Propositional Logic

Definition 1 *Let $X = \{x_1, x_2, \dots\}$ be a countably infinite set of propositional variables. Formulae of propositional logic are inductively defined as follows:*
- *Every propositional variable x_i is a formula.*
- *For every formula F, $\neg F$ is a formula.*
- *For all formulae F and G, $F \wedge G$ and $F \vee G$ are formulae.*

We derive the further connectives of implication, bi-implication, xor, etc. in precisely the expected way. Further, we note the set of formulae on variables X is $\mathcal{F}\langle X\rangle$.

Definition 2 *An assignment is a function $\mathcal{A} : X \to \{0,1\}$ that induces an assignment $\mathcal{A} : \mathcal{F}\langle X\rangle \to \{0,1\}$ as follows, in the expected way:*
- $\mathcal{A}[\neg F] := 1 - \mathcal{A}[F]$,
- $\mathcal{A}[F \wedge G] := \mathcal{A}[F] \cdot \mathcal{A}[G]$,
- $\mathcal{A}[F \vee G] := \mathcal{A}[F] + \mathcal{A}[G] - \mathcal{A}[F]\mathcal{A}[G]$.

Definition 3 *Let $F \in \mathcal{F}\langle X\rangle$ and $\mathcal{A} : X \to \{0,1\}$ be an assignment. If $\mathcal{A}[F] = 1$ then we write $\mathcal{A} \models F$. If F has at least one model then it is satisfiable, otherwise it is unsatisfiable.*

A formula G is entailed by a set of formulae $\mathcal{S}$ if every assignment that satisfies each formula in $\mathcal{F}$ also satisfies G. We write this $\mathcal{S} \models G$ (overloading the notation slightly). Further we say that two formulae are logically equivalent if $\mathcal{A}[F] = \mathcal{A}[G]$ for every assignment $\mathcal{A}$. We write this $F \equiv G$.

Normal forms

Definition 4 (Conjunctive and disjunctive normal forms) *We say a formula is a CNF (or is in CNF), where it is a conjunction of disjunctions. That is, F is a CNF if for some $\{C_1, \dots, C_n\}$, where each C_i is a formula lacking any occurrence of $\wedge$, we have*

$$F = \bigwedge_{i=1}^{n} C_i.$$

In the corresponding way, we say a formula is a DNF (or is in DNF), where we can write F using some set $\{C_1, \dots, C_n\}$, with each C_i a formula lacking any occurrence of $\vee$, as

$$F = \bigvee_{i=1}^{n} C_i.$$

Every formula has an equivalent CNF and DNF. To see this we induct on the structure of formulae to show the existence of both simultaneously, and in doing so demonstrate the existence of a recursive algorithm to construct both. Note however that we get potentially exponential blowup in the size of the formulas.

In general, the DNF form is far more informative. Indeed, what we get from a DNF formula is a specification of exactly the satisfying assignments, so we solve the satisfiability problem immediately by constructing a DNF with a nonempty clause. Normally, we end up dealing primarily with CNFs, or otherwise formulas that can be easily simplified to CNFs. It is usually more natural to construct formulas that add a sequence of constraints, thus giving a CNF.

Where n is the length of the formula, we can solve SAT in $O(2^n)$, but no sub-exponential algorithm is known. We can do better for special formula classes however.

Definition 5 *A CNF formula is a Horn formula if each clause contains at most one positive literal.*

This form allows them to be written as conjunctions of implications. Consequently, we can use an algorithm that begins by setting every variable to 0, then while the assignment doesn't satisfy the formula we look through the unsatisfied clauses. If the consequent is a variable then assign it true, and if it is false then return that the formula is unsatisfiable. The argument here is that for $P \to$ false to be unsatisfied then an element of P has been made true. Provided we only make things true once we're sure they have to be, we know we have unsatisfiability. In a similar way, if $P \to p$ is unsatisfied, then p is false and an element of P is true, so provided that element had to be made true, p must also be true. Thus we have that each step is logically valid.

Hence we have a linear time satisfiability check for Horn formulas.

Definition 6 *A 3-CNF formula, or Krom formula is a CNF formula F such that every clause has at most two literals.*

We write an implication graph $G = (V, E)$ where $V := \{x_1, x_2, \dots\} \cup \{\neg x_1, \neg x_2, \dots\}$, and E is defined using $a \vee b \equiv \neg a \to b$ to allow us to assign an edge between two variables where there is such an implication. Consequently we can say that a 2-CNF formula is satisfiable iff its implication graph has no $p \in X$ such that there is a path from p to $\neg p$ and $\neg p$ to p

In the forwards direction this is obvious: assume that there is such a cycle and we immediately get that we can derive the empty clause from this formula. In the backwards direction we begin by computing the topological ordering of the SCC's of the implication graph. We then pick the least SCC containing an unassigned variable, and assign 1 to every element of it (0 to every element of its complement). This guarantees that every literal reachable from C is assigned true. Consequently we end up with an assignment that is satisfying.

Theorem 1 *Given an arbitrary formula F, we can compute an equisatisfiable 3-CNF formula G in polynomial time.*

Take the subformulas $F_1, F_2, \dots, F_n = F$ of F, where $F_i, \dots, F_m$ are the atomic formulas. We write $p_1, \dots, p_m$ for these, and then add additional constraints to $p_{m+1}, \dots, p_n$. For example, where $F_i = F_j \wedge F_k$, we have $G_i = p_i \leftrightarrow p_j \wedge p_k$ (which can be written as a 3-CNF). This then allows us to get a 3-CNF with a polynomial number of connectives.

Walk-SAT is a SAT algorithm which works randomly on CNF formulae. We input a CNF formula with n variables and a repetition parameter r. Pick a random assignment of the n variables, and r times we pick an unsatisfied clause, flip a randomly selected literal, and at each point check if F is now satisfied.

Still don't understand how to get the Markov chain to work. Review (ask varun?).

The claim is that if we have

$$u_i = \max\{\mathbb{E}[\#(\text{steps to reach } \mathcal{A}^*)] : \text{at distance } i \text{ from } \mathcal{A}^*\},$$

then we get $u_0 = 0$, $u_n = 1 + u_{n-1}$, $u_i \le 1 + (u_{i-1} + u_{i+1})/2$. The claim is then that we can get a bound by making this equality strict, which is unclear.

We find that we have a bound that the probability of failure is $\le 2^{-m}$ where $r = 2mn^2$, so setting m reasonably high gives a good bound on the rate of error.

Another case to take into consideration is that of XOR-clauses. As the XOR operation behaves nicely with arithmetic in $\mathbb{F}_2$ (it's just the addition operation), the satisfiability problem is reduced to solving a system of equations with Gaussian elimination.

Compactness

Theorem 2 (Compactness) *A set $\mathcal{S}$ of formulas is satisfiable if and only if every finite subset of $\mathcal{S}$ is satisfiable.*

One direction of this is obvious – if there is $\mathcal{A}$ such that $\mathcal{A} \models \bigwedge_{F \in \mathcal{S}} F$ then we have $\mathcal{A} \models \bigwedge_{F \in \mathcal{S}'} F$ for every $\mathcal{S}' \subseteq \mathcal{S}$. The reverse direction is a bit more difficult due to the specification of 'finite'.

To begin with, write $\mathcal{S}_n$ as the set of formulas in $\mathcal{S}$ containing only the variables $x_1, \dots, x_n$. While this set could be infinite (e.g. $\{\bigwedge_{i=1}^{n} x_i : n \in \mathbb{N}\}$), up to equivalence there are only 2^{2^n} boolean functions in n variables, so there is necessarily a finite set of formulas in $S \subseteq \mathcal{S}_n$ equivalent to $\mathcal{S}_n$. We can then use the assignment satisfying S, $\mathcal{A}_n$, to build an assignment $\mathcal{A}$ satisfying $\mathcal{S}$. To do this take the sequence $(\mathcal{A}_n)$, and for each variable assign it in $\mathcal{A}$ assign it according to if one occurs infinite times in $(\mathcal{A}_n)$. Taking any arbitrary formula F in $\mathcal{S}$, this assignment satisfies F **(Why? - problems here with if F has infinite variables)**.

The importance of this theorem comes primarily in predicate logic **(where?)**.

Resolution

Resolution is a proof calculus for CNF formulas in propositional logic, whereby we can verify that a formula is unsatisfiable.

To begin, note that we can convert any CNF into clausal form: a set $\mathcal{S} = \{C_1, C_2, \dots, C_n\}$ where each C_i is a set of literals, representing the formula $\bigwedge_{C_i \in \mathcal{S}} \bigvee_{x \in C} x$. Intuitively, the task of asserting unsatisfiability is to do with verifying that the clauses relate with one another in a manner that gives a contradiction. This resolution considers two clauses, and outputs a derived clause.

Definition 7 *Let C_1 and C_2 be clauses. A clause R is called a resolvent of C_1 and C_2 if there are complementary literals $L \in C_1$ and $\overline{L} \in C_2$ such that*

$$R = (C_1 \setminus \{L\}) \cup (C_1 \setminus \{\overline{L}\}).$$

The reason for considering clause pairs of this form is that, in practice, these are the pairs that introduce difficulties. If we have for every clause that literals occur with the same sign, then we just make them positive and immediately get a satisfying assignment.

By having the sign change, resolution represents that derivation that for one of these clauses, satisfaction must be drawn from the remainder of its variables.

Lemma 3 *Let F be a CNF formula represented as a set of clauses. If R is a resolvent of clauses C_1 and C_2 of F then $F \equiv F \cup \{R\}$.*

Take any assignment $\mathcal{A} \models F$. If $\mathcal{A} \models L$, then $\mathcal{A} \models C_1 \setminus \{\overline{L}\}$, and otherwise $\mathcal{A} \models C_1 \setminus \{L\}$. so $\mathcal{A} \models R$. In the reverse direction for any assignment $\mathcal{A} \models F \cup \{R\}$, immediately $\mathcal{A} \models F$.

Having this lemma, we can now make some stronger claims about the capabilities of resolution to act as a calculus.

A derivation of a clause C from a set of clauses F is a sequence $C_1, C_2, \dots, C_m$ of clauses where $C_m = C$ and for each $i \in \{1, 2, \dots, m\}$ either $C_i \in F$ or C_i is a resolvent of C_j, C_k for some $j, k < i$. We can consider the set of possible derivations to get a framework for analysis:

Definition 8 *For a CNF F, we write*

$$\mathrm{Res}(F) = F \cup \{R : R \text{ is a resolvent of two clauses in } F\},$$

and furthermore,

$$\mathrm{Res}^0(F) = F$$
$$\mathrm{Res}^{n+1}(F) = \mathrm{Res}(\mathrm{Res}^n(F))$$
$$\mathrm{Res}^*(F) = \bigcup_{n \ge 0} \mathrm{Res}^n(F).$$

If $\square \in \mathrm{Res}^*(F)$, then there is some derivation sequence $C_1, C_2, \dots, C_n = \square$ from F. Assume without loss of generality that $C_1, \dots, C_n$ are the clauses of F. By induction we then get that $F \equiv \{C_1, \dots, C_n, \square\} \equiv \square$, so F is unsatisfiable. Thus we get that resolution is sound, because any refutation derived is genuine. Further, if F is unsatisfiable, then by induction on the number of distinct propositional variables we can show that there is a derivation of $\square$ (for each new variable x_n, show we can derive either assignment to it if the formula is unsatisfiable). Thus resolution is complete.

Resolution gives an immediate algorithm to determine satisfiability, however the key problem emerges in that we often compute a great deal of redundancy in each call of Res. This can be reduced sometimes for particular classes – for example with Horn clauses we still have completeness where we ensure that one of the clauses is always a unit clause.

The DPLL algorithm

The DPLL algorithm combines search and deduction to decide satisfiability of CNF-formulas, reducing the calculation of resolvents to more relevant ones. The idea is to use a depth-first search – at every unsaturated leaf of the search tree (called a conflict), use resolution to compute a conflict clause. Add this clause to the formula we're deciding about.

We initialise $\mathcal{A}$ as the empty assignment. While there is a unit clause $\{L\}$ in $F\big|_{\mathcal{A}}$ update $\mathcal{A} \mapsto \mathcal{A}[L \mapsto 1]$. If $F\big|_{\mathcal{A}}$ contains no clauses, stop and output $\mathcal{A}$. If this contains empty determine a new clause C to add to F by a learning procedure. If this is the empty clause, then F is unsatisfiable – otherwise backtrack to the highest level where C is the unit clause and loop.
Note for the following algorithm the distinction between F and $F\big|_{\mathcal{A}}$. Where F is the normal formula, $F\big|_{\mathcal{A}}$ is the formula only in variables not yet assigned to by $\mathcal{A}$.

```
DPLL(F):
1  Initialise partial assignment A := ∅.
2  Initialise assignment log S := ∅, count k := 0.
3  while TRUE
4      while ∃ℓ. there is a unit clause {ℓ} ∈ F|_A from clause C ∈ F
5          A[ℓ] := 1
6          Append(S, ℓ, C_ℓ); k := k + 1
7      if F|_A = ∅
8          return A
9      elseif □ ∈ F|_A from clause C ∈ F
10         A_{k+1} := C                          // Oop there's a conflict...
11         for i := k to 1
12             if S_i is an assignment to some p_i ∈ A_{i+1} implied by clause C_i
13                 A_i := Resolve(A_{i+1}, C_i, p_i)   // ...and p_i caused it.
14             else
15                 A_i := A_{i+1}
16         if A_1 = ∅
17             return UNSAT                      // We have proven □.
18     else
19         ℓ := F ∪ {A_i}
20         while A_i|_A is not a unit clause
21             A := A \ S_k
22             Remove(S, k); k := k − 1
23     else
24         (p ↦ b) := Decide(F, A)               // Can be arbitrary.
25         A[p] := b
26         Append(S, p ↦ b); k := k + 1
```

Described more intuitively: we want to build up an assignment gradually. At each stage we begin with some assignment which has yet to be considered, based on decisions made in previous iterations. We then learn via unit propagation the immediate consequences of these decisions, and one of three things may happen:
1. We find that our decisions were correct, and get a satisfying assignment which we can return.
2. A clause is made false by our decisions, so we need to learn which decisions made them false, and reassign.
3. Neither, so we need to make more decisions.

Cases (1) and (3) give relatively straightforward directions to go from, so there's little to do in verifying correctness there. Case (2) is where more work needs to be done however. In order to learn from our decisions, we take the clause that's been made false, using it along with all other clauses involved in assignments to derive a new clause that contains *only decision variables*.

By having this clause, we now know that in order for any of the previous decisions to end up working out, some of the more recent ones require a change (we will never resolve to a clause which is currently true, as $\textsc{Resolve}(A_{i+1}, C_i, p_i)$ has $C_i \setminus \{p_i, \overline{p_i}\}$ currently false and by induction A_{i+1} currently false).

Thus what we have is a depth-first search, because by introducing a new clause at each stage, we ensure that a decision variable is converted to a variable assigned through unit propagation, and thus as this clause was a conflict clause previously, we ensure that it is a different path now being taken.

This gives in the worst case $O(2^n)$ performance, but with improvements via pruning the tree.

Lower Bounds for Resolution

While no polynomial-time algorithm is known (or is believed to exist) for SAT, as SAT is **NP**. It's possible to provide an efficiently checkable certificate that a formula is satisfiable (namely, the satisfying assignment). The question then becomes whether it's possible to provide a certificate likewise of unsatisfiability – the question of whether SAT $\in$ **coNP** (and indeed, whether **NP** $=$ **coNP**). A candidate for such a certificate would be a resolution refutation, however we give an example below of resolution refutations being exponential in size.

The pigeonhole principle states that given n objects placed into $n-1$ boxes, there is a box which contains at least 2 objects. Formalised, we have x_{ij} represent the ith object in the jth box, and that each object is in exactly one box. To derive a contradiction we say additionally that each box contains exactly one object.

To begin we define the notion of a pseudo-refutation. This is a sequence of monotone clauses (that is, clauses where variables occur only positively) $C_1, \dots, C_m = \square$, such that given groundwork CRIT and a formula to refute $\bigwedge_{i \in S} P_i$, we have for each $1 \le i \le m$ either
1. CRIT $\wedge P \models C_i$ for some $P \in S$, or
2. CRIT $\wedge C_j \wedge C_k \models C_i$ for some $j, k < i$.

In this case we take CRIT_n as the statement that the mapping is bijective for n objects, $n-1$ boxes (every box contains exactly one object, and no object is in two boxes), and P_i is the statement that object i is in a box, so $\bigwedge_{i=1}^{n} P_i$ is the statement that the mapping is well-defined.

The point which makes these refutations 'pseudo' is the monotonicity. In a general case it's unlikely that we can get every possible refutation represented by monotone ones, so we require more detail from CRIT_n. To do this, just note that by CRIT_m, object i not being in box j is equivalent to another object being in box j – a negative statement equal to one containing only positive variables. Thus for any resolution refutation of the pigeonhole principle for n, we have a pseudo-refutation of the same length.

Lemma 4 *Every pseudo-refutation of the pigeonhole principle for n objects contains a clause with at least $2n^2/9$ variables.*

To begin with, every clause in a pseudo-refutation has a witness $W \subseteq \{1, \dots, n\}$ of the P_is that entail it. We take the weight of a clause as the minimum cardinality of its witnesses, entailing that C_m for any pseudo-refutation must have weight n. Further, we can at most double the weight on each step, so the first C with weight $\ge n/3$ must have weight $\le 2n/3$. Using this witness W, take an assignment $\mathcal{A}$ which leaves out $i_1 \in W$, and places $i_2 \notin W$ in box j_2. Swap this over via $\mathcal{A}[x_{i_1 j}] = 1, \mathcal{A}[x_{i_2 j}] = 0$, so we get $\mathcal{A}'$ and CRIT_{n-1} entailing C. By monotonicity thus $x_{i_1 j}$ must be mentioned in C, so thus C mentions $|W|(n-|W|) \ge 2n^2/9$ variables.

Now considering how this applies to more general refutations, given a pseudo-refutation $C_1, \dots, C_m$, we say a clause is long if it has at least $m^2/8$ variables. If we have l long clauses, by double counting we have that there is some variable occurring in more than $l/8$ clauses, and by rearranging labels we can make this $x_{n-1,n}$. We can then remove all references to the nth object or $(n-1)$th box, giving us a pseudo-refutation of the problem for $n-1$ objects with $\le 7l/8$ long clauses. Repeating this $n/4$ times gives a refutation for $3n/4$ objects with $\le (7/8)^{n/4} l$ long clauses, but $2(3n/4)^2/9 = n^2/8$ so $l \ge (8/7)^{n/4} \ge 2^{n/25}$, and thus for any pseudo-refutation we have an exponential number of clauses, meaning any resolution refutation has an exponential number of clauses.

Theories

Definition 9 *For a first first-order signature σ, a theory $\boldsymbol{T}$ is a set of σ-sctnences that is closed under entailment. That is, if $\boldsymbol{T} \models F$ then $F \in \boldsymbol{T}$.*

Given any σ-structure $\mathcal{A}$, we denote the set of sentences that hold in $\mathcal{A}$ as $\mathrm{Th}(\mathcal{A})$ (as this is a theory).

We say that a theory is complete if for any sentence F, either $F \in \boldsymbol{T}$, or $\neg F \in \boldsymbol{T}$. The theory of any particular structure is complete, however a theory of the form $\boldsymbol{T} = \{F : \boldsymbol{S} \models F\}$ (one defined by axioms $\boldsymbol{S}$) can fail to be.

A theory admits quantifier elimination if for any formula $\exists x F$, with F quantifier-free, there exists a quantifier-free formula G with the same free variables as $\exists x F$ such that $\boldsymbol{T} \models \exists x F \leftrightarrow G$. We further say that there is a quantifier elimination procedure if an algorithm exists to determine G from F.

A theory is decidable if there is an algorithm that, given a sentence F, decides whether $F \in \boldsymbol{T}$.

Add more to this.

First-Order Predicate Logic

First-Order logic extends propositional logic by allowing us to form statements not just in relation to preexisting assertions, but considering objects on which operations may be applied, and of which predicates can be made.

Syntactically, we have a signature σ consisting of function symbols $f_i, f_3, \dots$, and predicate symbols $p_i, p_2, \dots$. Every variable x is a σ-term, and further if $t_1, \dots, t_k$ are σ-terms and f is a k-ary function symbol then $f(t_1, \dots, t_k)$ is a σ-term.

Note it is possible to have 0-ary functions, which consequently return a constant value. Some texts define these separately as 'constant symbols', however this is not strictly necessary. Given then that constant symbols are a type of function, despite appearing as objects similar to variables they naturally lack certain properties (e.g. we cannot quantify over constants).

Next, if P is a k-ary predicate symbol and $t_1, \dots, t_k$ are σ-terms, then $P(t_1, \dots, t_k)$ is a formula. For each formulas $F \wedge \neg F$, $(F \wedge G)$ are all formulas. Further, if x is a variable and F is a formula, then $\exists x\, F$ and $\forall x\, F$ are both formulas.

Having now defined, in the same way as was done for propositional logic, the conditions which constitute a formula, it remains to state how we may assign to it in order to generate instances of it being true or false. We refer to such assignments as σ-structures.

A σ-structure $\mathcal{A}$ consists of a non-empty universe $U_{\mathcal{A}}$ and an interpretation $I_{\mathcal{A}}$, where for each variable x we have $I_{\mathcal{A}}(x) \in U_{\mathcal{A}}$, each k-ary predicate symbol P we have $I_{\mathcal{A}}(P) \subseteq U_{\mathcal{A}}^k$, and each k-ary function f we have $I_{\mathcal{A}}(f) : U_{\mathcal{A}}^k \to U_{\mathcal{A}}$. For ease we tend to omit the $I_{\mathcal{A}}$. and instead write $x_{\mathcal{A}}$, $P_{\mathcal{A}}$, and $f_{\mathcal{A}}$

We can then define $\mathcal{A}[t]$ inductively where for any variable x, $\mathcal{A}[x] = x_{\mathcal{A}}$, and for any k-ary function f in terms $t_1, \dots, t_k$, $\mathcal{A}[f(t_1, \dots, t_k)] = f_{\mathcal{A}}(\mathcal{A}[t_1], \dots, \mathcal{A}[t_k])$.

Finally, we can determine if the assignment makes the formula true, in the expected way. Given formulas F and G assignment behaves identically as with propositional logic, and so the only extra work to do is in relation to predicates and quantifiers.

For a k-ary predicate P in terms $t_1, \dots, t_k$ we have

$$\mathcal{A}[P(t_1, \dots, t_k)] = \begin{cases} 1 & \text{if } (\mathcal{A}[t_1], \dots, \mathcal{A}[t_k]) \in P_{\mathcal{A}} \\ 0 & \text{otherwise.} \end{cases}$$

Further, by introducing the notation $\mathcal{A}_{[x:=a]}$ to denote the assignment identical to $\mathcal{A}$ with the exception that $\mathcal{A}_{[x:=a]}[x] = a$, we define quantifiers as follows:

$$\mathcal{A}[\forall x\, F] = \begin{cases} 1 & \text{if for all } u \in U_{\mathcal{A}},\ \mathcal{A}_{[x:=u]}[F] = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{A}[\exists x\, F] = \begin{cases} 1 & \text{if there is a } u \in U_{\mathcal{A}} \text{ such that } \mathcal{A}_{[x:=u]}[F] = 1 \\ 0 & \text{otherwise} \end{cases}$$

It should hopefully be clear from this definition that one can express any propositional formula just as a first-order formula without any variables (so all predicates are 0-ary). Alternatively, if all quantifiers are removed, we end up with a propositional formula in various predicates, and just distinguish them in terms of which predicate it is, and whether the arguments are the same.

Normal forms

As with propositional logic, we would quite like a means of normalising different first-order formulas, so as to make them easier to deal with. While we have all of the normal forms as previously given by propositional logic, the particular complexity emerges in dealing with quantifiers.

As expected, we have quantifier duality ($\neg \forall x\, F \equiv \exists x \neg F$, and vice-versa), and that quantifiers of the same type commute with one another. Furthermore, if x does not occur free in G, then with $\circ \in \{\wedge, \vee\}$, $Q \in \{\forall, \exists\}$,

$$(Qx\, F) \circ G \equiv Qx(F \circ G).$$

Otherwise, we note that if y does not occur free in F, $Qx\, F \equiv Qy\, F[y/x]$, where $[y/x]$ is the operation of substituting every occurrence of x for one of y (it's useful to prove this lemma oneself). Thus we can apply the above statement in the same way.

As a final statement on quantifier behaviour, note that we have $\forall$ analogous to $\wedge$, and $\exists$ analogous to $\vee$, giving

$$(\forall x\, F) \wedge (\forall x\, G) \equiv \forall x\, F \wedge G$$
$$(\exists x\, F) \vee (\exists x\, G) \equiv \exists x\, F \vee G.$$

Definition 10 (Rectified) *A formula is rectified if no variable occurs both bound and free, and if all quantifiers refer to different variables.*

This serves as a very initial means of cleaning up formulas, which can be done just by variable renaming.

Definition 11 (Prenex form) *A formula is in prenex form if it has the form*

$$Q_1 x_1 Q_2 y_2 \dots Q_n b_n F$$

for $Q_i \in \{\forall, \exists\}$, $n \ge 0$, and where F does not contain a quantifier.

On a technicality it's possible to have a non-rectified formula in prenex form, however in practice this would never be constructed, and so we end up getting it aware that the inner quantifier overrides the outer quantifier, so in prenex form the outer quantifier is void.

By structural induction we can get any formula into (rectified) prenex form. Finally we now consider a normal form that is not considered, but rather equisatisfiable (but is used due to giving a rather more useful construction).

Definition 12 (Skolem form) *A formula is in Skolem form if it is in rectified prenex form, without any occurrence of the existential quantifier.*

To construct this, one takes an initial formula $F \equiv \forall x_1 \dots \forall x_k \exists y\, G$, and then introduces a new k-ary function symbol f in order to create a new formula $F' := \forall x_1 \dots \forall x_k G[f(x_1, \dots, x_n)/y]$. F and F' are equisatisfiable, because the existence of a function f indicates that we can always produce a y to satisfy G, and in the reverse direction if there does exist a y satisfying G then there is a function f giving such a y (note however that this requires the axiom of choice).

The immediate problem presented by first-order logic is the possibility for a variety of different universes, and so in addition to normalising the formulas themselves, we also introduce the notion of a Herbrand model.

Definition 13 (Herbrand universe) *The Herbrand universe $D(F)$ of a closed formula F (one with no free variables) in Skolem form is the set of all variable-free terms that can that can be built from the components of F.*

$$D^0(F) = \begin{cases} \{a\} & \text{if there is no constant symbol in } F \\ \varnothing & \text{otherwise} \end{cases}$$
$$D^{n+1}(F) = \{f(t_1, \dots, t_k) : k \ge 0,\ f \text{ a } k\text{-ary function symbol}, t_1, \dots, t_k \in D^n(F)\}$$
$$D(F) = \bigcup_{n \ge 0} D^n(F).$$

Note that if we don't have any constant symbols in F, the recursion can't ever begin, and we get an empty universe. We can add the constant symbol a without loss of generality, because by the definition of a universe it will be non-empty.

The point of this construction is that no element is created unnecessarily – in any other universe it's feasible that we create entire classes of redundant objects, which Herbrand's universe avoids. We then naturally create the structure, by interpreting each function as it is originally (given that $D(F)$ works on the original objects, so there's no need for change). We say that a Herbrand model is any assignment to a formula using a Herbrand structure, for which the formula is satisfied.

Theorem 5 (Herbrand's theorem) *Let F be a closed formula in Skolem form. Then F is satisfiable iff it has a Herbrand model.*

The forwards direction is obvious by the definition of satisfiability. To see the backwards direction, if F is satisfiable we take an arbitrary model $\mathcal{A}$, and convert it to a Herbrand model $\mathcal{B}$. for which by F being in Skolem form we can then show that $\mathcal{B}$ is a satisfying assignment.

One useful corollary of this is that we immediately see that any satisfiable formula in predicate logic has a countable model.

Definition 14 (Herbrand expansion) *Let $F = \forall x_1 \forall x_2 \dots \forall x_n F'$ be a closed formula in Skolem form. The Herbrand expansion is defined as*

$$E(F) = \{F'[x_1/t_1][x_2/t_2] \cdots [x_n/t_n] : t_1, \dots, t_n \in D(F)\}$$

This is essentially a conversion of F to a set of propositional formulas.

Theorem 6 (Gödel – Herbrand – Skolem) *For every closed formula F in Skolem form, F is satisfiable (as a first-order formula) iff $E(F)$ is satisfiable (as a set of propositional formulas).*

By the above this is clear.

Note that by compactness, F is unsatisfiable iff there exists a resolution refutation for some subset of $E(F)$, so we can determine unsatisfiability by using resolution to try to find a resolution refutation for each finite n formulas of $E(F)$, giving an algorithm that either halts with UNSAT or never halts at all.

Resolution

The Herbrand expansion, and the method of verifying unsatisfiability which it prompts, corresponds to a set of substitutions, of which we desire to find a minimal set to determine unsatisfiability. The ground resolution theorem states this more precisely (although admittedly doesn't add much – not entirely sure why this is called a theorem).

Theorem 7 (Ground resolution theorem) *A closed formula f in Skolem form $F = \forall x_1 \dots \forall x_n F'$ with its matrix F' in CNF is unsatisfiable iff there exists a finite sequence of clauses $C_1, C_2, \dots, C_m = \square$ where each C_i is either a ground instance of some $C \in F'$ (i.e. $C_i = C[x_1/t_1] \cdots [x_n/t_n]$), or it is a resolvent of two previous clauses.*

As this is, the algorithm we would immediately construct is just an application of resolution, which as we saw previously requires a in-depth search, with minimal possibility for improvement. The key point here is that in this instance we are allowed to choose our substitutions, thus allowing us to try to select substitutions that give the best opportunity for useful resolution.

In particular, what we want is that the substitutions unify two or more distinct literals, so to allow them to be resolved.

Definition 15 (Most general unifier) *A substitution sub is a unifier for a set of literals $\boldsymbol{L} = \{L_1, L_2, \dots, L_k\}$ if $\mathrm{sub}(L_1) = 1$ (considering substitution as a function on literals). Further, sub is a most general unifier if for every unifier sub′ there is a substitution s such that $\mathrm{sub}' = \mathrm{sub}s$.*

Note we can get this MGU via the following algorithm:

```
UNIFICATION(L)
1  sub := id
2  while |sub(L)| > 1
3      Find two distinct literals in sub(L)
           and find the first symbol at which they differ
4      if neither symbols have a variable subterm
5          return NON-UNIFIABLE          // Can't unify with no variables.
6      else
7          Write x for the symbol containing a variable, t for the other term.
8          if x occurs in t
9              return NON-UNIFIABLE      // Substitution would change both sides.
10         else
11             sub := sub[t/x]
12 return sub
```

To see that this terminates, note that the number of different variables present in $\mathrm{sub}(\boldsymbol{L})$ decreases by 1 each iteration. That this creates an MGU follows by the loop invariant that for any unifier sub′ of $\boldsymbol{L}$, $\mathrm{sub}' = \mathrm{sub}s\mathrm{ub}'$, so as long as sub eventually becomes a unifier, it will be a general one.

Decidability

Given that SAT is **NP**-complete, and correspondingly the problem of determining validity is **coNP**-complete, we immediately have that determining satisfiability of first-order formulas will be at least **NP**-hard, and validity **coNP**-hard. In the previous section we showed that unsatisfiability is **RE**, indicating that satisfiability is **coRE**, and by taking a negation, validity is **RE**. As it turns out, this is just about as far as we can get:

Theorem 8 *The problems of determining either whether a first-order formula is satisfiable, or whether it is valid, are undecidable.*

To see this, we reduce the Post Correspondence Problem to the validity problem for first-order formulas. To recall, the Post Correspondence Problem provides a sequence $(x_1, y_1), \dots, (x_k, y_k)$ of pairs in $\{0,1\}^*$, and asks whether there exists a sequence of indices $i_1, i_2, \dots, i_n$ in $\{1, \dots, k\}$ for $n \ge 1$ such that $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$.

To write this as a first-order formula, we want to construct a 2-ary predicate P, which over a universe $U_{\mathcal{A}} = \{0,1\}^*$, is the set of (α, β) with neither empty, such that there are indices $i_1, \dots, i_n$ with $\alpha = x_{i_1} \cdots x_{i_n}$, $\beta = y_{i_1} \cdots y_{i_n}$. To do this, we write

$$F_i = \bigwedge_{i=1}^{k} P(f_x(\alpha), f_y(\alpha))$$

representing the statement that we can always build each (x_i, y_i). Further, we then write

$$F_2 = \forall u\, \forall v\, (P(u, v) \to \bigwedge_{i=1}^{k} P(f_x(u), f_y(v)))$$

representing the statement that if we can build (u, v), then we can build (ux_i, vy_i) for any i. This leaves us with just the implication $F = (F_1 \wedge F_2) \to \exists z\, P(z, z)$, which provided the above interpretation is always reflected, should hold iff there is a string which can be constructed solving the PCP.

To see that it does, note immediately an interpretation where we have $f_x^A(\alpha) = \alpha x_i$, $f_y^A(\beta) = \beta y_i$ for each i, P as described and a representing ε, gives an assignment for which, if the formula is valid, the problem instance must be true by implication. In the converse, assuming that such a sequence $i_1, i_2, \dots, i_n$. then for any arbitrary structure $\mathcal{A}$ of F, either $\mathcal{A} \not\models F_1 \wedge F_2$, in which case immediately $\mathcal{A} \models F$, or $\mathcal{A} \models F_1 \wedge F_2$, in which case we can construct an embedding from $\{0,1\}^*$ to $U_{\mathcal{A}}$ to demonstrate $\exists z\, P(z, z)$.

As PCP is undecidable, thus validity must also be undecidable. and as satisfiability is the same as the negation of a formula being invalid, we thus have that satisfiability is undecidable.

Expressiveness

Definition 16 (Ehrenfeuicht-Fraïssé game) *Let $k, m \in \mathbb{N}$, σ be a finite relational signature, $\mathcal{A}, \mathcal{B}$ σ-structures, $a \in A^m$, $b \in B^m$, with the game $G_k((\mathcal{A}, a), (\mathcal{B}, b))$ a k-round game. In each round the spoiler chooses either $a \in A$ or $b \in B$, and the duplicator chooses an element of the other structure.*

After k-rounds, we have tuples $a' \in A^{m+k}$, $b' \in B^{m+k}$, and the duplicator wins iff for all atomic formulas φ in $m+k$ variables, $\mathcal{A} \models \varphi(a')$ iff $\mathcal{B} \models \varphi(b')$ (i.e. the structures are the same).

Add more to this.