PROPOSITIONAL LOGIC: SYNTAX AND SEMANTICS

Course “Computational Logic”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract Syntax

A propositional formula F describes a “sentence” that can be “true” or “false”.

Sentence:	“If it rains, then I get wet, but it does not rain.”
Formula:	$(rains \Rightarrow wet) \wedge (\neg rains)$

- BNF Grammar:

$$F ::= \ \bot \mid \top \mid p \mid (\neg F) \mid (F_1 \wedge F_2) \mid (F_1 \vee F_2) \mid (F_1 \Rightarrow F_2) \mid (F_1 \Leftrightarrow F_2)$$

  - Propositional constants $\bot$ (“false”) and $\top$ (“true”).
  - Propositional variables (“atoms”, “atomic formulas”) $p \in \mathcal{P}$.
  - Compound formulas constructed from the (logical) connectives $\neg$ (“not”, “negation”), $\wedge$ (“and”, “conjunction”), $\vee$ (“or”, “disjunction”), $\Rightarrow$ (“if then”, “implication”), $\Leftrightarrow$ (“if and only if”, “equivalence”).

Many other versions of concrete syntax do exist; these can be always transformed into above “standard form”.

Concrete Syntax

In practice we reduce the number of parentheses by the following conventions.

- We apply the following binding rules:

$$(\neg) \succ (\land) \succ (\lor) \succ (\Rightarrow) \succ (\Leftrightarrow)$$

  - $(C_1) \succ (C_2)$ ... connective C_1 binds stronger than connective C_2.
- We write $F \land G \land H \land \ldots$ for $((F \land G) \land H) \land \ldots$
- We write $F \lor G \lor H \lor \ldots$ for $((F \lor G) \lor H) \lor \ldots$
  - We will see later that conjunction and disjunction have associative semantics.

$$F \land \neg G \land H \Rightarrow F \lor G$$

$$\leadsto \quad (((F \land (\neg G)) \land H) \Rightarrow (F \lor G))$$

Be sure to (mentally) insert parentheses appropriately.

Abstract Syntax in OCaml

- OCaml Type:

```
type ('a)formula = False | True | Atom of 'a
| Not of ('a)formula | And of ('a)formula * ('a)formula | Or of ('a)formula * ('a)formula
| Imp of ('a)formula * ('a)formula | Iff of ('a)formula * ('a)formula | ... ;;

type prop = P of string;;
type propformula = prop formula;;
```

- Execution:

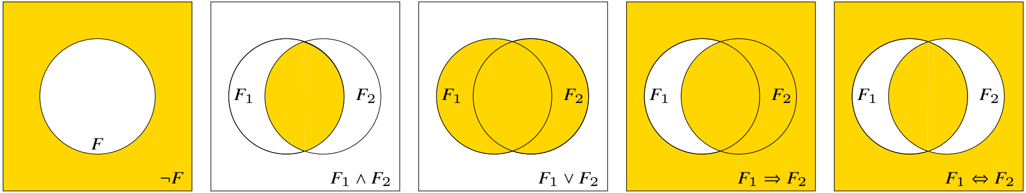
```
# let f = <<p /\ q ==> q /\ r>> ;;
val f : prop formula = <<p /\ q ==> q /\ r>>

# let g = Imp(And(Atom(P "p"),Atom(P "q")),And(Atom(P "q"),Atom(P "r"))) ;;
val g : prop formula = <<p /\ q ==> q /\ r>>
```

Propositional formulas are values of type `prop formula`.

Interpretation of Logical Connectives

F	$\neg F$
false	true
true	false

F_1	F_2	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \Rightarrow F_2$	$F_1 \Leftrightarrow F_2$
false	false	false	false	true	true
false	true	false	true	true	false
true	false	false	true	false	false
true	true	true	true	true	true



The interpretation of logical connectives can be defined by truth tables.

Formal Semantics: Valuations

- $\mathbb{B} := \{\text{true}, \text{false}\}$ is the domain of truth values.
- A valuation (assignment) $v : \mathcal{P} \to \mathbb{B}$ is a function that maps propositional variables to truth values.

$$v = [p \mapsto \text{true}, q \mapsto \text{true}, r \mapsto \text{false}]$$

$$v(p) = \text{true},\ v(q) = \text{true},\ v(r) = \text{false}.$$

- If $|\mathcal{P}| = n$, there are 2^n valuations on $\mathcal{P}$.

The semantics of a propositional formula depends on the truth values of the propositional variables that occur in the formula.

Formal Semantics: Formulas

- Given valuation v, the semantics $[\![F]\!]_v$ of formula F is defined as follows:

$$[\![\bot]\!]_v := \text{false} \qquad [\![\top]\!]_v := \text{true} \qquad [\![p]\!]_v := v(p) \qquad [\![\neg F]\!]_v := \begin{cases} \text{true} & \text{if } [\![F]\!]_v = \text{false} \\ \text{false} & \text{else} \end{cases}$$

$$[\![F_1 \wedge F_2]\!]_v := \begin{cases} \text{true} & \text{if } [\![F_1]\!]_v = [\![F_2]\!]_v = \text{true} \\ \text{false} & \text{else} \end{cases}$$

$$[\![F_1 \vee F_2]\!]_v := \begin{cases} \text{false} & \text{if } [\![F_1]\!]_v = [\![F_2]\!]_v = \text{false} \\ \text{true} & \text{else} \end{cases}$$

$$[\![F_1 \Rightarrow F_2]\!]_v := \begin{cases} \text{false} & \text{if } [\![F_1]\!]_v = \text{true and } [\![F_2]\!]_v = \text{false} \\ \text{true} & \text{else} \end{cases}$$

$$[\![F_1 \Leftrightarrow F_2]\!]_v := \begin{cases} \text{true} & \text{if } [\![F_1]\!]_v = [\![F_2]\!]_v \\ \text{false} & \text{else} \end{cases}$$

Function $[\![.]\!]$ maps a formula and a valuation to a truth value.

Formula Semantics in OCaml

```
# let rec eval fm v =
  match fm with
    False -> false | True -> true | Atom(x) -> v(x) | Not(p) -> not(eval p v)
  | And(p,q) -> (eval p v) & (eval q v)
  | Or(p,q) -> (eval p v) or (eval q v)
  | Imp(p,q) -> not(eval p v) or (eval q v)
  | Iff(p,q) -> (eval p v) = (eval q v);;
val eval : 'a formula -> ('a -> bool) -> bool = <fun>

# let v1 = (function P "p" -> true | P "q" -> false | P "r" -> true) ;;
val v1 : prop -> bool = <fun>
# eval f v1 ;;
- : bool = true
# let v2 = (function P "p" -> true | P "q" -> true  | P "r" -> false) ;;
val v2 : prop -> bool = <fun>
# eval f v2 ;;
- : bool = false
```

Semantics implemented on top of the builtin Boolean operations.

Computing Truth Tables

p	q	r	$p \wedge q$	$q \wedge r$	$(.) \Rightarrow (.)$
false	false	false	false	false	true
false	false	true	false	false	true
false	true	false	false	false	true
false	true	true	false	true	true
true	false	false	false	false	true
true	false	true	false	false	true
true	true	false	true	false	false
true	true	true	true	true	true

```
# print_truthtable <<(~p)\/(~q)\/(q/\r)>> ;;

p     q     r     | formula
---------------------------
false false false | true
false false true  | true
false true  false | true
false true  true  | true
true  false false | true
true  false true  | true
true  true  false | false
true  true  true  | true
```

A truth table illustrates the semantics of a formula; two formulas with the same truth table are “logically equivalent”.

Functional Completeness of Connectives

The set of logical connectives is “functionally complete”, i.e., sufficient to describe any truth table.

p	q	r	F =?
false	false	false	<u>true</u>
false	false	true	false
false	true	false	<u>true</u>
false	true	true	false
true	false	false	false
true	false	true	<u>true</u>
true	true	false	<u>true</u>
true	true	true	false

$$
\begin{aligned}
F = {} & (\neg p \wedge \neg q \wedge \neg r) \\
& \vee (\neg p \wedge q \wedge \neg r) \\
& \vee (p \wedge \neg q \wedge r) \\
& \vee (p \wedge q \wedge \neg r)
\end{aligned}
$$

A disjunction of conjunctions of (possibly negated) propositional variables suffices.

Validity and Satisfiability

- Definitions:
  - A propositional formula F is valid (a tautology) if F is true for *every* valuation:
  $$[\![F]\!]_v = \text{true}, \text{ for } \textit{every } v.$$
  - F is satisfiable if it is true for *some* valuation:
  $$[\![F]\!]_v = \text{true}, \text{ for } \textit{some } v.$$
  - F is unsatisfiable (a contradiction) if it is not satisfiable:
  $$[\![F]\!]_v = \text{false}, \text{ for } \textit{every } v.$$
  - F is refutable if it is not valid:
  $$[\![F]\!]_v = \text{false}, \text{ for } \textit{some } v.$$
- Theorems:
  - F is valid, if $\neg F$ is unsatisfiable.
  - F is satisfiable, if $\neg F$ is refutable.

Validity checking can be reduced to searching for satisfying valuations.

Validity and Satisfiability in OCaml

```
let tautology fm = onallvaluations (eval fm) (fun s -> false) (atoms fm);;
let unsatisfiable fm = tautology(Not fm);;
let satisfiable fm = not(unsatisfiable fm);;
```

```
# tautology <<p \/ ~p>> ;;
- : bool = true
# unsatisfiable <<p /\ ~p>> ;;
- : bool = true
# tautology <<p \/ q ==> p>> ;;
- : bool = false
# satisfiable <<p \/ q ==> p>> ;;
- : bool = true
# tautology <<(p \/ q) /\ ~(p /\ q) ==> (~p <=> q)>> ;;
- : bool = true
```

```
# print_truthtable <<(p\/q)/\~(p /\ q)==>(~p<=>q)>> ;;
p     q     | formula
---------------------
false false | true
false true  | true
true  false | true
true  true  | true
```

Auxiliary Functions

```
(* recursively compute (starting with valuation v) all valuations on the atoms in ats
(* yield true if subfn yields true for all these valuations *)
let rec onallvaluations subfn v ats =
  match ats with
    [] -> subfn v
  | p::ps -> let v' t q = if q = p then t else v(q) in
             onallvaluations subfn (v' false) ps & onallvaluations subfn (v' true) ps;;

(* compute f(a1,f(a2,...f(an,b)...)) for atoms {a1,...an} in formula fm *)
let rec overatoms f fm b =
  match fm with
    Atom(a) -> f a b | Not(p) -> overatoms f p b
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> overatoms f p (overatoms f q b)
  | _ -> b;;

(* compute f(a1)@f(a2)@...@f(an) for atoms {a1,..,an} in formula fn *)
let atom_union f fm = setify (overatoms (fun h t -> f(h)@t) fm []);;

(* compute the list of atoms in formula fm *)
let atoms fm = atom_union (fun a -> [a]) fm;;
```

Logical Consequence and Equivalence

- Definitions:
  - G is a logical consequence of F ($F \models G$):
    For every v, if $[\![F]\!]_v$ = true then $[\![G]\!]_v$ = true.
  - F and G are logically equivalent ($F \equiv G$):
    For every v, $[\![F]\!]_v$ = true if and only if $[\![G]\!]_v$ = true.
- Theorems:
  - $F \models G$ if and only if formula $F \Rightarrow G$ is a tautology.
  - $F \equiv G$ if and only if formula $F \Leftrightarrow G$ is a tautology.
- Logical Equivalence and Substitution:
  - Let $H[p]$ be a formula with propositional variable p. Let $H[F]$ and $H[G]$ be H where every occurrence of p has been replaced by formula F and G, respectively.
  - Then $F \equiv G$ implies $H[F] \equiv H[G]$.

Logical equivalence is a congruence relation for the logical connectives.

Logical Equivalences: Negation, Conjunction, Disjunction

$$\neg\top \equiv \bot$$
$$\neg\bot \equiv \top$$
$$\neg\neg A \equiv A$$
$$\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$$
$$\neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$$
$$\neg(A \Rightarrow B) \equiv A \wedge \neg B$$
$$\neg(A \Leftrightarrow B) \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$$

$$A \wedge A \equiv A$$
$$A \wedge B \equiv B \wedge A$$
$$A \wedge \top \equiv A$$
$$A \wedge \bot \equiv \bot$$
$$A \wedge (\neg A) \equiv \bot$$
$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$$
$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$
$$A \wedge (A \vee B) \equiv A$$

$$A \vee A \equiv A$$
$$A \vee B \equiv B \vee A$$
$$A \vee \top \equiv \top$$
$$A \vee \bot \equiv A$$
$$A \vee (\neg A) \equiv \top$$
$$A \vee (B \vee C) \equiv (A \vee B) \vee C$$
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
$$A \vee (A \wedge B) \equiv A$$

Logical Equivalences: Implication and Equivalence

$$A \Rightarrow B \equiv \neg A \vee B$$

$$A \Rightarrow A \equiv \top$$

$$A \Rightarrow \top \equiv \top$$

$$A \Rightarrow \bot \equiv \neg A$$

$$\top \Rightarrow A \equiv A$$

$$\bot \Rightarrow A \equiv \top$$

$$A \Rightarrow B \equiv (\neg B) \Rightarrow (\neg A)$$

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

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

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

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

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

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$$

$$A \Leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$$

$$A \Leftrightarrow A \equiv \top$$

$$A \Leftrightarrow \top \equiv A$$

$$A \Leftrightarrow \bot \equiv \neg A$$

$$\top \Leftrightarrow A \equiv A$$

$$\bot \Leftrightarrow A \equiv \neg A$$

$$A \Leftrightarrow B \equiv B \Leftrightarrow A$$

$$A \Leftrightarrow B \equiv \neg A \Leftrightarrow \neg B$$

Functional Completeness

A small subset of logical connectives may be sufficient to describe any truth table.

- The set $\{\neg, \wedge\}$ is functionally complete.

$$\bot \equiv A \wedge \neg A$$

$$\top \equiv \neg(A \wedge \neg A)$$

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

$$A \Rightarrow B \equiv \neg(A \wedge \neg B)$$

$$A \Leftrightarrow B \equiv \neg(A \wedge \neg B) \wedge (\neg(\neg A \wedge B))$$

- The set $\{\neg, \Rightarrow\}$ is functionally complete.
- The set $\{\bar{\wedge}\}$ with $F \bar{\wedge} G := \neg(F \wedge G)$ is functionally complete.
- ...

Useful to simplify proofs and implementations; nevertheless the full set of connectives is more appropriate for making formulas readable by humans.

Negation Normal Form

We may restrict our considerations to restricted forms of propositional formulas.

- A literal is a propositional variable (a positive literal) or a negation of a propositional variable (a negative literal).
- A propositional formula is in negation normal form (NNF) if it is either one of the constants $\top$ or $\bot$ or a propositional formula constructed from literals by application of the connectives $\wedge$ and $\vee$.
  - Every propositional formula has a logically equivalent NNF.
  - Application of some of the previously stated equivalences.

Negations can be “pushed” down to the level of propositional variables, implications and equivalences can be replaced.

Negation Normal Form in OCaml

```
let rec nnf fm =
  match fm with
  | And(p,q) -> And(nnf p,nnf q) | Or(p,q) -> Or(nnf p,nnf q)
  | Imp(p,q) -> Or(nnf(Not p),nnf q)
  | Iff(p,q) -> Or(And(nnf p,nnf q),And(nnf(Not p),nnf(Not q)))
  | Not(Not p) -> nnf p | Not(And(p,q)) -> Or(nnf(Not p),nnf(Not q))
  | Not(Or(p,q)) -> And(nnf(Not p),nnf(Not q)) | Not(Imp(p,q)) -> And(nnf p,nnf(Not q))
  | Not(Iff(p,q)) -> Or(And(nnf p,nnf(Not q)),And(nnf(Not p),nnf q))
  | _ -> fm;;
let nnf fm = nnf(psimplify fm);;

# let fm = <<(p <=> q) <=> ~(r ==> s)>>;;
val fm : prop formula = <<(p <=> q) <=> ~(r ==> s)>>
# let fm' = nnf fm;;
val fm' : prop formula =
  <<(p /\ q \/ ~p /\ ~q) /\ r /\ ~s \/ (p /\ ~q \/ ~p /\ q) /\ (~r \/ s)>>
# tautology(Iff(fm,fm'));;
- : bool = true
```

Auxiliary Simplifications

```
let psimplify1 fm =
match fm with
  Not False -> True | Not True -> False | Not(Not p) -> p
| And(p,False) | And(False,p) -> False | And(p,True) | And(True,p) -> p
| Or(p,False) | Or(False,p) -> p | Or(p,True) | Or(True,p) -> True
| Imp(False,p) | Imp(p,True) -> True | Imp(True,p) -> p | Imp(p,False) -> Not p
| Iff(p,True) | Iff(True,p) -> p | Iff(p,False) | Iff(False,p) -> Not p
| _ -> fm;;

let rec psimplify fm =
match fm with
| Not p -> psimplify1 (Not(psimplify p))
| And(p,q) -> psimplify1 (And(psimplify p,psimplify q))
| Or(p,q) -> psimplify1 (Or(psimplify p,psimplify q))
| Imp(p,q) -> psimplify1 (Imp(psimplify p,psimplify q))
| Iff(p,q) -> psimplify1 (Iff(psimplify p,psimplify q))
| _ -> fm;;
```

Combination of “constant folding” and “short-circuit evaluation”.

Disjunctive and Conjunctive Normal Form

- A propositional formula is in disjunctive normal form (DNF) if it is a disjunction of conjunction of literals (or one of the constants $\bot$ or $\top$).
  - Every propositional formula has a logically equivalent DNF.
  - Construction from the “true” rows of the truth table (already shown).
- A propositional formula is in conjunctive normal form (CNF) if it is a conjunction of disjunction of literals (or one of the constants $\bot$ or $\top$).
  - Every propositional formula F has a logically equivalent CNF.
  - Dual construction from the “false” rows of the truth table.
    - Consider the DNF arising from the table rows that yield “false”.
    - This DNF is apparently equivalent to $\neg F$.
    - Thus we may construct from the negation of this DNF the CNF of F:

$$\neg((A_1 \wedge A_2) \vee (B_1 \wedge B_2)) \equiv (\neg A_1 \vee \neg A_2) \wedge (\neg B_1 \vee \neg B_2)$$

Construction of DNF/CNF from truth table has exponential complexity.

DNF and CNF via Truth Tables

p	q	r	$F = ?$
false	false	false	true
false	false	true	<u>false</u>
false	true	false	true
false	true	true	<u>false</u>
true	false	false	<u>false</u>
true	false	true	true
true	true	false	true
true	true	true	<u>false</u>

$$\begin{aligned} F_{\text{DNF}} = {} & (\neg p \wedge \neg q \wedge \neg r) \\ & \vee (\neg p \wedge q \wedge \neg r) \\ & \vee (p \wedge \neg q \wedge r) \\ & \vee (p \wedge q \wedge \neg r) \end{aligned}$$

$$\begin{aligned} F_{\text{CNF}} = {} & (p \vee q \vee \neg r) \\ & \wedge (p \vee \neg q \vee \neg r) \\ & \wedge (\neg p \vee q \vee r) \\ & \wedge (\neg p \vee \neg q \vee \neg r) \end{aligned}$$

Both the DNF and the CNF can be deduced from the truth table.

Set Based Representation of CNF/DNF (Basics)

- A CNF/DNF can be represented as a set of sets of literals.

$$\{\{p, q\}, \{\neg p, r\}\}$$

  - CNF: $(p \vee q) \wedge (\neg p \vee r)$;   DNF: $(p \wedge q) \vee (\neg p \wedge r)$.
  - Empty conjunction: $\top$, empty disjunction: $\bot$.

```
let distrib s1 s2 = setify(allpairs union s1 s2);;
let rec purednf fm = (* assumes that fm is in nnf *)
  match fm with
    And(p,q) -> distrib (purednf p) (purednf q)
  | Or(p,q) -> union (purednf p) (purednf q) | _ -> [[fm]];;
let purecnf fm = image (image negate) (purednf(nnf(Not fm)));;

# purednf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
- : prop formula list list =
[[<<p>>; <<~p>>]; [<<p>>; <<~r>>]; [<<q>>; <<r>>; <<~p>>]; [<<q>>; <<r>>; <<~r>>]]
# purecnf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
- : prop formula list list =
[[<<p>>; <<q>>]; [<<p>>; <<r>>]; [<<~p>>; <<~r>>]]
```

Set Based Representation of CNF/DNF (Optimizations)

Filter out literal sets that are trivial (contain contradictory literals) or are subsumed by other literal sets (are supersets of other literal sets).

```
let trivial lits =
  let pos,neg = partition positive lits in
  intersect pos (image negate neg) <> [];;
let simpdnf fm =
  if fm = False then [] else if fm = True then [[]] else
  let djs = filter (non trivial) (purednf(nnf fm)) in
  filter (fun d -> not(exists (fun d' -> psubset d' d) djs)) djs;;
let simpcnf fm =
  if fm = False then [[]] else if fm = True then [] else
  let cjs = filter (non trivial) (purecnf fm) in
  filter (fun c -> not(exists (fun c' -> psubset c' c) cjs)) cjs;;

simpdnf <<(p \/ q /\ r) /\ (~p \/ ~r)>>;;
# - : prop formula list list = [[<<p>>; <<~r>>]; [<<q>>; <<r>>; <<~p>>]]
```

Definitional CNF

Given a formula F, the size of a logically equivalent CNF C may be exponentially larger than the size of F.

- But we may construct a much smaller CNF C' that is just equisatisfiable with F.
  - C' is satisfiable if and only if F is satisfiable.
  - C' is generally *not* logically equivalent to F.
  - Nevertheless, C' suffices to check the satisfiability of F.
- Tseitin transformation: compute an equisatisfiable definitional CNF.
  - Replace every subformula with a binary connective by a fresh predicate symbol.
  - Define the predicate symbol by an equivalence with the subformula.
  - Conjoin the defining equivalences with the transformed formula.
  - Finally transform each part of the conjunction to a disjunction.

The size of the definitional CNF C' is just the size of the original formula F multiplied by some constant.

Example

$$
\begin{aligned}
& (p \lor \underline{(q \land \neg r)}) \land s \\
\rightarrow\; & (\underline{(p \lor u)} \land s) \land (u \Leftrightarrow q \land \neg r) \\
\rightarrow\; & \underline{(v \land s)} \land (u \Leftrightarrow q \land \neg r) \land (v \Leftrightarrow p \lor u) \\
\rightarrow\; & w \land (u \Leftrightarrow q \land \neg r) \land (v \Leftrightarrow p \lor u) \land (w \Leftrightarrow v \land s) \\
\rightarrow\; & w \land (\neg u \lor q) \land (\neg u \lor \neg r) \land (u \lor \neg q \lor r) \land \\
& (\neg v \lor p \lor u) \land (v \lor \neg p) \land (v \lor \neg u) \land (\neg w \lor v) \land (\neg w \lor s) \land (w \lor \neg v \lor \neg s)
\end{aligned}
$$

From $F \Leftrightarrow G \equiv (F \Rightarrow G) \land (G \Rightarrow F) \equiv (\neg F \lor G) \land (F \lor \neg G)$ we have the following equivalences:

$$
\begin{aligned}
F \Leftrightarrow (G \land H) &\equiv (\neg F \lor G) \land (\neg F \lor H) \land (F \lor \neg G \lor \neg H) \\
F \Leftrightarrow (G \lor H) &\equiv (\neg F \lor G \lor H) \land (F \lor \neg G) \land (F \lor \neg H) \\
F \Leftrightarrow (G \Rightarrow H) &\equiv (\neg F \lor \neg G \lor H) \land (F \lor G) \land (F \lor \neg H) \\
F \Leftrightarrow (G \Leftrightarrow H) &\equiv (\neg F \lor \neg G \lor H) \land (\neg F \lor G \lor \neg H) \land (F \lor G \lor H) \land (F \lor \neg G \lor \neg H)
\end{aligned}
$$

Definitional CNF in OCaml

- Basic form:

```
# defcnf0 <<(p \/ (q /\ ~r)) /\ s>>;;
 - : prop formula =
<<(p \/ p_1 \/ ~p_2) /\
  (p_1 \/ r \/ ~q) /\
  (p_2 \/ ~p) /\
  (p_2 \/ ~p_1) /\
  (p_2 \/ ~p_3) /\
  p_3 /\
  (p_3 \/ ~p_2 \/ ~s) /\ (q \/ ~p_1) /\ (s \/ ~p_3) /\ (~p_1 \/ ~r)>>
```

- With some optimizations:

```
# defcnf <<(p \/ (q /\ ~r)) /\ s>>;;
 - : prop formula =
<<(p \/ p_1) /\ (p_1 \/ r \/ ~q) /\ (q \/ ~p_1) /\ s /\ (~p_1 \/ ~r)>>
```