots \wedge \rightarrow Q$. Remember, these are Horn clauses. It turns out not everything can be expressed this way, but lots of what we want can. (We can allow more generality in the queries.)

Now, one form of reasoning we can do is to use facts with implications in them as rules. That is, suppose we have facts in our knowledge base like the following:

(1) ∀x (Dog(x) → Mammal(x))
(2) ∀x (RR(x) → Dog(x))

- Suppose RR(Fido) were also in our knowledge base.

- Now suppose we want to know if Mammal(Fido).
- Let's look for implicational facts like the one above. That is, let's find an implication whose consequent unifies with our goal. E.g., "Mammal(Fido)" unifies with "Mammal(x)" in fact (1), with {x/Fido} as the unifier.
- Now let's try to prove the antecedent, after substitution with the unifier. In our example, we'll try to prove "Dog(Fido)".
- This still isn't in the KB. So let's look for another "rule".
- We unify "Dog(Fido)" with one side of fact (2); then we continue by trying to prove "RR(Fido)".
- This is easy; it's in the KB. So we're done.

The process we just described is called *backward chaining*. Backward chaining uses implicational facts to set up *subgoals* that have to be proven along the way.

How do we justify this process? It's a kind of combination of MP and UI. Note that it is an especially efficient application of these rules, because the variable instantiation is being done intelligently.

- Variable naming is especially important when we do chaining. For example, the same rule can be used more than once in a derivation.
- With chaining, we can have an infinite number of answers, (How? Imagine any kind of recursive rule, like

  $$\text{Person(x)} \rightarrow \text{Person(Father(x))}$$

  And suppose our query is "Person(y)". Now, if there are any people, there are an infinite number of them. Care must be exerted here.

5 Forward Chaining

Consider the our example above where we deduced that Fido was a mammal by virtue of his being a dog, by virtue of his being a RR. Now suppose we had lots of implications like these in the KB: e.g., cats are mammals, pigs are mammals, etc., and Siamese are cats, Persian-cats are cats, etc. Now when we try to deduce Fido is a mammal, we might first try to prove that Fido were a cat, and try to do this by trying to prove that he were a Siamese. Of course, we would fail. But then we might try to prove that Fido were a kind of mouse, etc. We would eventually deduce the right thing, but only after a lot of wasted effort.

If we are doing this kind of computation a lot, we can speed things up by having already put in the KB that Fido is a mammal. However, when we learn that Simba is a RR, we would like to have Mammal(Simba) automatically entired in the KB for us.

This is called *forward chaining*. We use the same kind of implication, but this time, when we add a fact to the KB, we check to see if the LHS of an implication is met, and, if so, we assert the RHS in the KB.

- This is just an efficiency consideration. It's really a standard speed/storage trade-off. (Backward chaining is in some sense more general, though: You can't use unconstrained forward chaining in the case where there are an infinite number of implication applications. With backward chaining, we just have to be careful about not generating big answers all at once.)

- We might try to have some slightly better deduction strategy instead. For example, instead of blind backward chaining, maybe we can look for facts that involve terms in our goal and use them. (E.g., in our example, look for facts about Fido.) But there might be just as many of these! Also, using them in effect amounts to something similar to forward chaining.
- We sometimes call implications used for BWC *if-neededs* and those used for FWC, *if-addeds*.
- Implementationally, we need to tell our system which implications to use in which direction. Sometimes folks use a different implication arrow for each case. For example, instead of $p \to q$, we write $q \leftarrow p$.
- Note that forward chaining changes the KB, so this complicates strategies that might requiring provisionally changing the KB.
- Combining forward chaining backward chaining is fine. But we need to be careful that we don't handicap our system this way. E.g., if we used **Dog(x)** $\to$ **Mammal(x)** only for forward chaining in the example above, we couldn't conclude **Mammal(Fido)** from **RR(Fido)**.

6 Resolution

Is backward/forward chaining good enough for everything we want to do? Suppose I know that, if Jan goes to the party, Pat will go to the party, but also, if Jan doesn't go to the party, Pat will go to the party. Now, I want to know if Pat will go to the party I.e., we have the following in our KB:

GoParty(Jan) $\to$ GoParty(Pat)
$\neg$GoParty(Jan) $\to$ GoParty(Pat)

and our goal is GoParty(Pat). This is obviously true, but, perhaps surprisingly, nothing we have discussed so far will establish this fact! I.e., our inference methods are not complete.

But there *is* a deductive method that is complete, and in fact, is as efficient as any complete method of inference. This is called *resolution*, and it's due to J. Robinson.

Let's look first at a simple version of this idea in propositional calculus. The basic observation is that we can deduce the following:

$$\frac{\begin{array}{c}\mathrm{P} \vee \mathrm{Q} \\ \neg\mathrm{P} \vee \mathrm{R}\end{array}}{\mathrm{Q} \vee \mathrm{R}}$$

This follows from the laws of logic; semantically, note that if P is false, Q is true, and if P is true, R is true; so no matter what, Q $\vee$ R.

Intuitively, we can think about this as joining the constraints of our facts together.

Now, suppose we have a bunch of facts of this form. For example suppose we know the following things:

P $\vee$ Q
$\neg$P $\vee$ R
$\neg$Q $\vee$ R

and suppose we want to know if

R $\vee$ S

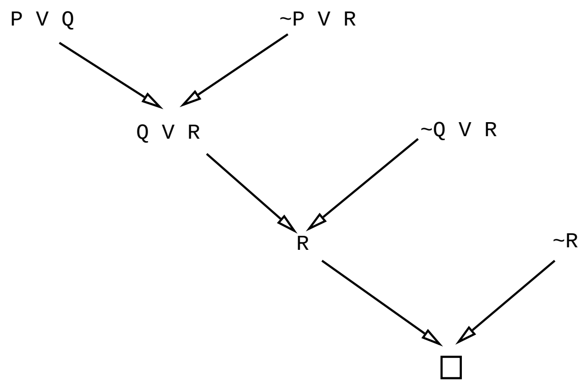
- First, let's negate what we want to prove, and add it to what we know. In this case, we add

$\neg$R
$\neg$S

- Now, let's try to arrive at a contradiction by rubbing pairs of elements in our KB together. Here's one sequence of deductions we can come up with:

$$\begin{array}{l} P \lor Q \\ \neg P \lor R \\ \hline Q \lor R \\ \neg Q \lor R \\ \hline R \\ \neg R \\ \hline \square \end{array}$$

We often diagram these deductions this way:



This process is called *resolution*. To apply resolution to predicate calculus, we just need to be careful about how we treat variables. More specifically:

- Assume that our KB contains nothing but *disjunctions of literals*; a *literal* being either an atomic formula or the negation of an atomic formula. We call such a disjunction a *clause*, and say that the KB is in *clausal form*.
- Note that this means:
  - no implications
  - no existential variables
  - no negation, except in literals
  - no conjunctions
  - no duplicates of literals
- Since our sentence format is so restricted, we can write clauses in an abbreviated way. E.g., if we want to express the fact

  $\forall$x,y (P(x) $\lor$ $\neg$Q(x,A))

  we will write

  {P(x), $\neg$Q(x,A)}

  (This is just set notation.)
- We negate what we are trying to prove and add it to the KB, again being sure it too is in clausal form.

- Find two clauses we can *resolve* together. This means that one clause contains a literal that can unify with the *negation* of the other. For example, if, in addition to the clause above, we have

  {R(w,z), Q(z,w)}

  we can unify Q(z,w) and Q(x,A), producing {x/z,w/A}.

- Form a new clause by adding the remaining disjuncts of the resolvants, and substituting as per the unifier. E.g.:

  {P(z), R(A,z)}

- Keep going until you deduce the "empty clause". Then we know the KB is inconsistent.

Let's look at some examples. Consider the case that we said was problematic for chaining. Recall that our knowledge base contained the following:

GoParty(Jan) → GoParty(Pat)
¬GoParty(Jan) → GoParty(Pat)

and that we want to prove

GoParty(Pat)

We can convert the implications to disjunctions, and then negate the goal and add it to the KB, converting all to clausal form. Then we can resolve as follows:

{GoParty(Jan),GoParty(Pat)}
{¬GoParth(Jan),GoParty(Pat)}

{GoParty(Pat)}
{¬GoParty(Pat)}

□

So we have proven the goal very easily.

For another example, let us prove something that we can prove by backward chaining. Thus, suppose our KB were the following:

Dog(x) → Mammal(x)
Dog(Fido)

and that the goal were

Mammal(Fido)

First, we would have to transform the KB into clausal form. To do this, let us recall that $\mathbf{P} \rightarrow \mathbf{Q}$ is equivalent to $\neg\mathbf{P} \vee \mathbf{Q}$:

{¬Dog(x), Mammal(x)}
{Dog(Fido)}

and negating the goal gives us

{¬Mammal(Fido)}

So we have a quick proof of this too:

$$\begin{array}{l} \{\neg\text{Dog(x), Mammal(x)}\} \\ \{\text{Dog(Fido)}\} \\ \hline \{\text{Mammal(Fido)}\} \\ \{\neg\text{Mammal(Fido)}\} \\ \hline \square \end{array}$$

6.1 Some highlights of resolution

- In an implementation, we probably won't really want to add the negation of the goal to the real KB, or at least, we would want to be sure to clean up when we're finished.
- Note that, unlike backward chaining, etc., the goal is interpreted just like any item in the KB. For example, unscoped variables would be treated universally rather than existentially.
- Indeed, our goal and the elements of our KB can be any sentences at all, as long as we can express them in clausal form.
- Good news: Resolution is complete in the sense that, if a goal follows from a KB, resolution is guaranteed to find a proof. (We sometimes call the idea of completeness *refutation completeness*. Actually, resolution is not complete in the sense that every clause logically implied by every KB can be produced by resolution. For example, the tautology {P,¬P} cannot be constructed by resolution from the empty KB. But this is not really relevant to our concerns.)
- Bad news: Resolution doesn't work for sentences involving equality. For example, if we have

  P(A)
  A=B

  then

  P(B)

  clearly follows. But resolution can't prove this. This can be fixed by another inference rule called *paramodulation*, and a lot of less general, more efficient hacks.
- More good news: Resolution is as fast as any complete algorithm.
- Bad news: Whether or not something follows from a set of premises is not decidable. I.e., if neither α nor $\neg\alpha$ is implied by the KB, there is no way to determine this. (This question is partially decidable for FOPC; it is not even recursively enumerable for second order logic.)
- More bad news: Resolution is hopelessly slow (hyperexponential in the number of literals).
- There are various strategies to help make things somewhat more efficient, but we won't concern ourselves with them here.

7 Clausal Form

A reservation you might have about resolution is that it requires our KB to be in clausal form. However, we can algorithmically convert any sentence of FOL into clausal form. Thus, resolution is a completely general procedure.

Here is a method for converting an arbitrary sentence into clausal form:

1. Eliminate implications by substituting logical equivalences. Recall that

$\alpha \rightarrow \beta$

is equivalent to

$\neg\alpha \vee \beta$

2. Distribute negations over other logical operations:
   - $\neg\neg\alpha$ is replaced by α
   - $\neg(\alpha \wedge \beta)$ is replaced by $\neg\alpha \vee \neg\beta$
   - $\neg(\alpha \vee \beta)$ is replaced by $\neg\alpha \wedge \neg\beta$
   - $\neg\forall$x α is replaced by $\exists$x $\neg\alpha$
   - $\neg\exists$x α is replaced by $\forall$x $\neg\alpha$

3. Rename variables so that each quantifier has a unique variable.

4. Eliminate existential quantifiers by *skolemization*, i.e., existential elimination. Recall that
   - existential variables not in the scope of any universal quantifiers can be replaced by "new" constants (so-called *Skolem constants*)
   - existential variables in the scope of some universal quantifiers can be replaced by "new" functions of all the universally quantified variables (so-called *Skolem variables*.

   For example,

   $\forall$x $\exists$y President(x,y)

   would become

   $\forall$x President(x,Newfn(x))

5. Drop universal quantifiers. (They have become redundant.)

6. Distribute disjunctions over conjunctions. That is, we take advantage of the fact that

   $\alpha \vee (\beta \wedge \gamma)$

   is logically equivalent to

   $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

   The resulting format is called *conjunctive normal form.*

7. Turn every conjunct into a clause (sanctioned by AE (AND Elimination)), and rewrite in clause notation. E.g., our sentence above would correspond to the following clauses:

   $\{\alpha, \beta\}$
   $\{\alpha, \gamma\}$

8. Rename variables so that no variable appears in more than one clause. This process is called *standardizing the variables apart.* It is justified because

   $\forall$x $(\alpha \wedge \beta)$

   is equivalent to

   $\forall$x $\alpha \wedge \forall$x β

   which is equivalent to

   $\forall$x $\alpha \wedge \forall$x' β\{x/x'\}

where x' is a brand new variable.

Let's work through a big example. Our initial sentence is the following:

$$\forall x\ (\forall y\ \text{Born-in}(x,y)) \rightarrow \neg(\forall y\ \text{Moving-to}(x,y) \rightarrow \text{Citizen}(x,y))$$

1. Eliminate implications:

   $$\forall x\ \neg(\forall y\ \text{Born-in}(x,y)) \vee \neg(\forall y\ \neg\text{Moving-to}(x,y) \vee \text{Citizen}(x,y))$$

2. Distribute negations:

   $$\forall x\ (\exists y\ \neg\text{Born-in}(x,y)) \vee (\exists y\ \text{Moving-to}(x,y) \wedge \neg\text{Citizen}(x,y))$$

3. Rename variables:

   $$\forall x\ (\exists y\ \neg\text{Born-in}(x,y)) \vee (\exists z\ \text{Moving-to}(x,z) \wedge \neg\text{Citizen}(x,z))$$

4. Eliminate existential quantifiers:

   $$\forall x\ \neg\text{Born-in}(x,\text{Country1}(x)) \vee (\text{Moving-to}(x,\text{Country2}(x)) \wedge \neg\text{Citizen}(x,\text{Country2}(x)))$$

5. Drop universal quantifiers:

   $$\neg\text{Born-in}(x,\text{Country1}(x)) \vee (\text{Moving-to}(x,\text{Country2}(x)) \wedge \neg\text{Citizen}(x,\text{Country2}(x)))$$

6. Distribute disjunctions:

   $$(\neg\text{Born-in}(x,\text{Country1}(x)) \vee \text{Moving-to}(x,\text{Country2}(x))) \wedge$$
   $$(\neg\text{Born-in}(x,\text{Country1}(x)) \vee \neg\text{Citizen}(x,\text{Country2}(x)))$$

7. Turn into clauses:

   $$\{\neg\text{Born-in}(x,\text{Country1}(x)),\ \text{Moving-to}(x,\text{Country2}(x))\}$$
   $$\{\neg\text{Born-in}(x,\text{Country1}(x)),\ \neg\text{Citizen}(x,\text{Country2}(x))\}$$

8. Standardize the variables apart:

   $$\{\neg\text{Born-in}(x1,\text{Country1}(x1)),\ \text{Moving-to}(x1,\text{Country2}(x1))\}$$
   $$\{\neg\text{Born-in}(x2,\text{Country1}(x2)),\ \neg\text{Citizen}(x2,\text{Country2}(x2))\}$$

Note that expressions in clausal form are hard to make sense of. I.e., they are bad for us humans, even if resolution likes them this way.

Let's consider one more example of using resolution to prove something. What we'll prove is the following fact:

> A relation that is symmetric, transitive and defined for the entire domain is reflexive.

In FOPC, the assumptions amount to the following:

$$\forall x,y\ R(x,y) \rightarrow R(y,x)$$
$$\forall x,y,z,\ R(x,y) \wedge R(y,z) \rightarrow R(x,z)$$
$$\forall x\ \exists y\ R(x,y)$$

and the goal is

$\forall x\ R(x,x)$

Now, let's convert the KB to clausal form:

1. $\{\ \neg R(x,y),\ R(y,x)\}$
2. $\{\ \neg R(x,y),\ \neg R(y,z),\ R(x,z)\}$
3. $\{\ R(x,F(x))\}$

and add the negated goal:

4. $\neg R(A,A)$

Then we have the following deduction:

5. $\{\ \neg R(A,y),\ \neg R(y,A)\ \}$ (from 2 and 4)
6. $\{\ \neg R(F(A),A)\}$ (from 3 and 5)
7. $\{\ \neg R(A,F(A))\ \}$ (from 1 and 6)
8. $\Box$ (from 3 and 7)