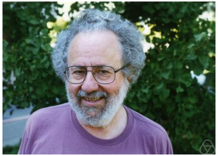
Informatics 1 – Introduction to Computation
Computation and Logic

Julian Bradfield

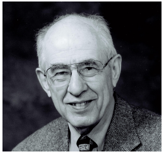
based on materials by

Michael P. Fourman

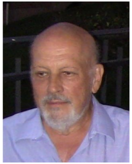
Finding Satisfying Assignments
DPLL



Martin Davis, 1928–
Photo: George Bergman



Hilary Putnam, 1926–2016



George Logemann, 1938–2012
Donald Loveland, 1934–

CNF language in Haskell

Last week in CL we looked at Karnaugh Maps as a way to convert boolean expressions to DNF or CNF.

Last week in FP you learned how to represent formal languages (arithmetic expressions and boolean propositions (WFFs)) in Haskell.

Now we will implement boolean propositions *in CNF* and use them to solve problems.

Recall that in CNF, a formula is a *conjunction* of *clauses*; a clause is a *disjunction* of *literals*; a literal is either an *atom* or a negated *atom*; and an atom is a basic boolean proposition.

CNF: the basic datatypes

Here is a simple implementation:

```
-- this lets us choose any Atom type without redefining things
data Literal atom = P atom | N atom
  -- positive and negative literals
data Clause atom = Or [ Literal atom ]
  -- "Or"  is a data constructor, no connection to "or" except
  -- in our heads
data Form atom = And[ Clause atom ]
-- and here is a simple atom type
data Atom = A|B|C|D|W|X|Y|Z deriving (Eq,Show)
  -- we have to be able to compare atoms
```

In practice we'll have everything deriving Eq, and add stuff to print formulae nicely – see the book or the attached file

CNF: utilities and examples


```haskell
-- function to negate literals
neg :: Literal a -> Literal a
neg (P a) = N a
neg (N a) = P a

-- an example CNF formula
eg = And[ Or[N A, N C, P D], Or[P A, P C], Or[N D] ]

-- instead of full environments, a valuation is just
-- a list of the true literals
data Val a = Val [ Literal a ]
```

Evaluating formulae

First, let's look at evaluating a formula *given* a valuation.

```
eval (Val tls) (And cs) =
  and [ or [ l `elem` tls | l <- c ] | Or c <- cs ]
```

Evaluating formulae

First, let's look at evaluating a formula *given* a valuation.

```
eval (Val true_literals) (And clauses) =
  and [ or [ literal `elem` true_literals
           | literal <- clause ] | Or clause <- clauses ]

eval (Val []) eg
  -- False
eval (Val [N C, P A, N D]) eg
  -- True
```

where we had defined

```
eg = And[ Or[N A, N C, P D], Or[P A, P C], Or[N D] ]
```

This notion of valuation is a bit strange: neither `P A` nor `N A` is in `[]`, so is `A` true or false?

Finding satisfying assignments

In most applications we have a formula Φ and we want to find a valuation that makes it true – if there is one.

What is a simple way to do this?

Finding satisfying assignments

In most applications we have a formula Φ and we want to find a valuation that makes it true – if there is one.

What is a simple way to do this?

It is easy to list all possible valuations and check Φ under each one in turn.

Finding satisfying assignments

In most applications we have a formula Φ and we want to find a valuation that makes it true – if there is one.

What is a simple way to do this?

It is easy to list all possible valuations and check Φ under each one in turn.

If there are n atoms, how many possible valuations are there?

Finding satisfying assignments

In most applications we have a formula Φ and we want to find a valuation that makes it true – if there is one.

What is a simple way to do this?

It is easy to list all possible valuations and check Φ under each one in turn.

If there are n atoms, how many possible valuations are there?

The brute force way always looks at all 2^n valuations. This is ok for a few atoms, but becomes quickly unmanageable. Can we do better?

Finding satisfying assignments

In most applications we have a formula Φ and we want to find a valuation that makes it true – if there is one.

What is a simple way to do this?

It is easy to list all possible valuations and check Φ under each one in turn.

If there are n atoms, how many possible valuations are there?

The brute force way always looks at all 2^n valuations. This is ok for a few atoms, but becomes quickly unmanageable. Can we do better?

Nobody knows how to (or whether we even can) avoid 2^n in general. But there are algorithms which do much better most of the time.

If you can find a fast way of finding a satisfying assignment, or prove it impossible, you will win \$1M and eternal fame. This is $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

Valuations as assumptions

With a formula in CNF, such as

$$\Phi = (\neg A \vee \neg C \vee \neg D) \wedge (A \vee C) \wedge \neg D$$

we want a valuation that makes every clause true.

This is not quite the previous example. Check to see what's different . . .

Valuations as assumptions

With a formula in CNF, such as

$$\Phi = (\neg A \vee \neg C \vee \neg D) \wedge (A \vee C) \wedge \neg D$$

we want a valuation that makes every clause true. We can see this as looking for a Γ such that

$$\Gamma \vDash \neg A, \neg C, \neg D \qquad \Gamma \vDash A, C \qquad \Gamma \vDash \neg D$$

Valuations as assumptions

With a formula in CNF, such as

$$\Phi = (\neg A \vee \neg C \vee \neg D) \wedge (A \vee C) \wedge \neg D$$

we want a valuation that makes every clause true. We can see this as looking for a Γ such that

$$\Gamma \models \neg A, \neg C, \neg D \qquad \Gamma \models A, C \qquad \Gamma \models \neg D$$

Recall that assuming false lets us prove anything.

Γ must be consistent – not contain both A and $\neg A$!

Valuations as assumptions

With a formula in CNF, such as

$$\Phi = (\neg A \vee \neg C \vee \neg D) \wedge (A \vee C) \wedge \neg D$$

we want a valuation that makes every clause true. We can see this as looking for a Γ such that

$$\Gamma \vDash \neg A, \neg C, \neg D \qquad \Gamma \vDash A, C \qquad \Gamma \vDash \neg D$$

Recall that assuming false lets us prove anything.

Γ must be consistent – not contain both A and $\neg A$!

But Γ does not need to contain every atom, only the ones that are needed: e.g. $\vDash A, \neg A$. That's why our 'valuations' were not full environments.

DPLL: the basics

The Davis–Putnam–Logemann–Loveland algorithm is still, 60 years after its invention, the fastest general purpose satisfiability algorithm.

The basic idea is:

- look at one atom at a time
- set it to $\top$ and simplify, recursively seek a satisfying assignment
- if that failed, set it to $\perp$, recursively seek a satisfying assignment

DPLL: basics: example

$\overset{?}{\not\models} \neg A, \neg C, \neg D \qquad \overset{?}{\not\models} A, C \qquad \overset{?}{\not\models} \neg D$

DPLL: basics: example

$$\overset{?}{\not\models} \neg A, \neg C, \neg D \qquad \overset{?}{\not\models} A, C \qquad \overset{?}{\not\models} \neg D$$

Choose A, set to $\top$, and simplify:

$$A \overset{?}{\not\models} \cancel{\neg A}, \neg C, \neg D \qquad \cancel{A \models A, C} \qquad A \overset{?}{\not\models} \neg D$$

Note two simplifications: remove RHS literals that contradict, remove clauses that match.

DPLL: basics: example

$$\overset{?}{\not\models} \neg A, \neg C, \neg D \qquad \overset{?}{\not\models} A, C \qquad \overset{?}{\not\models} \neg D$$

Choose A, set to $\top$, and simplify:

$$A \overset{?}{\not\models} \cancel{\neg A}, \neg C, \neg D \qquad \cancel{A \models A, C} \qquad A \overset{?}{\not\models} \neg D$$

Choose C, set to $\top$, and simplify:

$$A, C \overset{?}{\not\models} \cancel{\neg C}, \neg D \qquad A, C \overset{?}{\not\models} \neg D$$

Note two simplifications: remove RHS literals that contradict, remove clauses that match.

DPLL: basics: example

$$\overset{?}{\not\models} \neg A, \neg C, \neg D \qquad \overset{?}{\not\models} A, C \qquad \overset{?}{\not\models} \neg D$$

Choose A, set to $\top$, and simplify:

$$A \overset{?}{\not\models} \cancel{\neg A}, \neg C, \neg D \qquad \cancel{A \models A, C} \qquad A \overset{?}{\not\models} \neg D$$

Choose C, set to $\top$, and simplify:

$$A, C \overset{?}{\not\models} \cancel{\neg C}, \neg D \qquad A, C \overset{?}{\not\models} \neg D$$

Choose D, set to $\top$, and simplify:

$$A, C, D \not\models \cancel{\neg D} \qquad A, C, D \not\models \cancel{\neg D}$$

Note two simplifications: remove RHS literals that contradict, remove clauses that match.

Simplified to **empty** clauses, i.e. $\bot$. One of these is enough to fail!

DPLL: basics: example

$$\stackrel{?}{\models} \neg A, \neg C, \neg D \qquad \stackrel{?}{\models} A, C \qquad \stackrel{?}{\models} \neg D$$

Choose A, set to $\top$, and simplify:

$$A \stackrel{?}{\models} \cancel{\neg A}, \neg C, \neg D \qquad \cancel{A \models A, C} \qquad A \stackrel{?}{\models} \neg D$$

Choose C, set to $\top$, and simplify:

$$A, C \stackrel{?}{\models} \cancel{\neg C}, \neg D \qquad A, C \stackrel{?}{\models} \neg D$$

Choose D, set to $\top$, and simplify:

$$A, C, D \not\models \cancel{\neg D} \qquad A, C, D \not\models \cancel{\neg D}$$

Failed, so set D to $\bot$ and simplify:

$$\cancel{A, C, \neg D \models \neg D} \qquad \cancel{A, C, \neg D \models \neg D}$$

Note two simplifications: remove RHS literals that contradict, remove clauses that match.

Simplified to **empty** clauses, i.e. $\bot$. One of these is enough to fail!

DPLL: basics: example

$$\overset{?}{\models} \neg A, \neg C, \neg D \qquad \overset{?}{\models} A, C \qquad \overset{?}{\models} \neg D$$

Choose A, set to $\top$, and simplify:

$$A \overset{?}{\models} \cancel{\neg A}, \neg C, \neg D \qquad \cancel{A \models A, C} \qquad A \overset{?}{\models} \neg D$$

Choose C, set to $\top$, and simplify:

$$A, C \overset{?}{\models} \cancel{\neg C}, \neg D \qquad A, C \overset{?}{\models} \neg D$$

Choose D, set to $\top$, and simplify:

$$A, C, D \not\models \cancel{\neg D} \qquad A, C, D \not\models \cancel{\neg D}$$

Failed, so set D to $\perp$ and simplify:

$$\cancel{A, C, \neg D \models \neg D} \qquad \cancel{A, C, \neg D \models \neg D}$$

Nothing left to satisfy, so $A, C, \neg D$ works.

Note two simplifications: remove RHS literals that contradict, remove clauses that match.

Simplified to **empty** clauses, i.e. $\perp$. One of these is enough to fail!

DPLL: optimizations

$$\overset{?}{\nvDash} \neg A, \neg C, \neg D \qquad \overset{?}{\nvDash} A, C \qquad \overset{?}{\nvDash} \neg D$$

There is an obviously more sensible atom than A to start with!

DPLL: optimizations

$$\overset{?}{\nvDash} \neg A, \neg C, \neg D \qquad \overset{?}{\nvDash} A, C \qquad \overset{?}{\nvDash} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

DPLL: optimizations

$$\overset{?}{\nvDash} \neg A, \neg C, \neg D \qquad \overset{?}{\nvDash} A, C \qquad \overset{?}{\nvDash} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

- $\neg D$ is only literal in last clause, so we *must* set D to $\bot$.

DPLL: optimizations

$$\overset{?}{\not\models} \neg A, \neg C, \neg D \qquad \overset{?}{\not\models} A, C \qquad \overset{?}{\not\models} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

- $\neg D$ is only literal in last clause, so we *must* set D to $\perp$.
- D is pure: some clause has $\neg D$, and no clause has D. So setting $D = \perp$ is everywhere good.

$$\stackrel{?}{\nvDash} \neg A, \neg C, \neg D \qquad \stackrel{?}{\nvDash} A, C \qquad \stackrel{?}{\nvDash} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

- $\neg D$ is only literal in last clause, so we *must* set D to $\perp$.
- D is pure: some clause has $\neg D$, and no clause has D. So setting $D = \perp$ is everywhere good.

Hence: choose D, set to $\perp$ and simplify:

$$\cancel{\neg D \models \neg A, \neg C, \neg D} \qquad \neg D \stackrel{?}{\nvDash} A, C \qquad \cancel{\neg D \models \neg D}$$

DPLL: optimizations

$$\overset{?}{\models} \neg A, \neg C, \neg D \qquad \overset{?}{\models} A, C \qquad \overset{?}{\models} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

- $\neg D$ is only literal in last clause, so we *must* set D to $\bot$.
- D is pure: some clause has $\neg D$, and no clause has D. So setting $D = \bot$ is everywhere good.

Hence: choose D, set to $\bot$ and simplify:

$$\cancel{\neg D \models \neg A, \neg C, \neg D} \qquad \neg D \overset{?}{\models} A, C \qquad \cancel{\neg D \models \neg D}$$

The remaining clause(s) are a *consistent set of literals*, so make them all true: set $A = \top$, $C = \top$. And we're done.

DPLL: optimizations

$$\stackrel{?}{\models} \neg A, \neg C, \neg D \qquad \stackrel{?}{\models} A, C \qquad \stackrel{?}{\models} \neg D$$

There is an obviously more sensible atom than A to start with!

D has *two* properties that make it good to start with:

- $\neg D$ is only literal in last clause, so we *must* set D to $\bot$.
- D is pure: some clause has $\neg D$, and no clause has D. So setting $D = \bot$ is everywhere good.

Hence: choose D, set to $\bot$ and simplify:

$$\cancel{\neg D \models \neg A, \neg C, \neg D} \qquad \neg D \stackrel{?}{\models} A, C \qquad \cancel{\neg D \models \neg D}$$

The remaining clause(s) are a *consistent set of literals*, so make them all true: set $A = \top$, $C = \top$. And we're done.

In addition, it's a good rule of thumb (heuristic) to start with literals from shorter clauses.

The DPLL algorithm

Given a set Φ of clauses:

DPLL(Φ)
  if literals of Φ are consistent **then**
    set atoms to make all literals true
  else if Φ has an empty clause **then**
    no satisfying assignment
  else
    make each one-literal clause true and simplify Φ to Φ'
    set each pure literal true and simplify Φ' to Φ''
    choose a remaining atom a
    if DPLL(set a true; simplify Φ'') succeeds **then**
      return result
    else
      DPLL(set a false; simplify Φ'')

The DPLL algorithm

Given a set Φ of clauses:

DPLL(Φ)
  if literals of Φ are consistent **then**
    set atoms to make all literals true
  else if Φ has an empty clause **then**
    no satisfying assignment
  else
    make each one-literal clause true and simplify Φ to Φ'
    set each pure literal true and simplify Φ' to Φ''
    choose a remaining atom a
    if DPLL(set a true; simplify Φ'') succeeds **then**
      return result
    else
      DPLL(set a false; simplify Φ'')

See the book for a Haskell implementation – or try to write your own first!

Toy application of CNF-SAT: Sudoku

Sudoku is a popular puzzle game.

- Given: a 9×9 grid, divided into nine 3×3 subgrids, with some cells containing digits from 1 to 9
- Goal: complete the grid so that each row, each column, and each subgrid contains all nine digits

		4	8	3			7	2
	1	2					8	
		5	2		1	3		
				6	2		9	1
7			5		9			3
9	4		7	8				
		3	9		7	4		
	5					6	1	
	8			4	6	9		

Toy application of CNF-SAT: Sudoku

Sudoku is a popular puzzle game.

- Given: a 9×9 grid, divided into nine 3×3 subgrids, with some cells containing digits from 1 to 9
- Goal: complete the grid so that each row, each column, and each subgrid contains all nine digits

		4	8	3			7	2
	1	2					8	
		5	2		1	3		
				6	2		9	1
7			5		9			3
9	4		7	8				
		3	9		7	4		
	5					6	1	
	8			4	6	9		

6	9	4	8	3	5	1	7	2
3	1	2	6	7	4	5	8	9
8	7	5	2	9	1	3	6	4
5	3	8	4	6	2	7	9	1
7	2	6	5	1	9	8	4	3
9	4	1	7	8	3	2	5	6
1	6	3	9	5	7	4	2	8
4	5	9	3	2	8	6	1	7
2	8	7	1	4	6	9	3	5

This puzzle was solved by the $\LaTeX$ package that printed it. The solver is 1000 lines of $\LaTeX$, and it isn't doing CNF-SAT.

Sudoku expressed in logic

How do we express 'cell (7,1) is filled with digit 4'?

Sudoku expressed in logic

How do we express 'cell (7,1) is filled with digit 4'?

We use one atom for every combination of row, column and digit!

$$F_{ijn} \text{ where } 1 \leq i, j, n \leq 9 \text{ means 'cell } (i, j) \text{ has } n\text{'}$$

For readability we'll write $F(i, j, n)$ instead of F_{ijn}.

Sudoku expressed in logic

How do we express 'cell (7,1) is filled with digit 4'?

We use one atom for every combination of row, column and digit!

$$F_{ijn} \text{ where } 1 \leq i, j, n \leq 9 \text{ means 'cell } (i, j) \text{ has } n\text{'}$$

For readability we'll write $F(i, j, n)$ instead of F_{ijn}.

We shall concoct CNF formulae for the *rules* of the solution, and for the *initial state*, and try to satisfy the conjunction of these.

The Sudoku formulae

All indices range over $1 \dots 9$ unless given otherwise.

The Sudoku formulae

All indices range over $1 \ldots 9$ unless given otherwise.

No cell is double-filled:

$$\bigwedge_{i,j,n,n' \neq n} \neg F(i,j,n) \vee \neg F(i,j,n')$$

The Sudoku formulae

All indices range over $1 \ldots 9$ unless given otherwise.

No cell is double-filled:

$$\bigwedge_{i,j,n,n' \neq n} \neg F(i,j,n) \vee \neg F(i,j,n')$$

Every row has each digit and every column has each digit:

$$\bigwedge_{i,n} \bigvee_{j} F(i,j,n) \qquad \bigwedge_{j,n} \bigvee_{i} F(i,j,n)$$

The Sudoku formulae

All indices range over $1 \ldots 9$ unless given otherwise.

No cell is double-filled:

$$\bigwedge_{i,j,n,n' \neq n} \neg F(i,j,n) \vee \neg F(i,j,n')$$

Every row has each digit and every column has each digit:

$$\bigwedge_{i,n} \bigvee_{j} F(i,j,n) \qquad \bigwedge_{j,n} \bigvee_{i} F(i,j,n)$$

Every subgrid has each digit:

$$\bigwedge_{0 \leq a \leq 2, 0 \leq b \leq 2, n} \quad \bigvee_{3a+1 \leq i \leq 3a+3, 3b+1 \leq j \leq 3b+3} F(i,j,n)$$

The Sudoku formulae

All indices range over $1 \ldots 9$ unless given otherwise.

No cell is double-filled:

$$\bigwedge_{i,j,n,n' \neq n} \neg F(i,j,n) \vee \neg F(i,j,n')$$

Every row has each digit and every column has each digit:

$$\bigwedge_{i,n} \bigvee_{j} F(i,j,n) \qquad \bigwedge_{j,n} \bigvee_{i} F(i,j,n)$$

Every subgrid has each digit:

$$\bigwedge_{0 \leq a \leq 2, 0 \leq b \leq 2, n} \quad \bigvee_{3a+1 \leq i \leq 3a+3, 3b+1 \leq j \leq 3b+3} F(i,j,n)$$

Other rules (used during solving): can't have same digit twice in row/column/subgrid.

The Sudoku formulae

All indices range over $1 \dots 9$ unless given otherwise.

No cell is double-filled:

$$\bigwedge_{i,j,n,n' \neq n} \neg F(i,j,n) \vee \neg F(i,j,n')$$

Every row has each digit and every column has each digit:

$$\bigwedge_{i,n} \bigvee_{j} F(i,j,n) \qquad \bigwedge_{j,n} \bigvee_{i} F(i,j,n)$$

Every subgrid has each digit:

$$\bigwedge_{0 \leq a \leq 2, 0 \leq b \leq 2, n} \quad \bigvee_{3a+1 \leq i \leq 3a+3, 3b+1 \leq j \leq 3b+3} \quad F(i,j,n)$$

Other rules (used during solving): can't have same digit twice in row/column/subgrid.

The formula for the starting position is easy: just conjoin all the $F(i,j,n)$ for each digit n in position (i,j).

For the details in Haskell, see the book and the tutorial exercises.