Marco Marano

Heterogenous Knowledge Representation formalisms in the Semantic Web

PhD Thesis

November 29, 2010

Relatore Prof. Giovambattista Ianni

Contents

to my Love, Marina

1

Introduction and Motivation

"Stay hungry, stay foolish."

Steve Jobs, 5th June 2005

1.1 The Semantic Web

During the years, the World Wide Web has grown exponentially. The growing number of computers, their affordability, the invention of new tools made the Web one of the most successful inventions of human history.
People now spend hours on the Web every day, accessing knowledge and creating new knowledge: they write blogs, publish news and personal experiences on social networks, and most of all they *look for* information they are interested in.
For this purpose, search engines like Google or Yahoo perform "the dirty work", which makes life easier for those who search.
Nevertheless, here the problems arise. Often, in fact, information retrieved by such engines are too general, off topic, or incomplete. It is easy to have the feeling that the search engine did not "get" what the purpose of the search was about.
This is due to the fact that the original Web was designed around users: a collection of documents, linked in a merely syntactic way by anchors, which permit a hypertextual navigation between topics related to each other. Since those days, though, the Web has changed, and nowadays a huge quantity of information is available, often hidden, sometimes difficult to be found. The new Web, called Semantic Web [8] has been designed around machines, and it is an

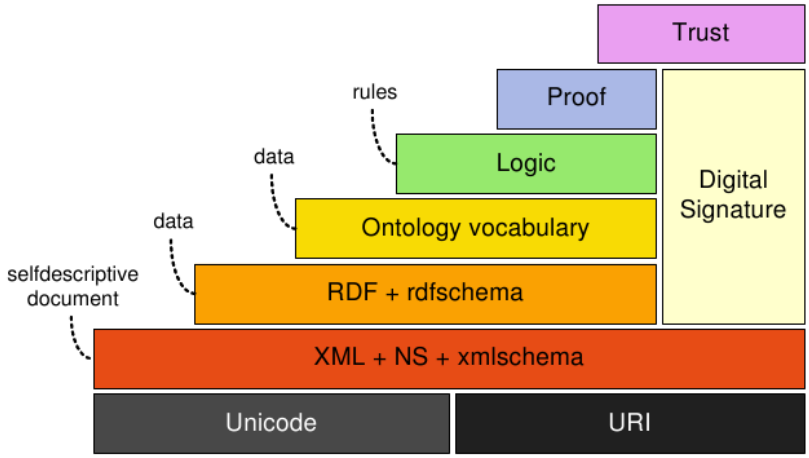

Fig. 1.1: The Semantic Web architecture

extension of the current Web through standards and technologies that enable machines to understand the information on the Web so that they can support richer discovery, data integration, navigation, and automation of tasks.
The Semantic Web will enhance the power of searches, making it possible to provide better answers, and it will also provide tools for integrating different sources, and useful automated services.
Roughly, the main idea behind the Semantic Web is to add a machine-readable meaning to Web pages, to use ontologies for a precise definition of shared terms in Web resources, to make use of knowledge representation technology for automated reasoning from Web resources, and to apply cooperative agent technology for processing the information of the Web.
According to the original design the Semantic Web is divided into layers

At the bottom layer, standards to identify resources have been placed: URI and Unicode. The first one is used to identify resources, the latter to represent typed text.
The above layer hosts languages used to represent data in a semi-structured way, and to provide annotations for them. We are talking, as it should be obvious to the most, of the XML family (XML, namespaces and XML schema).
the third layer provides formalisms to express meta-data expressions using RDF

and its extension RDF Schema.
the fourth layer is based on ontologies. It permits the expression of semantics related to concepts, as we will deal with thoroughly in the following of this thesis.
The final layers deal with logic, proof, and trust issues.
The Digital Signature layer is supposed to provide means to identify the proper origin of a specific resource. We are interested especially in the Ontology Layer and the Logic Level. The Ontology level has reached a good maturity, since the Web Ontology Language (OWL) is now a standard by the w3c. The Logic Level and the Rule Level are being developed actively. Several attempts have been done in order to integrate the World of Rules Languages with the world of Logics. We will deal with some of these ways in this document.
The question is due at this point: Is the Semantic Web the answer to user needs? Will it be successful? We don't know that, but we believe the technologies behind it will be exploited for sure, and it is likely that they are going to change the web as we mean it nowadays, even if the complete vision will not take place.

1.2 Knowledge Representation

Knowledge representation (KR) is a branch of artificial intelligence. It is mainly focused on represent information, or knowledge, in a way that is suitable to support inference. In fact, it goes together with reasoning, which is a service offered by "intelligent" systems in order to derive new knowledge from the already present one.
Knowledge representation tools are specified in a formal way: one must define the reality of interest (by means of a vocabulary or ontology as we will see later on), and the formal logics behind, which will then permit the user to derive the new information. Usually the logics are defined by means of symbols, operators and semantics, which in turn gives meaning to the sentences expressed in that logics.
When one specifies a logics, the compromise between expressivity and complexity is raised. The more expressive logic is, the more complex is to actually reason about it. You can represent more, but new information is harder to derive, or not feasible at all.

KR has been pushed, lately, by the Semantic Web, so it has been developed actively, especially in the field of Ontologies (OWL language and related formalisms).

1.3 Integrating Different Formalisms: Problems and Solutions

In the fast-changing world of Semantic Web, different Knowledge Representation Formalisms have been proposed, and new ones are proposed very often. This will be likely to happen until the Semantic Web is not a "solid" reality.
If you compare these formalisms, you will discover that they have deep differences, both in syntax and semantics. To try to be clear about them, we can divide them into families. On the one hand we find the Description Logic based languages. They are really used in the Semantic Web, especially for modeling Ontologies. We recall here that an Ontology is a formal specification of a collection of Concepts and the relationships between them, in a certain context.
The most used language, OWL (Web Ontology Language) has been inspired by the Description Logics [4]. This family of logic formalisms is derived from the world of First Order Logic, and brings many features of it. On the Contrary, logic programming languages, like Datalog or Prolog, derive from the Database World. When compared, leaving out for the moment the syntactic differences, we can single out the main reason why these formalisms "clash" when used for knowledge representation: Open World Assumption (OWA) vs Closed World Assumption (CWA).
To better explain the problem, we discuss it informally, using an example.

Example 1.1. We have a relation "hasWife". As the name suggests, it is used to store the couples of men and women which are married, stating that the man X has the woman Y as wife. To better specify this situation, we may add some *c*onstraints to this relation. In Description Logics-based languages it is possible to use the constructs *d*omain and *r*ange. In this case, one might say that:

$$domain(hasWife, man)$$

$$range(hasWife, woman)$$

In our knowledge base, there are the following axioms:

$$hasWife(\text{“}John''\text{, “}Jim'')$$

$$man(\text{“}John''), man(\text{“}Jim'')$$

Forgetting for a moment that this is illegal in some states, we ask if this is permitted, and what are the semantic consequences.

Under Closed World Assumption, everything that is not explicitly contained in the Model, i.e. what we know about the world, is supposed to be false. In this case, the only thing we know is that there is a constraints over the knowledge base, which is violated by the statement. So, under CWA the knowledge base is **inconsistent** On the contrary, under OWA, even if there is a constraint, since the semantics is monotonic, that statement is permitted. Constraints like domain and range are used solely for inference (by default, the system would say that a man is married to a woman), but not for excluding statement from the model. It is necessary to say that, from an historical point of view, these differences are closely connected with the scenario in which such assumptions are used. Closed World is typical of Database World. In a typical DB, we will have tables containing data, and this is everything we know. The answers to queries must consist of tuples from the DB, and nothing else (that is to say that nothing can be “invented”). On the other hand, Open World Assumption is used mainly in the world of Web, in particular in RDF and all its extensions (RDFS, OWL, etc.). In a web context, information is changing all the time, and a limited “vision of things” is more suitable than an absolute knowledge.
There is another semantic difference, which is linked to OWA and CWA, but slightly different. We are talking about the problem of Unique Name Assumption (UNA). In logics with the UNA, different names always refer to different entities in the world. To better exemplify this concept, an example is again well placed here.

Example 1.2.

$$hasWife(\text{“}John''\text{, “}Jane'')$$

Moreover we assume not to be in an Islamic country, therefore the following also holds:

$$cardinality(\text{“}hasWife\text{”}, 1)$$

In this case, the following information (false!) is given:

$$hasWife(\text{“}John\text{”}, \text{“}Jenny\text{”})$$

If UNA is adopted, an error is triggered, because there is inconsistency.
If, instead, UNA is not used here, the deduction is that “Jenny” and “Jane” are the same person, i.e. the same object in the knowledge base.

Usually, description-logic based systems don't assume Unique Names. This is due to the nature of the Web. It is not difficult to understand, in fact, that on the Web there are many links and many names which refer to the same entities (files, images, resources). In this scenario, it is useful to have the possibility to ensure equivalence between two syntactically different ways of reference to the same concept.
In logic programming, differently, constants are usually interpreted in terms of themselves, which means that two different constants relate always to different concepts.

1.4 Motivation and Related Work

In this thesis we focus on the problem of integrating hybrid logic formalisms. We concentrate towards studying methods of translation from Semantic Web formats to logic programming.
In particular:

1. We deal with the problem of conjunctive query answering in description logics, and propose a solution based on the translation of ontologies to logic programs.
2. We generalize the previous translation approach using modular translation for various formalisms, like frame logics and OWL2 fragments.
3. We describe the realization of a OWL2 RL reasoner based on RIF and dlvhex.

As for point one, we have designed a novel technique for dealing with description logics based ontologies, and turn them into logic programs,which can in turn be evaluated using state-of-the-art logical engines such as DLV. The fragment we chose id $\mathcal{ELHI}$, because of its computational properties. We formally

prove the effectiveness of our approach, which is also sperimentally confirmed by the prototype software we have designed and tested against competitors.

As for point two, we describe the modularization of different formalisms, using Logic Programming as final language. The first one is Frame Logic, which is translated to be integrated into the DLT framework, in order to be exploited for logic programming. Subsequently, we provide a formal translation for the three fragments of OLW2, pointing out distinctive features.

Finally, for point three, we introduce the language RIF, which we use as a middleware for realizing a OWL2 RL reasoner. The reasoner is based on the dlvhex platform. The implementation procedure is thoroughly described, and results in a prototype which already delivers most of the OWL2 RL functionality, including built-ins and datatypes.

1.4.1 Related Work

In addition to the abovementioned differences with respect to the "vision of the world" occurring between this variety of formalisms, the crucial point is that all of these logics have to coexist in a Web scenario: this a source of many technical problems. The supporters of this or that formalism push to apply their own language, but do the user need all this diversity? Web is meant to help people look for information and knowledge, and share data with others. It is possible to state that a coherent vision of things would improve significantly the user experience.
Different ways may be adopted in order to accommodate things. In the last years, proposals were made in the scientific community, that try to solve these problems.
There exists a more direct way of dealing with different and apparently incompatible formalisms. Given two formalisms $\mathcal{F}_1$ and $\mathcal{F}_2$, we can translate theories expressed in $\mathcal{F}_1$ into theories expressed in $\mathcal{F}_2$. Accomplishing this task is not always possible, depending solely on the two formalisms involved. In our fields of interest, i.e. Semantic Web and Logic Programming, many attempts have been done, with different results. In fact, this is still an open problem, since when two formalisms which are actually very different are put in contact, problems

like inconsistencies, excessive computational complexity and incompatibilities arise.

Translation between formalisms, though, has proven to be particularly useful when dealing with certain tasks of interest. One of these is, query answering, which is historically inefficient in Description Logics knowledge bases.

In fact, in such knowledge bases, queries of instance retrieval and instance checking are usually implemented by spawning a number of independent refutation queries, so that they perform very badly.

For this reason, many techniques have been proposed for rewriting this kind of knowledge bases to other formats, in which more efficient evaluation algorithms are known. Motik [54] presented a resolution-based algorithm for reducing very expressive DL KBs to disjunctive datalog programs.

Kazakov [37] has exploited saturation-based theorem proving to derive a range of decision procedures for various DLs of the EL family [2]. These approaches, however, do not deal with conjunctive queries, which were taken into account by Calvanese et al.[19] for the DL-Lite family of languages, for which query answering was shown to be in LogSpace w.r.t. data complexity; and by Rosati [63] for EL, for which query answering was shown to be PTime-complete w.r.t. data complexity.

Moreover, another rewriting technique has been recently proposed by Lutz et al. [47]. It is based on the EL family. They use a different approach w.r.t. the standard query rewriting techniques, since their algorithm rewrites both the query and the ABox w.r.t. the TBox.

All the aforementioned techniques are closely related; however, they have been designed to handle different DLs.

One of our goals is to propose a new technique for rewriting Description Logics in a scenario of conjunctive query answering. The efficiency gain we claim is based on the fact that we do not perform slow, exponentially costly rewriting by eliminating function symbols, for example.

As a further step, we propose techniques that are *modular*, meaning that it can be easily decomposed in basic steps, one for each of the aspects of starting language. It is furthermore extensible, in the sense that if one wants to add new features to the language, does not have to rewrite everything, but the additions can be made "on the fly".

Finally, we also investigate new Logic Languages such as RIF (Rule Interchange

Format), and prove its usefulness in being a middleware for translation of heterogeneous formalisms (in particular OWL2RL and dlvhex [24]). In particular, we will show how a novel Reasoner for OWL2RL has been realized. It is based on the powerful language called dlvhex, a external-source flavoured version of the popular DLV logic reasoner. RIF is used for the intermediate layer of translation. This is very important and general, as theoretically our reasoner can be used with other logic engines, after writing the correct translator.
It's important to say, in this case, that the prototype we realized is possibly the only one existing with such features, like full support to all RIF built-ins, and full-fledged reasoning in OWL2RL.

1.5 Structure of this Thesis

This thesis is divided into three parts.
In the first part we will investigate the relationship between fragments of Description Logics and Logic Programming, aiming at a translation which may ease the process of Instance Retrieval. In particular, we will introduce a fragment called $\mathcal{ELHI}$, which belongs to the EL family.
We will define the fragments used and the translation formally, and demonstrate it is sound and complete.
In the second part we will study the problem of translating different formalisms, passing from a particular fragment to some dialects of the Web Ontology Language. We will show that such translation can be easily performed using apposite axiomatic modules. We will give a detailed explanation of all the modules necessary for the translation, evidencing the differences occurring between them.
The fragments we are going to translate belong to different families. In the last part we will broaden the scope of the research, introducing the RIF language, and in particular how to employ its power to build a bigger framework. The target of such framework is the realization of a complete OWL2RL reasoner, which exploits the reasoning qualities of the dlvhex system. To this end, an intermediate step has proven to be necessary:using the RIF language as a middleware.

Conclusions will then follow.

2

Preliminaries

"Stop! Who approaches the bridge of death must answer me these questions three, and the other side he see..."

Monty Python and the Holy Grail

This chapter aims at introducing the main topics this thesis is about, giving concepts necessary to better understand the following chapters.
In particular, we will give concepts regarding the following topics:

- Logic Programming and Answer Set Programming: features, syntax and semantics.
- Frame Logic: syntax and semantics.
- Description Logics: introduction and basic language.
- OWL2 Profiles.
- RIF.

2.1 Logic Programming and Answer Set Programming

In this section we introduce some general concepts about Logic Programming, and of one of its flavors, Answer Set Programming (ASP from now on) which will be used very often in the rest of this document. In fact, it will be necessary for all the topics we will encounter, since logic programming is the ending point of all the translation we will propose.
Logic programming consists, in its broadest sense, in the usage of mathematical

logic for computer programming. In this view, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.
ASP is a branch of Logic Programming which has been receiving growing attention in the last years. It has been based on the notion of Stable Model Semantics, and lately on the Answer Set Semantics, which we will discuss in the following. It, as a paradigm, lets the users represent knowledge by means of logic theories, and to infer new knowledge, represented by the*m*odels of the given theory.
The most interesting feature of ASP, which distinguishes this paradigm from languages like Prolog, is its *f*ull declarativity. It means that it makes no difference whatever order is used in specifying facts and rules. On the contrary, Prolog, for example, has a procedural semantics, relying on the syntactic order of rules and subgoals thereof.
ASP is *n*onmonotonic: this means that the already present knowledge is defeasible, as the arrival of new knowledge can alter what is believed to be true.
A very important feature of this kind of programs is the presence of *N*egation as Failure, whose meaning is given in terms of the stable model semantics for normal logic programs. This kind of negation is also known as Default Negation. The semantics has been subsequently extended by the same authors by adding the support for Strong Negation, Disjunction in the head of rules and other useful features.
We will describe the features used by the system named DLV [42] enhanced with more features we have used in the thesis work. Most of these features are available through some systems derived from DLV, which we will describe subsequently.
After this, we will give a brief introduction of some systems which implement the discussed features: dlvhex,DLT,DLV-complex.

2.1.1 Syntax

Here we introduce the logic language we will use later. It is basic logic programming, augmented with the features that will prove to be necessary, especially

function symbols and external atoms, which are non-standard in classical logic programming.

$\mathcal{HLP}$ programs are Logic Programs supporting functions symbols, higher order atoms, external atoms, lists and disjunction in the heads of rules.

Let P, V, C, E be four countable disjoint sets of predicate symbols , variable symbols , constant symbols, external predicate symbols.

For convention (as used by most solvers) elements from V a are represented by strings beginning with uppercase letters, while elements from P and C are represented either as strings beginning with lowercase letters or as strings surrounded by double quotes.

A normal term is either a variable or a constant. Let $t_1, \ldots, t_n$ be terms and f be a function symbol (also called functor) of arity n, $f(t_1, \ldots, t_n)$ is a functional term.

tor) of arity n. A list term can be of the two forms: $[t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are terms; $[h|t]$, where h (the head of the list) is a term, and t (the tail of the list) is a list term.

A *term* is either a normal term,a functional term or a list term.

Each predicate p has a fixed arity $k \geq 0$. Let $p, t_1, \ldots, t_k$ be terms and, in particular, let p be a predicate of arity k, $p(t_1, \ldots, t_k)$ is an *higher order atom* (or *atom*). An atom having p as predicate name is usually referred as $p(t)$. If p is a constant then $p(t)$ is an *ordinary atom*.

An external atom has the form

$$\&g\{Y_1, \ldots, Y_n\}(X_1, \ldots, X_m)$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called input and output lists, $\&g \in G$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates.

A *(positive) disjunctive rule* r is of the form: $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n.$, where $k > 0; \alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_n$ are atoms or external atoms. The disjunction $\alpha_1 \vee \ldots \vee \alpha_k$ is called head of r, while the conjunction $\beta_1, \ldots, \beta_n.$ is the body of r. We denote by $H(r)$ the set of the head atoms, by $B(r)$ the set of body atoms; we refer to all atoms occurring in a rule with $Atoms(r) = H(r) \cup B(r)$. A rule having precisely one head atom (i.e., $k = 1$ and then $|H(r)| = 1$) is

called a normal rule. If r is a normal rule with an empty body (i.e., $n = 0$ and then $B(r) = \emptyset$) we usually omit the $\leftarrow$ sign; and if it contains no variables, then it is referred to as a fact. A LP^{FHELD} program P is a finite set of rules. A $\vee$-free program P is a program consisting of normal rules only.

2.1.2 Semantics

The semantics of LP^{FHELD} programs extends and generalizes the (consistent) answer sets semantics of dis- junctive datalog programs, originally defined in [31] and subsequently in [13].
For any program P, let U_P (the Herbrand Universe) be the set of all constants appearing in P. In case no constant appears in P, an arbitrary constant ψ is added to U_P .
For any program P, let B_P (Herbrand Literal Base) be the set of all ground (classical) literals constructible from the predicate symbols appearing in P and the constants of U_P.
For any rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of U_P . Note that for propositional programs, $P = Ground(P)$ holds.
An interpretation I is a set of ground classical literals, i.e. $I \subseteq B_P w.r.t.$ a program P. A consistent interpretation $X \subseteq B_P$ is called closed under P (where P is a positive disjunctive datalog program), if, for every $r \in Ground(P)$, $H(r) \cap X \neq whenever B(r) \subseteq X$. An interpretation $X \subseteq B_P$ is an answer set for a positive disjunctive datalog program P, if it is minimal (under set inclusion) among all (consistent) interpretations that are closed under P.

2.1.3 dlvhex

We will now introduce dlvhex, which will be used thereafter especially thanks to its external knowledge features. It will be exploited for importing ontologies in the first part, and as terminal reasoner in the third part of this thesis.
dlvhex is the name of a prototype application for computing the models of so-called HEX-programs, which are an extension of Answer-Set Programs towards integration of external computation sources.
In particular, HEX-programs are higher-order logic programs (which accommodate meta-reasoning through higher-order atoms) with external atoms for

software interoperability. Intuitively, a higher-order atom allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols. Look at the following example:

$$C(X) \leftarrow subClassOf(D, C), D(X).$$

An external atom facilitates to determine the truth value of an atom through an external source of computation. For instance, the rule

$$reached(X) \leftarrow \&reach[edge, a](X)$$

computes the predicate reached taking values from the predicate $\&reach$, which computes via $\&reach[edge, a]$ all the reachable nodes in the graph edge from node a, delegating this task to an external computation source (e.g., an external deduction system, an execution library, etc.).
The architecture of dlvhex consists of a core language,which is ASP with higher order, and a collection of plugins, i.e. external atoms written for answering special needs, like the *DL-plugin* (for description logics), the *Sparql plugin*(for using SPARQL queries to query the knowledge base) and the RIF plugin, which is still in development, and will be described in detail in the remainder of this thesis.

2.2 Frame Logic

We introduce next Frame Logic, which will be used in the second part. In particular, it will be integrated in logic programming using a modular translation. Frame Logic(F-logic) [39, 71] is a knowledge representation and ontology modeling language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modeling capabilities supported by the object oriented data model.
As such, F-logic constitutes both an important methodology and a tool for modeling ontologies in the context of Semantic Web. Also, F-logic features play a crucial role in the ongoing activity of the RIF Working group [38]. F-logic was originally defined under first-order semantics [39], while a well-founded semantics, satisfactorily dealing with nonmonotonic inheritance can be found in [71].

F-Logic offers a declarative, compact and simple syntax, as well as the well-defined semantics of a logic-based language. Features include, among others, object identity, complex objects, inheritance, polymorphism, query methods, encapsulation. F-logic stands in the same relationship to object-oriented programming as classical predicate calculus stands to relational database programming. The base of F-Logic is the definition of classes and individuals, as in any other representation language.

Example 2.1.

$$man :: person.$$

$$woman :: person.$$

$$marco : man.$$

$$marina : woman.$$

This states, that "men and women are people" and that "Marco is a man", and "Marina is a woman".

To add details to the classes, just like in object-oriented programming languages, it is possible to express attributes with values.

Example 2.2.

$$person[hasSon \Rightarrow man].$$

$$marco[hasSon \twoheadrightarrow \{eustachio, genoveffa\}].$$

$$married(marco, marina).$$

This defines that "the son of a person is a man", "Eustachio and Genoveffa are the sons of Marco" and "Marco and Marina are married".

In F-Logic it is also possible to express rules to describe relationships between concepts and/or instances, like the following example shows:

Example 2.3.

$$man(X) \leftarrow person(X)\ AND\ NOT\ woman(X).$$

$$FORALL X, Y \leftarrow X : person[hasFather \rightarrow Y \leftarrow Y : man[hasSon \rightarrow X].$$

These mean "X is a man if X is a person but not a woman" and "if X is the son of Y then Y is the father of X".

2.2.1 Syntax

We use here some compact definitions, which can be found in[68].
The alphabet of an F-logic language comprises some object constructors, playing the role of function symbols, a set of variables, and auxiliary symbols like $(,), [,], \rightarrow, \twoheadrightarrow, \bullet\!\!\rightarrow, \bullet\!\!\twoheadrightarrow, \Rightarrow, \Rightarrow\!\!\!\!\Rightarrow$ and the well-known first-order logic connectives.
For convention, object constructors start with lower case letters whereas variables start with uppercase ones.
We will use ID terms for names of objects, classes and methods. They are formed by object constructors and variables.
In the sequel let $o, c, c_1, c_2, m, p_1, \ldots, p_n$ and r be ID-terms for $n \geq 0$. An is-a atom is an expression of the form $o : c$ (the object o is a member of the class c) or $c_1 :: c_2$, that is to say that the class c_1 is a subclass of the class c_2.
We will call $d$ata atoms the following expressions:

$$o[m@(p_1, \ldots, p_n) \rightarrow r] \ (1)$$

$$o[m@(p_1, \ldots, p_n) \bullet\!\!\rightarrow r] \ (2)$$

$$o[m@(p_1, \ldots, p_n) \twoheadrightarrow r] \ o[m@(p_1, \ldots, p_n) \Rightarrow\!\!\!\!\Rightarrow r] \ (3)$$

(1) means that applying the scalar method m with the given parameters to o (object) results in r; in (2), o (class in this case) provides the inheritable scala method m to its members, which, if called with the given parameters, results in r; (3) is the same as above, but for multivalued methods.
An eq-atom is an expression of the form $p_1 = p_2$ with the meaning that p_1 and p_2 denote the same object.
A $r$ule $h \leftarrow b_1, \ldots, b_k$ with $k \geq 1$, is a logic rule over atoms $h, b_1, \ldots, b_k$. A $f$act is a formula $h.$, given an atom h.
A $q$uery is a formula $b_1, \ldots, b_k, k \geq 1$.
A emph program is a set of facts and rules.
Atoms can be combined in $m$olecules in a short notation.
Note that F-logic does not distinguish between classes, methods, and objects which uniformly are denoted by ID-terms; also variables can occur at arbitrary positions of an atom.

2.2.2 Semantics

The semantics of F-logic extends the semantics of first-order logic. Formulas are interpreted over a semantic structure. We restrict our discussion to Herbrand interpretations where the universe consists of all ground ID-terms. An H-structure is a set of ground atoms describing an object world, thus it has to satisfy several closure axioms related to general properties of Object Orientation.

Definition 2.4. *Let H be a (possibly infinite) set of ground atoms. H is an H-structure if the following conditions hold for arbitrary ground ID-terms $u, u_0, \ldots, u_n, u_r, u'_r$ and u_m:*

- $u :: u \in H$ *(subclass reflexivity).*
- *if* $u_1 :: u_2 \in H$ *and* $u_2 :: u_3 \in H$ *then* $u_1 :: u_3 \in H$ *(subclass transitivity).*
- $u_1 :: u_2 \in H$ *and* $u_2 :: u_1 \in H$ *then* $u_1 = u_2 \in H$ *(subclass acyclicity).*
- *if* $u_1 : u_2 \in H$ *and* $u_2 :: u_3 \in H$ *then* $u_1 : u_3 \in H$ *(instance-subclass dependency).*
- *if* $u_0[u_m@(u_1, \ldots, u_n \twoheadrightarrow u_r)] \in H$ *and* $u_0[u_m@(u_1, \ldots, u_n \twoheadrightarrow u'_r)] \in H$ *then* $u_r = u'_r \in H$. *where* $\twoheadrightarrow$ *stands for* $\rightarrow$ *or* $\bullet\!\!\rightarrow$ *(uniqueness of scalar methods).*

Furthermore the well known free equality axioms for $=$ have to hold.
With respect to an H-structure the meaning of atoms and formulas is given in the usual way, moreover, minimal models can be defined as standard meaning of a program.

2.3 Description Logics

Description Logics has a very important role in this thesis, since it involved in all topics we will deal with.
Description Logics [4](DLs) is a family of Knowledge Representation formalisms that represent the knowledge of an application domain by first defining its relevant concepts (terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description).

Description Logics, as the name suggests, is characterized by a formal, logic-based semantics.

Another distinguished feature is the emphasis on reasoning as a central service: reasoning allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base. Description Logics support inference patterns that occur in many applications of intelligent information processing systems, and which are also used by humans to structure and understand the world: classification of concepts and individuals.

A knowledge base (KB) comprises two components, the TBox and the ABox. The TBox introduces the terminology, i.e., the vocabulary of an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary.

The vocabulary consists of concepts, which denote sets of individuals, and roles, which denote binary relationships between individuals. In addition to atomic concepts and roles (concept and role names), all DL systems allow their users to build complex descriptions of concepts and roles. The TBox can be used to assign names to complex descriptions. The language for building descriptions is a characteristic of each DL system, and different systems are distinguished by their description languages. The description language has a model-theoretic semantics. Thus, statements in the TBox and in the ABox can be identified with formulae in first-order logic or, in some cases, a slight extension of it.

A DL system not only stores terminologies and assertions, but also offers services that reason about them. Typical reasoning tasks for a terminology are to determine whether a description is satisfiable (i.e., non-contradictory), or whether one description is more general than another one, that is, whether the first subsumes the second. Important problems for an ABox are to find out whether its set of assertions is consistent, that is, whether it has a model, and whether the assertions in the ABox entail that a particular individual is an instance of a given concept description. Satisfiability checks of descriptions and consistency checks of sets of assertions are useful to determine whether a knowledge base is meaningful at all.

With subsumption tests, one can organize the concepts of a terminology into a hierarchy according to their generality. A concept description can also be conceived as a query, describing a set of objects one is interested in. Thus, with instance tests, one can retrieve the individuals that satisfy the query.

In any application, a KR system is embedded into a larger environment. Other components interact with the KR component by querying the knowledge base and by modifying it, that is, by adding and retracting concepts, roles, and assertions. A restricted mechanism to add assertions are rules. Rules are an extension of the logical core formalism, which can still be interpreted logically. However, many systems, in addition to providing an application programming interface that consists of functions with a well-defined logical semantics, provide an escape hatch by which application programs can operate on the KB in arbitrary ways.

2.3.1 Basic Language

Here we give some notions of the basic constructors it is possible to use to built a DL knowledge base.
To *d*escribe a reality of interest, we need some constructors, useful to specify concepts. They can give either simple or complex descriptions. The simple ones are called *a*tomic concepts and roles. Starting from these, one can form a more complex description, the form of which depends on the language in use.
Here we will not focus on a language in particular, on the contrary this introduction will deal with general concepts, universally applicable.
We have some *a*tomic concepts and roles,i.e. they can be expressed only in terms of themselves.
We call *B*asic Concept a concept which can be expressed using the constructors of the language, starting from one or more atomic concepts. Without loss of generality we assume these constructors to be used:

$$A \sqsubseteq B(C_1)$$
$$A \sqcap B \sqsubseteq C(C_2)$$
$$A \sqsubseteq \exists R.B(C_3)$$
$$\exists R.A \sqsubseteq B(C_4)$$
$$\exists R.\top \sqsubseteq B(C_5)$$
$$B \sqsubseteq \exists R.\top(C_6)$$
$$P \sqsubseteq S(C_7)$$
$$R^- \sqsubseteq S(C_8)$$
$$R \sqsubseteq S^-(C_9)$$

Axiom $\mathcal{ELHI}$ H	FOL formula$\mathcal{F}(H)$
$A(a)$	$A(a)$
$R(a,b)$	$R(a,b)$
$A \sqsubseteq B$	$\forall x A(x) \rightarrow B(x)$
$A \sqcap B \sqsubseteq C$	$\forall x A(x) \wedge B(x) \rightarrow C(x)$
$A \sqsubseteq \exists R.B$	$\forall x A(x) \rightarrow [\exists y B(y) \wedge R(x,y)]$
$\exists R.A \sqsubseteq B$	$\forall x [\exists y A(y) \wedge R(x,y)] \rightarrow B(x)$
$\exists R.\top \sqsubseteq B$	$\forall x [\exists y R(x,y)] \rightarrow B(x)$
$B \sqsubseteq \exists R.\top$	$\forall x B(x) \rightarrow [\exists y R(x,y)]$
$R \sqsubseteq S$ or $R^- \sqsubseteq S^-$	$\forall x,y\, R(x,y) \rightarrow S(x,y)$
$R \sqsubseteq S^-$ or $R^- \sqsubseteq S$	$\forall x,y\, R(y,x) \rightarrow S(x,y)$

Table 2.1: Semantics of a DL knowledge base given in terms of the corresponding FO formulas.

Semantics for a DL knowledge is given by means of first order logics (FOL). In, particular, a DL ontology can be seen as a conjunction of FOL formulas. the models of each axiom are the same of the ones of the corresponding FOL formula. In the table 2.1, we show, for each axiom, the corresponding first order formula.
Given a knowledge base, the semantics is given by the following formula:

$$\mathcal{F}(\mathcal{KB}) = \bigwedge_{H \in \mathcal{T}} \mathcal{F}(H) \wedge \bigwedge_{H \in \mathcal{A}} \mathcal{F}(H).$$

2.4 OWL2 and OWL2 Profiles

OWL2 is the new version of the popular Ontology Web Language (OWL). It is divided into profiles, as we will see later in detail. It was designed in order to be more modular than its predecessor, and focused on tasks, giving different expressivity for different scenarios.
Similarly to OWL, it is based on description logics, in particular there is a different fragment for each OWL2 profile. It permits, for this reason, to define vocabularies and axioms, to model reality. This is possible by defining ontologies.

2.4.1 OWL2

An OWL 2 [41] ontology is a formal description of a domain of interest. OWL 2 ontologies consist of the following three different syntactic categories:

- Entities, such as classes, properties, and individuals, are identified by IRIs. They form the primitive terms of an ontology and constitute the basic elements of an ontology. For example, a class a:Person can be used to represent the set of all people. Similarly, the object property a:parentOf can be used to represent the parent-child relationship. Finally, the individual $a : Marco$ can be used to represent a particular person called "Marco".
- Expressions represent complex notions in the domain being described. For example, a class expression describes a set of individuals in terms of the restrictions on the individuals' characteristics. Axioms are statements that are asserted to be true in the domain being described. For example, using a subclass axiom, one can state that the class $a : Student$ is a subclass of the class $a : Person$. These three syntactic categories are used to express the logical part of OWL 2 ontologies that is, they are interpreted under a precisely defined semantics that allows useful inferences to be drawn. For example, if an individual $a : Marco$ is an instance of the class $a : Student$, and $a : Student$ is a subclass of $a : Person$, then from the OWL 2 semantics one can derive that $a : Marco$ is also an instance of $a : Person$.
- Expressions represent complex notions in the domain being described. For example, a class expression describes a set of individuals in terms of the restrictions on the individuals' characteristics. Axioms are statements that are asserted to be true in the domain being described. For example, using a subclass axiom, one can state that the class $a : Student$ is a subclass of the class $a : Person$.

In addition, entities, axioms, and ontologies can be annotated in OWL 2. For example, a class can be given a human-readable label that provides a more descriptive name for the class. Annotations have no effect on the logical aspects of an ontology that is, for the purposes of the OWL 2 semantics, annotations are treated as not being present. Instead, the use of annotations is left to the applications that use OWL 2. For example, a graphical user interface might choose to visualize a class using one of its labels.

Finally, OWL 2 provides basic support for ontology modularization. In particular, an OWL 2 ontology O can import another OWL 2 ontology O' and thus gain access to all entities, expressions, and axioms in O'.

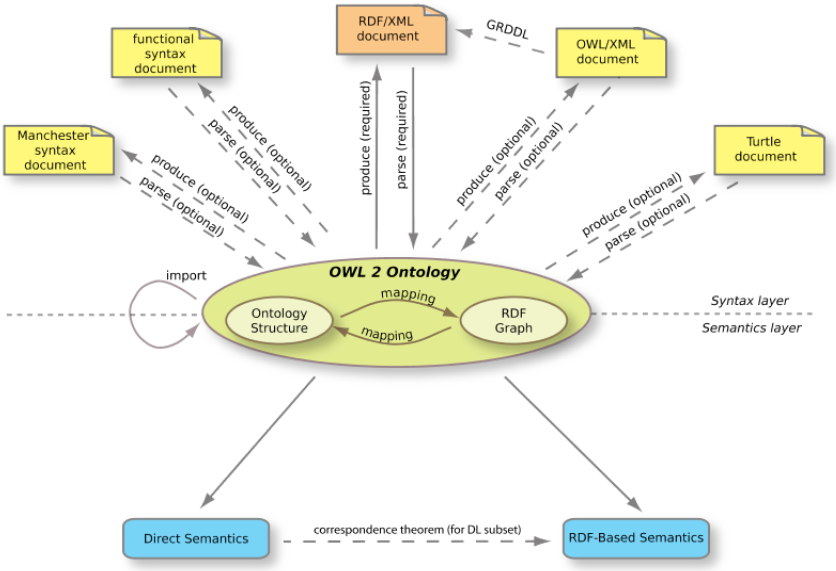

Fig. 2.1: OWL2 Typical Workflow.

2.4.2 OWL2 Profiles

An OWL 2 profile (commonly called a fragment or a sublanguage in computational logic) is a trimmed down version of OWL 2 that trades some expressive power for the efficiency of reasoning.
Here we describe three profiles of OWL 2, each of which achieves efficiency in a different way and is useful in different application scenarios. The profiles are independent of each other. The choice of which profile to use in practice will depend on the structure of the ontologies and the reasoning tasks at hand.

OWL2 EL

OWL 2 EL is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. This profile captures the expressive power used by many such ontologies and is a subset of OWL 2 for which the basic reasoning problems can be performed in time that is polynomial with respect to the size of the ontology [EL++]. Dedicated reasoning algorithms for this profile are available and have been demonstrated to be implementable in a highly scalable way. The EL acronym reflects the profile's basis in the EL family of description logics [EL++], logics that provide only Existential quantification.

OWL2 QL

OWL 2 QL is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. In OWL 2 QL, conjunctive query answering can be implemented using conventional relational database systems. Using a suitable reasoning technique, sound and complete conjunctive query answering can be performed in LOGSPACE with respect to the size of the data (assertions). As in OWL 2 EL, polynomial time algorithms can be used to implement the ontology consistency and class expression subsumption reasoning problems. The expressive power of the profile is necessarily quite limited, although it does include most of the main features of conceptual models such as UML class diagrams and ER diagrams. The QL acronym reflects the fact that query answering in this profile can be implemented by rewriting queries into a standard relational Query Language.

OWL2 RL

OWL 2 RL is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to accommodate OWL 2 applications that can trade the full expressivity of the language for efficiency, as well as RDF(S) applications that need some added expressivity. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines. The ontology consistency, class expression satisfiability, class expression subsumption, instance checking, and conjunctive query answering problems can be

solved in time that is polynomial with respect to the size of the ontology. The RL acronym reflects the fact that reasoning in this profile can be implemented using a standard Rule Language.

2.5 RIF

The *Rule Interchange Format* (RIF) is a new language, whose definition was started in 2005 by the RIF working group.
RIF focused on exchange rather than trying to develop a single one-fits-all rule language because, in contrast to other Semantic Web standards, such as RDF, OWL, and SPARQL, it was immediately clear that a single language would not satisfy the needs of many popular paradigms for using rules in knowledge representation and business modeling. But even rule exchange alone was recognized as a daunting task. Known rule systems fall into three broad categories: first-order, logic-programming, and action rules. These paradigms share little in the way of syntax and semantics. Moreover, there are large differences between systems even within the same paradigm.

Given this diversity, what is the most useful notion of rule exchange? The approach taken by the Working Group was to design a family of languages, called dialects, with rigorously specified syntax and semantics. The family of RIF dialects is intended to be uniform and extensible. RIF uniformity means that dialects are expected to share as much as possible of the existing syntactic and semantic apparatus. Extensibility here means that it should be possible for motivated experts to define a new RIF dialect as a syntactic extension to an existing RIF dialect, with new elements corresponding to desired additional functionality. These new RIF dialects would be non-standard when defined, but might eventually become standards.

Because of the emphasis on rigor, the word format in the name of RIF is somewhat of an understatement. RIF in fact provides more than just a format. However, the concept of format is essential to the way RIF is intended to be used. Ultimately, the medium of exchange between different rule systems is XML, a format for data exchange. Central to the idea behind rule exchange

through RIF is that different systems will provide syntactic mappings from their native languages to RIF dialects and back. These mappings are required to be semantics-preserving, and thus rule sets can be communicated from one system to another provided that the systems can talk through a suitable dialect, which they both support.

The RIF group, as stated previously, decided to create different dialects, to satisfy different needs. The existing dialects are:

- Rif-Core
- Rif-BLD
- Rif-PRD

Here we will focus mainly on the BLD dialect, that has been used for our purpose. In the following we will present the syntax and the semantics of this particular dialect.

2.5.1 Syntax

We will use the Rif *Presentation Syntax*, which is not a concrete syntax (the only concrete one is the XML syntax), but is well suited for better showing the language features.
It deliberately leaves out details such as the delimiters of the various syntactics components, escape symbols, and other similar symbols.
From now on, with a little abuse of notation, we talk about the elements of RIF, but actually we are dealing with the elements of RIF *presentation syntax*.

The *Alphabet* of RIF-BLD consists of a countably infinite set of constant symbols $\mathcal{C}$; a countably infinite set of variable symbols $\mathcal{V}$ (disjoint from $\mathcal{C}$); a countably infinite set of argument names, $\mathcal{A}$ (disjoint from $\mathcal{C}$ and $\mathcal{V}$); connective symbols $And, Or, \leftarrow$; quantifiers symbols $Exists, Forall$; the symbols $=$ $, \#, \#\#, ->, External, Import, Prefix, Base$; the symbols $Group, Document$; the symbols for representing lists: $List, OpenList$;the auxiliary symbols $(,), [,], <$ $, >, \hat{}\hat{}$.
The set of connective symbols, quantifiers, $=$, etc., is disjoint from $\mathcal{C}$ and $\mathcal{V}$.

The argument names in $\mathcal{A}$ are written as Unicode strings that must not start with a question mark, "?". Variables are written as Unicode strings preceded with the symbol "?".

Constants are written as "*literal*"^^symspace, where literal is a sequence of Unicode characters and symspace is an identifier for a symbol space. Symbol spaces are defined in Section Constants, Symbol Spaces, and Datatypes of RIF-DTB. For the description of RIF-DTB, please look at the following paragraphs.

The symbols $=, \#, \#\#$ are used in formulas that define equality, class membership, and subclass relationships. The symbol $->$ is used in terms that have named arguments and in frame formulas. The symbol *External* indicates that an atomic formula or a function term is defined externally (e.g., a built-in) and the symbols Prefix and Base enable compact representations of IRIs (RFC-3987).

The symbol *Document* is used to specify RIF-BLD documents, the symbol *Import* is an import directive, and the symbol *Group* is used to organize RIF-BLD formulas into collections.
The language of RIF-BLD is the set of formulas constructed using the above alphabet according to the rules given below.
In RIF-BLF it is possible to build several types of terms:

- Constants and variables. If $t \in \mathcal{C}$ or $t \in \mathcal{V}$ then t is a simple term.
- Positional terms. If $t \in \mathcal{C}$ and $t_1, \ldots, t_n, n \geq 0$, are base terms then $t(t_1 \ldots t_n)$ is a positional term.
- Positional terms correspond to the usual terms and atomic formulas of classical first-order logic [Enderton01, Mendelson97].
- Terms with named arguments. A term with named arguments is of the form $t(s_1 \rightarrow v_1 \ldots s_n \rightarrow v_n)$, where $n \geq 0, t \in \mathcal{C}$ and $v_1, \ldots, v_n$ are base terms and $s_1, \ldots, s_n$ are pairwise distinct symbols from the set $\mathcal{A}$.

The constant t here represents a predicate or a function; $s_1, \ldots, s_n$ represent argument names; and $v_1, \ldots, v_n$ represent argument values. The argument names, $s_1, \ldots, s_n$, are required to be pairwise distinct. Terms with named arguments are like positional terms except that the arguments are named and their order is immaterial. Note that a term of the form $f()$ is, trivially, both a positional

term and a term with named arguments.
Terms with named arguments are introduced to support exchange of languages that permit argument positions of predicates and functions to be named (in which case the order of the arguments does not matter).

- List terms. There are two kinds of list terms: open and closed. A closed list has the form $List(t_1 \dots t_m)$, where $m \geq 0$ and $t_1, \dots, t_m$ are terms. An open list (or a list with a tail) has the form $OpenList(t_1 \dots t_m t)$, where $m > 0$ and $t_1, \dots, t_m, t$ are terms. Open lists are usually written using the following: $List(t_1 \dots tm|t)$. The last argument, t, represents the tail of the list and so it is normally a list as well. A closed list of the form List() (i.e., a list in which m=0, corresponding to Lisp's nil) is called the empty list.
- Equality terms. $t = s$ is an equality term, if t and s are base terms.
- Class membership terms (or just membership terms). $t\#s$ is a membership term if t and s are base terms.
- Subclass terms. $t\#\#s$ is a subclass term if t and s are base terms.
- Frame terms. $t[p_1 \rightarrow v_1 \dots p_n \rightarrow v_n]$ is a frame term (or simply a frame) if $t, p_1, \dots, p_n, v_1, \dots, v_n, n \geq 0$, are base terms. Membership, subclass, and frame terms are used to describe objects and class hierarchies.
- Externally defined terms. If t is a positional or a named-argument term then $External(t)$ is an externally defined term. External terms are used for representing built-in functions and predicates as well as "procedurally attached" terms or predicates, which might exist in various rule-based systems, but are not specified by RIF.

Observe that the argument names of frame terms, $p_1, \dots, p_n$, are base terms and so, as a special case, can be variables. In contrast, terms with named arguments can use only the symbols from ArgNames to represent their argument names. They cannot be constants from $\mathcal{C}$ or variables from $\mathcal{V}$. The reason for not allowing variables for those is to control the complexity of unification, which is used by several inference mechanisms of first-order logic.
RIF-BLD distinguishes certain subsets of the set Const of symbols, including subsets of predicate symbols and function symbols. Section Well-formed Formulas gives more details, but we do not need those details yet.

Definition 2.5. *(Atomic Formula). Any term (positional or with named arguments) of the form p(...), where p is a predicate symbol, is also an atomic formula. Equality, membership, subclass, and frame terms are also atomic formulas. An externally defined term of the form External(φ), where φ is an atomic formula, is also an atomic formula, called an externally defined atomic formula.*

It is important to remark that simple terms (constants and variables) are not formulas.
More general formulas are constructed from atomic formulas with the help of logical connectives.

Definition 2.6. *(Formula). A formula can have several different forms and is defined as follows:*

- *Atomic: If φ is an atomic formula then it is also a formula.*
- *Condition formula: A condition formula is either an atomic formula or a formula that has one of the following forms:*
  - *Conjunction: If $\varphi_1, \ldots, \varphi_n, n \geq 0$, are condition formulas then so is And($\varphi_1 \ldots \varphi_n$), called a conjunctive formula. As a special case, And() is allowed and is treated as a tautology, i.e., a formula that is always true.*
  - *Disjunction: If $\varphi_1, \ldots, \varphi_n, n0$, are condition formulas then so is Or($\varphi_1 \ldots \varphi_n$), called a disjunctive formula. As a special case, Or() is permitted and is treated as a contradiction, i.e., a formula that is always false.*

  Existentials: If φ is a condition formula and $?V_1, \ldots, ?V_n, n > 0$, are distinct variables then Exists$?V_1 \ldots ?V_n(\varphi)$ is an existential formula.

Condition formulas are intended to be used inside the premises of rules. Next we define the notions of rule implications, universal rules, universal facts, groups (i.e., sets of rules and facts), and documents.

Rule implication: $\varphi \leftarrow \psi$ is a formula, called rule implication, if: φ is an atomic formula or a conjunction of atomic formulas, ψ is a condition formula, and none of the atomic formulas in φ is an externally defined term (i.e., a term of the form *External*(...)).
Universal rule: If φ is a rule implication and $?V_1, \ldots, ?V_n, n > 0$, are distinct

variables then $Forall?V_1 \ldots ?V_n(\varphi)$ is a formula, called a universal rule. It is required that all the free variables in φ occur among the variables ?V1 ... ?Vn in the quantification part. An occurrence of a variable ?v is free in φ if it is not inside a subformula of φ of the form $Exists?v(\psi)$ and ψ is a formula. Universal rules will also be referred to as RIF-BLD rules. Universal fact: If φ is an atomic formula and $?V_1, \ldots, ?V_n, n > 0$, are distinct variables then $Forall?V_1 \ldots ?V_n(\varphi)$ is a formula, called a universal fact, provided that all the free variables in φ occur among the variables $?V_1 \ldots ?V_n$. Universal facts are often considered to be rules without premises.

Group: If $\varphi_1, \ldots, \varphi_n$ are RIF-BLD rules, universal facts, variable-free rule implications, variable-free atomic formulas, or group formulas then Group(φ1 ... φn) is a group formula. As a special case, the empty group formula, $Group()$, is allowed and is treated as a tautology, i.e., a formula that is always true. Non-empty group formulas are used to represent sets of rules and facts. Note that some of the φ_i's can be group formulas themselves, which means that groups can be nested.

Document: An expression of the form $Document(directive_1 \ldots directive_n \Gamma)$ is a RIF-BLD document formula (or simply a document formula), if Γ is an optional group formula; it is called the group formula associated with the document. $directive_1, \ldots, directive_n$ is an optional sequence of directives. A directive can be a base directive, a prefix directive or an import directive. A base directive has the form $Base(< iri >)$, where iri is a Unicode string in the form of an absolute IRI [RFC-3987]. The Base directive defines a syntactic shortcut for expanding relative IRIs into full IRIs, as described in Section Constants, Symbol Spaces, and Datatypes of [RIF-DTB].

A prefix directive has the form $Prefix(p < v >)$, where p is an alphanumeric string that serves as the prefix name and v is an expansion for p – a Unicode sequence of characters that forms an IRI. (An alphanumeric string is a sequence of ASCII characters, where each character is a letter, a digit, or an underscore , and the first character is a letter.) Like the Base directive, the Prefix directives define shorthands to allow more concise representation of constants that come from the symbol space rif:iri (we will call such constants rif:iri constants). This mechanism is explained in [RIF-DTB], Section Constants, Symbol Spaces, and Datatypes.

An import directive can have one of these two forms: $Import(< loc >)$ or $Import(< loc >< p >)$. Here loc is a Unicode sequence of characters that forms an IRI and p is another Unicode sequence of characters. The constant loc represents the location of another document to be imported; it is called the locator of the imported document. The argument p is called the profile of import; it has the form of a Unicode character sequence in the form of an IRI – see [RIF-RDF+OWL].
A document formula can contain at most one Base directive. The Base directive, if present, must be first, followed by any number of Prefix directives, followed by any number of Import directives.
In the definition of a formula, the component formulas $\varphi, \varphi_i, \psi_i$, and Γ are said to be subformulas of the respective formulas (condition, rule, group, etc.) that are built using these components.

RIF-BLD Annotations in the Presentation Syntax

RIF-BLD allows every term and formula (including terms and formulas that occur inside other terms and formulas) to be optionally preceded by one annotation of the form (* id φ *), where id is a rif:iri constant and φ is a frame formula or a conjunction of frame formulas. Both items inside the annotation are optional. The id part represents the identifier of the term or formula to which the annotation is attached and φ is the metadata part of the annotation. RIF-BLD does not impose any restrictions on φ apart from what is stated above. This means that it may include variables, function symbols, constants from the symbol space rif:local (often referred to as local or rif:local constants), and so on.

Document formulas with and without annotations will be referred to as RIF-BLD documents.

Well formed formulas

Not all formulas and thus not all documents are well-formed in RIF-BLD: it is required that no constant appear in more than one context. What this means precisely is explained below. Informally, this means that each constant symbol in RIF-BLD can be either an individual, a plain function, a plain predicate, an externally defined function, or an externally defined predicate. However,

symbols can be polyadic: the same function or predicate symbol (normal or external) can occur with different numbers of arguments in different places. Note that polyadic symbols could be replaced by non-polyadic symbols with the arity information encoded in the function or predicate names.
The set of all constant symbols, $\mathcal{C}$, is partitioned into the following subsets:

- A subset of individuals.
- The symbols in Const that belong to the symbol spaces of Datatypes are required to be individuals.
- A subset of plain (i.e., non-external) function symbols.
- A subset for external function symbols.
- A subset of plain predicate symbols.
- A subset for external predicate symbols.

The above subsets do not differentiate between positional and named argument symbols. Also, as seen from the following definitions, these subsets are not specified explicitly but, rather, are inferred from the occurrences of the symbols.
Definition (Context of a symbol). The context of an occurrence of a symbol, Const, in a formula, φ, is determined as follows:
If s occurs as a predicate of the form s(...) (positional or named-argument) in an atomic subformula of φ then s occurs in the context of a (plain) predicate symbol. If s occurs as a function symbol in a non-subformula term of the form $s(\ldots)$ then s occurs in the context of a (plain) function symbol. If s occurs as a predicate in an atomic subformula External($s(\ldots)$) then s occurs in the context of an external predicate symbol. If s occurs as a function in a non-subformula term External($s(\ldots)$) then s occurs in the context of an external function symbol. If s occurs in any other context (in a frame: $s[\ldots], \ldots [s \rightarrow \ldots], or \ldots [\ldots \rightarrow s]$; or in a positional/named-argument term: $p(\ldots s \ldots)$, $q(\ldots \rightarrow s \ldots)$), it is said to occur as an individual.

Definition 2.7. *(Imported document). Let Δ be a document formula and Import(loc) be one of its import directives, where loc is a locator of another document formula, Δ'. We say that Δ' is directly imported into Δ.*
A document formula Δ' is said to be imported into Δ if it is either directly imported into Δ or it is imported (directly or not) into some other formula that is directly imported into Δ.

The above definition deals only with one-argument import directives, since only such directives can be used to import other RIF-BLD documents. Two-argument import directives are provided to enable import of other types of documents, and their semantics are supposed to be covered by other specifications, such as [RIF-RDF+OWL].
Definition (Well-formed formula). A formula φ is well-formed iff:
every constant symbol (whether coming from the symbol space rif:local or not) mentioned in φ occurs in exactly one context. if φ is a document formula and 'k are all of its imported documents, then every non-rif:local constant symbol mentioned in φ or any of the imported 'is must occur in exactly one context (in all of the is). whenever a formula contains a term or a subformula of the form External(t), t must be an instantiation of a schema in the coherent set of external schemas (Section Schemas for Externally Defined Terms of [RIF-DTB]) associated with the language of RIF-BLD. if t is an instantiation of a schema in the coherent set of external schemas associated with the language then t can occur only as External(t), i.e., as an external term or atomic formula.
Definition (Language of RIF-BLD). The language of RIF-BLD consists of the set of all well-formed formulas and is determined by:
the alphabet of the language and a set of coherent external schemas, which determine the available built-ins and other externally defined predicates and functions.

2.5.2 Semantics

Truth Values

The set TV of truth values in RIF-BLD consists of two values, t and f.

Semantic Structures

The key concept in a model-theoretic semantics for a logic language is the notion of a semantic structure [Enderton01, Mendelson97]. The definition is slightly more general than what is strictly necessary for RIF-BLD alone. This lays the groundwork for extensions to RIF-BLD and makes the connection with the semantics of the RIF framework for logic-based dialects [RIF-FLD] more obvious.

Definition 2.8. *(Semantic structure). A semantic structure, I, is a tuple of the form* <TV, DTS, D, Dind, Dfunc, IC, IV, IF, INF, Ilist, Itail, Iframe, Isub, Iisa, I=, Iexternal, Itruth>. *Here D is a non-empty set of elements called the domain of I, and Dind, Dfunc are nonempty subsets of D.* D_{ind}*is used to interpret the elements of Const that occur as individuals and Dfunc is used to interpret the elements of Const that occur in the context of function symbols. As before, Const denotes the set of all constant symbols and Var the set of all variable symbols. DTS denotes a set of identifiers for datatypes (please refer to Section Datatypes of [RIF-DTB] for the semantics of datatypes).*

The other components of I are total mappings defined as follows:
IC maps $\mathcal{C}$ to D. This mapping interprets constant symbols. In addition:
If a constant, $c \in \mathcal{C}$, is an individual then it is required that IC(c) We also define the following mapping from terms to D, which we denote using the same symbol I as the one used for semantic structures. This overloading is convenient and creates no ambiguity.
I_V maps V to D_{ind}. This mapping interprets variable symbols. I_F maps D to total functions $D^*_{ind} \rightarrow D$ (here D^*_{ind} is a set of all finite sequences over the domain D_ind). This mapping interprets positional terms. In addition if $d \in D_{func}$ then $I_F(d)$ must be a function $D^*_{ind} \rightarrow D_{ind}$. This means that when a function symbol is applied to arguments that are individual objects then the result is also an individual object. INF maps D to the set of total functions of the form $SetOfFiniteSets(ArgNamesxD_ind) \rightarrow D$. This mapping interprets function symbols with named arguments. In addition if $d \in D_{func}$ then $INF(d)$ must be a function $SetOfFiniteSets(ArgNamesxD_{ind}) \rightarrow D_{ind}$. This is analogous to the interpretation of positional terms with two differences: Each pair $< s, v > \in AxD_{ind}$ represents an argument/value pair instead of just a value in the case of a positional term. The arguments of a term with named arguments constitute a finite set of argument/value pairs rather than a finite ordered sequence of simple elements. So, the order of the arguments does not matter. I_{list} and I_{tail} are used to interpret lists. They are mappings of the following form: $I_{list} : D*_{ind} \rightarrow D_{ind}$; $I_{tail} : D_{ind} + xD_{ind} \rightarrow D_{ind}$
In addition, these mappings are required to satisfy the following conditions: (i) The function I_{list} is injective (one-to-one); (ii) The set $I_{list}(D^*_{ind})$, henceforth denoted D_{list}, is disjoint from the value spaces of all data types in DTS. $I_{tail}(a_1, \ldots, a_k, I_{list}(a_{k+1}, \ldots, a_{k+m})) = I_{list}(a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+m})$.

Note that the last condition above restricts I_{tail} only when its last argument is in D_{list}. If the last argument of I_{tail} is not in D_{list}, then the list is a general open one and there are no restrictions on the value of I_{tail} except that it must be in D_{ind}. I_{frame} maps $D_{ind}$to total functions of the form $SetOfFiniteBags(D_{ind}D_{ind})D$. This mapping interprets frame terms. An argument, $d \in D_{ind}$, to I_{frame} represents an object and the finite bag $< a_1, v_1 >, \ldots, < a_k, v_k >$ represents a bag of attribute-value pairs for d. We will see shortly how I_{frame} is used to determine the truth valuation of frame terms.

I_{sub} gives meaning to the subclass relationship. It is a mapping of the form $D_{ind}xD_{ind} \rightarrow D$. I_{sub} will be further restricted in Section Interpretation of Formulas to ensure that the operator ## is transitive, i.e., that $c_1\#\#c_2$ and $c_2\#\#c_3$ imply $c_1\#\#c_3$.

I_{isa} gives meaning to class membership. It is a mapping of the form D_{ind} D_{ind} D. I_{isa} will be further restricted in Section Interpretation of Formulas to ensure that the relationships # and ## have the usual property that all members of a subclass are also members of the superclass, i.e., that $o\#cl$ and $cl\#\#scl$ imply $o\#scl$.

$I_{=}$ is a mapping of the form $D_{ind}xD_{ind} \rightarrow D$. It gives meaning to the equality operator.

I_{truth} is a mapping of the form $D \rightarrow TV$. It is used to define truth valuation for formulas.

$I_{external}$ is a mapping from the coherent set of schemas for externally defined functions to total functions $D* \rightarrow D$. For each external schema $\sigma = (?X1 \ldots ?Xn; \tau)$ in the coherent set of external schemas associated with the language, $I_{external}(\sigma)$ is a function of the form $D_n \rightarrow D$.

For every external schema, σ, associated with the language, $I_{external}(\sigma)$ is assumed to be specified externally in some document (hence the name external schema). In particular, if σ is a schema of a RIF built-in predicate or function, $I_{external}(\sigma)$ is specified in [RIF-DTB] so that:

- If σ is a schema of a built-in function then $I_{external}(\sigma)$ must be the function defined in [RIF-DTB].

- If σ is a schema of a built-in predicate then $I_{truth} \circ (I_{external}(\sigma))$ (the composition of I_{truth} and $I_{external}(\sigma)$, a truth-valued function) must be as specified in [RIF-DTB].

We also define the following mapping from terms to D, which we denote using the same symbol I as the one used for semantic structures. This overloading is convenient and creates no ambiguity.

- $I(k) = IC(k)$, if k is a symbol in C;
- $I(?v) = IV(?v)$, if $?v$ is a variable in V;
- $I(f(t_1 \ldots t_n)) = IF(I(f))(I(t_1), \ldots, I(t_n))$;
- $I(f(s_1 \rightarrow v_1 \ldots s_n \rightarrow v_n)) = INF(I(f))(< s1, I(v_1) >, \ldots, < s_n, I(v_n) >)$;

Here we use $\ldots$ to denote a set of argument/value pairs.
For list terms, the mapping is defined as follows: $I(List()) = I_{list}(<>)$.
Here $<>$ denotes an empty list of elements of D_{ind}. (Note that the domain of I_{list} is D^*_{ind}, so D^0_{ind} is an empty list of elements of D_{ind}.)

$$I(List(t_1 \ldots t_n)) = I_{list}(I(t_1), \ldots, I(t_n)), n > 0$$

.

$$I(List(t_1 \ldots t_n|t)) = I_{tail}(I(t_1), \ldots, I(t_n), I(t))n > 0$$

.

$$I(o[a_1 \ldots v_1 \ldots a_k \rightarrow v_k]) = I_{frame}(I(o))(< I(a_1), I(v_1) >, \ldots, < I(a_n), I(v_n) >)$$

Here $\ldots$ denotes a bag of attribute/value pairs. Jumping ahead, we note that duplicate elements in such a bag do not affect the truth value of a frame formula. Thus, for instance, $[a \rightarrow ba \rightarrow b]$ and $o[a \rightarrow b]$ always have the same truth value.
$I(c_1\#\#c_2) = Isub(I(c_1), I(c_2))$
$I(o\#c) = Iisa(I(o), I(c))$
$I(x = y) = I = (I(x), I(y))$

$I(External(t)) = I_{external}(\sigma)(I(s_1), \ldots, I(s_n))$, if t is an instantiation of the external schema $\sigma = (?X_1 \ldots ?X_n; \tau)$ by substitution $?X_1/s_1 \ldots ?X_n/s_1$.
Note that, by definition, External(t) is well-formed only if t is an instantiation of an external schema. Furthermore, by the definition of coherent sets of external schemas, t can be an instantiation of at most one such schema, so $I(External(t))$ is well-defined.

The effect of datatypes.

The set DTS must include the datatypes described in Section Datatypes of [RIF-DTB].
The datatype identifiers in DTS impose the following restrictions. Given $dt \in DTS$, let LS_{dt} denote the lexical space of dt, VS_{dt} denote its value space, and $L_{dt} : LS_{dt} \rightarrow VS_{dt}$ the lexical-to-value-space mapping (for the definitions of these concepts, see Section Datatypes of [RIF-DTB]). Then the following must hold: $VS_{dt} \subseteq D_{ind}$; and For each constant $\text{“}lit\text{''}\hat{}\hat{}dt$ such that $lit \in LS_{dt}, IC(\text{“}lit\text{''}\hat{}\hat{}dt) = L_{dt}(lit)$.
That is, IC must map the constants of a datatype dt in accordance with L_{dt}.
RIF-BLD does not impose restrictions on IC for constants in symbol spaces that are not datatypes included in DTS.

Interpretation of Non-document Formulas

This section defines how a semantic structure, I, determines the truth value TVal$\|$(φ) of a RIF-BLD formula, φ, where φ is any formula other than a document formula. Truth valuation of document formulas is defined in the next section.
We define a mapping, TVal$\|$, from the set of all non-document formulas to TV. Note that the definition implies that TVal$\|$(φ) is defined only if the set DTS of the datatypes of I includes all the datatypes mentioned in φ and I$\mathrm{external}$ is defined on all externally defined functions and predicates in φ.
Definition (Truth valuation). Truth valuation for well-formed formulas in RIF-BLD is determined using the following function, denoted TVal$\|$:
Positional atomic formulas:

$$TVal\|(r(t_1 \ldots t_n)) = Itruth(I(r(t_1 \ldots t_n)))$$

Atomic formulas with named arguments:
$TValI(p(s_1 \rightarrow v_1 \ldots sk \rightarrow v_k)) =$
$Itruth(I(p(s1 \rightarrow v_1 \ldots sk \rightarrow v_k)))$.
Equality: $TValI(x = y) = Itruth(I(x = y))$.
To ensure that equality has precisely the expected properties, it is required that: $Itruth(I(x = y)) = t$ if $I(x) = I(y)$ and that $Itruth(I(x = y)) = f$ otherwise. This is tantamount to saying that $TValI(x = y) = t$ if and only if $I(x) = I(y)$. Subclass: $TValI(sc\#\#cl) = Itruth(I(sc\#\#cl))$. To ensure that the operator $\#\#$ is transitive, i.e., $c_1\#\#c_2$ and $c_2\#\#c_3$ imply $c1\#\#c_3$, the following is required:
$\forall c_1, c_2, c_3 \in D$, if $TValI(c1\#\#c2) = TValI(c_2\#\#c_3) = t$ then $TValI(c_1\#\#c_3)$ $= t$.
Membership: $TValI(o\#cl) = Itruth(I(o\#cl))$. To ensure that all members of a subclass are also members of the superclass, i.e., $o\#cl$ and $cl\#\#scl$ imply $o\#scl$, the following is required:
$\forall o, cl, scl \in D$, if $TValI(o\#cl) = TValI(cl\#\#scl) = t$ then $TValI(o\#scl) = t$.
Frame: $TValI(o[a_1 \rightarrow v_1 \ldots a_k \rightarrow v_k]) = Itruth(I(o[a_1 \rightarrow v_1 \ldots a_k \rightarrow v_k]))$.
Since the bag of attribute/value pairs associated with an object o represents the conjunction of assertions represented by these pairs, the following is required, if k $>$ 0:
$TValI(o[a_1 \rightarrow v_1 \ldots a_k -> v_k]) = t$ if and only if $TValI(o[a_1 -> v_1]) = \ldots =$ $TValI(o[a_k -> v_k]) = t$.
Externally defined atomic formula:
$TValI(External(t)) = Itruth(Iexternal(\sigma)(I(s1), \ldots, I(sn)))$, if t is an atomic formula that is an instantiation of the external schema $\sigma = (?X1 \ldots ?Xn; \tau)$ by substitution $?X1/s1 \ldots ?Xn/s1$. Note that, by definition, $External(t)$ is well-formed only if t is an instantiation of an external schema. Furthermore, by the definition of coherent sets of external schemas, t can be an instantiation of at most one such schema, so $I(External(t))$ is well-defined.
Conjunction: $TValI(And(c1 \ldots cn))$ = t if and only if $TValI(c1) = \ldots =$ $TValI(cn) = t$. Otherwise, $TValI(And(c1 \ldots cn)) = f$. The empty conjunction is treated as a tautology, so $TValI(And()) = t$.

Disjunction: $TValI(Or(c1 \ldots cn)) = f$ if and only if $TValI(c1) = \ldots =$ $TValI(cn) = f$. Otherwise, $TValI(Or(c1 \ldots cn)) = t$. The empty disjunction

is treated as a contradiction, so $TValI(Or()) = f$.

Quantification: $TValI(Exists?v_1 \ldots ?vn(\varphi)) = t \iff for\ some I^* TValI^*(\varphi) = t$.
$TValI(Forall?v_1 \ldots ?vn(\varphi)) = t \iff \forall I^* TValI*(\varphi) = t$.
Here I^* is a semantic structure of the form $< TV, DTS, D, Dind, Dfunc, IC, I*V, IF, INF, Ilist, Itail, Iframe, Isub, Iisa, I =, Iexternal, Itruth >$, which is exactly like I, except that the mapping I^*V, is used instead of IV. I^*V is defined to coincide with IV on all variables except, possibly, on $?v_1, \ldots, ?vn$.
Rule implication: $TValI(conclusion \leftarrow condition) = t$;
if either $TValI(conclusion) = t$ or $TValI(condition) = f$.
$TValI(conclusion \leftarrow condition) = f$ otherwise.
Groups of rules: If Γ is a group formula of the form $Group(\varphi_1 \ldots \varphi_n)$ then $TValI(\Gamma) = t$ if and only if $TValI(\varphi_1) = t, \ldots, TValI(\varphi_n) = t$.
$TValI(\Gamma) = f$ otherwise.
This means that a group of rules is treated as a conjunction. In particular, the empty group is treated as a tautology, so $TValI(Group()) = t$.

Interpretation of Documents

Document formulas are interpreted using semantic multi-structures, which are sets of closely related semantics structures. The need for multi-structures arises due to the fact that a RIF-BLD document can import other documents and thus is essentially a multi-document object. One interesting aspect of the multi-document semantics is that rif:local symbols that belong to different documents can have different meanings.

Definition 2.9. *(Semantic multi-structure). A semantic multi-structure $\hat{I}$ is a set of semantic structures of the form $J, I; Ii_1, Ii_2, ...$, where I and J are RIF-BLD semantic structures; and Ii_1, Ii_2, etc., are semantic structures adorned with the locators of distinct RIF-BLD formulas (one can think of these adorned structures as locator-structure pairs). All the structures in $\hat{I}$ (adorned and non-adorned) are identical in all respects except for the following:*
The mappings $JC, IC, ICi_1, ICi_2, \ldots$ may differ on the constants in C that belong to the rif:local symbol space.

As will be seen from the next definition, the structure I in the above is used to interpret document formulas, and the adorned structures of the form Iik are used to interpret imported documents. The structure J is used in the definition of entailment for non-document formulas.
The semantics of RIF documents is now defined as follows.

Definition 2.10. *(Truth valuation of document formulas). Let Δ be a document formula and let $\Delta_1, \ldots, \Delta_n$ be all the RIF-BLD document formulas that are imported (directly or indirectly, according to Definition Imported document) into Δ. Let $\Gamma, \Gamma_1, \ldots, \Gamma_n$ denote the respective group formulas associated with these documents. Let $\hat{I} = J, I; Ii_1, \ldots, Ii_n, \ldots$ be a semantic multi-structure that contains the semantic structures adorned with the locators $i_1, ..., i_n$ of the documents $\Delta_1, ..., \Delta_n$. Then we define:*
$TVal\hat{I}(\Delta) = t$ *if and only if* $TValI(\Gamma) = TValIi_1(\Gamma_1) = \ldots = TValIi_k(\Gamma_n) = t$.

Note that this definition considers only those document formulas that are reachable via the one-argument import directives. Two argument import directives are not covered here. Their semantics is defined by the document RIF RDF and OWL Compatibility [RIF-RDF+OWL].
Also note that some of the Γ_i above may be missing since all parts in a document formula are optional. In this case, we assume that Γ_i is a tautology, such as $And()$, and every $TVal$ function maps such a Γ_i to the truth value t.
For non-document formulas, we extend $TVal\hat{I}(\varphi)$ from regular semantic structures to multi-structures as follows. Let $\hat{I} = J, I; \ldots$ be a semantic multi-structure. Then $TVal\hat{I}(\varphi) = TValJ(\varphi)$.

The above definitions make the intent behind the rif:local constants clear: occurrences of such constants in different documents can be interpreted differently even if they have the same name. Therefore, each document can choose the names for the rif:local constants freely and without regard to the names of such constants used in the imported documents.

Logical Entailment

We now define what it means for a set of RIF-BLD rules (embedded in a group or a document formula) to entail another RIF-BLD formula. In RIF-BLD we

are mostly interested in entailment of RIF condition formulas, which can be viewed as queries to RIF-BLD groups or documents. Entailment of condition formulas provides formal underpinning to RIF-BLD queries.

Definition 2.11. *(Models). A multi-structure $\hat{I}$ is a model of a formula, φ, written as $\hat{I} \models \varphi$, $\Longleftrightarrow$ $TVal\hat{I}(\varphi) = t$. Here φ can be a document or a non-document formula.*

Definition 2.12. *(Logical entailment). Let φ and ψ be (document or non-document) formulas. We say that φ entails ψ, written as $\varphi \models \psi$, if and only if for every multi-structure, $\hat{I}$, $\hat{I} \models \varphi$ implies $\hat{I} \models \psi$.*

Part I

Translating heterogeneous formalisms

3

From Description Logics to Answer Set Programming

3.1 Introduction

In this chapter we address an important problem, which arises from the huge quantity of formalisms present in the Semantic Web: *I*ntegration.
Knowledge representation is a very important task for the Semantic Web, and many of the formalisms introduced to that end are ontology-based (see e.g. [34]).
To accomplish this task, we will focus on some particular fragments. In particular, we will exploit the representation power of Description Logics, as well as the reasoning power of logic programming.
The difficulties arise in choosing a "language family" which is nowadays suitable to our end.
In this respect, the OWL language made recently a significant step towards solid maturity after the introduction of the OWL 2 W3C Recommendation [52]. We gave several details about OWL2 in the previous chapter. This recommendation gives legitimate focus on fragments of the OWL 2 general language tailored at efficient performance taken from different perspectives and/or reasoning tasks [50]: classification of large ontologies (OWL EL), query answering (OWL QL), and expressiveness tailored at rule-based axiomatization and implementation (OWL RL). The OWL EL profile specifically identifies a fragment of OWL 2 (based on the description logic $\mathcal{EL}$++ [2, 3]) which sacrifices expressiveness-yet preserving many constructs used in practical ontologies- but allows classification in polynomial time.
The fragment $\mathcal{ELH}$ has a fair theoretical complexity also when conjunctive query (CQ, in the following) answering is considered (P-time complete, [63]),

but this is beyond what is commonly considered the current complexity requirement for scalability on large ABoxes: roughly speaking, it is highly desirable that conjunctive querying over ontologies comes at little or no additional cost with respect to conjunctive querying over plain databases (recall that the data complexity of CQ answering over plain databases is as elementary as the complexity class AC_0 [7]). This computational complexity requirement has direct impact on implementation of query answering, since it calls for overcoming some peculiar technical difficulties: while the OWL QL profile, based on the $DL{-}Lite_R$ description logics, enjoys the so called *FO-reducibility* property, this latter allowing scalable implementation on standard RDBMS systems, OWL2 EL does not enjoy this property [18].

On the other hand, the OWL2 RL profile allows a straightforward implementation via an axiomatization expressed in terms of FOL Horn clauses: this latter enables the possibility of bottom-up materializing inferred information in RDBMSs, via logic programming and/or deductive database tools. This is, for instance, the approach taken by the Oracle 11g Database Semantic Technologies [70]. OWL2 EL lacks the possibility of a direct implementation like the above, mostly due to the fact that unrestricted existential quantification implies that, when inferred information has to be materialized in practice, a possibly infinite number of new Skolem terms is obtained.

In order to demonstrate the practical viability of query answering on $\mathcal{EL}{+}{+}$ (or fragments/extensions thereof), a number of proposals has been devised, aimed at circumventing these technical difficulties: in particular, in [47] it is shown how $\mathcal{ELH}_{\bot}^{dr}$ ontologies can be queried by pre-materializing a canonical model which enlarges the database/ABox at hand. Furthermore, the query at hand is appropriately rewritten (this latter step being independent from terminological and instance data). This approach leads to the notion of *combined fo-rewritability*, in which both Abox and Tbox are converted into a FO structure (an enlarged version of the ABox), which can be queried using traditional first order queries (i.e. SQL).

As an alternative approach, in [55] query answering on the description logic $\mathcal{ELHIO}^{\neg}$ is treated by a preliminary resolution step which eliminates Skolem terms from the equivalent FO representation of the ontology at hand.

We focus in this thesis on the $\mathcal{ELHI}$ description logic, which covers most of the basic constructs of OWL EL, plus inverse roles. We propose a newly-devised

approach to Conjunctive Query answering, which takes new significant steps towards a viable implementation using (deductive) database technologies and/or logic programming techniques.
In particular:

1. TBoxes are rewritten into a set of corresponding Horn clauses, possibly containing function terms: the corresponding class of logic programs –called $\mathcal{ELHI}$– is identified and its properties analyzed.
2. We show that queries on $\mathcal{ELHI}$-programs can be answered against a *finite* portion of their ground instantiation, which is only polynomially larger than the original ABox. As a by-product result, we show how conjunctive querying in this class of programs can be finitely evaluated, thus enlarging the family of fragments of logic programming [26, 27, 15, 9] for which decidability of querying, even in the presence of function symbols, is known.
3. **Function terms are neither eliminated nor pre-processed in any way**. Our approach can be directly implemented:
   - over rule-based systems, provided that function terms or some form of value invention is supported (the language fragment herein considered is fully supported by solvers like XSB [65] or DLV [42]); or
   - directly over DBMS systems, by a stored procedure which iteratively invokes a series of SQL queries, producing a polynomially-bounded number of new values.
4. The approach lend itself to two query-evaluation strategies:
   - *Pre-processing with storage of inferred information.* Notably, and despite the presence of function terms, a materialization step which computes the least model of the finite ground logic programs needed for answering CQs, has only logspace complexity in the size of the ABox. Such a model is polynomially larger than the original ABox.
   - *On-the-fly evaluation of queries:* The non-ground $\mathcal{ELHI}$ program corresponding to the TBox can be stored together with the original ABox, avoiding the pre-materialization step. Then the program can be evaluated on a query per query basis. Known optimization techniques, such as magic sets [5], allow to significantly reduce the size of data processed when submitting queries.

   It is known that the first approach might be space-consuming and system bootstrap times might be non negligible when large ABoxes are loaded,

although small updates are usually performed in significantly shorter time: if storage space is a constraint, the second strategy might be desired.

5. Fast proof-of-concept prototyping: we show how our approach can be fast prototyped using existing technologies [23, 1, 42]. In particular, reasoning on knowledge bases can be axiomatized by meta-rules which can be declaratively specified, and subsequently evaluated on a rule-based system. Preliminary experimental results reveal encouraging performance of our prototype, despite the lack of ad-hoc optimizations.

3.2 The description logics fragment: ELHI

We are going to introduce the fragment in use for our project. We reintroduce syntax and semantics, specialized to the case, in order to better exemplify the subsequent techniques.

3.2.1 Basic language.

We consider the description logic $\mathcal{ELHI}$ extending the basic $\mathcal{EL}$ [2] language with inverse roles, and role containment.[1] It is given a set $\mathcal{N}_C$ of atomic concept names, a set $\mathcal{N}_R$ of atomic role names, and a set $\mathcal{N}_I$ of constant (individual) names. If R is an *atomic* role, then a *basic* role can be either R or R^-. A *basic concept* can be of the form A, $\top$, $\exists R.A$, or $B_1 \sqcap B_2$, where B_1 and B_2 are basic concepts, R is a role and A is an *atomic concept*. A TBox $\mathcal{T}$ is a set of concept (resp. role) *inclusion assertions* of the form $B_1 \sqsubseteq B_2$ (resp. $R_1 \sqsubseteq R_2$), where B_1 and B_2 are basic concepts (resp. R_1 and R_2 are basic roles). Without loss of generality we assume that axioms in a TBox $\mathcal{T}$ are of the form $A \sqsubseteq B$, $A \sqcap B \sqsubseteq C$, $A \sqsubseteq \exists R.B$, $\exists R.A \sqsubseteq B$, $\exists R.\top \sqsubseteq B$, $B \sqsubseteq \exists R.\top$, $P \sqsubseteq S$, $R^- \sqsubseteq S$, and $R \sqsubseteq S^-$ for A, B, C atomic concepts, R an atomic role and P and S basic roles.[2] An ABox $\mathcal{A}$ is a set of *membership assertions* in two possible forms: $A(a)$ and $R(a, b)$, where A is a concept, R is a role and a, b are individuals from the domain $\mathcal{N}_I$. A $\mathcal{ELHI}$ knowledge base is defined as $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox.

[1] This language basically covers most of OWL-EL, and includes in addition inverse roles.
[2] An equisatisfiable TBox of this form can be obtained from a general $\mathcal{ELHI}$ TBox by applying a number of syntactic substitutions.

$\mathcal{ELHI}$ axiom H	FOL sentence $\mathcal{F}(H)$	Rule set $\mathcal{L}(\mathcal{F}(H))$
$A(a)$	$A(a)$	$A(a).$ $(R1)$
$R(a,b)$	$R(a,b)$	$R(a,b).$ $(R2)$
$A \sqsubseteq B$	$\forall x A(x) \rightarrow B(x)$	$B(X) \leftarrow A(X).$ $(R3)$
$A \sqcap B \sqsubseteq C$	$\forall x A(x) \wedge B(x) \rightarrow C(x)$	$C(X) \leftarrow A(X), B(X).$ $(R4)$
$A \sqsubseteq \exists R.B$	$\forall x A(x) \rightarrow [\exists y B(y) \wedge R(x,y)]$	$B(f_{\mathcal{A}}(X)) \leftarrow A(X).$ $(R5)$ $R(X, f_{\mathcal{A}}(X)) \leftarrow A(X).$
$\exists R.A \sqsubseteq B$	$\forall x [\exists y A(y) \wedge R(x,y)] \rightarrow B(x)$	$B(X) \leftarrow A(Y), R(X,Y).$ $(R6)$
$\exists R.\top \sqsubseteq B$	$\forall x [\exists y R(x,y)] \rightarrow B(x)$	$B(X) \leftarrow R(X,Y).$ $(R7)$
$B \sqsubseteq \exists R.\top$	$\forall x B(x) \rightarrow [\exists y R(x,y)]$	$R(X, f_{\mathcal{A}}(X)) \leftarrow B(X).$ $(R8)$
$R \sqsubseteq S$ or $R^- \sqsubseteq S^-$	$\forall x,y\, R(x,y) \rightarrow S(x,y)$	$S(X,Y) \leftarrow R(X,Y).$ $(R9)$
$R \sqsubseteq S^-$ or $R^- \sqsubseteq S$	$\forall x,y\, R(y,x) \rightarrow S(x,y)$	$S(X,Y) \leftarrow R(Y,X).$ $(R10)$
	FOL Query $Q(\mathbf{X})$	Rule $\mathcal{L}(Q(\mathbf{X}))$
	$\exists \mathbf{Y}\, q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n)$	$ans_Q(\mathbf{X}) \leftarrow$ $(Q1)$ $q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n).$

Table 3.1: Semantics of $\mathcal{ELHI}$ given in terms of corresponding FOL sentences. For an axiom $\mathcal{A}$ in the form $A \sqsubseteq \exists R.B$, $f_{\mathcal{A}}$ denotes a fresh function symbol. Analogously, for a query Q, ans_Q denotes a fresh predicate name.

We give the semantics of a $\mathcal{ELHI}$ knowledge base in terms of a conjunction of first order sentences. In particular, in Table 3.1, each $\mathcal{ELHI}$ axiom H is associated to the corresponding first order sentence $\mathcal{F}(H)$. The semantics of $\mathcal{KB}$ is given by its corresponding first order theory $\mathcal{F}(\mathcal{KB}) = \bigwedge_{H \in \mathcal{T}} \mathcal{F}(H) \wedge \bigwedge_{H \in \mathcal{A}} \mathcal{F}(H)$.

3.2.2 Queries.

A *conjunctive query* $Q(\mathbf{X})$ (or simply, query) is a formula $\exists \mathbf{Y}\, q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n)$, where: *(i)* $\{q_1, \ldots q_n\} \subseteq \mathcal{N}_C \cup \mathcal{N}_R$; each $\mathbf{X}_i (1 \leq i \leq n)$ is a list of variables and constants, having according arity with the corresponding q_i. *(ii)* $\mathbf{X}$ is the non-empty list of free variables of $Q(\mathbf{X})$, while $\mathbf{Y}$ is the remaining list of bound variables of $Q(\mathbf{X})$. We assume Q is connected, that is there exist a permutation P of its atoms such that if q_i precedes q_j in P then q_i and q_j have at least one variable in common. Let k be the arity of $\mathbf{X}$. An *answer* to $Q(\mathbf{X})$ over a knowledge base $\mathcal{KB}$, is a k-tuple of constants of $\mathcal{N}_I$ such that $\mathcal{F}(\mathcal{KB}) \models$

$Q(x_1, \ldots, x_k)$. We denote the set of answers to $Q(\mathbf{X})$ as $ans(Q(\mathbf{X}), \mathcal{KB})$. A particular case of query answering is *Instance Retrieval (*IR*)* in which $n = 1$.

3.2.3 Logic Programs

We refer here the general concepts regarding logic programs $\mathcal{HLP}$, seen in 2, adding some details which are necessary in this chapter.
A logic program P is a set of closed universally-quantified FOL formulas (called *rules*) of the form [44]:

$$r : a(\mathbf{X}_0) \leftarrow b_1(\mathbf{X}_1), \ldots, b_n(\mathbf{X}_n).$$

For ease of notation, we omit universal quantifiers while ',' stands for '$\wedge$'. [3]. In a rule r the atom $a(\mathbf{X}_0)$ is called *head*, while the conjunction $b_1(\mathbf{X}_1), \ldots, b_n(\mathbf{X}_n)$ is called *body*. The set of atoms appearing in the body is denoted by $B(r)$, similarly, $H(r)$ denotes the head of r. A rule is *safe* if all the variables occurring in $H(r)$ also occur in some atom of $B(r)$; in the following, we assume that rules are safe. An atom (resp. rule) is said to be *ground* if it does not contain variables.

The semantics of a logic program P is usually given in terms of its *minimal Herbrand model* denoted by $LM(P)$. More in detail, let U_P be the set of all ground terms that can be built combining constants and functors appearing in P. U_P is called the *Herbrand Universe* of P. The *Herbrand Base* of P, denoted by B_P, is the set of all ground atoms obtainable combining predicate names occurring in P with elements from U_P. Given a ground atom $a(t_1, \ldots, t_n)$ we denote by $NL(a(t_1, \ldots, t_n))$ the number $NL(a(t_1, \ldots, t_n)) = max_{0<i<n} NL(t_i)$ $(n \geq 0)$. The set of all ground instances of the rules of P w.r.t. the universe U_P, denoted by $Ground(P, U_P)$, is called *(ground) instantiation* of P. An *Herbrand model* M is a subset of B_P such that, for each $r \in Ground(P, U_P)$, $H(r) \in M$ or there is an atom $a \in B(r)$ such that $a \notin M$. M is a minimal model if there does not exist a Herbrand model N of P such that $N \subsetneq M$. Given an atom a (resp. a set of atoms S), we write $M \models_{LP} a$ if $a \in M$ (resp. $M \models_{LP} S$ if $S \subseteq M$),

[3] Note that logic programs are by default interpreted under Unique Names Assumption (UNA), while, in principle, the semantics of $\mathcal{ELHI}$ knowledge bases could be given in terms of FOL theories without UNA. This latter choice has no impact on the description logic herein considered, we thus assume semantic of $\mathcal{ELHI}$ knowledge bases is given in terms of FOL theories with UNA.

otherwise $M \not\models_{LP} a$ (resp. $M \not\models_{LP} S$). A logic program P is guaranteed to admit a unique minimal model $LM(P)$, which coincides with the intersection of all its Herbrand models [44].
We recall some, easy to prove, relevant properties of logic programs. Hereafter we assume that a program P is given. Note that U_P, as well as $Ground(P, U_P)$ and $LM(P)$, might be infinite.

Proposition 3.1. *For each* $U \subseteq U_P$,
$LM(Ground(P, U)) \subseteq LM(Ground(P, U_P)) = LM(P)$.

Let be $U_k = \{t \in U_P | NL(t) \leq k\}$, $k \geq 0$, we denote by $LM_k(P)$ the minimal model of the program $Ground(P, U_k)$. The following properties hold.

Proposition 3.2. *(i) for* $0 \leq i < j$, $Ground(P, U_i) \subseteq Ground(P, U_j)$ *and* $LM_i(P) \subseteq LM_j(P)$*; and (ii) for each atom* $a \in LM_i(P)$, $NL(a) \leq i$.

Given a set $I \subseteq B_P$, we say that a ground atom a is *supported* in I if there exists a *supporting* rule $r \in ground(P, U_P)$ such that the $B(r) \subseteq I$ and $H(r) = a$.
Let P be a ground program and $a \in P$ be a ground atom. A *rule defining* a in P, is a rule $r \in P$ such that $H(r) = a$. The *program* P_a *defining* a in P is the smallest program $P_a \subseteq P$, such that it contains all rules defining a in P, and for each $b \in B(r)$, $r \in P_a$, $P_b \subseteq P_a$. The atom a is *well-supported* in I if there exists a strict well-founded partial ordering $<$ on elements of I such that there exists a rule $r \in Ground(P, U_P)$ with $H(r) = a$, $body(R) \subseteq I$ and for any $b \in B(r)$, $b < a$. I is *well-supported* if all its atoms are well-supported. The following known result holds

Theorem 3.3. *[29] For a logic program* P, $LM(P)$ *is well-supported.*

3.3 $\mathcal{ELHI}$-Programs

We now illustrate how $\mathcal{ELHI}$ knowledge bases can be translated to to corresponding logic programs. We then define and focus on the class of $\mathcal{ELHI}$-programs and show some of their properties.
Given a $\mathcal{ELHI}$ knowledge base $\mathcal{KB}$, the corresponding logic program $\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ is defined as In Table 3.1. For each type of axiom H it is reported the corresponding FOL formula $\mathcal{F}(H)$ and, for each formula $\mathcal{F}(H)$ it is reported the corresponding logic program $\mathcal{L}(\mathcal{F}(H))$ obtained by standard Skolemization.

For ease of notation, we assume that both the predicate vocabulary of $\mathcal{F}(H)$ and $\mathcal{L}(\mathcal{F}(H))$ contain $\mathcal{N}_C$ and $\mathcal{N}_R$. The logic program corresponding to $\mathcal{KB}$ is:

$$\mathcal{L}(\mathcal{F}(\mathcal{KB})) := \bigcup_{H \in \mathcal{KB}} \mathcal{L}(\mathcal{F}(H)).$$

We recall the following.

Theorem 3.4 ([44]). *Let $\mathcal{KB}$ be a $\mathcal{ELHI}$ knowledge base then:*

- *$\mathcal{F}(\mathcal{KB})$ satisfiable $\Rightarrow$ $\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ has a minimal model;*
- *$\mathcal{F}(\mathcal{KB})$ unsatisfiable $\Leftrightarrow$ $\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ is unsatisfiable.*

In general, the $LM(\mathcal{L}(\mathcal{F}(\mathcal{KB})))$ is not finite minimal model as shown in the following example.

Example 3.5. Let consider the knowledge base $\mathcal{K} = \{A \sqsubseteq \exists R.A, A(c)\}$, thus the corresponding $\mathcal{L}(\mathcal{F}(\mathcal{K}))$ is

$$A(f_{\mathcal{A}}(X)) \leftarrow A(X). \qquad R(X, f_{\mathcal{A}}(X)) \leftarrow A(X). \qquad A(c).$$

The minimal model of P is

$$\{A(c), A(f_{\mathcal{A}}(c)), R(c, f_{\mathcal{A}}(c)), A(f_{\mathcal{A}}(f_{\mathcal{A}}(c))), R(f_{\mathcal{A}}(c), f_{\mathcal{A}}(f_{\mathcal{A}}(c))), \ldots\}$$

which is infinite. ∎

Note that the binary predicates occurring in $LM(\mathcal{L}(\mathcal{F}(\mathcal{KB})))$ contain terms having a specific form as we will see later.

Definition 3.6. *A $\mathcal{ELHI}$-program P is a logic program containing set of rules in the form $(R1)$-$(R9)$ (Table 3.1) and (possibly) one rule in the form $(Q1)$.*

It is easy to see that a $\mathcal{ELHI}$-program is into one-to-one correspondence with a $\mathcal{ELHI}$ knowledge base.

Lemma 3.7. *Let P be a $\mathcal{ELHI}$-program, then for each binary predicate R occurring in P such that $R(t_1, t_2) \in LM(P)$, it holds that $|NL(t_2) - NL(t_1)| \leq 1$.*

Proof. (Sketch). As shown in Table 3.1, the binary predicates can be supported in $LM(P)$ only in rules of the form *(i)* $R(a, b)$ for $NL(a) = NL(b) = 0$, for which the Lemma trivially holds; *(ii)* $R(X, f_{\mathcal{A}}(X)) \leftarrow B(X)$ and $R(f_{\mathcal{A}}(X), X) \leftarrow$

$B(X)$: in this case, ground rules obtained by these two rules form can support only atoms in the form $R(t, f_{\mathcal{A}}(t))$ or $R(f_{\mathcal{A}}(t), t)$, for which the Lemma holds; and, *(iii)* $R(X,Y) \leftarrow S(X,Y)$, $R(X,Y) \leftarrow S(Y,X)$: for this two form of rules note that, when grounded, they can in principle support an atom $a = S(t_1, t_2)$ for which $|NL(t_2) - NL(t_1)| > 1$. Note however that a cannot be well-supported: thus it cannot appear in $LM(P)$ by Theorem 3.3.

3.4 Query answering

In this section we show how query answering on a $\mathcal{ELHI}$ knowledge base can be done in practice by exploiting the corresponding $\mathcal{ELHI}$-program $\mathcal{L}(\mathcal{F}(\mathcal{KB}))$. A query $Q(\mathbf{X})$ on $\mathcal{KB}$ can be rewritten in the rule $\mathcal{L}(Q(\mathbf{X}))$ as shown in the Table 3.1, and the following holds:

Theorem 3.8. *Let $\mathcal{KB}$ be a $\mathcal{ELHI}$ knowledge base, Q be a query on $\mathcal{KB}$, and $x_1, \ldots, x_n$ be constants occurring in $\mathcal{KB}$, then*

$$(x_1, \ldots, x_n) \in ans(Q(\mathbf{X}), \mathcal{KB}) \Leftrightarrow ans_Q(x_1, \ldots, x_k) \in LM((\mathcal{L}(\mathcal{F}(\mathcal{KB}))) \cup \mathcal{L}(Q)).$$

Proof. $ans_Q(x_1, \ldots, x_k) \notin LM((\mathcal{L}(\mathcal{F}(\mathcal{KB}))) \cup \mathcal{L}(Q(x_1, \ldots, x_k))) \Leftrightarrow$ the body of $(\mathcal{L}(Q(x_1, \ldots, x_k)))$ is not satisfied in any model of $\mathcal{L}(\mathcal{F}(\mathcal{KB})) \Leftrightarrow \mathcal{F}(\mathcal{KB}) \wedge \neg Q(x_1, \ldots, x_k)$ is satisfiable $\Leftrightarrow \mathcal{F}(\mathcal{KB}) \not\models Q(x_1, \ldots, x_k) \Leftrightarrow (x_1, \ldots, x_n) \notin ans(Q(\mathbf{X}), \mathcal{KB})$. ∎

Theorem 3.8 states that query answering on a knowledge base $\mathcal{KB}$ can be done via query answering on the corresponding $\mathcal{ELHI}$ program. Despite the possibly infinite size of $LM(\mathcal{L}(\mathcal{F}(\mathcal{KB})))$, we now show that query answering on $\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ can be done by considering only a finite subset of its ground instantiation. To this end we introduce the notions of *inclusion graph* and *existential depth*. In the following, we assume that a $\mathcal{ELHI}$ knowledge base $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$ is given.

Definition 3.9 (Inclusion Graph). *The Inclusion Graph (IG) of $\mathcal{KB}$ is a labeled directed graph $IG = (V, E)$ having a node in V for each concept/role occurring in $\mathcal{T}$, and:*

- *an arc (C, D, n) is in E for each axiom $C \sqsubseteq D$ in $\mathcal{T}$;*
- *arcs $(R, C, n), (B, C, n)$ are in E for each axiom $\exists R.B \sqsubseteq C$ in $\mathcal{T}$;*

- *arcs* $(B_1, C, n), (B_2, C, n)$ *are in* E *for each axiom* $B_1 \sqcap B_2 \sqsubseteq C$ *in* $\mathcal{T}$*;*
- *arcs* $(C, R, f), (C, B, f)$ *are in* E *for each axiom* $C \sqsubseteq \exists R.B$ *in* $\mathcal{T}$*.*

Let p *be a simple path in IG, the* f*-length of* p*, denoted by* $|p|$*, is the number of arcs* $e = (a, b, \ell)$ *of* p *such that* $\ell = f$*.*

Example 3.10. Let consider the knowledge base:

$$A \sqsubseteq \exists R.B \quad B \sqsubseteq \exists R.C \quad \exists R.C \sqsubseteq B$$

The corresponding inclusion graph $IG = (V, E, \ell)$ is as follows: $V = \{A, R, B, C\}$, $E = \{(A, R, f), (A, B, f), (B, R, f), (B, C, f), (R, B, n), (C, B, n)\}$. The *f-length* of $p =< (A, R, f), (R, B, n), (B, C, f) >$ is $|p| = 2$.

Definition 3.11 (Existential Depth). *Let IG be the Inclusion Graph of* $\mathcal{KB}$ *and,* E *be a concept or a role. The existential depth of* E *is defined as follows:*

$$ED(E) = \max_{p \in \Phi_E} |p|$$

where $\Phi_E = \{p \mid p$ *is a simple path in IG with* E *as ending node*$\}$*.*

Example 3.12. Consider the knowledge base of Example 3.10, the existential depth of concept C is $ED(C) = 2$.

The role played by the existential depth of a concept become clear in the following Theorem. We now show that instance retrieval of a concept/role can be done by computing only a finite portion of $LM(\mathcal{L}(\mathcal{F}(\mathcal{KB}))$, corresponding to $LM_0(\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ for a role R and corresponding to $LM_{ED(C)}(\mathcal{L}(\mathcal{F}(\mathcal{KB}))$ for a concept C.

Lemma 3.13. *Let* $P = \mathcal{L}(\mathcal{F}(\mathcal{KB}))$ *be a* $\mathcal{ELHI}$*-program, then:*

(1)if $R(t_1, t_2) \in LM(P)$ *with* $NL(R(t_1, t_2)) = 0$ *then* $R(t_1, t_2) \in LM_0(P)$*.*

(2)if $C(t) \in LM(P)$ *with* $NL(t) = 0$ *then* $C(t) \in LM_{ED(C)}(P)$*;*

Proof. (1) Since $NL(t_1) = NL(t_2) = 0$, the program defining $R(t_1, t_2)$ in $Ground(P, U_P)$ does not contain function symbols. From hypothesis $R(t_1, t_2) \in LM(P)$ it then holds that $R(t_1, t_2) \in LM_0(P)$.

(2) We proceed by induction, that is we demonstrate that the implication holds when $ED(C) = 0$ (base) and, then we show that the thesis follows for $ED(C) = n$ if the thesis holds for each $ED(C) < n$ (induction).
(base) If $ED(C) = 0$ then the program defining $C(t)$ in $Ground(P, U_P)$ does not contain function symbols, therefore $C(t) \in LM_0(P)$ and the thesis follows.
(induction) Assume that, for each concept E in $\mathcal{KB}$ such that $ED(E) = i$ and and $E(t) \in LM(P)$ it holds that $E(t) \in LM_i(P)$, for each $i < n$.
Suppose that $ED(C) = n$, $C(t) \in LM(P)$, and $C(t) \in LM_{n+1}(P)$ but $C(t) \notin LM_n(P)$.
Next, we show that in this case $ED(C)$ should be $n + 1$ contradicting the hypothesis.
Since $C(t) \in LM_{n+1}(P)$ and $C(t) \notin LM_n(P)$, there must be a rule $r \in Ground(P, U_{n+1})$ supporting $C(t)$ such that $LM_n(P) \not\models_{LP} B(r)$ and $LM_{n+1}(P) \models_{LP} B(r)$.
By Lemma 3.7, if we look at Table 3.1, r can be of the forms: (a) $C(t) \leftarrow E_1(t)$; or (b) $C(t) \leftarrow E_1(t), E_2(t)$; or (c) $C(t) \leftarrow R_1(t, t_1)$; or (d) $C(t) \leftarrow R_1(t, f_{\mathcal{A}}(t)), C_1(f_{\mathcal{A}}(t))$. Note that, both in case of rules of the form (a) and (b), it holds $E_j(t) \notin LM_n(P)$ (otherwise $C(t) \in LM_n(P)$, since $NL(E_j(t)) = NL(C(t)))$, $j = 1, 2$. Moreover, also in case of rules of the form (c), $R_1(t, t_1) \notin LM_n(P)$, from hypothesis $LM_n(P) \not\models_{LP} B(r)$.
Therefore, there must be a rule r_1 such that $LM_{n+1}(P) \models_{LP} B(r_1)$, which belongs to the program defining $C(t)$ in $Ground(P, U_{n+1})$, of the form:

$$r_1 : \bar{C}(t) \leftarrow R_1(t, f_{\mathcal{B}_1}(t)), C_1(f_{\mathcal{B}_1}(t)).$$

where $\bar{C}$ corresponds to a concept $\bar{C}$, which can either be C itself or $\bar{C}$ is subsumed by C in $\mathcal{KB}$ and $\bar{C}$ is defined by a set of axioms $\mathcal{C} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_z$ $(z \geq 1)$ of the form:

$$\mathcal{C}_i = \begin{cases} \{K_1 \sqsubseteq K_2\} \text{ or} \\ \{K_1 \sqcap K_2 \sqsubseteq K_3,\ K_1 \sqsubseteq K_2,\ K_1 \sqsubseteq K_2\} \end{cases}$$

Note that rule r_1 corresponds to the axiom $\mathcal{A}_1 := \exists R_1.C_1 \sqsubseteq \bar{C}$.[4] Since $LM_n(P) \not\models_{LP} B(r_1)$ then $R_1(t, f_{\mathcal{B}_1}(t)) \notin LM_n(P)$ or $C_1(f_{\mathcal{B}_1}(t)) \notin LM_n(P)$. We now show that $R_1(t, f_{\mathcal{B}_1}(t)) \in LM_n(P)$, thus $C_1(f_{\mathcal{B}_1}(t)) \notin LM_n(P)$. By

[4] Note that, r_1 can be of type (d).

looking at Table 3.1, we have that $R_1(t, f_{\mathcal{B}_1}(t))$ can be supported only if in $Ground(P, U_{n+1})$ there is a rule of the form:

$$re_1 : \bar{R}_1(t, f_{\mathcal{B}_1}(t)) \leftarrow Q_1(t)$$

where $\bar{R}_1$ denotes either R_1 itself or a relation $\bar{R}_1$ subsumed by R_1 trough a sequence of axioms in $\mathcal{KB}$ of the form $S' \sqsubseteq S''$. The rule re_1 corresponds to the axiom $\mathcal{B}_1 := Q_1 \sqsubseteq \exists \bar{R}_1.\bar{C}_1$ (note that function symbol $f_{\mathcal{B}_1}$ can be generated only by an axiom involving $\bar{C}_1$ of this form) and, thus $ED(Q) \leq ED(C) - 1 = n - 1$. Suppose that $\bar{R}_1(t, f_{\mathcal{B}_1}(t)) \notin LM_n(P)$ then by rule re_1, $Q_1(t) \notin LM_n(P)$, but, from the inductive hypothesis applied to $Q_1(t)$, we have that $\mathcal{KB} \not\models Q(t)$, therefore $Q(t) \notin M_g$, for each $g \geq n - 1$. Since rule re_1 is the only rule that allows for deriving $\bar{R}_1(t, f_{\mathcal{B}_1}(t))$, then it must be the case that $Q(t) \in LM_n(P)$, and also that $\bar{R}_1(t, f_{\mathcal{B}_1}(t)) \in LM_n(P)$. Since $R_1(t, f_{\mathcal{B}_1}(t)) \in LM_n(P)$, and $LM_n(P) \not\models_{LP} B(r_1)$ we have that $C_1(f_{\mathcal{B}_1}(t)) \notin LM_n(P)$.
Now, the same considerations made for $C(t)$ (i.e., there must be rule like r_1) can be done for $C_1(f_{\mathcal{B}_1}(t))$, and so there must be a rule $r_2 \in Ground(P, U_{n+1})$ of the form:

$$r_2 : \bar{C}_1(f_{\mathcal{B}_1}(t)) \leftarrow R_2(f_{\mathcal{B}_1}(t), f_{\mathcal{B}_2}(f_{\mathcal{B}_1}(t))), C_2(f_{\mathcal{B}_2}(f_{\mathcal{B}_1}(t)))$$

corresponding to the axiom $\mathcal{A}_2 := \exists R_2.C_2 \sqsubseteq \bar{C}_1$, such that
$R_2(f_{\mathcal{B}_1}(t), f_{\mathcal{B}_2}(f_{\mathcal{B}_1}(t))) \in LM_n(P)$ and $C_2(f_{\mathcal{B}_2}(f_{\mathcal{B}_1}(t))) \notin LM_n(P)$, and so on for each $1 < k \leq n$ there must a rule $r_k \in Ground(P, U_{n+1})$ of the form:

$$r_{k+1} : \bar{C}_k(t_k) \leftarrow R_{k+1}(t_k, f_{\mathcal{B}_{k+1}}(t_k)), C_{k+1}(f_{\mathcal{B}_{k+1}}(t_k))$$

where $t_k = f_{\mathcal{B}_k}(f_{\mathcal{B}_{k-1}}(\ldots(f_{\mathcal{B}_1}(t))$, corresponding to an axiom of the form $\mathcal{A}_k := \exists R_{k+1}.C_{k+1} \sqsubseteq \bar{C}_k$. Moreover, each $R_{k+1}(t_k,\ f_{k+1}(t_k))$ is derived from a rule of the form:

$$re_{k+1} : \bar{R}_{k+1}(t_k, f_{\mathcal{B}_{k+1}}(t_k)) \leftarrow Q_{k+1}(t_k)$$

corresponding to the axiom $\mathcal{B}_{k+1} := Q_{k+1} \sqsubseteq \exists R_{k+1}.\bar{C}_{k+1}$.
For each $k < n$ we have that $\bar{C}_k(t_k) \notin LM_n(P) \wedge R_{k+1}(t_k, f_{\mathcal{B}_{k+1}}(t_k)) \in LM_n(P) \wedge C_{k+1}(f_{\mathcal{B}_{k+1}}(t_k)) \notin LM_n(P)$; whereas, for $k = n$ since the atom $R_{n+1}(t_n, f_{\mathcal{B}_{n+1}}(t_n)) \notin LM_n(P)$ (having nesting level $n + 1$), then
$R_{\mathcal{B}_{n+1}}(t_n, f_{\mathcal{B}_{n+1}}(t_n))$ is derived in $Ground(P, U_{n+1})$ from rule re_{n+1}. Thus, by

construction of $\mathcal{KB}$ we have that $ED(C) = n + 1$, which contradicts the hypothesis. ∎

Theorem 3.14 (Instance Retrieval). *Let C be a concept, R a role, and t_1, t_2 be constants, then:*

1. $\langle t_1 \rangle \in ans(C(X), \mathcal{KB}) \Leftrightarrow C(t_1) \in LM_{ED(C)}$;
2. $\langle t_1, t_2 \rangle \in ans(R(X,Y), \mathcal{KB}) \Leftrightarrow R(t_1, t_2) \in LM_0$;

Proof. Thesis follows from Lemma 3.13 and Proposition 3.1.

The above result can be extended to general conjunctive queries.

Lemma 3.15. *Let $P = \mathcal{L}(\mathcal{F}(\mathcal{KB}))$ then:*

(1)if $C(t) \in LM(P)$ then $C(t) \in LM_n(P)$ where $n = NL(t) + ED(C)$;
(2)if $R(t_1, t_2) \in LM(P)$ then $R(t_1, t_2) \in LM_n(P)$ where $n \geq NL(R(t_1, t_2)) + ED(R)$.

Proof. Suppose that $C(t) \in LM(P)$, $ED(C) = n$, $NL(t) > 0$ and $C(t) \notin LM_{ED(C)+NL(t)-1}$ but $C(t) \in LM_{ED(C)+NL(t)}$; following analogous consideration done to prove Lemma 3.13, only a rule r_n of the following form can support $\bar{C}(t)$ in $Ground(P, U_{ED(C)+NL(t)})$:

$$r_n : \bar{C}_{n-1}(t_{n-1}) \leftarrow R_n(t_{n-1}, f_{\mathcal{B}_n}(t_{n-1})), C_n(f_{\mathcal{B}_n}(t_{n-1})).$$

here $t_{n-1} = f_{\mathcal{B}_{n-1}}(f_{\mathcal{B}_{n-2}}(\ldots(f_{\mathcal{B}_1}(t))$ with $NL(t_{n-1}) = n - 1 + NL(t)$, therefore $f_{\mathcal{B}_n}(t_{n-1}) \in U_{ED(C)+NL(t)}$ and $r_n \in Ground(P, U_{ED(C)+NL(t)})$. This means that $C(t) \in LM_{ED(C)+NL(t)}(P)$ if $C(t) \in LM(P)$.
(2) If $R(t_1, t_2) \in LM(P)$ and $NL(t_i) = 0$, $i = 1, 2$, $R(t_1, t_2) \in LM_0$ and $ED(R) = 0$ as shown in the Lemma 3.13. If $NL(t_i) > 0$, there must be a rule r_1, supporting $R(t_1, t_2)$, of the following form:

$$r_1 : \bar{R}_1(t, f_{\mathcal{A}}(t_k)) \leftarrow Q(t).$$

where $\bar{R}_1$ denotes either R itself or a relation $\bar{R}_1$ subsumed by R_1 trough a sequence of axioms in $\mathcal{KB}$ of the form $S' \sqsubseteq S''$. Therefore, $ED(R) = ED(Q) + 1$ from the Definition 3.11. Moreover, $Q(t) \in LM_k(P)$, where $k = ED(Q) + NL(t)$, from item (1). Therefore, the rules supporting $Q(t)$ are active in $Ground(P, U_k)$. Therefore,

Consequently,
hen $R(t_1, t_2) \in LM_n(P)$ where $n \geq k + ED(R)$, and $k = max\{NL(t_i)|i = 1, 2\}$.

Definition 3.16 (Level Mapping). *Let $Q := q(\mathbf{X}_0) \leftarrow q_1(\mathbf{X}_1), \ldots, q_n(\mathbf{X}_n)$ a query on an ontology $\mathcal{K}$ there is a level mapping $||\ ||_Q$ defined as follows:*

1. *for any $||q_i[p]|| = 0$, if $||q_i[p]|| \in \mathbf{X}_0$;*
2. *for any $||q_i[p']|| = ||q_j[p'']||$ if $q_i[p'] == q_j[p'']$;*
3. *for any $||q_i[p']|| \leq ||q_i[p'']||$ if $p' < p''$.*
4. *$||q_i|| = ||q_i[1]||$ if concept*
5. *$||q_i|| = ||q_i[2]||$ if role*

$||Q|| = max||q_i||$.

Lemma 3.17. *Let $\mathcal{K}$ be an $\mathcal{ELHI}$ knowledge base, R be a role in $\mathcal{K}$, and $ED(R) = m$. Therefore, for each atom $R(t_1, t_2) \in LM(\mathcal{L}(\mathcal{K}))$ such that $NL(R(t_1, t_2)) = n$ holds that:*

1. *$R(t_1, t_2) \in LM_{m-1}(\mathcal{L}(\mathcal{K}))$ iff $m > n$;*
2. *$R(t_1, t_2) \in LM_{m+n}(\mathcal{L}(\mathcal{K}))$ iff $m \leq n$.*

Proof. Next we denote $\mathcal{L}(\mathcal{K})$ by P and $LM_n(P)$ by M_n.
Note that, if $\mathcal{K} \models R(t_1, t_2)$ where $n = 0$ then $R(t_1, t_2) \in LM_0(P)$. Suppose that $n = 1$, $\mathcal{K} \models R(t, f(t))$, $R(t, f(t)) \in M_{m+1}$ and $R(t, f(t)) \notin M_m$. Since $R(t, f(t)) \in M_{m+1}$ holds, there exists a ground rule $r_1 \in Ground(P, U_{m+1})$ of the form:

$$r_1 : \bar{R}(t, f(t)) \leftarrow C_1(t)$$

where $\bar{R}$ denotes either R itself or a relation $\bar{R}$ subsumed by R trough a sequence of axioms in $\mathcal{K}$ of the form $S' \sqsubseteq S''$. Rule r_1 derives from an axiom of the form $\mathcal{A}_1 := C_1 \sqsubseteq \exists \bar{R}.C_2$. Therefore, from definition of the existential depth, $ED(C_1) = ED(R) - 1$. Therefore, $C_1(t) \in M_{m-1}$. Since $NL(t) = 0$ then $r_1 \in Ground(P, U_{m-1})$ and $R(t, f(t)) \in M_{m-1}$.
where $ED(C_1) = m - 1$

Lemma 3.18. *Let $\mathcal{K}$ be an $\mathcal{ELHI}$ knowledge base and C and R be a concept and a role in $\mathcal{K}$ then*

1. *If $\mathcal{K} \models C(t)$ then $C(t) \in LM_n(\mathcal{L}(\mathcal{K}))$ where $n = ED(C) + NL(C(t))$;*
2. *If $\mathcal{K} \models R(t_1, t_2)$ then $R(t_1, t_2) \in LM_n(\mathcal{L}(\mathcal{K}))$ where $n = ED(R) + NL(R(t_1, t_2))$.*

Proof. (1.) If $NL(t) = 0$ and $ED(C) = n$, $C(t) \in M_n$.
Suppose that the thesis holds for $NL(t) = m - 1$ and let $NL(t) = m$. From the construction of P_C, the following ground instance of the rules r_k are in the program $P_C \subseteq Ground(P, U)$ supporting $C(t)$:

$$r_{k+1} : \bar{C}_k(t_{m+k}) \leftarrow R_{k+1}(t_{m+k}, f_{m+k+1}(t_{m+k})), C_{k+1}(f_{m+k+1}(t_{m+k}))$$

where $NL(t_{m+k}) = m + k$, for each $k = 1, \ldots, n_1 - 1$. Therefore, $P_C \subseteq Ground(P, U_{n_1+m})$ and, since $C(t) \in M$ thus $C(t) \in M_{n+m}$.

Theorem 3.19 (Conjunctive Query Answering). *Let $\mathcal{K}$ be an $\mathcal{ELHI}$ knowledge base, $Q := q(\mathbf{X}_0) \leftarrow q_1(\mathbf{X}_1), \ldots, q_n(\mathbf{X}_n)$ be a conjunctive query, $m = max\{ED(q_i)|i = 1, \ldots, n\}$ and j the number of roles appearing in $B(Q)$ then*

$$\mathcal{K} \models q(c_0) \Leftrightarrow q(c_0) \in LM_k(\mathcal{L}(\mathcal{K})) \quad k = j + m$$

where $NL(c_0) = 0$.

Proof. ($\Leftarrow$) Obvious. ($\Rightarrow$) Since $\mathcal{K} \models q(c_0)$ then $\mathcal{K} \models q_1(c_1) \wedge \ldots \wedge q_n(c_n)$. If all q_i are concepts, then $NL(c_i) = 0$ for each $1 \leq i \leq n$.

If Q contains roles, then they are of the form $q_i(t, f(t))$ where $NL(t) = k$, $k \geq 0$.
Note that, if $ED(q_j) = i_1$ and $NL(c_j) = i_2$ then $q_j(c_j) \in LM_{i_1+i_2}$. Therefore, let $i_{max} = max_{i_1+i_2 \in \mathcal{K}})$ then $\{q_1(c_1) \ldots q_n(c_n)\} \in LM_{i_{max}}$

3.5 Complexity

Theorem 3.20. *It is given an integer k. For a concept or role CR appearing in a $\mathcal{ELHI}$ knowledge base $\mathcal{KB}$, deciding whether $ED(CR) > k$ is NP-complete.*

The above can be shown, e.g., by reduction to the Longest Path Problem (see [30], problem [GT23]). Note however, that, under data complexity regime, for $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$, we can assume $\mathcal{T}$ fixed and compute required values of $ED()$ once and for all. In the cases in which this computation is not desirable, note that the existential depth is bounded by the number Max of axiom of type $A \sqsubseteq \exists R.C$ belonging to $\mathcal{KB}$, and in virtue of Theorem regarding depth,

Theorem 3.21. *Given $\mathcal{ELHI}$ knowledge base $\mathcal{KB} = \langle \mathcal{T}, \mathcal{A} \rangle$, and an integer k, $G = grnd(\mathcal{L}(\mathcal{F}(\mathcal{KB}), U_k)$ has size $\mathcal{O}(|\mathcal{A}|^2 |\mathcal{T}| s^{2k+2})$ where s is the number of axioms of type $A \sqsubseteq \exists R.C$ in $\mathcal{T}$. G can be computed in* LOGSPACE *in the size of $\mathcal{A}$.*

Proof. As intermediate result, note that $P_T = \mathcal{L}(\mathcal{F}(\mathcal{T}))$ has size linearly proportional to the size of $\mathcal{T}$, while $P_A = \mathcal{L}(\mathcal{F}(\mathcal{A}))$ is at most a syntactical variant of $\mathcal{A}$; clearly, $Ground(P_A, U_k) = P_A$. Now, U_k consists of $|A| s^{k+1}$ different symbols while each rule $r \in P_T$ contains at most two variables ranging over U_k: the number of ground instances of r is thus in the worst case bounded by $(|A| s^{k+1})^2$. G can be easily generated by maintaining a fixed number of log-space counters ranging over elements of U_k and rules of P_T. It is also easy to see that one can avoid storing P_T and P_L as intermediate byproducts of the computation (or, standard composition techniques for LOGSPACE algorithms can be applied, see e.g. [53]). Hence the result.

3.6 System Prototyping

In this section we show how we managed fast prototype a proof of concept system. More details will follow later, about the implementation issues and solutions. The workflow of this latter is the following:
an input ABox A and a TBox T are given in OWL format; the DLVHEX system [23] takes in input: A and T in form of a triple stream, and a logic program S tailored at converting A and T into set of logic facts A' and T'. In order to obtain $L_T = \mathcal{L}(\mathcal{F}(T))$, we exploit the DLT system [1] which takes in input T', a set of meta-axioms S and outputs L_T. Meta-axioms are expressed using the higher order syntax accepted by DLT, such as, for instance
$S(Y, X) \leftarrow R(X, Y), inverseOf(R, S)$@ont.
which, when instantiated over actual role names r and s, produce the logic program rule $s(Y, X) \leftarrow r(X, Y)$.
L_T and A' can be then used by a logic programming solver such as DLV [42]. This latter system allows to generate $G_T = Ground(L_T \cup A', U_k)$ for a fixed k. k can be determined in terms of the maximum existential depth of T and query length. $LM_k(G_T)$ can be then generated and stored for subsequent fast query answering.

In alternative, for a given query Q, it is possible to apply the magic set transformation $M(L_T, Q)$ [5] on L_T and Q, and compute $LM(ground(M(L_T \cup A', Q), U_k))$.

3.7 Remarks and Related Work

The approach shown in this chapter has clear points of contact with *a)* former research on query answering over different supersets of $\mathcal{EL}$ [3, 47, 55]; also, $\mathcal{ELHI}$-programs have a remarkable connection with *b)* other attempts of identifying fragments of logic programming whose presence of function terms does not affect decidability of querying [15, 9, 27, 26] and with *c)* reasoning with chase techniques [14, 63, 64].

As for the first category, this paper tackles the issue of query answering on $\mathcal{EL}$ from a different perspective: with respect to [55], we deal directly with Skolemized logic programs and we do not require a function symbols elimination step. For what implementation is concerned, this allows the elimination of an intermediate resolution module: obtained $\mathcal{ELHI}$-programs can be directly piped towards current logic programming engines with little or no modification at all. Similarly with [47, 3], our approach can be seen as one exploiting the property of *combined FO rewritability* which $\mathcal{ELHI}$ knowledge bases enjoy. Note that in [47] it is suggested to permanently store a pre-computed *canonical model* $\mathcal{I}_\mathcal{K}$ of the knowledge base $\mathcal{K}$ at hand. Answering a query q by checking whether $\mathcal{I}_\mathcal{K} \models q$ is not sound however, thus q is properly rewritten into a query $q_\mathcal{K}$ in order to regain soundness. Similarly, we can opt for storing implicitly the model $L_k = LM_k(\mathcal{L}(\mathcal{F}(\mathcal{K})))$ for k a bound depending on the existential depth of $\mathcal{K}$ and the maximum allowed size of queries. The size of L_k is comparable to $|\mathcal{I}_\mathcal{K}|$): also, queries do not require rewriting and can be directly answered (in AC_0 data complexity [7]) over L_k. Furthermore, we can opt for storing $\mathcal{L}(\mathcal{F}(\mathcal{K}))$ only (whose size is linear in the size of $\mathcal{K}$) and perform on demand reasoning on a query per query basis.

As for point *b)* above, it is worth noting that recently much work has been devoted to the identification of classes of logic programs, mostly under answer set semantics, for which at least some form of reasoning is decidable in spite of the presence of function symbols. Example of this research are finitely-ground

(FG) programs [15] and finitely recursive programs [9]. Given the relationship between existential restrictions and function symbols, often such kind of research has been partially inspired by description logics, such as in the case of $\mathbb{FDNC}$ programs [27] and bidirectional programs [26]. For space reason we cannot report the formal definition of the abovementioned classes and we remaind the reader to the corresponding papers: it is however worth remarking that $\mathcal{ELHI}$ programs are not directly comparable with any of the above classes, thus constituting a new fragment of logic programming for which query answering is proven to be decidable.

Theorem 3.22. *The following hold:*

1. $P \in ELHI \not\Rightarrow P \in \mathcal{FG}$;
2. $P \in ELHI \not\Rightarrow P \in \mathbb{FDNC}$;
3. $P \in ELHI \not\Rightarrow P$ *is bidirectional;*
4. $P \in ELHI \not\Rightarrow P$ *is finitely recursive.*

Proof. Proof can be easily given by counterexamples: note that the $\mathcal{ELHI}$ program of Example 3.5 is not $\mathcal{FG}$ and also not $\mathbb{FDNC}$. The $\mathcal{ELHI}$ program $\{r_1 : R(X, f_{\mathcal{A}}(X)) \leftarrow A(X)., r_2 : B(f_{\mathcal{A}}(X)) \leftarrow A(X)., f : A(a).\}$ is not bidirectional, while the $\mathcal{ELHI}$ program $\{r_1 : B(X) \leftarrow A(Y), R(X,Y). f : A(a).\}$ is not finitely recursive.

Eventually, it is worth noting that our work has relationship with chase techniques used in the relational database field: it can be seen that $\mathcal{ELHI}$-programs can be ported to equivalent Guarded Tuple Generating Dependencies [14]; also, it has been shown how chase can be applied for answering conjunctive queries under $\mathcal{EL}$ [63]: as a remarkable difference, note that chase rules require a specific order of application, and they can not be straightforwardly specified in a declarative way. In [64] it is identified a bound in the number of Skolem terms necessary for building a finite chase in the case of databases with inclusion dependencies (these latter include as a special case $\mathcal{EL}$ existential restrictions). Such a bound depends on the number of atoms and occurrences of existential variables in the query at hand and on the number of attributes affected by inclusion dependencies; note that our similar notion of *existential depth* depends also on how existential restrictions are structurally related in TBoxes, and might enforce in practical cases a smaller bound. It is matter of future research to investigate about relationship among the two notions.

4

Implementation and Testing

4.1 System Prototype

To implement the aforementioned techniques, we have realized a system prototype, which relies on several technologies. Next we will give a short descriptions of the main components of such system, pointing out the particular features it has been given to better fulfill its end.

The input to the system is an Ontology O, and a query Q over such ontology,

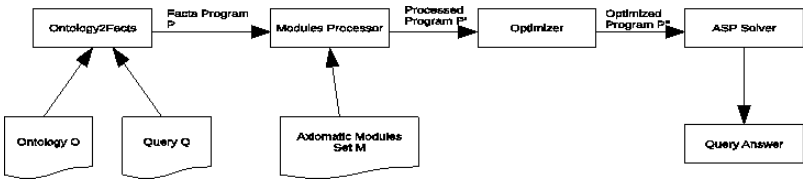

Fig. 4.1: The System Prototype

which in general is a conjunctive query. The ontology must obey the rules and constraints of the description logics in use. In our case that is $\mathcal{ELHI}$, which we described in detail earlier in this thesis.

The system consists of various components:

- Ontology2Facts
- Modules Processor
- Optimizer
- Answer Set Solver

Next we will describe them in detail.

4.1.1 the "Ontology2Facts" component

The original ontology, as well as the query, is usually expressed in RDF-like syntax (and structure). Actually it is a sequence of triples, describing concepts, roles and instances. This component performs the translation of the ontology and query into facts. To this end, it employs the power of dlvhex. In particular, we have used an external atom called *rdf*, the function of which is to retrieve ontology data from an URL (the ontology URL) and inject it in a normal logic atom. This

The so generated atoms are in turn converted into simple facts, named after the keywords of the ontology language, i.e. the names used to recognize a class from an object, etc. For example, if we give the system an RDF file corresponding to the following ontology:

$$\exists r.d_1 \sqsubseteq d_2$$
$$s \sqsubseteq r$$
$$d1 \sqsubseteq d_2$$

The system returns the following higher order facts program:

$$F_{HEX}:$$
$$existsInConcept(r, d_1, d_2).$$
$$subPropertyOf(s, r).$$
$$subClassOf(d_1, d_2).$$

This is realized using two separate modules: *F*act Translator and *T*ranslator, which we report:

FactTranslator:It transforms the ABox in unary and binary facts.

$$triple(X, Y, Z) \leftarrow \&rdf[\texttt{URL}](X, Y, Z).$$
$$C(X) \leftarrow triple(X, \text{“}rdf : type\text{”}, C), C\,! = \text{“}owl : Ontology\text{”}.$$
$$P(X, Y) \leftarrow triple(X, P, Y), P! = \text{“}rdf : type\text{”}, P\,! = \text{“}owl : imports\text{”}.$$

Translator:It Transforms the TBox in a set of facts, according to what is found.

```
triple(X, Y, Z) ← &rdf[URL](X, Y, Z).
subPropertyOf(P, Q)          ← triple(P, "rdfs : subPropertyOf", Q).
subClassOf(C, D)             ← triple(C, "rdfs : subClassOf", D).
range(P, R)                  ← triple(P, "rdfs : range", R).
domain(P, D)                 ← triple(P, "rdfs : domain", D).
label(P, L)                  ← triple(P, "rdfs : label", L).
type(O, T)                   ← triple(O, "rdf : type", T).
inverseOf(P, R)              ← triple(P, "owl : inverseOf", R).

someValuesFrom(B, R, C)      ← triple(B, "owl : someValuesFrom", C),
                               triple(B, "owl : onProperty", R),
                               type(B, "owl : Restriction").

class(C)                     ← triple(C, "rdf : type", "owl : Class").
property(P)                  ← triple(P, "rdf : type", "owl : ObjectProperty").
property(P)                  ← triple(P, "rdf : type", "owl : DatatypeProperty").

intersectionOf(A, B, C)      ← triple(A, "owl : intersectionOf", L),
                               triple(L, "rdf : first", B), triple(L, "rdf : rest", C).
intersectionOf(C, D, E)      ← intersectionOf(A, B, C), triple(C, "rdf : first", D),
                               triple(C, "rdf : rest", E), E! = "rdf : nil".
equivalentClass(C, D)        ← intersectionOf(A, B, C), triple(C, "rdf : first", D),
                               triple(C, "rdf : rest", "rdf : nil").
complementOf(C, D)           ← triple(C, "owl : complementOf", D).

transitive(R)                ← triple(R, "rdf : type", "owl : TransitiveProperty").
```

4.1.2 Modules Processor

The end of this component is to combine the facts generated with the semantic rules defining the description logic fragment in use. For this reason, we need a ruleset for each constructors of $\mathcal{ELHI}$. This component relies on the higher-order capabilities of ASP.

We report next the various axiomatic modules necessary for $\mathcal{ELHI}$. They correspond to the ones seen in 3, but have been adapted to use the keywords of the OWL2EL language. This is necessary since real-word ontologies are written in such languages. The keyword "someValuesFrom" corresponds to $\exists$; the inverse roles have been simulated with two rules which state that if r_1 is the inverse of r_2, it must also be true that r_2 is the inverse of r_1.

$$
\begin{array}{ll}
D(X) & \leftarrow C(X),\ subClassOf(C,D). \\
R(X,f(ERC,X)) & \leftarrow ERC(X),\ someValuesFrom(ERC,R,C). \\
C(f(ERC,X)) & \leftarrow ERC(X),\ someValuesFrom(ERC,R,C). \\
D(X) & \leftarrow R(X,Y), C(Y),\ existsInConcept(R,C,D). \\
R_1(X,Y) & \leftarrow R_2(Y,X),\ inverseOf(R_1,R_2). \\
R_2(X,Y) & \leftarrow R_1(Y,X),\ inverseOf(R_1,R_2). \\
C_2(X) & \leftarrow C_1(X),\ equivalentClass(C_1,C_2). \\
C_1(X) & \leftarrow C_2(X),\ equivalentClass(C_1,C_2). \\
R(X,Y) & \leftarrow S(X,Y), subPropertyOf(S,R).
\end{array}
\tag{4.1}
$$

4.1.3 The Optimizer: The Magic Set Rewriting Technique

The Optimizer module aims at reducing the size of the instantiation of the program. To this end, it employs the well known Magic Sets Rewriting Technique, originally defined in [5]. In the following, we will thoroughly describe this technique, and in particular the modifications we have performed to make it suitable for our needs.

Introduction

The Magic Sets rewriting technique takes a significant place in the literature about logic programming and deductive database systems, since its early definition.

Given a logic program P and a query Q over its vocabulary, this technique consists in rewriting P with respect to Q, by adding some predicates and some newly created rules: these latter are introduced in order to simulate the top-down computation of the program. By using Magic Sets it is possible to reduce the amount of unnecessary computation, due to portions of the ground version of P which cannot alter the answer to Q, but are however evaluated if a pure bottom-up scheme is used. Many extensions and modifications of the base technique have been proposed in literature, aimed at improving or extending it to more specific cases. Among them, we mention here the extensions to disjunctive logic programs in [20, 32], and the one realized for programs with (possibly unstratified) negation in [28]. In this paper we focus our attention on positive disjunctive logic programs with function symbols, applying the magic set technique to this kind of programs. Some particular issues arise when considering

this language, due to the presence of function symbols along with disjunction. The main contributions of this work are: (i) we extend the magic set technique to the case of positive disjunctive programs with function symbols by devising an appropriate transformation algorithm; (ii) we give an implementation of the algorithm, and we show how it works by example.

We define the following entailment notion with respect to an interpretation I. For a a ground atom: $I \models a$ iff $a \in I$; For $a_1, \ldots, a_n$ ground atoms:
$I \models a_1, \ldots, a_n$ iff $I \models a_i$, for each $1 \leq i \leq n$; $I \models a_1 \vee \cdots \vee a_n$ iff $I \models a_i$ for at least one i, $1 \leq i \leq n$. For a rule r: $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$;
A *model* for P is an interpretation M for P such that every rule $r \in grnd(P)$ is such that $M \models r$. A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M. The set of all minimal models for P is denoted by MM(P).
An interpretation I for a program P is an *answer set* for P if $I \in$ MM(P) (i. e., I is a minimal model for the positive program P). The set of all answer sets for P is denoted by $ans(P)$. We say that $P \models a$ for an atom a, if $M \models a$ for all $M \in ans(P)$.

Informal Overview

The *Magic Sets* rewriting technique consists of a simulation of the top-down evaluation of a query Q by modifying an original program P and producing a rewritten program $M(P, Q)$ which comprises additional rules, and updates to the original ones. $M(P, Q)$ is conceived in order to reduce computation to what is actually relevant for answering the query. In fact, $grnd(P)$ contains, in general, many ground rules that have no impact in answering Q as they are related to atoms which Q does not depends on. In general, it is expected that $grnd(M(P, Q))$ has smaller size than $grnd(P)$.
The original magic sets method was first described in [5] for the case of Datalog, i. e. logic programs without function symbols. Following work considered the presence of functional terms, yet not explicitly taking disjunction also into account (see e.g. []). Concerning the stable model semantics, it is known how to apply this rewriting technique to Datalog Programs with disjunction [20, 32] and also (with some restricting assumption) to unstratified programs [28].

To give an intuition about the general magic set technique for Datalog programs, we can consider the following (traditional) example. Let us consider the query $Q = path(1, 5)?$ on the following program P_1:

$path(X, Y)$:-$edge(X, Y)$. $path(X, Y)$:-$edge(X, Z), path(Z, Y)$.

As a first step, head predicates are "adorned". Basically, we simulate the top-down computation and annotate the way how the variable bindings are propagated from the head atom to body atoms. Each rule of the input program is replaced by an "adorned" one in which the name of each predicate is modified by appending the binding information.
Given an IDB predicate, we denote a bound argument with the b letter, while a free one is labeled with f. For instance $path^{bf}$ is a predicate which is in principle a subset of $path$: in particular its first argument is restricted to a set of values (the *magic set* of $path^{bf}$) which is usually much smaller than the range of $path$ on its first argument. The adornment process starts from the query Q. This latter is adorned in a very simple manner: all constants in the query become bound, all variables are marked as free (we obtain in this case the predicate $path^{bb}$). Adornment is propagated to rules' heads in which $path$ appears, and subsequently from the head to the body. If a new adorned predicate is created (as it is present in the head or the body of the rule), this is processed in turn in the same way of the original adorned query, until no more adorned predicates have to be processed. SIPs (Sideways Information Passing Strategies) are used in order to establish the adornment policy.
In our example, the arguments of the given query are both constants,and thus bound; we will build the adorned program according to $path^{bb}$:

Note that EDB predicates are excluded from adornment. The next step of the transformation consists in generating magic rules starting from the adorned program. These rules define *magic predicates*. A *magic predicate* defines the allowed range of values for bound arguments of a predicate. We start from the head of the rule.
Given an adorned head atom $a(\mathbf{t})$, we obtain the set of terms $\mathbf{t}'$, derived from $\mathbf{t}$ by removing all the terms corresponding to free arguments, and generate the magic atom $magic_a(\mathbf{t}')$. Then, for each atom b in the body, we create its magic version $magic_b(\dots)$. Subsequently, we generate a magic rule having $magic_b$ in

the head and $magic_a$ in the body, followed by all the atoms of the adorned rule which can propagate the binding.
The third step consists of the modification of the adorned rules. In this step we add to the bodies of the rules the magic atoms which have been generated in the previous step. For each rule with head h, an according magic atom $magic_h$ is inserted in the body of the rule.

$magic\_path^{bb}(1,5).\ magic\_path^{bb}(Z,Y)$:- $magic\_path^{bb}(X,Y), edge(X,Z).$
$path(X,Y)$:- $magic\_path^{bb}(X,Y), edge(X,Y).$
$path(X,Y)$:- $magic\_path^{bb}(X,Y),\ edge(X,Z),\ path(Z,Y).$

Finally, in the last step the query is processed by adding a magic fact $magic\_q\_ad$ if q is the query and ad its adornment; In our example we add $magic\_path^{bb}(1,5)$.
The resulting program is then evaluated w.r.t. the query.

Magic Sets for DLP with Function Symbols

Here we present an improved Magic Set technique. It is designed to be able to deal with programs containing both disjunction and functions symbols. Even if our programs do not contain disjunction, actually, we present the general technique, assuming the non-disjunctive programs as a special case of.
The algorithm is sketched in Figure 4.2. The main procedure is called **magify**. The function **magify** takes a program P and a query Q as input, and applies the Magic Sets Transformation, generating $M(P,Q)$ (the *magified program*). **magify** is made of other subprocedures, detailed in the following. Let us assume it is given the query: $Q_2 = a(f(1))?$ and the program P_2:

$r1 : a(X) \vee b(X)$:- $c(X), e(X).\ r2 : c(f(X))$:- $c(X).\ r3 : e(1).\ r4 : c(1).$

When a query is conjunctive, it is transformed into a rule, having in the head a new atom which contains all the variables from the atoms in the original query. The original query is replaced by a new one which consists of the head of the newly created rule. This procedure is performed by the function **normalizeQuery(Query Q)**.
The next step consists of creating the adorned program **AP**, by means of the function **createAdornedProgram(Program P, Query Q)**, reported in Fig-

```
Program magify(Program P, Query Q)
{
    Program M(P,Q)=∅;
        if(Q.isconjunctive())
        {
            Rule R',Query Q';
            (Q',R') = normalizeQuery(Q);
            P.addRule(R');
            Q=Q';
        }
    Program AP = createAdornedProgram(P,Q);
    Program MR = createMagicRules(AP);
    Program MP = addMagicAtoms(P);
    Fact MF = createMagicFact(Q);
    M(P,Q)=removeAdornments(MP∪MR∪MF);
    return M(P,Q);
}
```

Fig. 4.2: Function createAdornedProgram

```
Program createAdornedProgram( Program P,
Query Q )
{
    Stack S = ∅; Program AP = ∅;
    S.push(createAdornedVersionOf(Q));
    while(S. size > 0)
        {
            Atom x= S.pop();
            for(Rule r in P)
            Rules adornedRules = adornRule(r,x);
            AP.add(adornedRules);setDone(X);
            for(Rule ar in adornedRules)
                for(Atom a in ar)
                    if(!done(a)) S.push(a);
        }
    return AP;
}
```

Fig. 4.3: Function createAdornedProgram

ure 4.3. A stack S is used in order to keep the atoms scheduled for adornment. The query is adorned using the function **createAdornedVersionOf** and pushed in S at first. The main cycle pops out from S a given atom a and accordingly adorns each rule having in the head an atom whose name matches with it. When a certain adornment is generated for the first time for a predicate, this is pushed into S, in order to be processed. The algorithm iterates until S is empty.

The adornment of each rule is actually performed by the inner function **adornRule**(r, x) which returns a set of adorned rules according to the labels of x, to be added to the adorned program. If x is not in the head of r, **adornRule**

returns an empty set. More in detail, the output of **adornRule** contains a set R' of adorned rules for each atom $x' \in H(r)$ which unifies with x. Each $r' \in R'$ is built according to the following strategy: per each $x' \in H(r)$ which unifies with x, x' is labeled according to x, then such labelling is propagated to $B(r)$, according to a SIP. Successively, adornments are propagates from $B(r)$ to the remaining head atoms. Moreover, from the obtained adorned disjunctive rule r', corresponding to x', we obtain $|H(r)|-1$ auxiliary rules obtained by leaving in the head only one atom $x \in H(r) \setminus \{x'\}$ and having $B(r) \cup (H(r) \setminus x)$ as body. The obtained set of auxiliary rules in AP will not take part in the final program $M(P,Q)$, but will be further processed in order to obtain the set of magic rules MR. In turn, magic rules are created, according with the traditional strategy, by calling the **CreateMagicRules** function. In our example, we get first from rule $r1$ and $r2$, the adorned versions $r1' : a^b(X) \vee b^b(X) \text{:-} c^b(X), e(X)$ and $r2' : c^b(f(X)) \text{:-} c^b(X)$ then **createMagicRules(Program P)** obtains from $r1'$ and $r2'$ the corresponding magic rules; and from $r1'$ we get the two rules: $a^b(X) \text{:-} c^b(X), e(X), b^b(X).$ $b^b(X) \text{:-} c^b(X), e(X), a^b(X).$, while $r2'$ is left unchanged.

Now the function **createMagicRules** simply applies the normal Magic-Set strategy to these intermediate rules, as seen in previous section. In our example we obtain:

$magic\_c^b(X) \text{:-} magic\_a^b(X), e(X), b^b(X).$
$magic\_c^b(X) \text{:-} magic\_b^b(X), e(X), a^b(X).$
$magic\_c^b(X) \text{:-} magic\_c^b(f(X)).$

The third rule has been obtained by applying the algorithm for the non disjunctive case.

Now, the function **addMagicAtoms**(P) is called, which returns a version of **P** including magic predicates within the body of each rule of **P**. In this simple step, for each atom in the head of the rule the corresponding magic atoms are added in the body. Successively, a magic atom from the query is generated by the function **createMagicFact(Query Q))** to be added to the final output. In our example we get: $magic\_a^b(f(1))$. Finally, the function call **removeAdornments**$(MP \cup MR \cup MF)$ removes all adornments from the non-magic predicates. This is necessary as stated in [20]. The final output for our

example is the following:

$magic\_c^b(X)$:- $magic\_a^b(X), e(X), b^b(X)$.
$magic\_c^b(X)$:- $magic\_b^b(X), e(X), a^b(X)$.
$magic\_c^b(X)$:- $magic\_c^b(f(X))$. $magic\_a^b(f(1))$.
$a(X) \vee b(X)$:- $c(X), e(X), magic\_a^b(X), magic\_b^b(X)$.
$c(f(X))$:- $c(X), magic\_c^b(f(X))$.

It must be noted here that two aspects of the class of programs we are treating, disjunction and the presence of function symbols, need a particular treatment. In particular:

Disjunction

requires modifications on the adornment strategy. Let r be a rule of the form:

$$r1 : h_1(t_1) \vee \ldots \vee h_n(t_n) \text{:-} b_1(p_1), \ldots, b_m(p_m).$$

If we adorn the rule w.r.t. the atom $h_i(t_i)$, also other head atoms have to be taken into consideration, because they can contain variables which are actually important for the evaluation. The function acts as follows:(i) the atom $h_i(t_i)$ is adorned w.r.t. the query; (ii) the body is adorned w.r.t. the adornments of $h_i(t_i)$ by using a suitable SIP; (iii) other head atoms $h_1(t_1) \vee \ldots \vee h_{i-1}(t_{i-1}) \vee h_{i+1}(t_{i+1}) \vee \ldots \vee h_n(t_n)$ are adorned w.r.t. patterns found in the body.
In fact, it has been shown in [32] that if we want to keep the algorithm sound, other head predicates cannot propagate bindings, but can only receive them. In this case bindings are propagated from the body to the remaining head atoms.

Function symbols

have impact on the choice of the labelling for arguments: Given an atom a(. . . , t, . . .) for t a functional term t , the corresponding argument of a is labelled as bound iff all the subterms of t are set as bound at the moment of adornment of a.
Remark. Our transformation applies to programs with function symbols, thus, in general, an evaluation of the $M(P, Q)$ is not guaranteed to terminate. However, there are language restrictions that ensures termination, for instance see [16].

Implementation Notes

The prototype has been implemented in the Java programming language as a preprocessor able to generate a magified program $M(P, Q)$ compatible with the DLV input format [42] from a given program P and a, possibly conjunctive, query Q. The system uses a new Library, called DLVParser, which contains a full framework of classes useful for both the parsing and the manipulation of a Disjunctive Logic Programs in standard syntax.
Design patterns have been used, in order to keep the system flexible and easily extensible. In particular, the *S*trategy pattern has been used for allowing the implementation of multiple SIPs, so that the user of the API of our system is allowed to define his own strategy. To define a new SIP, only a few methods have to be implemented. We have implemented a default SIP, which mimics the propagation of bindings in the Prolog SLD resolution. Inclusion of other constructs such as negation and constraints are forthcoming.

4.1.4 The Solver

The solver is nothing else than the DLV system. It uses the last version of DLV, called DLV-complex, which has interesting properties. In fact, it exploits the higher-order capabilities of such system, since the program to evaluate is higher-order (we do not eliminate function symbols, as other systems do).
The submitted program is evaluated, and the answer set is generated. It is only one answer set because the original program does not contain disjunction in the head, as the chosen DL-fragment has no need for it.
The answer set represent the answer to the query, in term of Tuples.

4.2 Experimental Results

4.2.1 The Leigh University Benchmark (LUBM)

The Leigh University Benchmark [33] is a suite created for testing purposes. It provides the user an ontology of a University, called Univ-Bench. Univ-Bench describes universities and departments and the activities that occur at them. Its predecessor is the Univ1.0 ontology1, which has been used to describe data about actual universities and departments.

The authors created an OWL version of the Univ-Bench ontology. The ontology is expressed in OWL Lite, the simplest sublanguage of OWL.
To keep the ontology suitable for our needs, some constructs have been disabled, as $\mathcal{ELHI}$ is less expressive than OWL Lite. Test data of the LUBM are extensional data created over the Univ-Bench ontology. For the LUBM, it is available a method of synthetic data generation. This serves multiple purposes. Data generation is carried out by UBA (Univ-Bench Artificial data generator), a tool developed for the benchmark. The support for OWL datasets in the tool has been implemented. The generator features random and repeatable data generation. A university is the minimum unit of data generation, and for each university, a set of OWL files describing its departments are generated. Instances of both classes and properties are randomly decided. To make the data as realistic as possible, some restrictions are applied based on common sense and domain investigation. Examples are "a minimum of 15 and a maximum of 25 departments in each university", "an undergraduate stu- dent/faculty ratio between 8 and 14 inclusive", "each graduate student takes at least 1 but at most 3 courses", and so forth. A detailed profile of the data generated by the tool can be found on the benchmark?s webpage.
The generator identifies universities by assigning them zero-based indexes, i.e., the first university is named University0, and so on. Data generated by the tool are exactly repeatable with respect to universities. This is possible because the tool allows the user to enter an initial seed for the random number generator that is used in the data genera- tion process. Through the tool, we may specify how many and which universities to generate.
Finally, as with the Univ-Bench ontology, the OWL data created by the generator are also in the OWL Lite sublanguage.

4.2.2 Tests run

We have used the UBA to generate universities knowledge bases of various sizes.
In the following, we will indicate with"$Lubm_X$", with X a positive integer, the test set generated, which comprises X Universities, obviously connected to each other.
The Lubm Testsuite delivers 14 queries, but for our tests we decided to use only 3 of them, to better focus on the analysis of results.
The chosen queries are the following:

Query1:

```
(type GraduateStudent ?X)
(takesCourse ?X http://www.Department0.University0.edu/GraduateCourse0)
```

- This query bears large input and high selectivity. It queries about just one class and one property and does not assume any hierarchy information or inference.

Query7

```
(type Student ?X)
(type Course ?Y)
(teacherOf http://www.Department0.University0.edu/AssociateProfessor0 ?Y)
(takesCourse ?X ?Y)
```

This query is similar to Query 6 in terms of class Student but it increases in the number of classes and properties and its selectivity is high.

Query10

```
(type Student ?X)
(takesCourse ?X http://www.Department0.University0.edu/GraduateCourse0)
```

This query differs from Query 6, 7, 8 and 9 in that it only requires the (implicit) subClassOf relationship between GraduateStudent and Student, i.e., subClassOf relationship between UndergraduateStudent and Student does not add to the results.

We tested our reasoner against the famous emph Pellet Reasoner [67]. The test methodology is the following: We submit the queries to the reasoner, and take the timestamps. Then, we subtract the parsing time from the total time, because parsing could be done only once in a realistic scenario.
To check the effectiveness of the optimization technique (the Magic Sets), we used two versions of the reasoner, one of which does not exploit such technique. For what it concerns Pellet, again we do not include the parsing time in the timestamps reported, and set the dimensions of the Java Heap to the maximum available on the test machine.
All the tests have been run on a Apple MacPro machine, which is a Xeon-based multicore machine, with 8 Gigabytes of Ram memory. It runs Debian Gnu-Linux, in the 64-bit flavor.

The results shown above are very interesting. First of all, it is necessary to point out that Pellet resulted unable to handle a LUBM greater than 10. All

	Solver	Solver No-Magic	Pellet
Query 1	0.15	0.16	0.12
Query 7	0.73	0.74	0.1
Query 10	0.14	0.73	0.01

Table 4.1: Tests run on LUBM1

	Solver	Solver No-Magic	Pellet
Query 1	0.47	0.5	0.5
Query 7	2.54	2.78	0.74
Query 10	1.31	1.05	0.23

Table 4.2: Tests run on LUBM10

	Solver	Solver No-Magic	Pellet
Query 1	1.54	1.67	fail
Query 7	8.43	9.12	fail
Query 10	4.38	3.94	fail

Table 4.3: Tests run on LUBM30

	Solver	Solver No-Magic	Pellet
Query 1	3.2	3.57	fail
Query 7	20.51	22.76	fail
Query 10	8.17	7.39	fail

Table 4.4: Tests run on LUBM45

the queries resulted in a OutOfMemory exception, for any heap size chosen.This is probably due to the fact that Pellet stores all the data in memory, without using any streaming technique.

In contrast, our prototype just loads data on the fly, keeping a small portion of the program in memory at once. This results in the capacity of handling programs (and ABoxes) of any size. In case of small sizes, Pellet performs slightly better than our reasoner.

It is interesting that the Magic Sets rewriting technique works very well for the first two queries, but not on the third. This shows that the technique itself is

valid, but the results also depend on the Sideways Information Passing (SIP) used.

Part II

Axiomatization Techniques

5

Integrating Frame Logic in Answer Set Programming

5.1 Introduction

In this chapter we aim at closing the gap between F-logic based languages and Answer Set Programming, in both directions: on one hand, Answer Set Programming misses the useful F-logic syntax, its higher order reasoning capabilities, and the possibility to focus knowledge representation on objects, more than on predicates. On the other hand, manipulating F-logic ontologies under stable model semantics opens a variety of modeling possibilities, given the higher expressiveness of the latter with respect to well-founded semantics.
Our approach is set in between a pure model theoretic semantics (proper of F-logic and many of its extensions [39, 71]), and a pure “rewriting" semantics, in which inheritance is specified by means of an ad-hoc translation to logic programming [36]. More details on F-Logic may be found in the Preliminaries Chapter 2 In the former case, semantics is given in a clean and sound manner: however, the way inheritance (and in general, the semantics of the language) is modeled is hardwired within the logic language at hand, and cannot be easy subject of modifications. In the latter case, semantics is enforced by describing a rewriting algorithm from theories to appropriate logic programs. In such a setting the semantics of the overall language can be better tuned by changing the rewriting strategy. It is however necessary to have knowledge of internal details about how the language is mapped to logic programming, making the process of designing semantics cumbersome and virtually reserved to the authors of the language only.
Here we define a basic stable model semantics for FAS programs which does not purposely fix a special meaning for the traditional operators of F-logic, such

as class membership ":" and subclass containment "::". Indeed, FAS programs are conceived as a test-bed on which an advanced ontology designer is allowed to choose the behavior of available operators from a predefined library, or to design her own semantics from scratch. The ability to customize the semantics of the language is crucial especially in presence of inheritance constructs. In fact, when one has to model a particular problem, a specific semantics for inheritance may be more suitable than another, and it is often necessary to manipulate and/or combine the predefined behaviors of the language.
The topics we focus in the following of this chapter are:

1. We present the family of Frame Answer Set Programs (FAS programs), allowing usage of frame-like constructs, and of higher order atoms. Interestingly, positively *nested frames* may appear both in the head and in the body of rules. The language allows to reason in multiple *contexts* which are called *framespaces*.
2. We provide the model-theoretic semantics of FAS programs in terms of their *answer sets*.
3. We show how semantics features can be introduced on top of the basic semantics of the language by adding an appropriate axiomatization. Structural, behavioral, and arbitrary semantic for inheritance can be easily designed and coupled with user ontologies. In some cases, we show how these axiomatizations relate with F-logic under first order semantics.
4. We illustrate in which terms contexts can be exploited for manipulating hybrid knowledge bases having many data sources working under different entailment regime;
5. The language has been implemented within the DLT system, a front-end for answer set solvers. Besides the fragment of language herein presented, DLT allows negated nested molecules, and re-usable template programs. If coupled with a proper answer set solver, the same front-end allows usage of complex terms (e.g. functions, lists, sets), and external predicates [23]).

5.2 Syntax

We present here the syntax of FAS programs. Informally, the language allows disjunctive rules with negation as failure in the body; with respect to ordinary Ans-Prolog (the basic language of Answer Set Programming), there are three crucial differences. First, besides traditional atoms and predicates, the language

supports *frame molecules* in both the body and the head of rules, following the style of F-logic [39]. When representing knowledge, frame molecules allow to focus on objects, more than on predicates. An object can belong to *classes*, and have a number of *property* (attribute) values. As an example, the following is a frame molecule:

$$\begin{aligned} brown : employee\,[\ & surname \rightarrow \text{“Mr. Brown”}, \\ & skill \twoheadrightarrow \{java, asp\}, \\ & salary \rightarrow 800, \\ & gender \rightarrow male, \\ & married \rightarrow pink\,] \end{aligned}$$

The above molecule defines membership of the *subject* of the molecule (*brown*) to the *employee* class and asserts some values corresponding to the *properties* (which we will call also *attributes*) bound to this object. This frame molecule states that *brown* is *male* (as expressed by the value of the attribute *gender*), and is *married* to another employee identified by the subject *pink*. *brown* knows *java* and *asp* languages, as the values of the *skill* property suggest, while he has a *salary* equal to *800*. Intuitively, one can see a class membership statement in form $x : c$ as similar to a unary predicate $c(x)$. Accordingly, $x[m \rightarrow v]$ can be seen has a binary predicate $m(x, v)$.

As a second important difference, higher order reasoning is a first class citizen in the language: in other words, it is allowed quantification over predicate, class and property names. For instance, $C(brown)$ is meant to have the variable C ranging over the Herbrand universe, thus having $employee(brown)$ as possible ground instance.

Finally, our language allows the use of *framespaces* to place atoms and molecules in different contexts. For example, suppose there are two *Mr. Brown*, one working for *Sun* and the other for *Ibm*. We can use two different assertions, related to two different framespaces to distinguish them, e.g. $brown : employee@sun$ and $brown : employee@ibm$.

We formally define the syntax of the language next.

Let $\mathcal{C}$ be an infinite and countable set of distinguished constant and predicate symbols. Let $\mathcal{X}$ be a set of variables. We conventionally denote variables with uppercase first letter (e.g. X, $Project$), while constants will be denoted

with lowercase first letter (e.g. x, $brown$, $nonWantedSkill$). A *term* is either a constant or a variable.

Atoms can be either *standard atoms* or *frame atoms*. A standard atom is in the form $t_0(t_1, \ldots, t_n)@f$, where $t_0, \ldots, t_n, f$ are *terms*, t_0 represents the *predicate name* of the atom and f the *context* (or *framespace*) in which the atom is defined.

A *frame atom*, or *molecule*, can be in one of the following three forms:

- $s[v_1, \ldots, v_n]@f$
- $s \diamond c@f$
- $s \diamond c[v_1, \ldots, v_n]@f$

where s, c and f are terms, and $v_1, \ldots, v_n$ is a list of *attribute expressions*. Here and in the following, the allowed values for the meta-symbol $\diamond$ are ":" (*instance operator*), or "::" (*subclass operator*). Moreover, s is called the *subject* of the frame, while f represents the *context* (or *framespace*).

To simplify the notation, whenever the context term f is omitted, we will assume $f = d$, for $d \in \mathcal{C}$ a special symbol denoting the *default* context.

An attribute expression is in the form p, $p \rightharpoonup v_1$ or $p \rightharpoonup\!\!\!\rightarrow \{v_1, \ldots, v_n\}$, where p (*the property/attribute name*) is a term, and $v_1, \ldots, v_n$ (*the attribute values*) are either terms or frame molecules. Here and in the following, the meta-symbols $\rightharpoonup$ and $\rightharpoonup\!\!\!\rightarrow$ are intended to range respectively over $\{\rightarrow, \bullet\!\!\rightarrow\}$ and $\{\Rightarrow, \twoheadrightarrow, \Rightarrow\!\!\!\!\!\rightarrow, \bullet\!\!\twoheadrightarrow\}$. Note that, according to this definition, when used within attribute expressions, the symbols in the set $\{\Rightarrow, \twoheadrightarrow, \Rightarrow\!\!\!\!\!\rightarrow, \bullet\!\!\twoheadrightarrow\}$ allow sets of attribute values on their right hand side, while $\rightarrow$ and $\bullet\!\!\rightarrow$ allow single values.

A *literal* is either an atom p (positive literal), or an expression of the form $\neg p$ (strongly negated literal or, simply, negated literal), where p is an atom. A *naf-literal* (negation as failure literal) is either of the form b (positive naf-literal), or of the form $not\, b$ (negative naf-literal), where b is a literal.

A *formula* is either a naf-literal, a conjunction of formulas or a disjunction of formulas.

A *simple atom* is either a standard atom, or a frame atom in the forms $s \diamond c@f$, $s[p \rightharpoonup v]@f$ or $s[p \rightharpoonup\!\!\!\rightarrow \{v\}]@f$, for s, c, p, v and f terms of the language. The notion of simple literal and of simple naf-literal are defined accordingly on top of the notion of simple atom.

A Frame Answer Set *program* (FAS program) is a set of *rules*, of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m.$$

where $a_1, \ldots, a_n$ and $b_1, \ldots, b_k$ are literals, $not\, b_{k+1}, \ldots, not\, b_m$ are naf-literals, and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the head of r, denoted by $H(r)$, while the conjunction $b_1 \wedge \cdots \wedge b_k \wedge not\, b_{k+1} \wedge \ldots, \wedge not\, b_m$ is the body of r, denoted by $B(r)$. A rule with empty body will be called *fact*, while a rule with empty head is a *constraint*.
A *plain higher order* FAS program contains only standard atoms, while a *plain* FAS program contains only standard atoms with a constant predicate name. A *positive* FAS program do not contain negation as failure and strongly negated atoms. In the following, we will assume to deal with *safe* FAS programs, that is, programs in which each variable appearing in a rule r appears in at least one positive naf-literal in $B(r)$.

Example 5.1. The following one rule program is a valid FAS program. Intuitively, it represents the fact that each person is *male* or *female*.

$$P[gender \rightarrow \text{“male”}] \vee P[gender \rightarrow \text{“female”}] \text{ :- } P : person.$$

5.3 Semantics

Semantics of FAS programs is defined by adapting the traditional Gelfond-Lifschitz reduct, originally given for a ground disjunctive logic program with strong and default negation [31], to the case of FAS programs.
Given a FAS program P, its ground version $grnd(P)$ is given by grounding rules of P by all the possible substitutions of variables that can be obtained using consistently elements of $\mathcal{C}$[1]. A ground rule thus contains only ground atoms; the set of all possible simple ground literals that can be constructed combining predicates and terms occurring in the program is usually referred to as *Herbrand base* (B_P). We remark that the grounding process substitutes also nonground predicates names with symbols from $\mathcal{C}$ (e.g., a valid ground instance of the atom $H(brown, X)$ is $married(brown, pink)$, while a valid ground instance of $brown[H \rightarrow yellow]$ is $brown[color \rightarrow yellow]$).

[1] As shown next, our semantics implicitly assumes that elements of $\mathcal{C}$ are mapped to themselves in any interpretation, thus embracing the unique name assumption.

An *interpretation* for P is a set of simple ground literals, that is, an interpretation is a subset $I \subseteq B_P$. I is said to be *consistent* if $\forall a \in I$ we have that $\neg a \notin I$.
We define the following entailment notion with respect to an interpretation I. For a a ground atom:

($E1$) If a is simple, then $I \models a$ iff $a \in I$;
($E2$) $I \models not\, a$ iff $I \not\models a$.

For $l_1, \ldots, l_n$ ground literals:

($E3$) $I \models l_1 \wedge \cdots \wedge l_n$ iff $I \models l_i$, for each $1 \leq i \leq n$;
($E4$) $I \models l_1 \vee \cdots \vee l_n$ iff $I \models l_i$ for some $1 \leq i \leq n$.

For s, p, f ground terms, and $m_1, \ldots, m_n$ ground frame molecules:

($E5$) $I \models s[p \twoheadrightarrow \{m_1, \ldots m_n\}]@f$ iff $I \models s[p \twoheadrightarrow \{m_i\}]@f$, for each $1 \leq i \leq n$.

For s, s', c, p, f, f' ground terms, and $\overline{v} = \{v_1, \ldots, v_n\}$ a set of ground attribute value expressions:

($E6$) $I \models s[v_1, \ldots, v_n]@f$ iff $I \models s[v_1]@f \wedge \cdots \wedge s[v_n]@f$;
($E7$) $I \models s \diamond c[\overline{v}]@f$ iff $I \models s \diamond c\,@f \wedge s[\overline{v}]@f$;

($E8$) $I \models s[p \rightarrow s'[\overline{v}]]@f$ iff $I \models s[p \rightarrow s']@f \wedge s'[\overline{v}]@f$;
($E9$) $I \models s[p \twoheadrightarrow \{s'[\overline{v}]\}]@f$ iff $I \models s[p \twoheadrightarrow \{s'\}]@f \wedge s'[\overline{v}]@f$;

($E10$) $I \models s[p \rightarrow s'[\overline{v}]@f']@f$ iff $I \models s[p \rightarrow s']@f \wedge s'[\overline{v}]@f'$;
($E11$) $I \models s[p \twoheadrightarrow \{s'[\overline{v}]@f'\}]@f$ iff $I \models s[p \twoheadrightarrow \{s'\}]@f \wedge s'[\overline{v}]@f'$.

Note that rules ($E8$) and ($E9$) force $s'[\overline{v}]$, which does not have an explicit framespace, to belong to the context f of the molecule containing it. On the contrary, $s'[\overline{v}]@f'$ in ($E10$) and ($E11$) has a proper framespace f', and the entailment rules take care of this fact. Then, rules ($E6$) to ($E11$) define the context of a frame molecule as the *nearest* framespace explicitly specified.
For a rule r :

($E12$) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$;

A *model* for P is an interpretation M for P such that $M \models r$ for every rule $r \in grnd(P)$. A model M for P is *minimal* if no model N for P exists such

that N is a proper subset of M. The set of all minimal models for P is denoted by MM(P).

Given a program P and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of P w.r.t. I, denoted P^I, is the set of positive rules of the form $\{a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k\}$ such that $\{a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, not\ b_{k+1}, \cdots, not\ b_m\}$ is in grnd(P) and $I \models not\, b_{k+1} \wedge \cdots \wedge not\ b_m$. An interpretation I for a program P is an *answer set* for P if $I \in \mathrm{MM}(P^I)$ (i.e., I is a minimal model for the positive program P^I) *[58, 31]*. The set of all answer sets for P is denoted by $ans(P)$. We say that $P \models a$ for an atom a, if $M \models a$ for all $M \in ans(P)$. P is *consistent* if $ans(P)$ is non-empty.
For a positive program P allowing only the term d in context position, we define the F-logic first-order semantics in terms of its *F-models*. A *F-model* M_f is a model of P subject to the conditions

($F1$) "::" encodes a partial order in M_f;
($F2$) if $a{:}b \in M_f$ and $b{::}c \in M_f$ then $a{:}c \in M_f$;
($F3$) if $a[m \rightharpoonup v] \in M_f$ and $a[m \rightharpoonup w] \in M_f$ then $v = w$, for $\rightharpoonup \in \{\rightarrow, \bullet\!\!\rightarrow\}$;
($F4$) if $a[m \approx\!> v] \in M_f$ and $b{::}a$ then $b[m \approx\!> v] \in M_f$, for $\approx\!> \in \{\Rightarrow, \Rightarrow\!\!\!\!\Rightarrow\}$;
($F5$) if $c[m \Rightarrow v]$, $a{:}c$ and $a[m \rightarrow w] \in M_f$ then $w{:}v \in M_f$;
($F6$) if $c[m \Rightarrow\!\!\!\!\Rightarrow v]$, $a{:}c$ and $a[m \twoheadrightarrow w] \in M_f$ then $w{:}v \in M_f$;

We say that $P \models_f a$ for an atom a if $M_f \models a$ for all F-models of P.

Example 5.2. The program in Example 5.1 together with the fact $brown{:}person.$ has two answer sets, $M_1 = \{\, brown{:}person,\ brown[\, gender \rightarrow$ "male" $]\,\}$ and $M_2 = \{\, brown{:}person,\ brown[gender \rightarrow$ "female"$]\,\}$. Both M_1 and M_2 are F-models. Note that $M_3 = \{\, brown : person,\ brown[\, gender \rightarrow$ "female" $],\ brown[\, gender \rightarrow$ "male" $]\,\}$ is neither an F-model nor an answer set for different reasons: it is not an F-model because of condition ($F3$) given above, while it is not an answer set because it is not minimal. Note also that disjunctive rules trigger in general the existence of multiple answer sets, while the presence of constraints may eliminate some or all constraints: for instance, the same program enriched with the constraints $\leftarrow brown[gender \rightarrow$ "male"$]$ and $\leftarrow brown[gender \rightarrow$ "female"$]$ has no answer set[2].

[2] A constraint $\leftarrow c$ can be seen as a rule $f \leftarrow c, not\, f$, for which there is no model containing c.

5.4 Modeling semantics and inheritance

Given the basic semantics for a FAS program P, it is then possible to enforce a specific behavior for operators of the language by adding to P specific "axiomatic modules". An *axiomatic module* A is in general a FAS program. Given a union of axiomatic modules $S = A_1 \cup \cdots \cup A_n$, we will say that P entails a formula ϕ under the axiomatization S ($P \models_S \phi$) if $P \cup S \models \phi$. The answer sets of P under axiomatization S are defined as $ans_S(P) = ans(P \cup S)$. We illustrate next some basic axiomatic modules.

Basic class taxonomies.

The axiomatic module $\mathcal{C}$, shown next, associates to " : " and " :: " the usual meaning of monotonic class membership and subclass operator.

c_1 : $A::B \leftarrow A::C,\ C::B.$
c_2 : $A::A \leftarrow X:A.$
c_3 : $\leftarrow A::C,\ C::A,\ A \neq C.$
c_4 : $X:C \leftarrow X:D,\ D::C.$

Rules c_1 and c_2 enforce transitivity and reflexivity of the subclass operator, respectively. Rule c_3 prohibits cycles in the class taxonomy, while c_4 implements the class inheritance for individuals by connecting the " :: " operator to the " : " operator. The acyclicity constraint can be relaxed if desired: we define in this case $\mathcal{C}'$ as $\mathcal{C} \setminus c_3$[3].

Single valued attributes.

Under standard F-logic, the operators $\rightarrow$ and $\bullet\!\!\rightarrow$ are associated to families of single valued functions: indeed, in a F-model M it can not hold both $a[m \rightharpoonup v]$ and $a[m \rightharpoonup w]$, unless $v = w$. Under unique names assumption, we can state the above condition by the set $\mathcal{F}$ of constraints:

f_5 : $\leftarrow A[M \rightarrow V], A[M \rightarrow W], V \neq W$
f_6 : $\leftarrow A[M \bullet\!\!\rightarrow V], A[M \bullet\!\!\rightarrow W], V \neq W$

Structural and behavioral inheritance.

We show here how to model some peculiar types of inheritance, such as structural and behavioral inheritance.

[3] Note that the atom $A \neq C$ amounts to *syntactic* inequality between A and C.

Structural inheritance is usually associated to the operator $\Rightarrow$. Let P_1 be the following example program:

$webDesigner::javaProgrammer.\ javaProgrammer::programmer.$
$webDesigner::htmlProgrammer.\ javaProgrammer[salary \Rightarrow medium].$
$htmlProgrammer[salary \Rightarrow low].$

For short, we denote in the following *webDesigner* as *wd*, *javaProgrammer* as *jp* and *htmlProgrammer* as *hp*.

Under structural inheritance, as defined in [39], property values of superclasses are "monotonically" added to subclasses. Thus, since c_1 is subclass of c_2 and c_4, one expects that $P_1 \models_{\mathcal{C}\cup\mathcal{S}} webDesigner[salary \Rightarrow \{low, medium\}]$ for some axiomatic module $\mathcal{S}$.

The axiomatic module $\mathcal{S}$ shown next, associates this behavior to the operators $\Rightarrow$ and $\Rightarrow\!\!\!\!\!\Rightarrow$.

$s_7:\ D[A \Rightarrow T] \leftarrow D::C,\ C[A \Rightarrow T].$
$s_8:\ D[A \Rightarrow\!\!\!\!\!\Rightarrow T] \leftarrow D::C,\ C[A \Rightarrow\!\!\!\!\!\Rightarrow T].$

Note that s_5 (resp. s_6) do not enforce any relationship between "$\Rightarrow$" and "$\rightarrow$" (resp. "$\Rightarrow\!\!\!\!\!\Rightarrow$" and "$\rightarrow\!\!\!\!\!\rightarrow$") as in [39]. We will discuss this issue later in the section.

Behavioral inheritance [71], allows instead nonmonotonic overriding of property values. Overriding is a common feature in object-oriented programming languages like Java and C++: when a more specific definition (value, in our case) is introduced for a method (a property, in our case), the more general one is overridden. In case different information about an attribute value can be derived from several inheritance paths, inheritance is *blocked*. Let us assume to add to P_1 the assertions $jp[income \bullet\!\!\rightarrow 1000]$ and $hp[income \bullet\!\!\rightarrow 1200]$.

Under behavioral inheritance regime [71][4], the assertions $jp[income \bullet\!\!\rightarrow 1000]$ and $hp[income \bullet\!\!\rightarrow 1200]$ would be considered in conflict when inherited from *wd*. Indeed, both $wd[income \bullet\!\!\rightarrow 1000]$ and $wd[income \bullet\!\!\rightarrow 1200]$ under the three-valued semantics of [71] are left *undefined*. Under FAS semantics it is then expected to have some axiomatic module $\mathcal{B}$ where neither $P_1 \models_{\mathcal{B}\cup\mathcal{F}\cup\mathcal{C}} wd[income \bullet\!\!\rightarrow 1000]$ nor $P_1 \models_{\cap B\cup\mathcal{F}\cup\mathcal{C}} wd[income \bullet\!\!\rightarrow 1200]$ hold.

[4] Note that in [71] the above semantics is conventionally associated to the $\rightarrow$ operator, while we will use $\bullet\!\!\rightarrow$

The above behavior can be enforced by defining $\mathcal{B}$ as follows

$$
\begin{array}{rl}
b_9: & overridden(D,M,C) \leftarrow E[M \bullet\!\!\rightarrow V],\ C::E,\ E::D,\ C \neq E,\ E \neq D.\\
b_{10}: & inheritable(C,M,D) \leftarrow C::D,\ D[M \bullet\!\!\rightarrow V],\ not\ overridden(D,M,C).\\
b_{11}: & C[M \bullet\!\!\rightarrow V] \vee C[M \bullet\!\!\rightarrow V]@false \leftarrow inheritable(C,M,D),\ D[M \bullet\!\!\rightarrow V].\\
b_{12}: & exists(C,M) \leftarrow C[M \bullet\!\!\rightarrow V].\\
b_{13}: & \leftarrow inheritable(C,M,D),\ not\ exists(C,M).\\
b_{14}: & existsSubclass(A,C) \leftarrow A:C,\ A:D,\ D::C,\ C \neq D.\\
b_{15}: & A[M \rightarrow V]@candidate \leftarrow A:C,\ C[M \bullet\!\!\rightarrow V],\ not\ existsSubclass(A,C).\\
b_{16}: & A[M \rightarrow V] \vee A[M \rightarrow V]@false \leftarrow A[M \rightarrow V]@candidate.\\
b_{17}: & exists'(A,M) \leftarrow A[M \rightarrow V].\\
b_{18}: & \leftarrow inheritable(C,M,C),\ A:C,\ not\ exists'(A,M).
\end{array}
$$

The above module makes usage of stable model semantics for modeling multiple inheritance conflicts. By means of rule b_{11} and b_{16} it is triggered the existence of multiple answer set in the presence of inheritance conflicts, one for each possible way to solve the conflict itself.

Note that $ans_{\mathcal{B}\cup\mathcal{F}\cup\mathcal{C}}(P_1)$ contains two different answer sets M_1 and M_2 which respectively are such that $M_1 \models wd[income \bullet\!\!\rightarrow 1200]$ and $M_2 \models wd[income \bullet\!\!\rightarrow 1000]$. However, both assertions do not hold in all the possible answer sets. Thus, similarly to "well-founded optimism" semantics, we obtain that $P_1 \not\models_{\mathcal{C}\cup\mathcal{B}} wp[income \bullet\!\!\rightarrow X]$ for any X.

Constructive vs well-typed semantics.

The operator $\Rightarrow$ is traditionally associated to $\rightarrow$. For instance if both $jp[keyboard \Rightarrow americanLayout]$ and $jim : jp[keyboard \rightarrow ibm1050]$ hold, one might expect that $ibm1050 : americanLayout$.

However, one might wonder whether to implement the above required behavior under a *constructive* or a *well-typed* semantics.

The two type of semantics differ in the way incomplete information is dealt with. In a "well-typed" flavored semantics, most axioms are seen as hard constraints, which, if not fulfilled, make the theory at hand inconsistent.

In the first case, it may be desirable to use the "$\Rightarrow$" operator for defining strong desiderata about range and domain of properties, while the "$\rightarrow$" could be used to denote actual instance values such as in the following program P_2:

$programmer[salary \Rightarrow integer]$.
$g : programmer[salary \rightarrow aSalary]$.
$\leftarrow X : programmer[salary \rightarrow Y], not\, Y : integer$[5]

Note that $ans(P_2)$ is empty, unless it is not *explicitly* asserted (well-typed) the fact $aSalary : integer$.

On the other hand one may want to interpret *constructively* desiderata about domain and range of properties, as it is typical, e.g. of RDFS. Consider the program P_3:

$programmer[salary \Rightarrow integer]$.
$g : programmer[salary \rightarrow aSalary]$
$Y : integer \leftarrow X : programmer[salary \rightarrow Y]$

Here P_3 has a single answer set containing the fact $aSalary : integer$.

The two types of semantics stem from profound philosophical differences: well-typedness is commonly (but not necessarily) associated to modeling languages inspired from database systems, living under a single model semantics and Closed World Assumption. To a large extent one can instead claim that first order logics (and descendant formalisms, such as descriptions logics and RDFS), is much more prone to deal constructively with incomplete information.

It is however worth noting that despite their conceptual difference, constructive and well-typed semantics are often needed together. As a matter of example, modeling in Java (as well as C++ and F-logic) needs both flavors. Constructiveness comes into play in inheritance within class taxonomies (e.g., if $A::B$ and $B::C$ hold, the information $A::C$ does not need to be well-typed and is inferred automatically), but well-typedness is required in several other contexts, (e.g. strong type-checking prescribes that a function having a given signature can not be invoked using actual parameters which are not *explicitly known* to fulfil the function signature).

Whenever required, FAS programs can be coupled with axiomatic modules encoding both well-typed and constructive axioms.

[5] With some liberality we use here "integer" as class name more than a concrete datatype, without losing the sense of our example.

The following axiomatic module $\mathcal{CO}$ encodes constructively how the operators $\Rightarrow$ and $\rightarrow$ can be related each other:

$co_{15} : V:T \leftarrow C[A \Rightarrow T], I:C, I[A \rightarrow V].$

while $\mathcal{W}$, shown next, encodes the same relation under a well-typed semantics.

$w_{16} : \leftarrow C[A \Rightarrow T], I:C, I[A \rightarrow V], not\, V:T.$

5.5 Properties of FAS programs

FAS programs have some property of interest. First, F-logic entailment can me modeled on top of FAS programs by means of the axiomatic modules $\mathcal{C}, \mathcal{S}, \mathcal{F}$, and $\mathcal{CO}$. Let $\mathcal{A} = \mathcal{C} \cup \mathcal{S} \cup \mathcal{F} \cup \mathcal{CO}$.

Theorem 5.3. *Given a positive, non-disjunctive,* FAS *program P with default contexts only, and a formula ϕ, then $P \models_{\mathcal{A}} \phi$ iff $P \models_f \phi$.*

Proof. (Sketch). ($\Rightarrow$) Assume $P \cup \mathcal{A}$ is inconsistent. Given that P is a positive program, then inconsistency amounts to the violation of some instance of constraints c_3, f_5 or f_6. We can show that, accordingly, there is no F-model for P. On the other hand, if $P \cup \mathcal{A}$ is consistent, one can show that the unique answer set of P is the least F-model of P.
($\Leftarrow$) It can be shown that if P has no F-model, then $P \cup \mathcal{A}$ is inconsistent. Viceversa, if P has some F-model its least model corresponds to the unique answer set of $P \cup \mathcal{A}$. □

One might wonder at the significance of $\models_{\mathcal{A}}$-entailment for disjunctive programs with negation. This entailment regime diverges quickly from the behavior of monotonic logic as soon as negation as failure and disjunction is considered, and is thus incomparable with first order F-logic. It is matter of future research to investigate on the relationship between FAS programs and F-logic under well-founded semantics.
As a second important property, we show that contexts can be exploited for modeling hybrid environments in which more than one semantics has to be taken in account. For instance one might desire a context s in which only $\mathcal{C} \cup \mathcal{S}$ hold as axiomatic modules (this is typical e.g. of RDFS reasoning restricted to ρ-DF [51]), while in a context b we would like to have a different entailment regime, taking in account e.g. $\mathcal{B}$ and $\mathcal{F}$.

We will say that an axiomatic module (resp. a program, a formula) $\mathcal{A}$ is defined at context c if for each rule $r \in \mathcal{A}$, each atom $c \in r$ has context c. If an axiomatic module (resp. a program, or a formula) $\mathcal{A}$ is defined at the default context d, then the axiomatic module $\mathcal{A}@c$, defined at context c, is obtained by replacing each atom a appearing in $\mathcal{A}$ with $a@c$.

Example 5.4. Consider the program P_4 defined as follows. P_4 has two contexts, *rdf* and *inh*. P_4 contains knowledge coming from an RDF triplestore defined in term of the facts $t(gb, rdf{:}type, hp)@rdf$, $t(gb, name, \text{“Gibbi”})@rdf$, etc. Also P_4 contains the rules $X{:}C@rdf \leftarrow t(X, rdf{:}type, C)@rdf$, $X[M \rightarrow V]@rdf \leftarrow t(X, M, V)@rdf$, $C::D@rdf \leftarrow t(C, rdfs:subClassOf, D)@rdf$. Then, we add to P_4 the program $P_1@inh$ where P_1 is taken from Section 5.4, plus the rule $X : C@inh \leftarrow X : C@rdf$. We want that $\mathcal{C}$ and $\mathcal{S}$ hold under the *rdf* context, while $\mathcal{C}$ and $\mathcal{B}$ hold under the *inh* context. This can be obtained by defining $\mathcal{A} = (\mathcal{C} \cup \mathcal{S})@rdf \cup (\mathcal{C} \cup \mathcal{B})@inh$ and evaluating P_4 under $\models_{\mathcal{A}}$-entailment.
For instance, $P_4 \models_{\mathcal{A}} gb{:}[income \bullet\!\!\rightarrow 1000]@inh$.

We clarify next how contexts interact each other. First, we consider programs in which contexts are strictly separated: that is, each rule in a program contains only atoms either with context a or only atoms with context b. This way, a program can be seen as composed by two separate modules, one defining a and the other defining b. The following proposition shows that programs defined in separated context behave separately under their axiomatic regime.

Proposition 5.5. *It is given a program $P = P'@a \cup P''@b$, and axiomatic modules $A@a$ and $B@b$. Then, for formulas $\phi@a$ and $\psi@b$, we have that, if $P \cup A@a \cup B@b$ is consistent,*

$$P \models_{A@a \cup B@b} \phi@a \wedge \psi@b \Leftrightarrow P' \models_A \phi \wedge P'' \models_B \psi$$

Contexts can be seen in some sense as separate knowledge sources, each of which having its own semantics for its data. In such a setting, it is however important to consider cases in which knowledge flows bidirectionally from a context to another and viceversa.
This situation is typical of languages implementing hybrid semantics schemes. For instance, $\mathcal{DL}$+*log* [62] is a rule language where each knowledge base combines a description logic base D (living under first order semantics), with a

rule program P (living under answer set semantics). D and P can mutually exchange knowledge: in the case of $\mathcal{DL}$+*log*, predicates of D can appear in P, allowing flow of information from D to P.
Similarly, we are assuming to have a program P, two contexts a and b, each of which coupled with axiomatic modules $A@a$ and $B@b$. The program P freely combines atoms with context a with atoms with context b, possibly in the same rule.
For simplicity, the following theorem is given for programs containing simple naf-literals only.
Given an interpretation I we define I_a as the subset of I containing only atoms with context a. The *extended reduct* P^{*I_a} of a ground program P is given by modifying each rule $r \in P$ in the following way:

- if $l@a \in H(r)$ and $l@a \notin I_a$ then delete $l@a$ from r;
- if $l@a \in H(r)$ and $l@a \in I_a$ then delete r;
- if $l@a \in B(r)$ and $l@a \in I_a$ then delete $l@a$ from r;
- if $l@a \in B(r)$ and $l@a \notin I_a$ then delete r;
- if $not\, l@a \in B(r)$ and $l@a \notin I_a$ then delete $not\, l@a$ from r;
- if $not\, l@a \in B(r)$ and $l@a \in I_a$ then delete r;

Theorem 5.6. *Let P be a program containing only atoms with context a and b, and $A@a$ and $B@b$ be two axiomatic modules.*
Then,

$$M \in ans_{A@a \cup B@b}(P) \Leftrightarrow M_a \in ans_{A@a}(P^{*M_b}) \wedge M_b \in ans_{B@b}(P^{*M_a})$$

Roughly speaking, the above theorem states that from the point of view of context a one can see atoms from context b as external facts, and viceversa. An answer set M of the overall program is found when, assuming M_a as the set of true facts for a, we obtain that M_b is the answer set of $P^{*M_a} \cup B@b$, i.e. an answer set of the program obtained by assuming facts in M_a true. Viceversa, if one assumes M_b as the set of true facts for context b, one should obtain M_a as the answer set of $P^{*M_b} \cup A@a$.

Proof. (Sketch). ($\Rightarrow$) Assume $M \in ans(P \cup A@a \cup B@b)$, it is easy, yet tedious, to construct M_a and M_b and verify that $M_a \in ans(P^{*M_b} \cup A@a)$ and $M_b \in ans(P^{*M_a} \cup B@b)$. Given $P_a = P^{*M_b} \cup A@a$ and $P_b = P^{*M_a} \cup B@b$, the

proof is conducted by showing that M_a (resp. M_b) is a minimal model of $P_a^{M_a}$ (resp. $P_b^{M_b}$).
($\Leftarrow$) Given M_a and M_b such that $M_a \in ans(P^{*M_b} \cup A@a)$ and $M_b \in ans(P^{*M_a} \cup B@b)$, the proof is carried out by showing that $M = M_a \cup M_b$ is a minimal model of $P \cup A@a \cup A@b^M$. □

5.6 System Overview

FAS programs have been implemented within the DLT environment [35]. The current version of the system is freely available on the DLT Web page, together with examples, a tutorial, and the axiomatic modules herein presented.
DLT works as a front-end for an answer set solver of choice S. Programs are rewritten in the syntax of S and then processed. Resulting answer sets in the format of S are then processed back and output in DLT format. DLT is compatible with most of the languages of the DLV family such as DLV [42], dlvhex [24] and the recent DLV-complex. The native features of the solver of choice are made available to the DLT programmer: this way features such as soft constraints, aggregates (DLV), external predicates (dlvhex), and function, list and set terms (DLV-complex) are accessible. Limited support is given also for other ASP solvers.
DLT allows the syntax presented in this paper and implements the presented semantics. Atoms without context specification are assumed to have the default context d. In order to avoid typing, the default implicit context can be switched by using a directive in the form *@name.*, which sets the implicit context to *name* for the rules following the directive.
We overview next some of the other features of DLT, which, for space reasons, can not be focused in the present work.

Complex nested expression.

DLT allows the usage of negated attribute expressions. From the operational point of view, if a frame literal in the body of a rule r has subject o and a negative attribute $not\ m$, our prototype removes $not\ m$ from the attributes of o, adds $not\ a$ to the body of r, where a is a fresh auxiliary atom, and adds a new rule a:-$o[m]$. to the program. This procedure can be iterated until no negated attribute appears in the program. Then, the answer sets of the original

program are the answer sets of the rewritten program without auxiliary atoms. Since negated attributes can appear in negative literals and can be nested, they behave like the nested expressions of [43], allowing in many case to represent information in a more succinct way. The model-theoretical semantics of this aspect of the language is not focused in this paper and is matter of future work.

Example 5.7. The following rule states that a programmer P is suitable for project p_3 if P know *c++* and *perl*, but is not married to another programmer knowing *c++* and *perl*.

$$P[suitable \twoheadrightarrow p_3] \leftarrow X\!:\!programmer,\ P\!:\!programmer[skills \twoheadrightarrow \{\text{“c++”, “perl”}\},\ not\ married \rightarrow X[skills \twoheadrightarrow \{\text{“c++”, “perl”}\}].$$

Template definitions.

A DLT program may contain *template atoms*, that allow to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This feature provides a succinct and elegant way for quickly introducing new constructs using the DLT language, such as predefined search spaces, custom aggregates, etc. Differently from higher order constructs, which can be used for the same purpose, templates are based on the notion of generalized quantifier, and allow more versatile usage. Syntax and semantics of template atoms are described in [17].

5.7 Remarks and Related Work

We summarize here the main topics we focused in this chapter, pointing out significant features and issues.

Stable vs well-founded semantics.

FAS programs have some peculiar differences with respect to the original F-logic. Importantly, while well-founded semantics [31] is at the basis of the nonmonotonic semantics of F-logic, FAS programs live under stable model semantics. The two semantics are complementary in several respects. The well-founded semantics is preferable in terms of computational costs: at the same time, this limits

expressiveness with respect to the stable model semantics, which for disjunctive programs can express any query in the computational class Σ_2^p.
On the other hand, the well-founded semantics is three-valued. Having a third truth value as first class citizen of the language is an advantage in several scenarios, such as just in the case of object inheritance. Indeed, the undefined value is exploited in F-Logic when inheritance conflicts can not be solved with a clear truth value. Note, however, that the stable model semantics gives finer grained details in situations in which the well-founded semantics leaves truth values undefined. The reader can find a thorough comparison of the two semantics in [31]. FAS answer sets should not be confused with the notion of *stable object model* given in [71].

Semantic Web languages.

Since F-logic features a natural way for manipulating ontologies and web data, it has been investigated for a long as suitable basis for representing and reasoning on data on the web. The two main F-Logic systems Flora and Florid ([72, 46]) share with FAS programs the ability to work both on the level of concepts and attributes and on instances.
F-logic has been investigated as a logical way to provide reasoning capability on top of RDF in the system TRIPLE ([66]) that has native support for contexts (called *models*), URIs and namespaces. It is possible also to personalize semantics either via rule axiomatization (e.g. one can simulate RDFS reasoning by means of TRIPLE rules) or by means of interfacing external reasoners. The semantics of the full TRIPLE language has not been clearly formalized: its positive, non-higher order fragment coincides with Horn logic.
The possibility to define custom rule set for specifying the semantics which best fits the concrete application context is also allowed in OWLIM ([40]).

Answer Set Programming

Several works share some point in common with the described techniques in the field of Answer Set Programming. An inspiring first definition of F-logic under stable model semantics can be found in [21]. The fragment considered focuses on first order F-logic with class hierarchies, and do not explicitly axiomatize structural inheritance with constructive semantics and single valued attributes. Higher order reasoning is present in dlvhex [24]. Contexts were investigated under stable model semantics also in [56]. In this setting, context atoms are

exploited to give meaning to a form of scoped negation, useful in Semantic Web applications where data sources with complete knowledge need to be integrated with sources expected to work under Open World Assumption. Similarly to our work, multi-context systems of [10] are used in order to define hybrid system with a logic of choice. Contexts can transfer knowledge each other by means of *bridge rules*, while in our setting it is not necessary a clear distinction between knowledge bases and bridge rules.

Nested attribute expressions behave like nested expressions as in [43], although we do not allow the use of negation in the head of rules. A different approach to nonmonotonic inheritance in the context of stable model semantics was proposed in [12], in which modules (which can be overridden each other) are associated with each object, and objects are partially sorted by an *isa* relation. The idea of defining an object-oriented modeling language under stable model semantics has been also subject of research in [61] and [60].

6

Translating OWL2 Profiles to ASP with Axiomatic Modules

OWL2, as stated previously, is divided into three different profiles. Each profile has been created for answering to different problems, and is based on different description logic fragments. We give next some modular translations from the fragments of OWL2 to Logic Programming. Such translations may be used for improving query answering, as we have seen for the case of $\mathcal{ELHI}$. We propose separate translations for the three profiles, even if some of the constructs are in common, to better esemplify the language and what it takes to perform the operation of translating it to a different formalism. As in the case of $\mathcal{ELHI}$, we first trasform the rules in first order sentences, then we apply a classical skolemization procedure to have the corresponding logic programming rules.

6.1 OWL2-EL

For OWL2-EL, being it very similar to $\mathcal{ELHI}$, we just report the conversion table. In particular, we give the semantics of a $\mathcal{ELHI}$ knowledge base in terms of a conjunction of first order sentences. More specifically, in Table 6.2, each $\mathcal{ELHI}$ axiom H is associated to the corresponding first order sentence $\mathcal{F}(H)$. The semantics of $\mathcal{KB}$ is given by its corresponding first order theory $\mathcal{F}(\mathcal{KB}) = \bigwedge_{H \in \mathcal{T}} \mathcal{F}(H) \wedge \bigwedge_{H \in \mathcal{A}} \mathcal{F}(H)$.

$\mathcal{ELHI}$ axiom H	FOL sentence $\mathcal{F}(H)$	Rule set $\mathcal{L}(\mathcal{F}(H))$
$A(a)$	$A(a)$	$A(a).$ $(R1)$
$R(a,b)$	$R(a,b)$	$R(a,b).$ $(R2)$
$A \sqsubseteq B$	$\forall x A(x) \rightarrow B(x)$	$B(X) \leftarrow A(X).$ $(R3)$
$A \sqcap B \sqsubseteq C$	$\forall x A(x) \wedge B(x) \rightarrow C(x)$	$C(X) \leftarrow A(X), B(X).$ $(R4)$
$A \sqsubseteq \exists R.B$	$\forall x A(x) \rightarrow [\exists y B(y) \wedge R(x,y)]$	$B(f_{\mathcal{A}}(X)) \leftarrow A(X).$ $(R5)$ $R(X, f_{\mathcal{A}}(X)) \leftarrow A(X).$
$\exists R.A \sqsubseteq B$	$\forall x [\exists y A(y) \wedge R(x,y)] \rightarrow B(x)$	$B(X) \leftarrow A(Y), R(X,Y).$ $(R6)$
$\exists R.\top \sqsubseteq B$	$\forall x [\exists y R(x,y)] \rightarrow B(x)$	$B(X) \leftarrow R(X,Y).$ $(R7)$
$B \sqsubseteq \exists R.\top$	$\forall x B(x) \rightarrow [\exists y R(x,y)]$	$R(X, f_{\mathcal{A}}(X)) \leftarrow B(X).$ $(R8)$
$R \sqsubseteq S$ or $R^- \sqsubseteq S^-$	$\forall x,y\, R(x,y) \rightarrow S(x,y)$	$S(X,Y) \leftarrow R(X,Y).$ $(R9)$
$R \sqsubseteq S^-$ or $R^- \sqsubseteq S$	$\forall x,y\, R(y,x) \rightarrow S(x,y)$	$S(X,Y) \leftarrow R(Y,X).$ $(R10)$
	FOL Query $Q(\mathbf{X})$	Rule $\mathcal{L}(Q(\mathbf{X}))$
	$\exists \mathbf{Y}\, q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n)$	$ans_Q(\mathbf{X}) \leftarrow$ $q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n).$ $(Q1)$

Table 6.1: Semantics of $\mathcal{ELHI}$ given in terms of corresponding FOL sentences. For an axiom $\mathcal{A}$ in the form $A \sqsubseteq \exists R.B$, $f_{\mathcal{A}}$ denotes a fresh function symbol. Analogously, for a query Q, ans_Q denotes a fresh predicate name.

6.2 OWL2-QL

DL-LITE axiom H	FOL sentence $\mathcal{F}(H)$	Rule set $\mathcal{L}(\mathcal{F}(H))$	
$A(a)$	$A(a)$	$A(a).$	$(R1)$
$R(a,b)$	$R(a,b)$	$R(a,b).$	$(R2)$
$A \sqsubseteq B$	$\forall x A(x) \rightarrow B(x)$	$B(X) \leftarrow A(X).$	$(R3)$
$A \sqsubseteq \bot$	$\forall x A(x) \rightarrow$	$\leftarrow A(X).$	$(R3)$
$A \sqsubseteq \neg B$	$\forall x A(x) \rightarrow$ $\neg B(x), \forall x B(x) \rightarrow \neg A(x)$	$\neg B(X) \leftarrow A(X).$ $\neg A(X) \leftarrow B(X).$	$(R3)$
$A \sqsubseteq C_1 \sqcap C_2$	$\forall x A(x) \wedge B(x) \rightarrow C(x)$	$C_1(X) \leftarrow A(X),$ $C_2(X) \leftarrow B(X).$	$(R4)$
$\exists R.\top \sqsubseteq B$	$\forall x [\exists y R(x,y)] \rightarrow B(x)$	$B(X) \leftarrow R(X,Y).$	$(R7)$
$B \sqsubseteq \exists R.\top$	$\forall x B(x) \rightarrow [\exists y R(x,y)]$	$R(X, f_{\mathcal{A}}(X)) \leftarrow B(X).$	$(R8)$
$R \sqsubseteq S$	$\forall x,y\, R(x,y) \rightarrow S(x,y)$	$S(X,Y) \leftarrow R(X,Y).$	$(R9)$
	FOL Query $Q(\mathbf{X})$	Rule $\mathcal{L}(Q(\mathbf{X}))$	
	$\exists \mathbf{Y}\, q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n)$	$ans_Q(\mathbf{X}) \leftarrow$ $q_1(\mathbf{X}_1) \wedge \ldots \wedge q_n(\mathbf{X}_n).$	$(Q1)$

Table 6.2: Semantics of DL-LITE given in terms of corresponding FOL sentences. For an axiom $\mathcal{A}$ in the form $A \sqsubseteq \exists R.B$, $f_{\mathcal{A}}$ denotes a fresh function symbol. Analogously, for a query Q, ans_Q denotes a fresh predicate name.

6.3 OWL2-RL

For OWL2-RL we need to use a different approach. This fragment, in fact, is noticeably more complex than the others. In particular, it permits the use of more concept costructors, as well as equivalence between concepts. We can now define a OWL2-RL Tbox and Abox. The TBox is built using expressions of the form $Sub \subseteq Super$, or $e_1 \equiv e_2$, where $Sub, Super, e1, e2$ can have any value from the corresponding lists in the following:

$$Sub : class | Sub \sqcup Sub | Sub \sqcap Sub | \exists R.Sub$$

$$Super : class | \neg Sub | \forall R.Super | \exists R.Super$$

$$e : class | e_1 \sqcap e_2 | dataExpression$$

dataExpression deserves a special treatment. OWL2RL, as stated before, allows for the full xml-schema datatypes, as well as equivalence expressions. The increased expressivity introduces new kinds of problems.
Here we present an axiomatization based on [50], but adapted for the ASP syntax. It models the rdf-based semantics of OWL2RL, expressed in ASP. In particular, we need modifications for the variables and for the lists, which are different in ASP. Lists in OWL2RL (RDF syntax) are represented using the paradigm first-rest. To better esemplify this behaviour,we show a table containing the triples necessary to represent a list of elements $e_1, e_2, \ldots, e_{n-1}, e_n$.

$T(h, rdf:first, e_1)$	$T(h, rdf:rest, z_2)$
$T(z_2, rdf:first, e_2)$	$T(z_2, rdf:rest, z_3)$
...	...
$T(z_n, rdf:first, e_n)$	$T(z_n, rdf:rest, rdf:nil)$

Table 6.3
Pattern used to expand a list in OWL2RL

In classical ASP lists are not supported by default. The new version of DLV, called DLV-complex, introduces this useful feature, as well as other interesting capabilities, which enhance the expressive power of the language.
To implement OWL2RL lists in ASP, we use the following method: we introduce an atom called whose predicate name is "List". It has two arguments: the first is a variable , the second is a List term.
$list(X, [A]) \leftarrow first(X, A), rest(X, nil)$
$list(X, [A|B]) \leftarrow first(X, A), rest(X, Y), list(Y, [B])$
In DLV-complex Lists can be represented by a single construct. A list of elements $e_1, \ldots, e_n$ can be represented with the term $[e_1, \ldots, e_n]$. From now on we will use this construct when dealing with lists of any kind.

OWL2RL axiom code H	ASP axiom
eq-ref	$sameAs(S,S) \leftarrow P(S,O)$ $sameAs(P,P) \leftarrow P(S,O)$ $sameAs(O,O) \leftarrow P(S,O)$
eq-sim	$sameAs(Y,X) \leftarrow sameAs(X,Y)$
eq-trans	$sameAs(X,Z) \leftarrow sameAs(X,Y), sameAs(Y,Z)$
eq-rep-s	$P(S,O) \leftarrow sameAs(S,S')$
eq-rep-p	$P(S,O) \leftarrow sameAs(P,P')$
eq-rep-o	$P(S,O) \leftarrow sameAs(O,O')$
eq-diff-1	$\leftarrow sameAs(X,Y), differentFrom(X,Y)$
eq-diff-2	$\leftarrow allDifferent([z_1,\ldots,z_n]), sameAs(z_i,z_j)$
eq-diff-3	$\leftarrow allDifferent([z_1,\ldots,z_n]), sameAs(z_i,z_j)$
eq-diff-4	$\leftarrow allDifferent([z_1,\ldots,z_n]),$ $distinctMembers([z_1,\ldots,z_n]), sameAs(z_i,z_j)$

Table 6.4
Semantics of Equality in OWL2RL given in terms of corresponding ASP rules. For axioms eq-diff-2, eq-diff-3,eq-diff-4 the condition is that $1 \leq i < j \leq n$

The table above models the semantics for equality. In fact, it is necessary to specify how equality is handled by the system. Apart from the first axioms (reflexivity, symmetry and transitivity the meaning of which is trivial), the $allDifferent$ axioms are important, as they introduce constraints to avoid that an equivalence is forced by mistake.

Axiom Code	ASP axiom
prp-dom	$C(X) \leftarrow P(X,Y), domain(P,C)$
prp-rng	$C(Y) \leftarrow P(X,Y), range(P,C)$
prp-fp	$sameAs(Y_1,Y_2) \leftarrow functionalProperty(P), P(X,Y_1), P(X,Y_2)$
prp-ifp	$sameAs(X_1,X_2) \leftarrow inverseFunctionalProperty(P), P(X_1,Y), P(X_2,Y)$
prp-irp	$\leftarrow irreflexiveProperty(P), P(X,X)$
prp-symp	$P(Y,X) \leftarrow symmetricProperty(P), P(X,Y)$
prp-asyp	$\leftarrow asymmetricProperty(P), P(X,Y), P(Y,X)$
prp-trp	$P(X,Z) \leftarrow transitiveProperty(P), P(X,Y), P(Y,Z)$
prp-spo1	$P_2(X,Y) \leftarrow P_1(X,Y), subPropertyOf(P_1,P_2)$
prp-spo2	$P(U_1,U_{n+1}) \leftarrow propertyChainAxiom([p_1,\ldots,p_n]), P_1(U_1,U_2), P_2(U_2,U_3),\ldots,P_n(U_n,U_{n+1})$
prp-eqp1	$P_2(X,Y) \leftarrow P_1(X,Y), equivalentProperty(P_1,P_2)$
prp-eqp2	$P_1(X,Y) \leftarrow P_2(X,Y), equivalentProperty(P_1,P_2)$
prp-pdw	$\leftarrow propertyDisjointWith(P_1,P_2), P_1(X,Y), P_2(X,Y)$
prp-adp	$\leftarrow allDisjointProperties([P_1,P_n]), P_i(U,V), P_j(U,V)$ $\forall\ 1 \leq i < j \leq n$
prp-inv1	$P_2(Y,X) \leftarrow inverseOf(P_1,P_2), P_2(X,Y)$
prp-inv2	$P_1(Y,X) \leftarrow inverseOf(P_1,P_2), P_1(X,Y)$
prp-key	$sameAs(X,Y) \leftarrow hasKey(C,[P_1,\ldots,P_n]),$ $C(X), P_1(X,Z_1),\ldots P_n(X,Z_n)$ $C(Y), P_1(Y,Z_1),\ldots P_n(Y,Z_n)$
prp-npa1	$\leftarrow sourceIndividual(X,I_1), assertionProperty(X,P), targetIndividual(X,I_2), P(I_1,I_2)$
prp-npa2	$\leftarrow sourceIndividual(X,I), assertionProperty(X,P), targetValue(X,Lt), P(I,Lt)$

Table 6.5
Semantics of Properties in OWL2RL given in terms of corresponding ASP rules.

The above table defines all the axioms for the Properties. Between all the axioms, some are trivial (the first ones until prp-spo1). The axiom prp-spo2. on the contrary, is very interesting. It models the propertyChainAxiom, which is a brand new construct introduced in OWL2. It is used to model a set of property which are applicable in a chain. Using transitivity, one can derive that $P(U_1,U_{n+1})$ holds if the a chain of properties between them exist.
One can also model a set of Disjoint properties (prp-pdw, prp-adp), as well as

*C*omplex Keys. This is, as propertyChainAxiom, a new feature of OWL2. The axiom prp-key, in fact, lets the user express a key formed by a set of properties. This was not possible in the original OWL language.
Some of these new constructs exploit the power of lists, which make possible to express sets of elements in a compact way. Of course this is not directly expressible in OWL2 (xml syntax), but it is natural and easy in logic programming, making the notation easy to use and straightforward.

Axiom Code	ASP axiom
cls-thing	$class(thing)$
cls-nothing1	$class(nothing)$
cls-nothing2	$\leftarrow nothing(X)$
cls-int1	$C(Y) \leftarrow intersectionOf(C, [C_1, \ldots, C_n]), C_1(Y), \ldots, C_n(Y)$
cls-int2	$C_1(Y) \leftarrow intersectionOf(C, [C_1, \ldots, C_n]), C(Y)$ $\ldots$ $C_n(Y) \leftarrow intersectionOf(C, [C_1, \ldots, C_n]), C(Y)$
cls-uni	$C(Y) \leftarrow unionOf(C, [C_1, \ldots, C_n]), C_i(Y), \forall\ 1 \leq i \leq n$
cls-com	$\leftarrow complementOf(C_1, C_2), C_1(X), C_2(X)$
cls-svf1	$X(U) \leftarrow someValuesFrom(X, Y), onProperty(X, P),$ $P(U, V), Y(V)$
cls-svf2	$X(U) \leftarrow someValuesFrom(X, thing), onProperty(X, P),$ $P(U, V)$
cls-avf	$Y(V) \leftarrow allValuesFrom(X, Y), onProperty(X, P),$ $P(U, V), X(U)$
cls-hv1	$P(U, Y) \leftarrow hasValue(X, Y), onProperty(X, P), X(U)$
cls-hv2	$X(U) \leftarrow hasValue(X, Y), onProperty(X, P), P(U, V)$
cls-maxc1	$\leftarrow maxCardinality(X, 0),$ $onProperty(X, P), X(U), P(U, V)$
cls-maxc2	$sameAs(Y_1, Y_2) \leftarrow maxCardinality(X, 1),$ $onProperty(X, P), X(U), P(U, Y_1), P(U, Y_2)$
cls-maxqc1	$\leftarrow maxQualCardinality(X, 0), onProperty(X, P),$ $onClass(X, C), X(U), P(U, Y), C(Y)$
cls-maxqc2	$\leftarrow maxQualCardinality(X, 0), onProperty(X, P),$ $onClass(X, thing), X(U), P(U, Y)$
cls-maxqc3	$sameAs(Y_1, Y_2) \leftarrow maxQualCardinality(X, 1),$ $onProperty(X, P), onClass(X, C), X(U),$ $P(U, Y_1), C(Y_1), P(U, Y_2), C(Y_2)$
cls-maxqc4	$sameAs(Y_1, Y_2) \leftarrow maxQualCardinality(X, 1),$ $onProperty(X, P), onClass(X, thing), X(U), P(U, Y_1), P(U, Y_2)$
cls-oo	$C(Y_1) \leftarrow oneOf(C, [Y_1, \ldots, Y_n])$ $\ldots$ $C(Y_n) \leftarrow oneOf(C, [Y_1, \ldots, Y_n])$

Table 6.6
Semantics of Classes in OWL2RL given in terms of corresponding ASP rules.

The table above specifies the axioms about classes. The interesting axioms here are the ones regarding intersection: cls-int1 and cls-int2. In the first one,

an object $C(Y)$ is derived if it is defined as the intersection of n other classes, and $C_i(Y)$ exists. In the second one, the opposite condition is expressed: $C_i(Y)$ is derived if $C(Y$ exists.

Axiom Code	ASP axiom
cax-sco	$C_2(X) \leftarrow subclassOf(C_1, C_2), C_1(X)$
cax-eqc1	$C_2(X) \leftarrow equivalentClass(C_1, C_2), C_1(X)$
cax-eqc2	$C_1(X) \leftarrow equivalentClass(C_1, C_2]), C_2(X)$
cax-dw	$\leftarrow disjointWith(C_1, C_2), C_1(X), C_2(X)$
cax-adc	$\leftarrow allDisjointClasses([C_1, C_n]), C_i(Z), C_j(Z)\ \forall\ 1 \leq i < j \leq n$

Table 6.7
Semantics of Class Axioms in OWL2RL given in terms of corresponding ASP rules.

Axioms from the table above (semantics of class axioms) are rather straightforward, yet some of them appear interesting. In particular, the last two deal with the concept of *d*isjointness. cax-dw states that two disjoint classes cannot have individuals in common; cax-adc generalizes that to a list of pairwise disjoint classes.

For datatypes, the following holds:

- dt-type1: $rdfs : Datatype(dt)$ for each datatype dt supported by OWL2 RL.
- dt-type2: $dt(lt)$ for each literal lt and each datatype dt which are supported by OWL2 RL. Moreover, the data value of lt must belong to the value space of dt.
- dt-eq: $sameAs(lt_1, lt_2)$ for all literals t_1 and t_2 with the same data value.
- dt-diff: $differentFrom(lt_1, lt_2)$ for all literal t_1 and t_2 with different data values.
- dt-not-type: $\leftarrow dt(lt)$ for each literal lt and each datatype dt supported by OWL2 RL, for which lt does not belong to the value space of dt

Axiom Code	ASP axiom
scm-cls	$subClassOf(C,C) \leftarrow owl:Class(C)$ $equivalentClass(C,C) \leftarrow owl:Class(C)$ $subClassOf(C,owl:Thing) \leftarrow owl:Class(C)$ $subClassOf(owl:Nothing,C) \leftarrow owl:Class(C)$
scm-sco	$subclassOf(C_1,C_3) \leftarrow subclassOf(C_1,C_2), subclassOf(C_2,C_3)$
scm-eqc1	$subclassOf(C_1,C_2) \leftarrow equivalentClass(C_1,C_2)$ $subclassOf(C_2,C_1) \leftarrow equivalentClass(C_1,C_2)$
scm-eqc2	$equivalentClass(C_1,C_2) \leftarrow$ $subclassOf(C_1,C_2), subclassOf(C_2,C_1)$
scm-op	$subPropertyOf(P,P) \leftarrow objectProperty(P)$ $equivalentPropertyOf(P,P) \leftarrow objectProperty(P)$
scm-dp	$subPropertyOf(P,P) \leftarrow datatypeProperty(P)$ $equivalentPropertyOf(P,P) \leftarrow datatypeProperty(P)$
scm-eqp1	$subPropertyOf(P_1,P_2) \leftarrow equivalentProperty(P_1,P_2)$ $subPropertyOf(P_2,P_1) \leftarrow equivalentProperty(P_1,P_2)$
scm-eqp2	$equivalentProperty(P_1,P_2) \leftarrow subPropertyOf(P_1,P_2),$ $subPropertyOf(P_2,P_1)$
scm-dom1	$domain(P,C_2) \leftarrow domain(P,C_1), subclassOf(C_1,C_2)$
scm-dom2	$domain(P_1,C) \leftarrow domain(P_2,C), subpropertyOf(P_1,P_2)$
scm-rqn1	$range(P,C_2) \leftarrow range(P,C_1), subclassOf(C_1,C_2)$
scm-rqn2	$range(P_1,C) \leftarrow range(P_2,C), subPropertyOf(P_1,P_2)$
scm-hv	$subclassOf(C_1,C_2) \leftarrow hasValue(C_1,I), onProperty(C_1,P_1),$ $hasValue(C_2,I), onProperty(C_2,P_2) subPropertyOf(P_1,P_2)$
scm-svf1	$subclassOf(C_1,C_2) \leftarrow someValuesFrom(C_1,Y_1),$ $onProperty(C_1,P) someValuesFrom(C_2,P),$ $onProperty(C_2,P_2), subclassOf(Y_1,Y_2)$
scm-svf2	$subclassOf(C_1,C_2) \leftarrow someValuesFrom(C_1,Y),$ $onProperty(C_1,P_1), someValuesFrom(C_2,Y),$ $onProperty(C_2,P_2), subpropertyOf(P_1,P_2)$
scm-avf1	$subclassOf(C_1,C_2) \leftarrow allValuesFrom(C_1,Y_1),$ $onProperty(C_1,P), allValuesFrom(C_2,Y_2),$ $onProperty(C_2,P), subclassOf(Y_1,Y_2)$
scm-avf2	$subclassOf(C_2,C_1) \leftarrow allValuesFrom(C_1,Y),$ $onProperty(C_1,P_1), allValuesFrom(C_2,Y),$ $onProperty(C_2,P_2), subpropertyOf(P_1,P_2)$
scm-int	$subclassOf(C,C_1) \leftarrow intersectionOf(C.[C_1,\ldots,C_n])$ $\ldots$ $subclassOf(C,C_n) \leftarrow intersectionOf(C.[C_1,\ldots,C_n])$
scm-uni	$subclassOf(C_1,C) \leftarrow unionOf(C.[C_1,\ldots,C_n])$ $\ldots$ $subclassOf(C_n,C) \leftarrow unionOf(C.[C_1,\ldots,C_n])$

Table 6.8
Semantics of Schema Vocabulary in OWL2RL given in terms of corresponding ASP rules.

In the table regarding the semantics of Schema Vocabulary, all the constructors are used, in order to deliver the features they offer. For example, it is stated that *d*omain respects the subclassOf-subPropertyOf paradigm (sqc-dom1 and sqc-dom2) as well as range (scm-rqn1 and scm-rqn2). It is stated, moreover, the behaviour of subclass w.r.t. to **intersectionOf** and **unionOf** (scm-int and scm-uni); **equivalentClass** is considered, of course, as a special case of **subclassOf** (scm-eqc1 and scm-eqc2). **subclassOf** is put in relationship with **allValuesFrom** and **someValuesFrom** (scm-svf1, scm-svf2, scm-avf1, scm-avf2).

6.4 Remarks

The translation of OWL2 fragments permits, as said, to use the power of logic programming to deal with ontologies expressed in such languages. The *t*rickiest fragment is undoubtfully OWL2 RL, which was designed to be the most expressive one. Many translation rules are necessary, in fact, to implement all the features this language offer.
In this way, by circumventing the well known semantic problems we have dealt with in previous sections, it is possible to use the expressive power of such language combined with the efficiency of logic programs reasoning engines. It is interesting to take note that the translation is very general and, with some syntactic changes, other systems than DLV may be used for evaluation, given they support similar features.

Part III

Implementing a OWL2 reasoner with RIF and DLVHEX

7

Implementation of a OWL2RL Reasoner with RIF and DLVHEX

7.1 Introduction

The W3C is currently developing *RIF* (*Rule Interchange Format*) [38], a universal layer designed for exchanging rules between different and possibly heterogeneous systems over the Semantic Web. It is focused on the exchange more than on the development of a single system to fit all needs of all the already available rule systems, because it appears clear that a system which fits all needs is very difficult, if not impossible to build, due to the large syntactic and semantic differences between different systems or even in different modules of the same system. The RIF working group divided the language into *dialects* which are meant to be used in different situations, while maintaining the largest subset of rules in common. They are called *RIF profiles*: *Core*, *BLD* and *PRD*. While Core is formed by the base constructs of the language, BLD (Basic Logic Dialect) is focused on logic, while PRD (Production Rules Dialect) is based on the concept of production rules. Among other features, by treating F-Logic like frames equivalently to RDF triples, particularly the RIF Core and RIF BLD fragments, promise a standard format for publishing and exchanging rules on top of RDF.

Likewise, ontologies in *OWL2RL*[49], a rule-based sublanguage of the Web ontology language *OWL2* [52], enables the support of inference over ontologies directly in rule-based system. This is achieved by giving a partial axiomatisation of the RDF OWL2 semantics in terms of first-order implications that can be encoded as rules.

At the moment few implementations of OWL2RL and RIF-Core exist since both languages are quite new. Moreover, we are not aware of any implementations –

as of yet – that implement the combinations of RIF and OWL as standardized [11].

To fill this gap we propose and implemented a reduction of those languages to *DLVHEX* [24], a powerful disjunctive logic reasoner based on the Answer Set Programming paradigm. DLVHEX has it roots in *DLV*, a disjunctive Datalog system, but adds several features to the base language. The most interesting of them is the possibility to use natively higher order atoms and external atoms, which are added to the core language by means of a plug-in architecture. Through external atoms it is possible to inject procedural code in the otherwise purely declarative semantics of the language. This concept is very similar to libraries for other reasoners which enable interaction with external data sources, such as, e.g., the integration of RDF support in SWI-Prolog [69]. There already exist a rich collection of DLVHEX plugins for Semantic Web languages, such as SPARQL [57], RDF and OWL DL [25]. Our new plugin for RIF-Core and OWL2RL not only expands the interoperability of DLVHEX with these two new standards, but also enables the combination of both with the other data models and extensions, already accessible by plugins, for an evaluation, experiments and new applications by combining these languages with the expressive features of Answer Set Programming [6, 22].

Our plugin allows DLVHEX to load and process RIF rule sets as well as OWL2RL ontologies. These are transformed to DLVHEX programs in a two-step translation: we first rewrite from OWL2RL to RIF-Core, and then perform a translation into DLVHEX. To this end there exist two different OWL2RL-to-RIF reduction methods, though, a static RIF rule set [59, Appendix 8.1] or dynamic a translation function from OWL2RL ontologies to RIF documents which yields RIF rules specifically to the input ontology [59, Appendix 8.2]. In comparison, the former approach bears some limitations in relation to interoperability with other RIF rule sets, and the combination of RIF with OWL ontologies as specified in [11] is rather based on the latter. Despite these restrictions, our current version of the OWL2RL reasoner transforms OWL2RL ontologies into RIF rules by the static rule set for the sake of a rapid first implementation. We will explain the limitations of this approach when doing it naively, and approximate the full dynamic combination of [11] by some extensions of the naive first translation.

In the following we give a description of our system, its current development status as well as an accompanying example in Section 7.2 and 7.2, and conclude with a report on our future plans in Section 7.4.

7.2 System Description

Our plugin consists of three parts: the OWL2RL to RIF-Core translation following [59], a RIF-Core to DLVHEX translator component, and the DLVHEX reasoner. In sequel we will provide more details to these components while we describe the system's workflow partitioned into its three essential stages:
Phase I - Translation from OWL2RL to RIF-Core An OWL2RL ontology, given in RDF/XML, as input is forwarded to the OW2RL to RIF-Core translator which translates RDF triples of the input ontology to RIF frames and merges them with the static rule set from [59] to a RIF-Core document. The application of the static rule set to the RIF frames gained from the input will be performed during the evaluation of this RIF document later on.

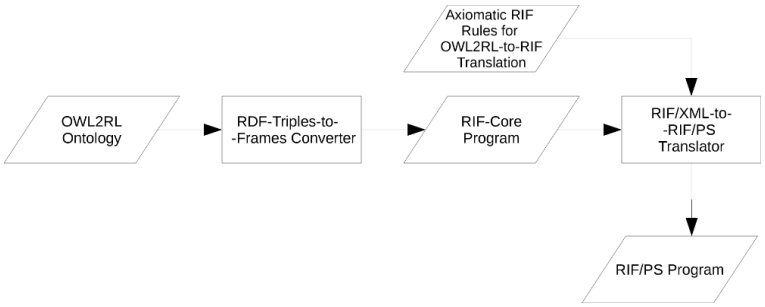

Fig. 7.1: Translation OWL2RL to RIF-Core

Phase II - Reduction of RIF-Core to DLVHEX The previously obtained RIF-Core document is preliminary reduced to a DLVHEX program. For that, the document is first parsed into an abstract syntax tree that is translated into a HEX program by a tree walking algorithm which gradually generates, adherent to a predefined set of translation rules, the corresponding HEX expressions from the visited tree nodes. This transformation includes reduction of features from RIF not directly expressible in our system to the processable input language of DLVHEX, e.g. Lloyd-Topor [45] transformation of rule bodies with disjunction, static type checking, or un-nesting of external predicates, i.e.

built-ins. Eventually, the generated program is forwarded to DLVHEX which returns a collection of answer sets.

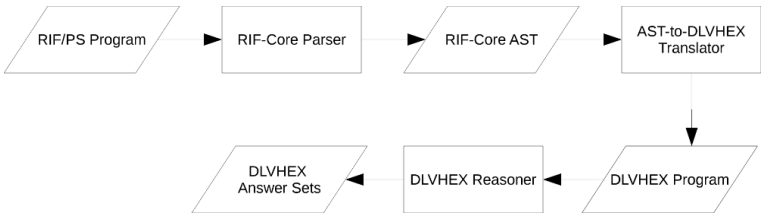

Fig. 7.2: Reduction of RIF-Core to DLVHEX

Phase III - Answer Construction from DLVHEX to OWL2RL Eventually, the answer sets, which are basically sets of ground facts, are simply transformed into a set of RIF ground atomic formulas.

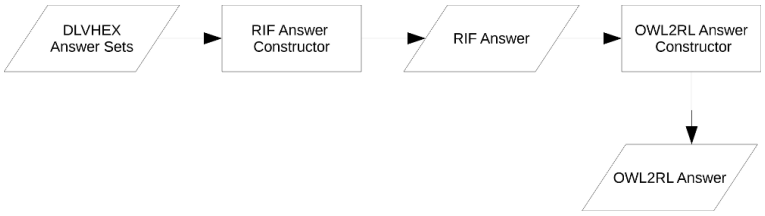

Fig. 7.3: Answer Construction from DVLHEX to OWL2RL

Example – RIF to DLVHEX

The OW2RL to RIF-Core translation, executed in Phase I is straightforward. We give here only a small example for the RIF-Core to DLVHEX translation, occurring in Phase II. We apply it here to a test case from the RIF development group, `http://www.w3.org/2005/rules/wiki/Factorial_Forward_Chaining`:

```
Document(
Prefix(pred <http://www.w3.org/2007/rif-builtin-predicate#>)
Prefix(func <http://www.w3.org/2007/rif-builtin-function#>)
Prefix(ex   <http://example.org/example#>)
Group
  (
    ex:factorial(0 1)

    Forall ?N ?F? ?N1 ?F1 (
```

```
        ex:factorial(?N ?F) :-
            And(External(pred:numeric-greater-than-or-equal(?N1 0))
              ?N = External(func:numeric-add(?N1 1))
                   ex:factorial(?N1 ?F1)
                   ?F = External(func:numeric-multiply(?N ?F1)) )
)  )  )
```

This document describes the computation of the factorial for a positive integer n. Our DLVHEX plugin rewrites the above RIF document into the following two DLVHEX rules:

```
"ex:factorial"("0", "1") :-  .
"ex:factorial"(VAR_N, VAR_F) :- &pred_numeric_geq[VAR_N1, "0"](),
                   equal(VAR_N, VAR_extOutput_1),
                   &func_numeric_add[VAR_N1, "1"](VAR_extOutput_1),
                   "ex:factorial"(VAR_N1, VAR_F1),
                   equal(VAR_F,VAR_extOutput_2),
                   &func_numeric_multiply[VAR_N,VAR_F1](VAR_extOutput_2) .
```

The translation generates two rules, a fact and a *proper* rule, corresponding to the two input RIF rules. The universal quantifier of the second RIF rule is omitted here since DLVHEX rules are per se universal. RIF constants (CURIes, typed literals, quoted unicode strings, etc.), such as `ex:factorial` or `1`, are embraced by double quotes. Prefix names in curies will generally be expanded, but for better readability we didn't resolve them here. RIF built-in predicates and functions, such as `pred:numeric-greaterthan-or-equal` and `func:numeric-add`, are rewritten to an corresponding external DLVHEX atoms[1]. So far we support all RIF built-ins which may appear in a RIF document yielded by the OWL2RIF to RIF-Core translation. Beyond that, we also support all numeric predicates and functions implementable via calls to an XPath/XQuery Functions&Operators library.
Besides, the lack of higher-order atoms in the resulting HEX program is no coincidence. In fact, those are not needed for a pure RIF-Core implementation. Our planned support for RIF-BLD as well as future RIF extensions similar to [25] will potentially demand higher-order features though.

Handling RIF-OWL2RL Combinations

The choice of a translation via the static rule set, applied in Phase I, seemed more convenient to implement at first view. since it supports a fast imple-

[1] Actually, for this particular example, we could have also exploited the built-in predicates of DLVHEX, which supports natively simple arithmetic functions such as sum, multiply and comparisons between variables. For the sake of the example, though, we decided to show how the systems can handle such external predicates and functions, in a simple way

mentation. However, several limitations arise when translating OWL2RL into RIF via the static rules. Firstly, this method is rather inefficient compared to Reynolds's dynamic, pattern based approach [59, Appendix8.2], which creates more efficient RIF rules containing fewer free variables thus smaller grounding. Further and more problematic, the static rules as such are not suitable for RIF-OWL2RL combinations [11], i.e, a blend of OWL2RL rules with arbitrary RIF-Core rules. As pointed out in [11] the static rules create problems w.r.t. equality if applied to a RIF-OWL2RL combination, even if the RIF component is of RIF-Core. The reason lies in the possible introduction of equality through OWL2RL (via `[Object|Data]MaxCardinality` and `{Universe}FunctionalObjectProperty`) that can also affect the predicates existing in the RIF-Core component. In RIF-Core equality is only allowed in rule bodies and, thus, implications of equalities are not natively expressible. Likewise, our base system, DLVHEX, does not support equality natively, so we represent equality (which may only appear in rule bodies in RIF-Core) using `owl:sameAs`. This works out perfectly for the equality resembled by `owl:sameAs` on the level of RDF triples in the OWL2RL component [49, rules `eq-ref, eq-sym, eq-trans, eq-rep-s, eq-rep-p, eq-rep-o`], by axiomatisation in the OWL2RL rule set, but it is not comprehensively applicable in an analogous way to terms in RIF-Core, since arbitrary predicates or deeply nested external functions might occur in RIF rule sets which are unaffected by this axiomatisation.

Since we use the static rule set for the OWL2RL to RIF translation, at least for the time being, we developed a approximative rewriting for RIF rule sets for RIF-OWL2RL combination that allows us to catch these effects of equality. For a given RIF-OWL2RL combination $< R, G >$, where R is a RIF rule set and G is an RDF Graph, potentially encoding an OWL2RL ontology, our algorithm runs through the following steps and outputs a rewritten RIF-Core program S:

1. Initialise S with R. Flatten all nestings of external predicates and functions in S by recursive substitution of nested terms with variables. For that, we need to express various equalities between arbitrary function terms. However, owl:sameAs is only applicable to express equality between simple terms. Thus, we need to introduce a new equality symbol '$\doteq$' which expresses equality between arbitrary terms. Since the value of each func-

tion term, by definition, belongs to an XML datatype we can think of $\doteq$ as equality as evaluated by XPath.[2]

2. Add the static RIF-Core rule set of Reynolds to S.
3. Add G in form of frame facts to S.
4. For any constant c that appears in R but not in G add the fact `c[owl:sameAs->c] .` to S.
5. For each rule of R in S rewrite any occurring atom $p(t_1, ..., t_n)$ where p is a constant and t_i is a simple RIF term $(1 \leq i \leq n)$ to an atom $p(X_1, ..., X_n)$ where $X_i = t_i$ if t_i is a variable, else (i.e., t_i is a constant) X_i is a fresh variable.
6. Apply Lloyd-Topr rewriting for non-conjunctive rule bodies in S.
7. Optimisation by removing unnecessary `owl:sameAs` and $\doteq$ statements from the rule bodies in S.

Let us illustrate the effects of this algorithm by an example. Say R[3] contains

```
p(?x) :- Or( q(?x) r(?x,b) ) .
r(c(2 * 2 + 2)).
q(a).
q(d) :- s( 1.3 + 0.7 ).
s(1+1).
```

and $G = \{(\texttt{a}, \texttt{owl:sameAs}, \texttt{b})\}$. Then we get the following intermediate results for S:

After step 1:

```
p(?x) :- Or (q(?x)  r(?x,b) ) .
r(c,?Y1) :- And( (?Y2 ≐ 2 * 2) (?Y1 ≐ ?Y2 + 2)   ).
q(a).
q(d) :- And( s( ?Y1 ) (?Y1 ≐ 1.3 + 0.7) ).
s(?Y1) :- (?Y1 ≐ 1 + 1).
```

After step 2: $S := S \cup$ "Static Rule Set"

After step 3: $S := S \cup \{\texttt{a[owl:sameAs->b]}\}$

After step 4: $S := S \cup \{\texttt{c[owl:sameAs->c]}, \texttt{2[owl:sameAs->2]}\}$

[2] In fact, on the stage of DLVHEX '$\doteq$' is evaluated by an external equality predicate implemented through XPath equality checks.

[3] Please note, that R deviates from the formal RIF syntax as we use here '+' and '$*$' for the built-in functions `func:numeric-add` and `func:numeric-multiply` in infix-notation for better readability

After step 5:

```
p(?x) :- And ( Or (q(?x)  r(?x,?X1) ) ?X1[owl:sameAs->b] ) .
r(?X1,?Y1) :- And( ?X1[owl:samAs->c] (?Y2 ≐ 2 * 2) (?Y1 ≐ ?Y2 + 2) ).
q(a).
q(?X1) :- And( ?X1[owl:samAs->d]  s( ?Y1 ) (?Y1 ≐ 1.3 + 0.7) ).
s(?Y1) :- (?Y1 ≐ 1 + 1).
```

After step 7:

```
p(?x) :- q(?x) .
p(?x) :- And ( r(?x,?X1) ?X1[owl:sameAs->b] ) .
r(?X1,?Y1) :- And( ?X1[owl:samAs->c] (?Y2 ≐ 2 * 2) (?Y1 ≐ ?Y2 + 2) ).
q(a).
q(?X1) :- And( ?X1[owl:samAs->d]  s( ?Y1 ) (?Y1 ≐ 1.3 + 0.7) ).
s(?Y1) :- (?Y1 ≐ 1 + 1).
```

Our translation is realized as a plugin[4] to the DLVHEX system[5]. Furthermore, RIF-Core contains many built-ins in form of external predicates and functions. These external functions are computed by use of a standard XML Library that implements most of the common XPath/XQuery Functions& Operators [48]. At present, we support a subset of those, as we focused our attention on the built-ins which are mandatory for the reduction of OWL2RL reasoning to DLVHEX via RIF.

- The reduction of OWL 2 RL to RIF is facilitated through the static rule set given in [59]. In this approach RDF-triples of an OWL 2 RL ontology are translated to RIF frames. Afterwards there semantics are represented by a static set of RIF-Core rules. In our implemented, we add this static rules as axioms to each RIF program together with the frames yielded from the initial OWL2RL ontology.
- Limitations of embedding OWL 2 RL into RIF via the static rule set from Dave Reynolds
  - According to Dave Reynolds this is approach is correct but inefficient; will be replaced with template or dynamic method in the future
  - According to the author this rule set is correct, but creates problems in terms of combination with existing RIF-Core rule sets: As Jos points out the static rules create problems w.r.t. equality if applied to a RIF-OWL2RL combination, even if the RIF component is of RIF-Core. The

[4] For the source code and installation/usage instructions, please refer to `http://sourceforge.net/projects/dlvhex-semweb/` as well as `http://dlvhex-semweb.svn.sourceforge.net/viewvc/dlvhex-semweb/dlvhex-rifplugin/`.

[5] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

reason lies in the potential introduction of equality through the OWL2RL rules for predicates in existing in the RIF-part, specifically:

- Equality in OWL2RL is represented by ObjectMaxCardinality and DataMaxCardinality restrictions, as well as FunctionalObjectProperty UniverseFunctionalObjectProperty, SameIndividual, and HasKey axiom.
- Equality in rule heads is not a part of RIF-Core, the static rules simulate equality by replication. This works fine for the equality in terms of triples. But replication is not analogously a comprehensive solution for RIF terms in Core, since arbitrary deeply nested external functions are allowed there and equality replication would yield infinity replica rules.
- Conclusion:
  - Thus the embedding of OWl-2-RL RIF-Core combos in RIF-Core can yield problems due to the lack of equality in RIF-Core, thus, a correct representation of equality in a RIF-Core - OWL2RL combination is not possible.
  - A reduction of OWL-2RL rule set to RIF-Core without the addition of any external rules is possible, but evidently prevents the possibility to use these reduced rule sets with RIF rules form other sources. This breaks the essential notion of RIF in terms of extensible rule sets.

7.3 Implementation Notes

We want to focus now on some details regarding implementation of the prototype.

The basis of our reasoner is the dlvhex system, which is a logic engine we described previously. At the time of working on our reasoner, the dlvhex system was already very complete and full of capabilities. which are available through a full-fledged system of plug-ins. We decided, in fact, to realize anything as a plug-in for dlvhex.

The point-of-view is important in this case: in fact if you consider dlvhex the main feature, you may see whole reasoner as a plug-in for it: if, on the contrary, one watches it from the OWL2 RL scenario, dlvhex is nothing else than a logic

engine, which is likely to be substituted with another one if the case.

We will focus here especially on two of the modules of the architecture.

The XML translator

Starting from the ontology, which is expressed in a XML-like syntax, the first step consists of a translation in the presentation syntax. This is necessary for two reasons:

- Human Readability: XML syntax is not human readable at all.
- Interoperability with other systems: some of them may use Presentation Syntax as well. In this case an additional layer is necessary.

The implementation of the translator has been realized using the C++ programming language. Moreover, we have exploited the power of a very famous GNU C library: libxml2. It was originally designed for the Gnome Desktop Environment, for handling xml documents.
The parser is top-down: parsing is started from the higher level symbol, down to the terminal symbols. Next we report an example of code used for parsing RIF documents.

Example 7.1. Function for translating an And of Formulas

```
xmlChar* RifParser::processAndFormulas(xmlTextReader* reader)
{
  bool hasAnnotation = false;
  xmlChar* annotation = xmlCharStrdup("");
  if(DBG)
    fout << "processing and of formulas" << endl;
  int ret;
  xmlChar* formulas = xmlCharStrdup("");
  do
  {
    ret = xmlTextReaderRead(reader);
    if(xmlStrEqual(xmlTextReaderConstName(reader),
    xmlCharStrdup("id"))==1
    && hasAnnotation == false)
```

```
    {
      hasAnnotation = true;
      annotation = processAnnotation(reader);
    }
    if( xmlStrEqual(xmlTextReaderConstName(reader),
xmlCharStrdup("formula"))==1)
    {
      formulas = xmlStrcat(formulas, processFormula(reader));
      formulas = xmlStrcat(formulas, xmlCharStrdup("\n"));
    }
  }
  while(xmlStrEqual(xmlTextReaderConstName(reader),
  xmlCharStrdup("And"))==0);

  xmlChar* And = xmlCharStrdup("And(");
  And = xmlStrcat(And, formulas);
  And = xmlStrcat(And, xmlCharStrdup(")\n"));
  if(hasAnnotation)
    return xmlStrcat(annotation, And);
  return And;
}
```

The code is used for translating a particular construct, i.e. the And of Formulas. We recall here a part of the RIF grammar:

```
FORMULA         ::= IRIMETA? 'And' '(' FORMULA* ')' |
                    IRIMETA? 'Or' '(' FORMULA* ')' |
                    IRIMETA? 'Exists' Var+ '(' FORMULA ')' |
                    ATOMIC |
                    IRIMETA? 'External' '(' Atom ')'
```

Please look at the first line. It explains that a formula may be expressed as an "And" of Formulas, i.e. a conjunction of N formulas, $N \geq 0$. Of course such definition is recursive. This conjunction can be preceded by an annotation, indicated by the word "IRIMETA". The question mark, as usually for EBNF, means that the annotation is optional.
First, the presence of an annotation is looked for. Its presence goes together

with the keyword "id". If the annotation is found, it is stored and the computation goes on. Then we look for the sequence of formulas, identified by the keyword "formula". Such sequence is closed by the word "And" (note that, being a top-down parser, the presence of the opening "And" had triggered the calling of this function). For each formula encountered, the "processFormula" function is recursively invoked.
The workflow of the parsing is a sequence of similar functions, with the obvious syntactic differences for different constructs.

The DTB Datatypes and built-ins

For the reasoner it has been necessary to implement a greatest part of the RIF Datatypes and Built-ins. The datatypes are based on Xml-schema. Again, to implement them libxml2 was used, which provided the necessary support. Datatypes handle several aspects related to integer values, floating point, strings, dates and times.
Built-ins regard utility function over strings and lists especially. To manage and manipulate lists, in fact, it is possible to rely on functions like union, difference, intersection, concatenation, etc. In the following example, the function reverse is defined. It takes a list $l : list(e_1, \ldots, e_n)$ as an argument, and returns the list l^r defined as $list(e_n, \ldots, e_1)$. Lists in RIF may be nested, so they are flattened before any operation on them. This is possible because according to the semantics nested lists are equivalent to flat list. For example, $list(a, b, c) = list(a, list(b, c)$.

Example 7.2. This function reverses the given list and returns the result

```
string ListChecker::reverse(string & list)
{
  string toReturn("List(");
   string flatList = flatten(list);


   int i = flatList.length()-1;
   int count = 0;
```

```
   string item("");
   while(i>=0)
   {
      count++;
      if(flatList.at(i) == ' ')
      {
        item = flatList.substr(i+1,i+count);
        toReturn += item + " ";
        item = " ";
        count = 0;

      }
      i--;
   }

   toReturn += ")";
   return toReturn;

}
```

7.4 Remarks and Future Work

We presented a DLVHEX plugin for OWL2RL and RIF-Core reasoning. The former is based on a 2-step reduction to DLVHEX via RIF-Core. This is, to our knowledge, the first attempt to implement RIF-OWL combinations a la [11], At our current stage of development we facilitate the translation to RIF by the static rule set of [59] which, as we have explained earlier, imposes restrictions on reasoning in combination with other RIF-Core documents. For the future we, therefore, will consider to modify the implementation of the first phase, switching from the static rule set to the dynamic rewriting by [59, Appendix8.2] similarly used in [11] which is based on RIF-BLD. Consequently, we will also try to extend the RIF-Core to DLVHEX translation in Phase II to more features of RIF-BLD. Moreover we plan to implement the remaining RIF built-ins to have a more complete translation from RIF-BLD to DLVHEX.

8

Conclusions and Future Work

In this thesis we have shown various techniques for the translation of logic formalisms in the Semantic Web.
First, we have introduced the background of the research, that is to say the tools, techniques and theories we have used in order to broaden our vision of the problem. Then, we have given a thorough insight of what the research activity was useful to find out.
In the first part, we discussed our approach to the problem of integration of heterogeneous formalisms in the Semantic Web. Such problem is actually very complex, and we proposed our solution, specifically oriented to query answering. In fact, since the query answering task in the Semantic Web Language is often not efficient, we have developed a technique which permits the translation of an ontology in a logic program, which can in turn be used as input to a logic engine, usually faster in such query tasks.
We have chosen a description logic fragment to be translated, $\mathcal{ELHI}$, and explained in detail how the translation is performed, and why it is sound and complete. We have shown, moreover, that our ideas have been realized with a prototype software, which makes use of reusable logic modules to translate in a semantic way the input ontology. In order to assess the validity of this approach, we have performed some tests, and shown the results compared to a direct competitor, as Pellet [67] is.
In the second part we have generalized the modular approach, proposing other formalisms which can be modularized and integrated with logic programming. One of them is Frame Logic, which is a formalism aimed at bringing the power of Object-Oriented Programming languages into logic. We designed a modular framework to implement all F-Logic constructs into a system called DLT, which

is a dialect of the popular logic programming system DLV.
Later on, we moved forward to the profiles of OWL2, which is the new version of the popular Ontology Web Language. This language has been divided into profiles, corresponding to different Description Logic Fragments. Again, we performed the translation using a modular approach.
Finally, we have moved our attention to a different representation formalism, although remaining in the Semantic Web scenario: RIF.
The latter is the name for a fresh standard for the exchange of Rules between heterogeneous logic systems. We have used it to build a Reasoner able to process OWL2RL ontologies. The Ontology is translated into a RIF program, which is in turn converted into a dlvhex program (dlvhex is another system based on DLV, with interesting capabilities). This implementation work has proven to be particularly interesting, as at the time of it we were not aware of any existing complete implementation of RIF, or any complete OWL2RL reasoner. Part of research work herein reported has been acknowledged by the scientific community on the following reported papers:

- Mario Alviano, Giovambattista Ianni, Marco Marano.Alessandra Martello "Versatile Semantic Modeling of Frame Logic Programs under Answer Set Semantics", ASWC 2008, Springer, pages 106-121, 2009.
- Marco Marano, Giovambattista Ianni, Francesco Ricca "A Magic set implementation for Disjunctive Logic Programming with Function Symbols", CILC 2009.
- Marco Marano, Phillip Obermeier, Axel Polleres "Translating OWL2RL to DLVHex via RIF", RR 2010, Springer, pages 244-250, 2010.
- Marco Marano, Giovambattista Ianni: "Semantic Modeling of heterogeneous Logics with Logic Programming" (Technical Report)
- Anna Bria, Giovambattista Ianni, Marco Marano, Francesco Ricca: "A pure forward-chaining approach for query answering on EL knowledge bases" (Technical Report)

Future Work

The work described in this thesis has shown good results. As scientists, though, we want to improve and complete the various techniques described previously,

For what concerns the first part, we mean to extend the described translation technique to more expressive logic fragments. $\mathcal{ELHI}$ is interesting, but it lacks some features that would be interesting to have in the future. Constructs like unrestricted universal quantification, bottom and top concepts, and others would prove very useful for more complex reasoning tasks.
Moreover, the prototype needs some polishing and additional features, in order to be used on a larger, non-academic scale.
The work on Frame Logic is also interesting. Possible developments in this direction could be the introduction of more semantics, configurable on-the-fly as usually, or a full integration with the DLV system, instead of DLT.
Finally, there seems to be a great interest around RIF in recent times. The reasoner already works, but it would be important to compare it with the other reasoners which are going to be released, to measure performances and accuracy.

References

1. Mario Alviano, Giovambattista Ianni, Marco Marano, and Alessandra Martello. Versatile semantic modeling of frame logic programs under answer set semantics. In *ASWC*, pages 106–121, 2008.
2. Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the el envelope. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 364–369. Professional Book Center, 2005.
3. Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the el envelope further. In Kendall Clark and Peter F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.
4. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
5. Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.
6. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.
7. David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.
8. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web (Berners-Lee et. al 2001). May 2001.
9. Piero A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1):75–111, 2004.
10. Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI*, pages 385–390, 2007.
11. Jos De Bruijn. RIF rdf and OWL compatibility. Proposed recommendation, W3C, October 2009. http://www.w3.org/TR/2010/PR-rif-rdf-owl-20100511/.
12. Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive logic programs with inheritance. *TPLP*, 2(3):293–321, 2002.

13. Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng.*, 12(5):845–860, 2000.
14. Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *KR*, pages 70–80, 2008.
15. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp: Theory and implementation. In *ICLP*, pages 407–424, 2008.
16. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Bottom-up evaluation of finitely recursive queries. In *CILC09*, 2009.
17. Francesco Calimeri and Giovambattista Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Commun.*, 19(3):193–206, 2006.
18. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. In *KR*, pages 260–270, 2006.
19. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Riccardo Rosati, and Guido Vetere. Dl-lite: Practical reasoning for rich dls. In *Description Logics*, 2004.
20. Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Enhancing the magic-set method for disjunctive datalog programs. In *ICLP*, pages 371–385, 2004.
21. Jos de Bruijn and Stijn Heymans. Translating ontologies from predicate-based to frame-based languages. In *RuleML*, pages 7–16, 2006.
22. Thomas Eiter, Giovambattista Ianni, Axel Polleres, and Roman Schindlauer. Answer set programming for the semantic web, June 2006. Tutorial at ESWC2006.
23. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pages 90–96, 2005.
24. Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *ESWC*, pages 273–287, 2006.
25. Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. In *KR*, 2004.
26. Thomas Eiter and Mantas Simkus. Bidirectional answer set programs with function symbols. In *IJCAI*, pages 765–771, 2009.
27. Thomas Eiter and Mantas Simkus. Fdnc: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM Trans. Comput. Log.*, 11(2), 2010.
28. Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets and their application to data integration. In *ICDT*, pages 306–320, 2005.
29. François Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Comput.*, 9(3/4):425–444, 1991.
30. Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
31. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.
32. Sergio Greco. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 15(2):368–385, 2003.

33. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, October 2005.
34. Pascal Hitzler, Rudolf Sebastian, and Markus Krötzsch. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, London, 2009.
35. Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In *NMR*, pages 233–239, 2004.
36. Hasan M. Jamil. Implementing abstract objects with inheritance in datalog$^{\text{neg}}$. In *VLDB*, pages 56–65, 1997.
37. Yevgeny Kazakov. *Saturation-Based Decision Procedures for Extensions of the Guarded Fragment*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, March 2006.
38. Michael Kifer and Harold Boley. RIF basic logic dialect. Proposed recommendation, W3C, October 2009. http://www.w3.org/TR/2010/PR-rif-bld-20100511/.
39. Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.
40. Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlim - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.
41. Markus Krötzsch, Peter F. Patel-Schneider, Sebastian Rudolph, Pascal Hitzler, and Bijan Parsia. OWL 2 web ontology language primer. Technical report, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-primer-20091027/.
42. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
43. Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Ann. Math. Artif. Intell.*, 25(3-4):369–389, 1999.
44. John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
45. John W. Lloyd and Rodney W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
46. Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with florid: A deductive object-oriented perspective. *Inf. Syst.*, 23(8):589–613, 1998.
47. Carsten Lutz, David Toman, and Frank Wolter. Conjunctive query answering in the description logic el using a relational database system. In *IJCAI*, pages 2070–2075, 2009.
48. Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 functions and operators. Recommendation, W3C, January 2007. http://www.w3.org/TR/xpath-functions/.
49. Boris Motik, Achille Fokoue, Ian Horrocks, Zhe Wu, Carsten Lutz, and Bernardo Cuenca Grau. OWL 2 web ontology language profiles. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/.
50. Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. Owl 2 web ontology language: Profiles. World Wide Web Consortium (W3C) Recommendation, 2009.

51. Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. Minimal deductive systems for rdf. In *ESWC*, pages 53–67, 2007.
52. W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-overview/.
53. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
54. Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Rewriting conjunctive queries over description logic knowledge bases. In *SDKB*, pages 199–214, 2008.
55. Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Tractable query answering and rewriting under description logic constraints. *J. Applied Logic*, 8(2):186–209, 2010.
56. Axel Polleres, Cristina Feier, and Andreas Harth. Rules with contextually scoped negation. In *ESWC*, pages 332–347, 2006.
57. Axel Polleres and Roman Schindlauer. dlvhex-sparql: A sparql-compliant query engine based on dlvhex. In *2nd Int. Workshop on Applications of Logic Programming to the Web, Semantic Web and Web Services (ALPSWS2007*, pages 332–347. Springer, 2007.
58. Teodor C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *ICLP*, pages 459–477, 1990.
59. Dave Reynolds. OWL 2 RL in RIF. W3C working draft, W3C, October 2009. http://www.w3.org/TR/2009/WD-rif-owl-rl-20091001/.
60. Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso, and Nicola Leone. Ontodlv: An asp-based system for enterprise ontologies. *J. Log. Comput.*, 19(4):643–670, 2009.
61. Francesco Ricca and Nicola Leone. Disjunctive logic programming with types and objects: The dlv$^+$ system. *J. Applied Logic*, 5(3):545–573, 2007.
62. Riccardo Rosati. Dl+log: Tight integration of description logics and disjunctive datalog. In *KR*, pages 68–78, 2006.
63. Riccardo Rosati. On conjunctive query answering in el. In *Proceedings of the 2007 International Workshop on Description Logics (DL2007)*, 2007.
64. Riccardo Rosati. On the finite controllability of conjunctive query answering in databases under open-world assumption. *Journal of Computer and System Sciences*, 2010. To appear.
65. Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.
66. Michael Sintek and Stefan Decker. Triple - a query, inference, and transformation language for the semantic web. In *International Semantic Web Conference*, pages 364–378, 2002.
67. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
68. Fang Wei, Technische Universitaet Dresden, and Computational Logic. F-logic semantics and implementation of internet metadata, 1999.
69. Jan Wielemaker, Michiel Hildebrand, and Jacco van Ossenbruggen. Prolog as the fundament for applications on the semantic web. In *ALPSWS*, 2007.

70. Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan. Implementing an inference engine for rdfs/owl constructs and user-defined rules in oracle. In *ICDE*, pages 1239–1248, 2008.
71. Guizhen Yang and Michael Kifer. Inheritance in rule-based frame systems: Semantics and inference. pages 79–135, 2006.
72. Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *CoopIS/DOA/ODBASE*, pages 671–688, 2003.