First-order Logic: Theory and Practice

The DPLL Calculus

Jens Otten

University of Potsdam

Outline

The DPLL Calculus

The DPLL (Davis-Putnam-Logemann-Loveland) calculus is the most popular and successful proof method for propositional classical logic.

- developed by Davis and Putnam in 1960 (for first-order classical logic)
- improved later by Davis, Logemann, Loveland

Main idea of the DPLL procedure:

- a propositional formula F is valid (i.e. a tautology), if its evaluate to true for all possible truth assignments to its propositional letters $X \in L$ (i.e. $v(F) = \texttt{true}$ for all assignments $v : L \rightarrow \{\texttt{false}, \texttt{true}\}$)
- step 1: choose a propositional letter X occurring in F
  step 2a: replace X with $\top$ (`true`) $\rightsquigarrow F_1$; simplify F_1 and continue
  step 2b: replace X with $\bot$ (`false`) $\rightsquigarrow F_2$; simplify F_2 and continue
- F is valid iff F_1 and F_2 are valid; stop if formula F is $\top$ or $\bot$

The Formal DPLL Calculus

The formal DPLL calculus consists of the following axiom and rules ($F[G\backslash G']$ is the formula F in which the subformula G is replaced by G').

- Axiom $$\frac{}{\top}$$
- Splitting $$\frac{F[X\backslash\top] \qquad F[X\backslash\bot]}{F}$$
- Simplification $$\frac{F[(G\vee\top)\backslash\top]}{F} \qquad \frac{F[(G\wedge\top)\backslash G]}{F}$$ $$\frac{F[(G\wedge\bot)\backslash\bot]}{F} \qquad \frac{F[(G\vee\bot)\backslash G]}{F}$$

Theorem: F is valid iff there is a proof for F in the DPLL calculus.

- proof of Theorem is straightforward
- DPLL calculus is confluent, i.e. no backtracking (over rules) necessary

Representing Formulae in Clausal Form

The DPLL calculus is usually applied to formulae F in clausal form, i.e. to formulae in disjunctive normal form.

- formula $L_1 \wedge ... \wedge L_m$ is represented by the clause $C = \{L_1, ..., L_m\}$
- formula $C_1 \vee ... \vee C_n$ is represented by the matrix $M = \{C_1, ..., C_n\}$

Each formula F is represented by its matrix M and the rules of the DPLL calculus are adapted accordingly.

- $\top$ in the axiom is replaced by $\{..., \{\,\}, ...\}$
- all $\top$ (`true`) literals are deleted from all clauses (literal deletion)
- all clauses that contain $\bot$ (`false`) are deleted from M (clause deletion)
- literal and clause deletion are integrated into the splitting rule, i.e. delete literal X (or $\neg X$) and delete all clauses that contain $\neg X$ (or X)

Representing Formulae in Clausal Form – Example

Example: $((\neg A \vee \neg B) \wedge C \Rightarrow \neg(C \Rightarrow A \wedge B)) \wedge (D \vee \neg D)$

$$((\neg A \vee \neg B) \wedge C \Rightarrow \neg(C \Rightarrow A \wedge B)) \wedge (D \vee \neg D)$$

$\downarrow$ translation into clausal form

$$(A \wedge B \wedge D) \vee (A \wedge B \wedge \neg D) \vee (\neg C \wedge D) \vee (\neg C \wedge \neg D) \vee (C \wedge \neg A \wedge D)$$
$$\vee (C \wedge \neg A \wedge \neg D) \vee (C \wedge \neg B \wedge D) \vee (C \wedge \neg B \wedge \neg D)$$

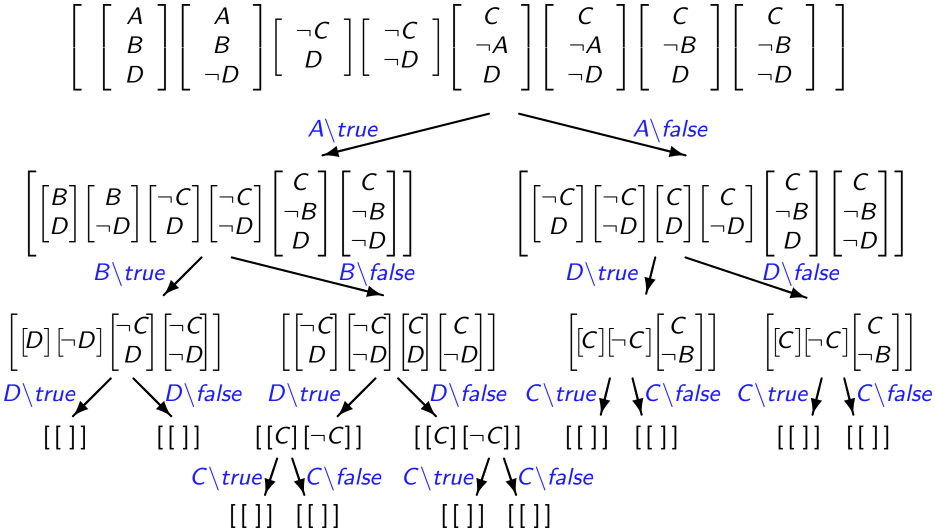
$\downarrow$ representation as matrix

$$\{\{A, B, D\}, \{A, B, \neg D\}, \{\neg C, D\}, \{\neg C, \neg D\}, \{C, \neg A, D\},$$
$$\{C, \neg A, \neg D\}, \{C, \neg B, D\}, \{C, \neg B, \neg D\}\}$$

$\downarrow$ graphical matrix representation

$$\left[\begin{bmatrix} A \\ B \\ D \end{bmatrix} \begin{bmatrix} A \\ B \\ \neg D \end{bmatrix} \begin{bmatrix} \neg C \\ D \end{bmatrix} \begin{bmatrix} \neg C \\ \neg D \end{bmatrix} \begin{bmatrix} C \\ \neg A \\ D \end{bmatrix} \begin{bmatrix} C \\ \neg A \\ \neg D \end{bmatrix} \begin{bmatrix} C \\ \neg B \\ D \end{bmatrix} \begin{bmatrix} C \\ \neg B \\ \neg D \end{bmatrix} \right]$$

The DPLL Calculus – Example



An Introduction to Prolog

Prolog (Programming in Logic) is a declarative programming language.

- invented in the 1970th by A. Colmerauer, R. Kowalski, and P. Roussel.
- declarative programming: specify the problem and let the computer solve it; algorithm = logic + control (Kowalski '79)

Problem is specified using facts and rules, which are stored in a database.

- a literal L has the the form "*predicateNAME*(*Arg1*,...,*ArgN*)."
- facts have the form "`L.`"; rules have the form "`L :- L1, ..., Ln.`"

Example:
```
factorial(0,1).
factorial(N,N1) :- N>0, N2 is N-1,
                   factorial(N2,N3), N1 is N*N3.
```

- user starts the program by entering a query of the form "`?- L.`"

Example: `?- factorial(6,N).` $\rightsquigarrow$ Prolog answers: `yes, N=720.`

Semantics of Prolog

The semantics of a program is specified by the following formula F.

`fact_1.`	`( fact_1`
`...`	$\wedge$ `...`
`fact_n.`	$\wedge$ `fact_n`
`head_1 :- body_1.`	$\wedge$ `head_1` $\Leftarrow$ `body_1`
`...`	$\wedge$ `...`
`head_m :- body_m.`	$\wedge$ `head_m` $\Leftarrow$ `body_m )`
`?- query.`	$\Rightarrow$ `query`

The `query` succeeds iff the Prolog program terminates and F is valid.

- inference engine is a theorem prover based on SL-resolution (only horn clauses, depth-first search (incomplete!), no occurs-check)
- additional concepts: predefined predicates, `!` ("cut"), if-then-else, unification, arithmetic, lists, ...
- lists: `[a,b,c,d]=[a|[b,c,d]]`, `member(c,[a,b,c,d])`

An Implementation: leanDPLL

Prolog list are used to represent sets.

- clause $\{L1, \ldots, Ln\}$ is represented by the Prolog list `[L1,...,Ln]`
- matrix $\{C1, \ldots, Cm\}$ is represented by the Prolog list `[C1,...,Cm]`
- propositional letters start with a small letter in Prolog
- negation $\neg$ is represented in Prolog by `-`
- Example: the matrix $\{\{A\}, \{\neg A, B\}, \{\neg B\}\}$ is represented by the list `[[a],[-a,b],[-b]]`

The DPLL implementation uses three Prolog predicates/functions.

- `dpll(M)` succeeds, if the formula represented by the matrix `M` is valid
- `reduce(M,L,N,M1)` succeeds, if the matrix `M1` is the result of deleting all occurrences of the literal `L` and all clauses containing the negated literal `N` of L from the matrix `M`
- `del(A,L,L1)` succeeds, if the list `L1` is the results of deleting `A` from `L`

An Implementation: leanDPLL

The following code implements the DPLL calculus in Prolog.

- Axiom

```
dpll(M) :- member([],M), !.
```

- Splitting

```
dpll(M) :- M=[[L|_]|_], (-N=L;-L=N) ->
           reduce(M,L,N,M1), dpll(M1),
           reduce(M,N,L,M2), dpll(M2).
```

- delete all occurrences of L and all clauses containing N

```
reduce([],_,_,[]).
reduce([C|M],L,N,M1) :-
    ( member(N,C) -> M1=M2 ; del(L,C,D), M1=[D|M2] ),
    reduce(M,L,N,M2).
```

- delete element from list

```
del(_,[],[]).
del(A,[B|C],D) :- (A=B -> D=E ; D=[B|E]), del(A,C,E).
```

Optimizations

If a clause contains only one literal L, then no splitting is required for L.

- this can reduce the size of a proof significantly
- it can be implemented by adding another Prolog clause to the code

```
dpll(M) :- member([L],M), (-N=L;-L=N), !,
           reduce(M,N,L,M1), dpll(M1).
```

ncDP is a Prolog implementation of a non-clausal DPLL calculus.

- no translation to clausal form necessary; uses a nested matrix
- uses an additional splitting rule that can reduce the size of proofs
- more details and download of code at `www.leancop.de/ncdp/`

Modern DPLL/SAT provers contain many more optimization techniques.

Instance-Based Methods

Instance-based methods lift the propositional DPLL calculus to first-order logic; in general, they consist of two components:

1. step: generate and add clause instances to the formula and ground it (remove quantifiers, replace variables by a single constant)
2. step: use propositional DPLL prover to find proof or counter-model; if no proof is found, go to first step and generate more instances

Example: $\{\{\neg Pa\}, \{Px, \neg Pfx\}, \{Pffx\}\}$

- first instance is not valid: $\{\{\neg Pa\}, \{Pa, \neg Pfa\}, \{Pffa\}\}$
- second instance is valid: $\{\{\neg Pa\}, \{Pa, \neg Pfa\}, \{Pfa, \neg Pffa\}, \{Pffa\}\}$

Successful implementations: iProver, Equinox, Darwin, ...

- good performance for "essentially propositional" formulae
- they use powerful SAT provers/solvers, e.g., miniSAT