PROPOSITIONAL LOGIC: PROOFS

Course “Computational Logic”

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

JOHANNES KEPLER
UNIVERSITY LINZ

The Sequent Calculus

A system for proving the validity of propositional formulas (Gentzen, 1933).

- Judgements: "sequents" of form $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$
    - Finite (possibly empty) sequences of formulas $F_1, \ldots, F_m$ (the antecedent) and $G_1, \ldots, G_n$ (the succedent).
    - Read: "if *all* of $F_1, \ldots, F_m$ are true, then *at least one* of $G_1, \ldots, G_n$ is true".
- Inference rules: formal rules to derive sequents.
    - Soundness: if the sequent $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ can be derived, then the formula $F_1 \wedge \ldots \wedge F_m \Rightarrow G_1 \vee \ldots \vee G_n$ is valid.
        - Special case: If $\vdash F$ can be derived, then F is valid, i.e., $\models F$ holds.
    - Completeness: if the formula $F_1 \wedge \ldots \wedge F_m \Rightarrow G_1 \vee \ldots \vee G_n$ is valid, then the sequence $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ can be derived
        - Special case: If F is valid, i.e., $\models F$ holds, then $\vdash F$ can be derived.

A proof is the derivation of a sequent by the inference rules of the calculus.

Inference Rules

- Inference Rules:

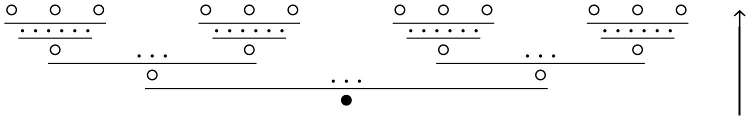
$$\frac{Fs_1 \vdash Gs_1 \quad \dots \quad Fs_p \vdash Gs_p}{Fs \vdash Gs} \text{ (rule)} \qquad \frac{}{Fs \vdash Gs} \text{ (axiom)}$$

  - Premises $Fs_i \vdash G_i$ and conclusion $Fs \vdash Gs$; axioms are rules without premises.
  - Forward reading: “If we can derive all premises, we have derived the conclusion.”
  - Backward reading: “To derive the conclusion, it suffices to derive all premises.”
- Pattern matching: antecedents and consequent may contain “meta-variables”.
  - Latin meta-variables $A, B, C, \dots$ will denote individual formulas.
  - Greek meta-variables $\Gamma, \Delta, \Lambda, \Phi, \dots$ will denote whole formula sequences.
  - Any instantiation of the meta-variables represents an instance of the rule that describes how to derive a concrete conclusion.

To apply a rule to a sequent of interest, we have to “match” this sequent against the conclusion of the rule; this determines the values of the meta-variables that have to be substituted in the premises to derive a valid instance of the rule.

Deduction Trees: Structure

A formal proof takes the form of a deduction tree:



- The root • represents the sequent to be derived by the deduction.
- Every node of the tree represents an instance of some inference rule.
- The tree is complete if all its leaves are instances of axioms.

The proof of a sequent is a complete deduction tree with the sequent as its root.

Deduction Trees: Construction

- Start with a tree whose single node $\bullet$ represents the sequent to be derived.
- Select some inference rule whose conclusion matches $\bullet$ and expand the tree by adding corresponding instances of the premises $\circ$:

$$\frac{\circ \quad \dots \quad \circ}{\bullet}$$

- Repeat the process with any leaf node that is still "open", i.e., that matches the conclusion of some inference rule that is not an instance of an axiom.

$$\frac{\dfrac{\circ \quad \dots \quad \circ}{\circ} \qquad \dots \quad \circ}{\bullet}$$

If all leaves have been "closed" (i.e., are instances of axioms), the proof is complete.

Deduction Trees: Pseudo-Code

```
procedure SEARCH($F_1, \ldots, F_m \vdash G_1, \ldots, G_n$)
    $T \leftarrow \langle F_1, \ldots, F_m \vdash G_1, \ldots, G_n \rangle$
    while $T$ has some open leaf node do
        choose some open leaf node $N$ in $T$
        EXPAND($N, T$)
    end while
    if $T$ is complete then
        WRITE("$T$ is a proof of the sequent")
    else
        choose some non-axiom leaf node $N$ in $T$
        WRITE("$N$ is a refutation of the sequent")
    end if
end procedure
```

```
procedure EXPAND($N, T$)
    Choose some inference rule $\dfrac{P_1 \quad \ldots \quad P_n}{C}$
        where $N = C[S]$ for some substitution $S$
    Replace $N$ by $\dfrac{P_1[S] \quad \ldots \quad P_n[S]}{N}$
end procedure
```

The procedure actually searches for a valuation that refutes $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$, i.e., that makes all of $F_1 \ldots, F_m$ true and all of $G_1, \ldots, G_n$ false.

The Sequent Calculus: Inference Rules

We replace $\bot$ by $(p \wedge \neg p)$, $\top$ by $(p \vee \neg p)$, $(A \Leftrightarrow B)$ by $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

$$\frac{}{\Gamma, A, \Delta \vdash \Lambda, A, \Phi}\ (\mathrm{AX})$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda}{\Gamma, \neg A, \Delta \vdash \Lambda}\ (\neg\text{-L}) \qquad \frac{A, \Gamma \vdash \Delta, \Lambda}{\Gamma \vdash \Delta, \neg A, \Lambda}\ (\neg\text{-R})$$

$$\frac{\Gamma, A, B, \Delta \vdash \Lambda}{\Gamma, A \wedge B, \Delta \vdash \Lambda}\ (\wedge\text{-L}) \qquad \frac{\Gamma \vdash \Delta, A, \Lambda \quad \Gamma \vdash \Delta, B, \Lambda}{\Gamma \vdash \Delta, A \wedge B, \Lambda}\ (\wedge\text{-R})$$

$$\frac{\Gamma, A, \Delta \vdash \Lambda \quad \Gamma, B, \Delta \vdash \Lambda}{\Gamma, A \vee B, \Delta \vdash \Lambda}\ (\vee\text{-L}) \qquad \frac{\Gamma \vdash \Delta, A, B, \Lambda}{\Gamma \vdash \Delta, A \vee B, \Lambda}\ (\vee\text{-R})$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda \quad \Gamma, B, \Delta \vdash \Lambda}{\Gamma, A \Rightarrow B, \Delta \vdash \Lambda}\ (\Rightarrow\text{-L}) \qquad \frac{A, \Gamma \vdash \Delta, B, \Lambda}{\Gamma \vdash \Delta, A \Rightarrow B, \Lambda}\ (\Rightarrow\text{-R})$$

One rule for every connective on the left respectively right side of the sequent.

The Sequent Calculus: Example Proof

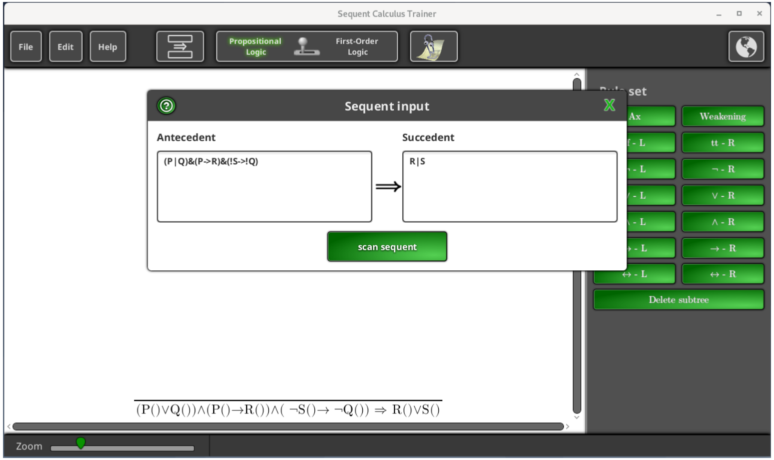
Proof of $(p \vee q) \wedge (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \Rightarrow r \vee s$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p, (\neg s \Rightarrow \neg q) \vdash p, r, s}\;(\text{AX}) \qquad \dfrac{}{p, r, (\neg s \Rightarrow \neg q) \vdash r, s}\;(\text{AX})}{p, (p \Rightarrow r), (\neg s \Rightarrow \neg q) \vdash r, s}\;(\Rightarrow\text{-L})}{p, (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \vdash r, s}\;(\wedge\text{-L}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q, s \vdash p, r, s}\;(\text{AX})}{q \vdash \neg s, p, r, s}\;(\neg\text{-R}) \quad \dfrac{\dfrac{}{q \vdash p, q, r, s}\;(\text{AX})}{q, \neg q \vdash p, r, s}\;(\neg\text{-L})}{q, (\neg s \Rightarrow \neg q) \vdash p, r, s}\;(\Rightarrow\text{-L}) \qquad \dfrac{}{q, r, (\neg s \Rightarrow \neg q) \vdash r, s}\;(\text{AX})}{q, (p \Rightarrow r), (\neg s \Rightarrow \neg q) \vdash r, s}\;(\Rightarrow\text{-L})}{q, (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \vdash r, s}\;(\wedge\text{-L})}{(p \vee q), (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \vdash r, s}\;(\vee\text{-L})}{(p \vee q) \wedge (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \vdash r, s}\;(\wedge\text{-L})}{(p \vee q) \wedge (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \vdash r \vee s}\;(\vee\text{-R})}{\vdash (p \vee q) \wedge (p \Rightarrow r) \wedge (\neg s \Rightarrow \neg q) \Rightarrow r \vee s}\;(\Rightarrow\text{-R})
$$

In each sequent, there may exist multiple possibilities for rule application; while the choice may influence the complexity of the proof, it does not (as we will see later) affect the final success or failure.

The Sequent Calculus Trainer

```
debian10!1> sct &
```


Sequent Calculus Trainer

Sequent input

Antecedent

```
(P|Q)&(P->R)&(!S->!Q)
```

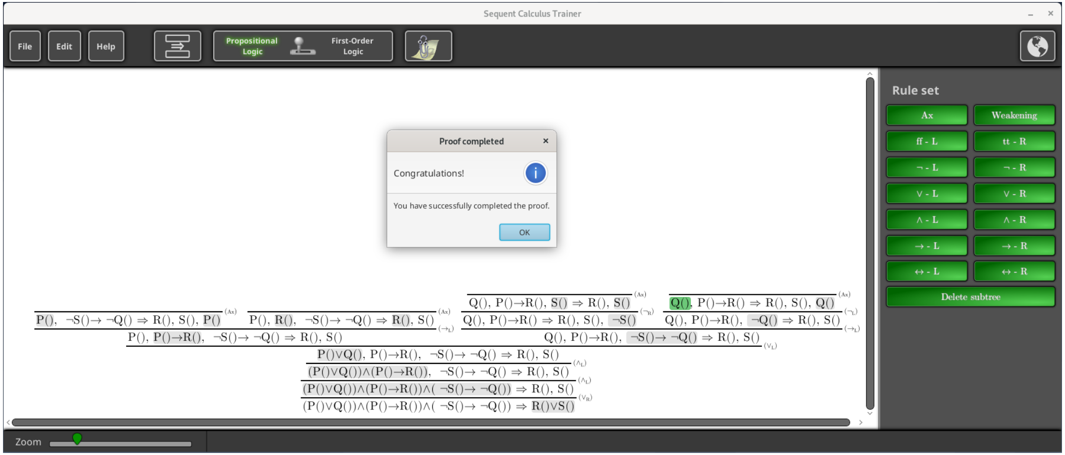
Succedent

```
R|S
```

$$(P()\vee Q())\wedge(P()\to R())\wedge(\neg S()\to \neg Q()) \Rightarrow R()\vee S()$$

Sequents are separated by ⇒ (implication is denoted by ⟶).

The Sequent Calculus Trainer



Proof completed

Congratulations!

You have successfully completed the proof.

$$\frac{\frac{\frac{}{P(),\ \neg S()\to\neg Q() \Rightarrow R(), S(), P()}(Ax) \quad \frac{}{P(), R(),\ \neg S()\to\neg Q() \Rightarrow R(), S()}(Ax)}{P(), P()\to R(),\ \neg S()\to\neg Q() \Rightarrow R(), S()}(\to_L) \quad \frac{\frac{\frac{}{Q(), P()\to R(), S() \Rightarrow R(), S()}(Ax)}{Q(), P()\to R() \Rightarrow R(), S(), \neg S()}(\neg_R) \quad \frac{\frac{}{Q(), P()\to R() \Rightarrow R(), S(), Q()}(Ax)}{Q(), P()\to R(), \neg Q() \Rightarrow R(), S()}(\neg_L)}{Q(), P()\to R(),\ \neg S()\to\neg Q() \Rightarrow R(), S()}(\to_L)}{\frac{\frac{\frac{P()\vee Q(), P()\to R(),\ \neg S()\to\neg Q() \Rightarrow R(), S()}{(P()\vee Q())\wedge(P()\to R()),\ \neg S()\to\neg Q() \Rightarrow R(), S()}(\wedge_L)}{(P()\vee Q())\wedge(P()\to R())\wedge(\neg S()\to\neg Q()) \Rightarrow R(), S()}(\wedge_L)}{(P()\vee Q())\wedge(P()\to R())\wedge(\neg S()\to\neg Q()) \Rightarrow R()\vee S()}(\vee_R)}(\vee_L)$$

Apply rules ¬-L and ¬-R for swapping negated formulas.

The Sequent Calculus in OCaml

```
let rec axiom lfms rfms lats rats level =
  (exists (fun lfm -> exists (fun rfm -> lfm=rfm) (rfms@rats)) (lfms@lats))
and lprove lfm lfms rfms lats rats level =
  match lfm with
    And(a,b) -> (gprove (a::b::lfms) rfms lats rats (level+1))
  | Or(a,b)  -> (gprove (a::lfms) rfms lats rats (level+1))
             && (gprove (b::lfms) rfms lats rats (level+1))
  | Imp(a,b) -> (gprove lfms (a::rfms) lats rats (level+1))
             && (gprove (b::lfms) rfms lats rats (level+1))
  | Not(a)   -> (gprove lfms (a::rfms) lats rats (level+1))
  | _        -> failwith "unsupported formula"
and rprove rfm lfms rfms lats rats level =
  match rfm with
    And(a,b) -> (gprove lfms (a::rfms) lats rats (level+1))
             && (gprove lfms (b::rfms) lats rats (level+1))
  | Or(a,b)  -> (gprove lfms (a::b::rfms) lats rats (level+1))
  | Imp(a,b) -> (gprove (a::lfms) (b::rfms) lats rats (level+1))
  | Not(a)   -> (gprove (a::lfms) rfms lats rats (level+1))
  | _        -> failwith "unsupported formula"
...
```

The Sequent Calculus in OCaml

```
...
and gprove lfms rfms lats rats level =
  (axiom lfms rfms lats rats level) &&
  (print_sequent level "A:" lfms rfms lats rats; true)
|| match rfms with
   [] -> (match lfms with
            [] -> print_sequent level "F:" lfms rfms lats rats; false
          | lfm::lfms0 ->
              (match lfm with
                Atom(_) -> (gprove lfms0 rfms (lfm::lats) rats level)
              | _ -> print_sequent level "L:" lfms rfms lats rats;
                     (lprove lfm lfms0 rfms lats rats level)))
   | rfm::rfms0 ->
       match rfm with
         Atom(_) -> (gprove lfms rfms0 lats (rfm::rats) level)
       | _ -> print_sequent level "R:" lfms rfms lats rats;
              (rprove rfm lfms rfms0 lats rats level) ;;
let gprove f = gprove [] [f] [] [] 0 ;;
```

The Sequent Calculus in OCaml

```
# gprove << (p \/ q) /\ (p ==> r) /\ (~s ==> ~q) ==> r \/ s >> ;;

R: |- <<(p \/ q) /\ (p ==> r) /\ (~s ==> ~q) ==> r \/ s>>
  R:<<(p \/ q) /\ (p ==> r) /\ (~s ==> ~q)>> |- <<r \/ s>>
    L:<<(p \/ q) /\ (p ==> r) /\ (~s ==> ~q)>> |- <<s>><<r>>
      L:<<p \/ q>><<(p ==> r) /\ (~s ==> ~q)>> |- <<s>><<r>>
        L:<<(p ==> r) /\ (~s ==> ~q)>><<p>> |- <<s>><<r>>
          L:<<p ==> r>><<~s ==> ~q>><<p>> |- <<s>><<r>>
            A:<<~s ==> ~q>><<p>> |- <<p>><<s>><<r>>
            A:<<r>><<~s ==> ~q>><<p>> |- <<s>><<r>>
        L:<<(p ==> r) /\ (~s ==> ~q)>><<q>> |- <<s>><<r>>
          L:<<p ==> r>><<~s ==> ~q>><<q>> |- <<s>><<r>>
            L:<<~s ==> ~q>><<q>> |- <<p>><<s>><<r>>
              R:<<q>> |- <<~s>><<p>><<s>><<r>>
                A:<<s>><<q>> |- <<p>><<s>><<r>>
              L:<<~q>><<q>> |- <<p>><<s>><<r>>
                A:<<q>> |- <<q>><<p>><<s>><<r>>
            A:<<r>><<~s ==> ~q>><<q>> |- <<s>><<r>>- : bool = true
```

A successful proof generates a complete deduction tree.

The Sequent Calculus in OCaml

```
# tautology << (p ==> q \/ r) /\ (r ==> s) ==> (p /\ q ==> s) >> ;;
 - : bool = false

# gprove << (p ==> q \/ r) /\ (r ==> s) ==> (p /\ q ==> s) >> ;;

R: |- <<(p ==> q \/ r) /\ (r ==> s) ==> p /\ q ==> s>>
  R:<<(p ==> q \/ r) /\ (r ==> s)>> |- <<p /\ q ==> s>>
    L:<<p /\ q>><<(p ==> q \/ r) /\ (r ==> s)>> |- <<s>>
      L:<<(p ==> q \/ r) /\ (r ==> s)>><<q>><<p>> |- <<s>>
        L:<<p ==> q \/ r>><<r ==> s>><<q>><<p>> |- <<s>>
          A:<<r ==> s>><<q>><<p>> |- <<p>><<s>>
          L:<<q \/ r>><<r ==> s>><<q>><<p>> |- <<s>>
            L:<<r ==> s>><<q>><<q>><<p>> |- <<s>>
              F:<<q>><<q>><<p>> |- <<r>><<s>>- : bool = false
```

A failed proof produces a refuting valuation.

Soundness of the Sequent Calculus

We identify $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ with the formula $F_1 \wedge \ldots \wedge F_m \Rightarrow G_1 \vee \ldots \vee G_n$ and thus generalize the semantical notions from formulas to sequents.

- Theorem: the root of every complete deduction tree denotes a valid sequent.
- Proof sketch: we have to show for every inference rule that, if valuation v satisfies all premises of the rule, v also satisfies its conclusion. E.g., consider the following case (the others are similar):

  $$\frac{\Gamma, \Delta \vdash A, \Lambda \quad B, \Gamma, \Delta \vdash \Lambda}{\Gamma, A \Rightarrow B, \Delta \vdash \Lambda} \ (\Rightarrow\text{-L})$$

  We assume that valuation v does not satisfy the conclusion and show that it it does not satisfy some premise. Since v does not satisfy the conclusion, it satisfies all formulas in Γ and Δ as well as $A \Rightarrow B$ but falsifies all formulas in Λ. From the semantics of $A \Rightarrow B$, we have two cases:

  - v falsifies A. Since v satisfies Γ and Δ but not A and Λ, v does not satisfy the first premise.
  - v satisfies B. Since v satisfies B, Γ, Δ but not Λ, v does not satisfy the second premise. $\square$

Completeness of the Sequent Calculus

- Theorem (Completeness): if a sequent is valid, then there exists a complete deduction tree whose root is this sequent.
  - Proof: a consequence of Part 1 of the following theorem.
- Theorem (Decidability): SEARCH decides the validity of every sequent.
  1. If the sequent is valid, SEARCH terminates with a complete deduction tree for it.
  2. If the sequent is invalid, SEARCH terminates with an incomplete deduction tree whose open leaf nodes denote valuations that falsify the sequent.
  - Proof sketch: see next slide.

SEARCH represents a decision procedure for propositional logic.

Proof of Decidability

Proof sketch: First, in every application of EXPAND, the number of logical connectives that occur in the instances of every premise of the selected rule is strictly smaller than in the conclusion. Thus the procedure must terminate.

Part 2 of the theorem: for every valuation v that falsifies the sequent, the soundness of the inference rules applied in EXPAND implies that v also falsifies some leaves of the deduction tree. Every such leaf node N cannot be an instance of the axiom of the calculus. However, since SEARCH only terminates when every leaf node is closed and there is an inference rule for every possible occurrence of a connective, N cannot contain any logical connective but must have form $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ where all formulas are atoms. Furthermore, since N is not an instance of the axiom, the sets $\{F_1, \ldots, F_m\}$ and $\{G_1, \ldots, G_n\}$ must be disjoint. Since v falsifies N, it satisfies all of $\{F_1, \ldots, F_m\}$ and falsifies all of $\{G_1, \ldots, G_n\}$; thus we can deduce v from N.

Part 1 of the theorem: consider a valid sequent. We assume that SEARCH terminates with an incomplete deduction tree, i.e., there exists some leaf node N that is not an instance of the axiom, and show a contradiction. Since EXPAND only applies sound inference rules, N must denote a valid sequent. However, since SEARCH terminates, although N is not an instance of the axiom, N must have form $F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ with disjoint sets $\{F_1, \ldots, F_m\}$ and $\{G_1, \ldots, G_n\}$. Then however we can construct a valuation that falsifies N which contradicts the validity of N.

□

The Resolution Method

A system for showing the unsatisfiability of formulas in CNF (Robinson, 1965).

- CNF: conjunctive normal form (in the "set of sets" representation).
  - Formula $F = \{C_1, \ldots, C_n\}$ interpreted as $F = C_1 \wedge \ldots \wedge C_n$, clause $C = \{L_1, \ldots, L_m\}$ interpreted as $C = L_1 \vee \ldots \vee L_m$, literal $L = p$ or $L = \neg p$.
- Judgement: $F \vdash$.
  - $F = \{C_1, \ldots, C_n\}$ in CNF.
  - Interpretation: "formula $F \Rightarrow \bot$ is valid".
    - Equivalent to: "formula $\neg F$ is valid".
    - Equivalent to: "formula F is unsatisfiable".

Refutation proof: to prove the validity of a formula F, we derive the judgement $\{C_1, \ldots, C_n\} \vdash$ for the CNF $\{C_1, \ldots, C_n\}$ of $\neg F$.

Inference Rules

$$\frac{\{\,\} \in F}{F \vdash}\ \text{(AX)} \qquad \frac{C \cup \{p\} \in F \quad D \cup \{\neg p\} \in F \quad F \cup \{C \cup D\} \vdash}{F \vdash}\ \text{(RES)}$$

- Terminology:
  - Rule (RES) describes the derivation of the resolvent $C \cup D$ by resolution of the clauses $C \cup \{p\}$ and $D \cup \{\neg p\}$ on literal p.
  - Resolution step: the derivation $C \cup \{p\}, D \cup \{\neg p\} \to C \cup D$.
- Soundness:
  - Rule (AX): $F \cup \{\{\,\}\}$ denotes the formula $F \wedge \bot$ which is unsatisfiable.
  - Rule (RES): we have the logical consequence $(C \vee p) \wedge (D \vee \neg p) \models C \vee D$ and thus the logical equivalence $(C \vee p) \wedge (D \vee \neg p) \equiv (C \vee p) \wedge (D \vee \neg p) \wedge (C \vee D)$. Therefore, if the premise is unsatisfiable, also the conclusion is.
- Completeness: the inference system is complete (see literature for details).
  - By applications of rule (RES), all consequences of the clause set can be derived.

A very compact (but not yet very efficient) calculus.

Example

We show the validity of $(p \Rightarrow (q \Rightarrow r)) \wedge (p \Rightarrow q) \wedge p \Rightarrow r$.

- We show the unsatisfiability of $(p \Rightarrow (q \Rightarrow r)) \wedge (p \Rightarrow q) \wedge p \wedge \neg r$.
- We derive a refutation proof for the clause set $\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}$.

$$\begin{tikzcd}
\{\neg p, \neg q, r\} \arrow[ddr] & & \{\neg p, q\} \arrow[drr] & & & & \{p\} \arrow[dll] \arrow[ddddl] & \{\neg r\} \arrow[dddllll] \\
& & & & \{q\} \arrow[dlll] & & & \\
& \{\neg p, r\} \arrow[drr] & & & & & & \\
& & & \{\neg p\} \arrow[drr] & & & & \\
& & & & & \{\} & &
\end{tikzcd}$$

Refutation proofs can be best visualized as directed acyclic graphs.

The Davis-Putnam Algorithm

A more efficient application of the resolution principle (Davis, Putnam, 1960).

```
function DP($F$)                                                    ▷ returns true, if clause set $F$ is satisfiable
    loop
        if $F = \{\ \}$ then return true
        if $\{\ \} \in F$ then return false
        if there is some literal $L$ and $C \in F$ with $C = \{L\}$ then                    ▷ unit propagation
            remove from $F$ every clause that contains $L$ and from every clause in $F$ the negation of $L$
        else if there is a literal $L$ such that no clause in $F$ contains its negation then     ▷ pure literal elimination
            remove from $F$ every clause that contains $L$
        else                                       ▷ resolution of all pairs of clauses on any literal $p$, discarding $p$
            choose $p, Cs, Ds, Es, Fs$ with $\bigcup(Cs \cup Ds \cup Es \cup Fs) \cap \{p, \neg p\} = \{\} \wedge$
              $F = \{C \cup \{p\} \mid C \in Cs\} \cup \{D \cup \{\neg p\} \mid D \in Ds\} \cup Es \cup \{F \cup \{p, \neg p\} \mid F \in Fs\}$
            $F \leftarrow \{C \cup D \mid C \in Cs, D \in Ds\} \cup Es$
        end if
    end loop
end function
```

Resolution now reduces number of propositional variables in formula; pure literal elimination ensures the applicability of resolution; unit propagation speeds up the execution.

Unit Propagation and Pure Literal Elimination: Example

- Unit Propagation: "a unit clause determines the truth value of its literal."

$$
\begin{array}{rl}
 & \{\neg p, \neg q, r\}, \{\neg p, q\}, \underline{\{p\}}, \{\neg r\} \\
\rightarrow & \{\neg q, r\}, \underline{\{q\}}, \{\neg r\} \\
\rightarrow & \underline{\{r\}}, \{\neg r\} \\
\rightarrow & \{\ \} \quad \text{(formula with empty clause, unsatisfiable)}
\end{array}
$$

- Pure Literal Elimination: "if no clause opposes, we may set a literal to true".

$$
\begin{array}{rl}
 & \{\neg p, q, \underline{r}\}, \{p, \underline{r}\}, \{\neg p, \neg q\}, \{\neg p, q, s\}, \{\neg q, \neg s\} \\
\rightarrow & \{\underline{\neg p}, \neg q\}, \{\underline{\neg p}, q, s\}, \{\neg q, \neg s\} \\
\rightarrow & \{\underline{\neg q}, \neg s\} \\
\rightarrow & \_ \quad \text{(empty formula, satisfiable)}
\end{array}
$$

These special rules may quickly decide satisfiability without requiring resolution.

Unit Propagation and Pure Literal Elimination: Soundness

These rules transform the formula F to some formula F' that does not contain literal L or its negation. We assume $L := p$ for some atom p and show that F is satisfiable if and only if F' is satisfiable.

- Unit Propagation: this rule transforms a formula $F \cup \{\{p\}\}$.
  - Assume v satisfies $F \cup \{\{p\}\}$, i.e., v satisfies F and $v(p) = \text{true}$. We show that v also satisfies F'. For this, we take arbitrary clause C' in F' and show that v satisfies C'. By the construction of F', there is a clause C in F from which C' is derived; since v satisfies F, v also satisfies C. From the rule, we have two cases. Either C' is C, then v satisfies C'. Otherwise, C' is C without $\neg p$. Since v satisfies p, it does not satisfy $\neg p$. But then, since v satisfies C, v must satisfy C'.
  - We assume that v' satisfies F' and show that v satisfies $F \cup \{\{p\}\}$ where $v = v'$ except $v(p) = \text{true}$. Thus v satisfies $\{p\}$; it remains to show that v satisfies every clause C in F. If C contains p, this is clearly the case. If C does not contain p, F' contains C without $\neg p$. Since v' satisfies F', v' satisfies some literal q of C different from $\neg p$, i.e., $v'(q) = \text{true}$. Since $v(q) = v'(q)$, also v satisfies C.
- Pure Literal Elimination: this rule transforms a formula F that contains p but not $\neg p$.
  - Assume v satisfies F. Since every clause of F' is also in F, v also satisfies F'.
  - Assume v' satisfies F'. Let $v = v'$ except $v(p) = \text{true}$. We show that v satisfies every clause C of F. If C is not in F', C contains p, thus v satisfies C. If C is in F', v' satisfies C and C does not contain p. Thus C contains another literal q with $v'(q) = \text{true}$. Since $v(q) = v'(q)$, v satisfies C.

The application of the rules does not affect the satisfiability of the formula.

The Davis-Putnam Algorithm in OCaml

```
let one_literal_rule clauses =
  let u = hd (find (fun cl -> length cl = 1) clauses) in
  let u' = negate u in
  let clauses1 = filter (fun cl -> not (mem u cl)) clauses in
  image (fun cl -> subtract cl [u']) clauses1;;

let affirmative_negative_rule clauses =
  let neg',pos = partition negative (unions clauses) in
  let neg = image negate neg' in
  let pos_only = subtract pos neg and neg_only = subtract neg pos in
  let pure = union pos_only (image negate neg_only) in
  if pure = [] then failwith "affirmative_negative_rule" else
  filter (fun cl -> intersect cl pure = []) clauses;;

let resolution_blowup cls l =
  let m = length(filter (mem l) cls)
  and n = length(filter (mem (negate l)) cls) in
  m * n - m - n;;
```

The Davis-Putnam Algorithm in OCaml

```
let resolve_on p clauses =
  let p’ = negate p and pos,notpos = partition (mem p) clauses in
  let neg,other = partition (mem p’) notpos in
  let pos’ = image (filter (fun l -> l <> p)) pos
  and neg’ = image (filter (fun l -> l <> p’)) neg in
  let res0 = allpairs union pos’ neg’ in
  union other (filter (non trivial) res0);;

let resolution_rule clauses =
  let pvs = filter positive (unions clauses) in
  let p = minimize (resolution_blowup clauses) pvs in
  resolve_on p clauses;;

let rec dp clauses =
  if clauses = [] then true else if mem [] clauses then false else
  try dp (one_literal_rule clauses) with Failure _ ->
  try dp (affirmative_negative_rule clauses) with Failure _ ->
  dp(resolution_rule clauses);;
```

The Davis-Putnam Algorithm in OCaml

```
let dpsat fm = dp(defcnfs fm);;
let dptaut fm = not(dpsat(Not fm));;

# dptaut << (p ==> (q ==> r)) /\ (p ==> q) /\ p ==> r >> ;;
 - : bool = true
# tautology << (p ==> (q ==> r)) /\ (p ==> q) /\ p ==> r >> ;;
 - : bool = true
```

The fastest of the decision methods discussed *so far* (a better one is to come).