Logics for the Semantic Web

Jos de Bruijn

DERI Technical Report 2006-06-03
June 2006
Logics for the Semantic Web

Jos de Bruijn

Abstract. This chapter introduces a number of formal logical languages which form the backbone of the Semantic Web. They are used for the representation of both ontologies and rules. The basis for all languages presented in this chapter is the classical First-Order Logic. Description Logics is a family of languages which represent subsets of first-order logic. Expressive description logic languages form the basis for popular ontology languages on the Semantic Web. Logic programming is based on a subset of first-order logic, namely Horn Logic, but uses a slightly different semantics and can be extended with non-monotonic negation. Many Semantic Web reasoners are based on logic programming principles and rule languages for the Semantic Web based on logic programming are an ongoing discussion. Frame Logic allows object-oriented style (frame-based) modeling in a logical language. RuleML is an XML-based syntax consisting of different sub-languages for the exchange of specifications in different logical languages over the Web.

Keywords: Formal Logics, Semantic Web, First-Order Logic, Description Logics, Logic Programming, Frame Logic

1Digital Enterprise Research Institute (DERI), Leopold-Franzens Universität Innsbruck, Technikerstraße 21a, A-6020 Innsbruck, Austria. E-mail: jos.debruijn@deri.org.

Copyright © 2006 by the authors
1 Introduction

An important property of the Semantic Web is that the information on it is specified using a formal language in order to enable machine-processability and the derivation of new knowledge from existing knowledge. Logical languages are such formal languages.

Using logical languages for knowledge representation allows one to derive new knowledge which is implicit in existing descriptions. Additionally, the use of formal languages allows to write unambiguous statements and allows machines to derived implicit information using formal rules of deduction associated with the language. Finally, logical languages have been extensively studied in the research areas of Databases and Artificial Intelligence. It is for these reasons that logical languages form the backbone of the Semantic Web.

Logical languages can be used for the representation of different kinds of knowledge, most notably ontologies and rules. In this chapter we describe a number of logical languages which are being used for the representation of knowledge on the Semantic Web.

Classical First-Order Logic (Fitting, 1996) is the basis for all the languages we survey in this chapter. Full first-order logic by itself is a very expressive language. In fact, the language is so expressive that reasoning with the language is in general very hard and the most interesting problems are undecidable. The answer to a question such as “Does sentence $\phi$ follow from theory $\Phi$” can not always be found. For these reasons, several subsets of first-order logic have been investigated and form the basis for several languages which are used on the Semantic Web, most notably description logics and logic programming. Nonetheless, full first-order logic has been proposed as a language for the Semantic Web (Battle et al., 2005; Horrocks et al., 2004; Patel-Schneider, 2005).

Description Logics (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2003) are a family of languages which generally represent strict subsets of first-order logic. Description logics were originally devised to formalize frame-based knowledge representation systems. Languages in this family typically allow the definition of concepts, concept hierarchies, roles and certain restrictions on roles. Description logics receive a lot of attention as a basis for ontology languages on the Semantic Web; most notably, the W3C recommendation OWL is based on an expressive description logic (Horrocks, Patel-Schneider, & Harmelen, 2003).

Logic Programming (Lloyd, 1987) is based on the Horn logic subset of first-order logic, which allows to write rules of the form “if $A$ then $B$”. In order to allow for efficient reasoning, the semantics of logic programming is built around Herbrand interpretation, rather than first-order interpretations. Logic programming is being used as an implementation platform for the Semantic Web, but has also been proposed as the basis for rule and ontology languages on the Semantic Web (Angele et al., 2005; Battle et al., 2005).

Frame Logic (Kifer, Lausen, & Wu, 1995) is an extension of first-order logic which allows an object-oriented (frame-based) style of modeling. Frame logic does not increase the theoretical expressiveness of first-order logic, but allows a more convenient style of modeling. F-Logic programming is a subset of Frame Logic which extends logic programming with frame-based modeling primitives; in this chapter, we will restrict ourselves to this subset. F-Logic programming has been proposed as a basis for ontology and rule languages for the Semantic Web (Kifer, 2005).
In this chapter we describe each of these languages from the point-of-view of knowledge representation, i.e., we describe which kind of knowledge can be described using the language. We also mention certain complexity results for reasoning with these languages, but do not describe the reasoning procedures in detail. Additionally, we describe the RuleML XML syntax for exchange of rules and logical specification in general over the Web.

2 FIRST-ORDER LOGIC

The basic building blocks of first-order logic (FOL) are constants, function symbols and predicates. Constants are interpreted as objects in some abstract domain. Function symbols are interpreted as functions and predicates are interpreted as relations over the domain. The domain may consist of objects representing such things as numbers, persons, cars, etc. The relations may be such things as “greater-than”, “marriage”, “top speed”, etc. Constants, predicates and function symbols are combined with variables and logical connectives to obtain formulas. We want to interpret such formulas as assertions. Whether such an assertion is true of false depends on the context, that is, on the choice of the domain.

In this section we will first define how formulas and theories in a first-order language are created from terms, predicates and a number of logical connectives. We will then define interpretations of first-order formulas and theories, and define when an interpretation is a model of a given formula or theory. The set of models defines the actual meaning, or semantics, of a theory. Using the definition of a model, we define entailment, i.e., the question whether a formula logically follows from a theory. For a more detailed treatment of first-order logic, included methods for automated theorem proving, see (Fitting, 1996).

2.1 Formulas and theories

The signature Σ of a first-order language \( \mathcal{L} \), also referred to as first-order signature, consists of countable sets \( C, F, P \) and \( V \) of constant, function, predicate and variable symbols, respectively. Each function symbol \( f \in F \) and each predicate symbol \( p \in P \) has an associated arity \( n \), which is a non-negative integer.

**Definition 1 (Terms).** We define the set of terms of the language \( \mathcal{L} \) as follows:

- every constant \( c \in C \) is a term in \( \mathcal{L} \),
- every variable \( x \in V \) is a term in \( \mathcal{L} \),
- if \( f \in F \) is an \( n \)-ary function symbol and \( t_1, \ldots, t_n \) are terms in \( \mathcal{L} \), then \( f(t_1, \ldots, t_n) \) is a term in \( \mathcal{L} \).

A ground term is a term with no variables.

**Example 1.** Given the signature \( \Sigma = \langle C, F, P, V \rangle \) with the constants \( C = \{a, b\} \), function symbols \( F = \{f, g\} \), both with arity 1, predicate symbols \( P = \{p, q, r\} \), where \( p \) and \( q \) have arity 2 and \( r \) has the arity 1, and variables \( V = \{x, y, z\} \), then the following are examples of terms: \( x, b, f(a), g(f(a)), g(y) \). Furthermore, \( b, f(a), g(f(a)) \) are examples of ground terms. \( \square \)
An atomic formula is either a predicate expression of the form \( p(t_1,\ldots,t_n) \) where \( p \) is an \( n \)-ary predicate symbol in \( \mathcal{L} \) and \( t_1,\ldots,t_n \) are terms in \( \mathcal{L} \), one of the propositional constants \( \bot,\top \) or \( t_1 = t_2 \), where \( t_1,t_2 \) are terms in \( \mathcal{L} \). A ground atomic formula is an atomic formula with no variables.

**Example 2.** Give the signature \( \Sigma \) as in the previous example, then the following are atomic formulas:

\[
p(a,b), p(x,f(g(y))), q(f(a),b), r(g(f(a))), r(z), a = f(b), f(x) = f(g(y)), \bot
\]

Of these atomic formulas,

\[
p(a,b), q(f(a),b), r(g(f(a))), a = f(b), \bot
\]

are ground atomic formulas.

**Definition 2 (Formulas).** Given the formulas \( \phi,\psi \in \mathcal{L} \), we define the set of formulas in \( \mathcal{L} \) as follows:

- every atomic formula is a formula in \( \mathcal{L} \),
- \( \neg \phi \) is a formula in \( \mathcal{L} \),
- \( (\phi \land \psi) \) is a formula in \( \mathcal{L} \),
- \( (\phi \lor \psi) \) is a formula in \( \mathcal{L} \),
- \( (\phi \rightarrow \psi) \) is a formula in \( \mathcal{L} \),
- given a variable \( x \in V \), \( \exists x.(\phi) \) is a formula in \( \mathcal{L} \),
- given a variable \( x \in V \), \( \forall x.(\phi) \) is a formula in \( \mathcal{L} \).

A variable occurrence is called free if it does not occur in the scope of a quantifier \( (\exists,\forall) \). A formula is open if it has free variable occurrences. A formula is closed if it is not open. A closed formula is also called a sentence of \( \mathcal{L} \).

**Example 3.** Give the signature \( \Sigma \) as before, then the following are sentences of \( \mathcal{L} \):

- \( \exists x.(\forall y.(p(x,y) \land q(f(a),x) \rightarrow r(y))) \)
- \( (p(a,b) \lor \neg r(f(b))) \lor \exists z.(q(z,f(z))) \)

The following is an example of an open formula:

\[ \exists x.(p(x,y)) \rightarrow r(y) \]

**Example 4.** Let's consider the sentences:

- All humans are mortal
• Socrates is a human

This can be written in first-order logic as follows:

\[
\forall x. \text{human}(x) \rightarrow \text{mortal}(x)
\]

\[
\text{human}(\text{socrates})
\]

Intuitively, these sentences can be read as:

• “For all objects it is the case that if they have the property ‘human’, they have the property ‘mortal’.”

• “The object ‘socrates’ has the property ‘human’.”

A first-order language \( \mathcal{L} \) consists of all the formulas which can be written using its signature \( \Sigma \) according to Definition 2. A first-order theory \( \Phi \) of a first-order language \( \mathcal{L} \) is a set of formulas such that \( \Phi \subseteq \mathcal{L} \).

### 2.2 Interpretations, models and entailment

The semantics (or meaning) of a first-order theory is defined by a set of interpretations. In particular, by all interpretations in which the theory is true. Thus, in a sense, the meaning of a theory is constrained by all the interpretations which make the theory true. It follows that a first-order theory does not say what is true in a particular world, or interpretation, but rather limits the number of possible worlds which may be considered. We now give a formal definition.

**Definition 3 (Interpretation).** An interpretation for a language \( \mathcal{L} \) is a tuple \( w = \langle U, I \rangle \), where \( U \) is a nonempty set, called the domain of the interpretation and \( I \) is a mapping which assigns:

- an element \( c^I \in U \) to every constant symbol \( c \in C \),
- a function \( f^I : U^n \rightarrow U \) to every \( n \)-ary function symbol \( f \in F \), and
- a relation \( p^I \subseteq U^n \), to every \( n \)-ary predicate symbol \( p \in P \).

A variable assignment \( B \) is a mapping which assigns an element \( x^B \in U \) to every variable symbol \( x \in V \). A variable assignment \( B' \) is an \( x \)-variant of \( B \) if \( y^B = y^{B'} \) for every variable \( y \in V \) such that \( y \neq x \).

We are now ready to define the interpretation of terms.

**Definition 4.** Given interpretation \( w = \langle U, I \rangle \), variable assignment \( B \), and a term \( t \) of \( \mathcal{L} \), we define \( t^w_B \) as follows:

- for every constant symbol \( c \in C \), \( c^w_B = c^I \),
- for every variable symbol \( x \in V \), \( x^w_B = x^B \),
- if \( t = f(t_1, \ldots, t_n) \), \( t^w_B = f^I(t_1^w_B, \ldots, t_n^w_B) \).

We can see from Definition 4 that, given an interpretation and a variable assignment, each term is interpreted as one object in the domain. We can now define satisfaction (truth) of first-order formulas.
Definition 5 (Satisfaction). Let $w = \langle U, I \rangle$ be an interpretation for $\mathcal{L}$, $B$ a variable assignment, and $\phi \in \mathcal{L}$ a formula. We denote satisfaction of $\phi$ in $w$ ($\phi$ is true in $w$), given the variable assignment $B$, with $w \models_B \phi$. Satisfaction is recursively defined as follows, with $\psi, \psi_1, \psi_2$ formulas, $p$ an $n$-ary predicate symbol and $t_1, \ldots, t_n$ terms:

- $w \models_B p(t_1, \ldots, t_n)$ iff $(t_1^w, \ldots, t_n^w) \in p^I$.
- $w \models_B \top$ and $w \not\models_B \bot$.
- $w \models_B t_1 = t_2$ iff $t_1^w = t_2^w$.
- $w \models_B \neg \psi$ iff $w \not\models_B \psi$.
- $w \models_B \psi_1 \land \psi_2$ iff $w \models_B \psi_1$ and $w \models_B \psi_2$.
- $w \models_B \psi_1 \lor \psi_2$ iff $w \models_B \psi_1$ or $w \models_B \psi_2$.
- $w \models_B \psi_1 \rightarrow \psi_2$ iff whenever $w \models_B \psi_1$, $w \models_B \psi_2$.
- $w \models_B \exists x.\psi$ iff for some $x$-variant $B'$ of $B$, $w \models_B \psi$.
- $w \models_B \forall x.\psi$ iff for every $x$-variant $B'$ of $B$, $w \models_B \psi$.

A formula $\phi$ is satisfied by an interpretation $w$, or $\phi$ is true in $w$, written as $w \models \phi$, if $w \models_B \phi$ for all variable assignments $B$. We that say $w$ is a model of $\phi$ if $w \models \phi$.

If a formula has at least one model, we call the formula satisfiable; conversely, if a formula has no models, it is unsatisfiable. We say that a formula $\phi$ is valid if $\phi$ is true in every interpretation $w$ of $\mathcal{L}$. An interpretation $w$ is a model of a theory $\Phi \subseteq \mathcal{L}$ if $w \models \phi$ for every formula $\phi \in \Phi$.

Example 5. Consider the first-order language $\mathcal{L}$ with the constant symbol $a$, the unary function symbol $f$, the binary function symbol $g$, and the binary predicate $p$. Now consider the following theory:

$$\begin{align*}
\forall x. (g(a, x) &= x) \\
\forall x. \forall y. (g(x, y) = g(y, x)) \\
\forall x. \forall y. \forall z. (g(x, g(y, z)) &= g(g(x, y), z)) \\
\forall x. (p(x, x)) \\
\forall x. (p(x, f(x))) \\
\forall x. \forall y. \forall z. (p(x, y) \land p(y, z) &\rightarrow p(x, z))
\end{align*}$$

Now consider the interpretation $w = \langle \mathbb{N}, I \rangle$, with the domain of the natural numbers (including 0) and $I$ assigns to $a$ the number 0 (zero), to $f$ the successor function, i.e., for every natural number $x$, $f(x) = x + 1$, and to $g$ the addition operator, i.e., for every pair of natural numbers $x, y$, $g(x, y) = x + y$. Finally, $I$ assigns to the predicate $p$ the relation $\leq$ (smaller-or-equal).

Now, the first formula is interpreted as “$0 + x = x$”, the second formula as “$x + y = y + x$” and the third formula as “$x + (y + z) = (x + y) + z$”. The fourth formula is interpreted as “$x \leq x$” (i.e., reflexivity of $\leq$), the fifth as “$x \leq x + 1$” and the sixth as “if $x \leq y$ and $y \leq z$ then $x \leq z$” (i.e., transitivity of $\leq$). All these statements are obviously true for the domain of natural numbers, thus the theory is true in $w$ and $w$ is a model for this theory. □
The theory of the previous example is satisfiable; the interpretation constructed in the example is a model. It is not valid; one can easily construct an interpretation which is not a model, e.g., any interpretation which assigns to \( p \) an anti-reflexive relation.

An example of a valid formula is:

\[
\forall x. (p(x) \lor \neg p(x))
\]

It is easy to verify that in every interpretation \( p(x) \) must either be true or false for every \( x \), and thus the formula is true in every possible interpretation. The following formula is unsatisfiable:

\[
\exists x. (p(x) \land \neg p(x))
\]

It is easy to verify that in every interpretation \( p(x) \) must either be true or false for every \( x \); it cannot be both. Therefore, \( p(x) \land \neg p(x) \) cannot be true for any \( x \) in any interpretation.

**Definition 6** (Entailment). We say that a theory \( \Phi \subseteq \mathcal{L} \) entails a formula \( \phi \in \mathcal{L} \), denoted \( \Phi \models \phi \), iff for all models \( w \) of \( \Phi \), \( w \models \phi \).

We can reformulate this definition of entailment using sets of models. Let \( \text{Mod}(\Phi) \) denote the set of models of some first-order theory \( \Phi \), then we can reformulate entailment as set inclusion: \( \text{Mod}(\Phi) \subseteq \text{Mod}(\phi) \) iff \( \Phi \models \phi \). In a sense, the entailing theory is more specific, i.e., allows fewer models, than the entailed theory.

We have characterized a first-order theory \( \Phi \) using its set of models \( \text{Mod}(\Phi) \). Another way of characterizing a first-order theory \( \Phi \) is using its set of entailments \( \text{Ent}(\Phi) \). The set of entailments of \( \Phi \) is the set of all formulas which are entailed by \( \Phi \): \( \phi \in \text{Ent}(\Phi) \) iff \( \Phi \models \phi \). Now, the less specific a theory is, the more models it has, but the fewer entailments it has. We can observe that a theory \( \Phi \) entails a theory \( \Psi \), that the set of entailments of \( \Phi \) is a superset of the entailments of \( \Psi \): \( \text{Ent}(\Phi) \supseteq \text{Ent}(\Psi) \) iff \( \Phi \models \Psi \).

**Example 6.** Given the sentences \( p \land q \) and \( q \), where \( p, q \) are null-ary predicate symbols, then clearly:

\[
p \land q \models q
\]

Because in all models where both \( p \) and \( q \) are true, \( q \) must be true.

But not the other way around:

\[
q \not\models p \land q
\]

Because there are models of \( q \) in which \( p \) is not true.

In the example, \( p \land q \) presents a more constrained view of the world than \( q \), namely both \( p \) and \( q \) must be true, whereas \( q \) only mandates that \( q \) must be true, but does not say anything about \( p \). Thus, the set of models of \( p \land q \), \( \text{Mod}(p \land q) \), is a subset of the set of models of \( q \): \( \text{Mod}(p \land q) \subseteq \text{Mod}(q) \), because every model of \( p \land q \) is a model of \( q \). It is also easy to see that the set of entailments of \( p \land q \), \( \text{Ent}(p \land q) \), is larger than the set of entailments of \( q \): \( \text{Ent}(p \land q) \supset \text{Ent}(p) \). For example, \( p \land q \) entails \( p \land q \), whereas \( q \) does not.
It turns out that checking entailment in first-order logic can be reduced to checking satisfiability. In order to check whether some formula $\phi$ is entailed by some theory $\Phi$: 

$$\Phi \models \phi$$

We can simply add the negation of $\phi$ to $\Phi$ and check whether this combination, $\Phi \cup \neg(\phi)$ is satisfiable, i.e., has a model. If $\Phi \cup \neg(\phi)$ is not satisfiable, then the entailment holds.

**Example 7.** Consider the entailment question from the previous example:

$$p \land q \models q$$

We have concluded earlier that this entailment must hold, because every model of $p \land q$ must be a model of $q$. Now, let’s rewrite this entailment problem to an unsatisfiability problem, i.e., we want to check whether:

$$\{p \land q, \neg q\}$$

has a model. Clearly, this formula cannot have a model, because both $q$ and $\neg q$ would have to be true in this model and this is not possible. Therefore, we can conclude that $p \land q$ entails $q$. □

We can now try to explain intuitively why we can use unsatisfiability of $\Phi \cup \neg(\phi)$ to check entailment $\Phi \models \phi$. We have seen earlier that $\Phi \models \phi$ if and only if $\text{Mod}(\Phi) \subseteq \text{Mod}(\phi)$. We know that the sets of models of $\phi$ and $\neg \phi$ are disjoint, because there can be no model in which both $\phi$ and $\neg \phi$ are true.

Now, if $\Phi \cup \neg(\phi)$ would be satisfiable, then there would be one interpretation in which both $\Phi$ and $\neg \phi$ are true. This means, by disjointness of $\text{Mod}(\phi)$ and $\text{Mod}(\neg(\phi))$, that $\Phi$ has a model which is not a model of $\text{Mod}(\phi)$, which means $\text{Mod}(\Phi) \nsubseteq \text{Mod}(\phi)$ and thus $\Phi$ does not entail $\phi$.

The satisfiability problem for first-order logic is the problem to decide whether there is a model for a given first-order theory $\Phi$. In other words: “Does $\Phi$ have a model?”

It turns out that this question is not so easily answered and in some cases it is even impossible to find an answer. This makes the satisfiability problem for first-order logic undecidable, i.e., the question of satisfiability cannot always be answered in a finite amount of time. However, it does turn out that if the answer to the satisfiability question is “yes”, the answer can always be found in a finite amount of time. Therefore, first-order logic is actually *semi-decidable*. It is possible to enumerate all sentences which are entailed by a first-order theory.

### 3. DESCRIPTION LOGICS

Description logics ([Baader et al., 2003](#)) (formerly called *Terminological Logics*) are a family of knowledge representation languages, which revolve mainly around concepts, roles (which denote relationships between concepts), and role restrictions. Description logics are actually based on first-order logic. Therefore, concepts can be seen as unary predicates, whereas roles can be seen as binary predicates. Although there are also Description Logic languages which allow n-ary roles, we will not discuss these here.

In this section we will illustrate description logics through the relatively simple description logic $\mathcal{ALC}$ (*Attributive Language with Complement*) which allows concepts, concept hierarchies, role
restrictions and the boolean combination of concept descriptions. The currently popular expressive description logics, such as SHIQ and SHOIQ, are all extensions of ALC.

3.1 The Basic Description Logic ALC

An ALC knowledge base has two parts: the TBox, with terminological knowledge, which consists of a number of class definitions, and the ABox, which consists of assertions about actual individuals.

Concept axioms in the TBox are of the form $C \sqsubseteq D$ (meaning the extension of $C$ is a subset of the extension of $D$; $D$ is more general than $C$) or $C \equiv D$ (where $C \equiv D$ is interpreted as $C \sqsubseteq D$ and $D \sqsubseteq C$) with $C$ and $D$ (possibly complex) descriptions. Descriptions can be built from named concepts (e.g., $A$) and role restrictions (e.g., $\forall R.C$ denotes a universal value restriction), connected with negation ($\neg$), union ($\sqcup$) and intersection ($\sqcap$).

Traditionally, descriptions are interpreted as sets, where the different parts of a description constrain the set. Take, for example, the description $A \sqcap \neg B \sqcap \exists R.C$. This can be read as “all elements which are member of the set $A$, but not of the set $B$ ($\neg B$); additionally, each member must be related to some member of the set $C$ via the relation $R$ ($\exists R.C$)

Descriptions can also be understood as formulas of first-order logic with one free variable. For example, the description $A \sqcap \neg B \sqcap \exists R.C$ corresponds to the formula $A(x) \land \neg B(x) \land \exists y.(R(x, y) \land C(y))$. The correspondence between descriptions and first-order formulas is given in Table 1. In the table, $\pi$ is a function which takes as parameters a description and a variable and returns a first-order formula.

<table>
<thead>
<tr>
<th>Description</th>
<th>First-Order Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(A, X)$</td>
<td>$A(X)$</td>
</tr>
<tr>
<td>$\pi(\top, X)$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\pi(\bot, X)$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\pi(C \sqcap D, X)$</td>
<td>$\pi(C, X) \land \pi(D, X)$</td>
</tr>
<tr>
<td>$\pi(C \sqcup D, X)$</td>
<td>$\pi(C, X) \lor \pi(D, X)$</td>
</tr>
<tr>
<td>$\pi(\neg C, X)$</td>
<td>$\neg(\pi(C, X))$</td>
</tr>
<tr>
<td>$\pi(\exists R.C, X)$</td>
<td>$\exists y.(R(X, y) \land \pi(C, y))$</td>
</tr>
<tr>
<td>$\pi(\forall R.C, X)$</td>
<td>$\forall y.(R(X, y) \rightarrow \pi(C, y))$</td>
</tr>
</tbody>
</table>

Table 1: Correspondence between descriptions and first-order formulas

The TBox of a Description Logic knowledge base consists of a number of axioms of the forms $C \sqsubseteq D$ and $C \equiv D$, where $C$ and $D$ are descriptions. In the set-based interpretation, $C \sqsubseteq D$ means that $C$ is interpreted as a subset of $D$ and $C \equiv D$ means that $C$ and $D$ are interpreted as the same set. We give the translation to first-order logic in table 2.
Description Logics have been devised to formally model terminologies. The two major benefits of the formal modeling of a terminology are that (1) it is possible to verify the consistency of the specification and (2) it is possible to automatically infer information which is hidden in the terminology. We illustrate both in the following example.

**Example 8.** Consider the following TBox $T$:

- $\text{Person} \equiv \forall \text{hasChild.} \text{Person} \sqcap \exists \text{hasFather.} \text{Father} \sqcap \exists \text{hasMother.} \text{Mother}$
- $\text{Person} \equiv \text{Man} \sqcup \text{Woman}$
- $\text{Parent} \equiv \exists \text{hasChild.} \top$
- $\text{Mother} \equiv \text{Woman} \sqcap \text{Parent}$
- $\text{Father} \equiv \text{Man} \sqcap \text{Parent}$

$T$ has two alternative definitions of the concept $\text{Person}$: (1) everyone who has a father which is a father, has a mother which is a mother and has only children which are persons (notice that this does not require a person to have children) and (2) the union of the sets of all men and women; note that both definitions must be valid. A parent is a person who has a child. A mother is a woman who is also parent and a father is a man who is also a parent.

The TBox $T$ does not contain any inconsistencies. We can infer some information from $T$. For example: $\text{Man}$ is subsumed by (is a more specific concept than) $\text{Person}$: $\text{Man} \sqsubseteq \text{Person}$.

We can see from $T$ that every $\text{Man}$ must be a $\text{Person}$. Therefore, it would be inconsistent to state that there is any $\text{Man}$ who is not a $\text{Person}$. We can add the following axiom to $T$:

$$\text{ManNotPerson} \equiv \text{Man} \sqcap \neg \text{Person}$$

A concept in a TBox is inconsistent if it is impossible for this concept to have any instances, i.e., the concept can only be interpreted as the empty set. In order to verify this, we translate the second and the last axiom of $T$ to first-order logic to obtain the theory $\pi(T)$:

- $\forall x. (\text{Person}(x) \rightarrow (\text{Man}(x) \lor \text{Woman}(x)))$
- $\forall x. ((\text{Man}(x) \lor \text{Woman}(x)) \rightarrow \text{Person}(x))$
- $\forall x. (\text{ManNotPerson}(x) \rightarrow (\text{Man}(x) \land \neg \text{Person}(x)))$
- $\forall x. ((\text{Man}(x) \land \neg \text{Person}(x)) \rightarrow \text{ManNotPerson}(x))$

The second formula says that every man is a person, whereas the third formula says that every $\text{ManNotPerson}$ is a man and not a person. This would be impossible, because by the second formula every man is necessarily a person. What follows is that every model of this theory interprets $\text{ManNotPerson}$ as the empty set.

If we were to add the formula $\text{ManNotPerson}(a)$ to $\pi(T)$, for some constant $a$, then the theory no longer has any model and is thus inconsistent.

Besides the concepts and role descriptions, which comprise the TBox, a Description Logic Knowledge base typically also contains individuals (instances) and relations between individuals.
and, in the case of OWL, equality and inequality assertions between individuals. These assertions about individuals comprise the ABox.

The assertions in the ABox are of the form \( i \in C \), where \( i \) is an individual and \( A \) is an atomic concept, or of the form \( \langle i_1, i_2 \rangle \in R \), where \( i_1 \) and \( i_2 \) are individuals and \( R \) is a (binary) role. Assertions of the form \( i \in A \) are interpreted as set membership and assertions of the form \( \langle i_1, i_2 \rangle \in R \) are interpreted as tuples in a relation. The translation to first-order logic can be found in Table 3.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>First-order Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(i \in A) )</td>
<td>( A(i) )</td>
</tr>
<tr>
<td>( \pi(\langle i_1, i_2 \rangle \in R) )</td>
<td>( R(i_1, i_2) )</td>
</tr>
</tbody>
</table>

Table 3: Correspondence between ABox assertions and first-order formulas

Example 9. The following ABox \( A \) is an ABox for the TBox \( T \) of Example 8:

- \( john \in \text{Person} \)
- \( \langle john, mary \rangle \in \text{hasChild} \)
- \( john \in \text{Man} \)

From this ABox \( A \), together with the TBox \( T \), we can derive a number of facts: \( mary \) is a person, since \( john \) is a person, \( mary \) is a child of \( john \), and all children of persons are persons; \( john \) is a parent, since he has a child; finally, \( john \) is a father, because he is both a parent and a man.

3.2 Reasoning in Description Logics

We are here only concerned with reasoning with description logic TBoxes; note, however, that reasoning with ABoxes can generally be reduced to TBox-reasoning (Baader et al., 2003, Section 2.3). As mentioned above, there are two main motivations for using description logics: (1) detecting inconsistencies in descriptions and (2) deriving implicit information from descriptions. Both detecting inconsistency and deriving implicit information require reasoning over the terminology. (1) and (2) can be reduced to the following reasoning tasks:

Satisfiability A concept \( C \) is satisfiable with respect to a TBox \( T \) if there exists at least one model of \( T \) where the interpretation of \( C \), \( C^I \), is non-empty.

Subsumption A concept \( C \) is subsumed by a concept \( D \) with respect to \( T \) iff \( C^I \subseteq D^I \) for every model of \( T \). This can also be written as \( C \sqsubseteq_T D \).

Both reasoning tasks can be reduced to reasoning problems in first-order logic. A concept \( C \) is satisfiable with respect to a TBox \( T \) if and only if \( \pi(T) \cup \{ C(a) \} \) is satisfiable. Essentially, we translate the TBox to first-order logic and add an instance \( a \) of the concept \( C \) and check whether the resulting theory is consistent, i.e., satisfiable. Notice that we have already applied this technique in Example 8.

Subsumption can be reduced to entailment in first-order logic: \( C \sqsubseteq_T D \) if and only if \( \pi(T) \models \pi(C \sqsubseteq D) \). Thus, we translate the TBox to a first-order theory and translate the subsumption axiom which we want to check for a first-order sentence and check whether the one entails the
other. We have already seen that entailment in first-order logic can be reduced to satisfiability checking. Similarly, subsumption can be reduced to satisfiability, namely, \( C \) is subsumed by \( D \) with respect to TBox \( T \) if and only if \( C \sqcap \neg D \) is not satisfiable with respect to \( T \).

Reasoning in most description logics is decidable and there exist optimized algorithms for reasoning with certain description logics.

4 LOGIC PROGRAMMING

Logic Programming is based on a subset of First-Order Logic, called Horn Logic. However, the semantics of Logic Programming is slightly different from First-Order logic. The semantics of logic programs is based on minimal Herbrand models (Lloyd 1987), rather than first-order models.

A logic program consists of rules of the form “if \( A \) then \( B \)”. Intuitively, if \( A \) is true, then \( B \) must be true. Logic Programming plays two major roles on the Semantic Web. On the one hand, it is used to reason with RDF (Klyne & Carroll 2004), RDF Schema (Brickley & Guha 2004) and parts of OWL (Dean & Schreiber 2004). On the other hand, it used is to represent knowledge on the Semantic Web in the form of rules.

Euler\(^1\) and CWM\(^2\) are examples of reasoners for the Semantic Web, based on logic programming. Euler and CWM both work directly with RDF data and can be used to derive new RDF data using rules.

Rules can be seen as a knowledge representation paradigm complementary to Description Logics. Description Logics are very convenient for defining classes, class hierarchies, properties and the relationships between them. More specifically, compared with logic programming, description logics have the following expressive power: existential quantification, disjunction and classical negation. Logic programs, on the other hand, have the following additional expressive power: predicates with arbitrary arities, chaining variables over predicates (there are no restrictions on the use of variables), and the use of nonmonotonic negation. An often quoted example which illustrates the expressive power of rules compared with ontologies is: “if \( x \) is the brother of a parent of \( y \), then \( x \) is an uncle of \( y \)”. This example cannot be expressed using Description Logics, because the variables \( x \) and \( y \) are both used on both sides of the implication, which is not possible in Description Logics.

In this section we will first explain the general syntax of logic programs. After that, we will explain the semantics of logic programs, based on minimal Herbrand models. An important result in the area of logic programming is the equivalence of both semantics. We then introduce default negation in logic programs and show how it differs from negation in classical first-order logics.

4.1 Logic Programs

Classical Logic Programming makes use of the Horn logic fragment of First-Order Logic. A First-Order formula is in the Horn fragment, if it is a disjunction of literals with at most one positive literal, in which all variables are universally quantified:

\[
(\forall) h \lor \neg b_1 \lor \ldots \lor \neg b_n
\]  

\(1\)

\(^1\)http://www.agfa.com/w3c/euler/
\(^2\)http://www.w3.org/2000/10/swap/doc/cwm.html
This formula can be rewritten to the following form:

\[ (\forall) h \leftarrow b_1 \land \ldots \land b_n \] (2)

Such a formula is also called a Horn formula. A Horn formula with one positive literal, and at least one negative literal is called a rule. The positive literal \( h \) is called the head of the rule. The conjunction of negative literals \( b_1 \land \ldots \land b_n \) is called the body of the rule. A rule without a body is called a fact and a rule without a head is called a query. A logic program consists of a set of horn clauses.

In this section we use a slightly different notation for rules, which diverges from the usual first-order syntax. A rule is written as follows:

\[ h_1 :\neg b_1, \ldots, b_n. \]

A fact is written as:

\[ h_1. \]

A query is written as:

\[ ?- b_1, \ldots, b_n. \]

A positive logic program \( P \) is a collection of rules and facts.

4.2 Minimal Herbrand Model Semantics

We will first define the minimal Herbrand model semantics for positive logic programs \( P \).

**Definition 7** (Herbrand universe). The Herbrand universe \( HU \) of \( P \) is the set of all ground terms which can be formed using the constant and function symbols in the signature \( \Sigma \) of \( P \) (in case \( P \) has no constants, we add some constant \( c \)).

**Definition 8** (Herbrand base and Herbrand interpretation). The Herbrand base \( HB \) of \( P \) is the set of all ground atomic formulas which can be formed with the predicate symbols in \( \Sigma \) of \( P \) and the terms in \( HU \), i.e., all formulas of the form:

\[ p(t_1, \ldots, t_n) \]

With \( p \) an \( n \)-ary predicate symbol and \( t_1, \ldots, t_n \in HU \). A Herbrand interpretation of \( P \) is a subset of \( HB \).

**Example 10.** Consider the logic program \( P \):

\[ p(a). \]
\[ q(b). \]
\[ p(X) :\neg q(X). \]
\[ p(X) :\neg p(f(X)). \]
This Herbrand universe \( HU \) of \( P \) consists of all ground terms which can be constructed out of the constant and function symbols which occur in \( P \). Therefore, \( HU = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\} \).

The Herbrand base of \( P \), \( HB \), consists of all ground atomic formulas which can be formed using the predicate symbols of \( P \) combined with the Herbrand universe \( HU \):

\[
HB = \{p(a), p(b), q(a), q(b), p(f(a)), q(f(a)), p(f(b)), q(f(b)), p(f(f(a))), \ldots\}
\]

Examples of Herbrand interpretations are \( w_1 = \{p(f(a)), q(b), q(f(b))\} \), \( w_2 = \{p(a), p(b), q(b)\} \), and \( w_3 = \{p(a), p(b), q(a), q(b), p(f(a))\} \).

In this example, the Herbrand universe is infinite. In fact, as soon as function symbols are used in the logic program, the Herbrand universe becomes infinite. An infinite Herbrand universe means also an infinite Herbrand base.

Note that a Herbrand interpretation \( w^H \) of \( P \) corresponds to a first-order interpretation \( w = \langle HU, I \rangle \) where \( HU \) is the Herbrand universe, and \( I \) satisfies the following conditions:

1. \( c^I = c \) for every constant symbol \( c \in C \),
2. \( (f(t_1, \ldots, t_n))^w = f(t_1, \ldots, t_n) \) for every \( n \)-ary function symbol \( f \in F \) and ground terms \( t_1, \ldots, t_n \), and
3. \( (p(t_1, \ldots, t_n))^w \) for \( p(t_1, \ldots, t_n) \in w^H \).

The grounding of a logic program \( P \), denoted \( \text{Ground}(P) \), is the union of all possible ground instantiations of \( P \). A ground instantiation of a logic program is obtained by, for each rule \( r \in P \), replacing each variable with a term in the Herbrand Universe \( HU \).

The definition of Herbrand models is as follows:

**Definition 9 (Herbrand model).** Let \( P \) be a positive logic program. A Herbrand interpretation \( w \) of \( P \) is a model of \( P \) if for every rule \( r \in \text{Ground}(P) \) the following condition holds:

- If \( b_1, \ldots, b_n \in w \) then \( h \in w \)

The intersection of all Herbrand models of \( P \) is also a model of \( P \) and is called the minimal Herbrand model. Therefore, each logic program has a minimal Herbrand model and this model is unique.

**Example 11.** Among the three interpretations in the previous example, \( w_1, w_2, w_3 \), only \( w_2 \) and \( w_3 \) are Herbrand models of \( P \). Furthermore, \( w_2 \) is the minimal model of \( P \). □

**Definition 10 (Ground entailment).** A logic program \( P \) entails a ground atomic formula \( A \), denoted \( P \models A \), if and only if \( A \) is included in the minimal Herbrand model of \( P \).

We say a conjunction of ground formulas \( A_1 \land \ldots \land A_n \) is entailed by a program \( P \), denoted \( P \models A_1 \land \ldots \land A_n \), if and only if each of the ground formulas in the conjunction is entailed by \( P \):

\[
P \models A_i \text{ for } 1 \leq i \leq n.
\]

By this definition of entailment it is only possible to check whether particular facts follow from a logic program, but not whether some rule or formula follows from the program. However, it turns out that this definition of entailment is sufficient for the most prominent reasoning task in logic programming: query answering.

A query \( q \) is a rule without a body, i.e., a conjunction of atomic formulas, with a number of free variables. We write a query as:
A variable substitution is an assignment of objects in the Herbrand universe to variables. A variable substitution is written as \([x_1/a_1, ..., x_n/a_n]\) and applying a variable substitution to a formula yields a new formula in which variables are replaced with terms as specified in the substitution.

**Example 12.** A number of variable substitutions:

- \(p(x)[x/a] = p(a)\)
- \((p(x, y) \land q(x) \land q(z))[x/f(a), y/a] = p(f(a), a) \land q(f(a)) \land q(z)\)
- \((q(y) \land r(z, f(y)))[y/g(b), z/a] = q(g(b)) \land r(a, f(g(b)))\)

An *answer* to a query \(q\) for a program \(P\) is a variable substitution \([x_1/a_1, ..., x_n/a_n]\) for all variables in \(q\) such that \(q[x_1/a_1, ..., x_n/a_n]\) is entailed by \(P\).

**Example 13.** Recall the program \(P\) from Example 10. Now, consider the query \(q:\)

\[- p(Y).\]

Now, \([Y/a]\) and \([Y/b]\) are answers to this query. In fact these are the only answers to the query. This can be easily verified by considering the minimal Herbrand model of \(P\).

### 4.3 Recursion in Logic Programs

Interest in the use of logic for databases has given rise to the field of *deductive databases*. Datalog (Ullman, 1988) is the most prominent language for deductive databases. Datalog can be seen as an expressive query language for relational databases, based on Logic Programming. Compared with the relational query language SQL, Datalog allows to specify recursive queries. Datalog can be seen as a Logic Programming without the use of function symbols.

**Example 14.** Given the following logic program:

\[
\begin{align*}
\text{parent}(john, mary). \quad & \\
\text{brother}(mary, jack). \quad & \\
\text{uncle}(X, Z) & \leftarrow \text{parent}(X, Y), \text{brother}(Y, Z).
\end{align*}
\]

This logic program consists of two parts, namely (a) the facts that *mary* is a parent of *john* and *jack* is a brother of *mary* and (b) the “uncle” rule which states that the brother of someone’s parent is that person’s uncle.

In order to test whether there is recursion in a logic program, one can build the *dependency graph* of the logic program. The dependency graph is a directed graph where the predicates in the logic program are represented by nodes in the graph. There is an arc from some predicate \(p\) to some predicate \(q\) if they occur in a rule with \(p\) in the body and \(q\) in the head. A logic program is *recursive* if and only if there is a cycle in the dependency graph.

\(^{3}\)Note that a recent version of SQL, namely SQL:99, allows a limited form of recursion in queries.
Example 15. Given the following logic program \( P \):

\[
\text{mother(mary,jill).} \\
\text{father(john,jill).} \\
\text{parent(jack,john).} \\
\text{parent(X,Y) :- mother(X,Y).} \\
\text{parent(X,Y) :- father(X,Y).} \\
\text{ancestor(X,Y) :- parent(X,Y).} \\
\text{ancestor(X,Z) :- ancestor(X,Y), ancestor(Y,Z).}
\]

The \textit{ancestor} relation is defined as the transitive closure of the \textit{parent} relation. From this, we can already see that the logic program is recursive, because \textit{ancestor} depends on itself. We can also verify this in the dependency graph of \( P \), depicted in Figure 1.

\[ \]

| Figure 1: Dependency graph of the ancestor program |

4.4 Negation in Logic Programs

An often used extension of logic programs is \textit{negation}. In this section we explain the basics of negation in logic programming. As the treatment is rather involved, the reader may skip this section on first reading.

A normal logic program \( P \) consists of a number of rules. Each rule is of the form:

\[
h :- b_1, \ldots, b_k, \neg n_1, \ldots, \neg n_l.
\]

where \( h \) is an atomic formula (as defined in the previous section), also called the \textit{head} of the rule, and \( b_1, \ldots, b_k, n_1, \ldots, n_l \) are atomic formulas, also called \textit{atoms}, and \( b_1, \ldots, b_k, \neg n_1, \ldots, \neg n_l \) is called the \textit{body} of the rule. \( b_1, \ldots, b_k \) are said to occur \textit{positively} and \( n_1, \ldots, n_l \) are said to occur \textit{negatively} in the body of the rule; \( b_1, \ldots, b_k \) are \textit{positive literals} and \( \neg n_1, \ldots, \neg n_l \) are \textit{negative literals}. A positive rule is a rule which has no negative literals in the body. A positive program is a program which consists only of positive rules. The signature of a logic program \( P \) is the first-order signature \( \Sigma_P \) where the constant, function, predicate, and variable symbols are exactly those which occur in \( P \).

The difference with the logic programs we have discussed above is that normal logic programs allow negation in the body of a rule. Note that the negation in logic programs, denoted with \( \neg \) differs from negation in classical first-order logic, which is denoted with \( \neg \). The negation in logic programs is also called \textit{default negation} because all facts are assumed to be false, unless we can infer otherwise. The default negation of some atomic formula \( \alpha \), denoted \( \neg \alpha \) is true, if \( \alpha \) cannot
be derived, whereas in first-order logic, the classical negation of $\alpha$, denoted $\neg \alpha$, is only true if it can be explicitly derived.$^4$

We can straightforwardly extend the Herbrand semantics to programs with negation. However, it turns out that there may be several minimal models.

**Definition 11 (Herbrand model).** Let $P$ be a normal logic program. A Herbrand interpretation $w$ of $P$ is a model of $P$ if for every rule $r \in \text{Ground}(P)$, $h \in w$:

- if all positive body literals are in $w$: $b_1, ..., b_k \in w$, and
- all negative body literals are not in $w$: $n_1, ..., n_l \notin w$.

**Example 16.** Consider the logic program $P$:

$$
\begin{align*}
p(a) & : - \neg p(b) . \\
p(b) & : - \neg p(a) .
\end{align*}
$$

This program has two minimal models: $\{p(a)\}$ and $\{p(b)\}$. \qed

The existence of multiple minimal models increases the complexity of finding a minimal model. A class of logic programs with negations which have a single minimal model which can be straightforwardly computed is the class of stratified logic programs. The predicates in a stratified program can be divided into a number of strata such that there is no negative dependency between predicates in the same stratum. A stratification has to fulfill two conditions:

1. if some predicate $q$ is at stratum $i$ and depends positively on some predicate $p$, then $p$ must be in a stratum $j$ such that $j \leq i$, and
2. if some predicate $q$ is at stratum $i$ and depends negatively on some predicate $p$, then $p$ must be in a stratum $j$ such that $j < i$.

In order to check whether a program is stratifiable, we can check the dependency graph. This graph is build in a similar way as discussed above, with the addition that if there is a rule with head $q$ and with a negative body literal $\neg p$, then there is an arc between $p$ and $q$ and this arc is marked with “not”. If there are cycles in the graph which include a negative arc, then the program is not stratifiable.

**Example 17.** Consider the logic program:

$$
\begin{align*}
p & : - q . \\
q & : - \neg r, p . \\
r & : - q .
\end{align*}
$$

This program is not stratifiable; this can be easily seen by constructing the dependency graph. Clearly, the second rule requires $r$ to be in a lower stratum then $q$, whereas the third rule requires $r$ to be in a higher (or the same) stratum than $q$.

Now consider the following logic program:

---

$^4$Note that there are extensions of logic programming which deal with classical negation (Gelfond & Lifschitz, 1991), but we will not discuss these here.
This program is stratifiable with the following stratification:

- stratum 0: \{r\}
- stratum 1: \{p, q\}

Now consider the following logic program:

\begin{align*}
\text{p} & : - q. \\
\text{p} & : - r. \\
\text{q} & : - \text{not } r. \\
\text{r} & : - s. \\
\text{t} & : - \text{not } q.
\end{align*}

This program has the following stratification:

- stratum 0: \{r, s\}
- stratum 1: \{p, q\}
- stratum 2: \{t\}

It is now straightforward to evaluate a stratified logic program by first computing the predicates in the lowest stratum and then working up stratum-by-stratum until the highest stratum is reached and the minimal model is computed. It turns out that each stratified program has a single minimal Herbrand model which is the intersection of all Herbrand models of the program.

There are different semantics for logic programs which are not stratifiable. The most popular ones are the Stable Model Semantics (Gelfond & Lifschitz, 1988) and the Well-Founded Semantics (Gelder, Ross, & Schlipf, 1991). These semantics are beyond the scope of this chapter.

5 FRAME LOGIC

Frame Logic (Kifer, 2005; Kifer et al., 1995) (F-Logic) is an extension of first-order logic which adds explicit support for Object-Oriented modeling. It is possible to explicitly specify methods, as well as generalization/specialization and instantiation relationships. The syntax of F-Logic has some seemingly higher-order features, for example, the same identifier can be used for both a class and an instance. However, the semantics of F-Logic is strictly first-order.

Although F-Logic was originally defined as an extension to full First-Order Logic, the original paper (Kifer et al., 1995) already defined a Logic Programming-style semantics for the subset of F-Logic based on Horn logic. Intuitively, the Horn subset is obtained in the usual way with the addition that, beside predicate symbols with arguments, F-Logic molecules can also be seen as atomic formulas. In the remainder, we will refer to the Horn subset of F-Logic with logic programming semantics as F-Logic programming. There exist several implementations of F-Logic programming, most notably (Yang, Kifer, & Zhao, 2003; Decker, Erdmann, Fensel, & Studer, 1999). Since most attention around F-Logic is around F-Logic Programming, we will restrict ourselves to the Logic Programming semantics for F-Logic and disregard the FOL semantics.
5.1 F-Logic programs

To simplify matters, we focus only on a subset of F-Logic. We do not consider parametrised methods, functional (single-valued) methods and we consider only non-inheritable methods. We also do not consider compound molecules.

Formally, an F-Logic theory is a set of formulas constructed from atomic formulas, as defined for first-order logic, and so called molecules. Let $\Sigma$ be a first-order signature, as defined before and let $T$ be the set of terms which can be constructed from the constants, function symbols, and variables in $\Sigma$.

Definition 12 (Molecule). A molecule in F-Logic is one of the following statements:

1. an is-a assertion of the form $C:D$ where $C, D \in T$,
2. a subclass-of assertion of the form $C::D$ where $C, D \in T$,
3. a data molecule of the form $C[D \rightarrow> E]$ where $C, D, E \in T$, or
4. a signature molecule of the form $C[D =>> E]$ where $C, D, E \in T$.

An F-Logic molecule is called ground, if it contains no variables.

An is-a assertion $C:D$ states that $C$ is an instance of the class $D$; a sub-class assertion $C::D$ states that $C$ is a subclass of $D$. Data molecules of the form $C[D \rightarrow> E]$ have the meaning that the attribute $D$ of the individual $C$ has the value $E$. Signature molecules of the form $C[D =>> E]$ indicate that the class $C$ has an attribute $D$ and that all values associated with this attribute are of type $E$.

An important concept in F-Logic is object identity [Khoshafian & Copeland, 1986]. Each object (e.g., class, instance, method) has a unique object identifier, where an object identifier is in fact a term. In F-Logic, classes and methods are interpreted intentionally, which means that class identifiers and method identifiers are interpreted by themselves and not directly as sets or as binary relations, as is the case with concepts and roles in description logics. Classes and methods are first interpreted as objects in the domain and these objects are then related to sets of objects and sets of binary tuples, respectively.

An F-Logic rule is similar to a logic programming rule as defined in the previous section, with the distinction that besides atomic formulas, F-Logic rules also allow molecules in place of atomic formulas.

Example 18. We will now model the description logic knowledge base of Example 8 using F-Logic programming.

We first define the attributes of the class Person. That can be done using a number of ground facts:

- `person[child =>> person].`
- `person[father =>> father].`
- `person[mother =>> mother].`

We now define Man and Woman as sub-classes of Person. It is not possible to say that every person is either a man and a woman, because disjunction is not allowed in the head of rules. We can also capture the facts that Mother is a sub-class of both Woman and Parent and that Father is a sub-class of both Man and Parent. These sub-class assertions are all facts in F-Logic programming:
man::person.
woman::person.
mother::woman.
mother::parent.
father::man.
father::parent.

Finally, we use a number of rules to capture the facts that anybody who has a child is a parent and that every woman who is also a parent is a mother; similar for father:

\[ X:parent \Leftarrow X[child \rightarrow> Y]. \]
\[ X:mother \Leftarrow X:woman, X:parent. \]
\[ X:father \Leftarrow X:man, X:parent. \]

As we can see from the example, there are, on the one hand, several things which can be expressed in basic description logics, but which cannot be expressed in F-Logic programming. Essentially, the two things which could not be expressed are: (1) every person is either a man or a woman and (2) every parent has a child. This is not surprising, since (1) would require disjunction in the head of a rule and (2) would require existentially quantified variables in the head of a rule. Both are not allowed in (F-)logic programming. On the other hand, there are certain kinds of knowledge which can be expressed using (F-)logic programming which cannot be expressed using basic description logics, for example, the uncle rule of Example 14.

Note that in F-Logic there is no distinction between classes and instances. An object identifier can denote a class, an instance, or an attribute, but there is no separation in the signature \( \Sigma \) for the identifiers denoting either. The advantage of such an overloading object notion is that objects denote classes, instances and attributes depending on the syntactic context, which allows certain kinds of meta-statements. For example, in the example above, we might define an attribute parent which is not related to the class parent:

\[ \text{person[parent } \rightarrow> \text{person].} \]

We will now make a few brief remarks about the semantics of F-Logic programming. A full treatment (see [Kifer et al., 1995]) of the F-Logic semantics is beyond the scope of this chapter.

In F-Logic programming, molecules are similar to atomic formulas. In fact, is-a and subclass-of molecules can be seen as binary atomic formulas and data and signature molecules can be seen as ternary atomic formulas. F-Logic does mandate that some additional restrictions hold in the face of is-a and subclass-of molecules. Namely, the subclass-of relation is transitive and an instance of a class is also an instance of all super-classes. However, F-Logic does not prescribe any dependency between data molecules and signature molecules. The definition of type-correctness, i.e., what it means for values of attributes to be correct with respect to the attribute definition, is up to the user of the language.

An F program (a collection of F-Logic rules) is said to be well-typed if all data atoms implied by the program comply with the signatures implied by the program [Kifer et al., 1995]. The notion
of type-correctness is not built into the logical language, but can be defined using a logical metatheory, which can be typically captured using rules. This has two main advantages: (1) it is possible to use different theories for type-correctness for programs in the same language and (2) it enables checking where in the program typing errors occur, instead of just saying that the entire program (or Knowledge Base) is unsatisfiable.

Frame logic and Description Logics (DL) both have constructs for modeling classes, class hierarchies, and attributes. The main difference in the way classes and attributes are modeled in F-Logic and DL is that in F-Logic classes and attributes are modeled intentionally, whereas in DL they are modeled extensionally. This means that, where in F-Logic classes and attributes are interpreted as objects in the domain, which are then associated to sets of objects and sets of (binary) tuples, in description logics classes are directly interpreted as subsets of the domain of interpretation and attributes are directly interpreted as binary relations over the domain. An advantage of the intentional interpretation of classes and attributes is that it is possible to make statements about classes and properties while staying in a first-order framework, whereas making statements about classes and attributes in description logics would require a higher-order logic.

6 RULEML

RuleML (Hirtle et al., 2005; Boley et al., 2005) provides an XML-based exchange syntax for different rule languages, as well as for first-order logic.

RuleML can be seen as an exchange format for most of the languages which have been mentioned in this chapter. In order to capture different logical languages, RuleML defines a number of so-called sub-languages. A sub-language is defined using an XML Schema which comprises a number of modules representing the features which are present in the sub-language. Using the XML Schema modules, a user may compose his/her own sub-language. We illustrate a number of sub-languages which correspond to the languages we have surveyed in this chapter:

fologeq  First-Order Logic with equality is captured using the sub-language fologeq.
  \url{http://www.ruleml.org/0.9/xsd/fologeq.xsd}

nafhornlog  Logic programming with default negation is captured using the sub-language nafhornlog.
  \url{http://www.ruleml.org/0.9/xsd/nafhornlog.xsd}

nafdatalog  Function-free logic programming (Datalog) with default negation is captured using the sub-language nafdatalog.
  \url{http://www.ruleml.org/0.9/xsd/nafdatalog.xsd}

datalog  Function-free logic programming (Datalog) without negation is captured using the sub-language datalog.
  \url{http://www.ruleml.org/0.9/xsd/datalog.xsd}

As an illustration of how the schema’s for the sub-languages are composed of modules, we show part of the schema for the sub-language nafdatalog in Figure \ref{fig:RuleML_Schema}. At the top of the schema we see the usual XML namespace declarations, as part of the top-level element xs:schema. This is followed
Figure 2: XML Schema for nafdatalog

by documentation for the human reader. After this we see that the module for the negation is imported using the xs:include element. Finally, the original schema datalog.xsd is refined to include the possibility of adding negation in the body of rules. For reasons of space we do not show this refinement.

Figure 3 shows the simple program from Example 14 in the datalog sub-language of RuleML.

7 CONCLUSIONS

Logical languages allow one to infer information which is implicit in descriptions one creates in these languages. The ability to infer new information from existing information is seen as an important feature for languages on the Semantic Web. Therefore, many Semantic Web languages, described in the other chapters of this book, are based on formal languages such as the ones we have seen in this chapter.

As we have seen in this chapter, there are many differences between these formal languages, but also in terms of modeling in the language.

We summarize the most important traits of the surveyed languages:

First-Order Logic A very expressive language. Reasoning with first-order logic (FOL) is in general undecidable.

Description Logics Description Logics (DL) are, in general, based on a decidable subset of first-order logic. An important property of Description Logic languages is that they allow, to some extent, frame-based modelling. The most important reasoning task in DL is subsumption reasoning, i.e., checking whether one description is more general than another.
Logic Programming Logic programming is based on the Horn Logic subset of FOL, but with extension of default negation in the body. In general reasoning with logic programs is undecidable, but when restricting to the datalog subset, i.e., disallowing function symbols, reasoning becomes decidable and for programs under the well-founded semantics even tractable.
Frame Logic  First-order logic does not have explicit constructs for modeling classes and attributes. Frame logic overcomes this limitation by introducing a number of constructs for object-oriented modeling which do not increase the complexity of reasoning in the language.

We have seen that RuleML provides an XML-based syntax for exchange of these languages over the Web.

Many of the formal languages described in this chapter have found their way to language recommendations and proposals for the Semantic Web. The language proposals SWSL-FOL (Battle et al. 2005) and SWRL-FOL (Patel-Schneider 2005) are based on full FOL. The W3C recommendation OWL (Dean & Schreiber 2004) is based on Description Logics (Horrocks et al. 2003). Finally, the rule language proposals WRL (Angele et al. 2005) and SWSL-Rules (Battle et al. 2005) are based on (F-)Logic programming.

References


8 Further reading

First-order logic ([Fitting 1996](#))

Logic programming ([Lloyd 1987](#))

Description logics ([Baader et al. 2003](#))

Frame Logic ([Kifer 2005; Kifer et al. 1995](#))

RuleML ([Hirtle et al. 2005; Boley et al. 2005](#))