Computational Logics: Proof Procedures, Resolution and Unification

(Acknowledgement: Enrico Franconi)

In logic, clearly distinguish the definitions of

- the *formal language*
  syntax and semantics || expressive power
- the *reasoning problem*
  decidability || computational complexity
- the *problem solving procedure*
  soundness and completeness || (asymptotic) complexity

The Ideal Computational Logic

- expressive
- with decidable reasoning problems
- with sound and complete reasoning procedures
- with efficient reasoning procedures – possibly sub-optimal

⇒ **Description Logics**: explore the "most" interesting expressive decidable logics with "classical" semantics, equipped with "good" reasoning procedures.

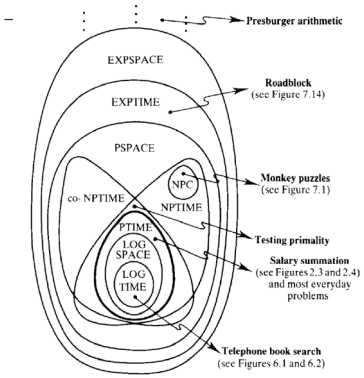
Computational Complexity

- The goal of complexity theory is to classify **problems** according to their intrinsic computational difficulty into *complexity classes*.
- Given a problem, how much computational power and/or resources (*time, space*) do we need in order to solve it **in the worst case**?
- The complexity class to which a problem belongs is a general property of the problem and not of a particular algorithm solving it.
- Distinguish among:
  - worst case
  - average case
  - hard and easy cases

Complexity Classes

- P (polynomial)
- NP — coNP (nondeterministic polynomial)
  Dual complexity classes such as coNP: class of problems whose complement (or dual version), in which the "yes" and "no" answers are interchanged, is in NP.
  It is not known whether NP = coNP, but it is known that if NP ≠ coNP, then also P ≠ NP.
- PSPACE
- EXPTIME
- NEXPTIME
- DECIDABLE

Complexity of Problems



Complexity of Problems

- Satisfiability of formulas in propositional logic is NP-complete.
- Unsatisfiability of formulas in propositional logic is coNP-complete.
- Satisfiability of formulas in FOL where there is a finite nesting of quantifiers is PSPACE-complete.
- Satisfiability of formulas in propositional *dynamic* normal modal logic is (PDL) is EXPTIME-complete.
- Satisfiability of $\mathcal{L}_2$ formulas is NEXPTIME-complete.
- Model checking of FOL formulas is polynomial (i.e., complexity depends on the reasoning problem).

Reasoning Procedures

A reasoning procedure is an algorithm trying to solve *specific instances* of a *specific reasoning problem* in a *given logic*.

- Whenever a **sound** reasoning procedure claims to have found a solution for a given instance of a problem, then this is actually a solution.
  - "no wrong inferences are drawn"
  - A sound procedure may fail to find a solution for some instances of the problem, when they actually have one.
- Whenever an instance of a problem has a solution, a **complete** reasoning procedure computes the solution for that instance.
  - "all the correct inferences are drawn"
  - A complete procedure may claim to have found a solution for some instances of the problem, when they do not have one.

Sound and Incomplete Algorithms

- are quite popular: They are considered *good* approximations of problem solving procedures.
- may reduce the algorithm complexity with respect to the computational worst case complexity.
- are often used due to the inability of programmers to find sound and complete algorithms.

Having **sound and complete** reasoning procedures is important!

Incompleteness and Completeness

Incompleteness

- Sequent calculus and natural deduction provide sound and complete procedures for computing logical implication.
- However, for FOL their termination is not guaranteed, because the problem is undecidable.
- It is easy to have incomplete but terminating procedures by simply dropping some of of the inference rules.
- This is a general method characterizing the incompleteness of algorithms:
  Find a complete set of inference rules, and characterize the incomplete procedure with a subset of them.

Completeness

- Given an incomplete reasoning procedure for a reasoning problem, it is sometimes preferred to modify the definition of the problem in order to obtain a complete procedure.
- This can be accomplished by slightly changing ("weakening") the semantics of the logical language.

Sub-optimal Algorithms

- Computational complexity considers only the **worst cases**.
- Sub-optimal (wrt. computational complexity) sound and complete algorithms can be faster for simple, average, and real instances of the problem, but less efficient for worst cases than optimal algorithms.
- Sub-optimal sound and complete algorithms can be compliant to software engineering requirements, i.e. they can be modular and expandable.
- Sub-optimal sound and complete algorithms may be optimal when considering sub-languages belonging to a lower complexity class.

The Resolution Principle: Informal Introduction

Forward inference: Derives all true consequences

Backward inference: Checks the truth of a proposition

Reasoning with the resolution principle: Proof by refutation of the negated proposition.

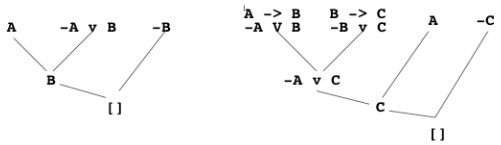
Basic idea:
Proof of B from the facts $A_1, A_2, \ldots$ and the rules $R_1, R_2, \ldots$,
by adding the negation of B to $A_1, A_2, \ldots, R_1, R_2, \ldots$
and deriving a contradiction.

Resolution: Resolving complementary literals

Examples:

1. $A, A \to B \vdash B$
by deriving the *empty clause* [] from the set of formulas
(clauses in conjunctive normal form)
$\{A, \neg A \vee B, \neg B\}$

2. $A \to B, B \to C \vdash A \to C$
in analogy by
$\{\neg A \vee B, \neg B \vee C, \neg(\neg A \vee C)\} = \{\neg A \vee B, \neg B \vee C, A, \neg C\}$



Theoretical Background of Resolution

Robinson (1965): *Automatic theorem proving*

Resolution: only inference rule

1. Algorithm to generate *skolemized prenex normal form.*
2. **Unification:** Test of trees for equality
   - Substitution: association of terms with variables
   - Unifier: substitution, which, when applied to two expressions, makes them equal
   - Most General Unifier: Robinson's algorithm
3. *Herbrand's theorem:* A theorem is unsatisfiable iff some finite subset of its Herbrand universe is inconsistent.

Remark: Herbrand Universe

Herbrand domain of A is the set of terms $\mathcal{H}$ obtained from A by:

1. If a is a constant in A then $a \in \mathcal{H}$; if A has no constants, an arbitrary constant symbol a is included in $\mathcal{H}$.
2. If $t_1, \ldots, t_n$ are terms in $\mathcal{H}$, and f is an n-ary function symbol in A, then $f(t_1, \ldots, t_n) \in \mathcal{H}$.

The **Herbrand universe** of A is the set of all substitution instances of A obtained by replacing each variable in A by a term from $\mathcal{H}$

Example:

x	y	$R(a, x, f(x), y)$
a	a	$R(a, a, f(a), a)$
a	$f(a)$	$R(a, a, f(a), f(a))$
$f(a)$	a	$R(a, f(a), f(f(a)), a)$
$f(a)$	$f(a)$	$R(a, f(a), f(f(a)), f(a))$
a	$f(f(a))$	$R(a, a, f(a), f(f(a)))$
$\vdots$	$\vdots$	$\vdots$

Generalized Modus Ponens

For prime formulas p_i, p_i', q, where a substitution θ exists, such that for all i SUBST$(\theta, p_i') =$ SUBST(θ, p_i):

$$\frac{p_1', p_2', \ldots, p_n', (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\mathrm{SUBST}(\theta, q)}$$

GMP is an efficient inference rule:

- makes bigger steps by combining several small inferences at once;
- makes relevant steps by using "productive" substitutions. The **unification algorithm** generates from two formulas a substitution which makes them equal, if it exists

- uses a precompilation step which converts all sentences (in the knowledge base, KB) into a **canonical form** – no further time cosuming conversions during the proof.

Canonical Form and Unification

All sentences (in the KB) should have a form which matches one of the premisses of the Modus Ponens rule.

Here: Either a prime formula or a subjunction with a conjunction of prime formulas on the lhs and a single prime formula on the rhs: **Horn formulas** ($\Rightarrow$ PROLOG)

Conversion steps: Elimination of existential quantifiers (Skolemization), elimination of conjunctions, dropping of universal quantifiers (next slide)

Remark: Not all formulas can be converted into Horn form,
e.g. $\bigwedge_x . \neg P(x) \rightarrow R(x)$. — cannot be used by MP.

Canonical Form: Conjunctive Normal Form

In the resolution rule each formula is a adjunction of literals, the whole KB a "big" conjunction of formulas: **"Conjunctive Normal Form" (CNF)**.

1. Eliminiate subjunctions
2. Push negations inside
3. Standardize (rename) variables
4. Pull quantifiers out (to the left)
5. Skolemize (introduce Skolem function terms for existential quantified var's)
6. Distribute $\wedge$ over $\vee$ (multiply out)
7. Flatten nested conjunctions and adjunctions.

(Further step for **Subjunctive NF**: turn adjunctions into subjunctions)

CNF Conversion Example

$\bigwedge_x(P(x) \rightarrow \bigvee_y((Q(x,y) \rightarrow P(x)) \wedge \bigwedge_z(Q(y,z) \rightarrow P(x))))$

1. $\bigwedge_x(\neg P(x) \vee \bigvee_y((\neg Q(x,y) \vee P(x)) \wedge \bigwedge_z(\neg Q(y,z) \vee P(x))))$
2. $\bigwedge_x(\neg P(x) \vee (\neg Q(x,f(x)) \vee P(x)) \wedge \bigwedge_z(\neg Q(f(x),z) \vee P(x)))$
3. $\bigwedge_x \bigwedge_z((\neg P(x) \vee \neg Q(x,f(x)) \vee P(x)) \wedge (\neg P(x) \vee \neg Q(f(x),z) \vee P(x)))$
4. $(\neg P(x) \vee \neg Q(x,f(x)) \vee P(x)) \wedge (\neg P(x) \vee \neg Q(f(x),z) \vee P(x))$
5. $\{\neg P(x) \vee \neg Q(x,f(x)) \vee P(x), \neg P(x) \vee \neg Q(f(x),z) \vee P(x)\}$

Unification

UNIFY$(p,q) = \theta$, such that SUBST(θ,p) = SUBST(θ,q)

θ is called **unifier**

Variable renaming to avoid name conflicts

Among the infinitely many unifiers, (if unifiable at all,) UNIFY shall produce the
most general unifier (MGU).

(Algorithms according to Russell/Norvig)

function UNIFY(x, y) **returns** a substitution to make x and y identical, if possible

UNIFY-INTERNAL(x, y, {})

---

function UNIFY-INTERNAL(x, y, θ) **returns** a substitution to make x and y identical (given θ)
inputs: x, a variable, constant, list, or compound
y, a variable, constant, list, or compound
θ, the substitution built up so far

if θ = failure **then return** failure
else if $x = y$ **then return** θ
else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) **and** COMPOUND?(y) **then**
return UNIFY-INTERNAL(ARGS[x], ARGS[y], UNIFY-INTERNAL(OP[x], OP[y], θ)
else if LIST?(x) **and** LIST?(y) **then**
return UNIFY-INTERNAL(REST[x], REST[y], UNIFY-INTERNAL(FIRST[x], FIRST[y], θ))
else return failure

---

function UNIFY-VAR(var, x, θ) **returns** a substitution
inputs: var, a variable
x, any expression
θ, the substitution built up so far

if {var/val} $\in \theta$
then return UNIFY-INTERNAL(val, x, θ)
else if {x/val} $\in \theta$
then return UNIFY-INTERNAL(var, val, θ)
else if var occurs anywhere in x /* *occur-check* */
then return failure
else return add {x/var} to θ

Forward Chaining Application of MP

Variable renaming
Composition of substitutions:
SUBST(COMPOSE(θ_1, θ_2), p) = SUBST(θ_2, SUBST(θ_1, p))

procedure FORWARD-CHAIN(KB, p)

if there is a sentence in KB that is a renaming of p **then return**
Add p to KB
for each $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$ **in** KB such that for some i, UNIFY(p_i, p) = θ succeeds **do**
FIND-AND-INFER(KB, $[p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n]$, q, θ)
end

---

procedure FIND-AND-INFER(KB, $premises$, $conclusion$, θ)

if $premises$ = [ ] **then**
FORWARD-CHAIN(KB, SUBST(θ, $conclusion$))
else for each p' **in** KB such that UNIFY(p', SUBST(θ, FIRST($premises$))) = θ_2 **do**
FIND-AND-INFER(KB, REST($premises$), $conclusion$, COMPOSE(θ, θ_2))
end

Backward Chaining Application of MP

function BACK-CHAIN(KB, q) **returns** a set of substitutions

BACK-CHAIN-LIST($KB, [q], \{\}$)

---

function BACK-CHAIN-LIST($KB, qlist, \theta$) **returns** a set of substitutions
inputs: KB, a knowledge base
$qlist$, a list of conjuncts forming a query (θ already applied)
θ, the current substitution
static: *answers*, a set of substitutions, initially empty

if $qlist$ is empty **then return** $\{\theta\}$
$q \leftarrow$ FIRST($qlist$)
for each q_i' **in** KB such that $\theta_i \leftarrow$ UNIFY(q, q_i') succeeds **do**
Add COMPOSE(θ, θ_i) to *answers*
end
for each sentence $(p_1 \wedge \ldots \wedge p_n \Rightarrow q_i')$ **in** KB such that $\theta_i \leftarrow$ UNIFY(q, q_i') succeeds **do**
answers $\leftarrow$ BACK-CHAIN-LIST(KB, SUBST$(\theta_i, [p_1 \ldots p_n])$, COMPOSE$(\theta, \theta_i)) \cup$ *answers*
end
return the union of BACK-CHAIN-LIST(KB, REST($qlist$), θ) for each $\theta \in$ *answers*

Resolution: A Complete Inference Procedure

The resolution rule for the propositional case (see above) can be viewed as

- Inferencing by distinction of cases, or
- Transitivity of subjunction.

MP allows to derive new prime formulas, not new subjunctions; therefore resolution is more powerful.
Resolution is a generalization of Modus Ponens.

Generalized resolution rule (adjunctions – in analogy for subjunctions):
For literals p_i and q_i with UNIFY$(p_j, \neg q_k) = \theta$

$$\frac{p_1 \vee \ldots p_j \ldots \vee p_m, , \quad q_1 \vee \ldots q_k \ldots \vee q_n}{\mathrm{SUBST}(\theta, (p_1 \vee \ldots p_{j-1} \vee p_{j+1} \ldots p_m \vee q_1 \ldots q_{k-1} \vee q_{k+1} \ldots \vee q_n))}$$

Example of a Resolution Proof

(1) Parents of parents are grandparents:
$\bigwedge_x \bigwedge_y \bigwedge_z (P(x, y) \wedge P(y, z) \rightarrow GP(x, z))$

(2) Everybody has a parent:
$\bigwedge_x \bigvee_y P(x, y)$

With (1) and (2) we want to prove

(3) Everybody has a grandparent:
$\bigwedge_x \bigvee_y GP(x, y)$

With the Skolem functions $f(x)$ and $g(x)$ we get the set of clauses
$\{\neg P(x_1, y_1) \vee \neg P(y_1, z) \vee GP(x_1, z), P(x_2, f(x_2)), \neg GP(x_3, g(x_3))\}$

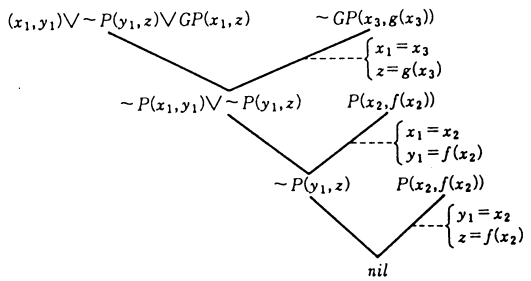


Figure 5.21.

The figure shows how the empty clause is derived.

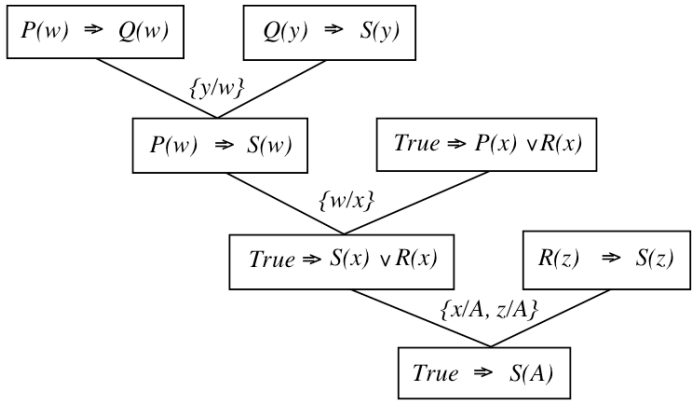
Substitution of the variables in this proof yields
$z = f(x_2) = f(x_1) = f(f(x_2)) = g(x_3) = g(x_1) = g(x_2)$,
therefore $g(x) = f(f(x))$, i.e. a parent of a parent is a grandparent.

Resolution Proofs: Further Examples

KNOWLEDGE BASE

Conjunctive NF	Subjunctive NF
$\neg P(w) \vee Q(w)$	$P(w) \rightarrow Q(w)$
$P(x) \vee R(x)$	True $\rightarrow P(x) \vee R(x)$
$\neg Q(y) \vee S(y)$	$Q(y) \rightarrow S(y)$
$\neg R(z) \vee S(z)$	$R(z) \rightarrow S(z)$

Combination by resolution: Proof of $S(A)$ from the KB
(last step: $True \Rightarrow S(A) \vee S(A)$ simplified)

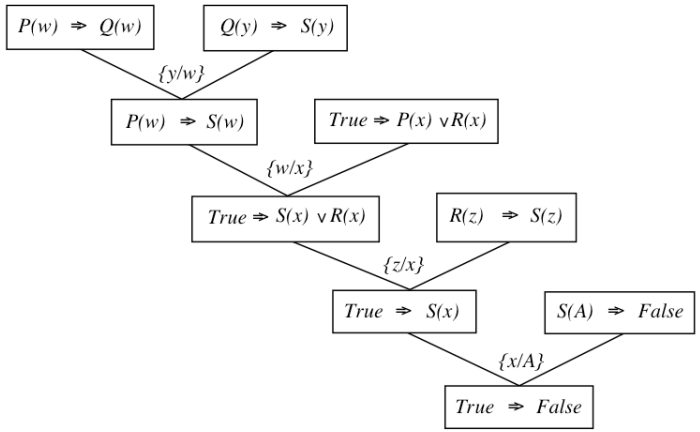


Resolution Proofs: Examples

Combination by resolution is more powerful than combination by MP, but not complete
. . . try to derive $P \vee \neg P$ from empty KB – what shall the resolution rule be applied to??

Refutation (proof by contradiction, reductio ad absurdum) is a **complete** inference procedure with resolution (for the proof of P the KB is added $\neg P$):
$(KB \wedge \neg P \Rightarrow$ False$) \Leftrightarrow (KB \Rightarrow P)$

In our example, to prove $S(A)$, we add $\neg S(A)$ (in subjunctive normal form: $S(A) \Rightarrow False$) to the KB, and apply resolution, until a contradiction is reached: $True \Rightarrow False$



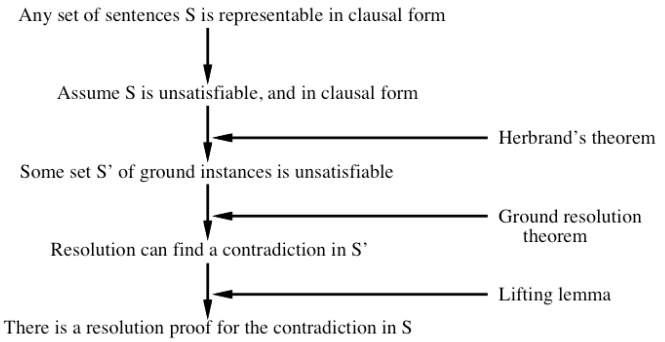
Resolution Strategies: Efficient Control

- Unit Resolution: One of the parent clauses must be unary (may contain only one literal) — otherwise no resolvents are generated.
- Input Resolution: Each resolvent must have at least one parent clause out of the inital set of clauses.
  Linear Resolution: Resolution steps between two resolvents, in which one is a "predecessor" of the other one, are admitted.
- Set of Support: Division of the clause set in clauses, which originate from the premisses, and in clauses, which originate from the negated theorem; a contradiction can result only with a participation of the negated theorem!

etc.

Augmentation: Equality — Demodulation, Paramodulation

Structure of the Completeness Proof for Resolution



Reminder: Logic Programming and Prolog

A *HORN clause* is a clause with at most one positive literal.

Remark: Each clause
$$\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_k$$
can be written as a subjunction:
$$(A_1 \wedge \ldots \wedge A_n) \rightarrow B_1 \vee \ldots \vee B_k$$

Horn clauses:
$$\neg A_1 \vee \ldots \vee \neg A_n \vee B$$
as:

$(A_1 \wedge \ldots \wedge A_n)$	$\rightarrow$	B
antecedent		consequent
clause body		clause head

HORN Clauses (2)

1. *Unit clauses:* no antecedent, consequent only
   B or $\rightarrow B$, resp.
   assert the truth of the consequent: **"facts"**.
2. *Non-unit clauses:* antecedent and consequent
   $A_1 \wedge \ldots \wedge A_n \rightarrow B$
   assert the truth of the consequent, if the antecedent is true: **"rules"**.
3. *Negative clauses:* antecedent only, no consequent
   $A_1 \wedge \ldots \wedge A_n \rightarrow$ or $\neg(A_1 \wedge \ldots \wedge A_n)$, resp.
   negate the truth of the antecedent: **"queries"** ("goals").
4. *Empty clause:* $[]$

(1) and (2): **Definite clauses**

Queries

ask for the conditions under which the antecedent is true according to:

- Let $\mathcal{P}$ a given set of facts and rules (a *"program"*) and a conjunction $A_1 \wedge \ldots \wedge A_n$ [1].
- To determine are values for the variables in [1], such that [1] is a consequence of $\mathcal{P}$, i.e. we seek a constructive proof of $(\bigvee_{x_0,\ldots,x_k})A_1 \wedge \ldots \wedge A_n$ [2] from $\mathcal{P}$.
- **Refutation proof:** Show that the conjunction of the clauses in $\mathcal{P}$ is inconsistent with the negation of [2] .

  Then one can infer that [2] follows from $\mathcal{P}$, because $\mathcal{P}$ as a set of definite clauses cannot be inconsistent.

Prolog Semantics

A Horn clause logic program has declarative and procedural semantics.

- The *declarative* semantics is given by $\mathcal{PC}$: Horn clauses are subjunctions. However, there are differences: closed world assumption, negation by failure.
- *Procedural* semantics: Each clause is interpreted as the definition of a procedure. The consequent is the name of the procedure (with variable list, i.e. its calling pattern), the antecedent is the procedure body.
  Procedure call "by unification":
  If a set of procedure calls is given, one must be selected and a procedure (clause) must be *sought* whose head (consequent) unifies with the selected procedure call.

Nondeterminism

in a twofold way:

1. The order in which the procedure calls in the body are executed can be chosen freely, in principle.
2. During the execution of a procedure call any procedure definition, whose head unifies with the call can be chosen.

The original goal and the selection of subgoals at each step in the computation (1) determine a search space which contains several possible computations; it contains all possible answers for the goal.
Each path from the goal to a [] represents a successful computation; different paths can result in different bindings for the variables of the (original) goal.

PROLOG's Resolution Strategy: SLD

SLD: *S*elected *L*iteral with *D*efinite clauses

- Binary resolution: exactly one literal from each parent is resolved upon
- Linear resolution: pursues deductions by generating descendants of one clause, the goal
- Input resolution: uses derived clauses only once, always resolving them with an axiom (= input clause)
- SLD resolution: binary, linear input resolution, where the clause to be refuted is negative, the axioms are definite clauses, and every resolution step involves a negative parent clause and a definite input clause (resolving upon a literal in the negative parent which is selected as a function of this parent)

If a definite clause theory is inconsistent with a negative clause C, there is always a binary, linear, input resolution of [] from C.

Refutation is linear and clause to be refuted is the goal
⇒ there is always a single answer substitution.

Extensions of PROLOG

- Non-logical extensions (⇒ Meta-programming)
  - `call(G)`
  - Cut (`!`) (for pruning "useless" branches from the search tree)
  - Negation-as-Failure (NAF):
    `\+ Goal :- call(Goal) -> fail; true.`
  - `setof(Term, Goal, S)`
  - `assert(T), retract(T)`
  - `var(X), atomic(X), Term =.. List`
    ⇒ programs are data, also in PROLOG
- New logics, e.g. modal operators
- Constraint Logic Programming: Combination with a constraint solver for systems of (in-) equations
- Combination with other programming styles (functional, object oriented) in new programming languages

Γ as a PROLOG Program

```
friend(john,susan).
friend(john,andrea).
loves(susan, andrea).
loves(andrea,bill).
female(susan).
not-female(susan).
not-female(X) :- female(X), !, fail.
not-female(X).
```

PROLOG reasons over the *unique minimal model* of the theory Γ, where `andrea` is not a female.

The **cut** predicate `!` cuts off a part of the search space (the mother or-node), returns *success* and hences keeps all instantiations for the goal in which the cut occurs.
`fail` enforces backtracking.

Querying Γ:

```
?- friend(john,X), female(X), loves(X,Y),
   not-female(Y).
X = susan, Y = andrea
yes

?- not-female(andrea).
yes

?- female(andrea).
no
```

Logic Programming with Frames: F-Logic

"F-Logic is a deductive, object-oriented database language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modelling capabilities supported by the object oriented data model."

⇒ "Higher-order syntax with first-order semantics"

- Object-oriented language constructs: frames
- Translation into Horn logic programs
- Evaluation with "well-founded semantics"

F-Logic Syntax

- Subclass: `C1::C2`
- Instance: `O:C`
- Multivalued qualified attributes: `C1[A=>>C2]`
- Single-valued qualified attributes: `C1[A=>C2]`
- Instance with attribute/value(s): `O[A->>V1,V2], O[A->V]`

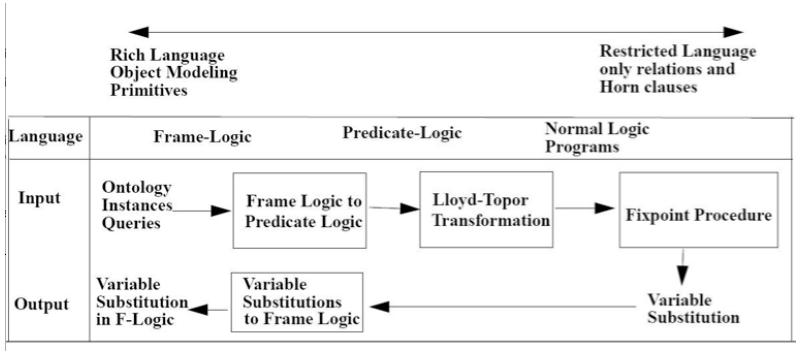
F-Logic: A First Example

```
/* facts */
abraham:man.
sarah:woman.
isaac:man[ father->abraham; mother->sarah] .
ishmael:man[ father->abraham; mother->hagar:woman] .
jacob:man[ father->isaac; mother->rebekah:woman] .
esau:man[ father->isaac; mother->rebekah] .

/* rules consisting of a rule head and a rule body */
FORALL X,Y X[ son->>Y] <- Y:man[ father->X] .
FORALL X,Y X[ son->>Y] <- Y:man[ mother->X] .
FORALL X,Y X[ daughter->>Y] <- Y:woman[ father->X] .
FORALL X,Y X[ daughter->>Y] <- Y:woman[ mother->X] .

/* query */
FORALL X,Y <- X:woman[ son->>Y[ father->abraham] ] .
```


F-Logic: Further Language Elements

- Rule names
- Name spaces
- Modules
- Built-ins, e.g. for
  - Lists and sets
  - Numbers and arithmetic
  - Strings
  - Data base systems integration

F-Logic: Execution



TRIPLE: A Query Language for the Semantic Web

Slides by Michael Sintek and Stefan Decker