Fundamentals of Artificial Intelligence and Knowledge Representation

Module 2

Matteo Donati

November 13, 2021

Contents

Chapter 1

Introduction

Humans are problem solvers, i.e. we use knowledge to solve problems. In particular, knowledge can be divided into, at least, two categories:

- Declarative knowledge. This type of knowledge is more intuitive and human readable but it cannot be used as a computational tool (e.g. a declarative program).
- Imperative knowledge. This type of knowledge can be thought as some kind of recipe and it cannot be easily understood (e.g. an imperative program).

1.1 Knowledge Representation

Knowledge representation (KR) is the field of artificial intelligence concerned with how (declarative) knowledge can be represented symbolically and manipulated in an automated way by reasoning programs. In particular:

- Knowledge representation languages are formal languages designed for representing (declarative) knowledge, where knowledge is usually encoded using a symbolic representation as a set of clauses.
- A **knowledge base** (KB) is a set of clauses representing the encoding of knowledge relevant for a given problem or domain.
- An **inference engine** is a program that takes as input a knowledge base, in a given knowledge representation language, and performs a reasoning process. These engines can be modelled by a set of inference rules having the following form:

$$\frac{\text{Preconditions}}{\text{Actions}} \tag{1.1}$$

For example:

$$\frac{\text{rule}^i \wedge \text{rule}^j}{\text{rule}^k}$$

if rule^i and rule^j are in the knowledge base, then it is possible to add rule^k to the given knowledge base.

- **Reasoning** is the formal manipulation of the symbols representing a collection of believed propositions to produce representations of new ones. A reasoning process (i.e. deduction) performed by an engine starts from an initial knowledge base and generates new knowledge bases executing inference rules for adding and/or removing clauses from the considered knowledge base.

$$\mathrm{KB}^0 \rightarrow \mathrm{KB}^1 \rightarrow \cdots \rightarrow \mathrm{KB}^n \tag{1.2}$$

  A reasoning process stops when a goal is reached (i.e. added to the knowledge base) or when no inference rules can be applied. Moreover, an engine properties are the following:

  - **Soundness**, i.e. clauses generated in the reasoning process are semantically *true*.
  - **Completeness**, i.e. all the clauses semantically *true* can be generated in the reasoning process.
  - **Decidability**, i.e. the reasoning process always terminate with success or failure.
  - **Efficiency**, i.e. the reasoning process is tractable.

Chapter 2

Symbolic Programming in LISP

LISP is a symbolic programming language invented in 1958 by John McCarthy. LISP-like syntax and notations are often used in artificial intelligence systems. LISP is also a functional language, i.e. a language used to write programs which are expressed as a set of function definitions:

```
f1(X, Y) = +(X, Y)
f2(X) = f1(X, X)
f3(Y, X) = f2(f1(Y, X))
```

In particular, program execution is expressed as a sequence of function evaluations:

$$\texttt{f2(3)} \rightarrow \texttt{f1(3, 3)} \rightarrow \texttt{+(3, 3)} \rightarrow \texttt{6}$$

Moreover:

- An example of LISP application is the following:

```
unix# sbcl
* 486
486
* (+ 2 3)
5
* (* 5 99)
495
* (a b c)
....undefined function: a....
* (quote (a b c))
(a b c)
* '(a b c)
(a b c)
* (quit)
unix#
```

In particular, `*` is the SBCL (Steel Bank Common Lisp) prompt. The LISP interpreter always tries to evaluate the input. The quote function, `quote` or `'`, suspends the evaluation for its argument.

- Symbols in LISP are used as variables, function names, parameters and symbolic information.
- A function is defined as follows:

```
(defun FUN-NAME (PAR1 PAR2 ... PARm) (EXPR1) (EXPR2) ... (EXPRn))
```

The body of a function is a sequence of expressions and a specific function always returns the value of the last expression, `EXPRn`. Functions without any parameter use the empty list. For example:

(defun FUN-NAME () ...)

```
* (defun square (X) (* X X))
square
* (square 21)
441
* (square (+ 2 5))
49
```

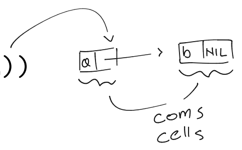
- Conditionals are expressed using a sequence of pairs (condition, expression):

```
(cond (<p1> <e1>) (<p2> <e2>) (<p3> <e3>) ... (<pn> <en>))
```

Such pairs are evaluated in sequence. When the first condition succeeds, the corresponding expression is evaluated and `cond` returns its value. If none of the conditions are *true*, then `cond` returns `NIL` (i.e. *false*).

- Lists are created using the ~~`cond`~~ cons function:

```
* (cons 'a 'b)
(a . b)
* (cons a NIL)
(a)
* (cons a (cons b NIL))
(a b)
```



A list can be accessed using the `car` and `cdr` functions:

```
* (car (cons 'a 'b)) # first
a
* (cdr (cons 'a 'b)) # rest
b
```

Nested `car` and `cdr` calls can be shrunk:

```
* (car (cdr (car (cdr '((1 2) (3 4))))))
4

* (cadadr '((1 2) (3 4)))
4
```

One can also assign a list to a symbol:

```
* (setq mylist (cons 'a (cons 'b (cons 'c ()))))
mylist
* mylist
(a b c)
```

Moreover, there exist multiple useful functions which work on lists:

- The `member` function takes in input a symbol and a list of symbols, and returns `T` (i.e. *true*) is the first symbol is contained into the list, `NIL` (i.e. *false*) otherwise:

  ```
  (defun member (A L)
    (cond ((null L) NIL)
          ((eq A (car L)) L)
          (T (member A (cdr L)))
    ))
  ```

- The `equal` function takes in input two lists and compares them:

  ```
  (defun equal (X Y)
    (cond ((and (atom X) (atom Y)) (eq X Y))
          ((equal (car X) (car Y))
           (equal (cdr X) (cdr Y))
          (T NIL)
  ```

  where `(and A B)` returns T if both `A` and `B` are different than `NIL`, and `(atom A)` returns `T` is `A` is an atom (i.e. a number, a sylbol or `NIL`).

- The `appen` function takes two lists and appends them:

  ```
  (defun append (L1 L2)
    (cond ((null L1) L2)
          (T (cons (car L1) (append (cdr L1) L2))
    ))
  ```

- The `mapcar` function applies the function `F` on all the arguments of the `L` list and builds the list of results:

```
(defun mapcar (F L)
  (cond ((eq L NIL) NIL)
        (T (cons (funcall F (car L)) (mapcar F (cdr L))))))
```

- LISP macros allow one to define new operators that are implemented by transformations. Such macros work differently from normal functions. In particular, macros produce expressions which when evaluated produce results. For example, if one wants to define a new conditional form:

  ```
  (defmacro newif (p e1 e2)
    (cons (quote cond) (cons (cons p (cons e1 NIL))
                             (cons (cons T (cons e2 NIL)) NIL)))
  ```

  ```
  (newif p e1 e2) => (cond (p e1) (T e2))
        ^
        || equivalent
        v
  (defmacro newif (p e1 e2)
     `(cond (,p ,e1) (T ,e2)))
  ```

  backquote operator (similar to quote but it can include entry points)

  (the expressions after commas are evaluated and then inserted in the list

Chapter 3

Propositional Logic (as KR language)

Propositional logic can be used as a knowledge representation language. In particular

- The syntax of propositional logic, in particular well-formed formulas, defines the knowledge base.
- The semantics of propositional logic is based on the notion of truth. An **assignment** of truth or **interpretation** is a function:

$$\alpha : \overset{\text{set of all atomic formulas}}{\mathbb{A}} \to \{0, 1\} \tag{3.1}$$

  Using truth tables one can extend α and assign a truth value to any formula. - semantic of a formula
- A **model** for a knowledge base is an assignment (i.e. a possible world) in which each clause in the knowledge base is *true*.
- A knowledge base is **satisfiable** if and only if it exists an assignment which is a model for it.
- A knowledge base is **unsatisfiable** if and only if it does not exists any assignment which is a model for it.
- A formula F is a **logical consequence** of a knowledge base, written $KB \models F$ if and only if whenever KB is *true* also F is *true*.

3.1 Reasoning in Propositional Logic

Semantic methods (i.e. model checking) for checking logical consequences of a knowledge base (e.g. in order to expand the knowledge base with new knowledge) are conceptually simple but the number of assignments of truth grows exponentially with the number of atomic formulas in the knowledge base. In order to avoid this, one can rely on formal methods which instead syntactically manipulates well-formed formulas:

- **Reasoning** is a syntax based mechanical process, if one or more well-formed formulas in the knowledge base matches a certain syntax, then an inference can be activated.

- $KB \vdash^E F$ means that the formula F can be deduced by the knowledge base KB using engine E.
- An example of engine is represented by **natual deduction**. Some useful natual deduction's inference rules are the following:
  - **Introduction rules**:

$$(\wedge I) \quad \frac{\varphi \qquad \psi}{\varphi \wedge \psi}\wedge I \qquad\qquad (\to I) \quad \frac{\begin{matrix}[\varphi] \\ \vdots \\ \psi\end{matrix}}{\varphi \to \psi}\to I$$

  - **Elimination rules**:

$$(\wedge E) \quad \frac{\varphi \wedge \psi}{\varphi}\wedge E\ , \qquad \frac{\varphi \wedge \psi}{\psi}\wedge E \qquad\qquad (\to E) \quad \frac{\varphi \qquad \varphi \to \psi}{\psi}\to E$$

  For example, starting from $A \wedge B$ it is possible to infer $(A \wedge B) \to A$:

$$\cfrac{\cfrac{A \wedge B}{A}\wedge E}{(A \wedge B) \to A}\to I$$

  However, natural deduction has many problems. For example: it uses many inference rules, it lacks effective heuristics for selecting the inference rules to apply and the connection between the reasoning process and the goal is not clear.
- In order to ~~simply~~ simplify the syntax, one can define a specific knowledge base as a set of clauses (i.e. a set formulas in disjunctive normal form).
- The **resolution inference** rule is another example of inference engine. In particular:

$$\frac{B \vee A \qquad \neg B \vee C}{A \vee C}$$

  In order to prove some theory, one can use the concept of refutation (i.e. $KB \cup \{\neg G\} \models \Box$ if and only if $KB \models G$) and the resolution inference rule to infer the empty clause $\Box$ (i.e. $false$). For example:

$$\frac{\neg G \qquad G}{\Box}$$

this means that $KB \models G$.

When using resolution, one can also follow a heuristic in order ~~to order~~ to select which clause to resolve. By doing so, refutation plus resolution solve the afore-mentioned problems related to natural deduction (i.e. two many inference rules, lacking of a heuristic and connection between reasoning process and goal not clear).

- However, checking if there exist a possible assignment of truth which satisfy a given formula is NP-complete (i.e. one has to list every possible combination of truth values by using the truth table). In order to reduce such complexity one can apply refutation and resolution using **Horn clauses**. In particular, there are three types of Horn clauses:
  - **Facts**, which are atomic formulas.
  - **Rules**, which are clauses having the form $\neg A \vee \neg B \vee \neg C \vee D$ (logically equivalent to $(A \wedge B \wedge C) \to D$, where $(A \wedge B \wedge C)$ is the body of the rule and D is the head of the rule).
  - **Goals**, which are negative literals that represent what needs to be proven (e.g. if one want to prove that $KB \models (P \wedge Q)$, then the corresponding negated goal is $\neg P \vee \neg Q \equiv \neg(P \wedge Q)$).
- A possible implementation of refutation and resolution using Horn clauses is given by the **SLD resolution**.

Chapter 4

First Order Logic

Propositional logic turns out to be too simple to make specific statements in specific domains. First order logic solves this problem by adding *quantifiers* and substituting propositions with *predicates* applied to *terms*. In particular:

- The syntax of first order logic is composed of the following symbols:
  - An infinite set of constants.
  - An infinite set of variables.
  - An infinite set of functional symbols of all arities.
  - An infinite set of predicates symbols of all arities.
  - $\wedge$, $\vee$, $\rightarrow$, $\neg$ connectives.
  - $\forall$, $\exists$ quantifiers.

  Based on these structures, one can define the set of possible terms as follows:

  - A term is a constant or a variable.
  - A term is a functional symbol of arity n.

  For example: bob, x, $room(\text{R3.4}, \text{engineering}, 100)$ are all terms.

- The set of well-formed formulas is composed as follows:
  - Predicate symbols of arity n (i.e. which take as inputs n terms) are well-formed formulas.
  - If F_1 and F_2 are well-formed formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$ and $\neg F_1$ are well-formed formulas.
  - If F is a well-formed formula which contains a variable x, then $\forall x.F(x)$ and $\exists x.F(x)$ are well-formed formulas.

- The semantic of a formula is related to the concept of interpretation. An interpretation I consists of:

  - A universe U which is non-empty.
  - A function $I(f) : U^n \to U$ for every n-ary function symbol f.
  - A relation, $I(p) \subseteq U^n$ for every n-ary predicate symbol p.

  Moreover, starting from an interpretation one can compute the semantics of all the first order logic formulas as follows:

  - If a formula F contains logic connectives (i.e. $F = F_1 \land F_2$ or $F_1 \lor F_2$ or $F_1 \to F_2$, or $\neg F_1$), one can use truth tables to determine the semantics of F.
  - A formula of the form $\exists x.F(x)$ is *true* if there exists an object *obj* such that $F(obj)$ is *true*.
  - A formula of the form $\forall x.F(x)$ is *true* is for all the objects *obj*, $F(obj)$ is *true*.

- A **model** for a knowledge base is an interpretation in which each formula is *true*.
- A knowledge base is **satisfiable** if and only if there exists an interpretation which is a model for the given knowledge base.
- A knowledge base if **unsatisfiable** if and only if there exists no interpretation which is a model for the given knowledge base.
- F is a **logical consequence** of a knowledge base, $KB \models F$, if whenever KB is *true* then F is *true*.
- Two formulas, F_1 and F_2 are **logically equivalent**, $F_1 \equiv F_2$ if they have the same semantics for all the possible interpretations.

4.1 Reasoning in First Order Logic

Reasoning in first order logic is similar to reasoning in propositional logic but more complex. In particular, in order to define an inference engine, one needs to introduce the concepts of substitution and unification:

- An occurrence of a variable x in a formula is said to be **bound** if it is contained within a sub-formula of the form $\forall x.A$ or $\exists x.A$. Such type of variables can be renamed without changing the semantic of a formula.
- An occurrence of a variable if said to be **free** if it is not bound.
- A **Closed formula** is a formula which contains no free variables.
- A **ground formula** is a formula which contains no variables.
- A **substitution** is a finite set of replacements for the variables of one or more terms.
- A substitution θ is a **unifier** of two terms, t_1 and t_2, if $t_1\theta = t_2\theta$. Moreover, the substitution θ is more general then ϕ if $\phi = \theta \circ \sigma$, for some other substitution σ. For example, $f(x, b)$ and $f(a, y)$ unify by considering $\theta = [x/a, y/b]$, and have the common instance $f(a, b)$.

- A substitution θ is the **most general unifier** (MGU) of two terms, t_1 and t_2, if:
  - θ unifies t_1 and t_2.
  - θ is more general than every other possible unifier.

  For example, $p(x)$ and $p(y)$ unify with unifier $\theta = [x/a, y/b]$ and many others. However, the MGU is given by $[x/y]$.

  one has to consider substitution

- All of the natural deduction rules from propositional logic carry over to first order logic. Moreover, there are four new rules for introduction and elimination of quantifiers:

$$\frac{A(x)}{\forall y.A(y)}\,\forall I \qquad \frac{\forall x.A(x)}{A(t)}\,\exists E \qquad \frac{A(t)}{\exists x.A(x)}\,\exists I \qquad \frac{\begin{array}{c}\overline{A(y)}\\ \vdots \\ B\end{array} \quad \exists x.A(x)}{B}\,\exists E$$

  Moreover, one can use the equality symbol to state that two terms refer to the same object. For example, $mother(john) = mary$. Natural deduction can be then extended by considering the following rules:

$$\frac{}{t = t}\,\text{refl} \qquad \frac{s = t}{t = s}\,\text{symm} \qquad \frac{r = s \quad s = t}{r = t}\,\text{trans}$$

$$\frac{s = t}{r(s) = r(t)}\,\text{subst} \qquad \frac{s = t \quad P(s)}{P(t)}\,\text{subst}$$

- It is also possible to adapt the refutation and resolution engine to first order logic. In this case the resolution tree becomes infinite and the resolution rules must be extended with unification and substitutions.

  In order to use such engine, it is necessary to transform a first order logic knowledge base in conjunctive normal form. In order to do so, one has to first apply the following transformations:

  1. Transform the knowledge base in **prenex normal form**, where all the quantifiers are moved at the front of a formula. This is achieved by applying equivalences for quantifiers.
  2. Eliminating quantifiers (**skolemization**).

Chapter 5

Knowledge Engineering

Knowledge engineering (KE) is the general process of knowledge base construction:

- Identifying the task by interacting with domain experts in order to evaluate if the problem can be effectively solved with a knowledge representation system.
- Knowledge acquisition: extract the relevant knowledge of the domain:
  - Define the questions that the knowledge base will be able to answer.
  - Identify the relevant knowledge and where this knowledge is used in the reasoning process.
- Define the vocabulary of predicates, functions and constants of the domain with the associated meaning.
- Encoding of knowledge:
  - Represent the knowledge of the domain with one or more knowledge representation formalism.
  - Encode specific problem instances.
- Test and debug the knowledge base. The used inference engine should be effective.

Chapter 6

Terminological Knowledge

The organization of objects into categories is crucial in knowledge representation. There is a large amount of reasoning that takes place at the category level. For example, by using first order logic one can express category in two different ways:

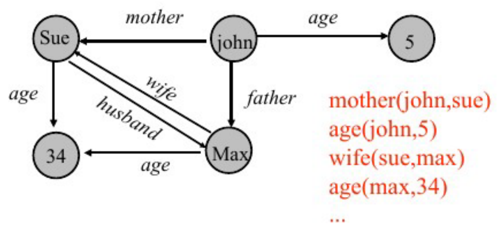
- Using predicates, e.g. $Professor(Mauro)$.
- Using objects, e.g. $Member(Mauro, Professor)$.

Categories are useful for organizing the knowledge base through inheritance, using a subclass relationship. Moreover, there exist two family of knowledge representation systems which work based on the category concept:

- **Semantic networks**, which provide graphical aids for visualizing hierarchies of categories and efficient algorithms for inferring properties of objects.
- **Description logic**, which provides a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships.

6.1 Semantic Networks

Semantic networks represent individual objects and categories of objects with ovals or boxes, and relationships among objects with labelled arcs. For example:

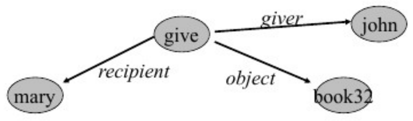


In particular:

- The possible link types are the following:

Link type	Semantics	Example
$A \xrightarrow{Subset} B$	$A \subset B$	$Cats \subset Mammals$
$A \xrightarrow{Member} B$	$A \in B$	$Bill \in Cats$
$A \xrightarrow{R} B$	$R(A, B)$	$Bill \xrightarrow{Age} 12$
$A \xrightarrow{\boxed{R}} B$	$\forall x\ x \in A \Rightarrow R(x, B)$	$Birds \xrightarrow{\boxed{Legs}} 2$
$A \xrightarrow{\boxed{\boxed{R}}} B$	$\forall x\ \exists y\ x \in A \Rightarrow y \in B \wedge R(x, y)$	$Birds \xrightarrow{\boxed{\boxed{Parent}}} Birds$

- Semantic networks only support binary relationships. Non-binary relationships can be represented by using **reification**, i.e. by turning a proposition into an object. For example, $give(John, Mary, Book)$ can be represented as follows:



  Using reification one can represent every ground atomic sentence in first order logic.

- The expressive power is still not that of first order logic. In particular, negation, disjunction, nexted function symbols and existential quantification are missing.

6.2 Description Logic

Description logics are notations that are designed to make it easier to describe definitions and properties of categories. The objective of DL is to formalize what a network means and to study the reasoning mechanisms. The principal inference tasks for DL are the following:

- **Subsumption**, i.e. checking if one category is a subset of another comparing their definitions.
- **Classification**, i.e. checking whether an object belongs to a category.
- **Consistency** of a category definition, i.e. checking whether the membership criteria are logically satisfiable.

In particular:

- DL describes a domain in terms of **concepts** (categories), **roles** (properties, relationships) and **individuals** (objects).
- DL family is a family of languages which has been developed for studying DL. The simplest logic in this family is named **AL**.

6.2.1 AL

The syntax of AL is composed of the following symbols:

- Atomic concepts, e.g. $Person$, $Doctor$.
- Roles (relationships), e.g. $HasChild$, $Loves$.
- Individuals (nominal objects), e.g. $John$, $Mary$.
- Boolean operators $\sqcap$, $\neg$.
- Restricted quantifiers: $\exists$, $\forall$.

Concepts are formed as follows:

$$\begin{array}{rcll} C, D & \rightarrow & A & \text{(atomic concept)} \\ & \mid & \top & \text{(universal concept)} \\ & \mid & \bot & \text{(bottom concept)} \\ & \mid & \neg A & \text{(atomic negation)} \\ & \mid & C \sqcap D & \text{(intersection)} \\ & \mid & \forall R.C & \text{(value restriction)} \\ & \mid & \exists R.\top & \text{(limited existential quantification)} \end{array}$$

Some examples are the following:

- $Person$ and $Female$ are atomic concepts.
- $Person \sqcap Female$, people which are female.
- $Person \sqcap^{\neg} Female$, people which are not female.
- $HasChild$ is an atomic role.
- $Person \sqcap \exists HasChild^{.\top}$, people which have a child.
- $Person \sqcap \forall HasChild.Female$, people which have children and these children are all female.
- $Person \sqcap \forall HasChild.\bot$, people which do not have a child.

The semantics of AL is based on the notion of interpretation. In particular, an interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (i.e. the domain of the interpretation) and an interpretation function which assigns to every atomic concepts A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpreation function is defined inductively as follows:

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}} \tag{6.1}$$
$$\bot^{\mathcal{I}} = \emptyset \tag{6.2}$$
$$(\neg A)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \tag{6.3}$$
$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} \tag{6.4}$$
$$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \forall b \ (a,b) \in R^{\mathcal{I}} \to b \in C^{\mathcal{I}}\} \tag{6.5}$$
$$(\exists R.\top)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \exists b \ (a,b) \in R^{\mathcal{I}}\} \tag{6.6}$$

Moreover two concepts C and D are **equivalent**, $C \equiv D$, if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$.

6.2.2 ALC

The basic description language AL is not expressive enough. Thus, ALC is introduced. ALC is sufficiently expressive to support many fundamental constructors for representing terminological knowledge. The syntax of AL is composed of the following symbols:

- Atomic concepts, e.g. $Person$, $Doctor$.
- Roles (relationships), e.g. $HasChild$, $Loves$.
- Individuals (nominal objects), e.g. $John$, $Mary$.
- Boolean operators $\sqcap$, $\sqcup$, $\neg$.
- Restricted quantifiers: $\exists$, $\forall$.

Concepts are formed as follows:

$$\begin{array}{rcll}
C, D & \to & A & \text{(atomic concept)} \\
& | & \top & \text{(universal concept)} \\
& | & \bot & \text{(bottom concept)} \\
& | & \neg A & \text{(atomic negation)} \\
& | & C \sqcap D & \text{(conjunction)} \\
& | & C \sqcup D & \text{(disjunction)} \\
& | & \exists R.C & \text{(existential restriction)} \\
& | & \forall R.C & \text{(universal restriction)}
\end{array}$$

The concept of interpretation is the same as the one considered in AL, but the interpretation function is defined as follows:

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}} \tag{6.7}$$
$$\bot^{\mathcal{I}} = \emptyset \tag{6.8}$$
$$(\neg A)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}} \tag{6.9}$$
$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} \tag{6.10}$$
$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}} \tag{6.11}$$
$$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \tag{6.12}$$
$$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \tag{6.13}$$

An interpretation $\mathcal{I}$ is a **model** for a concepts C if $C^{\mathcal{I}} \neq \emptyset$.

6.2.3 Description Logic Knowledge Bases

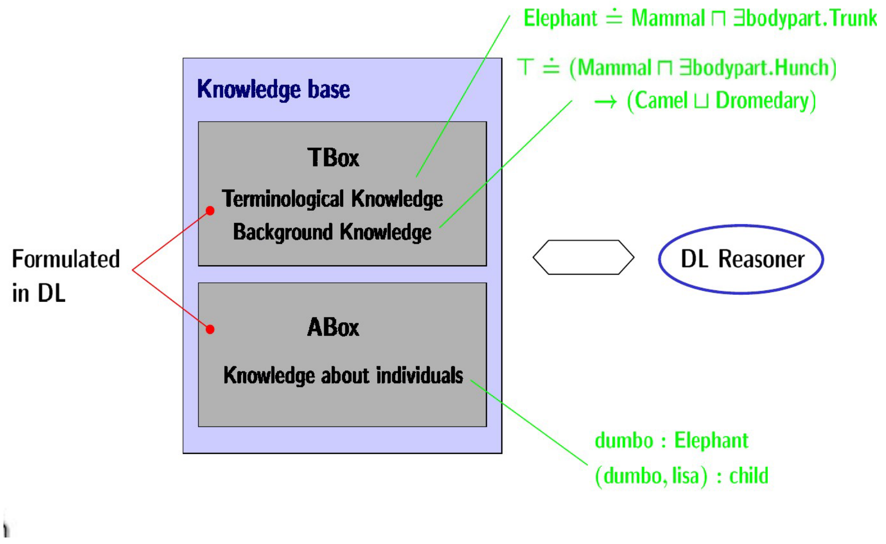
- A **ABox** is a set of data axioms (i.e. ground facts). For example:

$$\{John : HappyParent, JohnHasChildMary\}$$

- A **TBox** is a set of schema axioms (i.e. sentences). For example:

$$\{Doctor \sqsubseteq PersonHappyParent \equiv Person \sqcap \forall HasChild.(Doctor \sqcup \exists HasChild.Doctor)\}$$

- A knowledge base is composed of a TBox and a ABox:



The main reasoning tasks on description logic are the following:

- **Concept satisfiability**, i.e. finding a model for a given concept C.
- **Concept subsumption**, i.e. given two concepts C and D, finding if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ ($C \sqsubseteq D$) for every possible interpretations $\mathcal{I}$. Subsumption can be used to compute a concept hierarchy.