Fundamentals of Artificial Intelligence and Knowledge Representation (Module 2)

Last update: 09 December 2023

Academic Year 2023 – 2024
Alma Mater Studiorum · University of Bologna

Contents

1 Propositional and first order logic

See `Languages and Algorithms for AI (module 2)`.

2 Prolog

It may be useful to first have a look at the ”Logic programming” section of `Languages and Algorithms for AI (module 2)`.

2.1 Syntax

Term Following the first-order logic definition, a term can be a:

Term

- Constant (`lowerCase`).
- Variable (`UpperCase`).
- Function symbol (`f(t1, ..., tn)` with `t1`, ..., `tn` terms).

Atomic formula An atomic formula has form:

Atomic formula

$$\texttt{p(t1, ..., tn)}$$

where `p` is a predicate symbol and `t1`, ..., `tn` are terms.

Note: there are no syntactic distinctions between constants, functions and predicates.

Clause A Prolog program is a set of horn clauses:

Horn clause

Fact `A.`

Rule `A :- B1, ..., Bn.` (`A` is the head and `B1, ..., Bn` the body)

Goal `:- B1, ..., Bn.`

where:

- `A`, `B1`, ..., `Bn` are atomic formulas.
- `,` represents the conjunction ($\land$).
- `:-` represents the logical implication ($\Leftarrow$).

Quantification

Quantification

Facts Variables appearing in a fact are quantified universally.

$$\texttt{A(X).} \equiv \forall \texttt{X} : \texttt{A(X)}$$

Rules Variables appearing the the body only are quantified existentially. Variables appearing in both the head and the body are quantified universally.

$$\texttt{A(X) :- B(X, Y).} \equiv \forall \texttt{X}, \exists \texttt{Y} : \texttt{A(X)} \Leftarrow \texttt{B(X, Y)}$$

Goals Variables are quantified existentially.

$$\texttt{:- B(Y).} \equiv \exists \texttt{Y} : \texttt{B(Y)}$$

2.2 Semantics

Execution of a program A computation in Prolog attempts to prove the goal. Given a program P and a goal `:- p(t1, ..., tn)`, the objective is to find a substitution σ such that:

$$P \models [\texttt{p(t1, ..., tn)}]\sigma$$

In practice, it uses two stacks:

Execution stack Contains the predicates the interpreter is trying to prove.

Backtracking stack Contains the choice points (clauses) the interpreter can try.

SLD resolution Prolog uses SLD resolution with the following choices:

SLD

Left-most Always proves the left-most literal first.

Depth-first Applies the predicates following the order of definition.

Note that the depth-first approach can be efficiently implemented (tail recursion) but the termination of a Prolog program on a provable goal is not guaranteed as it may loop depending on the ordering of the clauses.

Disjunction operator The operator `;` can be seen as a disjunction and makes the Prolog interpreter explore the remaining SLD tree looking for alternative solutions.

2.3 Arithmetic operators

In Prolog:

Arithmetic operators

- Integers and floating points are built-in atoms.
- Math operators are built-in function symbols.

Therefore, mathematical expressions are terms.

`is` predicate The predicate `is` is used to evaluate and unify expressions:

```
T is Expr
```

where `T` is a numerical atom or a variable and `Expr` is an expression without free variables. After evaluation, the result of `Expr` is unified with `T`.

Example.

```
?- X is 2+3.
    yes X=5
```

Note: a term representing an expression is evaluated only with the predicate `is` (otherwise it remains as is).

Relational operators Relational operators (`>`, `<`, `>=`, `=<`, `==`, `=/=`) are built-in.

2.4 Lists

A list is defined recursively as: Lists

Empty list `[]`

List constructor `.(T, L)` where `T` is a term and `L` is a list.

Note that a list always ends with an empty list.
As the formal definition is impractical, some syntactic sugar has been defined:

List definition `[t1, ..., tn]` can be used to define a list.

Head and tail `[H | T]` where `H` is the head (term) and `T` the tail (list) can be useful for recursive calls.

2.5 Cut

The cut operator (`!`) allows to control the exploration of the SLD tree. Cut
A cut in a clause:

```
p :- q1, ..., qi, !, qj, ..., qn.
```

makes the interpreter consider only the first choice points for `q1, ..., qi`, dropping all the other possibilities. Therefore, if `qj, ..., qn` fails, there won't be backtracking and `p` fails.

Example.

```
p(X) :- q(X), r(X).
q(1).
q(2).
r(2).

?- p(X).
    yes X=2
```

```
p(X) :- q(X), !, r(X).
q(1).
q(2).
r(2).

?- p(X).
    no
```

In the second case, the cut drops the choice point `q(2)` and only considers `q(1)`.

Mutual exclusion A cut can be useful to achieve mutual exclusion. In other words, to represent a conditional branching:

```
if a(X) then b else c
```

a cut can be used as follows:

```
p(X) :- a(X), !, b.
p(X) :- c.
```

If `a(X)` succeeds, other choice points for `p` will be dropped and only `b` will be evaluated. If `a(X)` fails, the second clause will be considered, therefore evaluating `c`.

2.6 Negation

Closed-world assumption Only what is stated in a program P is true, everything else is false:

Closed-world assumption

$$\texttt{CWA}(P) = P \cup \{\neg A \mid A \text{ is a ground atomic formula and } P \not\models A\}$$

Non-monotonic inference rule Adding new axioms to the program may change the set of valid theorems.

As first-order logic in undecidable, closed-world assumption cannot be directly applied in practice.

Negation as failure A negated atom $\neg A$ is considered true iff A fails in finite time:

Negation as failure

$$\texttt{NF}(P) = P \cup \{\neg A \mid A \in \texttt{FF}(P)\}$$

where $\texttt{FF}(P) = \{B \mid P \not\models B \text{ in finite time}\}$ is the set of atoms for which the proof fails in finite time. Note that not all atoms B such that $P \not\models B$ are in $\texttt{FF}(P)$.

SLDNF SLD resolution with NF to solve negative atoms.

SLDNF

Given a goal of literals `:- L`$_1$`, ..., L`$_m$, SLDNF does the following:

1. Select a positive or ground negative literal $\texttt{L}_i$:
   - If $\texttt{L}_i$ is positive, apply the normal SLD resolution.
   - If $\texttt{L}_i = \neg A$, prove that A fails in finite time. If it succeeds, $\texttt{L}_i$ fails.
2. Solve the goal `:- L`$_1$`, ..., L`$_{i-1}$`, L`$_{i+1}$`, ...L`$_m$.

Theorem 2.6.1. If only positive or ground negative literal are selected during resolution, SLDNF is correct and complete.

Prolog SLDNF Prolog uses an incorrect implementation of SLDNF where the selection rule always chooses the left-most literal. This potentially causes incorrect deductions.

Proof. When proving `:- \+capital(X).`, the intended meaning is:

$$\exists \texttt{X} : \neg\texttt{capital(X)}$$

In SLDNF, to prove `:- \+capital(X).`, the algorithm proves `:- capital(X).`, which results in:

$$\exists \texttt{X} : \texttt{capital(X)}$$

and then negates the result, which corresponds to:

$$\neg(\exists \texttt{X} : \texttt{capital(X)}) \iff \forall \texttt{X} : (\neg\texttt{capital(X)})$$

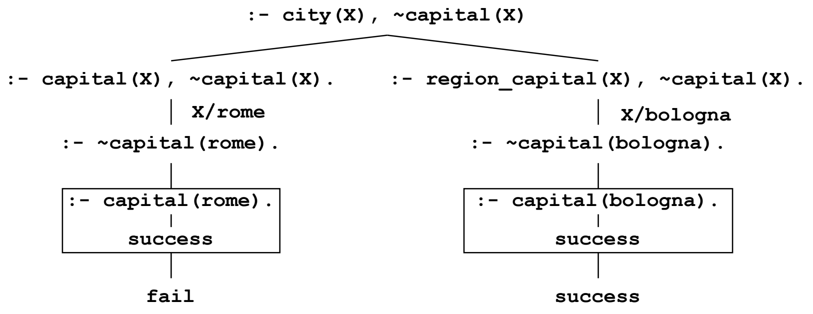
□

Example (Correct SLDNF resolution)**.** Given the program:

```
capital(rome).
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).

?- city(X), \+capital(X).
```

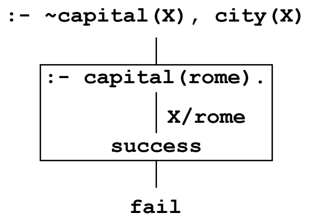
its resolution succeeds with `X=bologna` as `\+capital(X)` is ground by the unification of `city(X)`.



Example (Incorrect SLDNF resolution)**.** Given the program:

```
capital(rome).
region_capital(bologna).
city(X) :- capital(X).
city(X) :- region_capital(X).

?- \+capital(X), city(X).
```



its resolution fails as `\+capital(X)` is a free variable and the proof of `capital(X)` is ground with `X=rome` and succeeds, therefore failing `\+capital(X)`. Note that `bologna` is not tried as it does not appear in the axioms of `capital`.

2.7 Meta predicates

`call/1` Given a term `T`, `call(T)` considers `T` as a predicate and evaluates it. At the time of evaluation, `T` must be a non-numeric term. call/1

Example.

```
p(X) :- call(X).
q(a).

?- p(q(Y)).
    yes Y=a
```

`fail/0` The evaluation of `fail` always fails, forcing the interpreter to backtrack. fail/0

Example (Implementation of negation as failure)**.**

```
not(P) :- call(P), !, fail.
not(P).
```

Note that the cut followed by `fail` (`!, fail`) is useful to force a global failure.

`bagof/3` and `setof/3`

`bagof/3` The predicate `bagof(X, P, L)` unifies `L` with a list of the instances of `X` that satisfy `P`. Fails if none exists. bagof/3

`sefof/3` The predicate `setof(X, P, S)` unifies `S` with a set of the instances of `X` that satisfy `P`. Fails if none exists. sefof/3

In practice, for computational reasons, a list (with repetitions) might be computed.

Example.

```
p(1).
p(2).
p(1).

?- setof(X, p(X), S).
    yes S=[1, 2] X=X

?- bagof(X, p(X), S).
    yes S=[1, 2, 1] X=X
```

Quantification When solving a goal, the interpreter unifies free variables with a value. This may cause unwanted behaviors when using `bagof` or `setof`. The `X^` tells the interpreter to not (permanently) bind the variable `X`.

Example.

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, father(X, Y), S).
    yes X=X Y=giuseppe S=[giovanni];
        X=X Y=mario    S=[giovanni];
        X=X Y=paola    S=[mario]
```

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- setof(X, Y^father(X, Y), S).
    yes S=[giovanni, mario] X=X Y=Y
```

`findall/3` The predicate `findall(X, P, S)` unifies `S` with a list of the instances of `X` that satisfy `P`. If none exists, `S` is unified with an empty list. Variables in `P` that do not appear in `X` are not bound (same as the `Y^` operator). findall/3

Example.

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).

?- findall(X, father(X, Y), S).
    yes S=[giovanni, mario] X=X Y=Y
```

`var/1` The predicate `var(T)` is true if `T` is a variable. var/1

`nonvar/1` The predicate `nonvar(T)` is true if `T` is not a free variable. nonvar/1

`number/1` The predicate `number(T)` is true if `T` is a number. number/1

`ground/1` The predicate `ground(T)` is true if `T` does not have free variables. ground/1

`=../2` The operator `T =.. L` unifies `L` with a list where its head is the head of `T` and the tail contains the remaining arguments of `T` (i.e. puts all the components of a predicate into a list). Only one between `T` and `L` may be a variable. =../2

Example.

```
?- foo(hello, X) =.. List.
    List = [foo, hello, X]
```

```
?- Term =.. [baz, foo(1)].
    Term = baz(foo(1))
```

`clause/2` The predicate `clause(Head, Body)` is true if it can unify `Head` and `Body` with an existing clause. `Head` must be initialized to a non-numeric term. `Body` can be a variable or a term.

clause/2

Example.

```
p(1).
q(X, a) :- p(X), r(a).
q(2, Y) :- d(Y).

?- clause(p(1), B).
    yes B=true

?- clause(p(X), true).
    yes X=1

?- clause(q(X, Y), B).
    yes X=_1 Y=a B=p(_1), r(a);
        X=2 Y=_2 B=d(_2)
```

`assert/1` The predicate `assert(T)` adds `T` in an unspecified position of the clauses database of Prolog. In other words, it allows to dynamically add clauses.

assert/1

`asserta/1` As `assert(T)`, with insertion at the beginning of the database.

asserta/1

`assertz/1` As `assert(T)`, with insertion at the end of the database.

assertz/1

Note that `:- assert((p(X)))` quantifies `X` existentially as it is a query. If it is not ground and added to the database as is, is becomes a clause and therefore quantified universally: $\forall$`X` : `p(X)`.

Example (Lemma generation)**.**

```
fib(0, 0) :- !.
fib(1, 1) :- !.
fib(N, F) :- N1 is N-1, fib(N1, F1),
             N2 is N-2, fib(N2, F2),
             F is F1+F2,
             generate_lemma(fib(N, F)).

generate_lemma(T) :- clause(T, true), !.
generate_lemma(T) :- assert(T).
```

`generate_lemma/1` allows to add to the clauses database all the intermediate steps to compute the Fibonacci sequence (similar concept to dynamic programming).

`retract/1` The predicate `retract(T)` removes from the database the first clause that unifies with `T`.

retract/1

`abolish/2` The predicate `abolish(T, n)` removes from the database all the occurrences of `T` with arity `n`.

abolish/2

2.8 Meta-interpreters

Meta-interpreter Interpreter for a language L_1 written in another language L_2.

Meta-interpreter

Prolog vanilla meta-interpreter The Prolog vanilla meta-interpreter is defined as follows:

Vanilla meta-interpreter

```
solve(true) :- !.
solve( (A, B) ) :- !, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

In other words, the clauses state the following:

1. A tautology is a success.
2. To prove a conjunction, we have to prove both atoms.
3. To prove an atom `A`, we look for a clause `A :- B` that has `A` as conclusion and prove its premise `B`.

3 Ontologies

Ontology Formal (non-ambiguous) and explicit (obtainable through a finite sound procedure) description of a domain. Ontology

Category Can be organized hierarchically on different levels of generality. Category

Object Belongs to one or more categories. Object

Upper/general ontology Ontology focused on the most general domain. Upper/general ontology

Properties:

- Should be applicable to almost any special domain.
- Combining general concepts should not incur in inconsistences.

Approaches to create ontologies:

- Created by philosophers/logicians/researchers.
- Automatic knowledge extraction from well-structured databases.
- Created from text documents (e.g. web).
- Crowd-sharing information.

3.1 Categories

Category Used in human reasoning when the goal is category-driven (in contrast to specific-instance-driven). Category

In first order logic, categories can be represented through:

Predicate A predicate to tell if an object belongs to a category (e.g. `Car(c1)` indicates that `c1` is a car). Predicate categories

Reification Represent categories as objects as well (e.g. $\texttt{c1} \in \texttt{Car}$). Reification

3.1.1 Reification properties and operations

Membership Indicates if an object belongs to a category. (e.g. $\texttt{c1} \in \texttt{Car}$). Membership

Subclass Indicates if a category is a subcategory of another one. (e.g. $\texttt{Car} \subset \texttt{Vehicle}$). Subclass

Necessity Members of a category enjoy some properties (e.g. $(\texttt{x} \in \texttt{Car}) \Rightarrow \texttt{hasWheels(x)}$). Necessity

Sufficiency Sufficient conditions to be part of a category (e.g. $\texttt{hasPlate(x)} \land \texttt{hasWheels(x)} \Rightarrow \texttt{x} \in \texttt{Car}$). Sufficiency

Category-level properties Category themselves can enjoy properties (e.g. $\texttt{Car} \in \texttt{VehicleType}$) Category-level properties

Disjointness Given a set of categories S, the categories in S are disjoint iff they all have different objects:

Disjointness

$$\texttt{disjoint}(S) \iff (\forall c_1, c_2 \in S, c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset)$$

Exhaustive decomposition Given a category c and a set of categories S, S is an exhaustive decomposition of c iff any element in c belongs to at least a category in S:

Exhaustive decomposition

$$\texttt{exhaustiveDecomposition}(S,\ c) \iff (\forall o \in c \iff \exists c_2 \in S : o \in c_2)$$

Partition Given a category c and a set of categories S, S is a partition of c when:

Partition

$$\texttt{partition}(S,\ c) \iff \texttt{disjoint}(S) \land \texttt{exhaustiveDecomposition}(S,\ c)$$

3.1.2 Physical composition

Objects (meronyms) are part of a whole (holonym).

Part-of If the objects have a structural relation (e.g. `partOf(cylinder1, engine1)`).

Part-of

Properties:

Transitivity `partOf(x, y)` $\land$ `partOf(y, z)` $\Rightarrow$ `partOf(x, z)`

Reflexivity `partOf(x, x)`

Bunch-of If the objects do not have a structural relation. Useful to define a composition of countable objects (e.g. `bunchOf(nail1, nail3, nail4)`).

Bunch-of

3.1.3 Measures

A property of objects.

Quantitative measure Something that can be measured using a unit (e.g. `length(table1)` $=$ `cm(80)`).

Quantitative measure

Qualitative measures propagate when using `partOf` or `bunchOf` (e.g. the weight of a car is the sum of its parts).

Qualitative measure Something that can be measured using terms with a partial or total order relation (e.g. $\{\texttt{good}, \texttt{neutral}, \texttt{bad}\}$).

Qualitative measure

Qualitative measures do not propagate when using `partOf` or `bunchOf`.

Fuzzy logic Provides a semantics to qualitative measures (i.e. convert qualitative to quantitative).

Fuzzy logic

3.1.4 Things vs stuff

Intrinsic property Related to the substance of the object. It is retained when the object is divided (e.g. water boils at 100°C).

Intrinsic property

Extrinsic property Related to the structure of the object. It is not retained when the object is divided (e.g. the weight of an object changes when split).

Extrinsic property

Substance Category of objects with only intrinsic properties.

Substance

Stuff The most general substance category.

Stuff

Count noun Category of objects with only extrinsic properties.

Count noun

Things The most general object category.

Things

3.2 Semantic networks

Graphical representation of objects and categories connected through labelled links.

Semantic networks

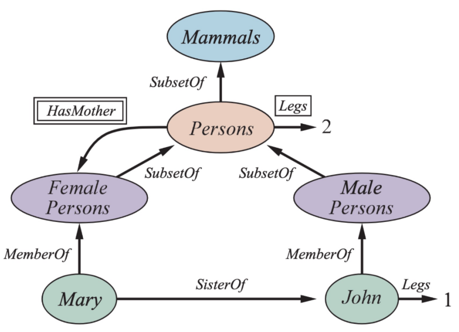


Figure 3.1: Example of semantic network

Objects and categories Represented using the same symbol.

Links Four different types of links:
- Relation between objects (e.g. `SisterOf`).
- Property of a category (e.g. 2 `Legs`).
- Is-a relation (e.g. `SubsetOf`).
- Property of the members of a category (e.g. `HasMother`).

Single inheritance reasoning Starting from an object, check if it has the queried property. If not, iteratively move up to the category it belongs to and check for the property.

Single inheritance reasoning

Multiple inheritance reasoning Reasoning is not possible as it is not clear which parent to choose.

Multiple inheritance reasoning

Limitations Compared to first order logic, semantic networks do not have:
- Negations.
- Universally and existentially quantified properties.
- Disjunctions.
- Nested function symbols.

Many semantic network systems allow to attach special procedures to handle special cases that the standard inference algorithm cannot handle. This approach is powerful but does not have a corresponding logical meaning.

Advantages With semantic networks it is easy to attach default properties to categories and override them on the objects (i.e. `Legs` of `John`).

3.3 Frames

Knowledge that describes an object in terms of its properties. Each frame has:

Frames

- An unique name
- Properties represented as pairs `<slot - filler>`

Example.

```
(
    toronto
        <:Instance-Of City>
        <:Province ontario>
        <:Population 4.5M>
)
```

Prototype Members of a category used as comparison metric to determine if another object belongs to the same class (i.e. an object belongs to a category if it is similar enough to the prototypes of that category). Prototype

Defeasible value Value that is allowed to be different when comparing an object to a prototype. Defeasible value

Facets Additional information contained in a slot for its filler (e.g. default value, type, domain). Facets

Procedural information Fillers can be a procedure that can be activated by specific facets:

`if-needed` Looks for the value of the slot.

`if-added` Adds a value.

`if-removed` Removes a value.

Example.

```
(
    toronto
        <:Instance-Of City>
        <:Province ontario>
        <:Population [if-needed QueryDB]>
)
```

4 Description logic

4.1 Syntax

Logical symbols Symbols with fixed meaning.

Logical symbols

Punctuation ( ) [ ]

Positive integers

Concept-forming operators `ALL`, `EXISTS`, `FILLS`, `AND`

Connectives $\sqsubseteq$, $\doteq$, $\rightarrow$

Non-logical symbols Domain-dependant symbols.

Non-logical symbols

Atomic concepts Categories (CamelCase, e.g. `Person`).

Roles Used to describe objects (:CamelCase, e.g. `:Height`).

Constants (camelCase, e.g. `johnDoe`).

Complex concept Concept-forming operators can be used to combine atomic concepts and form complex concepts. A well-formed concept follows the conditions:

Complex concept

- An atomic concept is a concept.
- If `r` is a role and `d` is a concept, then `[ALL r d]` is a concept.
- If `r` is a role and n is a positive integer, then `[EXISTS` n `r]` is a concept.
- If `r` is a role and `c` is a constant, then `[FILLS r c]` is a concept.
- If $\mathtt{d}_1 \dots \mathtt{d}_n$ are concepts, then `[AND` $\mathtt{d}_1 \dots \mathtt{d}_n$`]` is a concept.

Sentence Connectives can be used to combine concepts and form sentences. A well-formed sentence follows the conditions:

Sentence

- If $\mathtt{d}_1$ and $\mathtt{d}_2$ are concepts, then $(\mathtt{d}_1 \sqsubseteq \mathtt{d}_2)$ is a sentence.
- If $\mathtt{d}_1$ and $\mathtt{d}_2$ are concepts, then $(\mathtt{d}_1 \doteq \mathtt{d}_2)$ is a sentence.
- If `c` is a constant and `d` is a concept, then $(\mathtt{c} \rightarrow \mathtt{d})$ is a sentence.

Knowledge base Collection of sentences.

Knowledge base

Constants are individuals of the domain.

Concepts are categories of individuals.

Roles are binary relations between individuals.

Assetion box (A-box) List of facts about individuals.

Assetion box (A-box)

Terminological box (T-box) List of sentences (axioms) about concepts.

Terminological box (T-box)

4.2 Semantics

4.2.1 Concept-forming operators

Concept-forming operators

Let `r` be a role, `d` be a concept, `c` be a constant and n a positive integer. The semantics of concept-forming operators are:

`[ALL r d]` Individuals `r`-related to the individuals of the category `d`.

Example. `[ALL :HasChild Male]` individuals that have zero children or only male children.

`[EXISTS` n `r]` Individuals `r`-related to at least n other individuals.

Example. `[EXISTS 1 :Child]` individuals with at least one child.

`[FILLS r c]` Individuals `r`-related to the individual `c`.

Example. `[FILLS :Child john]` individuals with child `john`.

`[AND` $\texttt{d}_1 \dots \texttt{d}_n$`]` Individuals belonging to all the categories $\texttt{d}_1 \dots \texttt{d}_n$.

4.2.2 Sentences

Sentences

Sentences are expressions with truth values in the domain. Let `d` be a concept and `c` be a constant. The semantics of sentences are:

$\texttt{d}_1 \sqsubseteq \texttt{d}_2$ Concept $\texttt{d}_1$ is subsumed by $\texttt{d}_2$.

Example. $\texttt{PhDStudent} \sqsubseteq \texttt{Student}$ as every PhD is also a student.

$\texttt{d}_1 \doteq \texttt{d}_2$ Concept $\texttt{d}_1$ is equivalent to $\texttt{d}_2$.

Example. $\texttt{PhDStudent} \doteq$ `[AND Student :Graduated :HasFunding]`

$\texttt{c} \rightarrow \texttt{d}$ The individual `c` satisfies the description of the concept `d`.

Example. $\texttt{federico} \rightarrow \texttt{Professor}$

4.2.3 Interpretation

Interpretation

Interpretation An interpretation $\mathfrak{I}$ in description logic is a pair $(\mathcal{D}, \mathcal{I})$ where:

- $\mathcal{D}$ is the domain.
- $\mathcal{I}$ is the interpretation mapping.

  Constant Let `c` be a constant, $\mathcal{I}[\texttt{c}] \in \mathcal{D}$.

  Atomic concept Let `a` be an atomic concept, $\mathcal{I}[\texttt{a}] \subseteq \mathcal{D}$.

  Role Let `r` be a role, $\mathcal{I}[\texttt{r}] \subseteq \mathcal{D} \times \mathcal{D}$.

  `Thing` The concept `Thing` corresponds to the domain: $\mathcal{I}[\texttt{Thing}] = \mathcal{D}$.

  `[ALL r d]`

$$\mathcal{I}[\texttt{[ALL r d]}] = \{\texttt{x} \in \mathcal{D} \mid \forall \texttt{y} : \langle \texttt{x}, \texttt{y} \rangle \in \mathcal{I}[r] \text{ then } \texttt{y} \in \mathcal{I}[d]\}$$

  `[EXISTS` n `r]`

$$\mathcal{I}[\texttt{[EXISTS } n \texttt{ r]}] = \{\texttt{x} \in \mathcal{D} \mid \text{ exists at least } n \text{ distinct } \texttt{y} : \langle \texttt{x}, \texttt{y} \rangle \in \mathcal{I}[r]\}$$

[FILLS r c]

$$\mathcal{I}[\texttt{[FILLS r c]}] = \{\texttt{x} \in \mathcal{D} \mid \langle \texttt{x}, \mathcal{I}[\texttt{c}] \rangle \in \mathcal{I}[\texttt{r}]\}$$

[AND $\texttt{d}_1 \dots \texttt{d}_n$]

$$\mathcal{I}[\texttt{[AND } \texttt{d}_1 \dots \texttt{d}_n\texttt{]}] = \mathcal{I}[\texttt{d}_1] \cap \dots \cap \mathcal{I}[\texttt{d}_n]$$

Model Given an interpretation $\mathfrak{I} = (\mathcal{D}, \mathcal{I})$, a sentence is true under $\mathfrak{I}$ ($\mathfrak{I} \models$ sentence) if: Model

- $\mathfrak{I} \models (\texttt{c} \rightarrow \texttt{d})$ iff $\mathcal{I}[\texttt{c}] \in \mathcal{I}[\texttt{d}]$.
- $\mathfrak{I} \models (\texttt{d}_1 \sqsubseteq \texttt{d}_2)$ iff $\mathcal{I}[\texttt{d}_1] \subseteq \mathcal{I}[\texttt{d}_2]$.
- $\mathfrak{I} \models (\texttt{d}_1 \doteq \texttt{d}_2)$ iff $\mathcal{I}[\texttt{d}_1] = \mathcal{I}[\texttt{d}_2]$.

Given a set of sentences S, $\mathfrak{I}$ models S if $\mathfrak{I} \models S$.

Entailment A set of sentences S logically entails a sentence α if: Entailment

$$\forall \mathfrak{I} : (\mathfrak{I} \models S) \rightarrow (\mathfrak{I} \models \alpha)$$

4.3 Reasoning

4.3.1 T-box reasoning

Given a knowledge base of a set of sentences S, we would like to be able to determine the following:

Satisfiability A concept $\texttt{d}$ is satisfiable w.r.t. S if: Satisfiability

$$\exists \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}] \neq \varnothing$$

Subsumption A concept $\texttt{d}_1$ is subsumed by $\texttt{d}_2$ w.r.t. S if: Subsumption

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] \subseteq \mathfrak{I}[\texttt{d}_2]$$

Equivalence A concept $\texttt{d}_1$ is equivalent to $\texttt{d}_2$ w.r.t. S if: Equivalence

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] = \mathfrak{I}[\texttt{d}_2]$$

Disjointness A concept $\texttt{d}_1$ is disjoint to $\texttt{d}_2$ w.r.t. S if: Disjointness

$$\forall \mathfrak{I}, (\mathfrak{I} \models S) : \mathfrak{I}[\texttt{d}_1] \neq \mathfrak{I}[\texttt{d}_2]$$

Theorem 4.3.1 (Reduction to subsumption)**.** Given the concepts $\texttt{d}_1$ and $\texttt{d}_2$, it holds that: Reduction to subsumption

- $\texttt{d}_1$ is unsatisfiable $\iff \texttt{d}_1 \sqsubseteq \bot$.
- $\texttt{d}_1 \doteq \texttt{d}_2 \iff \texttt{d}_1 \sqsubseteq \texttt{d}_2 \wedge \texttt{d}_2 \sqsubseteq \texttt{d}_1$.
- $\texttt{d}_1$ and $\texttt{d}_2$ are disjoint $\iff (\texttt{d}_1 \cap \texttt{d}_2) \sqsubseteq \bot$.

4.3.2 A-box reasoning

Given a constant $\texttt{c}$, a concept $\texttt{d}$ and a set of sentences S, we can determine the following:

Satisfiability A constant $\texttt{c}$ satisfies the concept $\texttt{d}$ if: Satisfiability

$$S \models (\texttt{c} \rightarrow \texttt{d})$$

Note that it can be reduced to subsumption.

4.3.3 Computing subsumptions

Given a knowledge base KB and two concepts `d` and `e`, we want to prove:

$$KB \models (\texttt{d} \sqsubseteq \texttt{e})$$

The following algorithms can be employed:

Structural matching Structural matching

1. Normalize `d` and `e` into a conjunctive form:

   $$\texttt{d} = \texttt{[AND d}_1 \texttt{ ...d}_n\texttt{]} \qquad \texttt{e} = \texttt{[AND e}_1 \texttt{ ...e}_m\texttt{]}$$

2. Check if each part of `e` is accounted by at least a component of `d`.

Tableaux-based algorithms Exploit the following theorem: Tableaux-based algorithms

$$(KB \models (C \sqsubseteq D)) \iff (KB \cup (x : C \sqcap \neg D)) \text{ is inconsistent}$$

Note: similar to refutation.

4.3.4 Open world assumption

Open world assumption If a sentence cannot be inferred, its truth values is unknown. Open world assumption

Description logics are based on the open world assumption. To reason in open world assumption, all the possible models are split upon encountering an unknown facts depending on the possible cases (Oedipus example).

4.4 Expanding description logic

It is possible to expand a description logic by:

Adding concept-forming operators Let `r` be a role, `d` be a concept, `c` be a constant and n a positive integer. We can extend our description logic with: Adding concept-forming operators

`[AT-MOST` n `r]` Individuals `r`-related to at most n other individuals.

Example. `[AT-MOST 1 :Child]` individuals with only a child.

`[ONE-OF c`$_1$ `... c`$_n$`]` Concept only satisfied by `c`$_1$ `... c`$_n$.

Example. `Beatles` $\doteq$ `[ALL :BandMember [ONE-OF john paul george ringo]]`

`[EXISTS` n `r d]` Individuals `r`-related to at least n individuals in the category `d`.

Example. `[EXISTS 2 :Child Male]` individuals with at least two male children.

Note: this increases the computational complexity of entailment.

Relating roles Relating roles

`[SAME-AS r`$_1$ `r`$_2$`]` Equates fillers of the roles r_1 and r_2

Example. `[SAME-AS :CEO :Owner]`

Note: this increases the computational complexity of entailment. Role chaining also leads to undecidability.

Adding rules Rules are useful to add conditions (e.g. `if d`$_1$ `then [FILLS r c]`). Adding rules

4.5 Description logics family

Depending on the number of operators, a description logic can be:

- More expressive.
- Computationally more expensive.
- Undecidable.

Attributive language ($\mathcal{AL}$) Minimal description logic with:

- Atomic concepts.
- Universal concept (`Thing` or $\top$).
- Bottom concept (`Nothing` or $\bot$).
- Atomic negation (only for atomic concepts).
- `AND` operator ($\sqcap$).
- `ALL` operator ($\forall$).
- `[EXISTS 1 r]` operator ($\exists$).

Attributive language complement ($\mathcal{ALC}$) $\mathcal{AL}$ with negation for concepts.

$\mathcal{F}$	Functional properties
$\mathcal{E}$	Full existential quantification
$\mathcal{U}$	Concept union
$\mathcal{C}$	Complex concept negation
$\mathcal{S}$	$\mathcal{ALC}$ with transitive roles
$\mathcal{H}$	Role hierarchy
$\mathcal{R}$	Limited complex roles axioms Reflexivity and irreflexivity Roles disjointness
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse properties
$\mathcal{N}$	Cardinality restrictions
$\mathcal{Q}$	Qualified cardinality restrictions
$(\mathcal{D})$	Datatype properties, data values and data types

Table 4.1: Name and expressivity of logics

5 Web reasoning

5.1 Semantic web

Semantic web Method to represent and reason on the data available on the web. Semantic web aims to preserve the characteristics of the web, this includes: Semantic web

- Globality.
- Information distribution.
- Information inconsistency of contents and links (as everyone can publish).
- Information incompleteness of contents and links.

Information is structured using ontologies and logic is used as inference mechanism. New knowledge can be derived through proofs.

Uniform resource identifier Naming system to uniquely identify concepts. Each URI correspond to one and only one concept, but multiple URIs can refer to the same concept. URI

XML Markup language to represent hierarchically structured data. An XML can contain in its preamble the description of the grammar used within the document. XML

Resource description framework (RDF) XML-based language to represent knowledge. Based on triplets: Resource description framework (RDF)

```
<subject, predicate, object>
<resource, attribute, value>
```

RDF supports:

Types Using the attribute `type` which can assume an URI as value.

Collections Subjects and objects can be bags, sequences or alternatives.

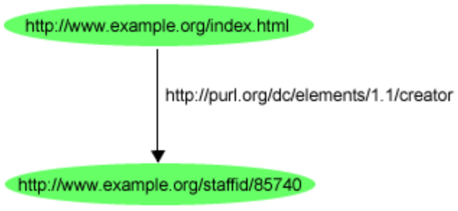
Meta-sentences Reification of the sentences (e.g. "X says that Y...").

RDF schema RDF can be used to describe classes and relations with other classes (e.g. `type`, `subClassOf`, `subPropertyOf`, ...) RDF schema

Representation

Graph A graph where nodes are subjects or objects and edges are predicates.

Example.


http://www.example.org/index.html
http://purl.org/dc/elements/1.1/creator
http://www.example.org/staffid/85740

The graph stands for: `http://www.example.org/index.html` has a `creator` with staff id `85740`.

XML

Example.

```
<rdf:RDF
xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
xmlns:contact=http://www.w3.org/2000/10/swap/pim/contact#>
    <contact:Person rdf:about="http://www.w3.org/People/EM/
   contact#me">
        <contact:fullName>Eric Miller</contact:fullName>
        <contact:mailbox rdf:resource="mailto:em@w3.org"/>
        <contact:personalTitle>Dr.</contact:personalTitle>
    </contact:Person>
</rdf:RDF>
```

Database similarities RDF aims to integrate different databases:

- A DB record is a RDF node.
- The name of a column can be seen as a property type.
- The value of a field corresponds to the value of a property.

RDFa

RDFa Specification to integrate XHTML and RDF.

SPARQL

SPARQL Language to query different data sources that support RDF (natively or through a middleware).

Ontology web language (OWL)

Ontology web language (OWL) Ontology based on RDF and description logic fragments. Three level of expressivity are available:

- OWL lite.
- OWL DL.
- OWL full.

An OWL has:

Classes Categories.

Properties Roles and relations.

Instances Individuals.

5.2 Knowledge graphs

Knowledge graph

Knowledge graph Knowledge graphs overcome the computational complexity of T-box reasoning with semantic web and description logics.

- Use a simple vocabulary with a simple but robust corpus of types and properties adopted as a standard.
- Represent a graph with terms as nodes and edges connecting them. Knowledge is therefore represented as triplets `(h, r, t)` where `h` and `t` are entities and `r` is a relation.
- Logic formulas are removed. T-box and A-box can be seen as the same concept. There is no reasoning but only facts.

- Data does not have a conceptual schema and can come from different sources with different semantics.
- Graph algorithms to traverse the graph and solve queries.

KG quality Quality

Coverage If the graph has all the required information.

Correctness If the information is correct (can be objective or subjective).

Freshness If the content is up-to-date.

Graph embedding Project entities and relations into a vectorial space for ML applications. Graph embedding

Entity prediction Given two entities `h` and `t`, determine the relation `r` between them.

Link prediction Given an entity `h` and a relation `t`, determine an entity `t` related to `h`.

6 Time reasoning

6.1 Propositional logic

State The current state of the world can be represented as a set of propositions that are true according the observation of an agent. State

The union of a countable sequence of states represents the evolution of the world. Each proposition is distinguished by its time step.

Example. A child has a bow and an arrow, then shoots the arrow.

$$\text{KB}^0 = \{\texttt{hasBow}^0, \texttt{hasArrow}^0\}$$
$$\text{KB}^1 = \{\texttt{hasBow}^0, \texttt{hasArrow}^0, \texttt{hasBow}^1, \neg\texttt{hasArrow}^1\}$$

Action An action indicates how a state evolves into the next one. It is described using effect axioms in the form: Action

$$\texttt{action}^t \Rightarrow (\texttt{preconditions}^t \iff \texttt{effects}^{t+1})$$

Frame problem The effect axioms of an action do not tell what remains unchanged in the next state. Frame problem

Frame axioms The frame axioms of an action describe the unaffected propositions of an action. Frame axioms

Example. The action of shooting an arrow can be described as:

$$\texttt{SHOOT}^t \Rightarrow \{\texttt{hasArrow}^t \iff \neg\texttt{hasArrow}^{t+1}\}$$
$$\texttt{SHOOT}^t \Rightarrow \{\texttt{hasBow}^t \iff \texttt{hasBow}^{t+1}\}$$

Note that with m actions and n propositions, the number of frame axioms will be of order $O(mn)$. Inference for t time steps will have complexity $O(nt)$.

6.2 Situation calculus (Green's formulation)

Situation calculus uses first order logic instead of propositional logic.

Situation The initial state is a situation. Applying an action in a situation is a situation: Situation

$$s \text{ is a situation and } \texttt{a} \text{ is an action } \iff \texttt{result}(\texttt{a}, s) \text{ is situation}$$

(Note: in FAIRK module 1, `result` is denoted as `do`).

Fluent Function that varies depending on the situation (i.e. tells if a property holds in a given situation). Fluent

Example. $\texttt{hasBow}(s)$ where s is a situation.

Action Actions are described using:

Action

Possibility axioms Indicates the preconditions $\phi_{\texttt{a}}$ of an action `a` in a given situation s:

Possibility axioms

$$\phi_{\texttt{a}}(s) \Rightarrow \texttt{poss}(\texttt{a}, s)$$

Successor state axiom The evolution of a fluent F follows the axiom:

Successor state axiom

$$F^{t+1} \iff (\texttt{ActionCauses}(F) \vee (F^t \wedge \neg\texttt{ActionCauses}(\neg F)))$$

In other words, a fluent is true if an action makes it true or does not change if the action does not involve it.

Adding the notion of possibility, an action can be described as:

$$\begin{aligned}\texttt{poss}(\texttt{a}, s) \Rightarrow \Big(F(\texttt{result}(\texttt{a}, s)) \iff \\ (\texttt{a} = \texttt{ActionCauses}(F)) \vee \\ (F(s) \wedge \texttt{a} \neq \neg\texttt{ActionCauses}(\neg F))\Big)\end{aligned}$$

Unique action axiom Only a single action can be executed in a situation to avoid non-determinism.

Unique action axiom

6.3 Event calculus (Kowalski's formulation)

Event calculus reifies fluents and events (actions) as terms (instead of predicates).

Event calculus ontology A fixed set of predicates:

Event calculus ontology

$\texttt{holdsAt}(F, T)$ The fluent F holds at time T.

$\texttt{happens}(\texttt{E}, T)$ The event `E` (i.e. execution of an action) happened at time T.

$\texttt{initiates}(\texttt{E}, F, T)$ The event `E` causes the fluent F to start holding at time T.

$\texttt{terminates}(\texttt{E}, F, T)$ The event `E` causes the fluent F to cease holding at time T.

$\texttt{clipped}(T_i, F, T_j)$ The fluent F has been made false between the times T_i and T_j ($T_i < T_j$).

$\texttt{initially}(F)$ The fluent F holds at time 0.

Domain-independent axioms A fixed set of axioms:

Domain-independent axioms

Truthness of a fluent

1. A fluent holds if an event initiated it in the past and has not been clipped.

$$\begin{aligned}\texttt{holdsAt}(F, T_j) \Leftarrow & \texttt{happens}(\texttt{E}, T_i) \wedge (T_i < T_j) \wedge \\ & \texttt{initiates}(\texttt{E}, F, T_i) \wedge \neg\texttt{clipped}(T_i, F, T_j)\end{aligned}$$

2. A fluent holds if it was initially true and has not been clipped.

$$\texttt{holdsAt}(F, T) \Leftarrow \texttt{initially}(F) \wedge \neg\texttt{clipped}(0, F, T)$$

Note: the negations make the definition of these axioms in Prolog unsafe.

Clipping of a fluent

$$\texttt{clipped}(T_i, F, T_j) \Leftarrow \texttt{happens}(\texttt{E}, T) \land (T_i < T < T_j) \land \texttt{terminates}(\texttt{E}, F, T)$$

Domain-dependent axioms Domain-specific axioms defined using the predicates `initially`, `initiates` and `terminates`.

Domain-dependent axioms

Deductive reasoning Event calculus only allows deductive reasoning: it takes as input the domain-dependant axioms and a set of events, and computes a set of true fluents. If a new event is observed, the query need to be recomputed again.

Example. A room with a light and a button can be described as:

Fluents `lightOn` · `lightOff`

Events `PUSH_BUTTON`

Domain-dependent axioms are:

Initial state `initially(lightOff)`

Effects of `PUSH_BUTTON` on `lightOn`

- $\texttt{initiates}(\texttt{PUSH\_BUTTON}, \texttt{lightOn}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOff}, T)$
- $\texttt{terminates}(\texttt{PUSH\_BUTTON}, \texttt{lightOn}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOn}, T)$

Effects of `PUSH_BUTTON` on `lightOff`

- $\texttt{initiates}(\texttt{PUSH\_BUTTON}, \texttt{lightOff}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOn}, T)$
- $\texttt{terminates}(\texttt{PUSH\_BUTTON}, \texttt{lightOff}, T) \Leftarrow \texttt{holdsAt}(\texttt{lightOff}, T)$

A set of events could be:

$$\texttt{happens}(\texttt{PUSH\_BUTTON}, 3) \cdot \texttt{happens}(\texttt{PUSH\_BUTTON}, 5) \cdot \texttt{happens}(\texttt{PUSH\_BUTTON}, 6)$$

6.3.1 Reactive event calculus

Allows to add events dynamically without the need to recompute the result.

Reactive event calculus

6.4 Allen's logic of intervals

Event calculus only captures instantaneous events that happen in given points in time.

Allen's logic of intervals Reasoning on time intervals.

Allen's logic of intervals

Interval An interval i starts at a time $\texttt{begin}(i)$ and ends at a time $\texttt{end}(i)$.

Interval

Temporal operators

Temporal operators

- $\texttt{meet}(i, j) \iff \texttt{end}(i) = \texttt{begin}(j)$
- $\texttt{before}(i, j) \iff \texttt{end}(i) < \texttt{begin}(j)$
- $\texttt{after}(i, j) \iff \texttt{before}(j, i)$
- $\texttt{during}(i, j) \iff \texttt{begin}(j) < \texttt{begin}(i) < \texttt{end}(i) < \texttt{end}(j)$
- $\texttt{overlap}(i, j) \iff \texttt{begin}(i) < \texttt{begin}(j) < \texttt{end}(i) < \texttt{end}(j)$

- $\texttt{starts}(i,j) \iff \texttt{begin}(i) = \texttt{begin}(j)$
- $\texttt{finishes}(i,j) \iff \texttt{end}(i) = \texttt{end}(j)$
- $\texttt{equals}(i,j) \iff \texttt{starts}(i,j) \land \texttt{ends}(i,j)$

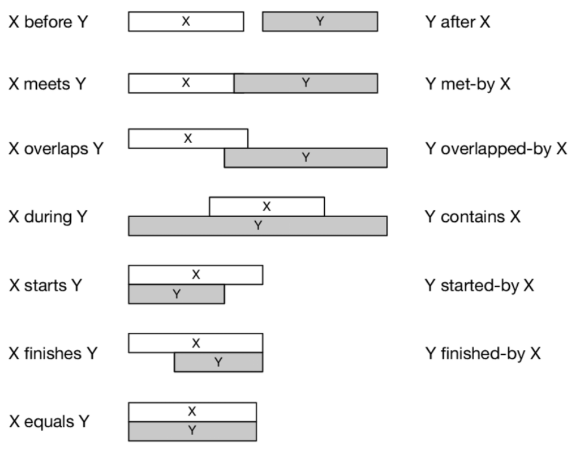


Figure 6.1: Visual representation of temporal operators

6.5 Modal logics

Logic based on interacting agents with their own knowledge base.

Propositional attitudes Operators to represent knowledge and beliefs of an agent towards the environment and other agents. Propositional attitudes

First-order logic is not suited to represent these operators.

Modal logics Modal logics have the same syntax of first-order logic with the addition of modal operators. Modal logics

Modal operator A modal operator takes as input the name of an agent and a sentence (instead of a term as in FOL).

Knowledge operator Operator to indicate that an agent $\texttt{a}$ knows P: Knowledge operator

$$\mathbf{K}_{\texttt{a}}(P)$$

Belief operator

Everyone knows operator

Common knowledge operator

Distribute knowledge operator

Depending on the operators, different modal logics can be defined.

Semantics An agent has a current perception of the world and considers the unknown as other possible worlds. Moreover, if P is true in any accessible world from the current one, the agent has knowledge of P.

Formally, semantics is defined on a set of primitive propositions ϕ using a Kripke structure $M = (S, \pi, K_{\texttt{1}}, \dots, K_{\texttt{n}})$ where:

- S is a set of states of the world.
- $\pi : \phi \to 2^S$ specifies in which states each primitive proposition holds.
- $K_{\texttt{i}} \subseteq S \times S$ is a binary relation where $(s, t) \in K_{\texttt{i}}$ if an agent i considers the world t possible (accessible) from s. In other words, when the agent is in the world s, it considers t to be a possibly valid world. Obviously, $(s, s) \in K_{\texttt{i}}$ for all states.

Example. Alice is in a room an tosses a coin. Bob is in another room an will enter Alice's room when the coin lands to observe the result.

We define a model $M = (S, \pi, K_{\texttt{a}}, K_{\texttt{b}})$ on ϕ where:

- $\phi = \{\texttt{tossed}, \texttt{heads}, \texttt{tails}\}$.
- $S = \{s_0, h_1, t_1, h_2, t_2\}$ where the possible states are divided in three stages: the initial state (s_0), the result of the coin flip (h_1, t_1) and the observation of Bob (h_2, t_2).
- $\pi(\texttt{tossed}) = \{h_1, t_1, h_2, t_2\}$
  $\pi(\texttt{heads}) = \{h_1, h_2\}$
  $\pi(\texttt{tails}) = \{t_1, t_2\}$
- $K_{\texttt{a}} = \{(s, s) \mid s \in S\}$ as Alice observes everything in each world and does not have doubts.
  $K_{\texttt{b}} = \{(s, s) \mid s \in S\} \cup \{(h_1, t_1), (t_1, h_1)\}$ as Bob is unsure of what happens in the second stage.

With this model, we can determine the truthness of sentences like:

$$(M, s_0) \models K_{\texttt{a}}(\neg\texttt{tossed}) \wedge K_{\texttt{b}}\Big(K_{\texttt{a}}\big(K_{\texttt{b}}(\neg\texttt{heads} \wedge \neg\texttt{tails})\big)\Big)$$

$$(M, t_1) \models (\texttt{heads} \vee \texttt{tails}) \wedge \neg\texttt{K}_{\texttt{b}}(\texttt{heads}) \wedge \neg\texttt{K}_{\texttt{b}}(\texttt{tails}) \wedge \texttt{K}_{\texttt{b}}(\texttt{K}_{\texttt{a}}(\texttt{heads}) \vee \texttt{K}_{\texttt{a}}(\texttt{tails}))$$

Axioms

Tautology All propositional tautologies are valid.

Modus ponens If φ and $\varphi \Rightarrow \psi$ are valid, then ψ is valid.

Distribution axiom Knowledge is closed under implication:

$$(K_{\texttt{i}}(\varphi) \wedge K_{\texttt{i}}(\varphi \Rightarrow \psi)) \Rightarrow K_{\texttt{i}}(\psi)$$

Knowledge generalization rule An agent knows all the tautologies:

$$\forall \text{ structures } M : (M \models \varphi) \Rightarrow (M \models K_{\texttt{i}}(\varphi))$$

Knowledge axiom If an agent knows φ, then φ is true:

$$K_{\texttt{i}}(\varphi) \Rightarrow \varphi$$

In belief logic, this axiom is substituted with $\neg K_{\texttt{i}}(\texttt{false})$.

Introspection axioms An agent is sure of its knowledge:

Positive $K_{\texttt{i}}(\varphi) \Rightarrow K_{\texttt{i}}(K_{\texttt{i}}(\varphi))$

Negative $\neg K_{\texttt{i}}(\varphi) \Rightarrow K_{\texttt{i}}(\neg K_{\texttt{i}}(\varphi))$

Different modal logics can be defined based on the valid axioms.

6.6 Temporal logics

Logics based on modal logic with the addition of a temporal dimension. Time is discrete and each world is labeled with an integer. The accessibility relation maps into the temporal dimension with two possible evolution alternatives:

Linear-time From each world, there is only one other accessible world. Linear-time

Branching-time From each world, there are many accessible worlds. Branching-time

6.6.1 Linear-time temporal logic

Operators

Next ($\bigcirc\varphi$) φ is true in the next time step. Next

Globally ($\Box\varphi$) φ is always true from now on. Globally

Future ($\diamond\varphi$) φ is true sometimes in the future. It is equivalent to $\neg\Box(\neg\varphi)$. Future

Until ($\varphi\mathcal{U}\psi$) There exists a moment (now or in the future) when ψ holds. φ is guaranteed to hold from now until ψ starts to hold. Until

Weak until ($\varphi\mathcal{W}\psi$) There might be a moment when ψ holds. φ is guaranteed to hold from now until ψ possibly starts to hold. Weak until

Semantics Given a Kripke structure $M = (S, \pi, K_1, \dots, K_n)$ where states are represented using integers, the semantic of the operators is the following:

- $(M, i) \models P \iff i \in \pi(P)$.
- $(M, i) \models \bigcirc\varphi \iff (M, i+1) \models \varphi$.
- $(M, i) \models \Box\varphi \iff \forall j \geq i : (M, j) \models \varphi$.
- $(M, i) \models \varphi\mathcal{U}\psi \iff \exists k \geq i : \big((M, k) \models \psi \wedge \forall j . i \leq j \leq k : (M, j) \models \varphi\big)$.
- $(M, i) \models \varphi\mathcal{W}\psi \iff ((M, i) \models \varphi\mathcal{U}\psi) \vee ((M, i) \models \Box\varphi)$.

Model checking Methods to prove properties of linear-time temporal logic based finite state machines or distributed systems. Model checking

7 Probabilistic logic reasoning

Probabilistic logic programming Adds probability distributions over logic programs allowing to define different worlds. Joint distributions can also be defined over worlds and allows to answer to queries.

Probabilistic logic programming

7.1 Logic programs with annotated disjunctions (LPAD)

7.1.1 Syntax

LPAD

`null` Atom that can only appear in the head of a clause and cancels the clause (i.e. equivalent of not having the clause).

The head of each clause is defined as a disjunction of atoms, each with a probability. More specifically, each clause has a probability distribution over its head.

Example.

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

7.1.2 Distribution semantics

Worlds Given a clause C and a substitution θ such that $C\theta$ is ground, the following operations are defined for LPAD:

World

Atomic choice An atomic choice (C, θ, i) is the selection of the i-th atom in the head of C for grounding.

Atomic choice

Composite choice A composite choice κ is a set of atomic choices. The probability of a composite choice is the following:

Composite choice

$$\mathcal{P}(\kappa) = \prod_{(C,\theta,i)\in\kappa} \mathcal{P}(C, i)$$

where $\mathcal{P}(C, i)$ is the probability of choosing the i-th atom in the head of C.

Selection A selection σ is a composite choice where an atom from the head of each clause for each grounding has been chosen. In other words, a selection can be defined only when the program is ground.

Selection

A selection σ identifies a world w_σ and has probability:

$$\mathcal{P}(w_\sigma) = \mathcal{P}(\sigma) = \prod_{(C,\theta,i)\in\sigma} \mathcal{P}(C, i)$$

Example. Given the program:

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

The possible worlds are:

$P(w_1) = 0.7 \cdot 0.8$

```
sneezing(bob) :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_2) = 0.3 \cdot 0.8$

```
null :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_3) = 0.7 \cdot 0.2$

```
sneezing(bob) :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

$P(w_4) = 0.3 \cdot 0.2$

```
null :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

Queries Given a ground query Q and a world w, the probability of Q being true in w is trivially:

Queries

$$\mathcal{P}(Q \mid w) \begin{cases} 1 & \text{if } Q \text{ is true in } w \\ 0 & \text{otherwise} \end{cases}$$

The overall probability of Q is:

$$\mathcal{P}(Q) = \sum_{w} \mathcal{P}(Q, w) = \sum_{w} \mathcal{P}(Q \mid w)\mathcal{P}(w) = \sum_{w \models Q} \mathcal{P}(w)$$

Example. Given the program:

```
sneezing(X):0.7 ; null:0.3 :- flu(X).
sneezing(X):0.8 ; null:0.2 :- hay_fever(X).
```

The probability of `sneezing(bob)` is:

$$\mathcal{P}(\texttt{sneezing(bob)}) = \mathcal{P}(w_1) + \mathcal{P}(w_2) + \mathcal{P}(w_3)$$

8 Forward reasoning

Logical implication Simplest form of rule: Logical implication

$$p_1, \dots, p_n \Rightarrow q_1, \dots, q_m$$

where:

Left hand side (LHS) $p_1, \dots, p_n$

Right hand side (RHS) $q_1, \dots, q_m$

Modus ponens If A and $A \Rightarrow B$ are true, then we can derive that B is true. Modus ponens

Production rules Approach that allows to dynamically add facts to the knowledge base (differently from backward reasoning in Prolog). Production rules

When a fact is added, the reasoning mechanism is triggered:

Match Search for the rules whose LHS match the fact and (arbitrarily) decide which to trigger.

Conflict resolution Triggered rules are put in an agenda where conflicts are solved.

Execution The RHS of the triggered rules are executed and the effects are performed. The knowledge base is updated with the (copies of the) new facts.

These steps are executed until quiescence as the execution step may add new facts.

Working memory Data structure that contains the currently valid set of facts and rules. Working memory

The performance of a production rules system depends on the efficiency of the working memory.

8.1 RETE algorithm

RETE is an efficient algorithm for implementing rule-based systems.

8.1.1 Match

Pattern The LHS of a rule is expressed as a conjunction of patterns (conditions). Pattern

A pattern can test:

Intra-element features Features that can be tested directly on a fact.

Inter-element features Features that involves more facts.

Conflict set Set of all possible instantiations of production rules. Each rule is described as: Conflict set

$$\langle \text{Rule, list of facts matched by its LHS} \rangle$$

Instead of naively checking a rule over all the facts, each rule has associated the facts that match its LHS patterns.

LHS network Compile the LHSs into networks:

- **Alpha-network** For intra-element features. The outcome is stored into alpha-memories and used by the beta network. Alpha-network
- **Beta-network** For inter-element features. The outcome is stored into beta-memories and corresponds to the conflict set. Beta-network

If more rules use the same pattern, the node of that pattern is reused and possibly outputting to different memories.

8.1.2 Conflict resolution

RETE allows different strategies to handle conflicts:

- Rule priority.
- Rule ordering.
- Temporal attributes.
- Rule complexity.

The best approach depends on the use case.

8.1.3 Execution

By default, RETE executes all the rules in the agenda and then checks possible side effects that modified the working memory in a second moment.
Note that it is very easy to create loops.

8.2 Drools framework

RETE-based rule engine that uses Java. Drools

Rule A rule has structure:

```
rule "rule␣name"
    // Rule attributes
when
    // LHS
then
    // RHS
end
```

Quantifiers

- `exists P(...)` Trigger the rule once if at least a fact `P(...)` exists in the working memory.
- `forall P(...)` Trigger the rule if all the instances of `P(...)` match. The rule can be executed multiple times.
- `not P(...)` Trigger the rule if the fact `P(...)` does not exist in the working memory. Note that a negation in different positions might result in different behaviors.

Consequences Drools allows two types of RHS operations:

- **Logic**
  - **Insert** Create a new fact and insert it in the working memory. Existing rules may trigger if they match the new fact.

    If the conditions of the rule that inserted a fact are no longer true, the inserted fact is automatically retracted.
  - **Retract** Remove a fact from the working memory.
  - **Modify** A combination of retract and insert executed consecutively. The `no-loop` keyword can be used to avoid loops.
- **Non-logic** Execution of Java code or external side effects.

Conflict resolution

- **Salience score**
- **Agenda group** Associate a group to each rule. The method `setFocus` can be used to prioritize certain groups.
- **Activation group** Only one rule among the ones with the same activation group is executed (i.e. mutual exclusion).

8.3 Complex event processing

Event Information with a description and temporal information (instantaneous or with a duration). Event

Simple event Event detected outside an event processing system (e.g. a sensor). It does not provide any information alone. Simple event

Complex event Event generated by an event processing system and provides higher informative payload. Complex event

Complex event processing (CEP) Paradigm for dealing with a large amount of information. Takes as input different types of events and outputs durative events. Complex event processing

8.3.1 Drools

Drools supports CEP by representing events as facts.

Clock Mechanism to specify time conditions to reason over temporal intervals.

Sliding windows

- **Time-based window** Select events within a time slice.
- **Length-based window** Select the last n events.

Expiration Mechanism to specify an expiration time to events and discard them from the working memory.

Temporal reasoning Allen's temporal operators for temporal reasoning.