First-Order Logic and Logic Programming

The programming languages based on the Lambda calculus model of computation are convenient for describing functional relationships that hold between the objects of a problem. The focus is on computing functions. For this reason, the programs still preserve a strong procedural flavor: input data are transformed into output data by means of computations specific to the type of the data. Associative languages, such as CLIPS, are more permissive. There are no inputs and no outputs and data are uniformly represented in a symbolic way as facts of a factual base. The program infers all facts that can be derived from the current facts according to a fact-driven set of rules. However, facts have a very simple structure and cannot capture complex relationships between objects. Moreover, the inference procedure does not suit those applications where a specific goal has to be achieved regardless of other goals that could be potentially satisfied according to the current factual base.

Apart from these classes of languages, logic programming languages provide enhanced descriptive power of any kind of relationship between the problem objects and, moreover, are equipped with efficient inference procedures proved complete and sound. The focus of a logic program is not restricted to the computation of the correct values of some properties of given objects. The main focus is to derive new relationships between objects, starting from a set of given relationships. The theoretical base of such languages is the First-Order Logic (abbreviated FOL, and equivalently known as first-order predicate calculus).

The syntax and semantics of first-order logic

In a similar way to Lambda calculus, first-order logic can be seen as a language in which sentences are constructed according to strict syntactic rules and have a given interpretation varying with the application domain. In what follows we prefer the term *universe of discourse* (UoD) instead of solved problem. The reason relies on the generic descriptive power of predicate calculus. What is described is a generic set of relationships between generic objects that can be subject to different interpretations according to different applications. The UoD can be any problem, domain, and world the sentences of a FOL description may apply to. This is similar to the Markov algorithmic machine, where the algorithms are working with generic alphabets, constants and variables. The basic symbols used to build sentences are:

- Constants: stand for given (generic) objects from the UoD (similar to the constants in the Markov algorithmic machine). By convention, the constants are represented by identifiers starting with a small-case letter[1]; e.g., **`pussycat`** is a constant that can represent a particular cat or a special beloved person.

- Variables: stand for generic objects from the UoD. A variable is represented as an identifier starting with a capital letter.

- Functional and predicate symbols are simply names of relations between the objects of the UoD. A functional identifier, called simply a function, describes a functional relationship. A predicate corresponds to any kind of relation. Recall that an $\mathbf{n}$-ary relation over a set $\mathbf{S}^{\mathbf{n}} = \mathbf{S}\times\mathbf{S}\times\ldots\mathit{n\ times}\ldots\times\mathbf{S}$ is a subset of tuples $\mathbf{<x_1,x_2,\ldots,x_n>}$, $\mathbf{x_i}\in\mathbf{S}$ from $\mathbf{S}^{\mathbf{n}}$. Assume that **`friend`** is such a binary relation **`friend={<fred,pussycat>, <bibo,pussycat>, <teo,pussycat>}`**. Notice that the constants can have different interpretations, as the relation itself.

A functional relation is restricted: if a tuple **`<x,y>`** is in the relation then no other tuple **`<x,z>`** with **`z`** different than **`y`** can be member of the relation.

[1] The naming conventions follow Prolog

The relation **`friend`** is a functional relation and corresponds to a non-injective function. The distinction between functions and predicates is important. Functions designate generic computations that do not have an implementation, in the sense that a function is not associated with specific code able to perform the computation. It simply declares what can be the result of a computation. A predicate declares an unrestricted relationship that does not necessarily hide a computation.

By convention, predicate and function identifiers start with a lower case letter. Moreover, if the tuple $<x_1,x_2,\ldots,x_n>$ is an element of a relation **`r`** we write $r(x_1,x_2,\ldots,x_n)$. In the binary case, the reading of **`r(x,y)`** is "**`x`** is (in) the relation **`r`** with **`y`**", e.g. **`friend(fred,pussycat)`** can be read as *`fred is friend of pussycat`*.

- Logical connectives: ⇒ (implication), ⇔ (bi-implication or equivalence), ∧(conjunction) ∨(disjunction), ¬(negation). They have the usual meaning described by truth tables. The precedence of these connectors is (from highest to lowest): ¬, ∧, ∨, ⇒, ⇔.

- Quantifiers: ∀(the universal quantifier that reads *for all*), ∃(the existential quantifier that reads *exists*). As far as the precedence is concerned we assume that the quantifiers have lower precedence than the logical connectors.

Using the above symbols, *terms*, *atomic sentences* and *sentences* can be created according to the following syntax rules:

```
term ::= function(term,...,term) | constant | variable

atomic_sentence ::= predicate(term,...,term) | term = term

sentence ::= atomic_sentence | (sentence) | ¬sentence |
             sentence connective sentence |
             quantifier variable,...,variable. sentence
```

Sentences are considered as generic. They specify generic relationships and properties of generic objects that are subject to different interpretations for particular UoDs. Depending on interpretation and UoD these relationships and properties may be true or false.

The equality *`term`* **`=`** *`term`* is a way of declaring that the two terms stand for the same object. For example, **`pussycat = margot`** shows that **`pussycat`** is an alias name of a generic object identified as **`margot`** (a constant).

Terms refer to objects whereas atomic sentences describe facts. For example, **`father(fred)`** is a term that under an appropriate interpretation stands for a functional relationship. The atomic sentence **`friend(father(fred),pussycat)`** shows that the father of **`fred`** (which is an unnamed object) is in relation **`friend`** with an object identified as **`pussycat`**.

Sentences that use quantifiers have more descriptive power. For example, the sentence "everybody has at least a friend that is different than himself" can be encoded as:

`∀Y.∃X.(friend(X,Y) ∧ ¬(Y = X))`

The universal quantifier has the same effect as the binding symbol λ in an expression from the Lambda calculus. It generalizes the domain of a variable in the sense that any object can substitute the bound variable. On the contrary, the variable bound by an existential quantifier stands for a specific object, possibly not unique, which can be substituted for the variable. Variables cannot appear free within sentences. They must be bound either universally or existentially. The variables bound by a quantifier have the quantified sentence as their scope.

The quantifiers have useful properties, similar to the properties of conjunction and disjunction. Such properties are the laws of De Morgan. Noting a sentence with **P**, we have:

$\neg\forall X.P \equiv \exists X.\neg P$	is the transcription of $\neg(A \wedge B) \equiv \neg A \vee \neg B$
$\neg\exists X.P \equiv \forall X.\neg P$	is the transcription of $\neg(A \vee B) \equiv \neg A \wedge \neg B$
$\forall X.P \equiv \neg\exists X.\neg P$	is the transcription of $A \wedge B \equiv \neg(\neg A \vee \neg B)$
$\exists X.P \equiv \neg\forall X.\neg P$	is the transcription of $A \vee B \equiv \neg(\neg A \wedge \neg B)$

Validity, satisfiability and proof

Assume that, in a description we have the sentences:

```
1.1) ∀X.friend(fred,X)        -- fred is a friend of everybody
1.2) pet(pussycat)            -- pussycat is a pet
1.3) ∀Y.∃X.(pet(Y) ∧ friend(X,Y) ∧ ¬(Y = X))
                              -- any pet has a friend different than itself
```

What can we infer intuitively from the sentences above?

From (1.3) and (1.2) we derive that, in a particular case, **Y** can be bound to **pussycat**. The sentence becomes

```
1.4) ∃X.(pet(pussycat) ∧ friend(X,pussycat) ∧ ¬(pussycat = X))
```

From (1.1) and (1.4) we can bound **X** to **fred** and decide that the sentence (1.3) is true for this particular case. Since it is the only particular case in our description, we conclude that the sentence (1.3) is true in our description if the sentences (1.1) and (1.2) are true. This last commitment depends on the particular UoD to which we apply the description and on the interpretation of the constants and predicates from the description. Therefore, the sentence (1.3) is not valid (true in general). Instead, we say that it is satisfiable.

A sentence is *valid* if and only if it is true under all possible interpretations in all possible UoDs. In other words, the sentence is true regardless of any of its possible meanings. A sentence is *satisfiable* if and only if there is some interpretation in some UoD for which the sentence is true.

In the example above, all three sentences are satisfiable but not valid. A valid sentence such as **friend(fred,bibo)** $\vee$ $\neg$**friend(fred,bibo)** is called a *tautology*.

The intuitive derivation of the fact that **pussycat** has **fred** as a friend introduces additional important elements of the first-order logic (and of any logic system): the inference procedure and the proof procedure. Call **Descr** a set of sentences in FOL.

If a sentence **S** is true whenever the sentences of the description **Descr** are true we say that **S** is *entailed* by **Descr**, and we write **Descr** $\models$ **S**. A mechanic procedure **inf** able to derive sentences entailed by **Descr** is an *inference procedure*. We write **Descr** $\models_{inf}$ **S** to specify that **S** is an entailed sentence of **Descr** and that **S** is derived by **inf**. If the inference procedure **inf** produces only sentences entailed by **Descr** then it is *sound* or *truth preserving*. If **inf** is able to generate all sentences entailed by **Descr** then it is *complete*.

Given the sentences of a description **Descr**, we can prove of disprove a given sentence **S**. Inference procedures are at the base of the proof procedure. The proof procedure uses a given set of inference procedures (or inference rules) and a specific control strategy for conducting the inference. If the proof procedure is able to prove only sentences entailed by **Descr** it is sound. It is complete if it can prove all sentences entailed by **Descr**.

Gödel completeness theorem shows that in the first-order logic any sentence that is entailed by a set of sentences can be proved from that set, i.e. there exists a complete proof procedure for the first-order logic. A complete and sound inference procedure, which can be used in a complete proof procedure, has been found by Robinson in 1965, 35 years after Gödel. It is called *resolution*. Resolution and all other inference procedures for FOL are based on unification. Moreover, mechanical proof is eased if the sentences have a special format called *canonical* or *normal form*.

Unification

The process of unification of two formulae has been used in type synthesis. We recall it in the context of FOL. Two atomic sentences t_1 and t_2 unify if they have the same predicate and there exists a substitution $S=\{v_1/x_1,\ v_2/x_2,...,v_n/x_n\}$ for the variables $x_1,x_2,...,x_n$ within t_1 and t_2 such that after performing the substitution of x_i by v_i within both t_1 and t_2, t_1 becomes the same as t_2. For example, observe that

t_1= **friend(father(X),Y)** and t_2=**friend(Z,father(Z))**

unify under the substitution **S ={father(X)/Z, father(Z)/Y}**. This substitution is the most general unifier of t_1 and t_2. Under the substitution **S** the atomic sentences are identical

t_1**/S ≡** t_2**/S ≡ friend(father(X),father(father(X)))**.

Unification is proved NP-complete. Unification can produce cyclic results. In the unification process of **friend(X,father(X))** with **friend(father(Y),Y)** produces the substitution **S={father(Y)/X, father(X)/Y}** which means that **X** corresponds to **father(father(X))**. The unification algorithm must perform a so called *occurrence check* in order to avoid producing cyclic objects and, eventually, for avoiding the non termination of the unification itself. The occurrence check searches for occurrences of a variable within the value bound to that variable as result of the unification process. This process is time costly.

Normal forms

- A set of sentences is in conjunctive normal form (CNF) if each sentence is a disjunction of atoms. An atom is an atomic sentence eventually negated. All the sentences in the set are assumed to be implicitly connected by the conjunction connector (the CNF name derives from this convention).

- A set of sentences is in implicative normal form (INF) if each sentence has the structure $(P \wedge Q \wedge \ldots) \Rightarrow (P' \vee Q' \vee \ldots)$. Therefore, it is an implication with a conjunction of atoms on the left hand side and a disjunction of atoms on the right hand side.

There is a particular format of CNF sentences. This format is restrictive but plays a major role in logic programming.

- A sentence is a Horn sentence (or a Horn clause) if it is a disjunction of atoms with only one positive atom (non-negated atom). The sentence $\neg q_1 \vee \neg q_2 ... \vee \neg q_n \vee q$ is a Horn sentence and corresponds to the implication $(q_1 \wedge q_2 \wedge ... \wedge q_n) \Rightarrow q$.

In both CNF and INF normal forms variables appear non-bound. They are considered universally bound implicitly. As an exercise, consider the sentences below and their CNF and INF equivalent.

```
     ∀X,Y.(father(X,Y) ∨ mother(X,Y) ⇒ parent(X,Y))
     ∀X,Y,Z.(parent(Z,X) ∧ parent(Z,Y) ⇒ sibling(X,Y))
```

They can be rewritten as the following Horn clauses:

```
CNF: ¬father(X,Y) ∨ parent(X,Y)
     ¬mother(X,Y) ∨ parent(X,Y)
     ¬parent(Z,X) ∨ ¬parent(Z,Y) ∨ sibling(X,Y)

INF: father(X,Y) ⇒ parent(X,Y)
     mother(X,Y) ⇒ parent(X,Y)
     parent(Z,X) ∧ parent(Z,Y) ⇒ sibling(X,Y)
```

Any FOL sentence can be converted into an equivalent set of sentences in normal form. Here the term *equivalent* means *with the same meaning*. Consider that the sentence converted is part of a set of sentences called **Descr**. The rules for converting a sentence into a set of sentences in CNF are:

1. Eliminate implications. A sentence **p ⇒ q** is replaced by **¬p ∨ q**.

2. Move negation inwards using De Morgan rules.
   ¬(p ∨ q) is replaced by **¬p ∧ ¬q**
   ¬(p ∧ q) is replaced by **¬p ∨ ¬q**
   ¬∀X.P is replaced by **∃X.¬P**
   ¬ ∃X.P is replaced by **∀X.¬P**

3. Rename variables. For sentences that bind the same variable more than once, the name of the variable is changed to a unique name for each separate binding. For instance, **∀X.P(X) ∨ ∀X.Q(X) ∧ ∃X.R(X)** is transformed into **∀X.P(A) ∨ ∀X'.Q(X') ∧ ∃X".R(X")**.

4. Move quantifiers to the top of the sentence preserving the order in which they occur. Since the variables they bind are distinct, the sentence preserves its meaning. A sentence that has all the quantifiers at the top is in *prenex form*.

5. Remove the existential quantifiers.

- For an existential quantifier that is not preceded by any universal quantifier simply replace the occurrences of the bound variable by a unique constant (not used elsewhere in **Descr**).
- For an existential quantifier that is in the scope of several universal quantifiers, replace the occurrences of an existentially bound variable by a Skolem term $\mathbf{f(x_1,x_2,...,x_n)}$ where $\mathbf{x_1,x_2,...,x_n}$ are the variables bound by the universal quantifiers in the order they appear at the head of the sentence. The function **f** (not used elsewhere in **Descr**) designates a function which "computes" a unique value for the existentially bound variable depending on the values of $\mathbf{x_1,x_2,...,x_n}$.

For example, in the sentence **∀Y.∃X.(pet(Y) ∧ friend(X,Y))** the existential quantifier can be dropped by replacing all the occurrences of the variable **X** by the Skolem term **a_friend_of(Y)**. The sentence becomes

```
                    ∀Y.(pet(Y) ∧ friend(a_friend_of(Y),Y))
```

6. Drop the universal quantifiers and pretend that all variables are universally quantified implicitly.

$$\mathtt{pet(Y)} \wedge \mathtt{friend(a\_friend\_of(Y),Y)}$$

7. Distribute $\wedge$ over $\vee$: $(x \wedge y) \vee z$ is replaced by $(x \vee z) \wedge (y \vee z)$.

8. Flatten nested conjunctions and disjunctions. $(x \wedge y) \wedge z$ is replaced by $x \wedge y \wedge z$ and $(x \vee y) \vee z$ is replaced by $x \vee y \vee z$.

The sentence is in CNF. One more step is necessary for obtaining the INF format.

9. For each disjunction, group the negated atoms $\neg x_1 \vee \ldots \vee \neg x_n$ and the positive atoms $y_1 \vee \ldots \vee y_m$ of the disjunction and replace the disjunction by the implication $x_1 \wedge \ldots \wedge x_n \Rightarrow y_1 \vee \ldots \vee y_m$.

Inference rules

A sound inference rule (or procedure) derives, starting from a given set of sentences, a new sentence entailed by the sentences in the set. The simplest inference rule is *modus ponens*.

$$\frac{p_1 \wedge p_2 \wedge \ldots \wedge p_n \qquad (q_1 \wedge q_2 \wedge \ldots \wedge q_n) \Rightarrow q \qquad (\mathtt{unify}(p_i,q_i),\ i=1,n) = S}{\mathtt{Subst}(S,q)}$$

The rule reads: if the sentences p_i are true and the antecedents q_i of the implication are true under the unification substitution S (that unify p_i and q_i for all $i=1,n$) then the sentence $\mathtt{Subst}(S,q)$ is true. Modus ponens is simple but is not complete. The reason is simple: not all sentences can be written as Horn sentences. Therefore, the rule cannot be applied and the sentences entailed by non-Horn sentences cannot be inferred.

A complete inference procedure is *resolution*. The generalized resolution rule for INF is:

$$\frac{\begin{array}{ll} p_1 \wedge \ldots \wedge p_i \wedge \ldots \wedge p_{n1} \Rightarrow r_1 \vee \ldots \vee r_{n2} & \\ s_1 \wedge \ldots \wedge s_{n3} \Rightarrow q_1 \vee q_2 \vee \ldots \vee q_k \vee \ldots \vee q_{n4} & \mathtt{unify}(p_i,q_k) = S \end{array}}{\begin{array}{c} \mathtt{Subst}(S,\ p_1 \wedge \ldots \wedge p_{i-1} \wedge p_{i+1} \wedge \ldots \wedge p_{n1} \wedge s_1 \wedge \ldots \wedge s_{n3} \Rightarrow \\ r_1 \vee \ldots \vee r_{n2} \vee q_1 \vee \ldots \vee q_{k-1} \vee q_{k+1} \vee \ldots \vee q_n) \end{array}}$$

For the particular case of Horn clauses the resolution rule is:

$$\frac{\begin{array}{ll} (q_1 \wedge q_2 \wedge \ldots \wedge q_{n1}) \Rightarrow q & \\ (p_1 \wedge p_2 \wedge \ldots \wedge q' \wedge \ldots \wedge p_{n2}) \Rightarrow r & \mathtt{unify}(q,q') = S \end{array}}{\mathtt{Subst}(S,\ p_1 \wedge p_2 \wedge \ldots \wedge p_{n2} \wedge q_1 \wedge q_2 \wedge \ldots \wedge q_{n1} \Rightarrow r)}$$

This last rule is used in Prolog as the base of the proof procedure. Proof procedures are built using inference rules the application of which is decided by specific control strategies. The proof problem is: given a set of initial sentences, call it **Descr**[2], and a sentence **P**, find out if

[2] **Descr** is considered a FOL description of a UoD, which can be generic or particular.

Descr can entail **P**. The sentences in **Descr** can be classified into axioms and theorems. An *axiom* specifies a basic fact of the UoD and cannot be derived from the rest of the sentences in **Descr**. A *theorem* is a sentence entailed by the axioms and other theorems of **Descr**. By convention, an axiom **x** that corresponds to the implication **true ⇒ x**, will be represented simply as **x**. It is also useful to observe that ¬**x** ca be rewritten **x ⇒ false**.

An example of proof by refutation

Consider the following FOL description, which is a modified version of an example from the book of Russell and Norvig "Artificial Intelligence - A Modern Approach".

```
∀P,W,Q.(person(P)∧weapon(W)∧person(Q)∧hostile(Q)∧sells(P,W,Q)⇒ criminal(P))
∀M.(missile(M) ⇒ weapon(M))
∀W,P.(person(P) ∧ owns(P,W) ⇒ ∃Q.(person(Q) ∧ sells(Q,W,P)))
∀P.(person(P) ∧ ∀Q.(person(Q) ∧ hates(P,Q)) ⇒ hostile(P))

owns(foo,m1)
missile(m1)
person(foo)
∀P.hates(foo,P)
```

Applying the algorithm for converting FOL sentences to CNF and INF, the description above can be rewritten as the following set of Horn clauses.

```
person(P)∧weapon(W)∧person(Q)∧hostile(Q)∧sells(P,W,Q)⇒ criminal(P)
missile(M) ⇒ weapon(M)
person(P) ∧ owns(P,W) ⇒ person(seller(W,P))
person(P) ∧ owns(P,W) ⇒ sells(seller(W,P),W,P)
person(P) ∧ person(Q) ∧ hates(P,Q) ⇒ hostile(P)

owns(foo,m1)
missile(m1)
person(foo)
hates(foo,P)
```

Above, **seller(W,P)** is the Skolem term introduced by the elimination of the existential quantifier. The problem is to check if, according to the description, is there a criminal.

A complete proof procedure using resolution is by *refutation*. We consider that the goal **g** to be proved is not true and then try to obtain a contradiction using the sentences of the given description, i.e. **(Descr ∧ ¬g ⇒ false) ⇔ (Descr ⇒ g)**. Hence, assume there is no criminal, i.e. ¬**criminal(X)** or, equivalently, **criminal(X) ⇒ false** and add this clause to the description above.

The inference rule applied is the resolution for Horn clauses. The order of resolution steps follows a backward chaining path, from the goal to the axioms. In addition, the order of these steps is selected in such a way as to avoid backtracking and to shorten as much as possible the resolvents.

During the proof a current substitution called **S** is constantly updated. It cumulates the substitutions resulted from all the valid unifying processes performed so far along the current path traversed in the AND/OR proof tree. Recall that **v/X** means the variable **X** is bound to **v** and notice that **v** can be a variable. All substitutions are performed in situ, although this is not happening in the real proof process (it will be difficult to backtrack). For each use of a Horn clause new variables are created for that clause. The resolvent is written in italic.

negated goal: criminal(X) ⇒ false

criminal(X) ⇒ false
person(X)∧weapon(Y)∧person(Q)∧hostile(Q)∧sells(P,W,Q)⇒ criminal(P)
S={X/P}

---

person(X)∧weapon(W)∧person(Q)∧hostile(Q)∧sells(X,W,Q) ⇒ false
person(foo)
S= {foo/Q,X/P}

---

person(X)∧weapon(W)∧hostile(foo)∧sells(X,W,foo) ⇒ false
misile(M) ⇒ weapon(M)
S= {M/W,foo/Q,X/P}

---

person(X)∧missile(M)∧hostile(foo)∧sells(X,W,foo) ⇒ false
missile(m1)
S={m1/M,m1/W,foo/Q,X/P}

---

person(X)∧hostile(foo)∧sells(X,m1,foo) ⇒ false
person(P1)∧owns(P1,W1) ⇒ person(seller(W1,P1))
S={seller(W1,P1)/X,m1/M,m1/W,foo/Q,seller(W1,P1)/P}

---

person(P1)∧owns(P1,W1)∧hostile(foo)∧sells(seller(W1,P1),m1,foo) ⇒ false
owns(foo,m1)
S={foo/P1,m1/W1,seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

person(foo)∧hostile(foo)∧sells(seller(m1,foo),m1,foo) ⇒ false
person(foo)
S={foo/P1,m1/W1,seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

hostile(foo)∧sells(seller(m1,foo),m1,foo) ⇒ false
person(P2)∧person(Q2)∧hates(P2,Q2) ⇒ hostile(P2)
S={foo/P2,foo/P1,m1/W1,seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

person(Q2)∧hates(foo,Q2)∧sells(seller(m1,foo),m1,foo) ⇒ false
hates(foo,P3)
S={P3/Q2,foo/P2,foo/P1,m1/W1,seller(m1,foo)/X,m1/M,
m1/W,foo/Q,seller(m1,foo)/P}

---

person(P3)∧sells(seller(m1,foo),m1,foo) ⇒ false
person(foo)
S={foo/P3,foo/Q2,foo/P2,foo/P1,m1/W1,seller(m1,foo)/X,
m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

sells(seller(m1,foo),m1,foo) ⇒ false
person(P4)∧owns(P4,W4) ⇒ sells(seller(W4,P4),W4,P4)
S={foo/P4,m1/W4,foo/P3,foo/Q2,foo/P2,foo/P1,m1/W1,
seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

person(foo)∧owns(foo,m1) ⇒ false
person(foo)
S={foo/P4,m1/W4,foo/P3,foo/Q2,foo/P2,foo/P1,m1/W1,
seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

owns(foo,m1) ⇒ false
owns(foo,m1)
S={foo/P4,m1/W4,foo/P3,foo/Q2,foo/P2,foo/P1,m1/W1,
seller(m1,foo)/X,m1/M,m1/W,foo/Q,seller(m1,foo)/P}

---

true ⇒ false i.e. **false**
contradiction

Therefore, there is a criminal in the story. Who? Look at the value bound to the variable **`X`** of the goal **`criminal(X)`**. Its value is **`seller(m1,foo)`**. Although we don't know precisely who is this person, we have been able to prove that such a person exists.

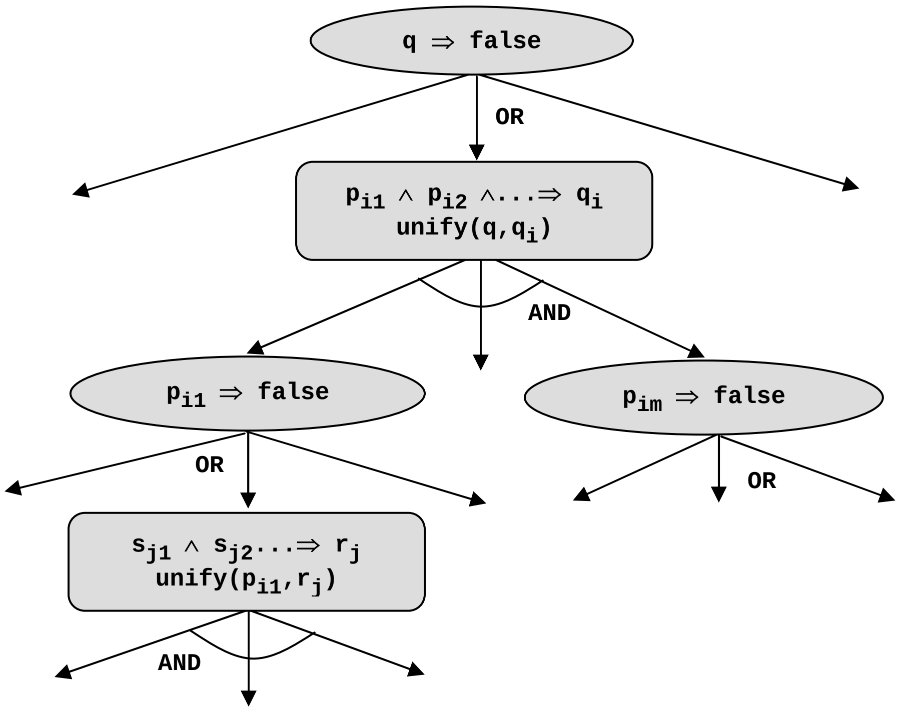

Figure 1. A proof AND-OR tree

In the presence of a control procedure the proof is no longer a linear structure as above. Some paths followed in the process of proof may fail. In this case backtracking occurs in order to try other possible resolvents. The structure of the proof process is an AND/OR tree. An AND node corresponds to a conjunction of resolvents. An OR node corresponds to the choice of the sentence to be used for producing a new resolvent.

To have a complete proof procedure the tree traversal should be breadth-first oriented. The depth-first traversal is making the proof procedure incomplete due to possible infinite paths that are created by circular sentences. It is interesting to notice that in the example above there are such infinite paths created by the circular clause:

$$\mathtt{person(P)} \wedge \mathtt{owns(P,W)} \Rightarrow \mathtt{person(seller(W,P))}$$

In a depth-first traversal of the proof tree we could enter an infinite loop trying to resolve the goal **`person(P)`**.

person(P) ⇒ false
person(P1) ∧ owns(P1,W1) ⇒ person(seller(W1,P1))
S={seller(W1,P1)/P}

---

person(P1) ∧ owns(P1,W1) ⇒ false
person(P2) ∧ owns(P2,W2) ⇒ person(seller(W2,P2))
S={seller(W2,P2)/P1,seller(W1,seller(W2,P2))/P}

---

person(P2) ∧ owns(P2,W2) ∧ owns(P1,W1) ⇒ false
. . .

Control strategies

There are two main control strategies: forward chaining and backward chaining. Forward chaining is similar to the **`Agenda`** control mechanism in CLIPS. It allows an inference procedure to derive all sentences that can be derived starting from **`Descr`**. It stops when the goal to be proved is eventually derived. Backward chaining focuses the application of the inference rule only on the relevant part of **`Descr`**, i.e. on these sentences that can effectively contribute to the derivation of the given goal.

The backward-chaining algorithm for Horn clauses resolution is given in Caml below. Except for the unification, it is close the executable kernel of a very simple theorem prover. The **`Backward_chaining`** function works with sentences that do not contain variables. As an exercise, the **`unify`** function could be rewritten to work with sentences containing variables.

The function **`Backward_chaining`** works with a list of rules, the rules from **`Descr`**. Each rule is represented as a list; the head of the list contains the conclusion of the rule, the tail contains the premises. The result returned is **`fail`** or **`succeed(substitution)`**, where **`substitution`** cumulates all the substitutions resulted from the unification process performed throughout the inference process. In a simple (but inefficient) implementation **`substitution`** can be an association list containing pairs **`(variable,value)`**.

The control engine is the local function **`solve`**. It gets a list of goals, a list of Horn clauses (called here rules), and the current substitution. If the list of goals is exhausted, i.e. all goals are proved, the result is **`suceed(final_substitution)`**. If the all rules are tried without succeeding to prove the goals, **`fail`** is signaled. For each goal, **`solve`** tries all rules the conclusion of which can unify with the goal (recall that the goal stands for a negated sentence while the conclusion is a positive sentence). For each applicable rule, the premises of the rule are appended at the end of the list of goals and **`solve`** is called recursively with the new goals, all the rules in **`Descr`**, and the new substitution. In this way, the proof tree is traversed breadth-first.

```
type 'subst Result = fail | succeed of 'subst list;;

let unify(atom1,atom2,substitution) =
    if atom1 = atom2 then succeed(substitution) else fail;;
unify : 'a * 'a * 'b list -> 'b Result = <fun>

let  conclusion = hd and  premises = tl;;
conclusion : 'a list -> 'a = <fun>
premises : 'a list -> 'a list = <fun>

let Backward_chaining all_rules goal =
    let rec solve =
        fun []    rules substitution -> succeed(substitution)
         |  goals []    substitution -> fail
         |  (goal::rest_goals as goals) (rule::rest_rules) substitution ->
                match unify(goal,conclusion(rule),substitution)
                with
                    fail -> solve goals rest_rules substitution
                  |  succeed(new_substitution) ->
                        match solve (rest_goals@(premises rule)) all_rules
                                    new_substitution
                        with
                            fail -> solve goals rest_rules substitution
                          | succeed(new_subst) -> succeed(new_subst)
    in
       solve [goal] all_rules [];;
Backward_chaining : 'a list list -> 'a -> 'b Result = <fun>
```

Observe that if **`Backward_chaining`** would be a coroutine, we could save the status of the proof tree while returning a solution. The traversal of the tree could be resumed from the interruption point producing step by step all the possible proofs of the given goal. This is what happens in Prolog.

The conclusion and the premises of each rule are atomic sentences represented as values of sum types. The representation is close to what a syntax checker would produce.

```
type Term =  constant of string
           | var of string
           | funct of string * Term list;;

type AtomicSentence = predicate of string * Term list;;

let Descr = [
   (* friend(bibo,pussycat) ∧friend(pussycat,fred) => friend(bibo,fred) *)
      [predicate("friend",[constant "Bibo"; constant "Fred"]);
            predicate("friend",[constant "Bibo"; constant "Pussycat"]);
            predicate("friend",[constant "Pussycat"; constant "Fred"])];

   (* friend(bibo,pussycat) *)
      [predicate("friend",[constant "Bibo"; constant "Pussycat"])];

   (* friend(pussycat,fred) *)
      [predicate("friend",[constant "Pussycat"; constant "Fred"])]
];;

let goal = predicate("friend",[constant "Bibo"; constant "Fred"]);;

Backward_chaining Descr goal;;
- : '_a Result = succeed []
```

The backward-chaining control strategy combined with the refutation-based resolution - as the proof procedure - offers an efficient proof engine for Horn clauses. There are logic programming languages (such as Prolog) that use this proof engine. What they add, apart from a convenient syntax, is a set of control mechanisms of the inference engine.

A program in a logic programming language consists of a set of axioms and inference rules that can be used to prove given goals. The axioms, rules, and goals are sentences that state properties and relationships of the objects of a given UoD. The aim of the program is merely to prove rather than compute. This is the reason why logic programming offers less on the side of arithmetic and more on the side of symbolic representation and on the way of combining formulae. A remarkable representative of the logic-programming paradigm is Prolog.