Full Clausal Logic - *Syntax*: clauses

compound terms aggregate objects

Add function symbols (functors), with an arity; constants are 0-ary functors.

object

proposition

```
  functor : single word starting with lower case
 variable : single word starting with upper case
     term : variable | functor[(term[,term]*)]
predicate : single word starting with lower case
     atom : predicate[(term[,term]*])]
   clause : head [:- body]
     head : [atom[;atom]*]
     body : proposition[,proposition]*
```

"adding two Peano-encoded naturals"

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Full Clausal Logic - *Semantics*: Herbrand universe, base, interpretation

analogous to relational clausal logic

Herbrand universe of a program P

```
{ 0, s(0), s(s(0)), s(s(s(0))),... }
```

infinite!

terms that can be constructed from the constants and functors

Herbrand base B_P of a program P

```
{ plus(0,0,0), plus(s(0),0,0),
  plus(0,s(0),0), plus(s(0),s(0),0),...}
```

set of all ground atoms that can be constructed using predicates in P and ground terms in the Herbrand universe of P

Herbrand interpretation I of P

is this a model?

```
{ plus(0,0,0), plus(s(0),0,s(0)),plus(0,s(0),s(0))} }
```

possibly infinite subset of B_P consisting of ground atoms that are true

Full Clausal Logic - *Semantics*: infinite models are possible

Herbrand universe is infinite, therefore infinite number of grounding substitutions

An interpretation is a **model for a program** if it is a model for each ground instance of every clause in the program.

```
plus(0,0,0)
plus(s(0),0,s(0)):-plus(0,0,0)
plus(s(s(0)),0,s(s(0))):-plus(s(0),0,s(0))
...
plus(0,s(0),s(0))
plus(s(0),s(0),s(s(0))):-plus(0,s(0),s(s(0)))
plus(s(s(0)),s(0),s(s(s(0)))):-plus(s(0),s(0),s(s(0)))
...
```

according to first ground clause, `plus(0,0,0)` has to be in any model but then the second clause requires the same of `plus(s(0),0,s(0))` and the third clause of `plus(s(s(0)),0,s(s(0)))` ...

all models of this program are necessarily infinite

Full Clausal Logic - *Proof Theory*: computing the most general unifier

analogous to relational clausal logic, but have to take compound terms into acount when computing the mgu of complementary atoms

atoms

`plus(s(0),X,s(X))` and `plus(s(Y),s(0),s(s(Y)))`

have most general unifier

`{Y/0, X/s(0)}`

yields unified atom `plus(s(Y),s(0),s(s(Y)))`

found by

renaming variables so that the two atoms have none in common
ensuring that the atoms' predicates and arity correspond
scanning the subterms from left to right to
- find first pair of subterms where the two atoms differ;
  - if neither subterm is a variable, unification fails;
  - else substitute the other term for all occurrences of the variable
  - and remember the partial substitution;
- repeat until no more differences found

`s(Y)` and `s(0)`

`{Y/0}`

Full Clausal Logic - *Proof Theory*: computing the most general unifier using the Martelli-Montanari algorithm

repeat
  select $s = t \in \mathcal{E}$

operates on a finite set of equations s=t

  case $s = t$ **of**
    $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \;\; (n \geq 0)$:
      replace $s = t$ **by** $\{s_1 = t_1, \ldots, s_n = t_n\}$
    $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n) \;\; (f/m \neq g/n)$:
      fail
    $X = X$:
      remove $X = X$ **from** $\mathcal{E}$
    $t = X \;\; (t \notin \mathbf{Var})$:
      replace $t = X$ **by** $X = t$
    $X = t \;\; (X \in \mathbf{Var} \wedge X \neq t \wedge X$ occurs more than once in $\mathcal{E})$:
      if Xoccurs in t
      then fail

occur check

      else replace all occurrences of X in $\mathcal{E}$ (except in $X = t$) by t
  esac
until no change

$$
\begin{array}{ll}
 & \{\underline{f(X, g(Y)) = f(g(Z), Z)}\} \\
\Rightarrow & \{X = g(Z), \underline{g(Y) = Z}\} \\
\Rightarrow & \{X = g(Z), \underline{Z = g(Y)}\} \\
\Rightarrow & \{X = g(g(Y)), Z = g(Y)\} \\
\Rightarrow & \{X/g(g(Y)), Z/g(Y)\}
\end{array}
$$

resulting set = mgu

$$
\begin{array}{ll}
 & \{\underline{f(X, g(X), b) = f(a, g(Z), Z)}\} \\
\Rightarrow & \{X = a, \underline{g(X) = g(Z)}, b = Z\} \\
\Rightarrow & \{\underline{X = a}, X = Z, b = Z\} \\
\Rightarrow & \{X = a, \underline{a = Z}, b = Z\} \\
\Rightarrow & \{X = a, \underline{Z = a}, b = Z\} \\
\Rightarrow & \{X = a, Z = a, \underline{b = a}\} \\
\Rightarrow & \textbf{fail}
\end{array}
$$

Full Clausal Logic - *Proof Theory*: importance of occur check

before substituting a term for a variable, verify that the variable does not occur in the term; if so: fail

program

```
loves(X,person_loved_by(X)).
```

query

```
:- loves(Y,Y).
```

without occur check, atoms to be resolved upon unify under substitution

```
{Y/X, X/person_loved_by(X)}
```

and therefore resolving to the empty clause

no semantics for infinite terms as there are no such terms in the Herbrand base

BUT

try to print answer:

```
X=person_loved_by(person_loved_by(person_loved_by(...)))
```

moreover, not a logical consequence of the program

omitting occur check renders resolution unsound

Full Clausal Logic - *Proof Theory:* occur check

not performed in Prolog out of performance considerations (e.g. unify X with a list of 1000 elements)

Martelli-Montanari algorithm

$$
\begin{array}{ll}
 & \{\underline{l(Y,Y) = l(X,f(X))}\} \\
\Rightarrow & \{\underline{Y = X}, Y = f(X)\} \\
\Rightarrow & \{Y = X, \underline{X = f(X)}\} \\
\Rightarrow & \textbf{fail}
\end{array}
$$

SWI-Prolog

```
?- l(Y,Y) = l(X,f(X)).
Y = f(**),
X = f(**).
?-
```

built-in unification operator

```
?- unify_with_occurs_check(l(Y,Y),l(X,f(X))).
false.
?-
```

in rare cases where the occurs check is needed

Full Clausal Logic - *Meta-theory*: soundness, completeness, decidability

sound

full clausal logic is sound

$P \vdash C \Rightarrow P \models C$

complete

full clausal logic is refutation-complete

$P \cup \{C\}$ inconsistent $\Rightarrow P \cup \{C\} \vdash \square$

decidability

The question "$P \models C$?" is only semi-decidable.

there is no algorithm that will always answer the question (with "yes" or "no") in finite time; but there is an algorithm that, if $P \models C$, will answer "yes" in finite time but this algorithm may loop if $P \not\models C$.

Clausal Logic: overview

	propositional	relational	full
Herbrand universe	-	`{a,b}` finite	`{a, f(a), f(f(a)), ...}` infinite
Herbrand base	`{p, q}`	`{p(a,a), p(b,a), ...}`	`{p(a, f(a)), p(f(a), p(f(f(a))), ...}`
clause	`p:-q`	`p(X,Z):- q(X,Y),p(Y,Z)`	`p(X, f(X)):- q(X)`
Herbrand models	`{}` `{p}` `{p,q}`	`{}` `{p(a,a)}` `{p(a,a),p(b,a),q(b,a)}` `...` finite number of finite models	`{}` `{p(a, f(a)),q(a)}` `{p(f(a), f(f(a)), q(f(a))} ...` infinite number of finite or infinite models
meta-theory	sound refutation-complete decidable	sound refutation-complete decidable	sound (occurs check) refutation-complete semi-decidable

Clausal Logic: conversion to first-order predicate logic (1)

Every set of clauses can be rewritten as an equivalent sentence in first-order predicate logic.

variables in a sentence cannot range over predicates

```
married;bachelor :- man,adult.
haswife :- married.
```

becomes

```
(man∧adult ⇒ married∨bachelor) ∧
(married ⇒ haswife)
```

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

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

or

```
(¬man ∨ ¬adult ∨ married ∨ bachelor )
∧ (¬married ∨ haswife)
```

conjunctive normal form: conjunction of disjunction of literals

```
reachable(X,Y,route(Z,R)):- connected(X,Z,L), reachable(Z,Y,R).
```

becomes

```
∀X∀Y∀Z∀R∀L : ¬connected(X,Z,L) ∨
             ¬reachable(Z,Y,R) ∨
             reachable(X,Y,route(Z,R))
```

variables in clauses are universally quantified

Clausal Logic: conversion to first-order predicate logic (2)

Every set of clauses can be rewritten as an equivalent sentence in first-order predicate logic.

```
nonempty(X) :- contains(X,Y).
```

becomes `∀X∀Y: nonempty(X)∨¬contains(X,Y)`

or `∀X: (nonempty(X)∨∀Y¬contains(X,Y))`

or `∀X: nonempty(X)∨¬(∃Y:contains(X,Y))`

or `∀X: (∃Y:contains(X,Y))⇒ nonempty(X))`

variables that occur only in the body of a clause are existentially qualified

Clausal Logic: conversion from first-order predicate logic (1)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X [brick(X)⇒(∃Y [on(X,Y)∧¬pyramid(Y)]∧
              ¬∃Y [on(X,Y)  ∧  on(Y,X)]∧
              ∀Y [¬brick(Y)⇒¬equal(X,Y)])]
```

1 eliminate ⇒ using $A \Rightarrow B \equiv \neg A \vee B$.

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)]∧
               ¬∃Y [on(X,Y)∧on(Y,X)]∧
               ∀Y [¬(¬brick(Y))∨¬equal(X,Y)])]
```

2 put into negation normal form: negation only occurs immediately before propositions

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)]∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)]∧
               ∀Y [brick(Y)∨¬equal(X,Y)])]
```

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg(\neg A) \equiv A$$
$$\neg \forall X\, [p(X)] \equiv \exists X\, [\neg p(X)]$$
$$\neg(\exists X\, [p(X)] \equiv \forall X\, [\neg p(X)]$$

Clausal Logic: conversion from first-order predicate logic (2)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)]∧
                ∀Y [¬on(X,Y)∨¬on(Y,X)]∧
                ∀Y [brick(Y)∨¬equal(X,Y)])]
```

model {loves(paul,anna)} can be converted to equivalent {loves(paul,person_loved_by(paul))}

$\exists X \forall Y$: loves(X,Y)

Skolem constants substitute for an existentially quantified variable which does not occur in the scope of a universal quantifier

$\forall X \exists Y$: loves(X,Y)

$\forall X$:loves(X,person_loved_by(X))

replace existentially quantified variable by a compound term of which the arguments are the universally quantified variables in whose scope the existentially quantified variable occurs

3 replace ∃ using Skolem functors (abstract names for objects, functor has to be new)

```
∀X [¬brick(X)∨([on(X,sup(X))∧¬pyramid(sup(X))]∧
                ∀Y [¬on(X,Y)∨¬on(Y,X)]∧
                ∀Y [brick(Y)∨¬equal(X,Y)])]
```

Clausal Logic: conversion from first-order predicate logic (3)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X [¬brick(X)∨([on(X,sup(X))∧¬pyramid(sup(X))]∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)]∧
               ∀Y [brick(Y)∨¬equal(X,Y)])]
```

4 standardize all variables apart such that each quantifier has its own unique variable

```
∀X [¬brick(X)∨([on(X,sup(X))∧¬pyramid(sup(X))]∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)]∧
               ∀Z [brick(Z)∨¬equal(X,Z)])]
```

5 move ∀ to the front

```
∀X∀Y∀Z [¬brick(X)∨([on(X,sup(X))∧¬pyramid(sup(X))]∧
                     [¬on(X,Y)∨¬on(Y,X)]∧
                     [brick(Z)∨¬equal(X,Z)])]
```

Clausal Logic: conversion from first-order predicate logic (4)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X∀Y∀Z[¬brick(X)∨([on(X,sup(X))∧¬pyramid(sup(X))]∧
                  [¬on(X,Y)∨¬on(Y,X)]∧
                  [brick(Z)∨¬equal(X,Z)])]
```

6 convert to conjunctive normal form using A∨(B∧C) ≡ (A∨B)∧(A∨C)

```
∀X∀Y∀Z[(¬brick(X)∨[on(X,sup(X))∧¬pyramid(sup(X))])∧
       (¬brick(X)∨[¬on(X,Y)∨¬on(Y,X)])∧
       (¬brick(X)∨[brick(Z)∨¬equal(X,Z)])]
```

```
∀X∀Y∀Z[((¬brick(X)∨on(X,sup(X)))∧(¬brick(X)∨¬pyramid(sup(X))))∧
       (¬brick(X)∨[¬on(X,Y)∨¬on(Y,X)])∧
       (¬brick(X)∨[brick(Z)∨¬equal(X,Z)])]
```

```
∀X∀Y∀Z[[¬brick(X)∨on(X,sup(X))]∧
       [¬brick(X)∨¬pyramid(sup(X))]∧
       [¬brick(X)∨¬on(X,Y)∨¬on(Y,X)]∧
       [¬brick(X)∨brick(Z)∨¬equal(X,Z)]]
```

A∨(B∨C) ≡ A∨B∨C

Clausal Logic: conversion from first-order predicate logic (5)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X∀Y∀Z[[¬brick(X)∨on(X,sup(X))]∧
       [¬brick(X)∨¬pyramid(sup(X))]∧
       [¬brick(X)∨¬on(X,Y)∨¬on(Y,X)]∧
       [¬brick(X)∨brick(Z)∨¬equal(X,Z)]]
```

7 split the conjuncts in clauses (a disjunction of literals)

```
∀X   ¬brick(X)∨on(X,sup(X))
∀X   ¬brick(X)∨¬pyramid(sup(X))
∀X∀Y ¬brick(X)∨¬on(X,Y)∨¬on(Y,X)
∀X∀Z ¬brick(X)∨brick(Z)∨¬equal(X,Z)
```

8 convert to clausal syntax (negative literals to body, positive ones to head)

```
on(X,sup(X)) :- brick(X).
:- brick(X), pyramid(sup(X)).
:- brick(X), on(X,Y), on(Y,X).
brick(X) :- brick(Z), equal(X,Z).
```

Clausal Logic: conversion from first-order predicate logic (6)

For each first order sentence, there exists an "almost equivalent" set of clauses.

`∀X: (∃Y:contains(X,Y))⇒ nonempty(X))`

1	eliminate ⇒	`∀X: ¬(∃Y:contains(X,Y))∨nonempty(X))`
2	put into negation normal form	`∀X: (∀Y:¬contains(X,Y))∨nonempty(X))`
3	replace ∃ using Skolem functors	
4	standardize variables	
5	move ∀ to the front	`∀X∀Y: ¬contains(X,Y)∨nonempty(X)`
6	convert to conjunctive normal form	
7	split the conjuncts in clauses	
8	convert to clausal syntax	`nonempty(X) :- contains(X,Y)`

Definite Clause Logic: motivation

how to use the clause depends on what you want to prove, but this indeterminacy is a source of inefficiency in refutation proofs

indefinite program

```
married(X);bachelor(X) :- man(X), adult(X).
man(peter). adult(peter). man(paul).
:-married(maria). :-bachelor(maria). :-bachelor(paul).
```

logical consequences that can be derived in two resolution steps

clause is used from right to left

```
married(X);bachelor(X):-man(X),adult(X)        man(peter)
married(peter);bachelor(peter):-adult(peter)   adult(peter)
married(peter);bachelor(peter)
```

indefinite conclusion

clause is used from left to right

```
married(X);bachelor(X):-man(X),adult(X)        :-married(maria)
bachelor(maria):-man(maria),adult(maria)       :-bachelor(maria)
:-man(maria),adult(maria)
```

both literals from head and body are resolved away

```
married(X);bachelor(X):-man(X),adult(X)        man(paul)
married(paul);bachelor(paul):-adult(paul)      :-bachelor(paul)
married(paul):-adult(paul)
```

Definite Clause Logic: syntax and proof procedure

for efficiency's sake

rules out indefinite conclusions

full clausal logic clauses are restricted: at most one atom in the head

$A :- B_1, \ldots, B_n$

fixes direction to use clauses

from right to left:
➠ procedural interpretation

"prove A by proving each of B_i"

Definite Clause Logic: recovering lost expressivity

semantics and proof theory for the not in a general clause will be discussed later; Prolog actually provides a special predicate not/1 which can only be understood procedurally

problem

can no longer express

```
married(X); bachelor(X) :- man(X), adult(X).
man(john). adult(john).
```

characteristic of indefinite clauses

which had two minimal models

```
{man(john),adult(john),married(john)}
{man(john),adult(john),bachelor(john)}
{man(john),adult(john),married(john),bachelor(john)}
```

general clauses

definite clause containing not

first model is minimal model of **general** clause

```
married(X) :- man(X), adult(X), not bachelor(X).
```

second model is minimal model of **general** clause

```
bachelor(X) :- man(X), adult(X), not married(X).
```

to prove that someone is a bachelor, prove that he is a man and an adult, and prove that he is not a bachelor