# How to Convert a Formula to CNF

**Declarative Methods, CS 325/425 - Prof. Jason Eisner**

This algorithm corresponds exactly to the one you saw on the lecture slides, but this presentation gives a somewhat different perspective along with some further discussion.

In Homework 1, you'll get to convert some formulas to CNF by hand. (Or if you want, you could implement the algorithm below and implement an automatic converter!)

## Representing Formulas of Propositional Logic

In lecture, we allowed 7 kinds of propositional formulas in our little language. It might be helpful to think about how you would implement formulas in a language like Java. You'd want a recursive datatype similar to trees, roughly like this:

```
abstract class Formula { }
class Variable extends Formula { String varname; }
class AndFormula extends Formula { Formula p; Formula q; }  // conjunction
class OrFormula extends Formula { Formula p; Formula q; }  // disjunction
class NotFormula extends Formula { Formula p; } // negation
class ImpliesFormula extends Formula { Formula p; Formula q; } // if-then
class EquivFormula extends Formula { Formula p; Formula q; }
class XorFormula extends Formula { Formula p; Formula q; }
```

Thus, if φ is an instance of `Formula`, we can ask whether it is an instanceof the subclass `AndFormula`. If so, we can look at its two conjuncts, `P` and `Q`, which are themselves instances of `Formula`.

A CNF formula is a restricted special case. It is a conjunction of "clauses," each of which is a disjunction of "literals," each of which is either a variable or a negated variable.

## Converting to CNF

Now here's a routine to convert any formula to CNF.

```
CONVERT(ϕ):   // returns a CNF formula equivalent to ϕ

// Any syntactically valid propositional formula ϕ must fall into
// exactly one of the following 7 cases (that is, it is an instanceof
// one of the 7 subclasses of Formula).

If ϕ is a variable, then:
   return ϕ.
   // this is a CNF formula consisting of 1 clause that contains 1 literal

If ϕ has the form P ^ Q, then:
   CONVERT(P) must have the form P1 ^ P2 ^ ... ^ Pm, and
   CONVERT(Q) must have the form Q1 ^ Q2 ^ ... ^ Qn,
   where all the Pi and Qi are disjunctions of literals.
   So return P1 ^ P2 ^ ... ^ Pm ^ Q1 ^ Q2 ^ ... ^ Qn.

If ϕ has the form P v Q, then:
   CONVERT(P) must have the form P1 ^ P2 ^ ... ^ Pm, and
   CONVERT(Q) must have the form Q1 ^ Q2 ^ ... ^ Qn,
   where all the Pi and Qi are dijunctions of literals.
   So we need a CNF formula equivalent to
      (P1 ^ P2 ^ ... ^ Pm) v (Q1 ^ Q2 ^ ... ^ Qn).
```

```
   So return (P1 v Q1) ^ (P1 v Q2) ^ ... ^ (P1 v Qn)
          ^ (P2 v Q1) ^ (P2 v Q2) ^ ... ^ (P2 v Qn)
            ...
          ^ (Pm v Q1) ^ (Pm v Q2) ^ ... ^ (Pm v Qn)

If ϕ has the form ~(...), then:
  If ϕ has the form ~A for some variable A, then return ϕ.
  If ϕ has the form ~(~P), then return CONVERT(P).           // double negation
  If ϕ has the form ~(P ^ Q), then return CONVERT(~P v ~Q).  // de Morgan's Law
  If ϕ has the form ~(P v Q), then return CONVERT(~P ^ ~Q).  // de Morgan's Law

If ϕ has the form P -> Q, then:
  Return CONVERT(~P v Q).   // equivalent

If ϕ has the form P <-> Q, then:
  Return CONVERT((P ^ Q) v (~P ^ ~Q)).

If ϕ has the form P xor Q, then:
  Return CONVERT((P ^ ~Q) v (~P ^ Q)).
```

See why this recursive algorithm works? It must return a correct answer, assuming it doesn't recurse forever.

(You could prove that it won't recurse forever. The idea is to use induction on the depth of the formula φ, measuring depth in a particular way.)

## Using Switching Variables to Keep the Converted Formula Small

Unfortunately, the case of `P v Q` above is very bad. If `CONVERT(P) v CONVERT(Q)` is a formula of size O(m+n), the result of `CONVERT(P v Q)` can be a formula of size O(mn), which is in general much worse.

Moreover, m and n may already be large, since P and Q may themselves contain v. Unfortunately this means that `CONVERT(ϕ)` may be exponentially larger than φ: consider the case where

```
   ϕ = (P1 ^ P2) v (Q1 ^ Q2) v (R1 ^ R2) v ...
```

Thus, for the `P v Q` case in the above pseudocode, it is preferable to use the above naive method only if m=1 (or n=1).

If m > 1, then you can use this alternative method: let Z be a new variable that does not appear in φ, and return `CONVERT((Z -> P) ^ (~Z -> Q))`. This formula is satisfiable if and only if φ is satisfiable, since any assignment must set Z to be either true or false, and then the formula accordingly requires the assignment to satisfy either P or Q, respectively.

This recursive call to `CONVERT` will end up calling `CONVERT(~Z v P)` and `CONVERT(Z v Q)`, each of which will trigger only the m=1 case since ~Z and Z are literals.

As a result, if φ starts out with only v, ^, ~, then `CONVERT(ϕ)` is at worst *quadratically* larger than φ (see lecture slides for an intuition as to why, and try to prove it!). So even though this alternative technique does result in a formula with additional variables, it keeps the total formula length far smaller.

*Note:* We still have problems with `P <-> Q` and `P xor Q`, since our pseudocode replaces these with expressions that are more than twice as long. That can lead to exponential blowup if P and Q themselves contain `<->` and `xor`. You can avoid this problem by using the Tseitin transformation (see below).

## A More Efficient Representation for CNF Formulas

The pseudocode above was written as if `CONVERT` returns a `Formula` instance that just happens to be in CNF.

But it would be more efficient for `CONVERT` to return an instance of the following `CNFFormula` class, which can *only* represent CNF formulas but does so more compactly, taking advantage of their restricted structure:

```
class CNFFormula { Vector clauses; }
class Clause { Vector literals; }
class Literal { String varname; boolean is_negated; }
```

Then a code fragment like this

```
  If ϕ has the form P ^ Q, then:
      CONVERT(P) must have the form P1 ^ P2 ^ ... ^ Pm, and
      CONVERT(Q) must have the form Q1 ^ Q2 ^ ... ^ Qn,
      where all the Pi and Qi are disjunctions of literals.
      So return P1 ^ P2 ^ ... ^ Pm ^ Q1 ^ Q2 ^ ... ^ Qn.
```

would come out looking like this:

```
  If ϕ has the form P ^ Q, then:
     return new CNFFormula(concatenate(CONVERT(P).clauses, CONVERT(Q).clauses))
```

assuming that `CNFFormula` has a constructor that takes a list of clauses.

## Building up CNF Formulas Directly

Here is another way of thinking about this whole thing. Forget about `Formula` and just do everything with `CNFFormula`.

To convert φ = `(X ^ Y) v ~Z` to CNF, we would just construct φ *as a `CNFFormula`*, using `CNFFormula` constructors as follows:

```
  ϕ = CNFFormula(or, CNFFormula(and, CNFFormula("X"), CNFFormula("Y"))
                       CNFFormula(not, CNFFormula("Z")))
```

Every intermediate step returns a formula that has already been converted to CNF. So the user never sees a non-CNF formula at all, except for the non-CNF formula that is implicit in the nested constructor calls above.

The `CNFFormula` constructors have to handle the same 7 cases as are in the pseudocode above, and they do it using exactly the same methods.

```
CNFFormula(String varname) { make a 1-clause formula with 1 literal }

CNFFormula(Operator op, CNFFormula P, CNFFormula Q) {
  if op is the "and" operator, then:
     return the concatenation of P and Q
         (each of which is a vector of clauses)
  if op is the "or" operator, then:
     ...
```

and so on, going through the 7 cases from above.

Note that when `CNFFormula(and,P,Q)` combines P and Q, it doesn't have to recursively *convert* them to `CNFFormulas` as in the previous approach, because in this approach they are already `CNFFormulas`, having been converted already at the time they were built.

## Converting to DNF

If you take our `CONVERT` routine above, and replace each "and" operator with "or" and vice-versa, you wil get a routine for converting to DNF instead of to CNF. Now, as remarked in lecture, it's easy to write a SAT solver as

```
SAT_SOLVER(ϕ) = DNF_SAT_SOLVER(CONVERT_TO_DNF(ϕ))
```

since it's trivial to write a linear-time `DNF_SAT_SOLVER`. So the slow part of this method must be `CONVERT_TO_DNF(ϕ)`. And indeed, our `CONVERT_TO_DNF` routine may take exponential time. This isn't surprising because many people suspect that there cannot be a SAT solver whose worst-case runtime grows less than exponentially as a function of |φ|. That's the "Satisfiability Hypothesis," or SH for short.

In fact, `CONVERT_TO_DNF` may take exponential time simply to print its output! An example where `CONVERT_TO_DNF(ϕ)` blows up in size is

```
ϕ = (P1 ^ P2) v (Q1 ^ Q2) v (R1 ^ R2) v ...
```

for the same reason that the dual version of that formula blows up in the naive version of `CONVERT_TO_CNF(phi)`:

```
ϕ = (P1 v P2) ^ (Q1 v Q2) ^ (R1 ^ R2) v ...
```

In the CNF case, we were able to find a smarter version that keeps this formula compact by using switching variables. But we saw in lecture why the switching variable trick doesn't work for DNF (it has to do with the asymmetry between SAT and TAUT).

## The Tseitin Transformation

The lecture slides conclude with a different scheme called the Tseitin Transformation (Tseitin, 1968). This is much more efficient in the worst case (although it may give longer formulas in many common cases).

The basic idea is to associate a new variable with each subformula, and add 3-CNF clauses that ensure that each new variable is true iff its subformula is true. Finally, to require the whole formula to be true, add the unit clause X, where X is the variable associated with the whole formula.

This scheme ensures that the length of the final 3-CNF formula is *proportional* to the number of distinct subformulas in the original formula. There is no quadratic or exponential blowup. Furthermore, repeated subformulas do not require additional clauses.

You can find a textual description at Wikipedia.

---


*Jason Eisner* - *jason@cs.jhu.edu* (suggestions welcome) Last Mod $Date: 2014/02/14 13:42:34 $